

Are you a Genius?

Official Bootcamp Edition of
The Enquestopedia



Dylan Hendrickson, Chia-Hong Chou
&
Namrata Keskar

The Enquestopedia

You've been forewarned
There is a terrifying tiger in here
A Tiger named Fangs
Beware of Fangs
Once he bites you
You will forever be his slave
Until you find
The Universal Key
Which you will
What else are you gonna do¹ in
Questopia

¹ And you thought Fortnite was addictive

Preface

This is the official Genius Bootcamp edition of the Enquestopedia, a series of 27 C++ programming quests that will take you from total noob to absolute pro at your own pace.

This book contains the spec sheets for all the quests. But of course, you have to earn the password to each quest before you actually do it. Only the first quest has a password out in the open.

Actually, that's not quite true. The password to the 10th quest (the first GREEN one) is *A platypus-bodied duck*. You only need to start at *A tiger named Fangs* if you absolutely need a refresher in basic C++

Quick links

Genius Registration: <https://genius.nonlinearmedia.org>

BLUE Level Reddit: <https://reddit.com/r/cs2a>

GREEN Level Reddit: <https://reddit.com/r/cs2b>

RED Level Reddit: <https://reddit.com/r/cs2c>

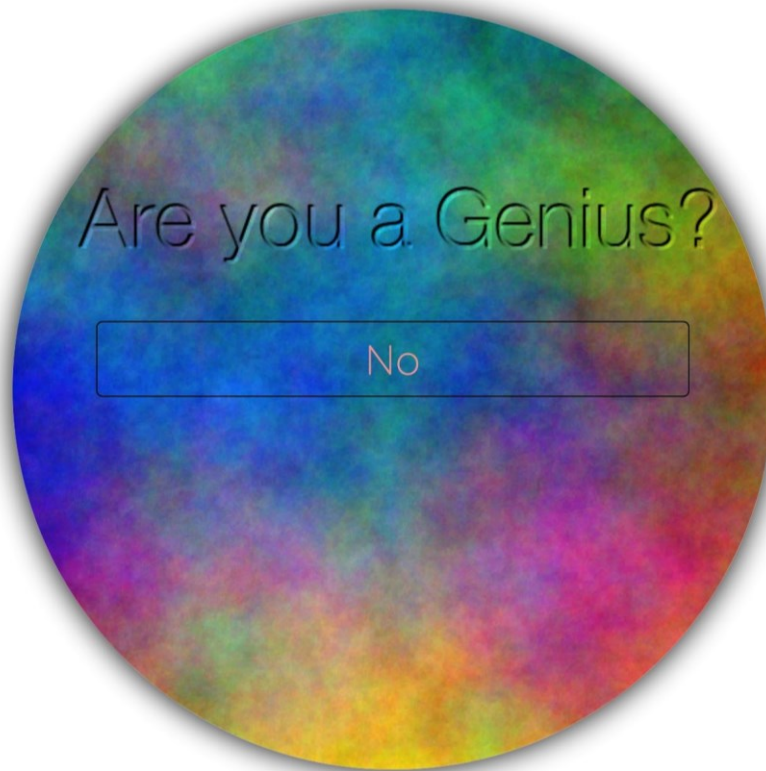
- The quests in this adventure are modeled after homework assignments for C++ students at Foothill College, Los Altos Hills, in California, USA. This bootcamp is offered to the general public, and runs at a much faster pace:
 - The same rules apply as for the students: Nobody in this bootcamp is allowed to *give answers* to anyone else. However, everyone is strongly encouraged to participate with conceptual questions and answers in our active and collaborative discussion forums
- Quest related discussions should be restricted to the official shared discussion forums for [BLUE](#), [GREEN](#), and [RED](#)
- For general Genius-related discussions and admin issues use the Genius flair in the [RED](#) sub.
- During the first few weeks of the bootcamp, there will be optional zoom classes for as many of you as can be accommodated. In these classes, you will get a chance to live-code important introductory c++ concepts. This is essential for those who have never programmed before, or are new to C++. They will help you gain momentum through the easy BLUE quests. See [our class lecture recordings](#). Our zoom lectures will be similar.

If you do this bootcamp sincerely, you will find out why you have ALWAYS been a genius.

Of course, you gotta practice lots after you know you're a genius. But that's up to you.

If you answered no at
<https://quests.nonlinearmedia.org/genius>

Nonlinear Genius



The Genii

You'll get a chance to find out why that was the wrong answer
at

The Questopic Genius Bootcamp

Hello, my name is &.

I am the Questmaster's virtual personal assistant.

I will guide you through the 27 quests you have to complete in this bootcamp.

Well, not really.

Most of the time I'll just watch you solve everything by yourself.



The Terrifying Tiger

In the first quest, as a fierce tiger called Fangs, your sabres bared as you bring the jungle to a halt with your fearsome roar, you will complete your first quest. You will write a program that must print the words "Hello world" on the screen.

In the last quest, as a humble mouse called Peony, whose soft steps will tread the well worn trails of humble mice before her, you will quietly toil to find the maximum flow of grains from anywhere to anywhere in a vast universe of hundreds of thousands of tunnels.

You are embarking on the mega quest of learning C++ through the Genius Bootcamp sequence. This is Level BLUE. It has nine quests you must complete, before moving on to GREEN.

The secret password to unlock each quest will appear after you have completed a certain number (a secret, known only to a select few) of *miniquests*. For instance, you will unlock the password for quest 2 by completing this one. You may not share these quest passwords with other bootcampers. Completion of a quest only counts if all previous quests have also been completed. Thus it doesn't help to get the secret passwords to locked quests from your friends anyway.

In most quests, the total number of trophies won will usually be indicated by some number of objects you would have collected - fruits, nuts, birds, flowers, etc. You can win these by completing *miniquests*.

Some miniquests have hidden rewards. You won't even see these until after you've won them. And the way to win the unseen rewards is to focus on micro detail in the spec and make sure you've got it right. Your fellow questers and I can give you hints. But you have to find these treasures yourself. That's where the fun is. And in giving out hints too. That's also kinda fun.

To get your trophy count recorded, you can include your Quester (or Student) ID within a comment (in the required format) in one of the files. You can then check your standing at the [/q](#) site.

Please don't publish cheat sheets! Be a good sport. Enjoy and leave it for others to enjoy just as you found it.

I have set up the rewards such that they can be won even after you have unlocked the next quest. Then you will have the option of advancing to the next quest as well as trying to eke more points out of the current one. You can even jump back and forth. When there are no more points to be won in a quest, I will usually tell you.



These quests start out small and easy, and progressively get harder and more detailed (after you befriend the serpent). In later quests you can expect to submit multiple files including header files that you will create. You will also have the opportunity to earn more rewards in the later quests.²

One other thing about these quests is that I will also be using them as instructional vehicles. Some information you will not find in the online references or your recommended reading may be present in your quest specs. Thus it is going to be important for you to carefully read the specs every time.

Your first quest - Hello world

It is **a trivial quest**. This means that you may as well copy your template code from most IDEs directly into your submission file and make minor adjustments to pass this quest. You will succeed.

Even though this is a trivial quest, it is important for you to complete it because:

1. It is meant to give you some practice in how you will submit future code.
2. You will only get the secret code to open up the next quest after you ace this one.

You must implement your source code for a new quest in one or more files. Then when you have them ready, you will submit them into the [questing site](#).

I will run tests on your code and report back with the rewards you scored in the quest. You can submit your code *as many times as you like whenever you want*, but I highly discourage many rapid blind submissions with random changes you hope to work. This will not get you far in the advanced quests, and so it's best not to get into that habit of spraying bullets in the dark.

² I think it's best to move on to the next quest as soon as you get the password for it and then keep coming back to improve earlier quests as and when you get time. In the past, many questers *got stuck* at early quests when they could have easily advanced to later, funner ones. This way you can stay in step with the rest of the bootcampers and take the odd free hour to polish earlier quests missing a trophy or two.

Let's get started

Here's what your c++ program ought to look like:

```
0 // Student ID: 12345678
1 // TODO - Replace the number above with your actual Student ID
2
3 #include <iostream>
4
5 using namespace std;
6
7 // Feel free to use the alternate signature of main() below
8 int main(int argc, const char * argv[])
9 {
10     // TODO - Your code goes here
11
12     return 0;
13 }
```

Most of the quest specs contain code like this that you can copy into your IDE or editor by typing in (not copy/paste). Then you will flesh them out by replacing the red TODO portions with your own code and testing them locally on your machine before submitting them. The number following the "Student ID:" is your Quester ID and should be replaced with your reddit username.

The program in the box above is in the form of "source code". That is, as human-readable c++. The numbers in the left margin are for our reference below. They are not part of the actual source file.

Once you type in, save, and *compile* your source code, the compiler will turn it into binary machine instructions that will no longer be easily readable by you. But it will be easily readable by the cpu which will run it to produce the programmed effect. Normally, we observe this effect (the output) in many different situations and make sure they all make sense before letting the program go wild on its own.

Your source code should be saved in a text file named `hello_world.cpp`. When compiled and run, it should print the words "Hello world" followed by a single newline character.

Note that everything should match exactly: character casing, spacing, indentation, etc. It's very easy to get it right, so you have to get it exactly correct in order to ace this quest. Read the provided modules and the recommended text, look up information on the internet, or discuss how to do this with your friends if necessary.

When your file is ready, you can submit it into the questing website (see below). If you pass the tests correctly, you will score rewards and also be given the secret password for your next quest.

Editor

Some questers find it easy to use a simple editor like TextEdit (in plain ASCII mode) or Notepad for their initial quests. But it gets unwieldy pretty soon. I recommend that you graduate to using a proper IDE like Xcode or VSCode at your earliest (ideally quest 2 or soon thereafter). It will greatly simplify your programming life.

Style

Style is a subjective thing and because it is a subjective thing and I don't want to force my style upon you. But keep in mind that if you follow a messy coding style you will very quickly find that

programming becomes a chore rather than a pastime as it should be. I recommend that you review and adopt the popular [K&R style for C++ coding](#).

Overview of the provided template code

In C++ you can assume that your code is always presented in the form of functions. Functions are blocks of code that perform a sequence of actions. Functions can invoke each other and when they are done executing their code they will return control back to their *callers*, to the block of code that invoked the just-concluded function.

A function's behavior can be controlled and modified via special variables you pass into them called parameters. When a function is invoked, the run-time system sets things up such that the called function has access to variables passed to it by its invoker. For example, a function called `my_function` might have the following *signature*:

```
int my_function(int param1, string param2, string param3);
```

This means that when this function is invoked by a caller, they will need to supply three variables to provide values for the parameters. The types of these variables can be found in the function's signature. Also the **int** keyword before `my_function` indicates that when `my_function` is done executing, it will return a value back to the function that invoked it. The data type of this return value will be **int**.

A typical invocation of `my_function` might be like:

```
int return_value = my_function(n, name, address);
```

where `n` is an integer, and `name` and `address` are string variables.

Also, you may call the function by passing *literal values* instead of variables. E.g.

```
int return_value = my_function(2, "John", "123 First Ave");
```

We'll study all this in more detail in the coming weeks. For now, it suffices to understand what functions do at the level described.

Ever wonder how your program actually starts? If all you have are a bunch of functions, which function gets called first?

It is `main()`. `main()` is a special function. It is called by the run-time system when your program is invoked. Once your `main()` has started, your code starts running and everything is up to you from then on. you can call other functions you have defined, or simply exit.

You will notice that `main()` returns a value. What is it?

It is the value used to report the status of completion of your program back to the run-time system. Again, conventionally, we return 0 to say that there were no errors. (However, consider that in C and C++, zero is synonymous with `false` and non-zero with `true`. Why does a program return `false` upon successful completion? Discuss this in the [forums](#)).

Let's now walk through the individual lines of the template code.

```
1 // Student ID: 12345678
```

This is an inline comment. These comments start with two forward slashes and continue to the end of the line. Replace the numeric student ID with your reddit handle in all submissions. It should be present in at least one of the files you submit. This is how I associate your quest score with the reddit handle you chose for this bootcamp. It's important that you don't mess up this line. Only replace the numeric portion. The rest of it should stay.

Also, the IDs in the multiple files all better be the same. I'll select a random one of them to extract your ID.

```
3 #include <iostream>
```

This is called an include-file directive. `iostream` is called a *header file*. It tells the compiler to import certain variables, types and data that have been pre-created for you and placed in that file. For example, the `cout` variable that you will use in your code is in the `iostream` header file.

```
5 using namespace std;
```

This is a using-namespace directive. Remember how the `iostream` library provides a bunch of variables for you? They are all provided within a *namespace* called `std`. You can think of it as if all those variables had the letters `std::` prefixed before their names. Thus we can avoid the problem of system variables conflicting with our variables of the same name (if we have any).

However, this makes for cumbersome typing. So the "using namespace" directive was created. It tells the compiler to look up variables by prefixing their names with the namespace qualifier if local matches to their names can't be found.

For example, suppose you have "using namespace std" in your code, and you use a certain variable, say `endl`, in your program without actually declaring such a variable.

Normally the compiler would complain about undefined variables and throw in the towel at this point. But because of the using-namespace directive, it will look for a variable called `std::endl` (that is `endl` defined within the `std` namespace) before doing so.

```
8 int main(int argc, const char *argv[])
```

This is the beginning of the definition of the `main()` function. The signature suggests that it accepts two parameters. The first is of type integer and the second is an array of pointers to characters. Now since `main()` is invoked by the operating system, you don't really have to worry too much about these parameters at this point. They are called command-line arguments and will be supplied by the OS. You don't need to use these parameters in any of our quests in BLUE. Feel free to look up information about them, or ignore them for now. FWIW, you can also declare `main` as:

```
8 int main()
```

In this alternate form, it doesn't take any parameters. The run-time system can handle either case correctly. We didn't use this form in our function.

Following `main` is a block of code enclosed in curly braces. This will contain the entire logic of your program. If you have other functions helping out your `main()`, they are only useful insofar as they are either invoked directly or indirectly (via another function that is likewise invoked by `main`) from `main`.

```
10 // TODO - Your code goes here
11
12 return 0;
```

Your code to print "Hello world" followed by a newline ought to go where the red TODO marker is.

The `return 0;` statement sends the exit status 0 back to the run-time system.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret code for this quest in the box.
3. Drag and drop your `hello_world.cpp` file into the button and press it. Make sure your filename is exactly as above and that it has your quester ID.
4. Wait for me to complete my tests and report back (usually a minute or less).



Need help?

You can find the BLUE discussion forum [here](#). Questers in the sub are a great resource for help (sometimes through not helping).

Happy Questin',

&



Jolly-good Jay

In this quest, you get to write three small and relatively simple programs. My recommendation is to keep them as simple as possible. These functions ought not be longer than about 5-10 lines of code each. While you can get over-engineered programs to pass the test suite, they won't gain you a foot in the door where cool things happen.

As usual, I will give you template code you can copy and flesh out.

Your first miniquest - Schrodinger's cat

In the file called `Draw_Cat.cpp`, complete the function with the following signature:

```
void draw_cat();
```

When invoked, this function must print the following lines on the console. You may use one or more hard-coded `"std::cout << ..."` statements to get the desired output.

```
-----  
|  /\_/\  |  
| (  o  o  ) |  
|  > ^ <  |  
-----
```

Schrodinger

The output must match exactly, character for character. Note the exact column spacings. There is no space after the last visible character on each line.

The `void` return type means that the function does not return a value back to its caller (in this case, your `main()` function).

Invoke this function from your `main()`. Thus when I run your program, it should print the cat on my console.

That's all.

Read this important note now

In all miniquests that require arithmetic calculations, you will find that there are multiple ways to calculate something.

For example, $x = (a + b)/2.0$ can be calculated as $x = a/2.0 + b/2.0$

But did you know that they are not the same in floating point arithmetic? You can't assume that $(a+b)/2$ will be exactly equal to $a/2 + b/2$. Why? Discuss it in the forums.

This has implications for your miniquests. Sometimes you will find that your results are very slightly different from mine. This means you're calculating the result a different way. You're not wrong to do so, but the challenge in the miniquest is to figure out (through intelligent trial-and-error) the exact calculation I'm using (without looking at my code or asking me) and replicate it at your end.

Avoiding the use of math library functions (except where noted as permitted) is a good start.

Your second miniquest - Limerick

Consider the following poem:

*A dozen, a gross and a score
and three times the square root of four
divided by seven
plus five times eleven
is nine squared and not a bit more.*

Suppose I don't know how much a dozen, a gross or a score is. I'm going to use a function you create to discover values for these variables such that the above poem is true.

In a file called Limerick.cpp, flesh out the function:

```
double eval_limerick(int dozen, int gross, int score);
```

It should calculate the left-hand-side (LHS) of the above poem using the supplied values for dozen, gross and score and return the result. The LHS is all the terms before the "is". Thus your function should return the quantitative result of the calculation:

$$\text{dozen} + \text{gross} + \text{score} + 3 * \text{sqrt}(4) / 7 + 5 * 11$$

But before you jump the gun, make sure to calculate it by hand and confirm that you correctly get the RHS (81). Use parentheses in the above expression to make sure you get the operand groupings that make the poem come true. You are allowed to use the math library function, `sqrt()`.

Note: Yes. I know that you can implement the function using non-intuitive values for these quantities and still pass the test. Just use common sense values. It's more fun that way.

Your third miniquest - Etox

In a file called Etox.cpp, implement the following function:

```
double etox_5_terms(double x);
```

It should return the value of e^x back to its caller. This value should be calculated as the sum of exactly the first five terms in its expansion below:

$$e^x = 1 + x + x^2/2! + x^3/3! + . . .$$

where $n!$ is the factorial of n , which is the product of all positive integers at most equal to n .

You would therefore return the result of the calculation:

$$1 + x + x^2/2! + x^3/3! + x^4/4!$$

Keep it simple and don't overcode. No need to write a function for calculating factorials. Simply use numeric literals in their place.

In your main() function, prompt the user as follows:

```
Enter a value for x:
```

There must be one space after the colon and no newline. You must accept console input on the same line as the prompt.

Now read the user's response into a variable, calculate e^x using the function you first defined, print the value on the console followed by a single newline.

How well you do in this quest depends a lot on your ability to pay attention to the small details of what is being asked. That's all. As I said, this week's quest is relatively easy. Hopefully you can solve it quickly and move on to the next one. But don't be in a hurry to face the main boss yet.

Starter code

Your Draw_Cat.cpp:

```
// Student ID: 12345678
// TODO - Replace the number above with your actual Student ID
//
// Draw_Cat.cpp
//
// When this program is run it should print the following lines on the console:
// For reference, Schrodinger on the last output line starts at col 1.
//
//      -----
//      | /\_/\ |
//      | ( o o ) |
//      | > ^ < |
//      -----
//      Schrodinger
//
#include <iostream>

using namespace std;

void draw_cat() {
    // TODO - Your code here
}

int main(int argc, const char * argv[]) {

    // TODO - Invoke your function from here.
    // I'll invoke your main()

    return 0;
}
```

Your Limerick.cpp:

```
// Student ID: 12345678
// TODO - Replace the number above with your actual Student ID
//
// Limerick.cpp
// BLUE-Quest-01
//
#include <iostream>
#include <sstream> // Need this for istringstream below

#include <cmath>    // needed for sqrt
#include <cstdlib>  // for exit()

using namespace std;

double eval_limerick(int dozen, int gross, int score) {
    // TODO - Your code here
}

// I'm using command line arguments below to let me test your program with
// various values from a batch file. You don't have to know the details for
// BLUE, but you're welcome to - Discuss in the forums any aspect of this
// program you don't understand.
```

```

int main(int argc, char **argv) {
    int dozen, gross, score;

    if (argc < 4) {
        cerr <<"Usage: limerick dozen-val gross-val score-val\n";
        exit(1);
    }
    istringstream(argv[1]) >>dozen;
    istringstream(argv[2]) >>gross;
    istringstream(argv[3]) >>score;

    // Invoke the eval_limerick function correctly and print the result
    // with a single newline at the end of the line.
    // TODO - Your code here (just invoke your function with the above
    // values for its params. Don't worry about argc, etc. for now)

    return 0;
}

```

As you can see, you don't have to make major modifications to the `main()` function for Limerick. But you do have to invoke your `eval_limerick` function using the values in your `dozen`, `gross` and `score` variables, and print the result.

Here is your `Etox.cpp` starter file:

```

// Student ID: 12345678
// TODO - Replace the number above with your actual Student ID
// Etox.cpp
// BLUE-Quest-01
//
#include <iostream>
#include <sstream>

#include <cmath> // needed for sqrt
#include <cstdlib> // for exit()

using namespace std;

double etox_5_terms(double x) {
    // TODO - Your code here
}

int main(int argc, char **argv) {
    string user_input;
    double x;

    cout <<"Enter a value for x: ";
    getline(cin, user_input);
    istringstream(user_input) >>x;

    // TODO - Your code here

    return 0;
}

```

Discuss anything you don't understand in the public discussion forums on Reddit to get all the clarifications you need.

Testing your own code

You should test your functions using your own `main()` function in which you try to call your functions in many different ways and cross-check their return value against your hand-computed results. But when you submit you must NOT change the behavior of the `main()` method provided in the starter code above.



Submission



When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box
3. Drag and drop your three cpp files (Draw_Cat.cpp, Limerick.cpp and Etox.cpp) into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).

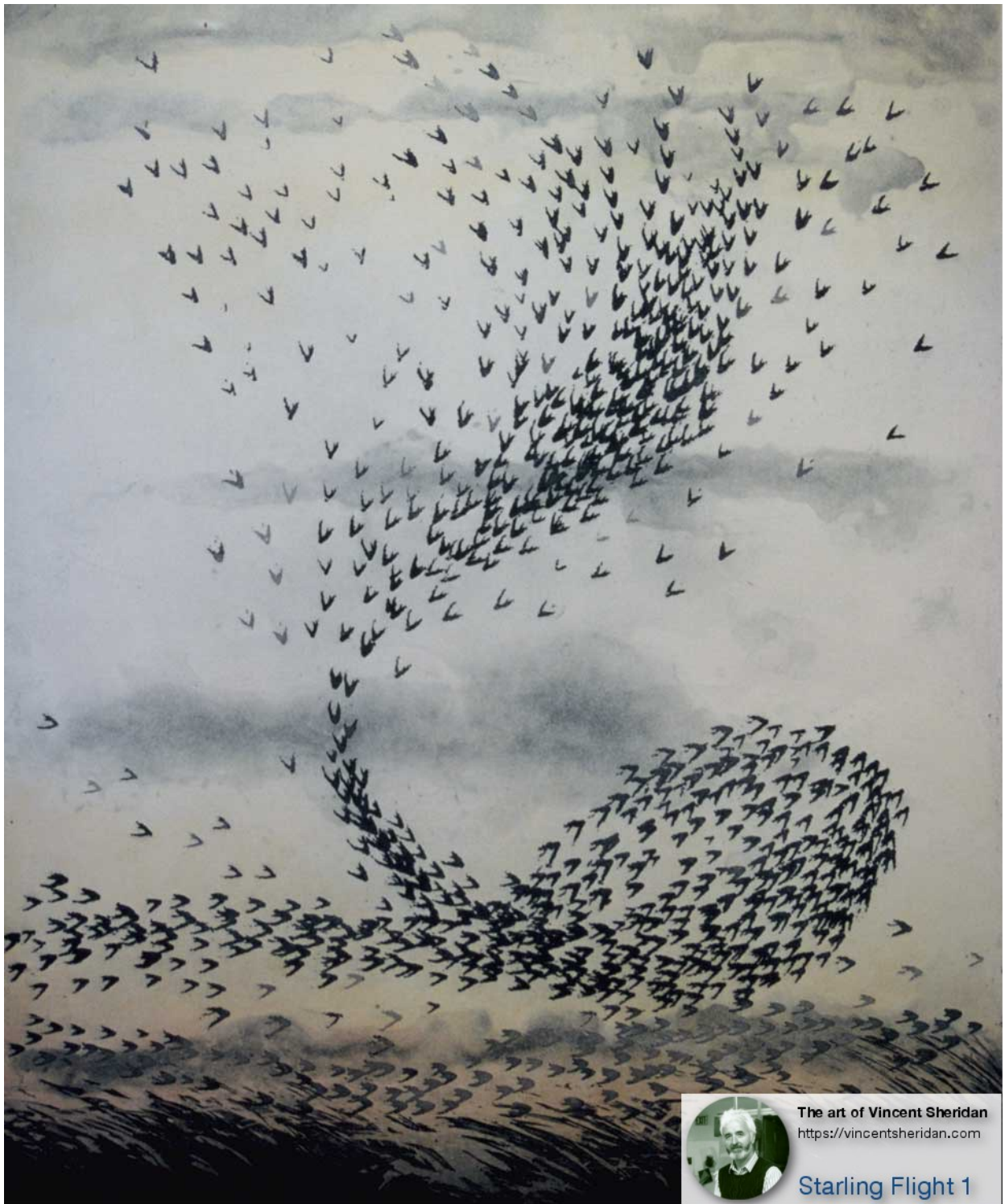
Need help?

You can find the BLUE discussion forum [here](#). Questers in the sub are a great resource for help (sometimes through not helping).

May the best coders win. That may just be all of you.

Happy Questin',

&



The art of Vincent Sheridan
<https://vincentsheridan.com>

Starling Flight 1

Starlings With a Sense

In this quest, you get to play with branching statements. You will create 6 functions, each of which does something that requires making decisions.

Each of these functions is worth varying numbers of trophies. It's up to you to find out how many trophies you can score by making the most progress you can.

As usual, I will give you template code you can copy and flesh out.

Your first miniquest - Mean of three

To get past this checkpoint you must correctly implement the following function in the template code:

```
double mean_of_3(int n1, int n2, int n3);
```

When I invoke this method, I will supply it three integer parameters. It must return their average value to me as a double.

Your second miniquest - Max of five

You must implement:

```
int max_of_5(int n1, int n2, int n3, int n4, int n5);
```

When I invoke this method, I will supply it five integer params. It must return the greatest of them to me.

Your third miniquest - Min of five

You must implement:

```
int min_of_5(int n1, int n2, int n3, int n4, int n5);
```

When I invoke this method, I will supply it five integer params. It must return the least of them to me.

Your fourth miniquest - Triangle from sides

You must implement:

```
bool sides_make_triangle(int a, int b, int c);
```

When I invoke this method, I will supply it three integer params. Return whether it is possible to form a triangle using the given numbers as side lengths. True means yes.

Your fifth miniquest - Triangle from angles

You must implement:

```
bool angles_make_triangle(int A, int B, int C);
```

When I invoke this method, I will supply it three integer params. Return whether it is possible to form a triangle using the given numbers as angles in degrees. True means yes.

Your sixth miniquest - Leap Test

You must implement:

```
bool is_a_leap_year(int year);
```

When I invoke this method, I will supply it an integer parameter. Return whether the given number is a leap year or not (according to the definition you can find in Wikipedia). True means yes.

Starter code

```
// Student ID: 12345678
// Replace the number above with your Student ID
//
// Branching-Functions.cpp
// BLUE-Quest-03
//
#include <iostream>

// This function returns the mean the three numbers passed
// in as parameters. Note that the mean may not be a round
// number. So you must use the double datatype for it.
double mean_of_3(int n1, int n2, int n3) {
    // TODO - YOUR CODE HERE
}

// This function returns the maximum of the 5 given numbers
int max_of_5(int n1, int n2, int n3, int n4, int n5) {
    // TODO - YOUR CODE HERE
}

// This function returns the minimum of the 5 given numbers
int min_of_5(int n1, int n2, int n3, int n4, int n5) {
    // TODO - YOUR CODE HERE
}

// Given three lengths, this function should return whether they can be the
// sides of some triangle. The heuristic you code should check if the
// sum of the two smallest sides is greater than or equal to the third side.
// Treat extreme cases as valid triangles. E.g. a+b == c means valid triangle.
// The challenge is to do it without using arrays
bool sides_make_triangle(int a, int b, int c) {
    // TODO - YOUR CODE HERE
}

// Given three angles as integer degrees, this function should return whether
// they can be internal angles of some triangle. Treat extreme cases as
// valid triangles. E.g. (0, 0, 180) is a valid triangle
bool angles_make_triangle(int A, int B, int C) {
    // TODO - YOUR CODE HERE
}

// Return true if the year yyyy is a leap year and false if not.
bool is_a_leap_year(int yyyy) {
    // TODO - YOUR CODE HERE
}
```


Testing your own code

You should test your functions using your own `main()` function in which you try and call your functions in many different ways and cross-check their return value against your hand-computed results. But when you submit you must NOT submit your `main`. I will use my own and invoke your functions in many creative ways. Hopefully you've thought of all of them.

Note

When you define your functions in one file and invoke them from another, the compiler has no way of telling if the number and type of parameters with which you're invoking them are correct. It can't *have your back*.

To help the compiler help you (yes, those who help others help them usually end up getting the most help), you can tell it in advance what the signatures of these functions you will use from elsewhere are. These function signatures are usually either:

- put at the top of the `cpp` file in which you plan to invoke them or
- collectively put within a header file that is then `#included` in the `cpp` source file

In this quest, simply put the below declarations at the top of your `main.cpp` file (or anywhere else you intend to call your functions from). Starting with the next quest, you will put them in a separate header file (and thus need to upload it together with your `cpp` file at the testing site).

```
// Forward declarations of functions that will be used in this file
// before their definitions are encountered by the compiler

double mean_of_3(int n1, int n2, int n3);
int max_of_5(int n1, int n2, int n3, int n4, int n5);
int min_of_5(int n1, int n2, int n3, int n4, int n5);
bool sides_make_triangle(int a, int b, int c);
bool angles_make_triangle(int A, int B, int C);
bool is_a_leap_year(int year);
```

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
1. Enter the secret password for this quest in the box
2. Drag and drop your `Branching_Functions.cpp` file into the button and press it.
3. Do not submit your source file that contains your `main()` function. If any of your files contains a `main()`, it must be commented out.
4. Wait for me to complete my tests and report back (usually a minute or less).



Need help?

You can find the BLUE discussion forum [here](#).
Questers in the sub are a great resource for help (sometimes through not helping).

May the best coders win. That may just be all of you.

Happy Questin',

&



Three Zebras

Keir Robertson

Loopy Zebras

In this quest, you get to play with looping. You will create a bunch of functions, each of which does something that requires you to iterate over a sequence of steps more than once.

Each of these functions is worth a varying number of trophies. It's up to you to find out how many trophies you can score by making the most progress you can.

As usual, I will give you template code you can copy and flesh out.



Important note

In miniquests where you have to calculate the value of some fractional quantity (e^x and \log terms) you may not use the Math library methods (e.g. `sqrt`, `pow`, etc). If you do, you'll find that your results may be very slightly different from mine. They have to match exactly for you to pass the miniquests. There's the challenge.

Your first miniquest - Guess it

To get past this checkpoint you must correctly implement the following function in the template code:

```
bool play_game(int n);
```

When I invoke this method, I will supply it an integer parameter, `n`, which you must treat as a secret number.

Then you must give the user 6 chances to guess the number using the exact script below:

As soon as the function begins, print the following on the console to greet the user:

```
Welcome to my number guessing game
```

followed by exactly two blank newlines.

Then your function must loop **at most** six times. During each iteration it must ask the user to guess the secret by printing the following exact prompt:

```
Enter your guess:
```

Note - there is one space following the colon at the end of the prompt. You must not end the prompt with a newline.

Then it must confirm what it read back to the user in a new line as follows:

```
You entered: NUMBER
```

where *NUMBER* should be replaced by the value you read from the user.

Now you will compare the user's guess to your secret number. If they are equal, you must return the value `true` to the caller after printing the following message to the console:

```
You found it in N guess(es) .
```

where you would replace the *N* with the actual number of guesses it took the user to find the number. This should be followed by exactly one newline.

If the user's guess is not equal to the secret number, you must simply loop back to the top and repeat the whole sequence.

Except, of course, if you've already done it six times. In that case, you must print a blank line, and the following message (two lines) on the console. Then return `false` to your caller.

```
I'm sorry. You didn't find my number.  
It was SECRET
```

You should replace *SECRET* with the actual secret number.

This shouldn't require more than 20-25 lines of code total (to give you a rough idea of when you're unknowingly over-coding). Refer to the provided code template for more information.

Important: Your user input mechanism must be robust. I may try and break it by inputting a string when an integer is expected. In this case, your program should treat it as a 0, rather than break out of the logic.

You will find it easiest to insulate `cin` from corruption by reading into a string using `getline()` and using `istringstream` to extract an integer from it. Look up or ask how to do it.

None of the remaining functions you write in this quest will require user input.

Your second miniquest - Etox

You must implement:

```
double etox(double x, size_t n);
```

This is simply an extension of the `etox()` function you wrote in a previous quest. The difference is that this function takes two parameters: `x` has the same meaning as before. The new parameter, `n`, is the number of terms you will use from the summation to calculate the required value. To recap, `etox(x, n)` calculates e^x using the formula:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^{n-1}}{(n-1)!}$$

where $n!$ is the factorial of n , which is the product of all positive integers at most equal to n .

Your third miniquest - Char counts

You must implement:

```
size_t count_chars(string s, char c);
```

This function must return the number of occurrences of the given character, c , in the string, s . 'Nuff said.

Your fourth miniquest - Greatest Common Divisor

You must implement Euclid's popular GCD finding algorithm through the function:

```
size_t gcd(size_t a, size_t b);
```

This function must return the GCD of the two given non-negative integers, a and b .

Your fifth miniquest - Terms of an AP

You must implement the function:

```
string get_ap_terms(int a, int d, size_t n);
```

which returns a string containing the first n terms of the arithmetic progression (AP) as a sequence of comma-separated values.

Recall that an AP is specified by two terms a and d . The first term is a , and the d is how much you add to each term to get the next term. So the first N terms of the above AP will be:

$a, a+d, a+2d, \dots, a+(n-1)*d$

Suppose, for example, that $a = 1$, $d = 3$, and $n = 5$. Then this function should return the string "1,4,7,10,13"

Note some important things about the returned value: There are no newlines or spaces. Only commas separate the numbers. There is NO comma before the first number or after the last number. The format has to match exactly.

Your sixth miniquest - Terms of a GP

Copy and adapt the function in the previous mini-quest to turn it into a function that returns the terms of a geometric progression, rather than an arithmetic one. We're talking about changes to a couple of lines.

However, note that the parameter types are different. You should operate in `double` space to keep high precision (not in `integer` space).

Recall that a GP is specified by two terms a and r . The first term of the sequence is a , and the r is by how much multiply each term to get the next term. So the first N terms of the above AP will be:

$$a, ar, ar^2, \dots, ar^{n-1}$$

Suppose, for example, that $a = 4$, $r = 0.5$, and $n = 6$. Then this function should return the string `"4,2,1,0.5,0.25,0.125"`

Your seventh miniquest - Fibonacci

Implement the following function:

```
double get_nth_fibonacci_number(size_t n);
```

which returns the n th number in the Fibonacci sequence. The first two terms of the sequence are both 1. Thereafter, successive terms are simply the sums of their two previous terms. Thus the sequence goes:

1, 1, 2, 3, 5, 8, 13, 21, . . .

Starter code

First your header file:

```
//
//  Looping_Functions.h
//
// This is your header file. No need to make any major changes. You can just feel free
// to copy it verbatim.

#ifndef Looping_Functions_h
#define Looping_Functions_h

// Declarations of the functions in looping_functions.cpp

bool play_game(int n);
double etox(double x, size_t n);
size_t count_chars(std::string s, char c);
size_t gcd(size_t n1, size_t n2);
std::string get_ap_terms(int a, int d, size_t n);
std::string get_gp_terms(double a, double r, size_t n);
double get_nth_fibonacci_number(size_t n);

#endif /* Looping_Functions_h */
```

And then your cpp file:

```
// Student ID: 12345678
// TODO - Replace the number above with your actual Student ID
//
// Looping_Functions.cpp
//
// Created by Anand Venkataraman on 8/9/19.
// Copyright © 2019 Anand Venkataraman. All rights reserved.
//

#include <iostream>
#include <sstream>

using namespace std;

// Give the user 6 chances to guess the secret number n (0-10). If they get it,
// say so and return true. Else say so and return false.
bool play_game(int n) {
    // TODO - Your code here
}

// Calculate e^x using the series summation up to exactly the first
// n terms including the 0th term.
double etox(double x, size_t n) {
    // TODO - Your code here
}

// Return the number of occurrences of char c in string s
size_t count_chars(string s, char c) {
    // TODO - Your code here
}

// Use Euclid's algorithm to calculate the GCD of the given numbers
size_t gcd(size_t a, size_t b) {
    // TODO - Your code here
}

}
```

```
// Return a string of the form n1,n2,n3,... for the given AP.
string get_ap_terms(int a, int d, size_t n) {
    // TODO - Your code here
}

// Return a string of the form n1,n2,n3,... for the given GP.
string get_gp_terms(double a, double r, size_t n) {
    // TODO - Your code here
}

double get_nth_fibonacci_number(size_t n) {
    // TODO - Your code here
}
```

Testing your own code

You should test your functions using your own `main()` method in which you try and call your functions in many different ways and cross-check their return values against your hand-computed results. But when you submit you must NOT submit your main method. I will use my own and invoke your functions in many creative ways. Hopefully you've thought of all of them.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your `Looping_Functions.*` files into the button and press it. Make sure your filename is exactly as above.
4. Wait for me to complete my tests and report back (usually a minute or less).



Need help?

You can find the BLUE discussion forum [here](#). Questers in the sub are a great resource for help (sometimes through not helping).

May the best coders win. That may just be all of you.

Happy Questin',

&



الحاجه و از ابر کشت و دست به سول شده بطرف آسمان و از شنبه پانچ تمام مردم سیدیند سپکا چارج و آن منی کمر جو خانی مدینه می یاد و کینه و در

Silly Snake

This should be a fun little quest. You will create a doctor program called Eliza.

Eliza will be able to carry on a simple, albeit contrived, conversation with her patient.



Your first miniquest - Hisspify

I mean lispify.

You must implement:

```
string lispify(string s);
```

When I invoke this method, I will supply it a string parameter. You must accept this parameter by copy (not reference). Look it up or ask and understand before proceeding.

It must return a string which is identical to the one I gave you except that all s's in the string have been substituted by th's. For example:

```
lispify("sixty six")
```

should return the string "thixty thix". Yeah, I know there's an s sound in the x, but leave it alone in this quest.

Once you have this working, try to see if you can implement a case-sensitive version. It's identical to the previous version except that upper-case S must get replaced by "Th", not "th".

Useful guide: If your code for this function is over 10 lines including comments, look for signs of over-coding.

Your second miniquest - Rotate Vowels

You must implement:

```
string rotate_vowels(string& s);
```

This function must accept a string parameter *s* by reference, and change it directly by replacing each vowel with its alphabetical successor among vowels defined circularly. That is, replace **a** with **e**, **e** with **i**, **i** with **o**, **o** with **u** and **u** with **a**. For example, after executing the following two lines,

```
string s = "that's really cool";  
rotate_vowels(s);
```

the variable *s* should contain the string "thet's rielly cuul"

In addition to changing the parameter passed in by reference, it must ALSO return this parameter.

See what happens if you implement a case-sensitive version of this function.

Useful guide: If your code for this function is over 15 lines including comments, look for signs of over-coding.

Your third miniquest - Enter

Now you'll weave the previous functions together with a third to create a fun experience.

You must implement:

```
void enter();
```

which should communicate with the user via the console. It has the following very tight script. You need to adhere to it as accurately as possible to score the most points.

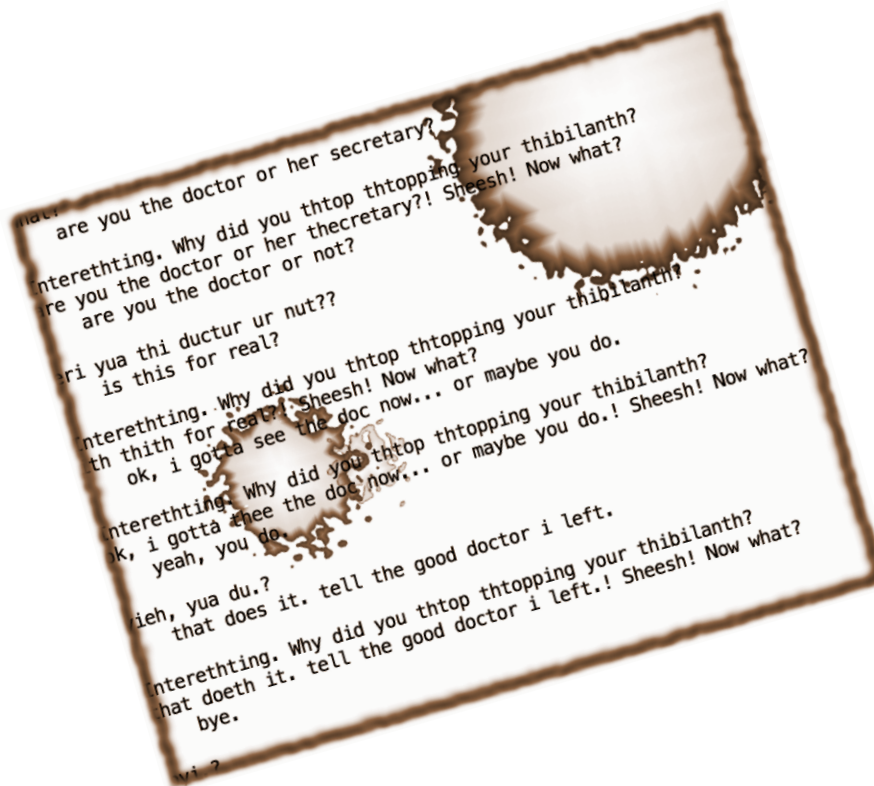
1. Immediately upon being invoked, it must curtly greet the user (via `std::cout`) using the string `"What?"` Note that there is no space after the question mark. It must be followed by one newline.
2. Then enter an infinite loop. In each iteration of this loop,
 - a. You must read a *whole line of input* from the user and respond to each as below (sequence is important, and responses must end in a newline)
 - b. If the user's input is an empty string, you must print the `"What?"` line again and short-circuit back to the top of the loop (skip the rest of the steps below).
 - c. Print 4 spaces followed by the user's input and two newlines.
 - d. If the user input contains an exclamation mark (`!`), you must print the line `"OMG ! You don't say!! "` followed by the user input and exactly five exclamation marks. Then revert to the top of the loop.
 - e. If the user input contains either of the words "why" or "what", you must print the string "I'm sorry, I don't like questions that contain what or why." Then revert to the top of the loop.
 - f. If the user input contains the letter s, then you must print two lines as follows and then revert to the top of the loop.
 - i. `"Intereththing. When did you thtop thtopping your thibilanth?"`
 - ii. the return value of your `lispify()` function called with `user_input` as its parameter, followed by the string `"! Sheesh! Now what?"`
 - g. If the user says bye or quit (or Bye or Quit), print the message `"Ok Bye. Nice being a force of change in your life."` and break out of your loop.
 - h. If none of the above conditions matches you must do the following:

- i. 80% of the time print the return value of `rotate_vowels()` invoked using the user input as its parameter, followed by a question mark.
- ii. 20% of the time print the string "Huh? Why do you say: " followed by the user input and then a question mark.

To do the 80/20 split, here's what you do:

1. Generate a random number (use `rand()`)
2. Calculate the remainder after dividing by 10 (This would be `rand() % 10`)
3. If the remainder is 8 or 9, then follow option (ii). Else follow option (i)
4. Important: Use portions of 10, not 100.

It is important that you don't call `srand()` ANYWHERE in your submitted code. I need to be able to control the seed to test your code.



Starter code

First your header file. You need to place declarations (not definitions) of your three functions in this file.

```
//  
// Eliza.h  
// BLUE.Quest-05-Eliza  
//  
  
#ifndef Eliza_h  
#define Eliza_h  
  
// TODO - place your Eliza function declarations here  
  
#endif /* Eliza_h */
```

And then your cpp file:

```
// Student ID: 12345678  
// TODO - Replace the number above with your actual Student ID  
  
// Eliza.cpp  
// BLUE.Quest-05-Eliza  
//  
//  
#include <iostream>  
#include <sstream>  
  
using namespace std;  
  
// Return a new string in which the letters (lowercase) a, e, i, o, and u  
// have been replaced by the next vowel in the sequence aeiou. Replace u by a.  
//  
// Note that the string is passed in by reference. So the caller may not  
// rely on the result being returned.  
// TODO - Your code for rotate_vowels goes here  
  
// Return a string in which all occurrences of s have been replaced by th  
// TODO - Your code for lispify goes here  
  
// Enter the user-interaction loop as described earlier  
void enter() {  
    // TODO - Your code here  
}
```

Testing your own code

You should test your functions using your own `main()` method in which you try and call your functions in many different ways and cross-check their return value against your hand-computed results. But when you submit you must NOT submit your main method. I will use my own and invoke your functions in many creative ways. Hopefully you've thought of all of them.



Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
1. Enter the secret password for this quest in the box.
2. Drag and drop your `Eliza.*` files into the button and press it.
3. Wait for me to complete my tests and report back (usually a minute or less).

Need help?

You can find the BLUE discussion forum [here](#). Questers in the sub are a great resource for help (sometimes through not helping).

May the best coders win. That may just be all of you.

Happy Questin',

&



Clever Crow

In this quest you get to implement an entire class.

This class is called `Pet`. In addition to implementing this class, you also get to implement a fun little function to make up names for the millions of pets you will be able to create.

Your first miniquest - Make a name for yourself

This is just warm up before you get to implement a class.

You must implement:

```
string make_a_name(int len);
```

When I invoke this method, I will supply it with a length parameter which tells you my desired name length.

You must then algorithmically manufacture a name and return it to me.

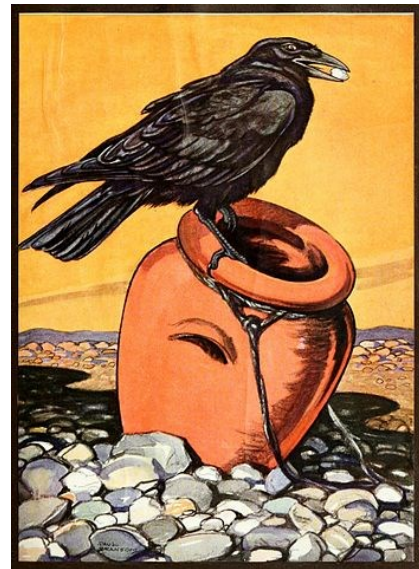
Before you start, read about the `rand()` function (and `srand()`) in the reference material or online. Ask questions in the forums to clarify if necessary.

Since I need to match up names generated by me and you to make sure your implementation is to spec, there are a few precautions you have to follow in implementing this function.

Specifically, you must:

- define your vowels to be the characters in the string "aeiou"
- define your consonants to be the characters in the string "bcdfghjklmnpqrstvwxyz"
- The name you return must be created by alternately selecting a random letter from each of the above two strings.
- The very first letter of your name must be either a vowel or consonant chosen using the condition `(rand() % 2 == 0)`. Specifically, if your random is even, then the first letter of the name must be a consonant.
- To select the index of the letter to use in each iteration, you must invoke `rand()` exactly once. For example: `rand() % consonants.length()`
- **IMPORTANT:** You must NOT invoke `srand()` anywhere in your code.

Now you must implement the `Pet` class as specified below and incorporate `make_a_name()` into it as a public static helper method.



The Pet Class

Your Pet object ought to contain the following four private members of which one is also static (static means that there is only one copy of the variable and it is attached to the class definition - the blueprint of the class - rather than several copies, one in each object created from that class definition. Discuss this in the forums to understand more clearly if necessary.)

- `string _name;`
- `long _id;`
- `int _num_limbs;`
- `static size_t _population;`

Note that I have prefixed each member's name with an underscore. This is a convention some programmers follow to help fellow programmers instantly spot private members in code manipulating an object. Try this strategy in this quest and see if it helps. My starter code will be set up to help you do it this way. It has the nice side-effect that you can now freely use the words `name`, `id` and `num_limbs` as names for your local variables without needing to use `this->` to disambiguate.

Miniquest 2 - Pet class constructors and destructor

Implement the default and non-default constructors of your Pet class as follows:

```
Pet(string name = "", long id = -1, int num_limbs = 0);
```

By providing the default values for your members in the declaration, you can skip defining an explicit default constructor. Your implementation of the above constructor gives you both the default and non-default varieties. Both will be tested. Make sure to retain the exact default values you see above.

```
~Pet();
```

The public destructor of the Pet object, with the above signature, will be automatically invoked by the compiler when the memory of a Pet object is to be released back into the resource pool. Most of the time you don't need to implement explicit destructors for simple tasks. In this case, we're using it because it gives us an opportunity to experiment with static (or class) variables. Read up what a class variable is and how it is different from regular class members.

Your destructor must decrement the value of your `_population` member - Just like a constructor should increment it. Make sure that after you create and destroy an arbitrary number of Pet objects your population count always shows 0.

Miniquest 3 - Getters (aka accessors)

Implement getters for each of your members. They should simply return the requested value while leaving the object untouched.

To accidentally prevent this from happening, we often annotate such methods with the **const** keyword (purple below). This tells the compiler to have our back. It will look out for us if we do something stupid like make getters (or functions it calls) change an object's properties.

- `string get_name() const;`
- `long get_id() const;`
- `int get_num_limbs() const;`

Miniquest 4 - Setters (aka mutators)

Implement setters for each of the three members as follows:

- `bool set_name(string name);`
- `bool set_id(long id);`
- `bool set_num_limbs(int num_limbs);`

Note that each setter returns a boolean value. Whenever possible, I recommend you make your setters return a success/failure value so the caller can know whether they succeeded. I think it is usually overkill to throw an exception from a setter. So this habit will serve you well even after you learn about exceptions and start coding advanced structures.

Each setter needs to validate its input as follows:

- `set_name()` should refuse to set empty strings as names
- both `set_id()` and `set_num_limbs()` should refuse to set negative numbers.

Miniquest 5 - Stringification

Implement the method

```
string Pet::to_string() const;
```

When called on a `Pet` object it should return a string representation of the object formatted *exactly* as follows:

```
" (Name: [NAME], ID: [ID], Limb Count: [NUMBER OF LIMBS]) "
```

where the parts in square brackets need to be replaced by the values of the members in the `Pet` object on which `to_string()` was invoked. To help you format properly I colored in the spaces in the string above. Each gray rectangle denotes exactly one space.

Miniquest 6 - Get a whole bunch of pets at once

Implement the method

```
static void get_n_pets(size_t n,
                      std::vector<Pet>& pets,
                      int name_length);
```

When this method returns the vector `pets` must be appropriately resized and each of its elements must be a `Pet` object. Eventually I'll figure out a better way to test this method, but for now, I'm afraid I have to choreograph much of its dance. Refer to the starter code for this function and complete it as directed.

In essence, it creates the requested number of pets, assigning them strictly increasing IDs and random names of the requested length.

We'll take advantage of this default ordering in the next quest.

Miniquest 7 - Population control

Make sure your constructors increment your population count and destructors decrement it. If you have your population under control you get extra points.

Miniquest 8 - Implement Equality for all

Implement

```
bool operator==(const Pet& pet1, const Pet& pet2);
```

See how c++ lets you redefine the functionality of basic operators you have grown accustomed to and love. Complete this global function to make it return `true` if and only if an object's components are each equal to the corresponding components of the object to which it is being compared. Note that this functionality is defined as a global function, not as an object method. What are the advantages and disadvantages of doing it each way? Discuss in the forums.

Miniquest 9 - Use equality to define what inequality is

Implement

```
bool operator!=(const Pet& pet1, const Pet& pet2);
```

However, don't implement the logic for it independently. You must define it in terms of the equals operator you defined earlier.

Miniquest 10 - Give everything a voice

Implement

```
ostream& operator<<(ostream& os, const Pet& pet);
```

c++ even lets you define the functionality of the output (some call it insertion) operator (<<). When someone issues a statement like:

```
std::cout << my_pet <<std::endl;
```

where `my_pet` is a variable of type `Pet`, then the code in this function will be executed. Since you already have a `to_string()` defined from before, you know exactly what to do here. Note that this function returns the output stream passed to it as a parameter. That is what allows us to chain output statements like this:

```
std::cout << a << b << c << . . .
```

Important implementation note

Unfortunately, I have to impose this constraint. Terribly sorry about that.

You may not call `srand()` anywhere in your submitted code, and you have to adhere to the very strict guidelines about exactly where your code is allowed to call `rand()`.

Obviously you have to call these functions in your own testing code. Just make sure that they're not in the code you submit.

Finally, keep in mind that none of these methods is allowed to talk to the user. If you try to print something to the console or read from it in your submitted code, it will likely fail a whole bunch of test cases.



Starter code

You will submit two files: `Pet.h` and `Pet.cpp` - The `.h` file should contain the definition of your `Pet` class. The `.cpp` file should contain its implementation. Usually these two files go together.

Here is your `Pet.h` starter code

```
// Student ID: 12345678
// TODO - Replace the number above with your actual student ID
//
// Pet.h
// Pets with an ID, a unique name and number of limbs.
// The Pet class also supports a static population member

#ifndef Pet_h
#define Pet_h

using namespace std;

class Pet {
private:
    string _name;
    long _id;
    int _num_limbs;

    static size_t _population;

public:
    Pet(string name = "", long id = -1, int num_limbs = 0);
    ~Pet();

    string get_name() const;
    long get_id() const;
    int get_num_limbs() const;

    bool set_name(string name);
    bool set_id(long id);
    bool set_num_limbs(int num_limbs);

    string to_string() const;

    static void get_n_pets(size_t n, std::vector<Pet>& pets, int name_length);
    static size_t get_population() { return _population; }

    static string make_a_name(int len);

    // Don't remove this line
    friend class Tests;
};

std::ostream& operator<<(std::ostream& os, const Pet& pet);
bool operator==(const Pet& pet1, const Pet& pet2);
bool operator!=(const Pet& pet1, const Pet& pet2);

#endif /* Pet_h */
```

And your Pet.cpp file:

```
// Pet.cpp
// BLUE-Quest-06-Pets
//

#include <iostream>
#include <sstream>
#include <vector>

#include "Pet.h"

using namespace std;

// This is a way to properly initialize (out-of-line) a static variable.
size_t Pet::_population = 0;

Pet::Pet(string name, long id, int num_limbs) {
    // TODO - Your code here
}

Pet::~Pet() {
    // TODO - Your code here
}

string Pet::get_name() const { return _name; }
long Pet::get_id() const {
    // TODO - Your code here
}

int Pet::get_num_limbs() const {
    // TODO - Your code here
}

bool Pet::set_name(string name) {
    // TODO - Your code here
}

bool Pet::set_id(long id) {
    // TODO - Your code here
}

bool Pet::set_num_limbs(int num_limbs) {
    // TODO - Your code here
}

string Pet::to_string() const {
    // TODO - Your code here
}

// Fill in the supplied pets vector with n pets whose
// properties are chosen randomly.
// Don't mess with this method more than necessary.
void Pet::get_n_pets(size_t n, std::vector<Pet>& pets, int name_len) {
    // TODO - Resize pets as necessary
    long prev_id = 0;
    for (size_t i = 0; i < n; i++) {
        long id = prev_id + 1 + rand() % 10;
        pets[i].set_id(id);
        pets[i].set_num_limbs(rand() % 9); // up to arachnids

        // TODO - make and set a name of the requested length
        // TODO - adjust prev_id as necessary
    }
}

// -----
```

```
string Pet::make_a_name(int len) {
    // TODO - Your code here
}

// Optional EC points
// Global helpers
bool operator==(const Pet& pet1, const Pet& pet2) {
    // TODO - Your code here
}

bool operator!=(const Pet& pet1, const Pet& pet2) {
    // TODO - Your code here
}

ostream& operator<<(ostream& os, const Pet& pet) {
    // TODO - Your code here
}
```


Testing your own code

You should test your functions using your own `main()` function in which you try and call your methods in many different ways and cross-check their return values against your expected results. But when you submit you must NOT submit your `main()` function. I will use my own and invoke your functions in many creative ways. Hopefully you've thought of all of them.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
1. Enter the secret password for this quest in the box
2. Drag and drop your `Pet.*` files into the button and press it. (Make sure you're not submitting a `main()` function)
3. Wait for me to complete my tests and report back (usually a minute or less).



Need help?

You can find the BLUE discussion forum [here](#). Questers in the sub are a great resource for help (sometimes through not helping).

May the best coders win. That may just be all of you.

Happy Questin',

&





Don Cobb Art



Don Cobb Artwork
<https://www.etsy.com/shop/DonCobbArtwork>

Purple Martin Time

The Purple Martin

In this quest you get to create a Pet Store using the Pets you created in the last quest.

The name of the class you will implement is `Pet_Store`. In addition to implementing this class, you will have some fun creating some efficient (and not-so-efficient) search functions.

And along the way you'll pick up a new concept: Enumerated types (enums). Now is the time to read up about enums in your reference material of choice, before you start your quest.



Your first miniquest - Make a Store

You must implement the class `Pet_Store` containing an inner enumerated type called `_SORT_ORDER` which can have three possible values:

- `BY_ID`
- `BY_NAME`
- `NONE`

Define it as follows:

```
enum _SORT_ORDER {  
    BY_ID,  
    BY_NAME,  
    NONE  
};
```

Once the compiler sees the above declaration, you can assume that you have created a brand new type of your own (just like `int` or `bool` or the other types you're used to). The special thing about this new type you just created is that it can only take the three values named in its declaration.

It is important to make sure you retain the ordering of the enum elements I give above. The compiler will internally represent them using consecutive integer values starting at 0 for the first element.

The only private members of the `Pet_Store` class are the following:

1. a `std::vector` of `Pet` objects called `_pets`;
2. an enum of type `_SORT_ORDER` called `_sort_order`.

Your public constructor for making a Pet Store object is:

```
Pet_Store::Pet_Store(size_t n = 0);
```

It should correctly size its internal Pet vector and set the `_sort` order to `NONE`.

Your second miniquest - Get the size

Implement the public method

```
size_t Pet_Store::get_size() const;
```

I'll try and create many different pet stores of different sizes and make sure the sizes you report are the sizes I asked for.

Your third miniquest - Set the size

Implement the public method

```
void Pet_Store::set_size(size_t n);
```

This should resize your pet store as requested. It should leave the `_sort_order` untouched.

Your fourth miniquest - Clear the store

Implement the public method

```
void Pet_Store::clear();
```

This should simply call the vector's `clear()` on the `_pets` vector. That's all.

Your fifth miniquest - Populate with a number of random pets

Implement the public method

```
void Pet_Store::populate_with_n_random_pets(size_t n);
```

Use the static method you created in the previous quest in the `Pet` class, `Pet::get_n_pets()`, and populate the `_pets` vector with `n` Pets whose names are each 7 letters long.

Since `get_n_pets` assigns IDs in strictly increasing order, you can (and should) now set the sort order correctly to `BY_ID`.

Your sixth miniquest - Find a pet by ID using linear search

Implement the public method

```
bool Pet_Store::find_pet_by_id_lin(long id, Pet& pet);
```

I'm using the suffix, `_lin`, in the name of the method to imply *linear search*. In a subsequent miniquest, you will be implementing the same functionality, but using binary search.

First, note the signature and ponder/discuss why it is like that. Kudos may await **short** insightful posts.

When I invoke this method, I will supply it a long ID value and an overwritable `Pet` object by reference. This method should do a sequential scan of `Pets` in its `_pets` vector checking to see if any of the pets has the requested ID. If one is found, it should fill in the passed `Pet` object with the first such found `Pet` object's details. It should then return `true`.

If the requested ID is not found in the `_pets` vector, it should simply return `false` without touching the passed `Pet` object.

Your seventh miniquest - Find a pet by ID using binary search

Implement the public method

```
bool Pet_Store::find_pet_by_id_bin(long id, Pet& pet);
```

The outside-facing behavior of this method is identical to that of the previous one (linear search). But internally, this method should try to locate the requested ID using binary search, not a sequential scan.

It is critical that the `_pets` vector is sorted in non-descending order by `_id` before you begin your search. Thus, the first thing you must do here is check the `_sort_order` property of the store. If it isn't `BY_ID`, then you must invoke the helper method (supplied) `_sort_pets_by_id()`. You can see the body of this simple function in the starter code I'm giving you. Feel free to ask for clarifications and discuss things in the forums.

Implementation of binary search can be tricky if you're doing it for the first time. Pay special attention to corner cases (values of variables near loop boundaries, etc.).

I highly recommend you implement your search iteratively. Don't mess around with recursion just yet. Don't write additional helper methods.

This miniquest is worth a lot of loot. If done correctly, it will be absolutely worth it.

Since the interface of this method is identical to that of the previous one, I hope you are not tempted to use the same sequential scan logic in this method. If you do, it will fail my random test cases when I try to search `LARGE` vectors `MANY` times.

Your eighth miniquest - Find a pet by name using linear search

Implement the public method

```
bool Pet_Store::find_pet_by_name_lin(string name, Pet& pet);
```



This would be the same as its id-searching counterpart from before, except that it looks for the given name rather than an ID.

Your ninth miniquest - Find a pet by name using binary search

Implement the public method

```
bool Pet_Store::find_pet_by_name_bin(string name, Pet& pet);
```

I bet you don't need me to tell you what this needs to do. But just in case... this should do the same thing as the previous method except that it should employ binary rather than linear search, internally.

Just like in binary search for an ID, you must first check if the `_sort_order` of the store is `BY_NAME`. If not, you must invoke a corresponding helper function (supplied) before starting your search.

Your tenth miniquest - Serialization

Implement the public method

```
string Pet_Store::to_string(size_t n1, size_t n2);
```

When I invoke this method, I will supply two numbers `n1` and `n2`. You should return to me a string of newline separated Pets at indices `n1` through `n2` (inclusive) provided such an index is valid (i.e. smaller than the size of your `_pets` vector). There should be exactly one newline character after each Pet object, including the last one.

Each `Pet` in the result string should itself stringified using its own `to_string()` method from your previous quest.

Important implementation note

Unfortunately, I have to impose the following constraint again. Terribly sorry about that.

You may not call `srand()` anywhere in your submitted code, and you have to adhere to the very strict guidelines about exactly where your code is allowed to call `rand()`.

Obviously you have to call these functions in your own testing code. Just make sure that they're not in the code you submit.

Finally, keep in mind that none of these methods is allowed to talk to the user. If you try to print something to the console or read from it in your submitted code, it will likely fail a whole bunch of test cases.

Starter code

You will submit four files, including the two from the previous quest): `Pet.h` and `Pet.cpp`. These files don't have to be identical to your submissions in the previous quest. You may have worked more on them, perhaps fixed more bugs or incorporated corrections from my model solution. The two new files are `Pet_Store.h` and `Pet_Store.cpp`. The `.h` file should contain the definition of your `Pet_Store` class. The `.cpp` file should contain its implementation.

Here is your `Pet_Store.h` starter code

```
// Student ID: 12345678
// TODO - Replace the number above with your actual student ID
//
// Pet_Store.h
//
#ifndef Pet_Store_h
#define Pet_Store_h

#include <algorithm> // Needed for sort
#include <vector>

#include "Pet.h"

class Pet_Store {
private:
    std::vector<Pet> _pets;

    enum _SORT_ORDER { BY_ID, BY_NAME, NONE };
    _SORT_ORDER _sort_order = BY_ID;

    // Provided functions (no points)
    static bool _id_compare(const Pet& pet1, const Pet& pet2);
    static bool _name_compare(const Pet& pet1, const Pet& pet2);
    void _sort_pets_by_id();
    void _sort_pets_by_name();

public:
    Pet_Store(size_t n = 0);

    size_t get_size() const;
    void set_size(size_t n);
    void clear();

    void populate_with_n_random_pets(size_t n);
    bool find_pet_by_id_lin(long id, Pet& pet);
    bool find_pet_by_id_bin(long id, Pet& pet);
    bool find_pet_by_name_lin(string name, Pet& pet);
    bool find_pet_by_name_bin(string name, Pet& pet);

    // Return a string with the concatenated string reps of valid pets with
    // index n1 to n2-1 (inclusive)
    std::string to_string(size_t n1, size_t n2);

    // Don't remove the following line
    friend class Tests;
};

#endif /* Pet_Store_h */
```

And your Pet_Store.cpp file:

```
// Pet_Store.cpp

#include <iostream>
#include <sstream>

#include <vector>

#include "Pet_Store.h"

using namespace std;

Pet_Store::Pet_Store(size_t n) {
    // TODO - Your code here
}

void Pet_Store::set_size(size_t n) {
    // TODO - Your code here
}

size_t Pet_Store::get_size() const {
    // TODO - Your code here
}

void Pet_Store::clear() {
    // TODO - Your code here
}

void Pet_Store::populate_with_n_random_pets(size_t n) {
    // TODO - Your code here
}

// These two functions can be conveniently made anonymous "lambda" functions
// defined within the scope of the functions where they're used (but only
// c++-11 on. For now we're just going to leave them here. It's straightforward
// to move them in. Just look up c++ lambda functions if you want. If you want
// to know but don't understand it, I'm happy to explain what they are. Ask me
// in the forums. It's not necessary to know about it to ace this course.
//
// You are free to experiment by hacking the functions below, but restore their
// correct functionalities before submitting your quest.
//
bool Pet_Store::_id_compare(const Pet& p1, const Pet& p2) {
    return p1.get_id() < p2.get_id();
}

bool Pet_Store::_name_compare(const Pet& p1, const Pet& p2) {
    return p1.get_name() < p2.get_name();
}

void Pet_Store::_sort_pets_by_id() {
    std::sort(_pets.begin(), _pets.end(), Pet_Store::_id_compare);
    _sort_order = BY_ID;
}

void Pet_Store::_sort_pets_by_name() {
    std::sort(_pets.begin(), _pets.end(), Pet_Store::_name_compare);
    _sort_order = BY_NAME;
}

bool Pet_Store::find_pet_by_id_lin(long id, Pet& pet) {
    // TODO - Your code here
}

bool Pet_Store::find_pet_by_name_lin(string name, Pet& pet) {
    // TODO - Your code here
}

// -----
```



```

// When this method starts, the _pets[] vector must be sorted in
// non-descending order by _id. If it is not already, then it will be resorted.

bool Pet_Store::find_pet_by_id_bin(long id, Pet& pet) {
    // TODO - Your code here
}

// When this method is called, the _pets[] vector must be sorted in
// lexicographic non-descending order by _name. If it is not already,
// then it will be resorted.

bool Pet_Store::find_pet_by_name_bin(string name, Pet& pet) {
    // TODO - Your code here
}

// Return a string representation of the pets with indexes n1 through n2
// inclusive, exclusive of non-existent indices
// Each pet is on a line by itself.

string Pet_Store::to_string(size_t n1, size_t n2) {
    // TODO - Your code here
}

```



Testing your own code

You should test your functions using your own `main()` function in which you try and call your methods in many different ways and cross-check their return values against your expected results. But when you submit you must NOT submit your `main()` function. I will use my own and invoke your functions in many creative ways. Hopefully you've thought of all of them.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
1. Enter the secret password for this quest in the box.
2. Drag and drop your four `Pet*.*` files into the button and press it. (Make sure you're not submitting a `main()` function)
3. Wait for me to complete my tests and report back (usually a minute or less).



Need help?

You can find the BLUE discussion forum [here](#). Questers in the sub are a great resource for help (sometimes through not helping).

May the best coders win. That may just be all of you.

Happy Questin',

&



An Elephant Never Forgets...

Well, for starters, I have no idea if that's true.

I mean, if an elephant ever challenged me and said, "I never forget. So there", I'd be wise to say "Whatever you say, boss. Ain't no matter to me."

Anyway, that's the name of this quest (not its password).

In this relatively simple quest, you get to implement a stack abstract data type. In fact you get to implement two stacks for the price of one.

Some novel things you can discover in this quest:

1. Your entire implementation will be inline within the class definition. No need to create a separate .cpp file (obviously this is not a suitable strategy for everything you do, but you can get a feel for doing it this way so that in future you can make more informed decisions about how to segment your code.
2. You will actually implement both your required classes in the same header file. This header file will be called `Stacks.h` and should contain implementations of: `Stack_Int` and `Stack_String`.

Here's a cool thing about this quest. The code for your `Stack_String` class will be ALMOST identical to the code for your `Stack_Int` class. You'll find yourself copying and pasting huge chunks of your code. Or better yet, copying the whole thing and making minor edits here and there.



There's a reason I suggest doing this potentially redundant work. In the process of porting your `Stack_Int` into a `Stack_String`, you will find yourself thinking "Hmm... what a dumb mechanical thing to be doing. I wonder if there's a way to get the compiler to do it for me."

If so, that's awesome. As you continue your C++ romance into GREEN, you'll find ways to get the compiler to do exactly that.

Here's another cool thing. You don't have to implement constructors or destructors for your classes. Instead, you will leverage the luxury of the compiler that gives you default ones for free. Your `Stack_Int` class should have a single private data member called `_data` which would be a `std::vector` of ints. You will leverage standard library methods provided by the vector class to accomplish all of what a Stack does without compromising performance.

Stack Overview

A stack is what you may call a Last In First Out (LIFO) data structure. Typically a stack is implemented using a far more versatile data structure by stripping away(or hiding) some of the things you can do with the more versatile data structure.

For example, consider an array, which allows you to pretty much do whatever you want to any place in it - You can assign values to arbitrary locations in the array. But you can also take an array and say, "I'm going to wrap up this array within my own container and prevent others from accessing its entire power. Instead, I'm only going to allow users to access one end of the array."

You may be thinking, "Who in their right mind would do something like that?"

I mean, why would you take something as powerful as an array and dress it down so someone can only use a small subset of the features it provides? That's like, if I go on Amazon and see 10-packs and 2-packs of something I like for the same price, me saying "Wow! *That's really cool. Lemme buy five two-packs instead of a ten-pack.*"

Incredibly, as we have discovered time and time again, you gain an enormous amount of creative power by sacrificing a small bit of freedom. In effect, you're saying "I hereby sacrifice my ability to do random crazy things to my array. So give me a version of the array that doesn't let me do that, and in return absolves me of the headache of having to remember potential code-gotchas involving invocations of these (self) forbidden functions. You will be pleasantly surprised at how creative you can get once you stop worrying about a whole bunch of things that could break (but now won't because you've hidden them away from prying eyes). This, basically, is the core insight behind Object Oriented Programming.

So, we'll do exactly what far-sighted common sense tells us. We'll use a powerful array (vector) under the covers to implement everything that a stack interface provides to its users. But we'll hide the fact that it's an array from the user. Thus they can only ever manipulate it using our own public functions as an intermediary. Since we have absolute control over our own public methods, we get to decide what users can and can't do to our underlying data array.

The following functionality is what you will implement for your stack. Each of the following is a miniquest worth a possibly different number of trophies. Some of these are critical miniquests. That means you can't proceed further in this quest or to the next one unless you complete it. Some other miniquests will let you proceed, but you may not get the trophies for the failed miniquests.

Here is the specification for your Stack_int class:

```
class Stack_Int {
private:
    std::vector<int> _data;

public:
    // No explicit constructor or destructor. You must implement the remaining
    // methods below in-line (directly in this header file) within the class-def.
    // See the starter code for more detail.

    size_t size() const;
    bool is_empty() const;
    void push(int val);
    int top(bool& success) const;
    bool pop();
    bool pop(int& val);
    string to_string() const;
};
```

As you can see, it's quite simple. Below are your miniquests. There's at least one for each of the public methods you must implement.

Your first miniquest - Check if your stack is empty

Implement:

```
bool Stack_Int::is_empty() const;
```

It should return true if the stack (_data vector) is empty and false otherwise. Just return the value of the `vector::empty()` method. It's a one-liner. Hopefully the first of countless one-liners you will write in your programming life.

Your second miniquest - Check your stack's size

Implement:

```
size_t Stack_Int::size() const;
```

It should return the size of the _data vector. Another one-liner! Just return the value of the `vector::size()` method.

Your third miniquest - Push

Implement:

```
void push(int val);
```

It should insert the given value into your stack. Choose which end of the stack you're going to treat as the top of your stack. This will become important as your stack grows in size (Why? Discuss it in the forums.)

Your fourth miniquest - Top

Implement:

```
int top(bool& success) const;
```

Note the unusual signature I've chosen for this method. What do you think is the reason for this? What problem am I trying to solve? What are the pros and cons of doing it this way as opposed to some other way?

When called on a `Stack_Int` object it must return the value currently occupying the top of the stack, without changing the stack data in any way (hence the `const` qualifier in the method signature).

If the stack is empty, you should return 0 and set the boolean parameter to `false`. Note that it's passed by reference. Otherwise, it should return the top element and set the parameter to `true`.



Your fifth miniquest - One kind of pop

Implement:

```
bool pop();
```

This method simply removes the top element off the data vector and return true.

You can do this using a one-line `std::vector` method. If the vector is empty, you would not do that and return false.

Your sixth miniquest - Another kind of pop

Implement:

```
bool pop(int& val);
```

This kind of `pop()` is virtually identical to your `top()` method, except that it has the side-effect of removing the element that is returned if the stack is not empty.

Your seventh miniquest - Stringify

Implement:

```
string to_string() const;
```

This method must return a string representation of the stack of integers in the following exact format. As usual, I've colored in the spaces. Each gray rectangle stands for exactly one space.

```
Stack (<N> elements):
```

```
<element 1>
```

```
<element 2>
```

```
<. . .>
```

```
Elements, if listed above, are in increasing order of age.
```

The parts in red above (also in angle brackets) must be replaced by you with the appropriate values.

There is no newline after the last line that ends with "age."

You would print a maximum of 10 elements this way. If the stack has more than 10 elements, you must print the top (most recent) 10 and then print a single line of ellipses (. . .) in place of all other elements.

Your eighth and final miniquest - Stack o' Strings

You get to implement a whole other class: `Stack_String`.

Copy the code for your `Stack_Int` class and adapt it to work with strings instead. This is an exercise in porting code which requires a level of understanding about what the code does.

Your `Stack_String` should do everything your `Stack_Int` does... except, with strings.



Starter code

You will submit one file: `Stacks.h`. Here is your starter code

```
// Student ID: 12345678
// TODO - Replace the number above with your actual student ID
//
#ifndef Stacks_h
#define Stacks_h

#include <vector>
#include <sstream>

class Stack_Int {
private:
    std::vector<int> _data;

public:
    // No explicit constructor or destructor
    size_t size() const {
        // TODO - Your code here
    }

    bool is_empty() const {
        // TODO - Your code here
    }

    void push(int val) {
        // TODO - Your code here
    }

    int top(bool& success) const {
        // TODO - Your code here
    }

    bool pop() {
        // TODO - Your code here
    }

    bool pop(int& val) {
        // TODO - Your code here
    }

    std::string to_string() const {
        // TODO - Your code here
    }

    // Don't remove the following line
    friend class Tests;
};

class Stack_String {
    // TODO - Your code here. Ask in the forums if you're stuck.
};

#endif /* Stacks_h */
```

Testing your own code

You should test your functions using your own `main()` function in which you try and call your methods in many different ways and cross-check their return value against your expected results. But when you submit you must NOT submit your `main()` function. I will use my own and invoke your functions in many creative ways. Hopefully you've thought of all of them.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
1. Enter the secret password for this quest in the box.
2. Drag and drop your `stacks.h` file into the button and press it. (Make sure you're not submitting a `main()` function)
3. Wait for me to complete my tests and report back (usually a minute or less).

Need help?

You can find the BLUE discussion forum [here](#). Questers in the sub are a great resource for help (sometimes through not helping).

May the best coders win. That may just be all of you.

Happy Questin',

&





ORNITHORHYNCHUS ANATINUS.

Illustrated from a specimen in the collection of the British Museum.

H. De la Beche del.

Playful Platypi

Personally, I've never played with one. But I'm thinking "Why shouldn't they be playful? They sure look like they wanna have fun."

I gotta say I've never seen one programming in C++, though. So who knows?

Some miniquests will give you some trophies for meeting a minimum requirement, and give you additional surprise points for doing something else right that wasn't explicitly asked.

In this quest, you will create an inline implementation for a class called `String_List` just like in your previous quest.

Here is what your `String_List` class should look like

```
class String_List {
private:
    struct Node {
        std::string data;
        Node *next;
        Node(std::string s = "") : data(s), next(nullptr) {}
    };
    Node *_head, *_tail, *_prev_to_current;
    size_t _size;

public:
    // TODO - You will need to fill in the inline implementations of
    // all the public methods below.
    String_List();
    ~String_List();

    size_t get_size() const;

    void clear();

    String_List *rewind();
    String_List *insert_at_current(std::string s);
    String_List *remove_at_current();
    String_List *push_front(std::string s);
    String_List *push_back(std::string s);

    // Remember that _prev_to_current can never be null because it's at
    // least equal to the list header. The current element visible to the user
    // is always the element AFTER _prev_to_current. So when you're advancing,
    // you can't advance further than the tail node (again a real non-null node).
    //
    String_List *advance_current();

    // This method is only a few lines, but please make sure you understand it
    // fully. Ask in the forums if any of it is unclear.
    //
    // The current item is the data element of whichever node is equal to
    // _prev_to_current->next.
    //
    // Ideally, we would throw an exception if _prev_to_current->next == null, which
    // should not happen, really. But since we haven't covered exceptions yet, we will
    // hack a return to a static sentinel, with the understanding that the caller
    // won't mess with the sentinel (see the spec).

    std::string get_current() const;

    // Find a specific item. Does NOT change cursor.
    //
    //
```



```
// The following returns a reference to the target string if found. But what will
// you do if you didn't find the requested string? Using sentinel markers is
// one way to handle that situation.
std::string &find_item(std::string s) const;

// Print up to max_lines lines starting at _prev_to_current->next. If the caller
// wants to print from the beginning of the list, they should rewind() it first.
// max_lines can be a local constant = 25;
std::string to_string() const;
};
```

Let's discuss the above fragment briefly before moving on. It's possible you haven't come across the `struct` keyword before (unless you were a C programmer). A `struct` is simply a way of grouping a bunch of other variables together and giving a name to that collection. At least that's what we're using it for here.

We're defining a struct called `Node`, which contains a string member called `data` and a pointer member (a memory address) called `next`.

What does `next` point to? Its value is the memory address at which a `Node` can be found. It might even be its own address - that is, point to itself. That's the cool thing about creating linked data structures. Not only can you jump all over the place in your heap, you can also define *self-referential structures*, which is a big deal.

If you're thinking that the third line of `Node` looks like an inline constructor, you'd be right. Although I didn't mention this before, you can think of a `struct` in C++ as simply a class in which every member is public by default. So we can define a constructor for our `Node` variables that can set default values to one or both of its members.



Moving on to the `String_List` itself, note that it has three pointer members and a `_size` member. `_size` is simple. You just use it to track the current size of your linked list so you can return this value immediately instead of traversing the list to compute it each time someone asks.

Refer to Figure 1 to see how to initialize the three pointer members.

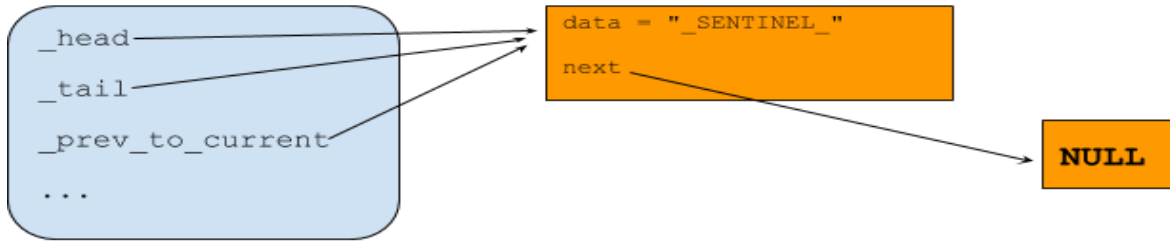


Figure 1. Pointers in an empty list

At minimum, a `String_List` should contain a single node whose data element is the string `"_SENTINEL_"`. This node performs two functions.

First, it is the node that holds the payload (content) you need to return when asked for content you cannot find. Its value should be the sentinel string.³

Second, it makes your list manipulation significantly easier. There are ways to implement linked lists without using a header node like this. But this is the easiest and simplest way to do it for minimal overhead (one extra node). I recommend that you also (in your own time) implement the complete functionality required in this quest in a linked list that doesn't use a header node. Compare the pros and cons of either approach and decide/discuss your findings in the forums.

In our approach, we are always guaranteed to find a head node. The head node is always equal to the very first actual node of the list. This is the sentinel and will never change. The user only gets to see what `_head->next` points to and downstream from there.

`_prev_to_current` is always equal to the node immediately *before* what we call the current node (by design). You can think of it as related to the *cursor* in this linked list. It always points to a node whose next element we define as *current*. All list operations are done at the location of this cursor.⁴

I think one of the best ways you'll get to appreciate the utility of this member is to try and program this quest without using it. This member saves you from having to scan the list from the beginning just to find where "here" is. If you use this member, then "here" (the current location) is always directly in front of your `_prev_to_current` node.

³ Maybe you object, saying that we're hard-coding the sentinel string here. What if the user of our list class wanted to store an actual data item whose value was `"_SENTINEL_"`? What are elegant ways to handle that situation?

⁴ In a previous version of this spec, when `p` is a pointer, I used to say "`p` points to". But that made me say things like the "header points to" and this confused questers who couldn't tell if I meant `_head->next` or `_head` itself. To avoid this confusion, I've decided to use the word *equals*.

In this spec, when I say the `_prev_to_curr` equals something, I mean that the contents of the pointer, `_prev_to_curr`, equals it. You can think of it as overloading our current understanding of the word equals with an added meaning.

This will make a whole lot more sense once you've programmed this functionality both ways. I highly recommend you do so (for your own edification).

We also maintain a tail node pointer. The tail node should always equal the node in the list whose next member is equal to NULL. That is, it is the last data node. There is a subtle interplay between the tail node and the `_prev_to_current` member that will become clear as you implement the following miniquests.



Notice an interesting thing about some of the public list manipulation methods - the ones that return a pointer to the current `String_List` object. Why?

We're doing it because it allows us to program using a very nice pattern:

```
my_list
    .push_back(string_1)
    ->push_back(string_2)
    ->push_back(string_3)
    -> . . .
;
```

In fact, it would be better if you returned a reference to the current object rather than a pointer. Then you can write the even more clean and consistent pattern:

```
my_list
    .push_back(string_1)
    .push_back(string_2)
    .push_back(string_3)
    . . .
;
```

I'll leave you to experiment with the esthetics and find out more. But in this quest, you're gonna have to return a pointer to the current object (essentially **this**).

On to the miniquests...

Your first miniquest - The Constructor and Destructor

Implement:

```
String_List::String_List();
```

This should create an empty `String_List` that matches Figure 1. So it's easy rewards. Go for it.

Since your constructor is going to allocate memory on the heap, you must have a destructor or you will suffer memory leaks. Implement

```
String_List::~~String_List();
```

I will not be testing your code for memory leaks in this quest, but your very first quest in GREEN involves an extension of this quest in which I WILL check for memory leaks. Make sure you have a destructor which clears out the linked list using a `clear` method (which you will define in one of the mini quests) and then deletes `_head` (which is what got allocated in the constructor).

Esthetics

A good rule of thumb I use when I have questions about where to deallocate memory is this: If the memory was allocated in the constructor, then it should only be deallocated in the destructor. In general, I try and match up my allocations and deallocations into symmetric locations in the causal chain leading up to some action and back.

Your second miniquest - Insert at current location

Implement:

```
String_List *String_List::insert_at_current(string s);
```

When I invoke it, I will pass it a string argument and it should insert this string at the *current location* of the `String_List` object. It should leave the current location unchanged. This means that your `_prev_to_current` member will now end up pointing to this newly inserted Node.

This is a tricky method and is worth getting correct before you move on. Refer to the picture in Figure 2 to see what needs to happen:

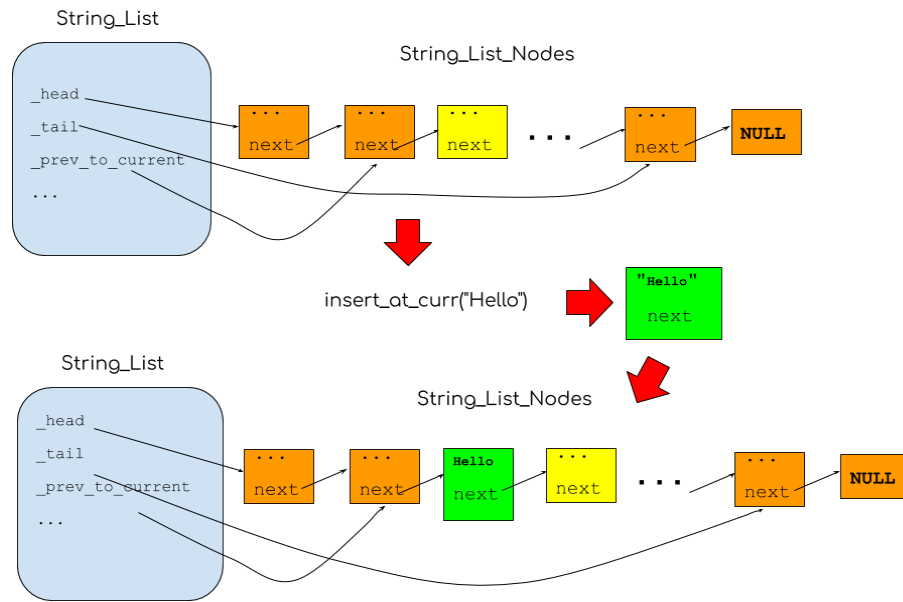


Figure 2. Insert at current location

Your third miniquest - Push back

Implement the public method: `push_back(std::string s);`

This method must insert the string `s` as a brand new data node at the end of the list. That is, this newly allocated `Node` will become the new `_tail`.

Since you have `insert_at_current()` already implemented and working now, simply use it to complete this mini quest. Save the current value of `_prev_to_current`, then set it to your `_tail`. Then insert the given string at the current position (which will now be the tail). Finally restore the value of `_prev_to_current` to your saved value.

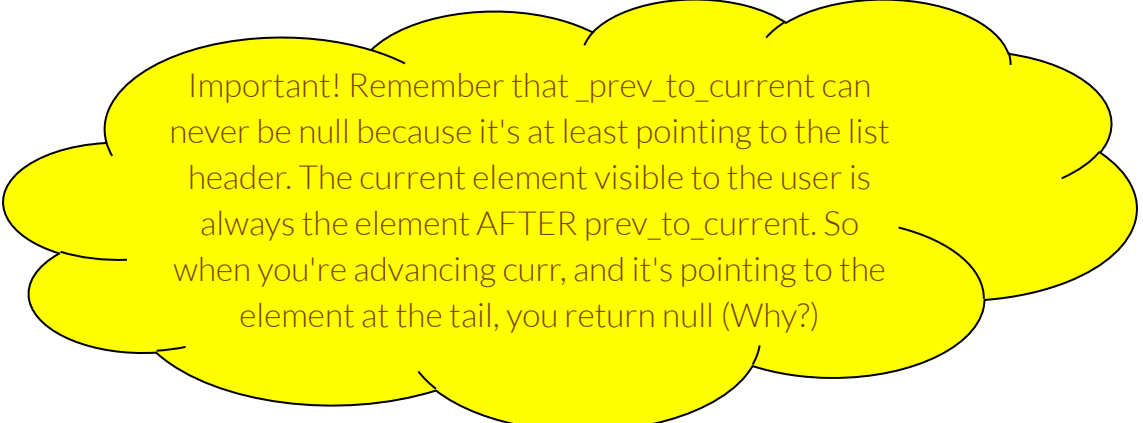
Before you implement this, convince yourself that this works by simulating the above sequence of steps on a piece of paper with a pencil.

Your fourth miniquest - Push front

Implement:

```
String_List *String_List::push_front(std::string s);
```

This method must insert the string `s` as a brand new data node at the front of the list. That is, this newly allocated `Node` will become the node that `_head->next` points to.



Important! Remember that `_prev_to_current` can never be null because it's at least pointing to the list header. The current element visible to the user is always the element AFTER `_prev_to_current`. So when you're advancing `curr`, and it's pointing to the element at the tail, you return null (Why?)

Implement it using the same strategy as for `push_back()`

Your fifth miniquest - Advance current to next

Implement:

```
String_List *String_List::advance_current();
```

If the `_prev_to_current` is the same as the tail node, then obviously you can't advance to the next element (There is none). In that case you would return a null pointer (`nullptr`). Otherwise, make `_prev_to_current` point to whatever its `next` member was pointing to.

Your sixth miniquest - Get current item

This method is only a few lines, but it is important to understand correctly.

Implement:

```
string String_List::get_current();
```

If the next element exists (that is, your cursor, `_prev_to_current`, is not `NULL`), then simply return its data member.

Otherwise, return the sentinel string you have remembered in your head. (Hmm... I wonder when that kind of thing might happen...)

Suppose I want to use your list, but one of my valid data items is the string `"_SENTINEL_"`, what would I do?

Your seventh miniquest - Remove current item

Implement:

```
String_List *String_List::remove_at_current();
```

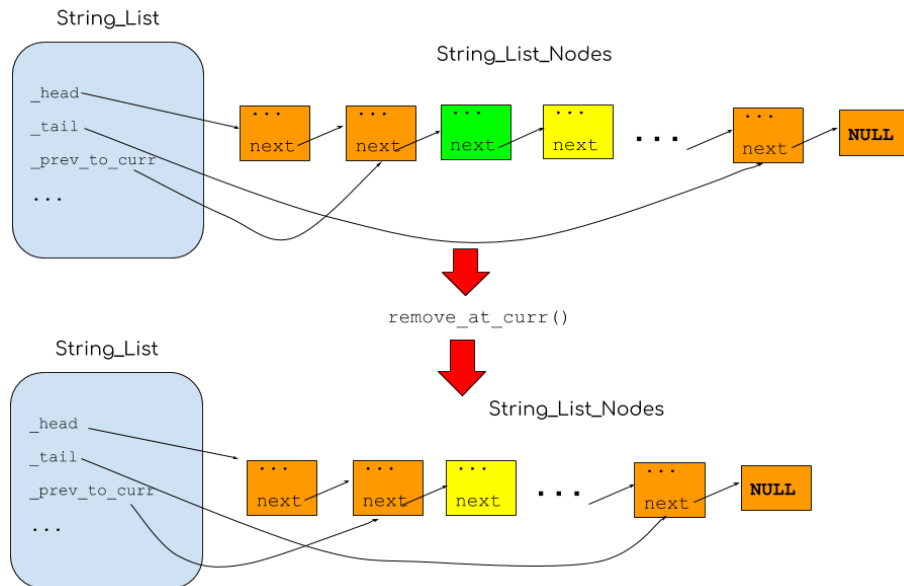


Figure 3. Remove at current location

Again, this is a tricky method and important to get right. Refer to the picture in Figure 3 to decide exactly what needs to happen in your code.

Your eighth miniquest - Get size

Implement:

```
size_t String_List::get_size() const;
```

I can't believe you're gonna get a reward for just reporting the value of size!

Still, I guess this is your last BLUE quest and it's time for a treat.

Your ninth miniquest - Rewind

Implement:

```
String_List *String_List::rewind();
```

This should reset `_prev_to_current` back to the `_head` node.

Your tenth miniquest - Clear

Implement:

```
void String_List::clear();
```

Yet another tricky method to get right. When invoked, it must

1. Iteratively (not recursively) delete all the non-null nodes in the chain starting at `_head->next` (Talking points: What might the problem be with a recursive delete instead?)
2. Reset `_prev_to_current` and `_tail` back to `_head`
3. Set the value of `_head->next` to `nullptr`

Note that you are not deleting the memory associated with the `_head` node itself. That is the job of the destructor. What the constructor created, only the destructor should undo.

Your eleventh miniquest - Find an item

Implement:

```
string& String_List::find_item(std::string s) const;
```

Note an important aspect of the signature. It returns a reference to a string. Not a copy of a string. If the requested string exists in the list, then what gets returned is a reference to the actual data element. That means that if I assign something to this reference, it will change the contents of the list node that contains that string.

It's important to understand exactly what that means. Please do discuss it in the forums and help each other out whenever possible.



What will this method return if the requested string is not found in the list? In that case, return a reference to a static string defined within this method. Its value should be `"_SENTINEL_"`.⁵ Implement it the same way you did for `get_current()`.

⁵ Note that you must account for the case when a malicious or noobie user changed the value of this mutable sentinel string to something else in a previous call. How can you do this? And why is this a problem? Hit the BLUE sub discussions to find out.

Your twelfth miniquest - Stringify

Implement:

```
string String_List::to_string() const;
```

This method must return a string representation of the list of strings in the following exact format. As usual, I've colored in the spaces. Each gray rectangle stands for exactly one space.

```
#String_List-<N>entries total. Starting at cursor:  
<string 1>  
<string 2>  
<...>
```

The parts in red above (also in angle brackets) must be replaced by you with the appropriate values. By *current elem*, I mean the element that `_prev_to_current` points to.

You would print a maximum of 25 elements this way. If the list has more than 25 strings, you must print the first 25 and then print a single line of 3 periods (`. . .`) in place of the remaining elements.

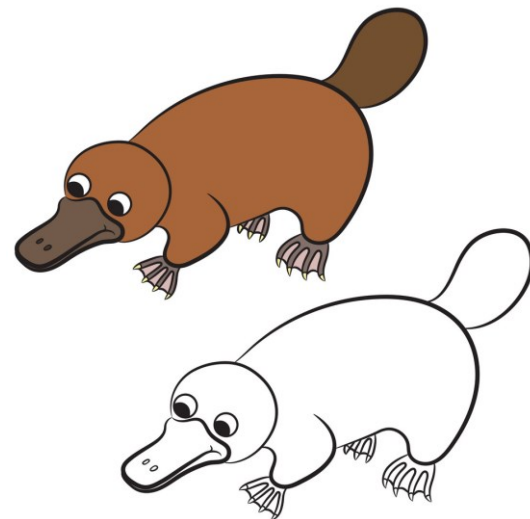
There should be one (and only one) newline after the last line (which may be ellipses).

Review

Now I recommend that you rewind and review the specs one more time, making notes along the way. It will help you when you start cutting code.

You may be thinking, "Whew! That's a lot." But really most of these are simple one-liners. If you find yourself coding more than 20 lines for any one of these methods (including comments), then you must seriously look for other signs of over-engineering in your code.

If you're up to it, consider this your real challenge (optional): The complete functionality described in this spec must be implemented in your `String_List` class in under 200 clearly readable lines including comments.



VectorStock

VectorStock.com/4872204

Starter code

You will submit one file: `String_List.h`. Here is your starter code

```
// Student ID:
// TODO - Type in your student ID after the colon above.
// String_List.h
//
#ifndef String_List_h
#define String_List_h

#include <iostream>
#include <sstream>

// Important implementation note: With the exception of to_string(),
// find*() and clear(), all list methods below should operate in a constant amount
// of time regardless of the size of the String_List instance.
//
// The semantics of _prev_to_current is such that it always points to the
// node *BEFORE* the current one. This makes the manipulations easy because
// we can only look forward (and not back) in singly linked lists.
//
// I've included some method header comments below where there's likely to
// be confusion.
//
class String_List {
private:
    struct Node {
        std::string data;
        Node *next;
        Node(std::string s = "") : data(s), next(nullptr) {}
    };

    Node *_head, *_tail, *_prev_to_current;
    size_t _size;

public:
    String_List() {
        // TODO - Your code here
    }

    ~String_List() {
        // TODO - Your code here
    }

    String_List *insert_at_current(std::string s) {
        // TODO - Your code here
    }

    String_List *push_back(std::string s) {
        // TODO - Your code here
    }

    String_List *push_front(std::string s) {
        // TODO - Your code here
    }

    String_List *advance_current() {
        // TODO - Your code here
    }

    std::string get_current() const {
        // TODO - Your code here
    }

    String_List *remove_at_current() {
        // TODO - Your code here
    }
};
```

```

}

size_t get_size() const {
    // TODO - Your code here
}

String_List *rewind() {
    // TODO - Your code here
}

void clear() {
    // TODO - Your code here
}

// Find a specific item. Does NOT change cursor.
//
// The following returns a reference to the target string if found. But what will
// you do if you didn't find the requested string? Using sentinel markers is
// one way to handle that situation. Usually there's only one copy of the
// sentinel that's global. We will use a local one so it's cleaner with a
// little more risk (what's the risk?)
//
std::string& find_item(std::string s) const {
    // TODO - Your code here
}

// Print up to max_lines lines starting at _prev_to_current->next. If the caller
// wants to print from the beginning of the list, they should rewind() it first.
//
std::string to_string() const {
    // TODO - Your code here
}

friend class Tests; // Don't remove this line
};

#endif /* String_List_h */

```

Testing your own code

You should test your methods using your own `main()` function in which you try and call your methods in many different ways and cross-check their return value against your expected results. But when you submit you must NOT submit your `main()`. I will use my own and invoke your methods in many creative ways. Hopefully you've thought of all of them.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
1. Enter the secret password for this quest in the box.
2. Drag and drop your `String_List.h` file into the button and press it. (Make sure you're not submitting a `main ()` function)
3. Wait for me to complete my tests and report back (usually a minute or less).

Need help?

You can find the BLUE discussion forum [here](#). Questers in the sub are a great resource for help (sometimes through not helping).

May the best coders win. That may just be all of you.

If everything went according to plan, this should be your last quest of BLUE. Hopefully you aced all of them and have unlocked the password for the first quest in GREEN.

Happy Questin',

&



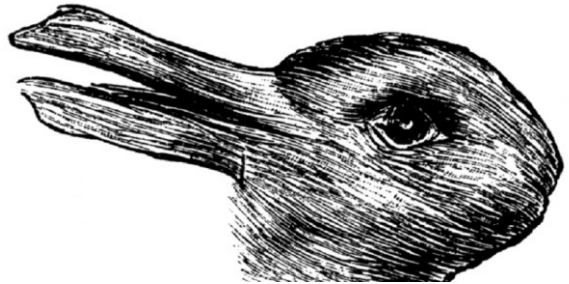
*One time I was a worrying duck.
But now I just don't give*



The duck that was...

The platypus quest - I hope you guys still remember it from BLUE. If you had aced it, this quest should be a piece of cake. Otherwise, this is your first major challenge in GREEN territory. Make sure you're solid with what it takes to solve it before proceeding to the next one.

As you solve this quest, you'll find yourself leveraging a lot of the knowledge and skills you learned in BLUE. You will also learn a few new concepts (e.g. inner classes, overloading of operators, etc.) Where necessary, I simply apply these concepts in the starter code I give you. You are free to dive into it to any level of detail and clarify them in the forums (I'll help you), or you can simply copy my starter code as-is and follow the directions in the spec for now. The skipped concepts will make more sense as you progress forward on your questing trail.



Some miniquests in this quest will give you trophies for meeting a minimum requirement, and give you additional surprise rewards for doing something else right that wasn't explicitly asked.

Overview

In this quest you will implement a class called `Playlist` as a singly linked list of nodes with `Song_Entry` objects as their payloads. Your code should be organized into a header file (`Playlist.h`) and a separate implementation file (`Playlist.cpp`).

Your `Playlist` class will encapsulate things the client (the user of the class) doesn't have to know about. Make them inner classes. An inner class is simply a class that is defined inside another class - duh! If an inner class is private, then only the containing class can create or manipulate objects of that class. If it is public, then anyone can instantiate the inner class and invoke its public methods. But they'd have to access the class name with the outer-class qualifier.

For example, if `Song_Entry` is a public inner class of `Playlist`, a user, in their `main()` method, can refer to it as `Playlist::Song_Entry`. This kind of structuring also gives a nice namespace-like separation of the type from other classes elsewhere that may have the same name.

You will have two inner classes in your `Playlist` class:

- `Playlist::Song_Entry`, which is public
- `Playlist::Node`, which is private

You don't have to modify anything in the structure (declaration) of the above inner classes. But you will have to flesh out some of the method implementations in them.

Keep the payload class, `Playlist::Song_Entry`, simple because that's not the focus of this quest. Define it as follows:

```
// This class def should be placed in the public section of Playlist

class Song_Entry {
private:
    int _id;
    string _name;

public:
    Song_Entry(int id = 0, string name = "Unnamed")
        : _id(id), _name(name) {}

    int get_id() const { return _id; }
    string get_name() const { return _name; }

    bool set_id(int id);
    bool set_name(string name);

    bool operator==(const Song_Entry& that) {
        return this->_id == that._id && this->_name == that._name;
    }
    bool operator!=(const Song_Entry& that) {
        return !(*this == that);
    }

    friend std::ostream& operator<<(ostream& os, const Song_Entry& s) {
        return os << "{ id: " << s.get_id() << ", name: " << s.get_name() << " }";
    }
    friend class Tests; // Don't remove this line
};
```

Perform some basic data validation in the setters. Don't set negative IDs or empty song names. Make the setters return true on success and false on failure.

Define the `Playlist::Node` inner class as follows:

```
// This class def should be placed in the private section of Playlist
// at an "appropriate" location. (What might be an inappropriate location?)
class Node {
private:
    Song_Entry _song;
    Node *_next;

public:
    Node(const Song_Entry& song = Song_Entry()) : _song(song), _next(nullptr) {}
    ~Node(); // Do not do recursive free

    Song_Entry& get_song() { return _song; }
    Node *get_next() { return _next; }
    Node *insert_next(Node *p);
    Node *remove_next();

    friend class Tests; // Don't remove this line
};
```

Let's discuss the preceding fragment briefly before moving on. It defines a class called `Node`, which contains a `Song_Entry` member called `_song` and a pointer member (a memory address) called `_next`.

What does `_next` point to? Its value is the memory address at which a `Node` can be found. It might even be its own address - that is, point to itself. That's the cool thing about creating linked data structures. Not only can you jump all over the place in your memory, you can also define *self-referential structures*, which is a big deal.

Important Note

In a previous version of this spec, when `p` is a pointer, I used to say "`p points to`". But that made me say things like the "header points to" and this confused questers who couldn't tell if I meant `_head->next` or `_head` itself. To avoid this confusion, I've decided to use the word *equals*.

In this document, when I say the `_prev_to_current` equals something, I mean that the contents of the pointer, `_prev_to_current`, equals it. You can think of it as *overloading* our current understanding of the word *equals* with an added meaning.

c++ lets us do cool things like this in programming too!

A few special things about `Node`

`Node::get_song()` returns a reference to its `Song_Entry` object - not a copy. Why? Because that's the only way you can easily modify the contents of an existing node in this implementation. Kudos await meaningful discussion of this point, as well as thoughts on other ways in which you might change, say, the 5th song in a list from A to B.

`Node::get_next()` simply returns the node's `_next` pointer.

Besides a `Node`'s getters and setters, note the two special methods: `insert_next()` and `remove_next()`.

You will invoke these from your `Playlist` instance methods. Since the idea of a `Node` needs to be *abstracted away* from your end-user (the programmer who uses your `Playlist` class), none of the `Playlist`'s public methods will ever communicate in terms of `Nodes` with the outside world. Instead they will interface using `Song_Entry` objects - inserting songs, removing songs, and so on. However, these methods that insert and remove songs will, opaquely to the user, themselves use their private `Node` insertion and removal methods.

`Node::insert_next(Playlist::Node *p)`

should insert the given node, `p`, directly in front of itself. That is, after the operation, its own `_next` member should be pointing at `p`. It should return `p`, the pointer to the node that just got inserted.

`Playlist::Node *remove_next()` should unlink and remove the node pointed to by `_next`. Refer to a diagram later in this document to see what needs to get done. But here is an important question before you start cutting its code. What is the node that `remove_next()` returns? Is it the node that just got unlinked so the caller can extract and use the value that it pulled out of the list?

No. It returns a pointer to itself (the `this` value). What is a good reason for doing so?

Remember that the remove operation also deletes the node it unlinks (frees its memory). So you cannot return a node which doesn't belong to you any more. But one may object, saying that it makes sense to return the unlinked node to the caller and also shift the responsibility of deleting the node to the caller. Does this alternative approach make sense? Discuss the pros and cons (including esthetic reasons) for doing it one way versus the other (kudos for insightful discussions).



After you delete a successor node for a given node, make sure to set its value to `NULL` for easy debugging. This way you're sure to know which nodes are pointing to allocated memory. (There's one more important thing to keep in mind here. Look for my picture elsewhere in this document to know what that is.)

Finally, here is the `Playlist` class. Heed the header comment

```
// Important implementation note: With the exception of to_string() and find...()
// all Playlist methods below should operate in a constant amount of time
// regardless of the size of the Playlist instance.
//
// The semantics of prev_to_current is such that it always points to the
// node *BEFORE* the cursor (current). This makes the manipulations easy because
// we can only look forward (and not back) in singly linked lists.

class Playlist {
public:
    // Inner public class -----
    // The client can refer to it by using the qualified name Playlist::Song_Entry
    class Song_Entry {
        // TODO - your code here
    };

private:
    // This is going to be our inner private class. The client doesn't need to
    // know.
    class Node {
        // TODO - your code here
    };

private:
    Node *_head, *_tail, *_prev_to_current;
    size_t _size;

public:
    Playlist();
    ~Playlist();

    size_t get_size() const { return _size; }
    Song_Entry& get_current_song() const;

    // The following return "this" on success, null on failure. See the spec
    // for why.
    Playlist *clear();
```

```

Playlist *rewind();
Playlist *push_back(const Song_Entry& s);
Playlist *push_front(const Song_Entry& s);
Playlist *insert_at_cursor(const Song_Entry& s);
Playlist *remove_at_cursor();
Playlist *advance_cursor();
Playlist *circular_advance_cursor();

// The following return the target payload (or sentinel) reference on success
Song_Entry& find_by_id(int id) const;
Song_Entry& find_by_name(string songName) const;

string to_string() const;

friend class Tests; // Don't remove this line
};

```

On returning a pointer to the list

Note that some of the public list manipulation methods return a pointer to the current list object. Why do we do that? We do it because it allows us to program using a very nice pattern:

```

my_list
    .push_back(song_1)
    ->push_back(song_2)
    ->push_back(song_3)
    -> . . .
;

```

In fact, it would be better if you returned a reference to the current object rather than a pointer. Then you can write the even nicer pattern:

```

my_list
    .push_back(song_1)
    .push_back(song_2)
    .push_back(song_3)
    . . .
;

```

I'll leave you to experiment with the esthetics and find out more. But in this quest, you're gonna have to return a pointer to the current object (essentially **this**).

On being clear where your head is

Note that you are guaranteed to find a head node in any valid list with this approach. The head node is always equal to the very first actual node of the list. It is not a data node, and we will use it to double as a sentinel (see next section).

`_prev_to_current` is always equal to the node immediately *before* what we call the current node (by design). You can think of it as the *cursor* in this linked list. It always points to (or has a next element that is equal to) the node that we define as *current*. All list operations are done at the location of this cursor.



I think one of the best ways you'll get to appreciate the utility of this member is to try and program this quest without using it. This member saves you from having to scan the list from the beginning just to find where "here" is. If you use this member, then "here" (the current location) is always directly in front of your `_prev_to_current` node. This will make a whole lot more sense once you've programmed this functionality both ways. I highly recommend you do so (for your own edification).

On sentinels

Some of the `Playlist` methods return a `Song_Entry` object. What would you do if the requested operation failed and you don't have a `Song_Entry` object to return? For example, what would you return if I invoked your `find_by_id()` and passed it a non-existent song ID?

The most common way to handle such a situation is using what are called exceptions. However, we don't cover exceptions until a week or so from now. So until then, we'll use a stopgap measure and make the method return a sentinel. A sentinel is a known invalid object.

In this case, conveniently use the header node in a linked list as a sentinel. When you create a header node, make sure the object in it has an id of -1. This is the sentinel and will never change. The user only gets to see what `_head->next` points to and downstream from there.

On to the miniquests ...

Your first miniquest - Know your songs, nodes and lists

Implement the following public methods:

- `Song_Entry::set_id()`
- `Song_Entry::set_name()`
- `Node` constructor
- `Playlist` constructor

Make sure you read about the special things to do with Nodes (earlier in this document) before you start implementing.

The `Playlist` constructor should create an empty `Playlist` that matches Figure 1. So it's easy rewards. Go for it.

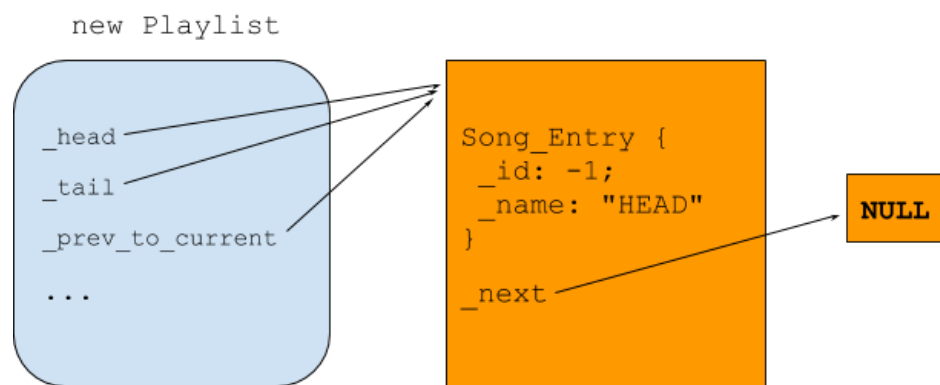


Figure 1

The head node should be a `Song_Entry` object with an id of -1. Remember that you cannot set negative ids in a `Song_Entry` object using a setter on it. The only way to do it would be at construction time (that is, when you create the head node).

Your second miniquest - Destructors

Since your `Playlist` constructor is going to allocate memory on the heap, you must have a destructor or you will suffer memory leaks. Implement

```
Playlist::~~Playlist();
```

Make sure your destructor clears out the linked list by freeing all downstream nodes first and then deleting `_head` (which got allocated in the constructor). To accomplish this you would delete `_head`, which should trigger `Playlist::~~Node`, the `Node` destructor. However, once invoked, the `Node` destructor will be called, and it should iteratively peel off one node at a time (adjacent to `_head`) and delete (free) it. When all downstream nodes have been released thus, the `Node` destructor should delete `_head` and return.

Note that there is no guarantee a memory location won't be overwritten the moment you delete the object to which it belongs. Thus, before you delete a node's descendants (starting at its `_next` member), make sure that you have taken a copy of that member into your local variable. Don't try to access it AFTER the node has been deleted. Then you will likely encounter unpredictable errors that cause your code to work sometimes and not other times.

Carelessness here has resulted in forum posts that went "I'm sure I had it working, but something must have changed on the testing site because it's not working any more."

Esthetics

A good rule of thumb I use when I have questions about where to deallocate memory is this: If the memory was allocated in the constructor, then it should only be deallocated in the destructor. In general, I try and match up my allocations and deallocations into symmetric locations in the causal chain leading up to some action and back.

Also, keep in mind that you should not delete the "this" object. It is the deletion of the current object that triggers its destructor to be called. Here, you should only delete downstream nodes.



This paragraph will likely save you several hours of debugging frustration: When you destroy a node, you will have to delete its `_next` member. This means that a node's deletion will auto-invoke the delete on its `_next` pointer and so on. You want to make sure you delete no more than you should. Therefore make sure to (1) set every pointer you delete to `nullptr` so you can check before you attempt to double-delete something and (2) clear out a Node's `_next` pointer (set it to null) before you delete the node. This will prevent the delete from cascading to affect all downstream nodes.

Your third miniquest - Insert song at current location

Implement:

```
Playlist *Playlist::insert_at_cursor(const Song_Entry& s);
```

When I invoke it, I will pass it a `Song_Entry` object as its argument. It should insert *a copy* of that object at the *current location* of the `Playlist` object. It should leave the current location unchanged. This means that your `_prev_to_current` member will now end up having a `_next` member pointing to this newly inserted Node.

This is a tricky method and is worth getting correct before you move on. Refer to the picture in Figure 2 to see what needs to happen:

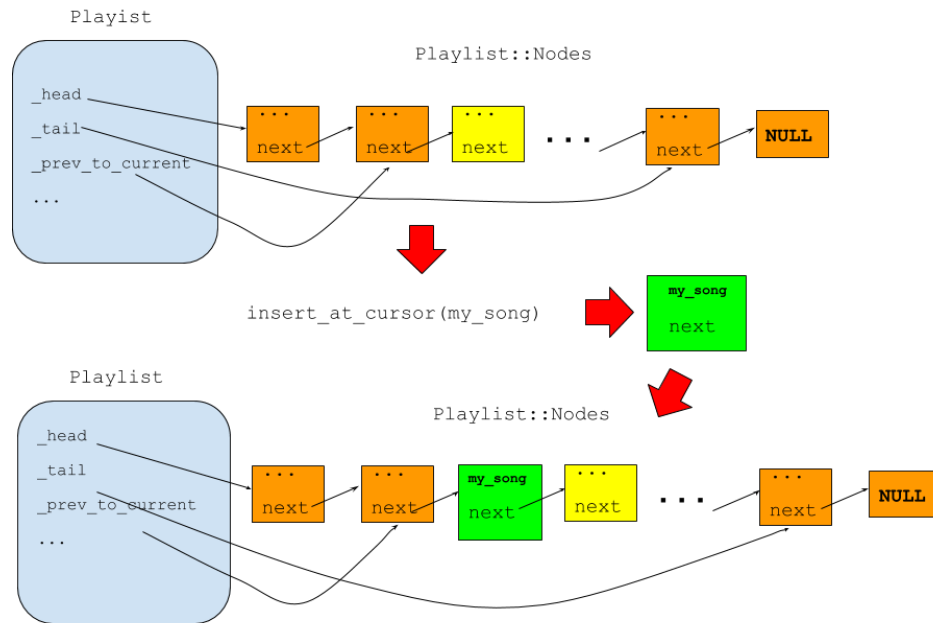


Figure 2 - insert at cursor

Your fourth miniquest - Push back

Implement:

```
Playlist *push_back(const Song_Entry& s);
```

This method must insert (a copy of) the song `s` as a brand new node at the end of the list. That is, this newly allocated `Node` will become the new `_tail`.

Since you have already implemented `insert_at_cursor()`, simply use it to complete this mini quest.

1. Save the current value of `_prev_to_current`, then set it to your `_tail`.
2. Then insert the given string at the current position (which will now be the tail).
3. Finally restore the value of `_prev_to_current` to your saved value.

Before you implement this, convince yourself that this works by simulating the above sequence of steps on a piece of paper with a pencil.

Your fifth miniquest - Push front

Implement:

```
Playlist *Playlist::push_front(const Song_Entry& s);
```

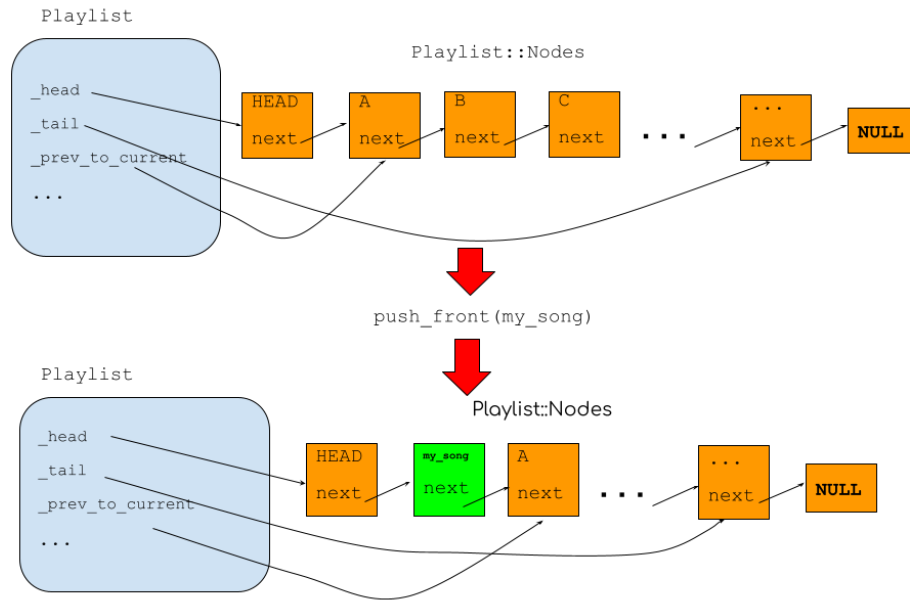


Figure 3 - push front

This method must insert the song `s` as a brand new data node at the front of the list. That is, this newly allocated `Node` will become the node that `_head->next` points to.

Implement it using the same strategy as for `push_back()`

Your sixth miniquest - Advance

Implement:

```
Playlist *Playlist::advance_cursor();
```

If `_prev_to_current` is the same as the tail node, then obviously you can't advance to the next element (there is none). In that case you would return a null pointer (`nullptr`). Otherwise, make `_prev_to_current` point to whatever its `_next` member was pointing to.

Important! Remember that `_prev_to_current` can never be null because it's at least pointing to the list header. The current element visible to the user is always the element AFTER `_prev_to_current`. So, when you're trying to advance the cursor and it's pointing to the element before `_tail`, you return null (Why?)

Your seventh miniquest - Circular Advance

Implement:

```
Playlist *Playlist::circular_advance_cursor();
```

This is essentially the same as `advance_cursor()` EXCEPT for the fact that if the cursor is pointing to the very last data node, the advance will silently reset to point to the first node. Refer to the picture below to see what needs to happen:

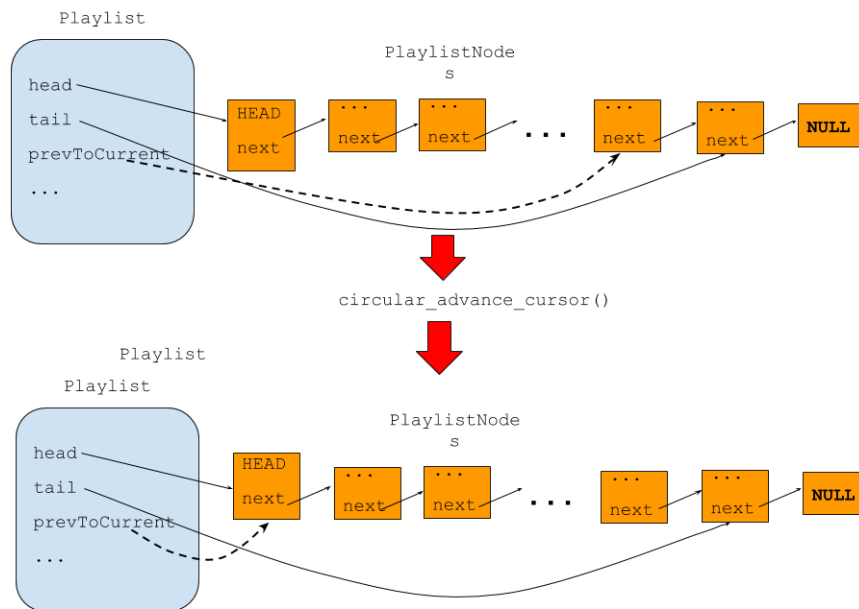


Figure 4 - circular advance

Your eighth miniquest - Get current song

Implement:

```
SongEntry Playlist::get_current_song();
```

If a `_next` element exists, then simply return its `_song` member.

If no next element exists (this could happen if `_prev_to_current` is the same as your tail node), then you must return the sentinel (`this->_head->_song`)

This method is only a few lines, but make sure you understand it fully. Ask in the forums if any of it is unclear.

The current item is the song member of whichever node is equal to `_prev_to_current->_next`.

If `_prev_to_current->_next == nullptr`, which should not happen, we would ideally throw an exception. But since we haven't covered exceptions yet, we will return the sentinel.

Your ninth miniquest - Remove song at cursor

Implement:

```
Playlist *Playlist::remove_at_cursor();
```

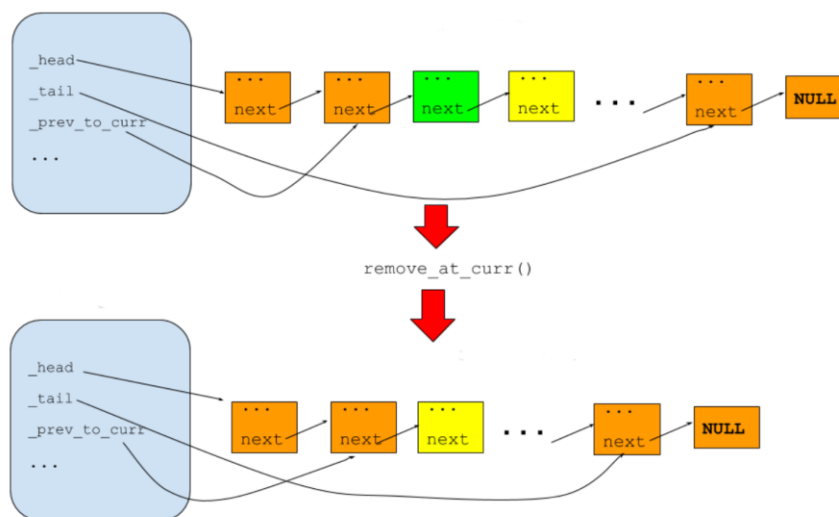


Figure 5 - remove at current position

Again, this is a tricky method and important to get right. Refer to the picture in Figure 3 to decide exactly what needs to happen in your code.

Your tenth miniquest - Get size

Implement:

```
size_t Playlist::get_size() const;
```

I can't believe you're gonna get a reward for just reporting the value of size!

Still, I guess this is your first GREEN quest and you gotta have some freebies...

Your eleventh miniquest - Rewind (but not relax just yet)

Implement:

```
Playlist *Playlist::rewind();
```

This should reset `_prev_to_current` back to the `_head` node.

Your twelfth miniquest - Clear

Implement:

```
Playlist *Playlist::clear();
```

Yet another tricky method to get right. When invoked, it must

1. Iteratively (not recursively) delete all the non-null nodes in the chain starting at `_head->_next` (Talking points: What might the problem be with a recursive delete instead?) Consider using the `Node` destructor you wrote.
2. Reset `_prev_to_current` and `_tail` back to `_head`
3. Set the value of `_head->next` to `nullptr`



Note that you are not deleting the memory associated with the `_head` node itself. That is the job of the destructor. What the constructor created, only the destructor should undo.

Your thirteenth miniquest - Find an item

Implement the following *linear-search* methods:

```
Song_Entry &Playlist::find_by_id(int id) const;
```

```
Song_Entry &Playlist::find_by_name(string id) const;
```

Note an important aspect of the signature. They return references to `Song_Entry` objects. Not copies. If the requested song exists in the list, then what gets returned is a reference to the actual data element. That means that if I assign something to this reference, it will change the contents of the list node that contains that song.

It's important to understand exactly what that means. Please do discuss it in the forums and help each other out whenever possible.

What will this method return if the requested song is not found in the list? In that case, return a sentinel in the same way you did for `get_current_song()`.

Your fourteenth miniquest - Stringify

Implement:

```
string Playlist::to_string() const;
```

This method must return a string representation of the list of strings in the following exact format. I've colored the spaces. Each colored rectangle stands for exactly one space.

```
Playlist: [N] entries.  
{ [id]: [id1], name: [Name of 1st song] }  
{ [id]: [id2], name: [Name of 2nd song] }  
.  
.  
.  
{ [id]: [idk], name: [Name of kth song] } [P]  
.  
.  
.  
{ [id]: [idN], name: [Name of last song] } [T]
```

The parts in red above (also in square brackets) must be replaced by you with the appropriate values.

The `_prev_to_current` element, *if visible*, should be marked with a `[P]` tag (see above).

Similarly The very last element (the tail), if visible, should be marked with a `[T]` tag.

You would print a maximum of 25 elements this way. If the list has more than 25 strings, you must print the first 25 and then print a single line of ellipses (`...`) in place of the remaining elements.

There is one newline after the last line (which may be ellipses).

Here's a point to ponder and discuss. Why do we need to say "*if visible*" when talking about the cursor above?

Review

Now I recommend that you rewind and review the specs one more time, making notes along the way. It will help you when you start cutting code.

Starter code

There is no starter code in this quest. You're on your own in the wildies (actually, that's not true, if you've been reading along).

Testing your own code

You should test your methods using your own `main()` function in which you try and call your methods in many different ways and cross-check their return value against your expected results. But when you submit you must NOT submit your `main()`. I will use my own and invoke your methods in many creative ways. Hopefully you've thought of all of them.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your `Playlist.h` and `Playlist.cpp` files into the button and press it. (Make sure you're not submitting a `main()` function)
4. Wait for me to complete my tests and report back (usually a minute or less).



Need help?

You can find the GREEN discussion forum [here](#). Questers in the sub are a great resource for help (sometimes through not helping).

May the best coders win. That may just be all of you.

Happy Questin',

&



*Excusez-moi. I don't mean to annoy
But how do you goa from hare to hanoi?*



... from hare to Hanoi?

In this quest you get to take recursion to the next level. Think about recursion, Fibonacci, memoization and dynamic programming. That's what you're gonna do in this relatively easy quest.

Well, almost. Except the dynamic programming part. But I will be happy to see a thoughtful discussion in the forums on exactly what the trade-offs are between memoized recursion and DP, how a table-driven approach may work, and whether such an approach gives any real wins. And why, or why not.



<https://quests.nonlinearmedia.org/genius/demos/hanoi>

Overview

The problem statement is this: You have N discs of decreasing radius and three poles⁶ on which these discs could be slotted. You can only move one disc at a time, and you cannot place a bigger disc on a smaller one. Suppose all N discs are on pole 1, what is the sequence of moves you need to transfer all discs to pole 2? Assume that you can use the third pole, 3, as your temporary holding space. You can use it to store any number of discs, as long as you don't violate the disc ordering property (a bigger disc may not rest on a smaller one).

You are welcome to try out the recursive (but not memoized) Javascript demo on my home page. Work it out on a piece of paper for 1 disc, 2 discs, and 3 discs. Then work out a general technique for N discs. For example, suppose I denote the action of moving the top disc on pole 1 to pole 2 (if valid) by the string "1->2". Then I might say:

1. To move 1 disc from 1 to 2, I need to do: 1->2
2. To move 2 discs from 1 to 2, I need to do: 1->3, 1->2, 3->2
3. To move 3 discs from 1 to 2, I need to do: etc.
4. ...
5. To move N discs from 1 to 2, I need to:...



Your challenge is to discover the pattern in the sequence above and code it up, so I can ask your program to give me the move sequence for any number of discs. Of course, I'll keep the number reasonable. But I don't know what that value is right now.

There are two parts to this quest. The first part is easy. It's just straight recursion. For the second part, you're going to have to memoize the first part. Here is the mostly-filled-in `Hanoi` class you need to put in your `Hanoi.h`:

⁶ I don't remember why I used the word *poles*. Maybe I was thinking of slotted discs in some actual puzzle I had seen. We'll continue to use that word, but we just need 3 places to rest the discs - that's all.

```

class Hanoi {
private:
    // TODO: Declare the _cache member using an appropriate
    // level of nesting within std::vectors to put each string
    // of moves. You should be able to access the cache like so:
    // _cache[num_discs][src][dst] = "move1\nmove2\n..."

    std::string lookup_moves(int num_discs, int src, int dst) const;
    std::string get_moves(int num_discs, int src, int dst, int tmp);

public:
    // Use freebie default constructor
    std::string solve(int num_discs, int src, int dst, int tmp);

    friend class Tests; // Don't remove this line
};

```

You need to supply the implementation file, `Hanoi.cpp`, yourself. The following miniquests will shed more light on the individual methods.

Your first miniquest - Basis Cases

All you need to do to pass this miniquest is to return the right responses for your basis cases. These are the situations where you return a result immediately without recursing deeper. You can be defensive and define two basis catches: for `num_discs = 1` and `num_discs = 0`.

For each case, you can imagine 6 different puzzles - yes? For example, my problem might be to move 1 disc from pole 1 to pole 2. Or it might be from pole 3 to pole 1. How many such cases can you imagine? Your `get_moves()` method must return the sequence of required moves for each case as a series of newline separated strings of the form `A->B`. The arrow is an ASCII hyphen followed by a greater-than sign. Don't copy and paste these symbols as your output will not be processed before being checked.

Of course, you won't have more than one move in the basis cases. So there is only ever a maximum of one newline character at the end (none for 0). But in general, you can expect to return a sequence of moves. For example, to move 2 discs from 1 to 2, you would return the string `"1->3\n1->2\n3->2\n"`

Your second miniquest - Get moves

Make sure your `get_moves()` method is now able to handle input for cases other than the basis case. E.g. you may want to find the sequence of moves to transfer 12 discs from pole 2 to pole 1.

That's all. If your recursion is clean and tight, you'll find your method is no longer than 5-10 lines of code. However, remember that you haven't built in memoization yet (you can expect the method to grow to about 15-20 lines after it's been memoized).



Your third miniquest - Solve

A piece of cake. This method should simply return the result of a `get_moves()` call prefixed with the following line:

```
# Below, 'A->B' means 'move the top disc on pole A to pole B'
```

Note that there are no spaces before or after the first and last visible characters. The quotes used in the string are plain ASCII single quotes. Don't use curly or other fancy quotes which might creep in if you copy/paste from non-plain-text sources.

Your fourth miniquest - Memoize

The real challenge in this miniquest is in implementing and managing the data structure for the cache correctly. The abstract representation of the cache should look like in Figure 1.

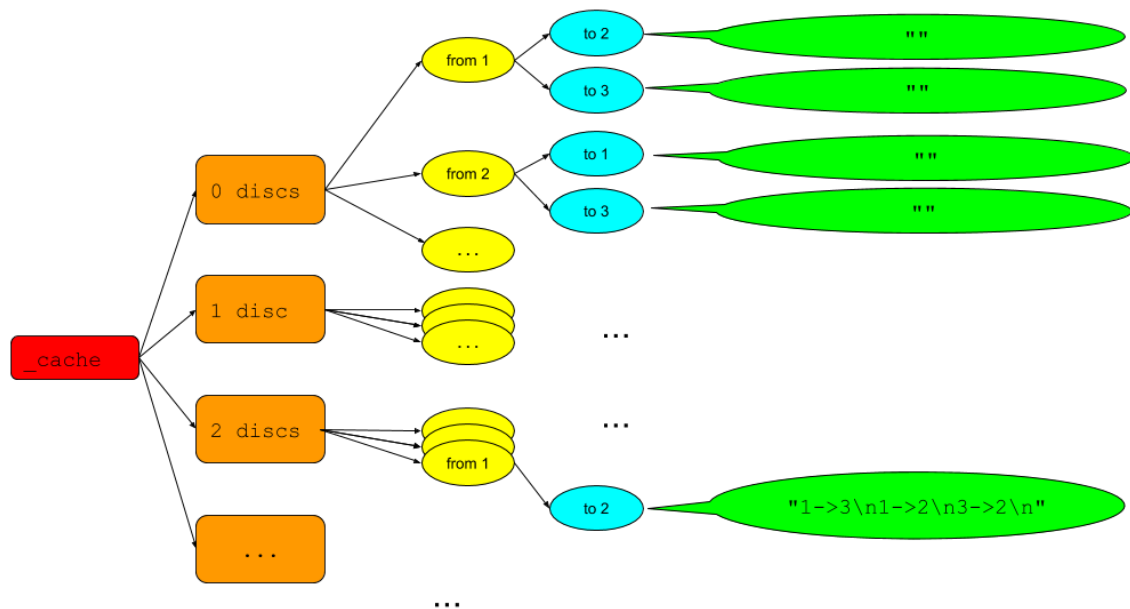


Figure 1: Cache of Hanoi

To get this functionality, implement the cache as a `vector` of `vector` of `vector` of `strings`, using Figure 1 as a guide in understanding the semantics of the structure. For example, the second index, `j`, in the sequence of nested indexes in `_cache[n][j][k]` stands for the source pole number to solve a puzzle with `n` discs.

Note that the above pole numbers span values 1, 2 and 3, but your cache is a `vector` whose elements start at `n=0`. If you look at our implementation, it doesn't really matter what these numbers are - just that they are different. However, there is a relatively tiny bit of wastage you will incur if you use large valued pole numbers (e.g. 100, 150, 200). Exactly how much wastage? Why do you think we can consider it tiny? relatively? Or can we? (Discuss)

Important: Use pole labels 1, 2 and 3 (not 0, 1 and 2 or any other). This means that `_cache[n][0]` and `_cache[n][j][0]` are *dead locations* you won't use in your program. But the savings you get in reduced cognitive load (from a mismatch between emitted pole numbers in the output and vector indices) will make it a good deal in this quest. Or does it?

In order to ace this miniquest, you need to do the following correctly. It's all or nothing, I'm afraid:

1. Declare the `_cache` object in your Hanoi class
2. Implement the method `lookup_moves()` which should
 - a. look up the cache for a given triplet (`num_discs`, `source`, `target`)
 - b. Return the cached string if it exists
 - c. Else return the empty string `""`

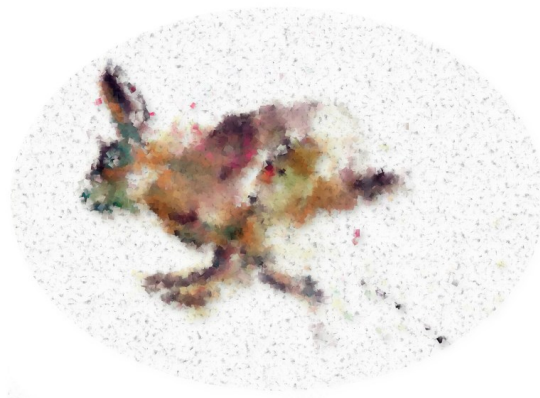
3. Edit the method `get_moves()`, which should return the string of moves required to transfer `num_discs` discs from the `src` pole to the `dst` pole using the `tmp` pole as a holding area. I might invoke it like: `your_hanoi.get_moves(5, 2, 3, 1)` to get the moves to transfer 5 discs from pole 2 to pole 3 using pole 1 for holding.
4. You should not hold on to any `_cache` entry longer than needed for a 1-time calculation with a given number of discs (think like you need to calculate `fibonacci(n)`) - You will have to call `vector::clear()` on some levels as you descend. Which ones?
5. You should only create `_cache` nodes lazily. That is, don't create `_cache[i][j]` until such time as you actually have something to put in it.

How should you change `get_moves()`?

- When a request for a move sequence arrives, check the cache first using the lookup method.
- If a cache entry exists, return it.
- Otherwise, calculate the sequence of moves as before, but store it in the cache before returning it to the user.
- **Seriously** think about whether you need to always carry your whole cache with you.

This is a tricky miniquest to get right and it is worth a significant number of trophies. But a correct implementation here will give you solid insight into how to make a table-driven algorithm that does the same thing as memoized recursion (except with no recursion at all). I'm not testing your table-driven version. But kudos await interesting thoughts shared on this topic.

Other questions and worthy discussion topics are welcome too. Here's something for starters: How long does a cache entry need to live? How expensive is it to maintain?



Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret code for this quest in the box.
3. Drag and drop your Hanoi . * files into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).

Happy Questin',

&



Now, ya press da button
doc.

*Ey! My et'unda mynah
Engadi 1-ode nainah*



My et'unda mynah...

Automata

previously by Michael Locuff

If you haven't read it yet, read the text, check the internet or refer to the reference material for discussions of *Cellular Automata*.

Provided you understand what Cellular Automata are, you'll be clear on what we're going to try and do in this quest.

To help you understand the topic better, here's a slightly different (but related) take on the subject from what you may find

elsewhere. Read them all and one might make more sense than it did before. Or just read this spec (it's sufficient).

In this quest, you will implement your own copy of the one dimensional CA you can find at <https://quests.nonlinearmedia.org/genius/demos/ca1d>

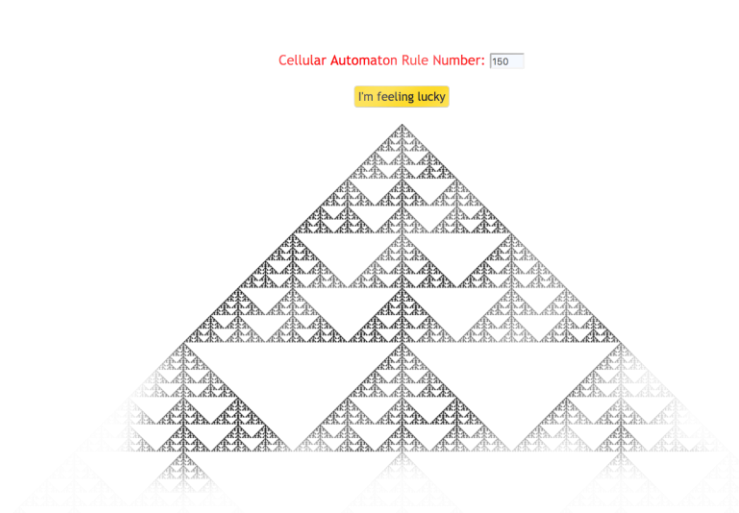
Let's define an *automaton* to be a deterministic (aka programmatic) process by which a given string of bits is transformed into another string of bits. So you can simply consider the automaton to be a collection of rewrite rules. It's like a mathematical function $y = \text{Automaton}(x)$ where $\text{Automaton}(x)$ is a function of the variable x , our bit string.

Let's say that a *generation* is an infinite bit string. If you feed it to the automaton, it will return to you another infinite bit string, your *next generation*.

In making the next generation string, the automaton will deterministically produce each bit by mapping a particular sequence of bits in the current generation into this single bit in the next generation.

Obviously, the possibilities here are endless - you can inject all kinds of variations into this mapping process - even external factors like time of day or cosmic ray intensity. But we want to keep it simple (almost trivial) to see if anything interesting emerges with the least amount of external input). That's why we're restricting our attention to automata that only consider a small set of *parent* bits when deciding what the next generation *child* bits should be.

In fact, we consider only possibilities where the value of a bit in the next generation (child) depends on some *contiguous* number of bits in the current generation (parents).



What this means is that 1 or more contiguous parents in an infinite string of bits *determine* each child in the next generation - another infinite string of bits.

You can imagine a sliding window of N bits that moves over the current generation bits to generate the next generation. See Figure 1.

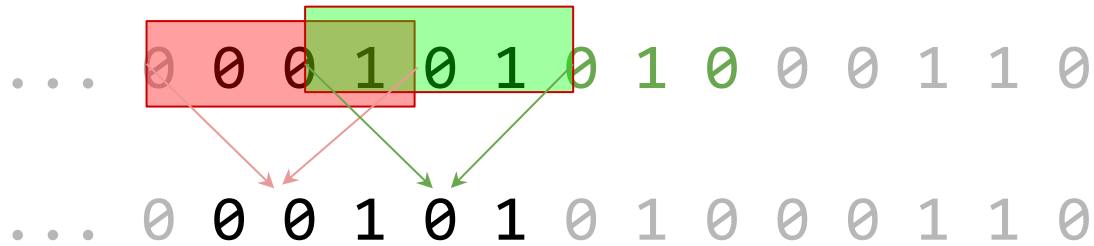


Figure 1:How a 5-parent Automaton stitches together the next generation (bottom) from the current (top)

Now, let's go one step further and exclude more automata. Let's ignore automata that have non-odd or negative numbers of parents⁷. That means we're saying "*Maybe there are interesting rule sets for those parent combinations too, but we'll save that for another day.*" Read Robert Frost, maybe.

This is the small subset of automata you'll be building in this quest. You will implement an `Automaton` class, which I can use to stamp out my own `Automaton` objects.

The way the automaton should work is by turning one *generation* of bits into the next. If I invoke it iteratively by feeding its own output back to it, I should be able to generate successive generations of bit strings.

But how do I get started? Let's assume that the very first generation, which I call the *seed generation*, is a single 1 (*the seed bit*) in an infinite sea of all 0s. By restricting our attention to this seed bit and a finite *interesting portion* around this seed that may grow with each successive generation, we can get both easier programming, and aesthetically pleasing outcomes.

Since every bit in the next generation will be based on an odd number of parent bits in the current generation, we'll say that it is the child of a sequence of parent bits which are centered around this child bit (if the two generations, like infinite ticker tapes, were lined up with corresponding bits under each other (as in Figure 1).

Now consider the interesting case when an automaton considers exactly one parent bit for each child bit. Then it needs two rules in its inventory:

1. value (1 or 0) to use if the parent bit is 1
2. value (1 or 0) to use if the parent bit is 0

⁷ Actually the miniquests will only check 1 and 3 parent versions. Kudos await insightful posts regarding 5-parent Automata.

Since the two values (read bottom to top in the above ordered sequence) themselves form a 2-bit number, you could kinda say that your 1-parent Automaton is characterized simply by one of the four possible 2-bit patterns: 00, 01, 10 or 11. See Table 1.

parent	child
0	p
1	q

Table 1. The generic 1-parent automaton. p and q are binary. Each "pq" defines a different 1-parent automaton

If p and q are both 0 (00), you get a particular kind of 1-parent automaton (one that always gives you all 0s no matter what the parent is). Similarly, if "pq" is "11", you get another automaton which always gives you all 1s no matter what. How many such distinct 1-parent automata can there be?

It is simply the number of distinct 2-bit strings, the values of p and q . Table 2 shows all four of them:

Parent	Auto-A emits this child (nullifier?)	Auto-B emits this child (copier?)	Auto-C emits this child (inverter?)	Auto-D emits this child (fullifier?)
0	0	0	1	1
1	0	1	0	1

Table 2. The 4 possible 1-parent automata

Obviously there can't be any other 1-parent automata than the above. But if you get 4 possible automata with just 1 parent, you can imagine how many automata you'd have with 3 or more parents. Try this (on paper) before reading on.

3 parents can come in $2^3 = 8$ combinations. (Convince yourself: 000, 001, 010, etc.) Each of those combinations can result in a child that can either be a 1 or a 0. This automaton will be a program that translates 3-bit binary strings into single bits. Its rules will consist of a table of 8 rows, which maps every possible 3-parent combination into a separate child. Reading off the child values in this table from bottom to top we'd get an 8-bit binary string. There are a total of 256 possible 8-bit binary strings (Convince yourself: 00000000, 00000001, etc.) Thus there are 256 possible 3-parent automata.

Oh, wait. We named our 1-parent automata A, B, C and D. That's not gonna work here 'cuz we have way more than Z automata. What might be a better naming strategy then? Here's one: We'll just call the automata by the n-bit number formed by reading off the child bits in reverse. That means we would have Automata 0, 1, 2, and 3 for the 1-parent case, and Automata 0 through 255 for the 3 parent case.

Care to extrapolate for more parent cases?

Implementation detail

How do you print out infinite bit strings on finite media in finite time? Well, actually, you can't.

But we'll introduce a clever abstraction called an `_extreme_bit` which stands for an infinite sequence of the same bit. Initially, we'll say that `_extreme_bit` is 0, and our `current_generation` is simply all `_extreme_bits`. Then we'll drop a seed in this *infinite ocean of extreme bits*, a 1.⁸

Then at each generation, we'll find that we just have three chunks of the infinite current generation to process, and we can process them all in finite time.

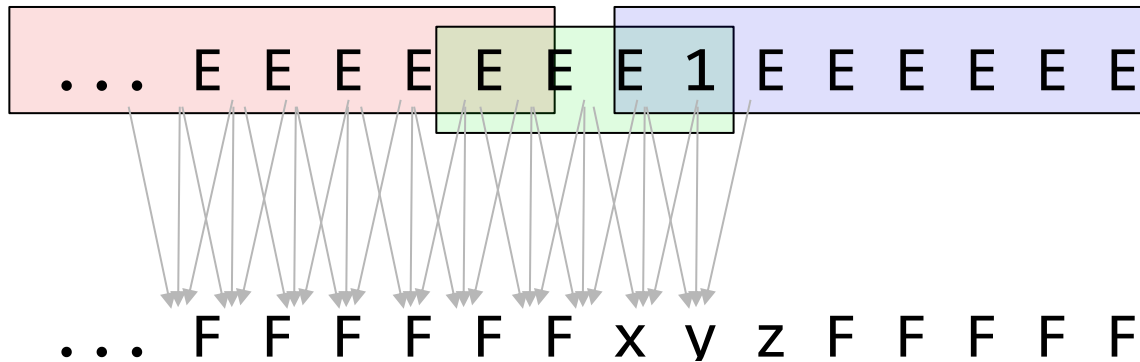


Figure 2: This 3-parent automaton is able to process an infinite bit stream in finite time by processing it in 3 chunks.

The first chunk processed by the automaton in Figure 2 (the reddish rectangle to the left) consists entirely of extreme bits (E) in the current generation (which is internally stored just as a single bit proxy in the boolean variable `_extreme_bit`). The value of the extreme bit for the next generation (F) is simply what the automaton maps 3 consecutive extreme bits (EEE) to.

The second chunk (the greenish rectangle in the middle) consists of a window that moves through the non-extreme bits. In this example, 1 is the only non-extreme bit. And so, the automaton produces values x, y and z, which are values mapped from EE1, E1E and 1EE by the automaton.

Then the next generation can now be represented by just a 3 character string "xyz" and a single value for the new extreme bit, F.

To find the generation after that, use this next generation as the new current one. This is the way to propagate the seed through multiple generations. At each generation, the *interesting part* of the bit string, the part around the seed trimmed of extreme bits, grows just by a few bits on each side (1 for the 3-parent automaton, 2 for the 5 parent automaton, and so on. How many padding bits should I use for an N-parent automaton?)

When it comes time to print a generation, we will use a similar approach. That is, we will simply print its *interesting portion* padded by an equal number of extreme bits on either side for a given display width.

⁸ Here is an interesting forum discussion starter: What kind of infinite strings can you NOT represent on finite media using the extreme-bit abstraction?



Hmm... I wonder which rule this snail used...

Overview

I managed to get a sneak peek at the header file and capture a snap. Unfortunately parts of it got obscured. But I figured something is better than nothing. And you guys can easily recreate the missing bits.

```
Automaton {
public:
    static const size_t MAX_PARENTS = 5;

    bool _is_valid;
    size_t _num_parents; // 2^{_num_parents} = possible parent combos

    vector<bool> _rules; // size() = 2^{_num_parents} rules. One for each parent combo
    int _extreme_bit;

    Automaton(size_t num_parents, size_t rule);
    void set_rule(size_t rule);
    bool equals(const Automaton& that);

    void make_next_gen(const vector<int>& current_gen, vector<int>& next_gen);
    void get_first_n_generations(size_t n = 100, size_t width = 101);

    / instance utility
    string generation_to_string(const vector<int>& gen, size_t width);

    / static utilities
    static size_t pow_2(size_t n) { return 1 << n; }
    static size_t translate_n_bits_starting_at(const vector<int>& bits, size_t pos, size_t

friend class Tests; // Don't remove this line
```

Figure 1: A fuzzy photo of the Automaton class def

Here is some helpful information to go with it:

- Save yourself by setting a hard-limit. `MAX_PARENTS` is your friend. I'll only test cases for 1 and 3 parents, though you're free to try 5 on your own.
- `_is_valid` is a private internal boolean that you use to tell if your current automaton is valid. The idea is that you would set this flag on Automata that were created using rules or parent counts your other methods choose not to handle. They can simply check the value of this flag instead of querying a bunch of variables to see if they'll succeed.

- `_num_parents` is the number of parents this Automaton considers (1, 3 or 5)
- `_rules` is a vector of booleans. It should have as many elements as the number of possible parent combinations (e.g. 8 for 3-parent automata). This is where you will look up what the child should be given the values of its N parent bits.

Also, note that the *better thing to do* is to package the extreme bit with a generation, not the automaton. However, it got done the other way here for historical reasons and I figured it's better to leave it like it is as an example of *improvable structuring*. Check our [sub](#).

And here is more detail in the following mini-quest specs.

Your first miniquest - Utilities

Just like in Monopoly.

Get these right and you'll have an ongoing sense of security that will let you focus on the things that matter in this quest.

Implement `pow_2` inline within the header file to return a power of two as the result of a bit shift.

Note that $2^0 = 1$. If you left-shift a 1 one time, you get 2 (in binary) and so on. Return the value got by left-shifting `n` times, where `n` is the exponent. Don't invoke the Math function `pow()` to calculate small integer powers of 2 if it's a calculation that will be done many times (as in our case).

Implement (complete the code between the braces in the header file, in one line):

```
size_t Automaton::pow_2(size_t n);
```

You can expect `n` to be small (< 64) in my tests.

Now implement:

```
size_t Automaton::translate_n_bits_starting_at(
    const vector<int>& bits, size_t pos, size_t n
);
```

It should interpret the `n`-bit sub-vector of `bits`, specifically `bits[pos]` to `bits[pos + n - 1]` as the digits of a binary number, and return the corresponding numerical quantity. `pos` starts at 0 for the first element.

For example, if I invoke it on the bit string "101010110101" to translate the 5 bits from pos 2, it should return the quantity 0b10101, which is 21 in decimal.

Make sure you check this method for correct behavior at the edges: It must return 0 if `n` bits starting at `pos` extend past the end of `bits`.

Your second miniquest - Generation to string

There will be a time when you want to print out a generation to the console.

Implement:

```
string generation_to_string(const vector<int>& gen,
                           size_t width);
```

It should convert the contents of the given `vector` into a sequence of binary values, and stitch together a string of these bits, with 0 represented by space and 1 represented by an asterisk (*).

Further, this string should be centered within a larger string of the given width. Since the string's length will always be odd (why?) it can only be centered within the given width if the width is also odd. Therefore this method should refuse to stringify a generation into a field of even width. If width is even, or the automaton is invalid, simply return the empty string.

If it is odd, then you can easily calculate by how much to pad (or clip) the generation's bits on either side to get a string of the required width. The question now becomes: *"With what value should we pad the generation? 0 or 1?"*

Recall that this is precisely where the extreme bit comes in. In fact, it is also the reason I had to make this method a stateful instance method rather than a stateless static helper, which I would have preferred. Can you think of a good way for me to have my cake and eat it too?



Your third miniquest - Set Rule

Implement:

```
bool Automaton::set_rule(size_t rule);
```

You can assume that when this method is invoked, the `_num_parents` member of the current object will be set. However, you need to check its value. If it is greater than permitted by the `MAX_PAREMITS` value in the spec, you should return `false` immediately.

Furthermore, you should also return `false` if the value is greater than the maximum permitted by the `_num_parents` value (what might that be?). These cases result in invalid automata.

Otherwise, you must fill in the `_rules` vector of the current object with the appropriate values for the given rule and return `true`. You can also assume that the vector would already have been sized correctly in the constructor (when the number of parents was set).

Before returning, reset the `_is_valid` flag to `true` if you have succeeded in setting this rule correctly.

Your fourth miniquest - Constructor

This one is going to be easy if you've got `set_rule()` right.

Implement:

```
Automaton::Automaton(size_t num_parents, size_t rule);
```

Since we aren't throwing exceptions when the constructor fails, we'll do the next best thing. Remember how every Automaton has a boolean member called `_is_valid`? If either the requested number of parents is illegal or the rule failed to set (you can invoke `set_rule()` from your constructor), you must set `_is_valid` to `false` and return immediately.

Your fifth miniquest - Equals

Implement:

```
bool Automaton::equals(const Automaton& that);
```



It should return `true` if the `*this` Automaton is equal to `that`. Equality is defined thus: Two Automata are equal if either of the following conditions is true. They are unequal otherwise.

1. They are both invalid
2. They are both valid AND have the same number of parents, the same extreme bit and the same rules.

Your sixth miniquest - Make next gen

This is the biggie. Implement:

```
bool Automaton::make_next_gen(const vector<int> &current_gen,  
                             vector<int> &next_gen);
```

This method should accept a `const` vector of ints (by reference), which it will use as the current generation. Then, using the rules in the current automaton (`this`), it should proceed to stitch together the next generation in place (in `next_gen`).

Interpret the vector as the *interesting portion* of the infinite string of bits of the current generation. Likewise, your returned `next_gen` should also be just the interesting portion. Make sure that you update the value of the extreme bit in the automaton after processing a generation.

Importantly, this method must return false if either the automaton it is invoked on is marked invalid, or if the length of the current_gen is an even number, with the exception of 0.

If the length of the current generation is 0, make the next generation the *seed*, which is a vector with a single element = 1. That is, { 1 }.

Here is an interesting forum discussion starter for you:

It is obviously possible to maintain state within the Automaton by storing the *current generation* as a member. In this implementation, I decided to reduce the *statefulness* of this class and move the generation into the hands of the client (the code that constructs an Automaton).

Did I achieve my goal of statelessness? If I didn't, discuss what more is left to do and some elegant ways to do it.



Your seventh miniquest - Get first n gens

Implement

```
string Automaton::get_first_n_generations(size_t n, size_t width);
```

It should return a string representation of the automaton. *n* is the number of generations to draw and *width* is the number of characters to use for each line of output. Return the empty string for even *width* (including 0) or invalid automatons.

Use newlines in this string to break it up into a number of separate equal width lines. For example, the string

```
"      *      \n     ***      \n    * * *      \n   ** * **      \n  *      *      \n"
```

represents the first 5 generations (0-4) of an Automaton(150), drawn on a canvas 9 characters wide:



How long is a well-written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in about 250 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret code for this quest in the box.
3. Drag and drop your source files (excluding your main function) into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).



Happy Questin',

&

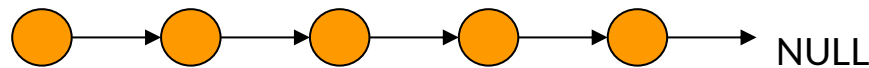
Yippee! Look. I found a tree. How very high the top is!
I hope I found another one. A yummy yooka laptus



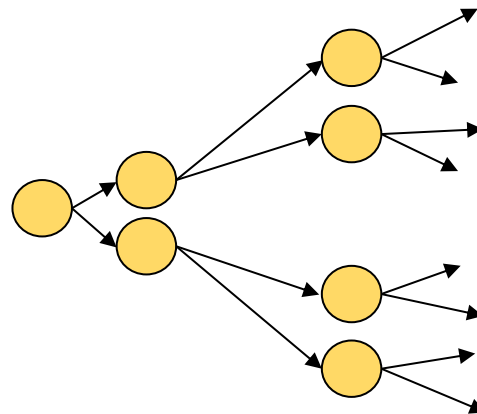
General Tree

previously by Michael Loceff

This is a really cool quest because it shows how a simple perspective change can make a data structure fundamentally different from another one to which it is structurally identical. Consider the linked lists you implemented in Quest 1. Remember how each node has a pointer to the next node. Suppose that each node had 2 pointers instead. Then your linked structure would look more like a binary tree, yes? See Figure 1.



A singly linked list



A binary tree

Figure 1: Two structurally and functionally different data structures

Now suppose you wanted a data structure in which every node had 5 children instead of 2, how might you implement it?

There are a bunch of different alternatives a programmer might consider (each has its own pros and cons - discuss them). Among them are these:

- Make each node have 5 separate pointers (`_child_1`, `_child_2`, etc.)
- Make each node have a linked list of children (5-children is now just a special case)
- Make each node have a vector of children (ditto)

#1 seems like a tailor-mode option for 5 children. But most of the time we realize pretty quickly that it's also its weakness. It will seem that it's not the right level of generality to code into your logic. What is?

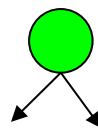
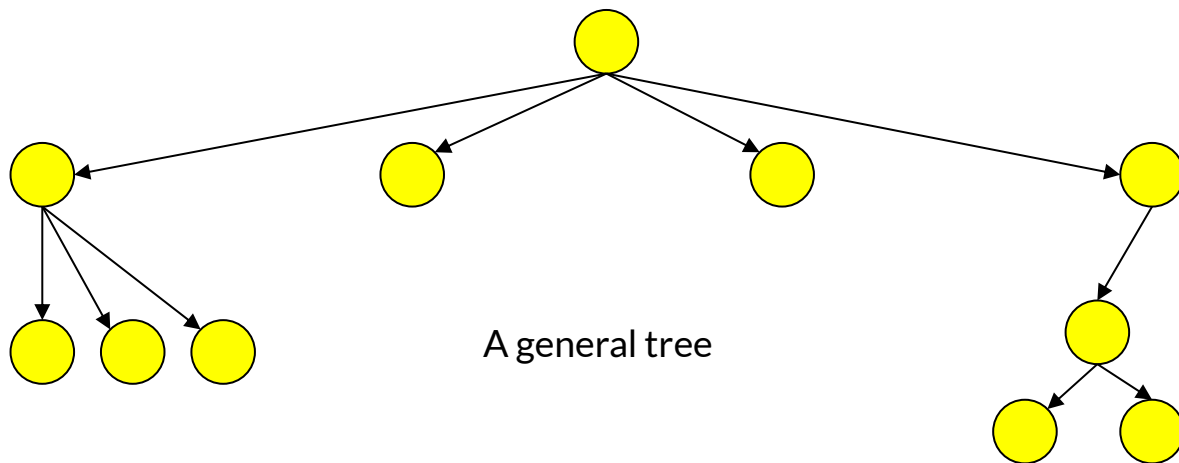
Ideally, you want a general tree in which nodes can have arbitrary numbers of children. How would you implement it?

The two surviving options from the previous list are now:

- Make each node have a linked list of children
- Make each node have a vector of children

Note that node objects from each of these options are structurally distinct from each other. The Nodes defined by these classes are packed differently even though they represent the exact same abstract data structure (a tree where nodes could have arbitrary numbers of children). These objects don't have equally sized sub-objects at corresponding byte locations.

Now you can ponder the central challenge in this quest: Can you use the binary tree's node structure as your node in a general tree? That is, can you make a general tree using just binary branching nodes?



A single node from a
binary tree

Incredible as it may sound, it is possible.

STOP READING THIS SPEC NOW.

Think up a solution.

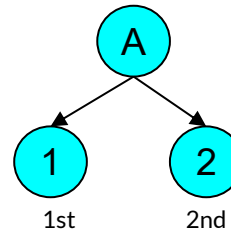
Come back to this spec in a little while...

Welcome back.

See if you have the same solution as I do. I was very surprised when I first saw the equivalence: Leave the concrete data structure alone, but switch your own perspective!

In other words, *look at your binary branching node differently*. Instead of imagining the two node pointers as children of the current node, imagine them as the first child and next sibling.

Instead of looking at a node like this:



Look at it like this:

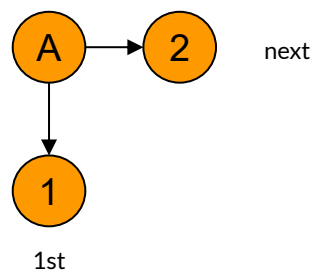


Figure 2: How to use a binary tree node to represent a general tree

Hey Jill, I can show you a cool way to represent a binary tree using NO pointers.

Now take a break again to marvel at the incredibly transformative power of a simple perspective shift. Or not.

Either way, if you want, this is a good time to read your reference material on General Trees. They're also discussed at length in the recommended text. But if you understand the general concept, you can feel free to continue with this quest spec.

In fact, you may even be thinking whether you can implement a general tree using a node that has ONLY ONE pointer in it. Awesome, but wrap up this quest before experimenting further along those lines.

Gee! That's awesome, Jack. Thanks a HEAP!

Now you get to implement a general tree using the 1st-child next-sibling (2-pointer) representation.

Starter Code

Here is part of the Tree class def. You need to find out what's missing and fill it in.⁹

```
class Tree {
private:
    struct Node { // Inner class
        std::string _data;
        Node *_sibling, *_child;
        static bool is_equal(const Node *p1, const Node *p2);

        Node(std::string s = "") : // TODO
        Node(const Node &that);      // TODO
        const Node &operator=(const Node& that); // Deep clone
        ~Node();

        std::string get_data() const { return _data; }
        void set_data(std::string s) { _data = s; }

        Node *insert_sibling(Node *p);
        Node *insert_child(Node *p);

        std::string to_string() const;

        bool operator==(const Node &that) const;
        bool operator!=(const Node &that) const;
    };

    Node *_root;

public:
    Tree();
    ~Tree();

    Tree(const Tree& that) { *this = that; }
    Tree& operator=(const Tree& that); // Deep clone

    std::string to_string() const;
    void make_special_config_1(const std::vector<std::string>& names);

    bool operator==(const Tree &that) const {
        // TODO
    }
    bool operator!=(const Tree &that) const {
        // TODO
    }

    friend std::ostream& operator<<(std::ostream &os, const Tree &tree) {
        // TODO
    };

    friend class Tests; // Don't remove this line
};
```

Before you proceed any further, review the `Tree` class definition until you have many questions (or not). You can also check out our subreddit (r/cs2b) to pick up questions from your classmates if they have extra ones to spare.

⁹ If you don't remember what a `struct` is (from BLUE concepts), look it up for more detail. It's simply a class where everything is `public` by default.

More detail on the individual methods can be found in the following miniquests. Please read them armed with the above questions. If you ask any that are still unanswered after reading the miniquests, I can use that info to improve this spec for your successors.



Your first miniquest - Node constructor

```
Node(std::string s = "") : // TODO
```

The Tree Node supports both default and non-default constructors. The non-default constructor takes a string parameter to use as the data element of the node. While you don't have to implement it inline, I recommend you take this opportunity to play with and learn the inline constructor definition syntax. You'll be coding a single line to go in the above red `// TODO` part in the header file.

Your second miniquest - Node insertions

Now is probably a good time to refer back a few pages to read about how a general tree is represented using child and sibling pointers.

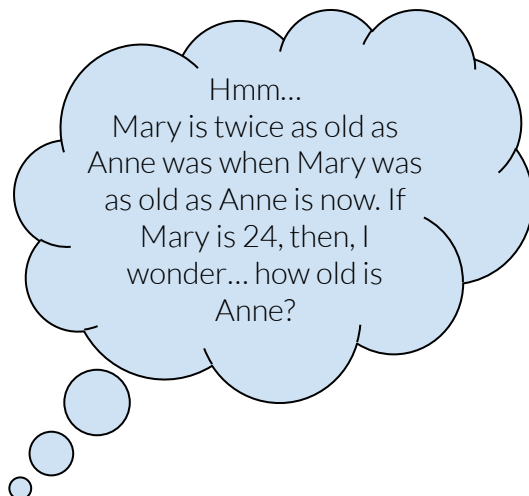
Implement:

```
Node *insert_sibling(Node *p);  
Node *insert_child(Node *p);
```

In both cases, return the node pointer parameter back to the caller. You may NOT assume that `p` or the current node have no siblings or children of their own.

In `insert_sibling()`, first navigate to the last node in the list of siblings and insert `p` at the end.

In `insert_child()`, insert `p` as the child of the current node if it doesn't already have children. Otherwise add `p` as a sibling of your first child using the `insert_sibling()` method.



Your third and fourth miniquests - Node assignment

This one's a biggie. It's only ten lines or so, but it's a biggie. Make the assignment do the work of your copy constructor and your next miniquest will be a cake-walk.

Implement:

```
const Tree::Node &Tree::Node::operator=(const Tree::Node &that);
```

There's two parts to this miniquest. In the first part you have to simply create a copy of the original. You can easily get these points (even by cheating - how?)

To pass the second part, you have to make sure that your copy is unaffected even if your original gets clobbered. It's harder to cheat and pass this one. The straight solution is probably easier.

To make it even easier, here's some starter code for this method. Why do you think the IF statement in this method is important (if it is)?

```
const Tree::Node& Tree::Node::operator=(const Tree::Node &that);
    if (this != &that) {
        // TODO
    }
    return *this;
}
```

Your fifth miniquest - Node copy

Implement the copy constructor:

```
Node(const Node &that);
```

Hopefully you have node assignment working well by now. Collect this reward and move on.

Your sixth miniquest - Node comparisons

Implement:

```
static bool is_equal(const Node *p1, const Node *p2);

bool operator==(const Node &that) const;

bool operator!=(const Node &that) const;
```

It is ok to implement != in terms of == (or vice-versa). The equals operator checks for deep equality defined as follows:

Two nodes are the same if they have the same set of siblings in the same order AND the same set of children in the same order.

Be recursive here and save yourself a ton of time. The `is_equal()` method is your private utility method that compares two nodes for equality, which you can invoke recursively on a node's children and siblings.



Your seventh miniquest - Node to string

Biggie. But just a bookkeeping biggie. When invoked on a `Node` object, it should return a string containing, in order:

1. The `Node`'s data element, followed by a space and a colon (no newline yet)
2. The data-element of each of its children (if any) with exactly one space before each
3. A newline
4. If the node has a first child, the following additional strings:
 - a. `"# Child of [X]\n"`
where the red `X` in square brackets has been replaced by the node's data element.
 - b. A string that is the result of a recursive invocation of `to_string()` on its first child.
5. If the node has a sibling, the following additional strings:
 - a. `"# Next sib of [X]\n"`
where the red `X` in square brackets has been replaced by the node's data element.
 - b. A string that is the result of invoking `to_string()` on its next sibling.

`to_string()` should not print anything. It must simply return the string that looks like the above output description.

Getting this right is largely a matter of taking care of little details, and remembering how the tree is structured (e.g. how do you traverse all of your children?)

If you're not passing this miniquest when your output *looks* identical to the reference output, it's possible that you just can't see the difference. Wow! That's a whopping big hint.

Your eighth miniquest - Node destructor

This miniquest isn't worth a lot of loot.¹⁰ So don't futz around trying to implement an iterative sibling/child deleter like you had to do in the platypus quest. Just do a simple 5-line recursive version to score these easy rewards and move on.

However, you must remember to assign `nullptr` to any pointer you delete.



pandolas

Your ninth miniquest - Tree constructor and destructor

On to `Tree` methods. Implement the default constructor and destructor:

```
Tree();  
~Tree();
```

An empty tree is defined as a `Tree` with no siblings and no children, whose own data element is equal to the string "ROOT". This is the tree a default constructor should make.

These are relatively straightforward methods. But again, remember to assign `nullptr` to any pointers you have to delete (e.g. `_root`)

Your tenth miniquest - Tree copy

Implement the copy constructor and the assignment operator, though not necessarily in that order.

```
Tree(const Tree &that);  
Tree &operator=(const Tree& that); // Deep clone
```

The assignment operator should clone the RHS tree (perform a deep copy). The copy constructor should simply invoke the assignment operator. Again, make sure to check for the special case when someone accidentally assigns a tree to itself. What would happen then if you were careless?

¹⁰ Sssh! Here's a secret: The quester tester won't try and delete a node with an ENORMOUS number of siblings. (But why does that matter?)

Your eleventh miniquest - Tree comparisons

Implement:

```
bool operator==(const Tree &that);  
bool operator!=(const Tree &that) const;
```

The == operator returns true if the two trees are structurally identical. Actual node pointers may be different. By *structurally identical*, I mean that they have the equal root nodes, where node equality is defined as in the earlier miniquest (I think 7).

Your twelfth miniquest - Tree to string

Implement:

```
to_string() const;
```

This is a very easy miniquest if you've managed to ace the `Node::to_string()` miniquest. Otherwise... not so much. It is also an optional miniquest (return "" to skip).

Simply return the stringified version of the root wrapped within *comment* lines as shown below:

```
# Tree rooted at [X]  
# The following lines are of the form:  
# node: child1 child2...  
[...]  
# End of Tree
```



There is exactly one space before each word, except the word "node" on the last line where there are 3 spaces (gray rectangle). As before, the red X in square brackets must be replaced by the name of the root node (which should be ROOT if everything is set up right). `\n` stands for a newline.

Your thirteenth miniquest - Special tree

This is a fun miniquest. All you have to do is to construct a Tree that looks like Fig 3. That's all.

Implement the method:

```
void make_special_config_1(const vector<string> &names);
```

Make sure to `delete` non-null sibling or children pointers before you start assembling the tree. Then insert nodes in the right order at the right places to recreate the tree in Figure 3.

Here is the list of node names¹¹ you must use for the three rows of nodes in the picture, top to bottom, left to right:

- ROOT, AABA, ABAB, ABBA, BABA,
- COBO, COCO, CODO, COFO, COGO, COHO, COJO, COKO,
- DIBI, DIDI, DIFI, DIGI, DIHI, DIJI, DIKI, DILI

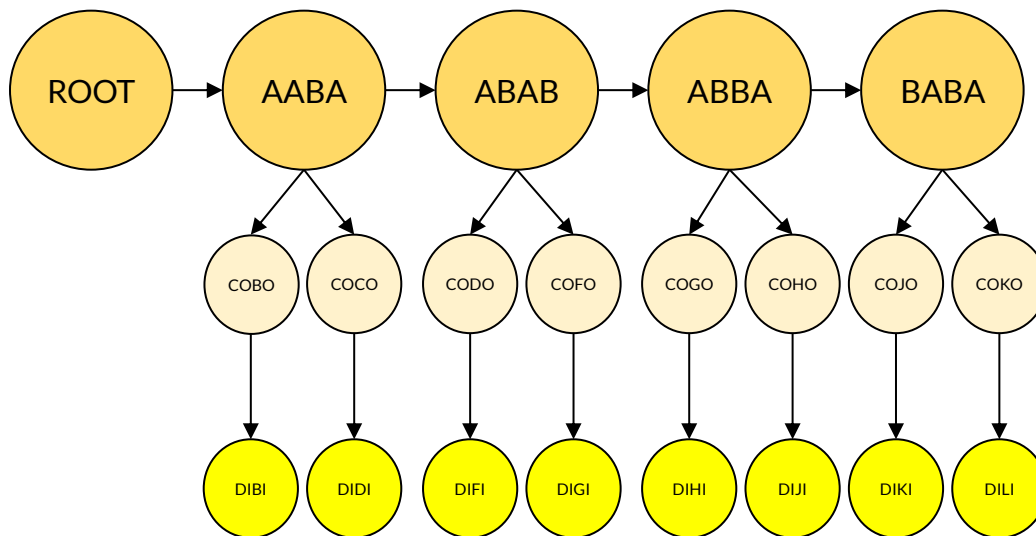


Figure 3: Special Configuration 1 (ROOT has siblings but no children).

In this abstract representation, horizontal arrows point to siblings and non-horiz ones to children (unlike in the UI). The leftmost-child is the first child of the parent. Absent arrows stand for an absent target node (e.g. no siblings or children).

¹¹ The vector `param.names = { AABA, ABAB, ABBA, BABA, COBO, COCO, CODO, COFO, COGO, COHO, COJO, COKO, DIBI, DIDI, DIFI, DIGI, DIHI, DIJI, DIKI, DILI }`

How long is a well written solution?

Well-written is subjective, of course. But I think you can code up this entire quest in about 300 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret code for this quest in the box.
3. Drag and drop your `source` files into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).



Happy Questin',

&

*Kiwis can be complex. Kiwis have a queen.
But mostly they're just fuzzy. Fuzzy, sweet and green.*



Complex Numbers

Solving this quest will earn you 1.0 real tickets to board the Argand plane.

It's a plane that can take you on flights of fancy into the deepest reaches of your imagination.

To ace this quest, you simply need to know how to throw and handle exceptions. You also get to override operators for this quest, but since you've already done that in previous quests, I'm not considering that as something new.

Overview

Obviously, this is not a primer on [Complex Numbers](#). Refer to other sources for that. This quest will require you to have a basic understanding of complex numbers (as the sum of real and imaginary numbers). Here are some selected complex number features (among others) that you get to implement in this quest:

- The Complex Object: You encode it as a pair of double precision floating point numbers which represent the real and imaginary components of the complex number.
- The plus operator: The complex sum is simply another complex number in which the real component is the sum of the real components of all the operands and the imaginary component is the sum of the imaginary components of all the operands.
- The minus operator: The complex difference between two complex numbers is simply another complex number in which the real component is the difference between the real components of the corresponding operands and the imaginary component is the difference between the imaginary components of the corresponding operands.
- The times operator: The complex product of two complex numbers is another complex number. I'll give you more detail on how to calculate it in its miniquest description..
- A reciprocal method: The reciprocal of a complex number. I'll give you more detail about this also in its miniquest description.

Most of the implementation of this Complex class is routine, just like all the other classes you've implemented already. What's different with this one is that some of its methods may encounter exceptional situations (such as attempts to divide by 0, as when calculating the reciprocal). In situations like this you will make and throw an exception.

Note: In this spec, I often refer to a complex number $z = x + yi$ using the tuple (x, y) .

This can be your chance to experiment with and explore the world of exceptions. Find out how they can make your programming life easier.

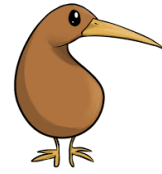


Starter code

I asked the Questmaster for some starter code.

But he said "No way! Go away. This quest is way too easy."

So I said "Ok. Another day. But don't mind me being nosy."



Then I started poking around in his things and managed to get a quick (but) fuzzy photo of the class def from his screen. See if it helps.

```
19 class Complex {
20 private:
21     double _real, _imag;
22
23 public:
24     static constexpr double FLOOR = 1e-10; // Threshold
25
26     Complex (double re = 0.0, double im = 0.0) : _real(re), _imag(im) {};
27     double get_real() const { return _real; }
28     double get_imag() const { return _imag; }
29     void set_real(double re) { _real = re; }
30     void set_imag(double im) { _imag = im; }
31
32     string to_string() const;
33     double norm_squared() const { return _real*_real + _imag*_imag; }
34     double norm() const { return sqrt(norm_squared()); }
35
36     Complex reciprocal() const;
37     Complex& operator= (const Complex & rhs);
38
39     class Div_By_Zero_Exception {
40     public:
41         string to_string() { return "Divide by zero exception"; }
42         string what() { return to_string(); } // more conventional
43     };
44
45     // operators (only the most common ones)
46     Complex operator+(const Complex& that) const;
47     Complex operator-(const Complex& that) const;
48     Complex operator*(const Complex& that) const;
49     Complex operator/(const Complex& that) const;
50
51     bool operator==(const Complex& that) const;
52     bool operator!=(const Complex& that) const { return !(*this == that); }
53     bool operator<(const Complex& that) const;
54
55     friend ostream& operator<<(ostream& os, const Complex& x);
56
57     friend class Tests; // Don't remove this line
```

Your first miniquest - Constructors make things

Implement the default and non-default constructor for the Complex class:

```
Complex(double re, double im);
```

The constructor should take default values for both the real and imaginary values. The constructor with no parameters should give you 0 (which is $0 + 0i$). The constructor with a single real valued parameter x , should give you x (which is $x + 0i$). And the constructor with two parameters x and y should give you the complex number $x + yi$.

If your constructor is correctly implemented, then you get a freebie implicit constructor. You can say something like:

```
Complex c = 3.14;
```

Although the RHS is a single real number, the compiler will auto-invoke your constructor implicitly as if you had written:

```
Complex c = Complex(3.14, 0);
```

Your second miniquest - Equal means equal

Implement the equality comparison operators:

```
bool operator==(const Complex& that);
```

```
bool operator!=(const Complex& that);
```

Ideally, one of these is implemented in terms of the other. But here's an opportunity to reward me with some discussion about whether it is better to do it that way rather than implement each comparator independently.

Your third miniquest - Assign foretells incoming goodies

Yeah! Do you even have to implement this puppy?

Or is this, like, a TOTALLY FREEBIE reward!

I can't believe it, peeps.



Your fourth miniquest - What's the norm around here?

The norm of a Complex number $c = a+bi$ is defined as follows:

$$|c| = \sqrt{a^2 + b^2}$$

Implement:

```
double norm() const;
```

which will return the norm of the Complex, c , on which it is invoked.

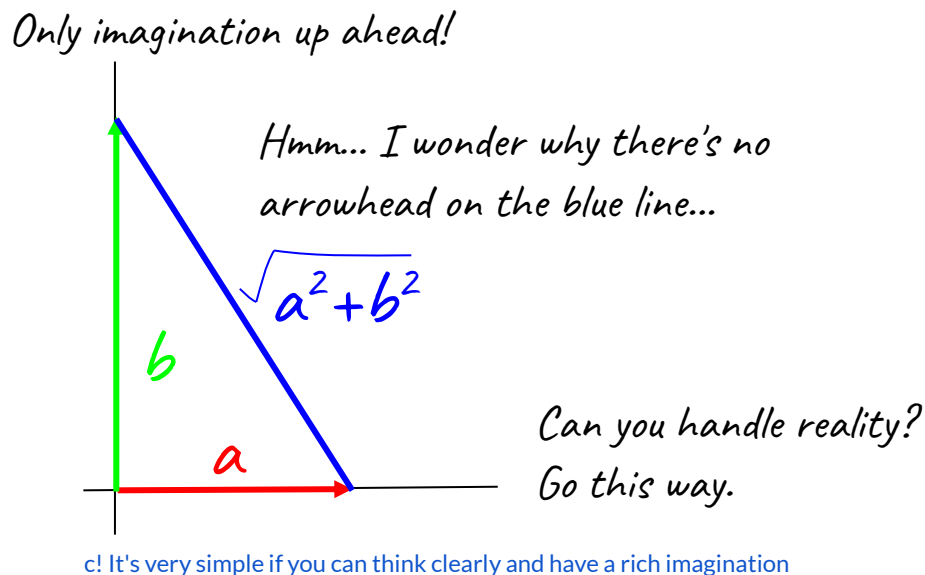


Figure 1 (The norm as the hypotenuse on the Argand plane)

Your fifth miniquest - Compare and contrast

Implement the quantitative comparison operators:

```
bool operator<(const Complex& that) const;
```

Someone said "Hey! I thought you said operators? But I only see one operator up there."

That would be right. You can implement all of them, but you only need to implement the other comparison operators to make your interface sweeter to REAL users of your Complex class. As for me, I'll use your $<$ operator to manufacture the other ones (How?)

The only downside I can see is that if you got your $<$ wrong, then all your comparison operations will be wrong. Still... I think it's less work to get it right.

We'll define less than to refer to the norms of the complex numbers internally. That is, $c_1 < c_2$ if $\text{norm}(c_1) < \text{norm}(c_2)$.

Your sixth miniquest - Plus... don't be nonplussed

Implement

```
Complex operator+(const Complex& that);
```

which returns a new complex number that is the sum of its two operands ($*this + that$)

Your seventh miniquest - Merry Minus

Yeah, I know it sounds like a quest in itself. But it's really quite simple. Implement:

```
Complex operator-(const Complex& that);
```

which returns the difference between its operands ($*this - that$)

Your eighth miniquest - Times and Times again

The product of two complex numbers (a, b) and (c, d) is defined as $(ac-bd, ad+bc)$.

Is it commutative? Why? (or why not?)

Regardless, implement

```
Complex operator*(const Complex& that);
```

which returns the product (assuming $*this = a + bi$ and $that = c + di$)



Your ninth miniquest - A reciprocal arrangement

Implement a reciprocal arrangement that works well enough to handle all non-empty quantities.

```
Complex reciprocal() const;
```

Assuming that a complex number, z , is not zero, we want $1/z$, that is, a number z^{-1} such that $z * z^{-1} = 1$.

if $z = a + bi$, then we want:

$$\begin{aligned} 1/z &= 1/(a + bi) \\ &= \frac{a - bi}{(a + bi)(a - bi)} \\ &= \frac{a - bi}{a^2 + b^2} \end{aligned}$$

In your code, you can leverage the fact that the denominator is simply the square of the norm. The reciprocal is the complex number $(a / (a^2 + b^2), -b / (a^2 + b^2))$

Your tenth miniquest - The Daily Divide

If you nailed the basic reciprocal miniquest, the daily divide is practically a freebie.

Implement the division operator well enough to work on everyday denominators.¹²

```
Complex operator/(const Complex& that);
```

For a non-zero complex z_2 (which is all this miniquest needs to handle):

$$z_1 / z_2 = z_1 * (\text{reciprocal of } z_2)$$



¹² I mean, you don't get to divide by zero everyday, do you?

Your eleventh miniquest - Impossible to repay!

Now we're talking.

Extend your reciprocal function to handle the case when it's invoked on the zero complex number. Really, I should be saying *a* zero complex number because we defined our zero to be a range of numbers between two limits.

When asked to reciprocate with something for nothing, it's always better not to throw a fit. But when asked to find the reciprocal of zero, it's always better to throw an exception (since there is no single correct answer).¹³

The exception object you will create and throw must be defined as an inner public class within the `Complex` class (as `Complex::Div_By_Zero_Exception`)

This public inner class, `Div_By_Zero_Exception`, must provide two public methods:

```
string Div_By_Zero_Exception::what();  
string Div_By_Zero_Exception::to_string();
```

Either method (one may simply relay the other if you want), if invoked on this exception object, should return the exact string: "Divide by zero exception"

Usually, the person who asked you to do something is the one who has immediate responsibility for your actions. And this includes exceptions. So the caller of your method is responsible for catching and handling any exceptions your method may throw. If they don't catch it, the exception will pass over to their caller, and so on.

The idea here is that anytime you come to a situation you can't handle, you make an informative exception and throw it. Someone had better catch that exception. If nobody catches it, it will roll all the way down to the run-time system which will KILL all programs that throw things at it.

In computer programs, exceptions often allow us to easily propagate all exceptional situations to an appropriate common layer of processing so we can factor our error handling in the same way as we factor the rest of the code.



Here is how I would create and throw a brand new `Div_By_Zero_Exception` object, maybe in my reciprocal code:

```
...  
if (my_denominator <= Complex::FLOOR) // watch for round-off  
    throw Div_By_Zero_Exception();  
...
```

¹³ I think Ken Thompson (Unix) said this. But you may want to look up the source: *Don't test for errors you don't know how to handle*. (Throw an exception and let your caller handle it if they know how).

Define a suitably small number in your complex class and use it as a proxy for zero. When you're working with doubles and floats (which are more similar to density functions than discrete distributions - Discuss), it helps to define ranges a value should fall in rather than be exactly equal to. The range I picked for my Complex 0 is -10^{-10} to $+10^{-10}$

So I defined my `Complex::FLOOR` to be $1e-10$. If any absolute denominator is smaller than this `FLOOR`, I consider that an attempt to divide by zero.

Here is how I might catch your exception object:

```
try {  
    ...  
    // I try to do things with complex numbers here  
    ...  
} catch (Complex::Div_By_Zero_Exception e) {  
    cerr << e.what() << endl;  
    exit(-1);  
}
```

If anything about this miniquest is unclear, all you have to do is ask (in our [subreddit channel](#))

Your twelfth miniquest - The Great Divide

Just like before, extend the division operator's domain to handle division by 0. But rather than make and throw an exception yourself, you can blissfully ignore the possibility and let your reciprocal calculation handle the exception throwing.

Oh wait! Isn't that already done? Umm.. ok. Do you got to do anything at all for this miniquest?

Bummer... Looks like I didn't really think this one through. Still a reward is a reward. You earned it if you got this far.

Your thirteenth miniquest - To string

Implement

```
string Complex::to_string() const;
```

When invoked on a complex number $a + bi$, return the string `(a,b)`. There is no space after the comma. To format the numbers, I use:

```
sprintf(buf, "(%.11g,%.11g)", _real, _imag);  
return string(buf);
```

You may already know how to format numbers for stream output using c++ precision specifiers. This is your chance to learn about an ancient and powerful formatting technique that is making a comeback. Look up "printf formatting" online to see what the above specification means. Share your finding summaries on our [subreddit](#).

Your fourteenth miniquest - ... and beyond

Make a friend. Implement:

```
friend ostream& operator<<(ostream &os, const Complex& x);
```

Something tells me that this reward is also as good as a freebie if you've aced the previous miniquest.

Do you think that's way too many freebies for a quest?

How long is a well written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in under about 150 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret code for this quest in the box.
3. Drag and drop your source files (`Complex.*`) into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).



Happy Questin',

&

*Oy! Just b'cuz me digits be oight
It don't mean me name be boit*



meg the manitoed octopus

Shapes

In this quest, you get to play with inheritance and polymorphism in C++.

What is polymorphism? Take a break to look up your reference material now. TL;DR? Here is the abstract scoop (followed by the more concrete example which you will implement in this quest):

Suppose you had many different classes, x_1, x_2, \dots , that all inherited from the same base class A and that there is some method of A , say `foo()`, that is overridden (possibly differently) by each of its descendants.

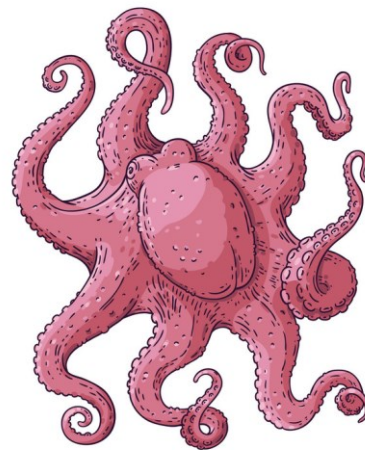
Let's say you have an array of pointers to objects of type A .¹⁴ Each object pointed to by this array's elements may be a different x_i and each x_i may implement its own overriding `foo()` according to its own specific requirements.

Polymorphism lets you write code in which you can invoke `foo()` via a pointer to an object of type A , only to have it execute the `foo()` code that is attached to its derived class x_j rather than the *fallback-code* that may be supplied with class A .

I bet you're still confused. Cuz I almost done confusin' meself writin that. This will help: Here is the concrete example (which you will also implement in this quest).

Suppose I have a generic `class` called `Shape` (a geometric shape). There may be many different subclasses of `Shape`: `Circle`, `Square`, `Triangle`, etc. Each of these subclasses must have its own `draw()` method (enforced by the compiler). 'Cuz I can't draw a triangle from vertices in the same way I draw a square, yes?

Polymorphism is a way that lets us invoke `draw()` blindly on a set of `Shape` objects when we don't know the precise type of each object. The idea is that the system will automatically invoke the correct `draw()` on each object depending on the specific subclass it belongs to (Get yer eyes off the dancing octopus and look at Figure 1 already!)



¹⁴ Why pointers? Can we not have an array of the objects themselves?

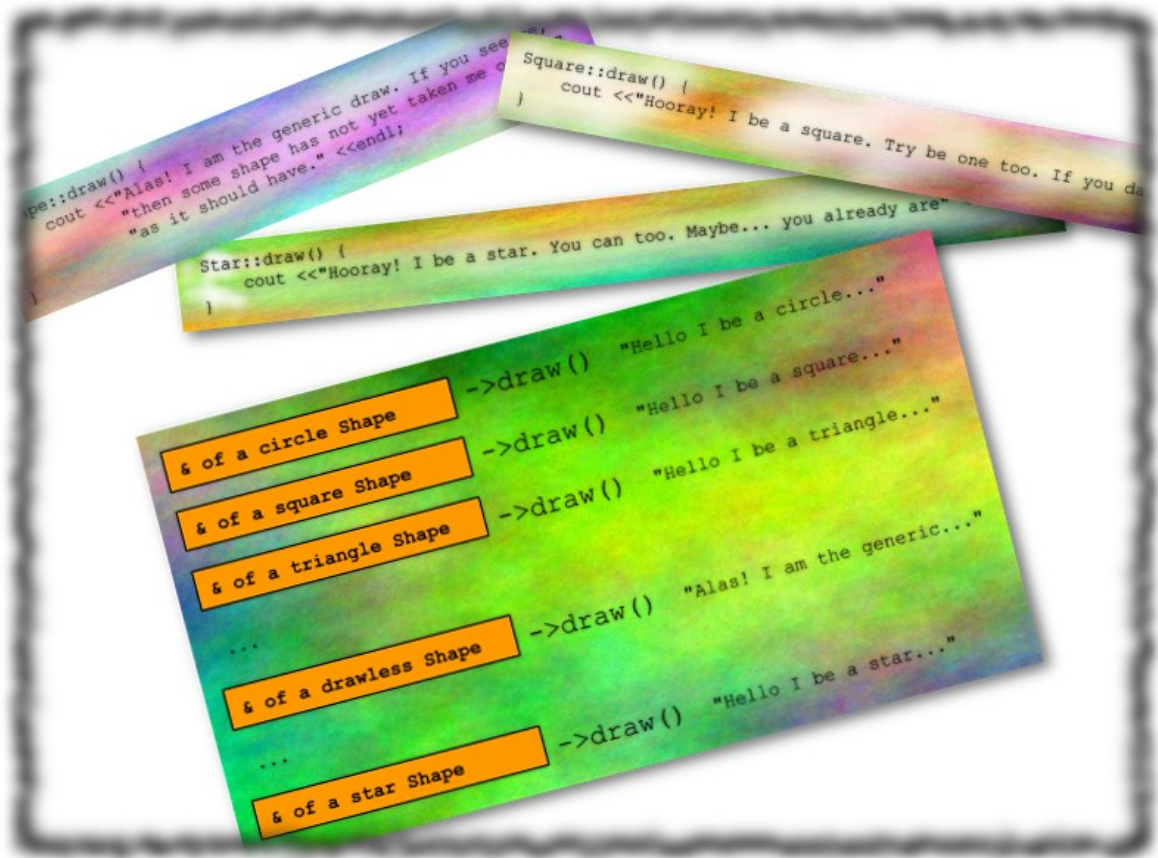


Figure 1: `draw()` is a virtual function that resolves to the appropriate derived class member function automatically. Imagine the orange boxes to be cells in an array of pointers to `Shape` objects.

In this quest, you will implement many little classes. Some of these will interact with each other in fun ways (See Figure 2).

- A `Screen` class - Your canvas, on which you can paint using ASCII characters (That is, each pixel on this canvas is an ASCII character).
- A `Shape` class - The base class for all the shapes you're gonna create. In fact, we'll make it an *abstract* class by assigning the value 0 (the null pointer) to the method we want all subclasses to implement: `draw()`. This *empty implementation assignment* tells the compiler that it is what we call a *pure virtual function*. Any class that subclasses (derives from) `Shape` MUST implement `draw()` or the compiler will consider that derived class also an abstract class. An abstract class cannot be instantiated because it has not yet been completely defined. The missing method implementations complete it. Then it becomes a *concrete* class. You can instantiate objects of a concrete class.
- You will also implement the following concrete subclasses of `Shape`: `Point`, `Line`, `Quadrilateral`, `Upright_Rectangle` and `Stick_Man`
- Each subclass should know how to draw itself (and not other Shapes) on a given screen using a given character. Thus each should have its own implementation of the `draw()` method.

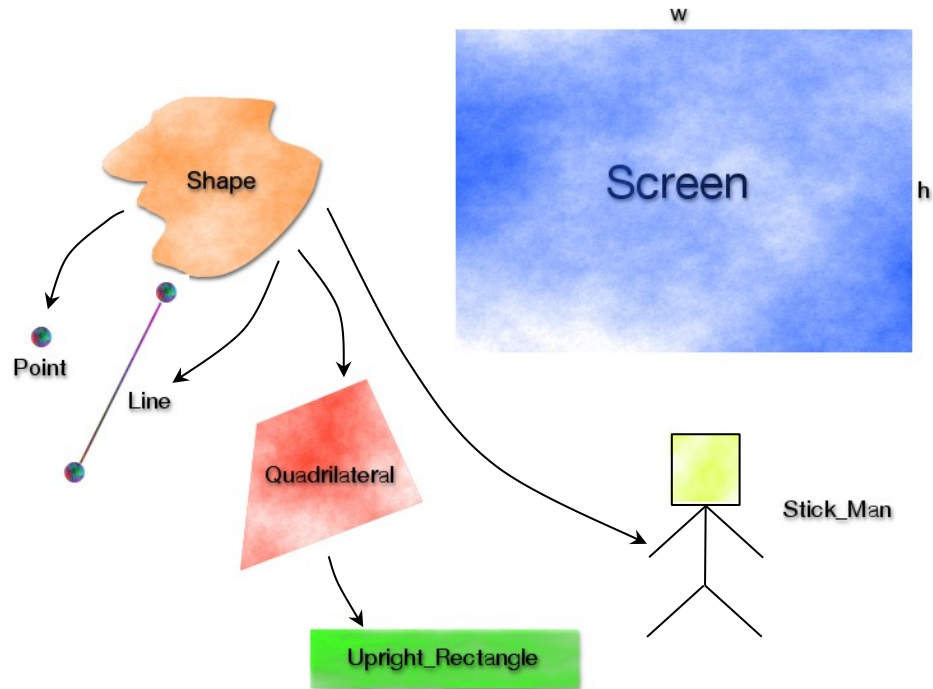


Figure 2: Classes you must implement. Arrows represent inheritance. `Shape` is an abstract class. Its subclasses must implement `draw()`

There's a lot you can do in this quest. Most of it is easy, but some miniquests are non-trivial. So I would suggest this:

Work on this quest until you get the password to move on. Then keep coming back to this quest to revisit skipped miniquests until the second great freeze.

Your first miniquest - Screen

Implement the `Screen` constructor:

```
Screen::Screen(size_t w, size_t h)
```

It should correctly set the size of its `_pix` vector to have `h` rows and `w` columns. This means `_pix[]` should end up with `h` `vector<char>`s, each of which is `w` chars big.

Points to keep in mind about the `Screen` class (not necessarily to do with the constructor):

- Vectors at `_pix[0...h-1]` represent screen pixel rows 0 to `h-1` from bottom to top.
- The origin, `(0, 0)`, visually located at the bottom left of screens, is now the top left of the `_pix` vector if you imagine the vector growing downwards with row 0 at the top. So you have to get used to flipping the image vertically as you process it (like in the eye?). Or do you need to?

Your second miniquest - Fill

Implement:

```
void Screen::fill(char c)
```

Just fill up the screen (all elements of all rows of `_pix`) with the supplied character, `c`.

Your third miniquest - Clear

Implement:

```
void Screen::clear()
```

Just fill up the screen (all elements of all rows of `_pix`) with the background character, `Screen::BG`. You can simply invoke `fill()` from here. Hey look! I already gave you this function line! Well, I guess I can't take it back now.



Your fourth miniquest - To string

Well, there's always a tricky one in every quest. I guess. But this doesn't have to be tricky if you remember that the bottom row of the vector is the top row of your screen and vice-versa.

This is also where you might wonder if you'd have been better off mapping bottom to bottom and top to top, and then removing the corresponding adjustments from the draw methods.

Implement:

```
string Screen::to_string() const
```

It should return a string made of `h` newline delimited strings, each of which represents the corresponding row on a screen (line 0 is the top of the screen). That is, I should be able to say something like:

```
cout << my_screen.to_string() <<endl;
```

and have it print out a rectangular frame of characters in the same order I would see if the frame was stored in a flat file and dumped to the console. For example, if `my_screen` had a width of 11 and a height of 6, and it contained the following characters:

```
_pix[5] = '  - - - - -  '  
_pix[4] = ' | /\_/\ | '  
_pix[3] = ' | ( o o ) | '  
_pix[2] = ' | > ^ < | '  
_pix[1] = '  - - - - -  '  
_pix[0] = 'Schrodinger'
```

it's output should be what miniquiest 1 of Quest 2 requires.

Oh wait! Wrong Quest. But still... you get the idea.

If not, bring it up in our [subreddit](#).

Your fifth miniquiest - Output

Implement:

```
friend std::ostream &operator<<(std::ostream &os, const Screen &scr)
```

Hey! It looks like I gave you that one too. I'm gonna have to think up some harder miniquiests.



Your sixth miniquiest - Point

We start our Shapes with `Point`, seeing as I've already given you the `Shape` class in the starter code.

Complete the implementation of the concrete class, `Point`. Remember that the word *concrete* means that the class is not abstract. It implies that the class is instantiable - that there are no inherited pure virtual methods left without implementations.

Essentially, what this miniquiest boils down to is just that you implement:

```
bool Point::draw(Screen &screen, char c)
```

If the point falls within the given screen's boundaries, place the character `c` at the corresponding location within the screen's `_pix` array and return true. If not, return false.

How can `Point` access the private members of `Screen` (`_h`, `_w`, and `_pix`)? Although you can do this through friendship (experiment, discover and share your findings), it's just as easy to keep the classes opaque to each other and use the getters provided by `Screen`. Discuss the pros and cons in the [subreddit](#) if you have time.

Your seventh miniquest - Line by ...

Implement:

```
bool Line::draw_by_x(Screen &screen, char c, size_t x1, etc.)

bool Line::draw_by_y(Screen &screen, char c, size_t x1, etc.)
```

A `Line` is represented by 4 unsigned integers (not two `Points`): `x1`, `y1`, `x2` and `y2`. It is defined as the set of points connecting the points `(x1,y1)` and `(x2,y2)`.

When it comes to slanting lines, you'll find that simple strategies don't always work. The problem is that either `x` or `y` will need to be incremented by a fractional quantity for every unit-increment of the other. This will make more sense after you read the following general strategy to draw a line:

- Always draw the line from left to right. If it is vertical, draw it from bottom to top.
- If the line is wider than it is tall, then starting at the leftmost pixel, increment `x` by one each time and `y` by the fraction determined by the slope of the line (See Figure 4).
- If the line is taller than it is wide, then starting at the bottommost pixel, increment `y` by one each time, and `x` by the reciprocal of the slope of the line.
- Finally, return true if the line was *entirely* contained within the screen and false if even one of the points of the line falls outside screen bounds.

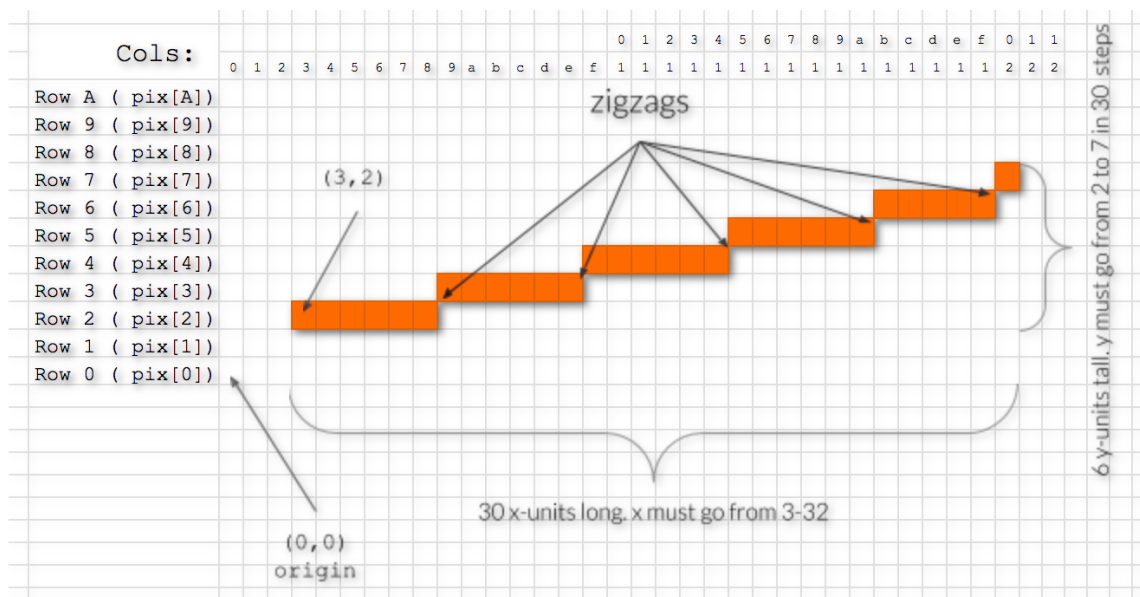


Figure 4: How to draw a slanting line

In Figure 4, you can see how to select the pixels to turn on for a line that goes from (3,2) to (32,7). Since the line is wider than it is tall, our strategy would draw the line along the x-direction. What this means is that you will calculate N coordinates where N is the horizontal length of the line (30). Your x would be steadily incremented by 1 during each of N iterations. The corresponding y-coordinate for each x will be calculated by adding a fraction, which I call dy in the code, to the previous value of y.

The basic idea is to try and draw the slanting lines such that the number of zigzags is *maximized* and they don't occur at aesthetically jarring locations.¹⁵ In this miniquest, our chosen strategy inserts $h-1$ zigzags (where h is the vertical length of the line and distributes the zigzags to be equally spaced between the ends of the line.

In the example, y needs to go from 2 to 7 in 30 iterations (x from 3 to 32). This means your $dy = 5/29.0$, which is the quantity by which you would increment y for each increment in x. If done properly, you will find that y rolls over from one coordinate to the next once every 6 iterations (when x would have moved by 6 steps) on average. (Why on average?)

Try the technique on a piece of paper and convince yourself it works. This method is called `draw_by_x()` in the spec.

Because this method involves floating point calculations, it's possible to calculate quantities slightly differently and get a correct answer that is a little different from my reference answer. Rounding and adjusting will catch most of these drifts. But in rare cases, even small mismatches accumulate over time and manifest at the ends of long lines (mainly at the end points). To avoid this confusion, try to model your calculations to mirror mine as much as possible.

Here is a photo of my `draw_by_x()` method. I suggest you implement your own first, and then AFTER you have your version working, use the photo below as a reference to troubleshoot when your lines are off-by-one from mine. Once you have your `draw_by_x()` working you must create a corresponding `draw_by_y()` method.

```
// Static helper
// Draw pixels on the screen along the X direction, using the supplied
// char (ch) as the pixel. The number of segments will be abs(y2-y1).
// Each segment will be abs(x2-x1)/abs(y2-y1) pixels long.
//
bool Line::draw_by_x(Screen& scr, char ch, size_t x1, size_t y1, size_t x2, size_t y2) {
    if (x1 > x2)
        return draw_by_x(scr, ch, x2, y2, x1, y1);    // reorder

    double dy = ((double) y2-y1)/((double) x2-x1);
    bool contained = true;
    double x = x1, y = y1;
    while (x <= x2) {
        contained &= Point((size_t) x, (size_t) y).draw(scr, ch);
        x += 1; y += dy;
    }

    return contained;
}
```

Figure 5: `draw_by_x()`

¹⁵ However, this is not entirely true in this quest. Our strategy allows some aesthetically sub-optimal zigzag placements to squeak through. Specifically, which are these and what might be a good way to handle them also?

Note that my `draw_by_x()` always draws from left to right. If my two points are ordered differently, then rather than futz around with swapping values within the code, I simply make a tail-recursive invocation of the same method with swapped points. You don't have to do it this way, of course. But if you do, note that the recursion overhead here is small and capped at one stack frame.¹⁶

Your eighth miniquest - Line

Implement:

```
bool Line::draw(Screen &screen, char c)
```

Once you have the two `draw_by_...()` methods done, this one is eazy peazy. All this method has to do is to determine if the line is short and fat or tall and thin, and invoke the corresponding `draw_by_...()` method.



Remember that it must also return a boolean indicating non-overflow. You can blindly relay what your subordinate method returns to your caller.

Your ninth miniquest - Quadrilateral

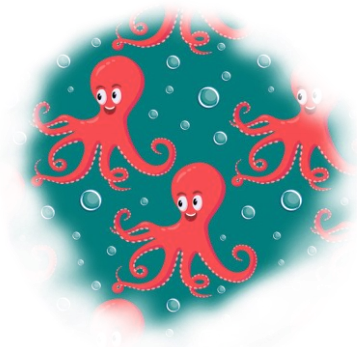
Implement:

```
bool Quadrilateral::draw(Screen &screen, char c)
```

You just have to determine the correct end points of the four lines that make this quadrilateral and draw them.

In the past, I've noticed that some questers put in a great deal of effort to *avoid* redrawing corner pixels (where the edges overlap and intersect).

Let me just wish you good luck if you get sucked into that rabbit hole. Once you've established that overwriting an existing pixel with the same value is *safe*, you'll discover that it is a wiser choice to overwrite it than to build elaborate checks and guards. These kinds of educated tradeoffs are what come in handy as you gain more experience in programming.



¹⁶ Discuss this in our [subreddit](#).

The Upright Rectangle

A quarryshmaterell uses a lot of letters and it's hard to spell right. It also requires 4 points (and thus 8 numerical coordinates) to specify. Sure looks messy and if you're ever in a situation where the only quads you need are upright rectangles, you can simplify your life a great deal by using just 4 numbers to encode your shape - the bottom left and top right.

Thus was born the `Upright_Rectangle`.

This is not a miniquest. You'll notice that the `Upright_Rectangle`'s implementation is already completely fleshed out in the provided starter code. But you do need to know how to instantiate and use it because it is the head of the next class you need to create.

Nonetheless, FWIW, you get a freebie reward just for reading this section. Hooray!

Your tenth miniquest The Stick Man constructor

Implement:

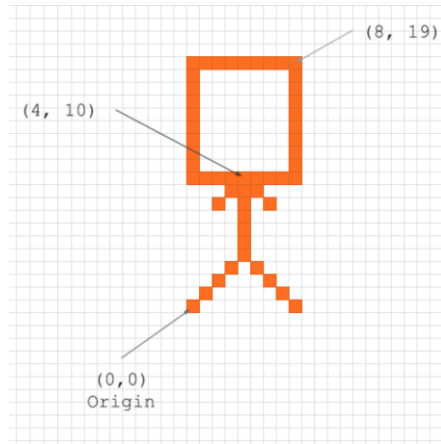
```
Stick_Man::Stick_Man(size_t x, size_t y, size_t w, size_t h)
```

Imagine that your stick man is contained within a rectangular portion of a screen. Then the coordinates of the bottom left of this portion is (x, y) and its width and height are given by w and h respectively.

- First clear out the `_parts` vector and set any required member variables.
- Then create and push pointers to following `Shapes` into this vector:
 - The stick man's head: An upright rectangle with bottom left at $(x+0, y+h/2)$ and top right at $(x+w-1, y+h-1)$.
 - His torso: A line from $(x+w/2, y+h/2)$ to $(x+w/2, y+h/4)$
 - His left arm: A line from $(x+w/2, y+h/2)$ to $(x+w/4, y+h/4)$
 - His right arm: A line from $(x+w/2, y+h/2)$ to $(x+3*w/4, y+h/4)$
 - His left leg: A line from $(x+w/2, y+h/4)$ to (x, y)
 - His right leg: A line from $(x+w/2, y+h/4)$ to $(x+w-1, y)$

That is your stickman. Your default height is 40 and default width is 20. If the constructor is passed values of 0 or 1 for h , you should silently set it to this default height (don't worry about throwing exceptions). Likewise for w .

Here is a picture of a nice little 9x20 stick man. Of course, not all stick men are nice or little, never mind both!



Note: If the shapes you create are local to the constructor (i.e. on the stack), you'll find that they're gone when the constructor exits. To make sure the shapes persist beyond the lifecycle of the constructor, they must be on the heap. This means you must delete them in your destructor or suffer memory leaks.

Your eleventh miniquest - Draw a Stick Man

Implement:

```
bool Stick_Man::draw(Screen &screen, char c)
```



Just like the other inherited draw methods, this method should also draw its shape (in this case a `Stick_Man`) on the given screen using the character `c`.

Simply invoke the `draw()` method on each of the objects whose pointers you can find in your `_parts` vector. You don't have to worry about the exact type of each object because you know that every one of those objects is a *drawable* object. That is, it supports a `draw()` method whose signature is exactly as given in the base class.

Feel free to gush prosaic in our [subreddit](#) at this nice solution to the problem of invoking distinct methods (with the same signature) on many objects sharing the same parent.

Yippee! Problem solved!

Your twelfth and final miniquest in this quest

Yeah, I know an octopus only has 8 feet. But we're not mapping the miniquests to creature feet in our quests. Otherwise you'd never be able to complete my millipede quest.

Implement:

```
Stick_Man::~~Stick_Man ()
```


You must take care to explicitly delete every shape you created in your constructor before the Stick_Man destructor forgets the (presumably only) pointers to them in your code. When would that happen?

Starter code

Here are the (sometimes incomplete) class definitions you can use to flesh out your classes. I suggest you implement this quest in two files: A `Shapes.h` file which contains all the class definitions (skeleton below), and a `Shapes.cpp` file that contains implementations for all of the non-inline methods you declared in your classes.

Before you decide to copy/paste the following code, remember that this is a PDF file. Some IDEs (esp. Xcode) are notoriously bad at making certain control characters visible. So your code may fail to compile with no obvious syntax issues. I think your best bet is to read the code and type in your own version of it.



```
// ----- Screen, friend of Shape -----
// A virtual screen with pixels x: 0-(w-1) and y: 0-(h-1)
// NOTE: (0,0) is the bottom left - Pixels can be any character, determined
// by each Point.
//
class Screen {
    friend class Shape;

private:
    size_t _w, _h;
    std::vector<std::vector<char>> _pix;

public:
    static const char FG = '*', BG = '.';

    Screen(size_t w, size_t h);

    size_t get_w() const { return _w; }
    size_t get_h() const { return _h; }
    std::vector<std::vector<char>>& get_pix() { return _pix; }

    void set_w(size_t w) { _w = w; }
    void set_h(size_t h) { _h = h; }

    void clear() { fill(BG);}
    void fill(char c); // TODO - implement in the cpp file
    std::string to_string() const;

    friend std::ostream &operator<<(std::ostream &os, const Screen &scr) {
        return os << scr.to_string();
    };

    friend class Tests; // Don't remove this line
};

//-----
//-----
```

```

// ----- Shape -----

// Abstract base class for circle, rectangle, line, point, triangle, polygon, etc.
//
class Shape {
public:
    virtual ~Shape() {}

    virtual bool draw(Screen &scr, char ch = Screen::FG) = 0;

    friend class Tests; // Don't remove this line
};

// ----- Point -----

class Point : public Shape {
private:
    size_t _x, _y;

public:
    Point(size_t x, size_t y) : _x(x), _y(y) {}
    virtual ~Point() {}

    bool draw(Screen &scr, char ch = Screen::FG);

    friend class Tests; // Don't remove
};

// ----- Line in two point notation -----

class Line : public Shape {
private:
    size_t _x1, _y1, _x2, _y2;

    // Helpers
    static bool draw_by_x(Screen &scr, char ch,
                          size_t x1, size_t y1, size_t x2, size_t y2);
    static bool draw_by_y(Screen &scr, char ch,
                          size_t x1, size_t y1, size_t x2, size_t y2);

public:
    Line(size_t a, size_t b, size_t c, size_t d) : _x1(a), _y1(b), _x2(c), _y2(d) {}
    virtual ~Line() {}

    bool draw(Screen &scr, char ch = Screen::FG);

    friend class Tests; // Don't remove
};

// ----- Quadrilateral -----

// A general quadrilateral with points (x1,y1) ... (x4,y4), clockwise
// from bottom left. For the special case when x1==x2, y2==y3, x3==x4
// and y4==y1, we'd use an Upright_Rectangle.
//
class Quadrilateral : public Shape {
private:
    size_t _x1, _y1, _x2, _y2, _x3, _y3, _x4, _y4;

public:
    Quadrilateral(size_t a, size_t b, size_t c, size_t d,
                  size_t e, size_t f, size_t g, size_t h) :
        _x1(a), _y1(b), _x2(c), _y2(d), _x3(e), _y3(f), _x4(g), _y4(h) {}
    virtual ~Quadrilateral() {}

    bool draw(Screen &scr, char ch = Screen::FG);

    friend class Tests; // Don't remove
};

```

```
// ----- UprightRectangle, a special Quadrilateral -----
// A Rectangle is a special upright Quadrilateral so we don't have to
// parameterize the constructor with a ton of numbers
//
class Upright_Rectangle : public Quadrilateral {
public:
    Upright_Rectangle(size_t x1,size_t y1, size_t x2, size_t y2) :
        Quadrilateral(x1,y1, x1,y2, x2,y2, x2,y1) {}
    virtual ~Upright_Rectangle() {}
};

// ----- StickMan, a composite Shape -----

class Stick_Man : public Shape {
    static const size_t DEFAULT_W = 20, DEFAULT_H = 40;

private:
    size_t _x, _y, _w, _h;
    std::vector<Shape *> _parts;

public:
    Stick_Man(size_t x = 0, size_t y = 0, size_t w = DEFAULT_W, size_t h = DEFAULT_H);
    virtual ~Stick_Man(); // Needed to deallocate parts

    const std::vector<Shape *>& get_parts() const { return _parts; }
    bool draw(Screen &scr, char ch = Screen::FG);

    friend class Tests; // Don't remove
};
```

How long is a well written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in under about 350 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret code for this quest in the box.
3. Drag and drop your source files (Shapes.*) into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).



Happy Questin',

&

*behold the pilgrims, serpentine
like ants that merge in endless line*



an ant is endless

Queues

Well, not just any old queues. But queues on top of arrays.

In BLUE, you got to implement a stack on top of an array. Here, you get to build a queue like that.

But to keep it challenging, you may not explicitly access the heap or invoke the `new/delete` operators. You can, however, use the built-in `std::vector` class.



Even though `vector` gives you an array that is re-sizeable at run-time, you should not resize your vector except when your queue itself is resized by its user. When your queue is instantiated (or resized), you get one chance to make a (or the) vector just big enough to accommodate exactly as many elements (plus any small fixed overhead you may need).

Ideally, if you haven't yet encountered this problem, stop reading the spec now and try to implement it on your own. If you get stuck, then you can always refer to the section below for tips and insights.¹⁷

Implementation details

If you got here, then you've probably at least thought about implementing queues the same way as you implemented stacks (with arrays). Maybe you found that it's not as easy. If you found yourself having to shift elements around in your vector every now and then to *make space*, then it's likely your implementation was off. All queueing and dequeuing (sometimes called push and pop) must operate in constant time that is *independent* of the size of the queue. How?

Again, a simple perspective shift will help. Just look at this dependency from the other side. By making the enqueue and dequeue operations dependent on the size of the queue, you can avoid making the queue size dependent on them (which is what you'd have had to do before). How do you code this insight in practice?

Take a break now to think up a solution to this problem.

The answer, if you haven't managed to zero in on it yet, is to treat the array as *circular*. Every element in a circular array has a unique successor and predecessor, including the first and the last. The successor of the last element is the first element and the predecessor of the first element is the last element.

You can get this effect by indexing into location `j % array.size()` whenever you want to index into location `j`.

¹⁷ Remember: Nobody likes data structures with unpredictable performance. For example, if your enqueue (or dequeue) operations were, like, blindingly fast most of the time, but every once in a while something kicks in to slow it down to a grind, I'll stop using it. And, oh: faster is better than slower, for the same price.

See Figure 1 for a pictorial representation of this idea.

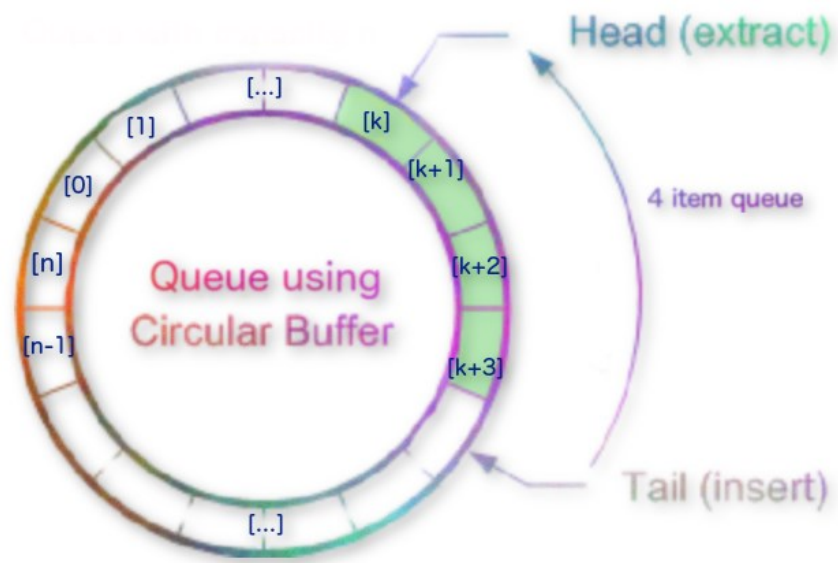


Figure 1: A queue (size n) using a circular array (size $n+1$). This queue has 4 elements in it, with the first at array index k and the last at array-index $k+3$.

Nuance: Here is a tip that may save you a great deal of book-keeping pain. To implement a queue of N elements, use an array of $N+1$ elements. This 1-element overhead will be one of the easiest ways for you to distinguish between an empty queue and a full queue. How?

If tail always points to the spot *after* the last element in the queue (that is, the potential location of an incoming element), then it is empty if `head == tail` and full if `head == (tail + 1) % array.size()`.

With these checks, you can disallow insertions if your last element is occupying an index, which, if incremented (circularly) would end up making it point at the head (because then it would be indistinguishable from an empty queue).

Every enqueue and dequeue operation will incur this limit check cost. But can you think of a less expensive way to maintain the queue? Even if you had to use moderately more space?¹⁸

¹⁸ The version with the one-element overhead is the one you will have to implement in this quest. This means that the size of the queue (as visible to the outside world through your `size()` method) is at most one less than the size of your backing array, which should contain at the very least your *special* element. Please discuss this in the forums if it isn't clear because I'll know how to revise this paragraph better in the next iteration.

One final detail before you can get started on the coding: The Queue class you implement must be a template class. If you're not familiar with templating in c++, this would be a good time to hit your reference material (or simply clarify all your doubts in our [subreddit](#)). Here's the TL;DR version:

If you created two different Stacks (an int stack and a string stack) in the elephant quest (BLUE) you might have wondered at the amount of code you duplicated. You were probably already thinking "Gee! I have to copy my `Int_Stack` code into a new file and replace all occurrences of `'int'` with `'string'` ... I wonder if this could be automated..."

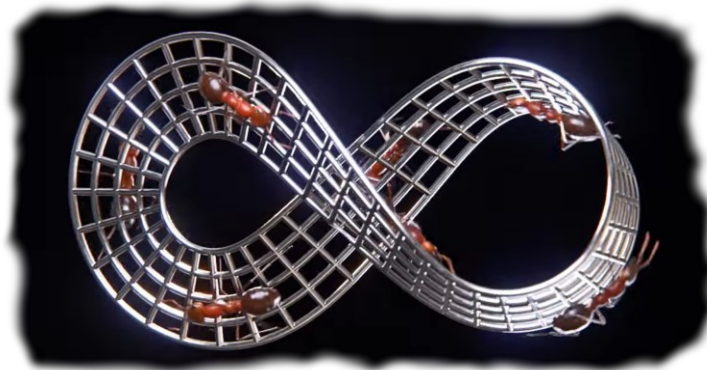
That automation is exactly what templating gives you. Now you could create *template* code for the compiler and tell it - "Here's the template. If someone wants a stack of a particular type, just plug in that type anywhere you see `T` and you'll get their source code".

In our example,

```
class Stack<T> { ... }
```

will make the compiler auto-generate a brand new Stack class depending on the value of `T`, the template parameter.

The queue you implement in this quest will be a template class.



Starter code

Because of the way in which most compilers process template code, you may have to include the entire class implementation within the header file.

Here is the incomplete listing of your Queue class and its implementation (Queue.h).

```
// TODO: Fill in as necessary

template <typename T>
class Queue {
private:
    std::vector<T> _data;
    size_t _head, _tail;
    static T _sentinel;

public:
    static const int MAX_DISP_ELEMS = 100;

    Queue(size_t size);

    static void set_sentinel(const T& elem) { _sentinel = elem; }
    static T get_sentinel() { return _sentinel; }

    bool is_empty() const;
    size_t size() const;
    void resize(size_t size);

    const T& peek() const;
    bool dequeue();
    bool enqueue(const T& elem);

    std::string to_string(size_t limit = MAX_DISP_ELEMS) const;

    friend class Tests; // Don't remove this line
};
template <typename T> T Queue<T>::_sentinel = T();

template <typename T>
Queue<T>::Queue(size_t size) {
    // TODO
}

// TODO - Fill in the missing implementations. Experiment with the aesthetics by moving
// some of these implementations inline (into your class def above). See which gives
// you more readable code.
```

This queue has a sentinel. When `peek()` is invoked on an empty queue, you must return the sentinel instead of throwing an exception.

What about the value of the sentinel? Well, there's no way you can tell what it is, because its type is only known at compile time from your actual template parameter. So you simply provide a facility to allow the user of your class to set the sentinel to whatever value they deem is illegal in their set of elements.

If I use your Queue to create my own queue of integers, I might instantiate it thus:

```
...  
Queue<int> my_queue(100);  
Queue<int>::set_sentinel(0);  
...  
if (my_queue.peek() == 0) {
```

Your first miniquiest - Constructor

Implement:

```
template <typename T> Queue<T>::Queue(size_t size);
```

The constructor needs to size the `_data` element correctly, and set initial values for the `_head` and `_tail` members. It doesn't matter that you set them to any specific values. Only that the values are consistent across multiple queue operations.

The constructor is not the place to set the sentinel, which is a static member. In fact, only the user of your Queue class is responsible for setting the sentinel. You don't have to worry about it at all.

Your second miniquiest - Enqueue

Implement:

```
template <typename T>  
bool Queue<T>::enqueue(const T& elem);
```

If the queue is not already full, insert a copy of the given element into the end of the queue and return true. Otherwise (if full) return false.

Your third miniquiest - Dequeue

Implement:

```
template <typename T> bool Queue<T>::dequeue();
```

If the queue is not empty, remove the (front) element and return true. Otherwise (if empty) return false.

Where is this dequeued (popped) element? It's not returned to you.

That's correct. Please discuss possible reasons why a programmer might choose to make methods like `Stack::pop()` or `Queue::dequeue()` *not* return the removed element.



Your fourth miniquest - Peek

Implement:

```
template <typename T> const T& Queue<T>::peek() const;
```

Return a copy of the front element in the queue (without changing it). Why do we need it? Or... do we need it?

Your fifth miniquest - Is Empty

Implement:

```
template <typename T> bool Queue<T>::is_empty() const;
```

Please check in our subreddit or ask if you don't know what to do here.

Your sixth miniquest - Resize

Implement:

```
template <typename T> void Queue<T>::resize(size_t size);
```

Whoo! Biggie.

I don't know about you, but the best way I've found to [resize](#)¹⁹ a queue like this one is to create a brand new queue and dequeue everyone off the old queue and enqueue them in the new queue, in order.

See if you can find a cheaper way to do the same thing. Remember that if the old queue has more elements than can fit in the new one, some of the latecomers may have to be booted. Yeah. That's too bad.

¹⁹ Here, size means capacity, not the number of items actually in the queue. A queue of size N can contain a maximum of N items. But it may contain fewer.

Your seventh miniquest - Popalot

Implement the global (non-instance) method:

```
template <typename T>
    void popalot(Queue<T>& q);
```

Thought I'd give you guys another easy freebie. Just implement a global scope template method (not instance or class method - what's the difference?) with the above signature. All it has to do is to clear the queue by emptying it.

Do it however you want. But the queue I give you should be empty when I get it back.

Your eighth miniquest - To string

As usual, `to_string()` is all about attention to detail.

The two examples below hopefully shed enough light on how the queue is to be serialized.²⁰

Implement:

```
template <typename T> string Queue<T>::to_string(size_t lim) const;
```

This method should return a string that looks like in Figure 2:

```
# Queue - size = {SIZE reported by your size() method}↵
data : {_data[_head]} {_data[...]} {etc. up to limit}↵
```

Figure 2 - output of `to_string(size_t limit)`

The enter key character (↵) stands for a single newline. Portions between curly braces (also in red) must be replaced by you. There are spaces where they're obviously there in the above pic. There are none where you wouldn't put them.

A more concrete example (Figure 3):

```
# Queue - size = 2
data : four five
```

Figure 3: A Queue of size 2, with the items "four" and "five"



²⁰ Thanks to Paul Hayter for help in defining the output format.

The line that starts with *data* should list the items in the queue. The rest of the line after the colon may be empty. If there are more than `limit` items, you must print the first `limit` items followed by a space and a single token of 3 periods (`...`) in place of everything else.

Finally, note that *data*, above, denotes the data in the queue from the user's perspective. Not the queue's underlying `_data` element (See Figure 4).

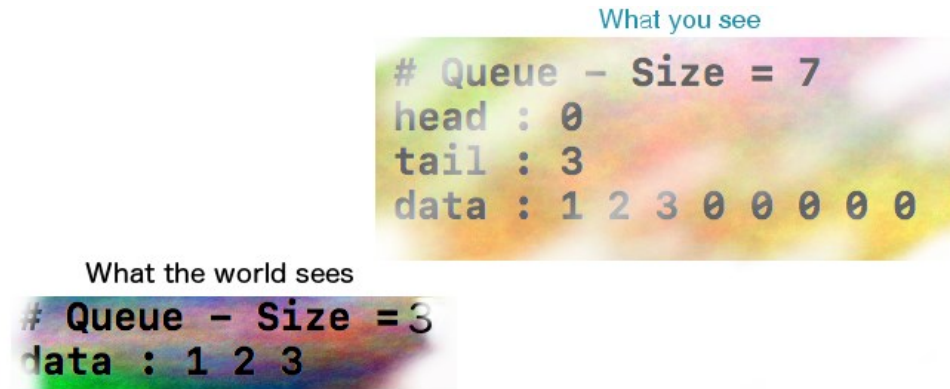


Figure 4 - What the world sees (output of `to_string`), and what you see

Your ninth miniquest - Queue of Objects

You don't have to do anything special here. All your previous miniquests were with integers. If your templating works as expected, it should succeed on non-integer types as well. I'll test that bit here and you get a freebie reward just for doing things a particular way. Cool!

How long is a well written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in about 150 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary. You can use the provided starter code as one example of acceptable code density.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret code for this quest in the box.
3. Drag and drop your `source` files into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).

Happy Questin',

&





*photo by Sasikumar
of Vellore*

*Our lives and times have ever made
Just footnotes²¹ to a tardigrade*



²¹*Nebuloid-Gamma-Cluster TG34765.14 - A galaxy of standard configuration (like the Milky Way). Existed from 9123E123 to 9123E129 qsecs.*

The Prefix Tree

In under 200 lines of readable, commented code, (around 25-50 chars per line, I think), you can implement a brand new kind of exciting data structure. It is a data structure that is very useful to have in your inventory because it has uses that other data structures have yet to fill efficiently (and as elegantly).

It goes by many names in many places, including *prefix trees* and retrieval trees.

We'll just use the word *trie* in this spec, which, I believe, comes from the middle of "retrieve".



We tries. Only then we will succeeds

Here is an interesting problem. Suppose we have a lot of words, and want to find all words that share the same prefix, what's a good way to do it?

If you aren't familiar with this problem, consider yourself lucky. Take a break now, go for a walk and think about it before resuming.

One way is to create a hash-table (a dictionary, which we'll look at next quarter) which maps every possible prefix into a set of strings that can follow it. But that would be awfully wasteful, yes?

The trie presents another (I think, elegant) way. By now you're already familiar with the idea of using vector indices to encode data.²² Consider the way you stored the cache in Quest 2 (Hanoi), for instance.

In a trie, the data you want to store is not stored explicitly under a data label, but is instead implicit in the presence or absence of a particular child.

You can think of the trie as an abstraction over a general tree (the koala quest) with a fixed number of children. In this spec, I'm using the vector of children technique to represent this particular kind of general tree.

We'll apply yet another perspective shift to this representation and assume that the data element of a child node is equal to the index into the parent node's vector that led to it.

²² There is a subtle detail to appreciate here: Using *indices* to store data is fundamentally different from using the cells themselves. In the latter (common) case, the cell holds the data. In the former, it's implicit in one of many paths leading from that cell to another. This is like putting labels on the edges rather than nodes in a graph.

Then:

1. Every node in the trie can be reached from the root by following a prefix, P, of one or more words encoded in the trie.²³
2. If the k'th element of the vector of node pointers contained in a node is NULL, then *P appended with the character whose ascii value is k* is NOT a prefix of any word encoded in the trie.
3. If it is not NULL, then it points to a unique node that can be reached by following a prefix constructed by appending to P the character whose ascii value is k.
4. A valid whole sentence ends in the 0th element of its terminal node's vector.

Check out Figure 1, in which the vector is 256 elements large. This allows that trie to encode any ASCII string.

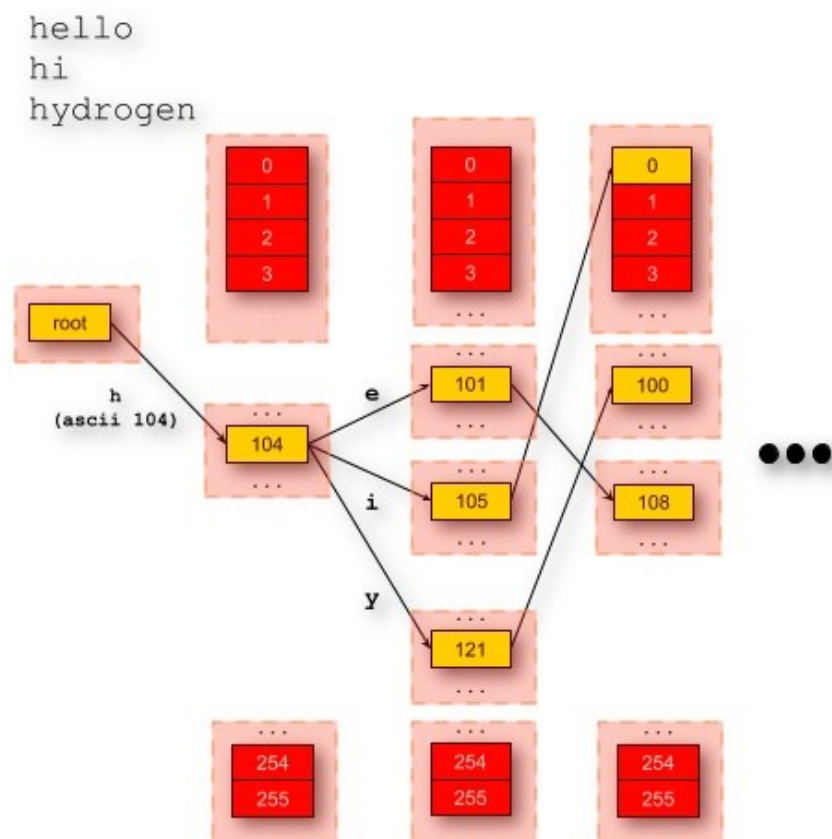


Figure 1: A Trie. Red cells are not live (null pointers). Each cell is within a vector contained in a trie node. What kind of storage overhead or gain are we talking about? (Compare to the next best working representation you can think of - e.g. a dictionary). What's the trade-off?

²³ Assume that by *following* I mean that you traverse an ordered sequence of nodes from the root corresponding to a valid prefix.

Starter code

Again, this is a relatively easy quest once you get past the *pig picture*.²⁴

So the quest master said "No can do".

But he let me snoop around and take photos with an old camera I found. Here's a fuzzy photo of the class def.

```
class Trie {
private:
    struct Node {
        std::vector<Trie::Node *> next;

        ~Node();

        void insert(std::string s);
        const Node *traverse(std::string s) const;
        bool lookup(std::string s) const;
        size_t get_completions(std::vector<std::string>& completions, size_t limit) const;
    } *_root;

    // Private Trie:: helper. Returns the interior node traversing s from _root
    const Node *traverse(std::string s) const { return _root->traverse(s); }

public:
    Trie();
    ~Trie();

    void insert(std::string s) { _root->insert(s); }
    bool lookup(std::string s) const { return _root->lookup(s); }
    size_t get_completions(std::string s, std::vector<std::string>& completions, size_t limit) const;
    size_t trie_sort(std::vector<std::string>& vec) const;

    std::string to_string(size_t n) const;
    std::ostream& operator<<(std::ostream& os) { return os << to_string(100); }

    friend class Tests; // Don't remove
};
```

More details can be found in the following miniquest descriptions.



Your first miniquest - Make a trie

Implement the trie constructor:

```
Trie::Trie()
```

²⁴ pig pictures stand in your line of sight of the big picture.

All this needs to do is allocate a brand new `_root` node and return.

And you get a reward for that? Hmm...

Your second miniquest - Insert

Implement the private `Node` method:

```
void Trie::Node::insert(string s)
```

which is needed to complete the `Trie::insert()` method. It should insert the given string into the trie at the current node (duh!). But it must be iterative. How might I be able to tell in my testing code? You may decide to recurse and sneak through. But it'll be a worse implementation. (Why?)

Tip: Exploit the fact that, in c++, the underlying c-style representation of any string contains a NUL²⁵ delimited sequence of the ASCII values of the characters of the string. It fits the bill PERFECTLY because it lets us directly use the NUL character (ascii value 0) as the index of the node into which the very last character of the string should branch. Can it get any better?

Here is a fuzzy photo of the reference implementation. Of course, it doesn't have to look like this. But it may help you come up with yours:



iteratively insert the given string into the trie starting at the current node

```
Trie::Node::insert(string s) {
Trie::Node *curr = this;
for (const char *str = s.c_str(); *str; str++) {
    char ch = *str;
    if ((size_t) ch >= curr->next.size())
        curr->next.resize(ch+1);

    if (curr->next[ch] != nullptr)
        curr = curr->next[ch];
    else
        curr->next[ch] = new Trie::Node;
}

// the \0
if (curr->next[0] == nullptr)
    curr->next[0] = new Trie::Node;

if (curr->next[0] == nullptr)
    curr->next[0] = new Trie::Node;
```

At a high level, you want to start at the head of your string, jumping to your next nodes which are got by indexing into your `_next` vectors by the character in the string you must move over to get there. At the end of the string (at the node indexed into from the last character), jump into the 0th child.

Here's a useful tip: Watch out for duplicates. Unfortunately, you'll only know what I mean after you get bitten. But this tip might help you recognize the bite sooner before it ruins your experience.

²⁵ NUL is usually taken to mean the NUL character (byte) whose value is 00000000. Sometimes you'll find it coded as a single (escaped) character, `'\0'`. In C, this is a "formal" invalid character and used as a sentinel in string processing algorithms. Don't confuse it with `NULL`. That is the memory address 0 (null pointer).

Your third miniquest - Traverse

Implement the private `Node` helper method

```
const Node *Trie::Node::traverse(string s)
```

which is needed to complete `Trie::traverse(string s)`.

Uh Oh! No photo here.

Thankfully, the logic is very similar to `insert()`.

It should return a pointer to the node reached by traversing the trie from the root, one node per character in the supplied string `s`, in order.

As an example, traversing the trie for "hydro" starting at the root node in Figure 1 should return a pointer to the Trie node you would eventually end up at by entering it on the ascii value of the last letter 'o'.

If the string (or its superstring) cannot be located in the trie, `traverse()` must return a null pointer. You can return a null pointer as soon as you know for sure a prefix cannot be found in the trie. What's the earliest in your search when you can know this for sure?

You'll sure be glad you implemented this method when it comes time to implement `lookup()`.

Your fourth miniquest - Lookup

"Hey! did I just hear my name?"

Implement the private `Node` method

```
bool Trie::Node::lookup(string s)
```

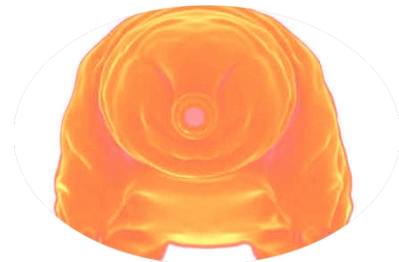
which is needed to complete `Trie::lookup()`.

It should return `true` if the string `s` is *wholly* contained within the trie. This means its last encoded (but not visible) character must be the sentinel (`\0`).

Leverage `traverse()` to get to the internal node that ends at the last visible character of the string. This node should encode the sentinel as a valid continuation. That is, `next[0]` at this node should be non-null.

As an example in Figure 1, the string "hydr" will end in a node whose `next[0]` will be null (since "hydr" is not a valid word. However, "hi" will end in a node whose `next[0]` will be non-null.

If `lookup("my-string")` succeeds, then it means that "my-string\0" is a valid prefix contained in the trie.



Your fifth miniquest – Destructor

Implement the destructors:

```
Trie::Node::~~Node()

Trie::~~Trie()
```

The trie destructor only has to delete `_root` and let the node destructor do everything else. It's ok for the node destructor to be recursive... in many use cases of a trie. (Why?)

All that the destructor needs to do is to delete each of the allocated children. There's nothing else to do. But if your bookkeeping is not up to snuff or your implementation is confusing, you may end up with nasty memory related errors.

Side note on memory errors

Many times you can get "lucky" and receive a fatal segmentation violation error when you read from memory that doesn't belong to you (or memory that has been freed by you). But luck is a fickle companion for computer programs. You can't rely on it.

How then can you minimize the likelihood of segfaults?

Although asking the run-time system to monitor ALL of your unallocated heap for illegal access is unfeasible, it is indeed possible, and a wise investment by the run-time system, to monitor a single location continuously. This location is memory address 0, the null pointer (`nullptr`).

You can exploit this fact to come up with a useful design technique:

1. In the first iteration of your program (or product), always explicitly set all deleted pointers to the value NULL.
2. This will guarantee that your program will crash AS QUICKLY AS POSSIBLE after an illegal access, thereby GREATLY SIMPLIFYING your debugging.
3. Users of your product will be quick to report crashes, and their reports will be more accurately representative of the bug in your code.
4. Release millions (figuratively) of copies of your app (instrumented thus). Get many bug reports and fix as many of them as possible.
5. When the bug reports slow down to a trickle, revisit your code and pull out the null assignments, getting a small performance improvement to reward your faithful users.

While we're on the subject of destructors, here is a worthwhile exercise to try in your spare time. Instrument the destructor at the appropriate points with the following lines:

- `static int r_depth = 0;`
- `cout << "~Node() Recursion depth " << ++r_depth << endl;`
- `cout << "~Node() Descending from depth " << r_depth << endl;`
- `cout << "~Node() Back to depth " << --r_depth << endl;`

Look at the output when you delete a trie containing many sentences. Share your insights on our [subreddit](#). Did it help you in making your program leak-proof?

Make sure you remove the instrumentation before you submit your code into the questing site.

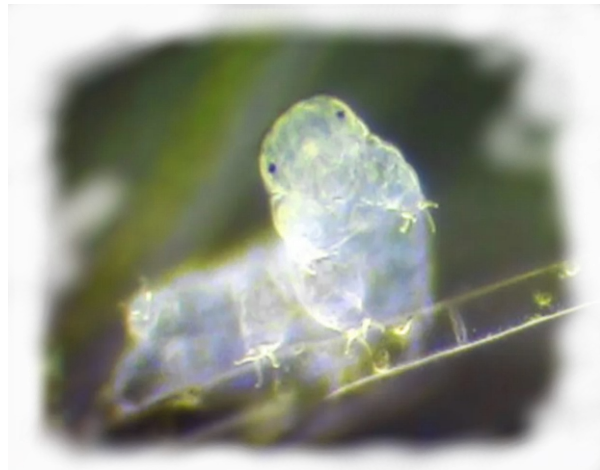
Your sixth miniquest - Get completions at node

Implement the private Node method:

```
size_t Trie::Node::get_completions(  
    vector<string> &completions, size_t limit  
) const
```

Clear the vector, `completions`, and fill it with up to `limit` strings that are possible completions of the empty string at `this` node. Essentially, this is the same as a *breadth-first traversal*.

It involves a number of subtleties. Read on. Some of it may not make sense right now. That's ok. I suggest that you read it anyway until it seems to not make any sense without a great deal of effort. Then try to rewind all the way back to the description of this miniquest and say *"Screw this spec. I'm just gonna do this by myself. It sure looks easier than trying to understand this gibberish."*



If you end up solving the miniquest your way, collect the reward and move on. Otherwise, read the section below and you'll find it makes a whole lot more sense even though your hardcopy would have been unchanged!

You can achieve a breadth first traversal by *processing* the nodes of any general tree using the strategy below. You didn't get to do this in the Koala quest. But you get to do it here:

1. Start with a queue that only has the start node
2. Until the queue becomes empty:
 - a. Pop the front node.
 - b. Add all the children of that node to the end of the queue
 - c. Process the node's data element (if appropriate)

(What happens if you had accidentally typed "stack" instead of "queue"?)

If you do this with a tree, you'll find that you traverse it in level order left-to-right.

All well and good if each node can tell you what its own data element is. But what do we do in situations like ours where a node's data element is implicitly hidden in the index of the parent's branch that led into it? This information is lost in the recursive invocation in which the `"this"` object points to the child and not its parent.

The way I handle it in this quest is by pushing packages of two things on the queue instead of single nodes. These packages will each contain the node to be processed, but also, importantly, the much needed history information of the prefix that led into it, which I've called `partial`.

BTW, a `struct` in `c++` is simply a class in which every member is public by default.

At each node, consider all possible continuations (every ASCII char) and handle each of the following 3 cases for each child in sequence:

1. The character that leads into that child is NUL (`\0`)
2. The character has no continuation out of the current node
3. The character is a valid continuation out of the current node



Here is the starter code for `Trie::Node::get_completions()`.

```
size_t Trie::Node::get_completions(vector<string> &completions, size_t limit) const {
    struct Continuation {
        const Trie::Node *node;
        string partial;
        Continuation(const Node *p, string s) : node(p), partial(s) {}
    };

    // All string descendants of this node are completions
    queue<Continuation> unprocessed_nodes;

    // TODO - Your code here

    while(!unprocessed_nodes.empty()) {
        Continuation cont = unprocessed_nodes.front();
        unprocessed_nodes.pop();

        // TODO - Your code here
    }

    return completions.size();
}
```

Your seventh miniquest - Get completions in Trie



Implement

```
size_t Trie::get_completions(  
    string s, vector<string> &completions, size_t limit  
) const
```

If you had managed to get to this miniquest, you'll find it's nothing at all. Simply traverse your trie's root on the given string, and then generate all possible completions of the empty string at that node.

Of course, if you can't traverse a trie with a given string, then you can only return a nothin' thing.

Your eighth miniquest - To string

Implement

```
string Trie::to_string(size_t limit)
```

No complex logic here. The returned string should contain, in order:

1. The line: "# Trie contents"
2. Up to `limit` entries from the list of all completions of the empty string ("") in the trie, at one per line. The completions must be obtained using your previously completed `get_completions()` method call.
3. If there are more entries left after having added `limit` entries, add the line "..."

Your ninth miniquest - Trie Sort

Eazy peazy points. Just for surviving thus far, I guess.

Implement:

```
size_t Trie::trie_sort(vector<string>& vec) const;
```

It should clear and suitably size `vec` to hold all the distinct strings in the trie. Then it should fill it with all the completions of the empty string (no limit). Finally, it should return the size of this `vec`.

If you look at the result you'll see the encoded strings ordered a particular way. What is this order? Can you think of any practical applications for such a permutation of a list? How long does it take (as a function of the number of input strings) to generate a trie-sorted sequence of `n` strings?

For extra fun, (if you're so inclined) [share](#) a Unix/Linux one-liner (not necessarily a single command) that will trie-sort an input list of strings.

How long is a well written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in about 200 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret code for this quest in the box.
3. Drag and drop your `source` files into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).



Happy Questin',

&

*Don't wanna be a wannabe
But be a bee - Jus' Simplee bee.*



J. Simplee Bee

Graphs



Yeah! Since this is the lastest quest of this level, it is also gonna be the funnest quest.

You get to implement a simple Graph data structure with a WHOLE LOT OF freedom around how you do it.

All you have to do in this quest is to make various shapes. I don't care how you make them as long as you make them correctly.

Pick and choose. Each shape is worth a certain number of rewards. Generally, the simpler ones are also cheaper and easier to get.

But before you get to it, here's the representation I'll be looking for:

```
class Graph {
protected:
    struct Edge {
        int _dst;
        std::string _tag;
        Edge(int dst = -1, std::string tag = "") : _dst(dst), _tag(tag) {}
    };
    std::vector<std::vector<Edge> > _nodes;

    // Suggested private helpers. Not tested.
    void add_edge(int src, int dst, std::string tag);
    std::string to_string() const;

public:
    void make_silly_snake();
    void make_mr_sticky();
    void make_driftin_dragonfly();
    void make_slinky_star();
    void make_kathy_da_grate();
    void make_dodos_in_space();
    void make_purty_pitcher();

    friend class Tests; // Don't remove this line.
};
```

We're keeping everything as simple as possible so you can focus on your visual art in this quest. Our Graph nodes only have numbers, not names. Edges from one node to another may have optional string tags you can stick on them.

This lets us model the graph as a collection of *nodes* which each contain a collection of *edges*.

This is it:

```
std::vector<std::vector<Edge> > g;
```

where the entire graph is simply the collection of nodes, called `g` above. In this case, we're using `vectors` for collections.

Important: The graph is NOT a 2D matrix. It is a vector of nodes, where each node contains a vector of edges. Nodes that don't have edges leaving them will be represented by the cells that hold empty vectors. Put another way, the edge from node A to node B can be found in the vector located at `g[A]`. But that edge is NOT `g[A][B]`. Instead you must scan this inner vector to find the edge with the destination of interest.

If you think about it, the `Edge` object is already an added level of complexity that serves only a cosmetic purpose. We could have made the inner vector a vector of ints, where the ints stand for the destination nodes, and thereby kept things much simpler (We'd still get our shapes).

The reason I have it is to give me a way to stick a label on each edge. I thought it'd be cool. I'll revisit this decision later. For now, it's needed to pass this quest.

Note the golden comment in the starter code. The suggested helper methods are NOT tested. The only methods you need to implement are the public ones, described below in the miniquests. Feel free to implement or not implement the suggested helpers, or implement as many more as you want.

The only thing the miniquests will be looking for is whether your `Graph` object looks like what it's looking for (in hyper-space, of course. No serialization required).

See I told you it's gonna be fun.



Your first optional miniquest - The Silly Snake

Implement:

```
void Graph::make_silly_snake();
```

When invoked, it should clear itself and be reborn as a silly snake. See Figure 1.

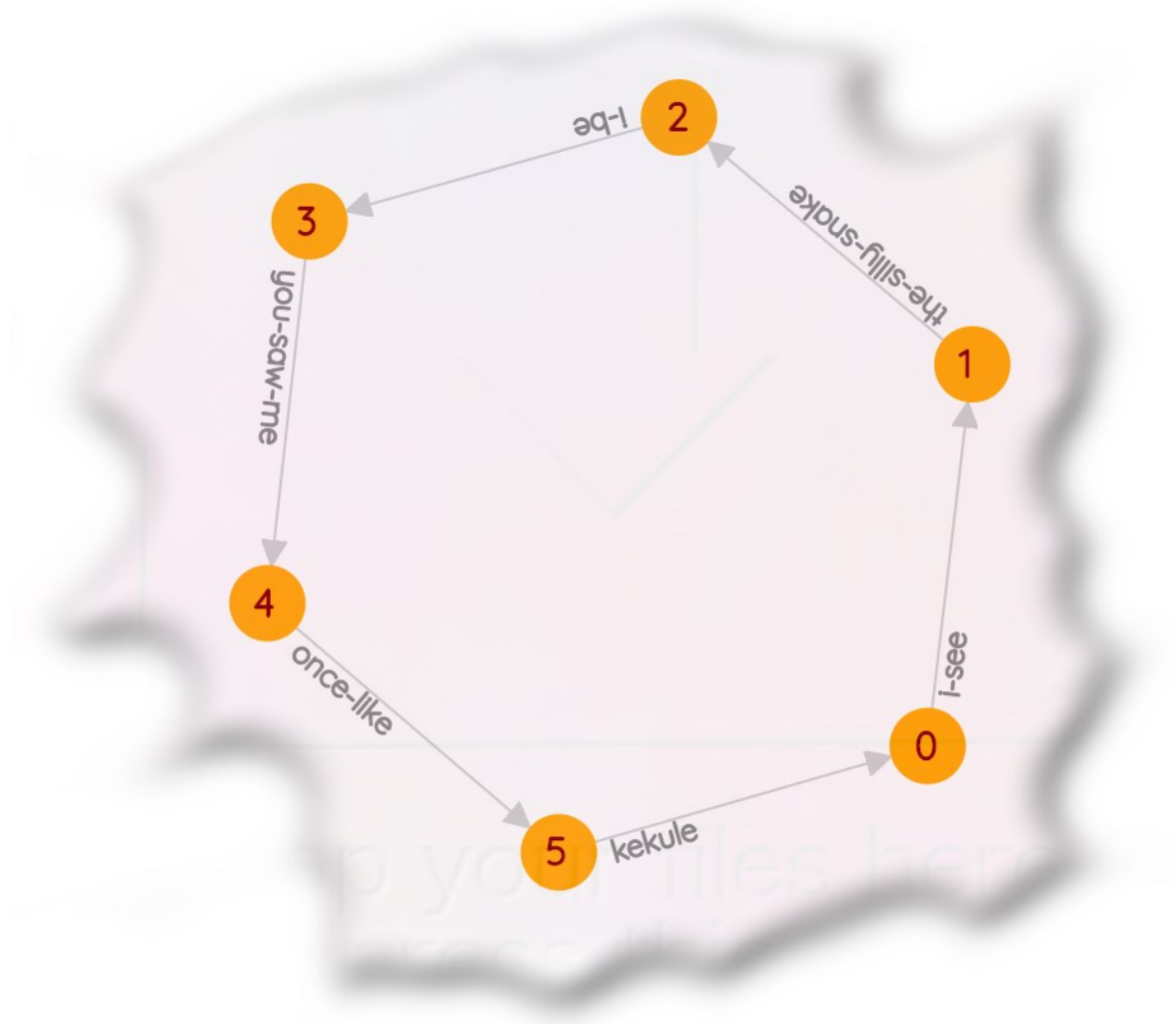


Figure 1: Silly Snake

Make sure to get the edge labels correct.

Your second optional miniquest - Mr Sticky

Implement:

```
void Graph::make_mr_sticky();
```

When invoked, it should clear itself and be reborn as a mr sticky. See Figure 2.

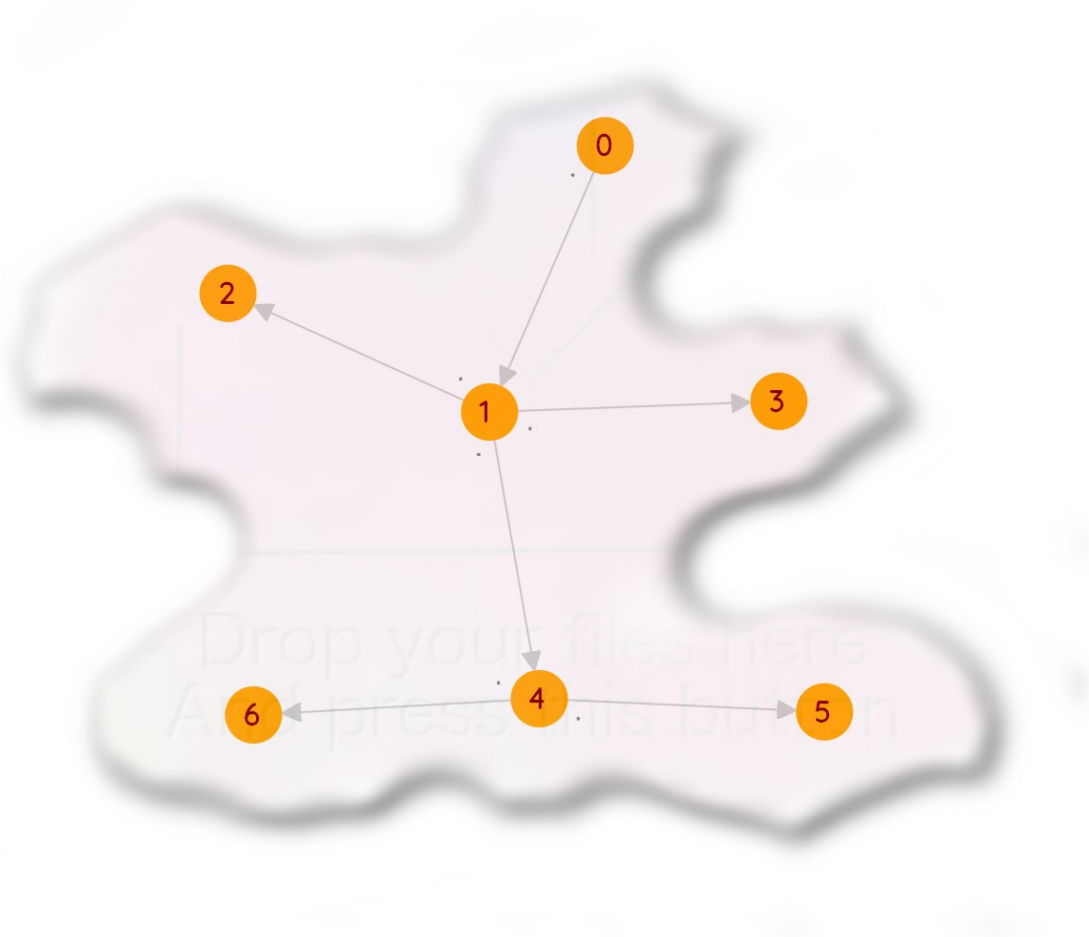


Figure 2: Happy Mr Sticky

What are the edge labels here?

Hmmm... I can't quite tell. Is that a period?

Or a hyphen?

Oh well, there's only 64 possibilities (or is it 729?)

Your third optional miniquest - The Driftin Dragonfly

Implement:

```
void Graph::make_driftin_dragonfly();
```

When invoked, it should clear itself and be reborn as a driftin dragonfly. See Figure 3.

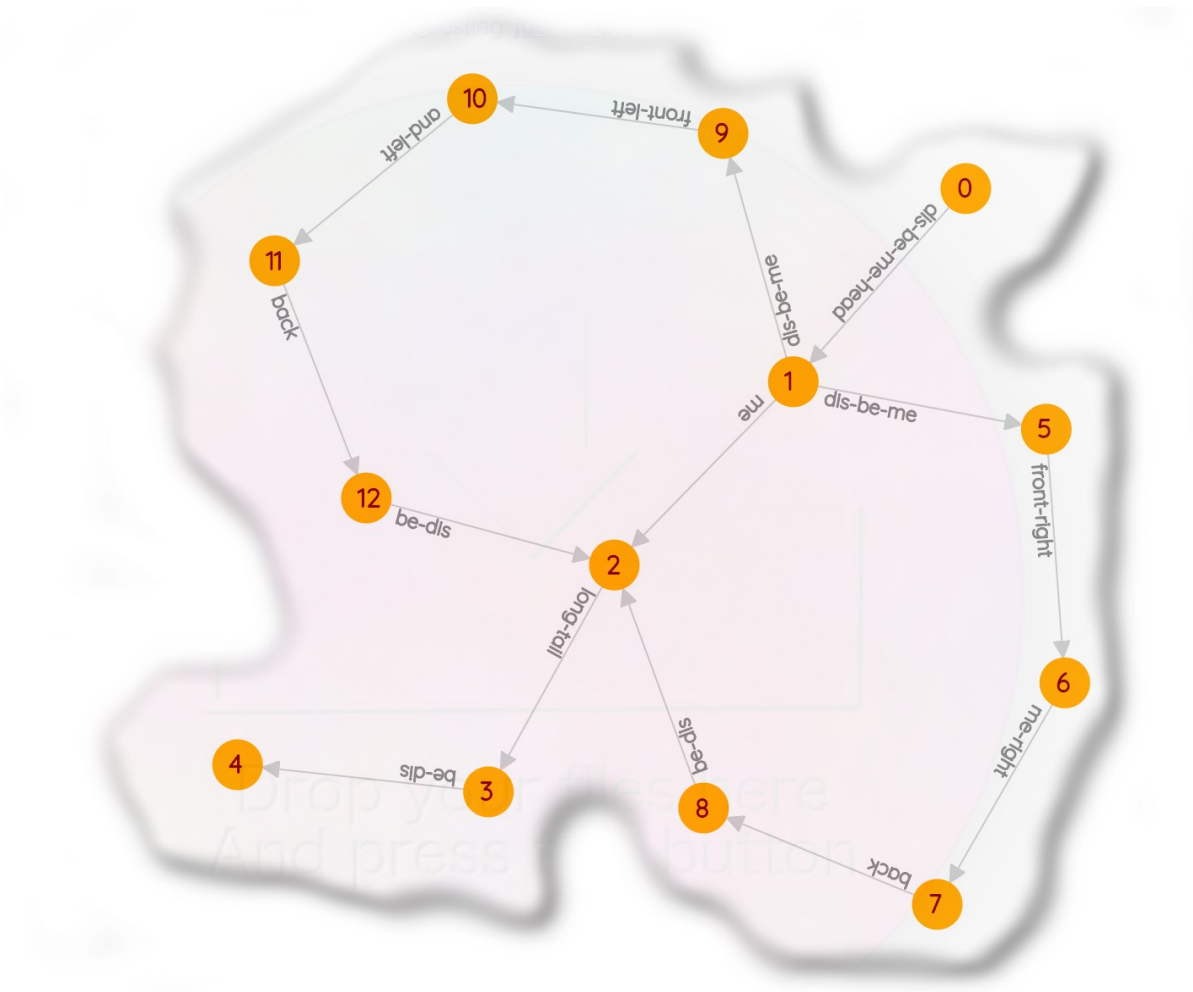


Figure 3: The Driftin Dragonfly

Again, make sure you get the edge labels right, or your dragon don't fly.

Your fourth optional miniquest - The Slinky Star

Implement:

```
void Graph::make_slinky_star();
```

When invoked, it should clear itself and be reborn as a slinky star. See Figure 4.

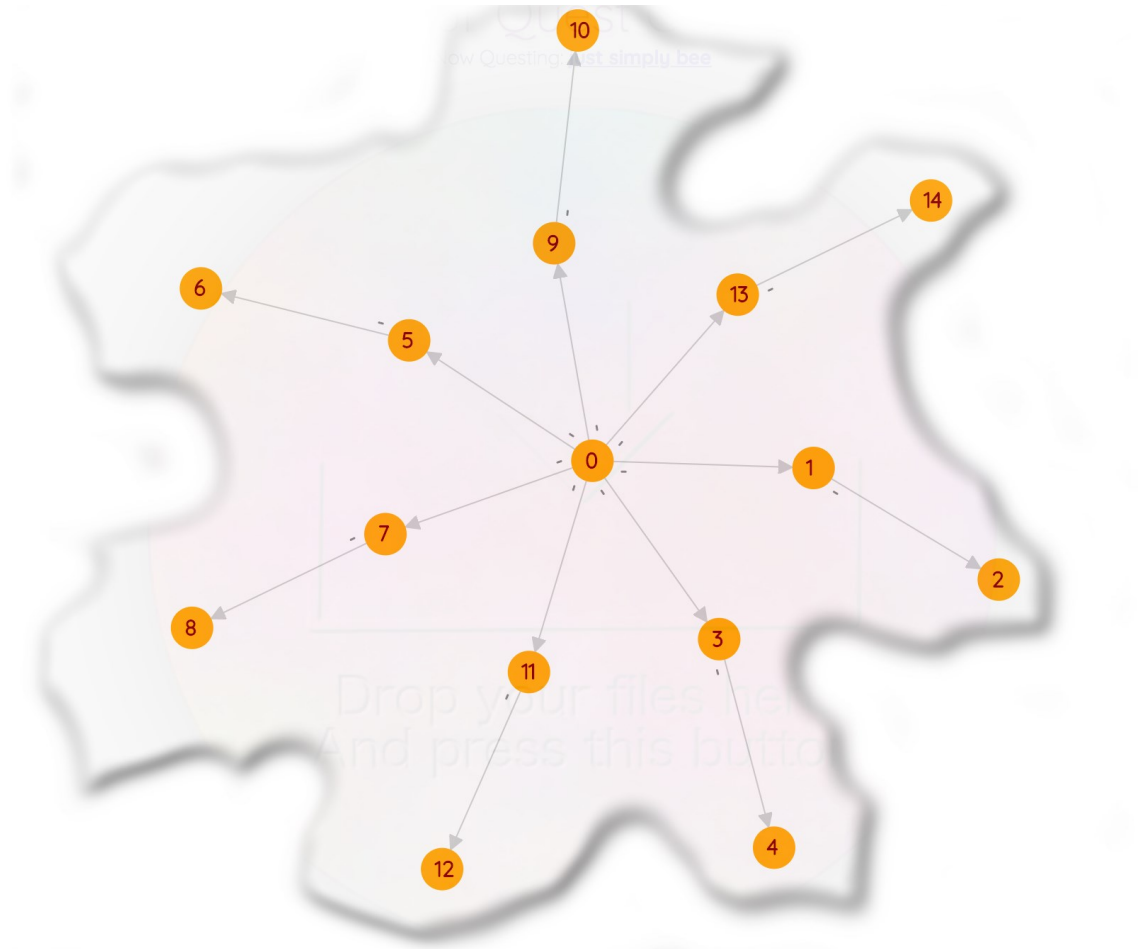


Figure 4: The Slinky Star

A star you are. A star you be,



Your fifth optional miniquest - Kathy da grate

Implement:

```
void Graph::make_kathy_da_grate();
```

When invoked, it should clear itself and be reborn as Kathy da Grate. See Figure 5.

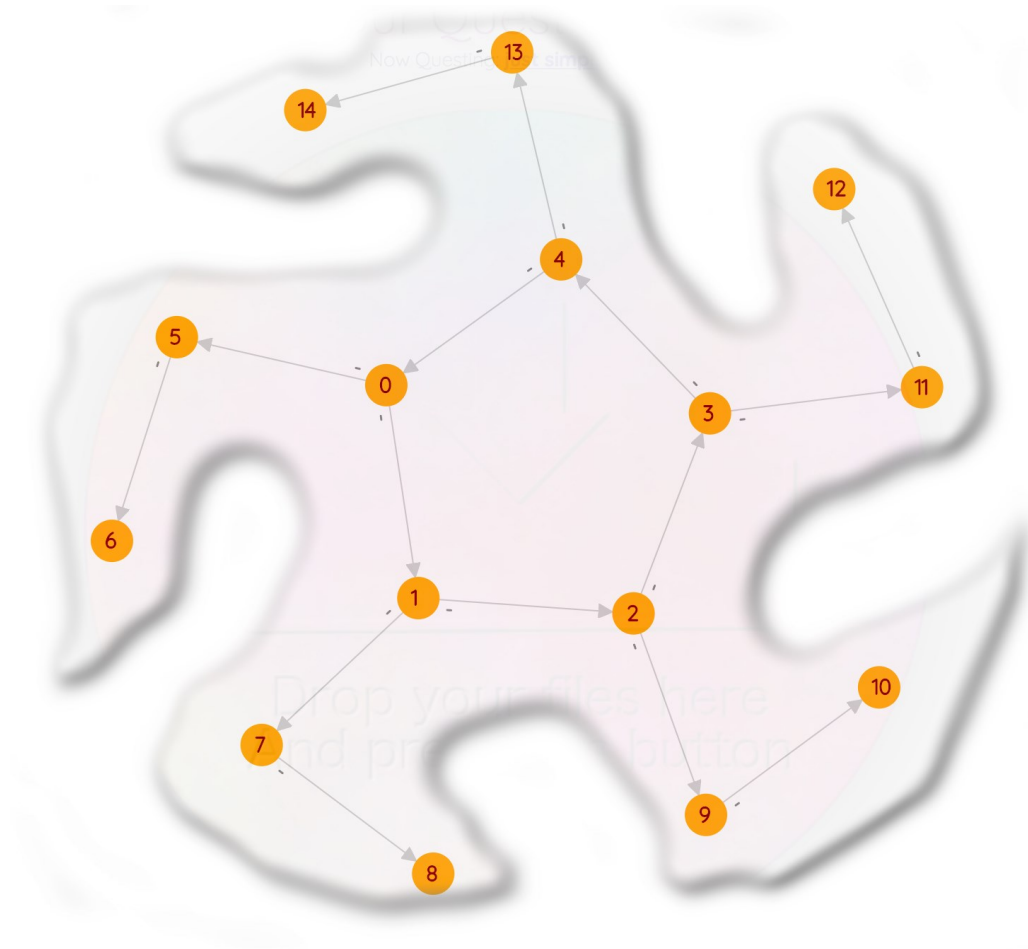


Figure 5: Kathy da Grate



Your sixth optional miniquest - Dodos in space

Implement:

```
void Graph::make_dodos_in_space();
```

When invoked, it should clear itself and be reborn as 25 dodos in space.

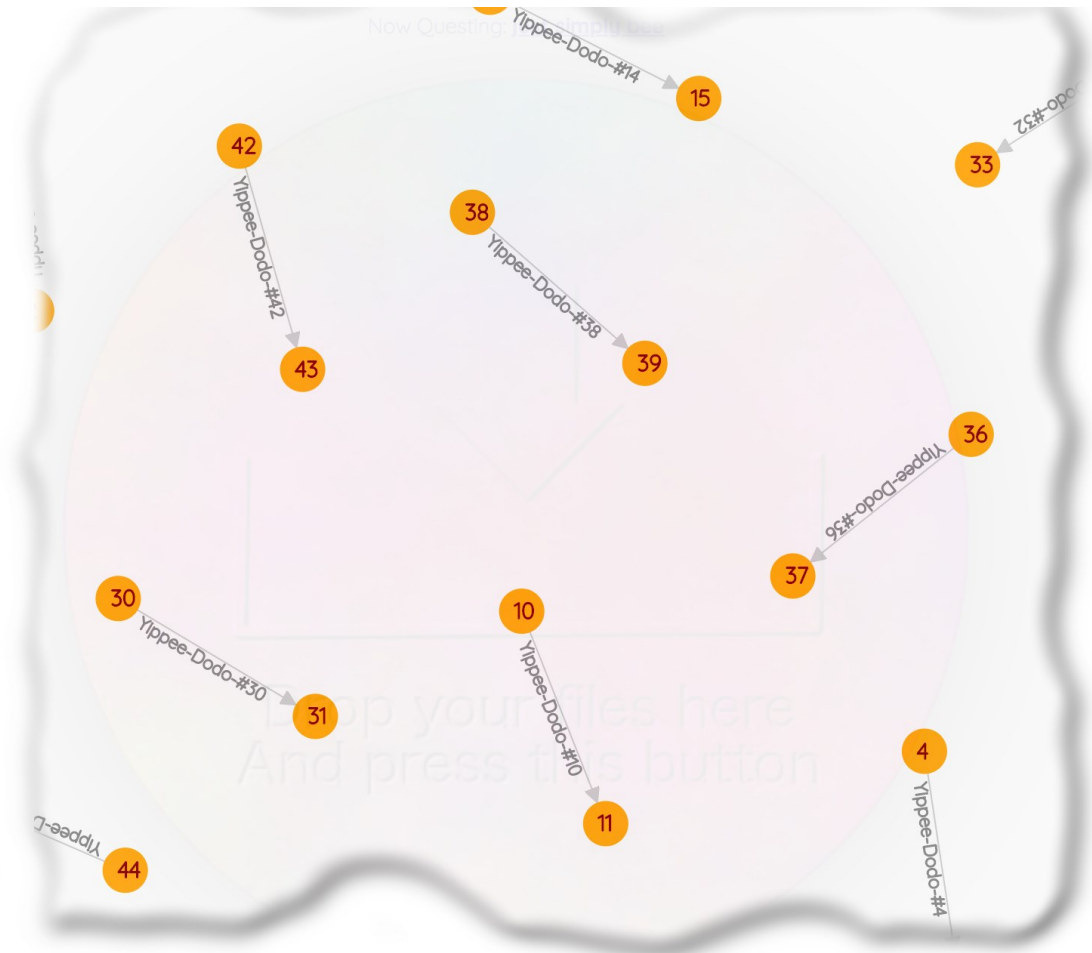


Figure 6: Dodos in space (Warning: Only a representation of reality. Verify your own numbering)

Each dodo is an even numbered node which points to the next odd numbered node via an edge that is tagged 'Yippee-Dodo-#' post-fixed with the dodo's node number.

There should be 25 free dodos in your graph. See Figure 6.

The first dodo is numbered #0 and the rest are sequentially numbered after that.

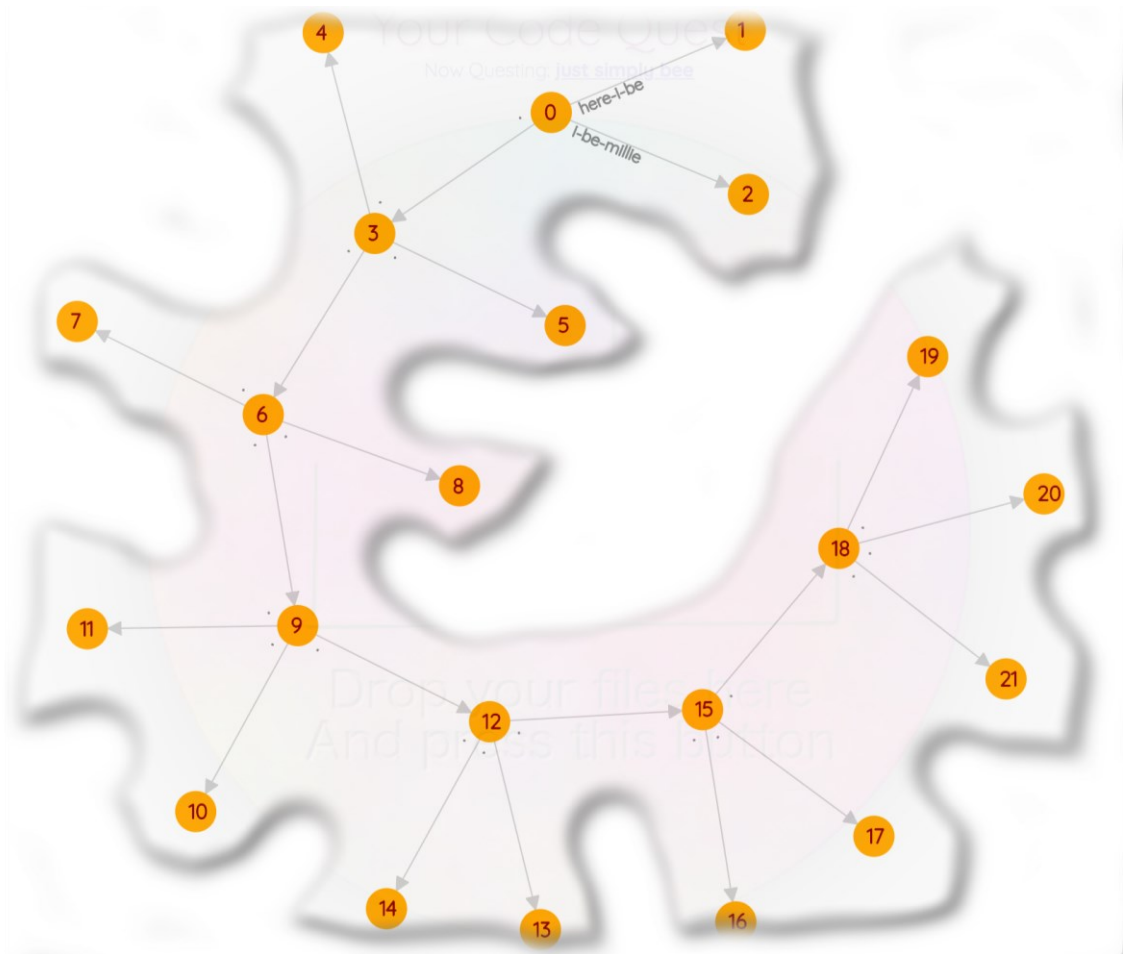
Your seventh optional miniquest - Purty Pitcher

Implement

```
void Graph::make_purty_pitcher();
```

It can make any graph you want. If it's purty nuff, upvotes on our [subreddit](#) earn kudos.

Here's mine. Yours don't have to be like dis.



Knock yerselves out.

Submission

When you think you're happy with your code and it passes all your own tests. And then...

1. head right over to over to <https://quests.nonlinearmedia.org>
2. enter the secret code for this quest in the box.
3. drag and drop your source files (Graph.*) into the button and press it.
4. wait for me to complete my tests and report back (usually a minute or less).

Byeee!

Thank you for enjoying the flight with me. I look forward to flying with you again in RED.

Happy Questin',

&



*Gattaca loves a power set. Except to enumerate
So there. Don't becurse it. You had better just honor it.*



A goldfish called Gattaca

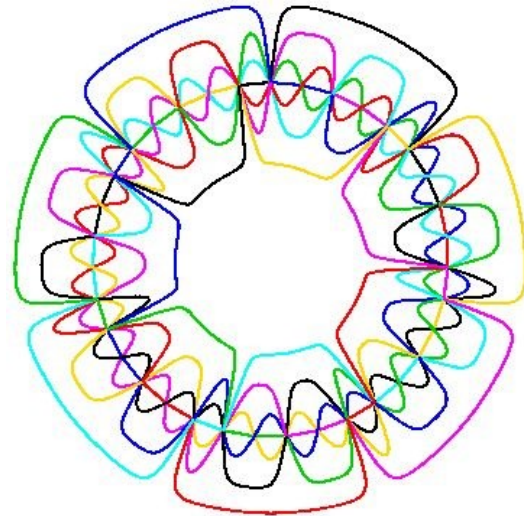
The Subset Sum Problem

previously by Michael Locciff

Here is an interesting puzzle: A radio show host wants to select a group of tunes or commercials from a set whose total running time is as close to the duration of the show or commercial break (e.g. three hour FM music show, or a two minute commercial break) without going over the time allotment. How does he pick items to play?

You can formulate this problem in a way that lets us approach a solution programmatically (why only *approach*, and why *a* solution?)

Consider any set of positive²⁶ integers, $S = \{x_1, x_2, \dots, x_n\}$. Given a target integer t , we want to identify the subset of S whose sum is as large as possible, but not larger than t .



The Algorithm

Here is what you can call a brute-force method:

1. Form all possible subsets (also called the "power set of S "). For example, if $S = \{4, 1, 7\}$, the power set of S is the set containing $\{\}$ (the empty set), $\{4\}$, $\{1\}$, $\{7\}$, $\{4, 1\}$, $\{1, 7\}$, $\{4, 7\}$, and $\{4, 1, 7\}$.
2. Find a subset whose members add up to the largest number possible $\leq t$.

It always yields a solution (although it may not be unique - there may be many subsets that add to the same maximal number $\leq t$). It is a very costly technique in terms of timing because it has what we call exponential time complexity (we'll learn what this means soon). The problem is inherently exponential, so there is not much we can do about that.

Nonetheless, we can find improvements to brute-force algorithms by trading off some amount of accuracy and/or completeness for a large improvement in speed.²⁷ The algorithm you will implement in this quest makes a modest efficiency improvement. It will also explicitly state how you iterate through the power set of S . Specifically, it gives you a heuristic with which you can prune your search space by shutting doors early on that you know to not lead to your solution.

²⁶ Really, peeps? Is this necessary? Can't I just say *integers*, or *non-negative integers*?

²⁷ Which of these two are we giving up here? Or is it both or neither?

Here's how I think about this problem:

Let's say that I can make N subsets out of n items. If I add just one more item, then I can make a whole set of N new subsets (How? I simply form new sets by adding this new item to every set I already have). Each time I complete that, I double the number of sets I have.

So if I start with no sets at all, and then add the empty set, and then after that I want to add the first item, I have two possible subsets: The empty set and the set with just the new item. Then after the second item comes along, I can form two more sets in the same way, bringing the total to 4. Then 8, and 16, and so on. It's easy to see how this is simply the value 2^n for n items. That's the number of subsets I can form from n items. This is also the size of the *power set* of a set.

Now the cool thing is that we can follow this exact technique to iteratively generate our subsets. Start off with an empty set and, iterating through the set of all items, add each item to every set you already have to generate a whole bunch of *new* sets. They're *new* in the sense that you haven't yet seen (or evaluated) any of them.

As you can see, this is a cool way to generate your subsets, but it does nothing for the running time of our search. It's still going to take time proportional to 2^n to visit all the candidate sets.

I think I can hear you say "This is really screaming out for an optimization - Skip elements larger than the target while iterating over the elements" - I think you'll find many more cool rules like that. But remember they are all just special cases of your overarching plan - *to discard unviable candidates as soon as you find them, and thereby prune the search space of all its descendants*.

How? One way is to reject all unpromising searches as soon as we know for sure.

"How do we know for sure?" you ask. Well, you simply keep track of the running total (sum of elements) of each set.



The moment you find a set whose total is greater than the target, you can immediately ignore all its *descendants* without even looking. By *descendants*, I mean every set of which this *unviable* set is a subset.

Suppose you identify one such subset early on... say when you have M items still left to process. That means you've effectively pruned out 2^M potential candidates without needing to explicitly test them. Imagine this - Just two quarters ago, you were mind-blown by the power of binary search over linear. But if you think about it, you'll see that's exactly what's going on here at various levels of intensity. Every time you find an unviable set, all its descendants are gone. Poof! Like that.

Clearly, it makes sense to prune sets early in the process (when M is close to N), right? In the best possible case, you encounter such descendant-killers early in the lineage of a set - when it's only a few items big. I wonder if there's something we can do to the input set to handle each descendant killer as early as possible. Hmm... Well, it looks like there's lots of juicy opportunities to ponder with this one.



Note that every singleton set which is unviable effectively cuts the remaining search space in half. Intuitively this makes sense because it is the same as saying that you first scan all items in linear time and discard elements greater than the target. If you discarded K items, you have thereby just reduced your search space by *up to* 2^K items²⁸ (how do I estimate this number?). So it seems like this strategy should on average cut the search space by a factor of somewhere between 0.5 and some small number (which should be $1/2^N$). Knowing this you can put some definite upper and lower bounds of performance on this algorithm in your mind: It is at least as efficient as linear search and at most as efficient as binary search. But we stay modest 'cuz we're talking about linear search over a potentially exponential number of items (when?). Hmm... Does that really qualify as linear search? Go figure.

High-Level Plan

Maintain a vector of viable candidates. Initially start out with just an empty set. Then gradually add viable sets to this list as you process each item from the master set in turn. At any time, if you find an exact match to the requested target value, you can output that set and end the search. If you don't find an exact match after exhausting the powerset, output a viable candidate with the greatest sum less than the target.

Implementation Specifics

Create a template class called `Set`. Your list of candidates is now `std::vector<Set>`. However, since your `Set` is a template class, I, as the user of your class, will decide what things I want sets of. But to keep things simple on your side, you only need to write the logic to handle sets of integers. As long as the integer operations required by you (e.g. addition) are supported with identical syntax by some other type, you can be blissfully unaware of its other details.

For example, it's up to me to make sure that if I use `SongEntry` as my template parameter, my `SongEntry` class behaves like an integer, in supporting the ability to be added to an integer to yield an integer.

Since your `Set` is going to be a template class, you'll submit a single source file, a file called `Set.h`. It's not a great deal of code, and probably just under a hundred lines. But you'll need to think through the problem carefully to write it.

The Representation

How do you represent a set internally?

At first thought it might seem like a simple problem. The trivial solution is to have each set be a collection (e.g. a vector) of objects.

²⁸ In the best case, I can eliminate half the *remaining* search space with each examination. This will result in a total of 2^K elements you can skip checking (How? It's simply the sum $1 + 2 + \dots + 2^{K-1}$ - yes? In the worst case, you have to check every element. How do you estimate the average case?)

While this seems like a simple solution on face value, it can be horribly inefficient. What if I wanted to make sets of media objects that were each unmanageably large?

A better solution is to store all possible set members, the so-called Universal or Master set, within the object as a class (static) member. Each instantiated set would then be simply composed of a collection of integers, which are simply pointers to the actual objects in the Master set.

This sounds like a good idea. But there's an even better one. With a static master set, you still have to initialize it by copying over the supplied elements into it. With an understanding that you will not change the master set, why not have your client (the instantiator of the Set class) create and maintain the master set as a vector, and you only deal with integer indices into this (externally maintained) master set?²⁹

That's the approach we will take in this quest. Although you don't have to code a lot, most of the time, I presume, may be taken up by simply wrapping your head around this:

- The Set object will contain a pointer to an externally maintained vector of elements we call the master set. This pointer must be set during set initialization.
- Each Set object will contain a vector of integers, each of which is an index in the master set to a valid element contained in that Set. (See Figure 1)



I'm not a Figger. Quit starin' at me

²⁹ Although the contents of this master list is vulnerable to change from outside (since it actually lives outside the class), trying to overcome this shortcoming by making our own static copy doesn't seem to me to be such a good tradeoff. In the interests of learning the algorithm, we'll assume a certain level of intelligence and self-preservation instinct in the users of your class.

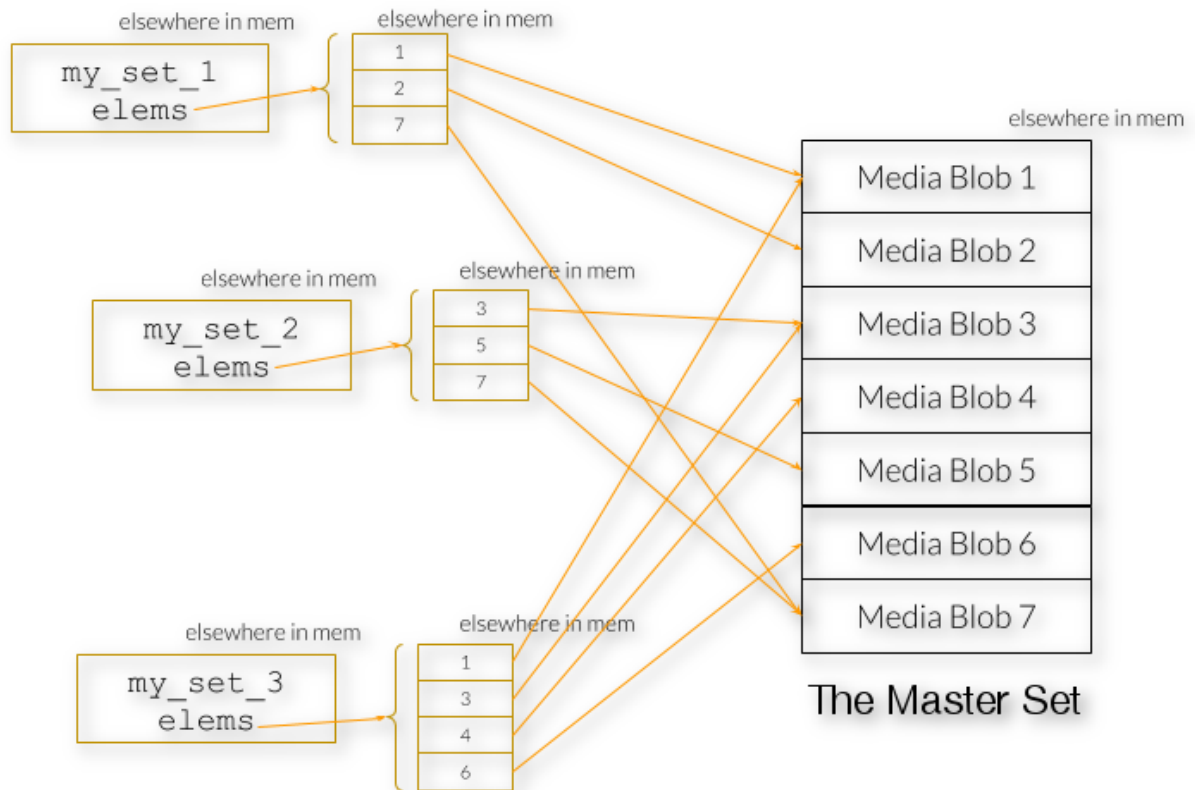


Figure 1. Many sets contain indices into elements stored in the master set

Tips

1. You don't want to iterate over each candidate set to calculate the sum of its elements in real time. Instead, maintain a `sum` member in your `Set` class such that every `Set` object knows its own sum.
2. You'll have a nested loop in your search method: One over all the items in your master vector, and another over all the subsets in your vector of candidates. Be careful as you craft these loops. If you end up adding subsets inside the loop, well, ... good luck. Make sure that at each pass over the inner loop, only direct descendants of the current set are processed. Recall that by *descendants of a set*, I mean all the sets you can obtain by adding one or more extra elements to it. A direct descendant is a descendant that differs from its parent by exactly one element.
3. You can and should test in your inner loop to see if you have added a new sub-list whose `sum() == t` because, if you have, you are done and should break out of the outer loop. Normally there will be zillions of subsets and we can usually reach the target long before we have tested them all.

Figure 2 shows a fuzzy photo of the Set class.

```
template <typename T>
class Set {
private:
    vector<T> *_master_ptr;
    vector<size_t> _elems; // List of indices into this->master
    size_t _sum;

public:
    Set(vector<T> *mast_ptr = nullptr) : _master_ptr(mast_ptr), _sum(0) {}

    const size_t size() const { return _elems.size(); }
    const vector<T> *get_master_ptr() const { return _master_ptr; }
    const size_t get_sum() const { return _sum; }

    bool add_elem(size_t n); // n is the index in master
    bool add_all_elems();   // Add everything in the master

    Set<T> find_biggest_subset_le(size_t target);

    friend ostream &operator<<(ostream& os, const Set<T> &set) {
        const vector<T> *mast_ptr = set.get_master_ptr();
        os << "{\n";
        for (size_t index : set._elems)
            os << " " << mast_ptr->at(index) << "\n";
        return os << "}";
    }

    friend class Tests; // Don't remove this line
};
```

Figure 2. A fuzzy pic of the Set class

Figure 3 shows some code that I use to test your Set. Use yer own.



One important point to keep in mind before forging ahead: In general, you may have multiple sets adding to the same target. How do you know which one is the winning set? It's like lotto. Except you can actually figure out how to get the winning ticket.

Onward to the miniquests... If messages you receive seem too cryptic, see if anybody on our [subreddit](#) knows what they mean.



Miniquiest 1 - Default constructor

Implement the default constructor for your Set class. It should correctly make an empty set.

Miniquiest 2 - The Empty Master

You don't need to do anything here, but if your implementation agrees that an empty master has nothing to give, you get a reward. How's that?

Oh yeah, you can't move forward unless you do. Small detail.

```
vector<int> master;
size_t master_total = 0;
for (size_t i = 0; i < NUM_ENTRIES; i++) {
    int val = rand() % 300;
    master_total += val;
    master.push_back(val);
}

// Select an arbitrary target.
size_t target = master_total/2;

Set<int> master_set(&master);
Set<int> best_subset = master_set.find_biggest_subset_le(target);

cout << "Target = " << target << endl;
cout << "Best sum = " << best_subset.get_sum() << endl;
cout << best_subset << endl;
```

Figure 3. Some test code

Miniquiest 3 - Non-default constructor

Actually, this one's a freebie too. 'Cuz the fuzzy photo gives you both constructors.

Miniquiest 4 - Add 'em all

I'm gonna try and add all the items in the master. If it works, you get a reward! Sweet.

Since the method name is a verb, assume that it should return true on success and false otherwise. Hmmm... When might it fail?



Miniquiest 5 - Be Legal

I'm gonna tempt you and test you. You get rewards for staying on the right side of the law!

Miniquest 6 - Nothing and Everything

I think that to make anything, you need to first be able to make nothing.

If you can't make nothing, then you can make nothing.

Take your `find_biggest_subset_le(size_t target)` for instance.

If I invoke it with a target of 0, you must return nothing, the empty set.

If I invoke it with a target it can never meet, then I expect it to give me its all. That means everything.

These may seem like two special cases you can handle separately for easy rewards.

However, whether you treat them as special cases or let them naturally fall out of your search makes all the difference to how easy you'll have it when troubleshooting.

Remember - *Once you can make nothing, you have a way to make anything, including everything.*

Finally, note that `find_biggest_subset_le()` returns a copy of a set - not a reference or pointer. (Why is it *not* a big deal to return a copy here?)



Miniquest 7 - Solve small cases

I'm gonna do a bunch of random moves here. All you have to do is to watch and do exactly the same.

Well, YOU don't have to do anything. Your code will. Let's see if it can dance my small dances.

Miniquest 8 - Solve small song lists

Yeah - Same as before. Let's see if your code can keep up with my disc jockeying.

Miniquest 9 - Solve big cases

Ok. These bigger dances will weed out the slow pokes. I'll see how many of y'all master these moves.

Miniquest 10 - Solve bigger song lists

Or these songs.

Submission

Yeah. This is one of those cool quests where it seems like a lot of work, but really, you only need to do a couple of things right and everything else just slides into place satisfyingly. When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your Set .h file into the button and press it

Wait for me to complete my tests and report back (usually a minute or less).

Some questions to get you started on our [sub](#)

1. Does a particular order of processing the master set elements work better than others? (e.g. Does sorting the list of items help?)
2. Does the value of the target matter (i.e. are certain targets likely to be reached sooner)?
3. Does the nature of the probability distribution from which the master set elements are drawn matter? (e.g. Can the algorithm take advantage of the fact if you knew that the master set elements were drawn from a particular probability distribution (e.g. a Gaussian, uniform or exponential distribution, etc.)



Self calibration for the RED level

If you complete this first RED quest on your own, only looking for information and/or help from your friends (or online) to understand what to do next, you'll get a very good idea of how you will fare in RED territory. The remaining quests will be of the same (or gradually increasing) level of difficulty as this one (with an occasional easy treat).

If you enjoyed working on this quest without being frustrated by endless test failures, that means you will have a fantastic time in RED.

Otherwise, you can still have a fantastic time by investing a little bit of effort before. I suggest you clear the GREEN quests first. To get to the GREEN quests, ofc, you may need to go to a tiger named Fangs. But don't worry. [He no bite](#).

Happy Questin',

&

*I bet that if I got meself some 6 or 7 stilts
Altho' I be tall alreddy, I be taller even stills.*



i scares em with my stilts and pecks em with my bilts

Matrix and Sparse Matrix

previously by Michael Locuff

Sometimes we want to store an enormous matrix. So many rows and cols that maybe even all the available bits on the planet wouldn't suffice to hold 'em.

But it may be that most of the elements happen to be the exact same value, say zero.

Then it should technically be possible to store any (possibly infinite)³⁰ amount of such data on a significantly small finite device as long as the *effective data density* (EDD) is below some threshold. By EDD, I mean the ratio of the number of non-default values to the total number of values.

You can offer the user a partial illusion of the existence of such a data set by making your opaque getters generate the bulk of their values algorithmically. This is a cool way in which you let the software take on more of the hardware's functionality. (Come back when you've found out where this is happening in this quest and post your thoughts on our [sub](#))

Here, you'll get to implement one such data structure as a template class - A Sparse Matrix. You can think of it as a two dimensional array in which some relatively small number of cells are expected to contain non-default values.

Clearly, you can't afford to have a vector of vectors. That would defeat the purpose, right? So you'll store your data in a vector of lists, which is a second best from a storage perspective. (Why second?)

Read on to learn more.

Implementation Specifics

First things first. Starting with this quest, you will stop seeing explicitly spelled out miniquests. The spec will give a high-level description of what is required and discuss implementation specifics. You'll find that trophies automatically become yours as you do more things right.

How many more?

Well. That you gotta find out yerself.

Before you get to implementing the sparse matrix, you must first implement a regular matrix, also a template class: `Matrix<T>`. This class will allow you to extract relatively small rectangular regions, called *slices*, from a large Sparse Matrix (to come).

Refer to Figure 1 for a hazy picture of the Matrix class.



³⁰ Can you think of a quest you did (may not be in RED) in which you had to store *infinite data* using clever abstractions?

```

template <typename T>
class Matrix {
protected:
    vector<vector<T> > _rows;

public:
    Matrix(size_t nr = 0, size_t nc = 0);

    size_t get_num_rows() const { return _rows.size(); };
    size_t get_num_cols() const { return (_rows.size()>0? _rows[0].size() : 0); };
    void clear() { _rows.clear(); }

    void resize(size_t nr, size_t nc);
    T &at(size_t r, size_t c); // throws OOB
    string to_string() const;

    class OOB_exception : public exception {
    public: string what() { return "Out of bounds access"; }
    };

    // Friends -----

    friend ostream &operator<<(ostream& os, const Matrix<T> &mat) {
        return os << mat.to_string();
    }

    friend bool operator==(const Matrix<T> &m1, const Matrix<T> &m2) {
        if (m1.get_num_rows() != m2.get_num_rows()) return false;
        if (m1.get_num_cols() != m2.get_num_cols()) return false;

        for (int r = 0; r < m1.get_num_rows(); r++)
            for (int c = 0; c < m1.get_num_cols(); c++)
                if (m1.at(r, c) != m2.at(r, c)) return false;

        return true;
    }

    friend bool operator!=(const Matrix<T> &m1, const Matrix<T> &m2) {
        return !(m1 == m2);
    }

    friend class Tests; // Don't remove this line
};

```

Figure 1. A fuzzy photo of the Matrix<T> class

Most of what you will find rewarding with the Matrix class is shown in Figure 1. But, alas! Some of it is missing. Just fill in the missing implementations.

Imagine that I would instantiate a Matrix of doubles or strings and that I'd invoke each of the public methods you see in the figure, and the friend methods of Matrix.

As you read ahead, remember that you may still not understand everything even after reading this spec. If so, that's great because you can start a discussion about it on our [subreddit](#) and improve its next version.

About the Matrix class

There is nothing sparse about your non-sparse Matrix. It is a vector full of vectors. The inner vectors are your rows and your outer vector is your vector of rows. Each row has as many cells as the number of columns of this matrix. This vector should be called `_rows`, as shown in the figure. Don't change it.

Here are some rewarding facts about the `Matrix`:

The Matrix constructor

The constructor takes two `size_t` params: `nr`, which gives the number of rows, and `nc`, which gives the number of columns. You must correctly size the member, `_rows`, to hold as many elements and sub-elements as needed. Each element will be automatically assigned the default value of the type `T` by the `vector` constructor.

Done correctly, you should *not* have to maintain two separate members tracking the height and width of your matrix.

My personal preference is to try and keep constructor code as lean as possible, I think I discussed this somewhere on our [GREEN sub](#). However, this time, I let the interests of better code factoring override this desire.

Since I have a `resize()` implemented and verified *beyond reasonable doubt* AND it is lexically positioned adjacent to my constructor, I gave myself the luxury of invoking it from within.

You can choose to do it differently, of course. See how it goes. One way may be rewarding and another not. I have no way of telling what works better for you.

Equality

Yeah. It's important. If equal mats ain't equal then yer quest ain't got no sequel.

At

This one's a biggie. The thing about `Matrix::at()` is that it is the ONLY interface `Matrix` presents to get at its innards. You will make it return a reference to the element and completely eschew the unchecked `get()` method, or the bracket operators.³¹

In terms of naming, I picked `at()` over `get()` because our functionality is more similar to that of `vector::at()`, which returns a reference, rather than to a traditional const getter.

`Matrix::at()` should be a checked method that throws an `OOB_exception` if either `r` or `c` is an invalid index.

³¹ What are the implications if `at()` returned a copy of the element rather than a reference to it?

To string

Simply print the line "# Matrix" by itself, followed by data lines, at one row per line. Each element of each row should be in a field 6 characters wide³² with one space before every field except the first. There is one newline after each line (including the last). Figure 2 shows sample output from myMatrix<int>::to_string()

```
# Matrix
437605 642314 836725 159909 506331 815035 816302
409351 452462 833879 296251 89079 975107 252029
241127 392454 28736 10327 994523 901792 811147
597315 24573 826460 198260 683811 440188 348233
584421 302083 764371 720702 733245 806895 514515
351438 150475 76576 105777 542228 875713 405431
198472 37512 408578 413258 251483 111951 367854
344877 860043 685 290380 308258 762205 836872
709951 691467 841524 48538 487976 931525 460761
```

Figure 2. Sample Matrix::to_string() output



³² Use std::setw(6)

The Sparse Matrix

Like it said in the introduction, you will implement your sparse matrix as a vector of lists.³³

A user should be able to instantiate a concrete version of your template sparse matrix class like this (example uses `double`):

```
Sparse_Matrix<double> my_sparse_matrix;
```

The constructor of the sparse matrix should take 3 parameters:

```
Sparse_Matrix(size_t nr, size_t nc, const T &default_val)
```

The number of rows is given by `nr` and the number of columns by `nc`. Indexing should begin at 0, and so the actual rows and columns will span indices 0 to `nr-1` and 0 to `nc-1`.

Why do we even need `nr` and `nc`? After all, isn't our sparse matrix supposed to accommodate arbitrary numbers of elements (limited only by available memory)?

Good questions. Kudos await thoughtful discussion in our [sub](#).

But what is this other thing called `default_val`?

This I can help with. It is a value stored in the instance (not the class) and returned to the user whenever they ask for the contents of a cell *not physically stored* in the representation. Since the sparse matrix may be as large (e.g. 1 billion cells tall and wide) it's not feasible to store every single value explicitly. Rather, it will only store the significantly fewer non-default values, and return `default_val` whenever a request for an unstored value arrives.³⁴

I managed to buy a photo off a character on the Milk³⁵ road. But the seller was shady and the pic was even worse. Still, I managed to salvage nuff of it, though some parts are still obscure. FWIW, it's in Figure 3. See if it helps.



³³ What might be some interesting similarities and differences between this and the general tree from GREEN?

³⁴ Hey, quiet! Can anyone hear a *mynah* in here somewhere?

³⁵ I'm told it's one of dem places where you could buy a quest solution for something that rhymes with bitcoin.


```

template <typename T>
class Sparse_Matrix {
private:
    static double constexpr FLOOR = 1e-10;

    struct Node { // private inner
        size_t _col;
        T _val;

        Node(size_t c, const T &v) : _col(c), _val(v) {}
        size_t get_col() const { return _col; }
        const T get_val() const { return _val; }
        void set_val(const T &val) { _val = val; }
        virtual const T &operator=(const T& val) { return _val = val; }

        friend ostream& operator<<(ostream& os, const Node &node) {
            return os << "{ C: " << node.col << ", V: " << node.value << " }";
        }
    };

    vector<list<Node> > _rows;
    size_t _num_rows, _num_cols;
    T _default_val;

public:
    Sparse_Matrix(size_t nr = 0, size_t nc = 0, const T &default_val = T()) :
        _num_rows(nr), _num_cols(nc), _default_val(default_val) {}

    size_t get_num_rows() const { return _num_rows; };
    size_t get_num_cols() const { return _num_cols; };

    bool is_valid(size_t r, size_t c) const;
    void clear();
    const T get(size_t r, size_t c) const;
    bool set(size_t row, size_t col, const T &val);
    Matrix<T> get_slice(size_t r1, size_t c1, size_t r2, size_t c2) const; // rect slice

    friend class Tests; // don't remove

```

Figure 3. The partially obscure `Sparse_Matrix<T>` class

As you can see, your sparse matrix maintains a *spine*, which is a vector of lists of nodes. Each node contains two things: a value, and the column number the value occupies.

Given a row, `r`, and column, `c`, you can now index into the corresponding list, `_rows[r]`, and scanning this list linearly, find whether the column of interest resides in it or not. If it does, you would have located non-default values for the contents of the requested cell. If it doesn't then you can assume that the cell contains `default_val` in virtual space.

I hope that makes sense. See Figure 4 for a pictorial representation of this idea.



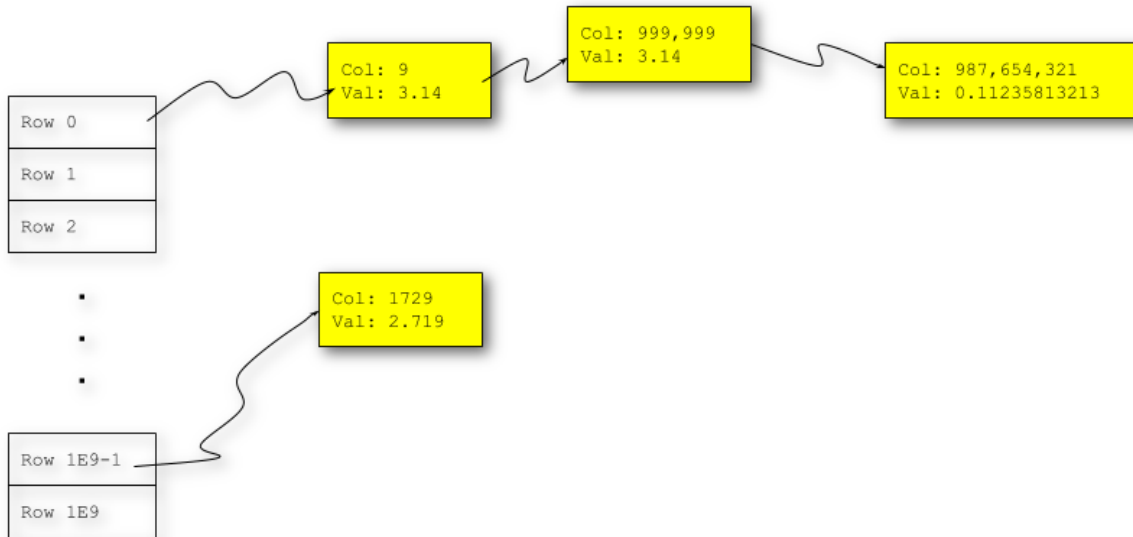


Figure 4: The Sparse Matrix as a vector of linked lists of Nodes containing cols and vals. Any value that cannot be found in a yellow cell is assumed to be `default_val`. This sparse matrix has a ++billion rows and a billion columns. 4 of these billion billion cells have non-default values

Important subtlety

Since your rows are linked lists and not vectors, you may as well exploit their $O(1)$ in-place insertion complexities. Your linked lists of Nodes must be kept in ascending³⁶ order by column number. Although not terribly significant in our use case, this does buy you about half the time you might have spent searching on average. Why?

Here are further rewarding things to know about the `Sparse Matrix`:

- I found it useful to have a private `is_valid(r,c)` helper to make sure I didn't accidentally access something that didn't belong to me.
- `clear()` should keep the spine of the sparse mat, but clear out its elements.
- `get(r,c)` should return the stored value at the requested location or the default value if a value couldn't be found. It doesn't have to mess around with exceptions.³⁷ If the requested row and column are either out of bounds or invalid, it may return the default value.
- `set(r,c,val)` is more subtle:
 - Make sure the Nodes in the list at `_rows[r]` preserve their ordering property (ascending order by column id). To achieve this you simply have to do this:
 - Scan the list from head to tail looking for the target column, `c`. There are only 3 cases as you test each node:

³⁶ Do I need to say non descending instead? Why? Or why not?

³⁷ Riley Short (Spring, 2022) asked why we don't throw an OOB in this case because a `Sparse_Matrix`, though large, is not infinite and has real boundaries that can be breached. Can you think of good reasons to do it one way or the other? [Link to the discussion on our sub.](#)

1. the current column $< c$: move on to the next node
 2. The current column $> c$: remember this location and break out of the loop.
 3. (else) The current column $= c$: if `val` is the default value, delete this node and you're done. Else, whatever the current value of this node, simply reset it to `val` and you're done.
- This is when you have broken out of the loop, but are not yet done (when does this happen?) If you're here, it means that you're exactly at the node behind which a new node with column `c` should be added, but of course, only if `val` is not the default value. Go figure.
- `get_slice(r1, c1, r2, c2)` is straightforward:
 - Rearrange `r1, c1` and `r2, c2` so they form a rectangle with `r1, c1` at the top left and `r2, c2` at the bottom right.³⁸
 - Extract values from the sparse matrix at every location from within this rectangle (includes the end points) and stuff it into corresponding locations in a regular matrix whose size is just enough to hold the extracted slice.
 - Use `spmat.get(r, c)` to extract values. Use `mat.at(r, c) = val` to insert them.
 - I guess this is your ultimate test of comfort with corners.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your `Matrix.h` and `Sparse_Matrix.h` files into the button and press it.



Wait for me to complete my tests and report back (usually a minute or less).

Happy Questin',

&

³⁸ Hmm... Did Meg say there was an easy way to do this?

*O' China taught us how to fish and fly
Out here, we simply sit. And multiply.*



a cormorant who multiplies

To code less and to multiply

previously by Michael Locuff

Having large matrices is all well and good. But we also want to be able to multiply them (if they are compatible).

In this quest, You get to implement a class that houses Matrix utilities (though, of course, you will only implement multiplication). Through friendship to the Matrix and Sparse Matrix classes from your previous quest, you can *carefully* access the guts of those classes and implement efficient Matrix (or Sparse_Matrix) functions.

I hope this relatively easier (and shorter) quest gives you some breathing room to get ready for the next one, which is loaded with rewards.

About Matrix Multiplication

Look up Matrix multiplication in a math text or online. You need to understand it clearly in order to complete this quest, especially the part to do with multiplication of sparse matrices.

Figure 1 shows how Matrix multiplication works: Consider three matrices $A \times B = C$:

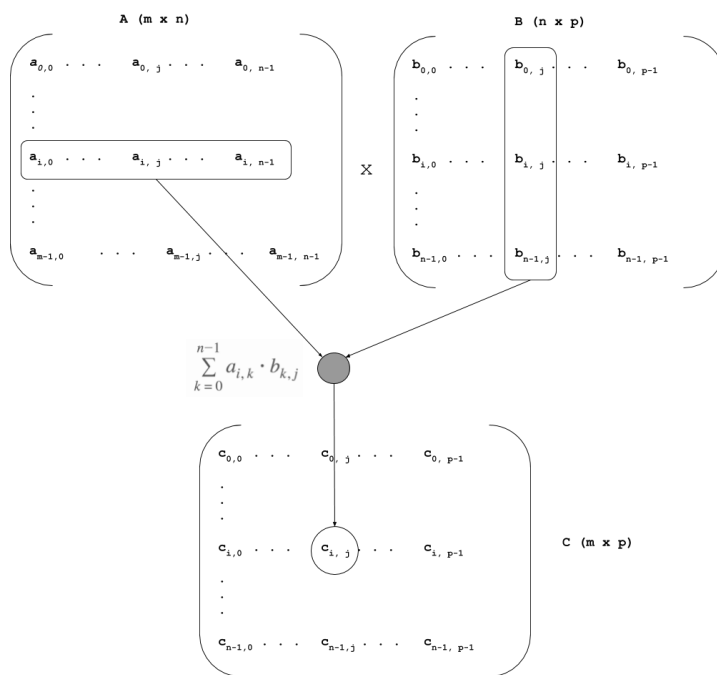


Figure 1. Matrix Multiplication

So...

1. This quest only makes sense for numerical `Matrices` and `Sparse_Matrices`.
2. These must be compatible for multiplication. The number of columns in the first matrix must equal the number of rows in the second (no constraint on the rows of the 1st and cols of the 2nd). Yes, that means that matrix multiplication is not commutative.
3. Each element of a product matrix, C , is obtained by summing the products of corresponding elements from A and B . Specifically, $c[i, j]$ is obtained by summing the corresponding elements from the dot-product of the i 'th row of A and the j 'th column of B . If A and B are compatible for multiplication, then the number of elements in the i 'th row of A must equal the number of elements in the j 'th column of B . Thus you are guaranteed to have a correspondence for each element of either vector. This sum, the dot product between two vectors, is written as:

$$\underline{a} \cdot \underline{b} = \sum_k a_k b_k \quad - \text{which is just shorthand for } a_0 b_0 + a_1 b_1 + \dots + a_{n-1} b_{n-1}$$

Not so clear? Please discuss on our [sub](#). Most likely, you're not the only one.

It's important to completely understand how this works because when you get around to implementing multiplication for sparse matrices, you can't afford to do unnecessary lookups. In a regular matrix, it doesn't matter how many times you refer to a certain cell because access happens in constant time.

In a sparse matrix, element access time is proportional to the number of non-default elements contained in the row in which the element exists (according to the specs of the `Stilt` quest). Your utility functions must be aware of the underlying implementations of the data structures they are dealing with and handle them appropriately, even though the external presentations of both structures (`Matrix` and `Sparse_Matrix`) are the same and conceptually the same operations apply to both.



Implementation specifics

Implement this quest in three files (suggested number):

1. `Matrix.h` (from the Stilt quest - needs editing)
2. `Sparse_Matrix.h` (from the Stilt quest - needs editing)
3. `Matrix_Algorithms.h` (new file)



Set up your project by copying over the first two header files from your Stilt quest. Create a new file called `Matrix_Algorithms.h`

In this file (I suggest) define a new class called `Mx` (See Figure 2). It will be home to all of your matrix³⁹ manipulation methods. In this quest you will only need to implement the matrix multiplication function. But you can imagine that this class may eventually be home to many other matrix functions.

Note that `Mx` is itself not a template class although it provides template utility functions over matrices.

In order for `Mx` to be able to directly access internal data elements of the `Matrix` and `Sparse_Matrix` classes, it must be their friend. So insert the required friendship statements into the original classes.

In addition, here are assumptions I made in my reference implementation. Your assumptions had better match if you want to max out on your rewards:

- When dealing with floating point values, it's usually a bad idea to compare for exact equalities. So we define *ranges of values* to look for. The default value in your sparse matrices is 0.0. But to avoid the risky exact comparison, define a `static constexpr` called `Sparse_Matrix<T>::FLOOR` whose value is 10^{-10} (1e-10).
- Then define a utility method:

```
bool is_default(const double &val);
```

It should return `true` if the absolute value of the difference between `val` and the instance's `_default_val` member is smaller than your `FLOOR`, and `false` otherwise.

³⁹ Some of what is rewarding for a regular matrix may earn you even more rewards with sparse matrices. You'll never know until you try.


```

s Mx {
ic:
template <typename T> static bool can_multiply(const Matrix<T> &a, const Matrix<T> &b) {
    // TODO
    return true;
}

template <typename T> static bool multiply(const Matrix<T> &a, const Matrix<T> &b, Matrix<T> &res) {
    // TODO
    return true;
}

// Sparse_Matrix utils -----
template<typename T> static bool can_multiply(const Sparse_Matrix<T> &a, const Sparse_Matrix<T> &b)
    // TODO
    return true;
}

template<typename T> static bool add_to_cell(Sparse_Matrix<T> &spmat, size_t r, size_t c, const T &val)
    // TODO
    return true;
}

template <typename T> static bool multiply(const Sparse_Matrix<T> &a, const Sparse_Matrix<T> &b, Sparse_Matrix<T> &res)
    // TODO
    return true;
}

```

Figure 2. A View of Mx

All you gotta do in this quest is to fill in the // TODO portions of the five inline methods.

Here is a little more detail that is hopefully helpful:

- `can_multiply(a, b)` returns `true` if matrices `a` and `b` are compatible for multiplication in the given order (i.e. $a \times b$). Be sure to check for corner cases.
- `add_to_cell(r, c, val)` will add `val` to the sparse matrix cell at (r, c) . You can imagine its functionality to be similar to but yet subtly different in important ways from the `Sparse_Matrix`'s `set()` method from the Stilt quest. Remember that if the addition of `val` to the existing value makes it equal to the default value (as determined by `is_default()`) then the node should be removed from the `Sparse_Matrix`. The reference sparse matrix preserves its sparseness through operations like this, and yours needs to match in order to get past a miniquest. It should return `false` if the supplied coordinates are invalid or if the spine of the matrix is not long enough.

When you have passed all the other miniquests, the site will tell you how long my multiplication code took on 2 relatively large sparse matrices. It will also tell you how long yours took.⁴⁰

Feel free to discuss these times in our [sub](#). You should be able to run as fast as the reference code, or better. Discuss and vote-up optimizations and unclever hacks.⁴¹

⁴⁰ Hmm... if I multiply two sparse matrices with density p each, what can I say about the density of the product?

⁴¹ *unclever hacks* are hacks you can be sure to understand when you're much much older and not so much much cleverer.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files⁴² into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

Happy Questin',

&



⁴² These would contain the definitions of the three necessary classes.

*Dis mockingbird - He make da purty pet
His name be Greg. And oh, he like to peck.*



the mockingbird, a sadhu

The BST & the Lazy BST

previously by Michael Locuff

In this quest, which is simply LOADED with rewards, you get to implement a Binary Search Tree (BST) and then generalize the class to support *lazy deletion*.

Roar or purr. It's up to you which you do first. But you get to do 'em both in this quest. It assumes that you already know how to make BSTs. The main focus of this quest will be its doozy cousin, the `Lazy_BST`.

You need to know how to work BSTs with little to no effort because you may need all the time you've got to debug `Lazy_BST` issues.



Those of you who saved up a ton of time by acing your previous easy quest way too soon (I hope that's all of you) - here is where you can put that saved time to good use.

A new notation

One of the things that this quest will make you accustomed to is a new notation: `*&p` (no I'm not swearing).

```
// parameter p is a reference to a pointer to a Bar object.  
void foo(Bar *&p);
```

Remember how you had the luxury of having a head node in the Platypus quest and the head node always pointed to the first element of the collection?

Well, here's a great way to find out how to handle the situation when you don't have that luxury. If you don't have a fixed handle to a pointer that may end up getting changed in a function call, then you have to pass *a reference to the pointer*. The notation for this, though apparently awkward when you first encounter it, can quickly be internalized into an idiom you can (and should) get comfortable recognizing immediately.

This is important when you're being handed a tree's root to manipulate and you end up changing the root node itself. The reference to the root node allows you to reset the root pointer in the object within which your tree's root is held. Once you've done this a few times, it will stop looking mysterious. Yet, FWIW, I'd suggest that you try to implement a BST (in your own time) using the other way (a head node and a cursor) and see how that goes.

The BST

Not much to say here. Except that if you're not up to snuff yet on Binary Search Trees, this would be a good time to hit your reference material. Whether that's your text, video, interactive app, or discussion forum - go for it. Check in our [sub](#) if you're stuck.

Figure 1 shows a picture of the BST class. As you can see, it is a template class. I should be able to instantiate binary search trees of any type I want, as long as my type supports the less-than comparison operator.

You'll notice that a few of its public methods and all of its private helpers are missing implementations. Figure out which ones, and implement them.

Your entire implementation should be contained within the file `BST.h`, which you will submit.

Here are some *possibly* rewarding details to know about some of the methods:

- Note that a null tree is a tree. Thus it is a good sentinel. Methods that return node pointers may return the null pointer on failure.
- The `Node`'s deep copy method should return an exact (and disjoint) clone of the given node. Note that it is a static method which takes the source node as a parameter.



- `_insert(Node *&p, const T &elem)` should insert the given element into the subtree rooted at `p`. Note that `p` may be null. In that case, you would have to create a brand-new tree rooted at `p` (this would be the time when the reference parameter comes in handy). It is an error to insert a duplicate (return `false`).
- `_remove(Node *&p, const T &elem)` should remove the given element from the subtree rooted at `p`. Return `false` if the element does not exist in the subtree.
- `_find_min(Node *p)` should return a pointer to the node with the least element in the subtree rooted at `p`. It may be `p` itself. (When?)

- The public `find(const T &elem)` method should attempt to find `elem` starting at the root of the tree. If the element is found, it should return a *constant* reference to the element. Otherwise, it should throw an instance of `Not_found_exception`.
- `_recursive_delete(Node *&p)` should free up all heap space allocated for the subtree rooted at `p`. As the name suggests, save yourself a lot of effort by being recursive here.⁴³ Remember to set deleted nodes to null and to manage your `_size` correctly through the process.
- The public facing `to_string()` simply puts a wrapper around the serialized root node of the BST.
 - The first line should say `"# Tree rooted at [X]"`
 - The next line should say `"# size = [X]"`
 - This should be followed the result of `_to_string(_root)`
 - The last line should say `"# End of tree"`
- The private helper `_to_string(const Node *p)` should serialize the subtree rooted at `p` in depth-first left to right order:
 - Append a line with the given node (it's `_data` element), a space, a colon, and a space, then the left child's `_data` element, a space, and the right child's `_data` element.
 - Recursively append the left child's serialization
 - Recursively append the right child's serialization
 - If a node is null, then append the string `"[null]"` where it appears as one of the children of another node.
 - If a node has NO children (both left and right are null), then don't print a line for that node (i.e. You will not produce a line like `"X : [null] [null]"`)
 - As an example, the following `to_string()` output describes the tree shown in Figure 2.

```
# Tree rooted at 19
# size = 9
19 : 9 29
9 : 4 14
4 : -1 [null]
29 : 24 34
34 : [null] 39
# End of Tree
```

⁴³ Any time the size of a data structure grows exponentially by a factor of 2 or more) along some dimension (e.g. the height of a tree), I think it makes sense to see if recursive methods might work. This is because if the dimension grows logarithmically with size (e.g. tree height), you can count on the fact that the maximum stack overhead of methods recursing in that dimension will diminish exponentially. (Discuss this).

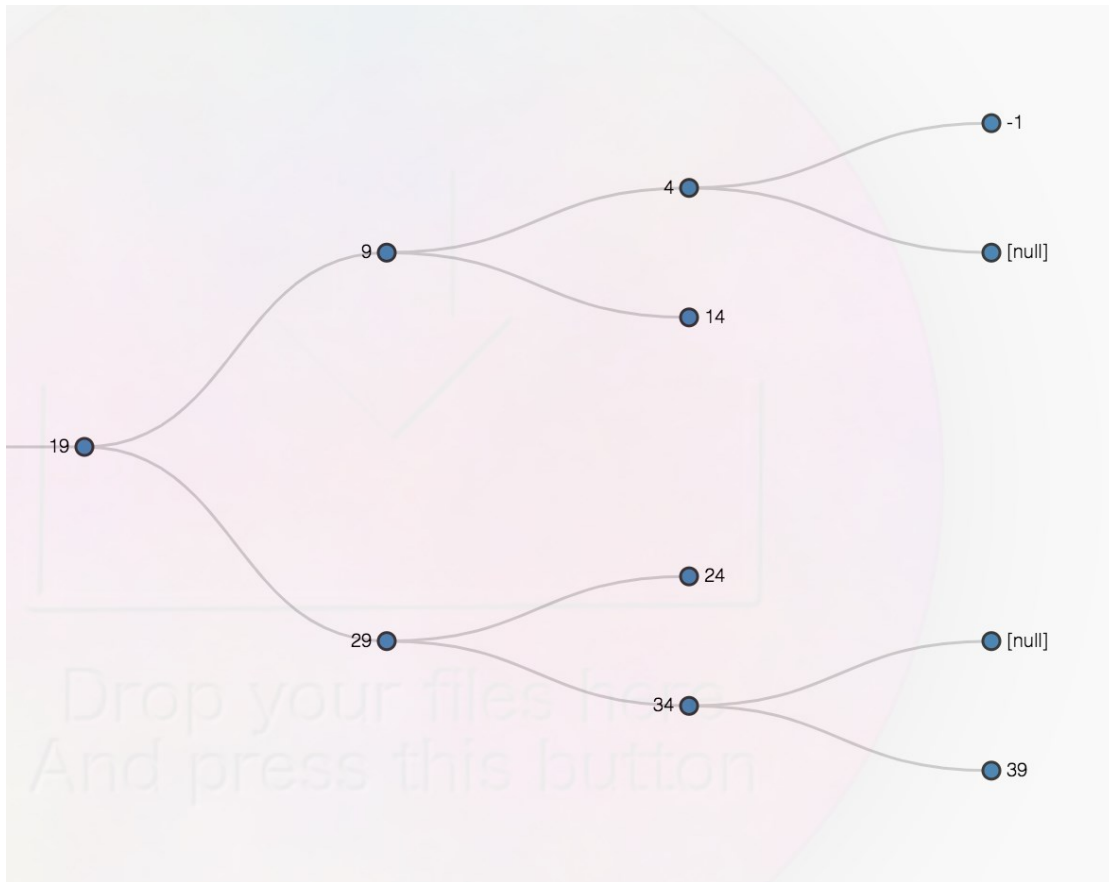


Figure 2. The Tree Rooted at 19

That's it. Let me know if you find out how many of these are tested for rewards in the quest.

Caution: Those of you who are familiar with another common implementation may think about using an extra class member called `mRoot` to code tree membership into each node (every node knows the exact tree to which it belongs). This is so your tree's methods can check before accidentally operating on nodes from another tree. You are not required to do that check in this quest. So, beware!




```

// T must be Comparable. That is, must support ordering via <
template <typename T>
class BST {
private:
    struct Node {
        T _data;
        Node *_left, *_right;
        Node(const T &d, Node *l = nullptr, Node *r = nullptr) : _data(d), _left(l), _right(r) {}
    };
    Node *_root;
    size_t _size;

    // Helpers
    static Node *_deep_copy(const Node *p);
    bool _insert(Node *&p, const T &elem);
    bool _remove(Node *&p, const T &elem);
    bool _recursive_delete(Node *&p);
    const Node *_find_min(Node *p) const;
    Node *_find(Node *p, const T &elem) const;
    string _to_string(const Node *p) const;

public:
    BST() : _root(nullptr), _size(0) {}
    virtual ~BST() { _recursive_delete(_root); }

    virtual size_t get_size() const { return _size; }

    virtual bool insert(const T &elem) { return _insert(_root, elem); }
    virtual bool remove(const T &elem) { return _remove(_root, elem); }
    virtual bool clear();

    virtual bool contains(const T &elem) const { return _find(_root, elem) != nullptr; }
    virtual T &find(const T &elem) const;
    virtual string to_string() const;

    class Not_found_exception : public exception {
    public:
        string to_string() { return "Not found exception"; }
    };

    friend class Tests; // Don't remove
};

```

Figure 1. The BST Class



The Lazy_BST

What exactly is lazy about this BST? Deletion is.

When you try to delete a node in a `Lazy_BST`, it is not unlinked and released into the heap immediately, but simply marked as deleted. You need to store and manage this extra bit of information within each node.

What do you gain by doing this? Do any of your tree operations improve in complexity because of it? Under what circumstances does using a `Lazy_BST` make sense? All worthy questions to discuss in our [sub](#).



As you implement the `Lazy_BST`, if you have any questions regarding its public interface - simply let the following guide your decision making process:

The fact that your `Lazy_BST` is lazy is something that should only be known to you, the developer of the class. No matter what its underlying implementation, the public interfaces for methods `Lazy_BST` and `BST` share should be identical. In other words, the `Lazy_BST` should present a successful illusion to its user that it is a plain old binary search tree. The only exceptions are: `collect_garbage()` and `to_string()` - both discussed below.

Figure 2 shows you a picture of the `Lazy_BST` class.⁴⁴

Here are some *possibly* rewarding things to know about it:

⁴⁴ `Lazy_BST`, your lazy version of the `BST`, should not derive from `BST`. It should be a completely separate class. You may find it helpful to do a whole bunch of copy/pastes here.

`insert()`, `remove()`, `find_min()`, and `find()`

These are similar to their `BST` relatives. But they must now account for the fact that a node might be marked as deleted. For example, when inserting an element, you may find the element already in the tree in a deleted state. You would then have to unset its deleted flag.

`_find_real_min()` and `_really_remove()`

These correspond to your `BST`'s `find_min()` and `remove()` because now you can't believe their public versions (above) any more. They may now say a node has been removed, when *in reality*, it's still lurking in there!

In the node removal algorithm for a lazy tree, a node to be deleted must be replaced by the node with the real minimum in the tree (even if it has been marked as deleted), not with the minimum undeleted element returned by the regular `find_min()` (Why?).

The `_really_remove()` method really removes a node from the tree, not just mark it as deleted. This is the method that should be called from the lazy tree's garbage collector.

`_collect_garbage(Node *p)`

This is the private helper for the new public-facing garbage collector in `Lazy_BST`.

When it is invoked, it must scan the subtree rooted at `p` and release marked nodes into the heap. It's ok for this helper method to be recursive. If you code it systematically, you'll find it's short and simple:

- If the node is null, you don't have to do anything.
- Otherwise, recurse first on the left sub-tree, then on the right, and finally check `p`. In one of the basis cases, `p` would be marked as deleted. In this case, you would simply (really) remove `p` from the tree and return true. (What about other basis cases?)
- In any case, this method should return true if *at least one node* was released into the heap. That is, if collecting garbage resulted in a structural change to the tree. (When might you use this boolean value? Are there other, better, ways to tell if a tree needs a cleanup?)

`to_string()`

This is almost identical to that of a `BST` with one simple difference:

If a node is marked as deleted, then its element should have an asterisk appended to its name in the serialization.



```

class Lazy_BST {
protected:
    struct Node {
        T _data;
        Node *_left, *_right;
        bool _is_deleted;

        Node(const T &d) : _data(d), _left(nullptr), _right(nullptr), _is_deleted(false) {}
    };
    Node *_root;
    size_t _size, _real_size;

    // Private helpers
    bool _recursive_delete(Node *&p);
    bool _insert(Node *&p, const T &elem);
    bool _remove(Node *&p, const T &elem);
    bool _collect_garbage(Node *&p);
    const Node *_find_min(const Node *p) const;
    const Node *_find_real_min(const Node *p) const;
    const Node *_find(const Node *p, const T &elem) const;
    bool _really_remove(Node *&p, const T &elem);
    string _to_string(const Node *p) const;

public:
    Lazy_BST() : _root(nullptr), _size(0), _real_size(0) {}
    ~Lazy_BST() { _recursive_delete(_root); }

    size_t get_size() const { return _size; }
    size_t get_real_size() const { return _real_size; }
    bool insert(const T &elem) { return _insert(_root, elem); }
    bool remove(const T &elem) { return _remove(_root, elem); }
    bool collect_garbage() { return _collect_garbage(_root); }
    bool contains(const T &elem) const { return _find(_root, elem) != nullptr; }
    const T &find(const T &elem) const;
    string to_string() const;
    bool clear();

    class Not_found_exception : public exception {
    public:
        string what() { return "Element not found exception"; }
    };
    friend class Tests; // Don't remove this line

```

Figure 2. The Lazy_BST Class



Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files⁴⁵ into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

Happy Questin',

&



⁴⁵ `BST.h` and `Lazy_BST.h`

Mo' de Le Na Crocodile
As gators go, he quite docile



Mo' de Le na also first. Dis gator aces every quest

Playing with Splaying

If you're here, it means you've conquered the BST.

All well and good with the BST, but what if you keep getting stuck with an unlucky sequence of inserts? (What might these be?)

Although a well-structured BST gives you worst-case $O(\log N)$ search, you're not guaranteed a well-structured BST if your numbers were inserted in an unfavorable sequence. You remain vulnerable to the $O(N)$ beast, even if only slightly.

How can you insulate yourself against this worst-case possibility? By *insulating* I don't mean that you reduce the likelihood of the worst case from happening. I mean that you *eliminate* it. How can you *guarantee* that the worst case is never worse than logarithmic in the size of the tree?

One way is to keep your tree always *balanced*. You are by now, probably, familiar with one kind of a balancing act - AVL balancing. If not, this would be a good time to hit the reference material, experiment a bit and come back.

Overview

Welcome back.

If you've tried to do AVL balancing, you would have found that you might have performed extra book-keeping in your tree manipulation algorithms. Even worse - you might have had to store extra information in each node (what is this extra info?).

What do you get for all that extra work?

Guaranteed $\log N$ search times.

Hrrmmph! I can hear some people grumbling.

Good! As you may have discovered by now, the cool thing about many of these data structures is that we know many ways in which to deal with them depending on what makes sense to trade-off for what in a particular application.



Suppose we use BSTs in an application where space is at a premium and you need your Nodes to be as thin as possible. Then it makes sense to ask *"I wonder if this particular application needs this function's runtime to be absolutely $\log N$ bounded or whether amortized $\log N$ times⁴⁶ will do"*



As it happens, there are many applications in which we can live with amortized $\log N$ access times if it means you get a simpler data structure requiring less book-keeping than always balanced trees (like AVL).

Like in real life, we get to say *"Hey! Just 'cuz I know how to balance don't mean I gotta keep balancing ALL of the time."*

This lazy strategy of not trying to futz with any more of the BST than you absolutely have to gives us a simple and useful variety of BSTs - Splay trees. Like its balanced tree cousins, Splay trees will use left and right rotations (among other operations) to transform their internal tree representations at appropriate times.

But Splay trees don't have the headache of having to maintain the height of each node. In fact, you don't even have to make any structural changes to your tree. You don't have to ride the beast or its nose.

You can simply write a bunch of processing routines that operate on an existing BST. Much as you did with the Cormorant and matrices.

oops, I mean override
BST or its Nodes

In this quest, you must implement a set of operations on a BST that, if used exclusively to operate on the tree, guarantee $O(\log N)$ amortized access time. You will do it by splaying your BST behind the scenes:

1. Create a class called `Tx` in a file called `Tree_Algorithms.h`
2. Make `Tx` a friend of your BST class
3. Implement the algorithms in this quest as static methods of `Tx`, operating on BSTs passed in by reference as their first parameters.

⁴⁶ Amortized means that you only consider the average time of M random calls to an operation in the limit. Although an individual call may end up taking $O(N)$ time, the algorithm should guarantee that the sum of times for M calls is bounded by $M \log N$. Note that the value of M matters. If M needs to be 1, then you have to suffer the overhead of continuous rebalancing (e.g. AVL). OTOH, techniques like splaying give you guaranteed average time behavior over a minimum of M calls. The smaller the value of M before the average is closer than some threshold to $M \log N$, the better the algorithm (runtime-wise)

The algorithm for each miniquest is discussed at length in many places on the Internet if you look. The basic idea in all of them is simple. I'll try and give you a slightly different perspective from the others. If you understand one, you'll understand them all.

Overview of Splaying

Splaying is an exercise in spin-offs and mergers. When you splay a BST for a target X, you will essentially:

1. Dismantle the input tree and break it into three separate trees.
 - a. One part is known to be less than X (the *left* tree)
 - b. One part is known to be greater than X. (the *right* tree)
 - c. One part is known to either be rooted at X or not contain X (the middle tree)
2. You reassemble a new tree by using these three parts.

How do you dismantle the input tree?

As you barrel towards X (or where you think it might be) in your middle tree starting at the root, you make decisions about entire sub-trees which you can know for sure to not contain X. You also know if the elements in these subtrees are greater than X or less than it. You can leverage this knowledge to come up with the following strategy.



It uses a 2-step lookahead. By this, I mean that specific actions you choose will be determined by the next two nodes you may visit in your search for X in the middle tree.⁴⁷

1. Start with 3 trees:
 - a. A tree with all items $< X$ (*left - empty at first*)
 - b. A tree with all items $> X$ (*right - empty at first*)
 - c. A tree with items you don't yet know about (*middle - the entire input tree at first*)
2. Standing at the root of the middle tree and looking towards where X might be,
 - a. if you find you have to travel in the same direction (left-left or right-right), select and perform the *zig-zig* dance (appropriately adjusted for direction). See Fig 1.
 - b. If you find you have to travel first in one direction and then the other (left-right or right-left), select and perform the *zig-zag* dance (appropriately adjusted for direction). See Fig 2.

⁴⁷ Do you think it makes sense to look ahead more than 2 steps? Do you think we might find one of 8 as-good-or-better reconfigurations if we looked 3 steps ahead? What if the answer to the previous question was "yes"? (Actually the answer has got to be yes. why?)

These dances will remove entire chunks in the middle tree which you now know to be either $< X$ or $> X$ and attach them to the $< X$ or $> X$ trees as appropriate.⁴⁸

3. Finally, use the three trees to reassemble a new BST. (BTW, how do you know for sure that this new BST is *improved* for future access? By *improved* I mean that it is restructured in a way that has resulted in reduced amortized access time. One way is to believe me because I say so. But I might be bluffing.)

The Dances

Figure 1 shows the steps needed for the catchy little number called zig-zig. Make sure you do each step exactly once and in the right order. Cuz, otherwise, you'll trip over yourself.

Maybe you want to start with the easier zig-zag dance? You can find the steps in Figure 2.

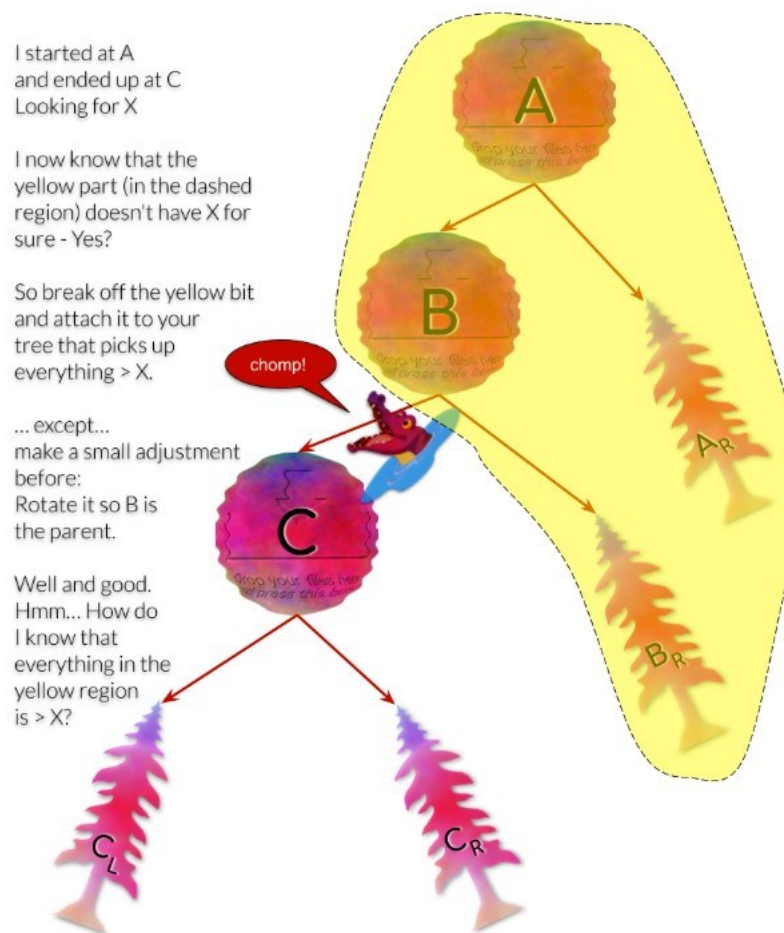


Figure 1. The zig-zig dance⁴⁹

⁴⁸ Why these particular sequences of moves (involving rotations and all)? Can you choreograph any other move sequences that provide the same or better run-time guarantees than this does?

⁴⁹ If you did everything except rotation, what happens? Do you still end up with a tree rooted at X (or its neighbor)? What does rotation give you?

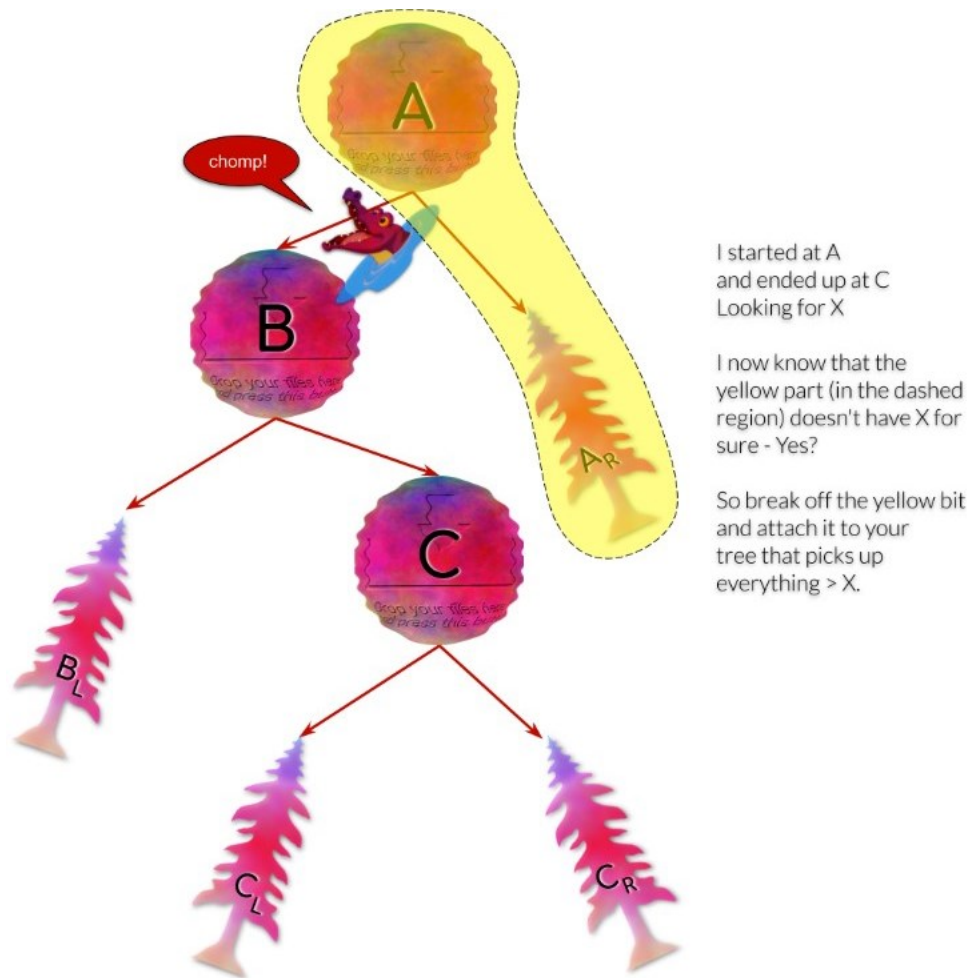


Figure 2. The zig zag dance

Merger

After you zig and zag your way through the middle tree, throwing its branches left and right, you will eventually arrive at a node where you have either found your target, X, or you know for sure that you will not find it. Figure 3 shows how to handle this situation:



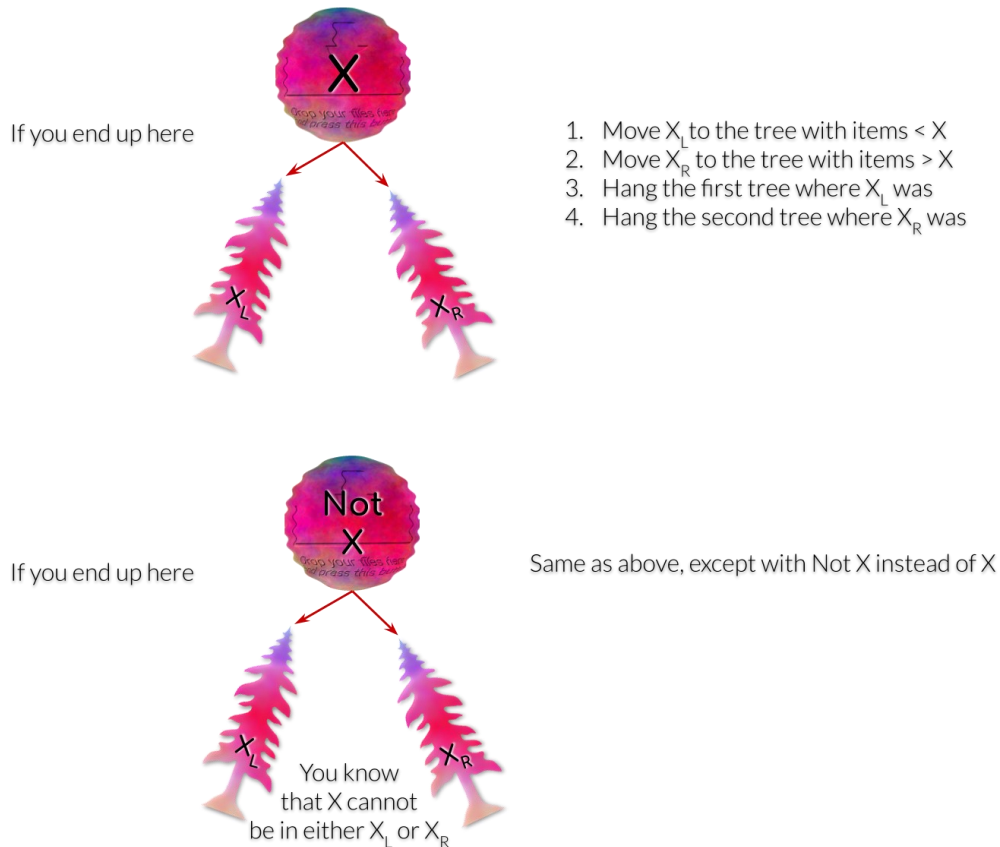


Figure 3. The final step. Merger.

The move sequences you need to execute as you search for your target can be found in multiple reference materials. Discuss the miniquiest *moves* in our [sub](#). Use pictures if you can. And also, ponder:

1. How can you be sure that your particular sequence of moves results in a tree whose worst case access time is $<$ what than you started with?
2. What if it is not $<$ but rather \leq ? Is this a possibility? Or is that what you're trying to avoid?
3. What happens when you don't find the target in the tree?

Brainstorm your own moves, your own dance, and try to prove other desirable things that your choreography will yield (does not need to be restricted to running times).



Starter code

I managed to get a fuzzy pic of the very top of the `Tree_Algorithms.h` file. And then the quest master said "What? You already took almost 20 of my 160 lines!"

So I couldn't risk going back for more. But I got an ok to put what I got up in Figure 4 here.

```
11
12 #include "BST.h"
13
14 class Tx {
15 private:
16     template <typename T> static void _splay(typename BST<T>::Node *p, const T &x);
17     template <typename T> static void _rotate_with_left_child(typename BST<T>::Node *p);
18     template <typename T> static void _rotate_with_right_child(typename BST<T>::Node *p);
19
20 public:
21     template <typename T> static const T &splay_find(BST<T> &tree, const T &x);
22     template <typename T> static bool splay_contains (BST<T> &tree, const T &x);
23     template <typename T> static bool splay_insert(BST<T> &tree, const T &x);
24     template <typename T> static bool splay_remove(BST<T> &tree, const T &x);
25
26     friend class Tests; // Don't remove
27 };
28
```

Figure 4. Tx - A regular class with static template methods (a handy packaging idea!)

Moar Detail

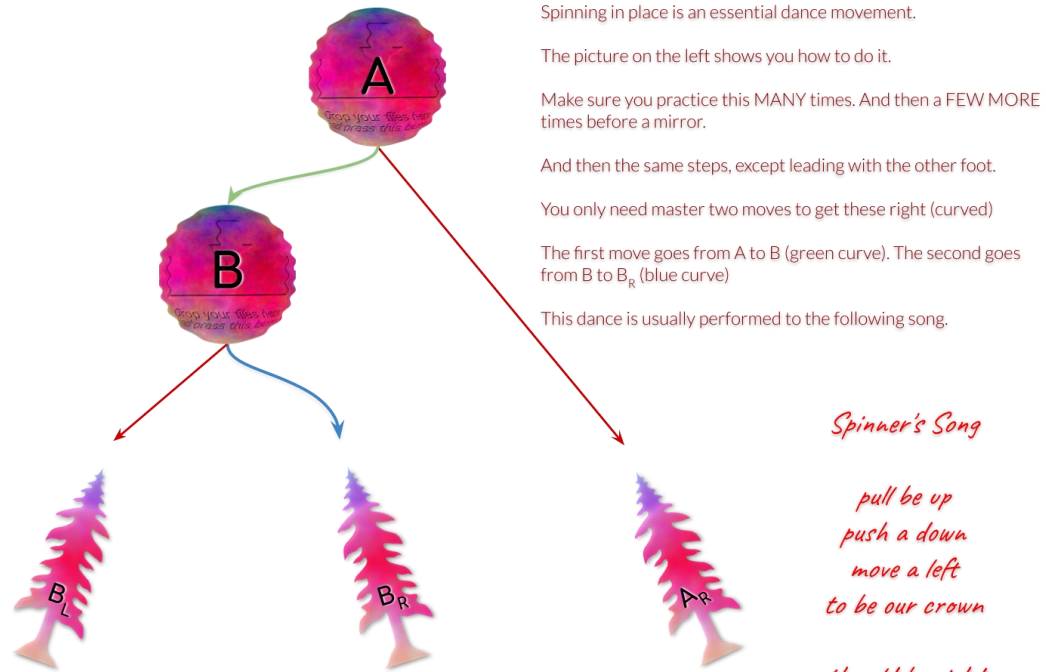
Here are some possibly rewarding facts to know about the `Tx` class:

Spin left and spin right

These take a node parameter by reference (this is important; why?) Upon return, the subtree rooted at the given node must be reconfigured as dictated by the moves of the corresponding rotation dance. If you want to review how to spin (which you presumably did a lot of in AVL trees), see Figure 5 for the choreography of the *Spin-with-your-left*. In the code, the corresponding method would be called `_rotate_with_left_child()`



little gangaram playing questball



This move sequence can be used to segue between hoppets in the dance of Dindin a'Dash.

If you got all the steps right, you should end up at:

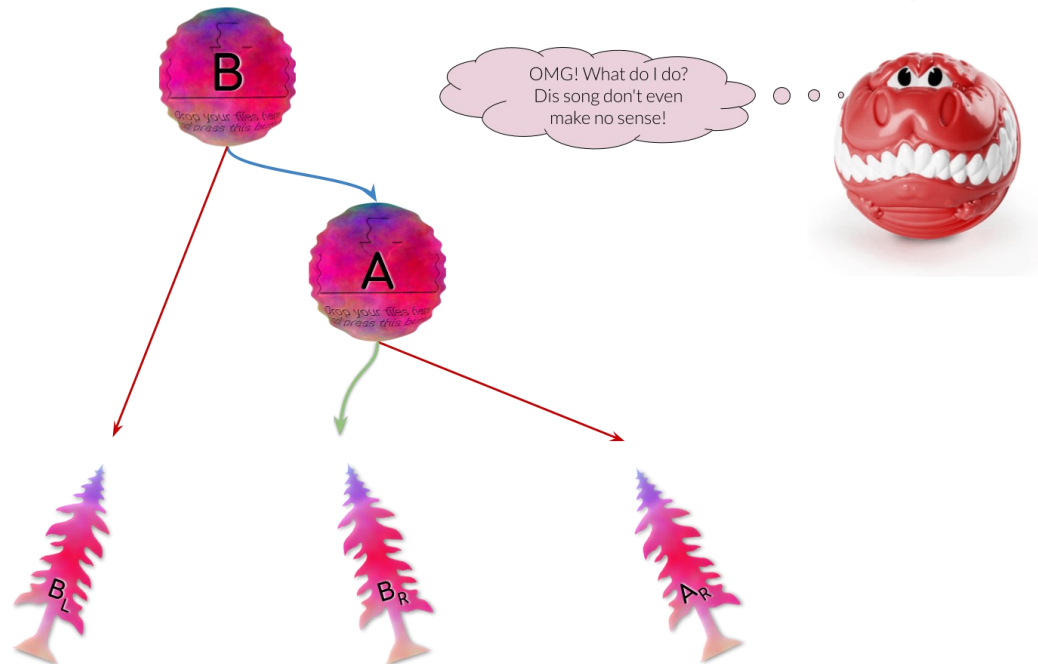


Figure 5. Moves for the *Spin-with-your-left* dance

Splay

This is the biggie. When invoked it must splay the tree rooted at the given node for the given target value. Note that the node is passed in by reference as usual.

Your implementation must be the top-down approach described in this spec. Also, it must be iterative (not recursive).

Find and Contain

Like before, `find()` should return a reference to the found item or throw an exception. But `contains()` should always quietly return with a bool. `contains()` should invoke `find()` and `find()` should make its decision by examining the root of the input tree after splaying it for the target.

Insert

To insert a new value into the tree, first splay it on the given value. Now look at the root of the splayed tree.

If it is equal to the new value, it was already there in the tree. No need to do anything. Return false to say so.

If the root is not equal to the new value, then you can dismantle the splayed tree and stick its parts under a brand new node created to contain the given value. See Figure 6.

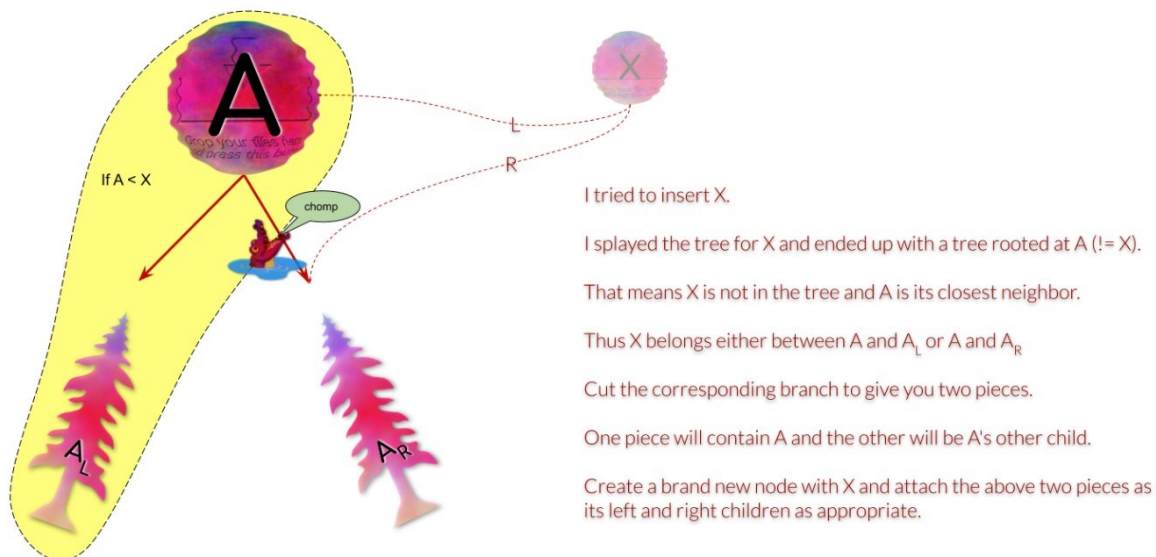


Figure 6. Inserting X whose closest neighbor in the tree is A

Remove

First, splay the tree on the given value like you did for `insert()`.

If the splayed root is not equal to the given value, it was not in the tree to begin with. Return false to say so.

If the root is equal to the given value, it must be removed. How?

Simply splay one of its children (I'm afraid it has to be the left child to agree with my reference implementation) on the given value. Since this value is not present in the left sub-tree, it will bring its closest smaller neighbor up to the root *with an empty right child* (why smaller neighbor and why empty right?)

Now all you have to do is to break the right branch off the original and attach it to the left child's right. Y now becomes the new root. See Figure 7.

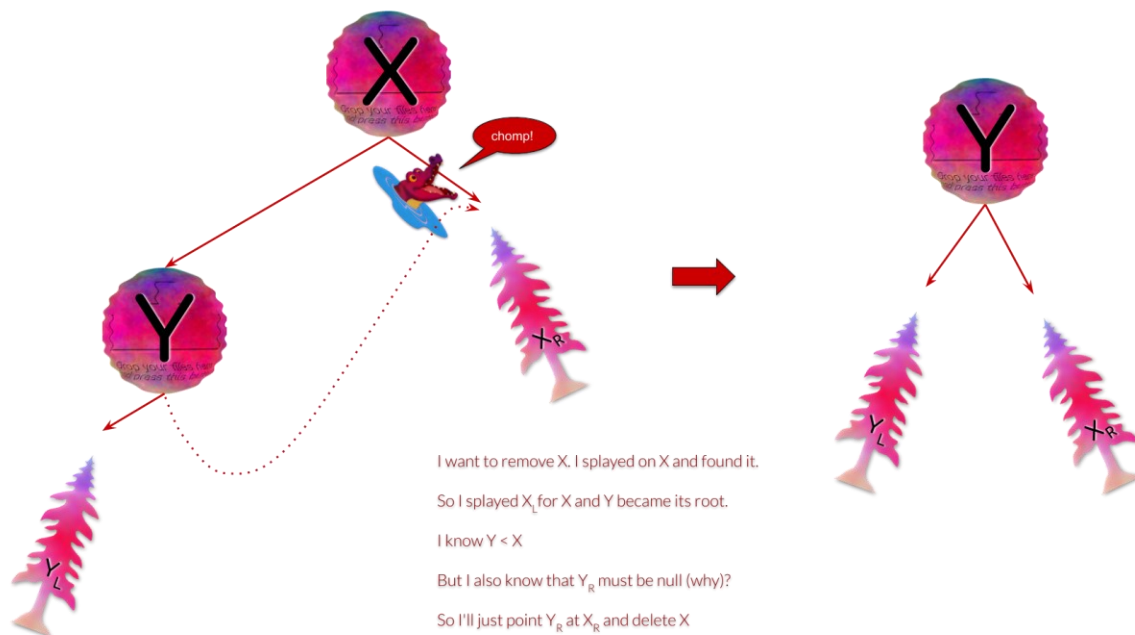
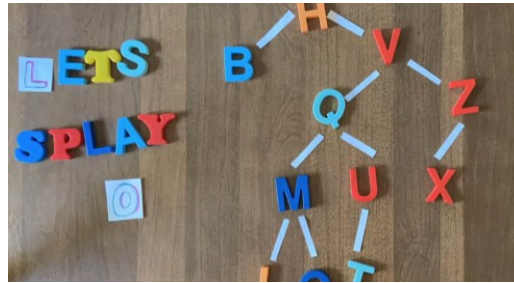


Figure 7. Remove.



Icing (Edit Jun 1, 2021)

Thanks to https://reddit.com/u/lane_johnson, I'm able to share the following really cool video he put together. It's at the end of the spec for a reason. Understand the concept first, try and fail many times, and then watch this video for maximum enjoyment. Lane is currently a CS professor at Foothill.



[Lane's Splay Vid](#)

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files⁵⁰ into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

Happy Questin',

&



nonlinear gator

⁵⁰ `BST.h` and `Tree_Algorithms.h`

*I'm a Kingaroo! I own dis ring! I hop to places never seen.
Who be you? And what dis thing? Stops for spaces in-between?*



I hoppet and then I boppit

How to Hop in Hash Tables

previously by Michael Locuff

In this cool and easy quest, the first of its kind, you get to play with two kinds of probing techniques in hash tables. The first one does linear probing for a vacant cell and the second does quadratic probing.

Sounds scary? Well, it don't have to be. After you're done reading any reference material, you'll see there's nothing to fear about these two new beasts.

In fact, this quest turned out to be so easy that I got worried if you folks will start complaining. I asked the Quest Master. He say

"Well Then. Turn off the lights."

"What a kewl idea! I much much likes."

From this quest on, you're gonna have to learn to fly with your own inner lights. No more feedback or diagnostic debug output from the site. You'll have to train your good sense to figure out what's right to implement where. Maybe I'll speak up just a liiiiittle bit in this one. But I better learn how to pipe down asap.

'Cuz in a few weeks, you're gonna be flyin on yer own and you can't rely on me to stick around for like ever.



Overview of Probing in Hash Tables

Hash tables⁵¹ are the backing stores for your dictionary objects, like in Python or JS. I'm not gonna review the idea behind hash tables here. Hit your favorite reference material.

In this quest, you'll implement two different ways to overcome the hash table collision problem. By *collision* I mean the situation when two items hash and index to the same value. Keep in mind that a hash table typically has multiple items with the same hash because the hash function is many-to-one. When you ask for a particular record, you can imagine the equivalent of a (likely linear) search happening for the desired key over all keys with the same hash as yours. (Is this always right? Discuss why. Or why not. Or when not.)

There are many ways to tackle the collision problem. For example, you could make a vector, linked list, or tree of all items with the same value, but different on a secondary key, etc. You can get as fancy as you want.⁵²

⁵¹ What do stilts say about hash tables?

⁵² Is this how they do it in Hanoi?

But most of us would try a few simple techniques first.

If an item indexes into a location that is already occupied, then simply move it to the next available location. See Figure 1.

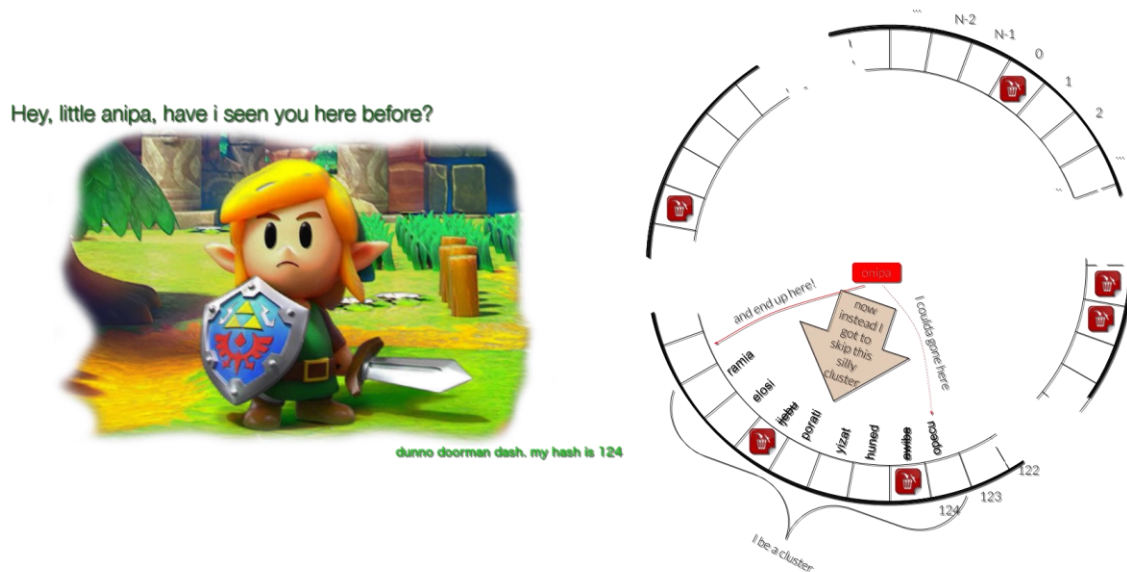


Figure 1. Probing for a vacant cell (Assume $N \gg 124$)

This technique is called *linear probing*. This is the first way in which you will try and solve the collision problem.

You must implement a class called `Hash_Table_LP`. When trying to search for an item in it, you must essentially probe for it linearly starting at the item's indexed *hash*⁵³ in your array. When trying to insert a new element, this is how you eventually end up at a vacant cell in the array after having looked over all non-vacant cells that didn't match.

Figure 2 shows a fuzzy photo of the `Hash_Table_LP` template class.

⁵³ How is this like an ant?

```

template <typename T>
class Hash_Table_LP {
protected:
    struct Entry {
        T _data; // payload
        enum STATE { ACTIVE, VACANT, DELETED } _state;
        Entry(const T &d = T(), STATE st = VACANT) : _data(d), _state(st) {}
    };

    static const size_t DEFAULT_INIT_CAPACITY = 3; // first odd prime. (Don't change)
    vector<Entry> _elems;
    size_t _size; // doesn't count deleted elems
    size_t _num_non_vacant_cells; // does
    float _max_load_factor;

    virtual size_t _get_hash_modulus(const T &item) const; // uses Hash(item), ext.
    virtual void _rehash();

    // Most common overrides
    virtual bool set_max_load_factor(float x);
    virtual float _get_biggest_allowed_max_load_factor() const;
    virtual size_t _find_pos(const T &item) const;
    virtual void _grow_capacity();

public:
    Hash_Table_LP(size_t n = DEFAULT_INIT_CAPACITY);
    size_t get_size() const { return _size; }
    bool is_empty() const { return _size == 0; }
    bool contains(const T &item) const;
    T &find(const T &item);
    bool clear();
    bool insert(const T &item);
    bool remove(const T &item);

    class Table_full_exception : public exception {
    public: const string to_string() const throw() { return string("Table full exception"); }
    };
    class Not_found_exception : public exception {
    public: const string to_string() const throw() { return string("Not found exception"); }
    };

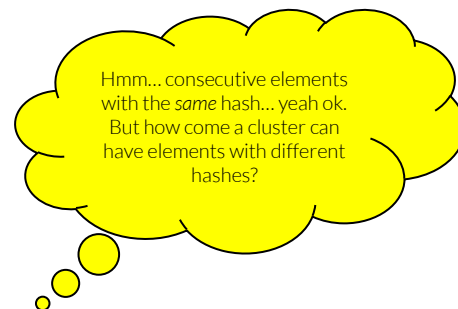
    friend class Tests;

```

Figure 2. The Hash_Table_LP Class

As you may have been able to guess already, this linear probing technique is susceptible to the formation of clusters. A cluster is a collection of consecutive array elements that may or may not have the same hash. Once you enter a cluster, you are doomed to repeated collisions until you break out of it.

Just like a COVID-19 hotspot, the likelihood that a cluster will grow is directly related to its size. The bigger the cluster, the more likely that it's going to get even bigger (without proper cluster distancing measures). Why? Hit our [sub](#) and discuss the surprisingly efficient way in which a cluster grows (but please don't bring covid into this any more. Hash Table clusters are interesting enough by themselves).



How do we keep the cluster sizes optimally small (and thus better scattered)? One way is to make sure we always maintain sufficient free space in the table, which lets us implement effective cluster distancing measures.

So we deliberately introduce some redundancy into our data structure to avoid worst case behaviors. It's kinda like underbooking your cruise seats so you never have to make a potential passenger wait endlessly while you look for a free room.

As a consequence of this redundancy, our hash table backing data stores (vectors) will no longer have 100% occupancy rates. Some fraction of cells in the vectors will always have to be vacant. This is the overhead of the hash table.

Load Factor

When your hash table has relatively few elements, then the fraction of its vacant cells is high. Incoming elements get inserted without too much effort. You can say that the load on the hash table is low because it's not really doing a great deal of work.

As it gets used, you may insert and delete elements, and clusters begin to grow. Note that a cluster, once formed, is permanent and never shrinks until a rehash event. No cell within a cluster is a vacant cell, and a cell, once marked non-VACANT, never gets marked VACANT again. And so the hash table's load can only increase with use.

This lets us define a property called the *load factor*. This is the ratio of non vacant cells to the actual capacity of the backing data store. Since the best programmers also tend to be the *most lazy*⁵⁴, this is the quantity we want to control from getting way outta hand. Any time an insert operation looks like the next one's gonna take a lot of work, we'll simply rehash the whole table and bring the load factor down again.

How much? Let's say you decide that the maximum load your employees can handle while doing a decent job is 75%. Then just before the rehash, your table would have been at 75% load using a backing store of N cells. After doubling the size of the array, you would have a backing store of $2N$ cells, but still only $0.75N$ elements. Thus the load factor would go from 0.75 to 0.375 (half the value).

As it happens, 0.75 is the default maximum load factor you should allow users to set in your linear probing hash tables.⁵⁵ You will give your users the ability to set their own maximum load factors to any number they want as long as that number is *reasonable* ($0 < n \leq 0.75$). Like a black Ford.



⁵⁴ If you're just ordinary lazy, you don't wanna work if you don't gotta. If you're like extraordinary lazy, you don't want anyone to work if they don't gotta. But if you're like... the *most lazy*, then you don't even want dem machines to work if they don't hafta.

⁵⁵ Hit the reference material or our sub to talk about the derivation of this optimal value. Why 0.75?

To incorporate this into your implementation, calculate the load factor in your `insert()` method (after inserting). If the load factor of your hash table has gotten way too big (as big as your `_max_load_factor`), you would say (Maybe to yourself):

"Ok. That's it. Things are getting too waaay close for comfort. I gotta breathe man!"

At this time you must `rehash()` your table by growing its capacity to twice⁵⁶ its previous size and reinserting all currently active elements into your new backing store.

To minimize the likelihood of your floating point result being different from mine, try to model your load-factor calculation and comparison like I do in Figure 3.

```
if (_num_non_vacant_cells > _elems.size() * _max_load_factor)
    _rehash();
```

Figure 3. Checking if the load factor > `_max_load_factor`



Hash_Table_LP Details

Obviously, not all of the detail below is reward-worthy.⁵⁷

`_size` and `_num_non_vacant_cells`

`_size` is what users of your class think to be the number of elements actually in the table. Even though they both start off the same, the latter does not get adjusted upon a delete.

`_get_biggest_allowed_maximum_load_factor`

This private method should simply return the floating point value 0.75. In subclasses of `Hash_Table_LP`, you'll have the opportunity to override this virtual function to return different thresholds if you want.

the constructor

It not only sizes the backing data store (`_elems`), but also initializes the member variables.

It should also set the table's max load factor to the optimal value, which is also the biggest allowed value.⁵⁸

When you have considerable visibility into the domain in which you might deploy a hash table, you get to make such fine adjustments as deliberately underloading or overloading your table because your budget may dictate more compute/less memory or vice-versa. How's that for fun?

`set_max_load_factor`

As discussed already, it should fail and return false if the given load factor is either non-positive or greater than your biggest allowed max load factor.

⁵⁶ Why double? And is it always double? Exactly?

⁵⁷ But you still gotta try and do everything to make the customer a happy chappie. Make 'em yell "Hooray" in ecstasy.

⁵⁸ See https://www.reddit.com/r/cs2c/comments/nhvool/lp_find_pos_help/gyzbkxg

_get_hash_modulus

The hash modulus of an item `X` of type `T` is defined as the result of the modulus operation `Hash<T>(X) % _elems.size()`

Sometimes people loosely call this the hash of `X`. No. The functor⁵⁹ `Hash<T>(X)` returns the hash of `X`. The remainder after modding with the size of the backing store is the hash modulus. It is guaranteed to be bounded by the array size.

How is the hash itself calculated? Clearly it's not possible for you to define the `Hash<T>()` method for type values `T` you don't even know yet.

So you don't do it. You're off the hook for this one. A user of your hash table class (that'd be me) will have to define the relevant `Hash<T>` functors for the types they intend to use with your hash table.

For instance, if I want to instantiate a `Song_Entry` hash table, I had better make sure I define

```
template<> size_t Hash<Song_Entry>(const Song_Entry &s) {
    // return a size_t computed from the key of the s
}
```

somewhere in my code or the loader will complain about unresolved externals. The above syntax means "`Hash` is the name of a template function (<>) and following it is a specialization of it for the `Song_Entry` type. A *specialization* is one concrete implementation. Here's another, and one that I actually use:

```
template<> size_t Hash<string>(const string &s) {
    return hash<string>{}(s)
}
```

This says that `Hash<string>(string &s)` simply uses a specialization of the library's `std::hash<>` class to calculate the hash of a string using an anonymous functor of type `hash<string>`.

In your testing, you will have to define your own hash methods like you see above. But you shouldn't submit `Hash` specializations for the `int` and `string` data types since I'll be using my own.

grow_capacity

It should double the size of the backing vector. Since your default `Entry` constructor gives you a `VACANT` entry, the `resize` will automatically fill the new cells with `VACANT` entries.

rehash

It should save a copy of the current elements, `grow_capacity()` and then insert all `ACTIVE` elements in the old array into the new hash table.

⁵⁹ A functor, in case you aren't familiar with it from GREEN, is simply an object of a class that has defined the `operator ()`. Yes, c++ lets you override parentheses too. If `My_class` defines `operator ()` and `my_obj` is of type `My_class`, then I'd be able to invoke `my_obj ()` as if it were a function. Cool?

`_find_pos`

This is a core private helper method that is used by `insert()`, `remove()`, and `find()`. A few important design decisions influence its algorithm. You have to make the same decisions I made in my reference code in order to pass miniquests. What are the engineering implications of these decisions?

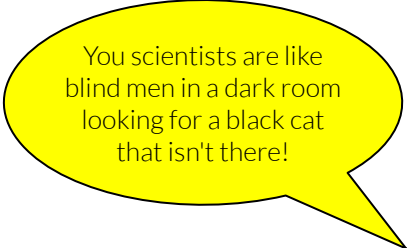
Note that `_find_pos()` scans the backing array linearly for a target element. Its scan is not terminated by ACTIVE cells nor DELETED (deleted) cells, but only by VACANT cells. What happens if the `_elems` array has no more VACANT cells? Note that `_find_pos()` cannot rehash a table (it is, in fact, marked `const`). Since the backing store is circular, there is a real risk of getting stuck like an ant on a mobius strip if someone tries to look for an element that is NOT in the table and the array is full.⁶⁰

"Well" you could say, "That's what we have the `_max_load_factor` for. Using it will make sure that the table is never more than 75% occupied."

All good. But to be on the safe side, you must code defensively, and let the method immediately fail (return `string::npos`) when asked for the position of an element in a table in which the backing store is completely full (unlikely though it is).

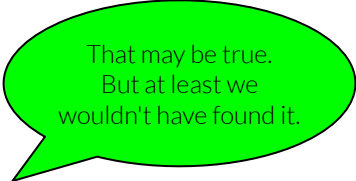
If you put your engineering hats on and survey the situation, you'll find that the following are some of the consequences (some welcome, and some we willingly accept in return for other goods):

1. `_find_pos()` will falsely report that a present element is absent in a backing array that has run out of VACANT cells.
2. `_remove()` will fail to remove a present element in a backing array that has run out of VACANT cells.



You scientists are like blind men in a dark room looking for a black cat that isn't there!

Note that this situation is unlikely to happen if your maximum load factor is less than 1. The trade-off we're talking about concerns the programming experience of an exceedingly small number of programmers who ignored warnings, messed around with the private internal settings of the hash table (like its max load factor), filled up its backing store *completely* despite suffering horrible performance on the last few inserts, and then asked `_find_pos()` to behave when asked for an element that isn't there.



That may be true. But at least we wouldn't have found it.

Our decision implies that we'd return an incorrect result in this extreme case rather than loop forever. Better this silly programmer's code crash than chew up good machine cycles for no good reason.

⁶⁰ What are some other ways you can handle this situation? Can you get your GREEN *ant* technique to work here? Why or why not?

What's the alternative? Hit our [sub](#). These are the kinds of decisions you will be making in your life as a programmer.

find and contains

`contains()` should return a boolean value, while `find()` should return a reference to the found item and throw the appropriate exception if it isn't able to.⁶¹

insert

Use `_find_pos()` to locate either a VACANT cell or the element itself. If you found the element, and it is ACTIVE, then you can simply return false (you didn't have to do anything).

If you found the element but it was marked deleted, then you can simply reset its state, adjust the hash table's size and return.

If you landed on a VACANT cell you will put the element there, and also need to update your count of non-vacant cells so that your load factor calculations will be correct.

remove

This is just like `insert`.

Well, no. But you get the idea.

Use `_find_pos()` to locate the item. If it isn't there to start with, simply return false (as you would if the table is full). If not, all you have to do is to change the state of the entry from ACTIVE to DELETED.

⁶¹ What are the pros and cons of returning a const reference rather than a plain reference? Should you do one rather than the other?

Hash_Table_QP Details

If you've completed the implementation of the `Hash_Table_LP` class successfully, this one will likely just fall outta your bag for almost free.

Figure 4 shows a picture of the `Hash_Table_QP` class.



```
template <typename T>
class Hash_Table_QP : public Hash_Table_LP<T> {
public:
    Hash_Table_QP(size_t n = Hash_Table_LP<T>::DEFAULT_INIT_CAPACITY) : Hash_Table_LP<T>(n) {
        this->_max_load_factor = 0.49;
    }

protected:
    virtual float _get_biggest_allowed_max_load_factor() const;
    virtual size_t _find_pos(const T& item) const;
    virtual void _grow_capacity();

    // Private helper
    static size_t _next_prime(size_t n);

    // Don't modify below
    friend class Tests;
};
```

Figure 4. The `Hash_Table_QP` Class

As you can see, it is a subclass of `Hash_Table_LP`. You can leverage almost everything you did before. You only need to override the constructor and three protected methods in the class def. There's also a new private helper method called `_next_prime()`.

When your workhorse methods (`insert`, `remove`, etc.) are invoked by a user, and they in turn invoke any of the three overridden methods, the compiler will make sure the methods attached to the correct class (whether LP or QP) are used. Note that we're not accessing the overridden methods through pointers. They're just straight method calls on objects that the compiler can handle in the usual way (i.e. no runtime resolution required).

the constructor

Not much to do here. You can chain back to your base class constructor and have it do all the work. However, once that is done, you must reset the `_max_load_factor` member to the maximum permissible value for quadratic probing. To do that you must first override the `_get_biggest_allowed_max_load_factor()` method to return 0.49 (Why 0.49? Hit your refs and discuss in our sub).

`_find_pos()`

Essentially, this is almost the same as the `_find_pos` from your LP table. However, when you have a collision and need to examine another cell, you don't look at the next higher numbered cell, but rather a cell that is *the next perfect square away* from the current index. What does that mean?

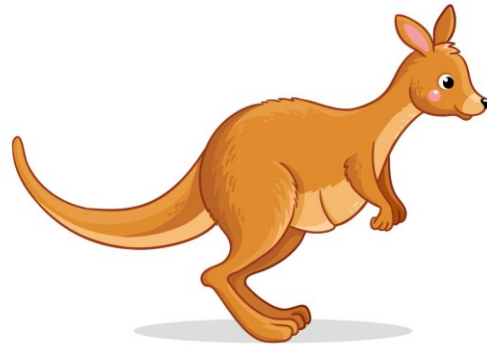
Suppose you're looking for an item X , and the hash modulus of X is K . If you find that `_elems[K]` is occupied, then

- the next location you try will be `_elems[K+1]` (so far the same as in LP)
- If that is also occupied, then you try `_elems[K+4]` ($2^2 = \text{next perfect square after } 1$)
- If that is also occupied, then you try `_elems[K+9]` ($3^2 = 9$)
- etc.

How can you be sure that when you hop around your table like that you'll eventually end up at a location you've never seen?⁶² Discuss in our [sub](#).

Leverage the useful fact that the next perfect square can be found simply by adding the next odd number to the previous one. Don't hit the math library. See:

- $0 + 1 = 1$
- $1 + 3 = 4$
- $4 + 5 = 9$
- $9 + 7 = 16$



Cool 'nuff?

`next_prime()`

If you're here, it means you have probably found a way to ensure that you won't end up hopping endlessly.⁶³

`next_prime(size_t n)` should return the least prime number greater than n . You need to test every number $> n$ for primality and return the first one that passes. Use a simple variation of a common test for primality: A number is prime if it leaves a non-zero remainder when divided by any integer > 1 that is less than its square root (This is the only place where you should call a math library function, and BTW, why square root?)

But rather than test for divisibility by every positive integer $< \text{sqrt}(\text{val})$, you can skip a candidate and move on to testing its successor as soon as you find that it's not a prime. Since you know that every number (except 1) has a prime factorization,⁶⁴ we can come up with an improved algorithm:

A number N is composite (not prime) if:

- it is divisible by 2 or 3
- or it is divisible by some other prime less than \sqrt{N}

"But there's the rub" you say. "How do I find all primes less than \sqrt{N} ? Do I recurse? Do I XYZ?"

⁶² And what happens if you do?

⁶³ If your table has a load factor < 0.5 and the size of its backing store is a prime number, then you can show that you don't gotta hop endlessly no matter what.

⁶⁴ Curious why? You may want to check out a Discrete Math overview.



"Whoa there, Buster. Stop right there."

Here's a cool way to test for divisibility by all primes less than some number. Exploit the fact that every prime number has at least one neighbor which is divisible by 6. Since a prime > 2 is necessarily odd, both its neighbors must be even. In addition, if one of its neighbors is not divisible by 3, the other must be. Leverage this to come up with this test:

A number N is composite (not prime) if:

- it is divisible by 2 or 3
- or if it is divisible by either $(6k-1)$ or $(6k+1)$ for any positive $k \leq \text{a sixth of}^{65} \sqrt{N}$

Together, $(6k+1)$ and $(6k-1)$ cover all known primes ≥ 5 for various positive values of k . Thus, if a number is not divisible by any of them, it can't be divisible by any prime. Of course the less efficient way is to test for divisibility by all primes $\leq \sqrt{N}$. The challenge is to do it with as few as or fewer tests than required by this algorithm.

This technique leveraged information about the two immediate neighbors of the candidate. Can you get more information by looking further? If so, where do you draw the line between spelling out special cases and letting the computer crunch? This kind of reasoning is another kind of seasoning you will be adding to your code sandwiches in the future.

`grow_capacity()`

What can I say? It's the same as its elpie cuzzin, but it got to set the size to the next prime $> 2N$.



⁶⁵ Wee..elll... That's just loosetongue for a sixth of the ceiling of the root of N . But you get the point. In Salestongue you might say *"Add the Acme test screener to your cart (Improves run time by as much as 500%)"*

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files⁶⁶ into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

Happy Questin',

&



⁶⁶ Hash_Table_LP.h, Hash_Table_QP.h

*when turns are sharp, i don't cut corners questing for my pivot
i'm a coding shark. i have honor. i'll code it and i'll give it*



i sort em sorry small fry to one side - my search for bigass tuna simplified

Pivoting

Hooray! In this quest you get to implement quicksort all by yourself. All 10 lines of it, I think. This may just be your quest with the maximum reward per byte ratio. Actually, no. I forgot that time we danced in Hanoi.

Anyway, before you get to do this quest, you have to pass the free entry challenge.



Entry Challenge

Implement the following global scope in-place sort method in a file called `Entry_Pass.cpp`

```
void my_questing_sort_in_place(vector<int> &elems);
```

On return from this method, `elems` must be sorted in place in non-descending order.⁶⁷

There are two ways to get through this entrance test.

One is to make your method secretly invoke the standard library's `sort()` on the vector. I'm not gonna check. But the coding shark don't do it that way - even to just peek and see what's ahead.

In fact, if you have never implemented a sorting algorithm before, I would say that you are among an EXCEEDINGLY LUCKY FEW. Don't squander away this incredible opportunity to appreciate some of the nuances of sorting. Do it yourself first without looking up any references. This is a fantastic way to get to appreciate the magnitude of work that has gone into crafting about 10 lines of code that are *as simple as possible, but no simpler*.

BTW, in-place means you don't get to allocate more space on the heap. You are allowed to recurse if you pass the elements by reference, but you don't have to.

Minis

Ok. Now comes the exciting part. But before you start, here is a helpful general note about this quest: If any of your methods needs a vector to be passed in as a parameter by reference, I'll make sure that the vector is not empty. This should save you from having to build in redundant checks in your code for things that can be easily controlled from outside.

⁶⁷ That means there should be no pair of elements in the vector that are out of order. If you visually imagine your array spanning cells from left to right, then you'll call out all pairs in which the right is smaller than the left as an out of order pair. These also go by the simpler name, *inversion*.

Figure 1 shows you a picture of a template class⁶⁸ called `Pivoting`. It is a wrapper for a collection of algorithms you could conceivably implement, all of which share the common theme of *pivoting*. This quest concerns the implementation of three such public methods:

- `find_kth_least()`
- `find_median()`
- `do_qsort()`

and some private helpers.

```
template <typename T>
class Pivoting {
private:
    static size_t _partition(vector<T> &elems, size_t lo, size_t hi);
    static void _do_qsort(vector<T> &elems, size_t lo, size_t hi);
    static T _find_kth_least_elem(vector<T> &elems, size_t lo, size_t hi, size_t k);

public:
    static T find_median(vector<T> &elems);
    static T find_kth_least_elem(vector<T> &elems, size_t k);
    static void do_qsort(vector<T> &elems);

    friend class Tests;
};
```

Figure 1. The Pivoting Class

Fun facts

- No news may be good news. But it most certainly is no good news. If you don't implement any of the methods and simply submit an empty template class called `Pivoting`, I'm not gonna complain. But trophies will automatically appear for things done right, and the password will automatically appear after a certain number of trophies have been won.
- In what order should you implement the methods? Should you implement one before another? In some cases the answer is obvious. In others, not so much! It's like a metaquest. Go figure.



⁶⁸ How might you generalize this implementation even further? I mean, it's already a template class, allowing you to operate on vectors of any type that supports the less than comparator.

Rewarding facts

`_do_qsort(vector<T> &elems, size_t lo, size_t hi)`

You're being given a vector of things that guarantee the less than operator can be used to compare them.

Not only should this method quicksort the segment of the vector spanning `elems[lo] ... elems[hi]`, but it should do it using the exact same strategy (including partitioning) that the reference code does.⁶⁹ Or you'll ouch out with not much more else to say.

After you partition the array to find the partition index `k`, you may find it helpful to sort the sub-arrays `[lo ... k]` and `[k+1 ... hi]` rather than some other division. You can usually afford one quick check at the start of the method and immediately return if possible. This kind of early-checking has saved me many convoluted downstream checks.



`find_median(vector<T> &elems)`

Return the median element from `elems`. That'd be the middle element if `elems` was sorted.

Except you don't have to sort `elems`. How much faster can this be than if you had to sort it?

What if there's no one middle element (even-sized vectors)?

In that case, rather than mess around with arithmetic requirements on the template type, simply return the second of the two middle items.⁷⁰

If you had to choose between `find_median` and `get_median`, which would you pick for the name of this method?

`_find_kth_least_elem(vector<T> &elems, size_t lo, size_t hi, size_t k)`

This should return the element that would be at index `k` in `elems`, *if `elems` was sorted*. But you don't have to sort it. Sorting is how the non-programming types do it if they don't know better.

Leverage these two important insights:

1. Your partition⁷¹ algorithm will divide up the vector around a *pivot* value into 2 pieces.
2. Your *k*th least element is *certainly*⁷² located in at most one of them (you can afford to completely ignore the other).



Remember that your *k*th least element will keep jumping around in your array as you search for it.⁷³ That's perfectly fine. You only need to return it after it has settled down. When it has settled it will be in index `k` (but not *only* then).

⁶⁹ There's only a few different ways to do it right. So try 'em all and one might click.

⁷⁰ What would it take to make the method return the correct arithmetic median of the vector?

⁷¹ Partitioning is like tiling. The partitions must not overlap, and together they must cover the whole area.

⁷² *certainly* or *only*?

⁷³ After each jump, it will usually only have about enough energy left for a jump half as long as the previous one. Why?

You can recursively invoke yourself after adjusting for just the one partition in which your target can be found. By ignoring half the search space (on average) at every recursion, you can find the target in at worst linear time.

There are MANY corner cases here, and although this method is only a few lines long, it could take you many hours to get right if you go down a wrong path. I wish you a struggle that will be worth it for years to come.



```
find_kth_least_elem(vector<T> &elems, size_t k)
```

This is the public facing version.

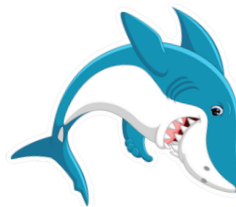
Just so we're all on the same page, let's define k to be the array index, rather than conventional notions of 1st, 2nd, etc.

You should check if k is valid and invoke your private helper with the appropriate parameters. If k is not valid, you must simply return a default T object. Here is where you should do this check. Not in the private helper. Why? Also, note that it returns a default T for invalid k , rather than throw an exception. You should be able to change this in a snap if an application demands it.

```
size_t_partition(vector<T> &elems, size_t lo, size_t hi)
```

This method should choose an appropriate pivot (described below) and partition the elements in `elems` into two disjoint chunks such that the first chunk consists of elements *no bigger than* the pivot and the right chunk consists of elements *no smaller than* the pivot. It should then return the index of the first element of the second partition.⁷⁴

This is the most important method to get right for this quest. Obviously there are many different ways of partitioning an array. And you should try at least a few different variants *on your own before* you read on (and before you hit a reference). But to pass this miniquest, you have to partition it using the strategy described below. Both your pivot as well as your final permutation need to match up with the reference's.



⁷⁴ Note that it would have crossed-over by now.

Partitioning

We want to divide up `_elems` into two disjoint parts. The first consists of all elements no bigger than a value we call the pivot, and the second consists of all elements at least as big as the pivot. If there's only one element and that's the pivot, it should obviously only go to one part.

You then return the array index of the first element of the second part. See Figure 2 for an example.

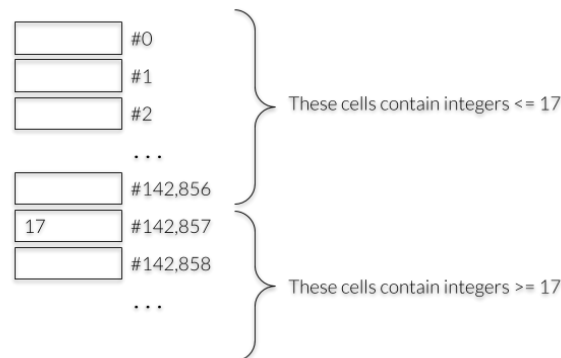


Figure 2. Suppose your `partition()` returned 142,857 and that the value at that location is 17. You can then make the above inferences.



Overview of the winning strategy⁷⁵

- Select your pivot index = $lo + (hi-lo)/2$. Your pivot is the element at this index⁷⁶
- Now set up two runners (indices) at the two ends of the array. They're going to race towards each other as fast as they can. Except:
 - The left runner can't cross cells $>$ pivot
 - The right runner can't cross cells $<$ pivot
 - When they both get stuck, check to see if they have met each other.
 - If they have, then you can consider the location of the right runner your partition point (How can you say that for sure?)
 - If they haven't met each other, then we know for sure that either (1) the elements that they are stopped at are both equal to the pivot or (2) the left element is greater than the right element. In either case, it is safe to swap them. So swap, and restart runners at the next locations.⁷⁷

⁷⁵ You must have convinced yourself by now that there are several correct strategies, but only one winning strategy. If not, this is another chance to try and find some other correct strategies.

⁷⁶ Hmm... Why the added complexity? When might this be different from the average of `lo` and `hi`?

⁷⁷ I wish you many hours of wonderful exploration as you look for cool reasons, like for choosing to allow redundant swaps (pivot with pivot) instead of skipping them conditionally (why?).

Here is an important design decision about `partition()` and its rationale. You don't have to do anything, but I'm offering this nugget here as an example of decisions you may have to make when you get to the final mile with your own algorithms.



Note that `partition()` first calculates the value of the pivot index and gets the pivot from `elems`. If you blindly follow rules laid by others you may argue that the method should check `elems[]` for emptiness before accessing it.

However, when you eventually get to squeezing cycles out of tight frequently touched code, you also get to ask how many of these checks you can eliminate if you can guarantee certain input conditions. Sometimes you say

"Well, it's easier and better to enforce that `elems` NOT be empty through the documentation." That is why the recursive `qsort()` or `partition()` methods are able to eschew MANY tests.⁷⁸

The win from this decision looks like a couple of lines of code, but when you profile your code with and without these checks, you'll notice the difference.

Where is this win?

Where is the real win in the Quicksort algorithm that you see, for example, on [Wikipedia](https://en.cppreference.com/algorithm/quick)?

The real win is not in what you see. But rather, in what you don't see.

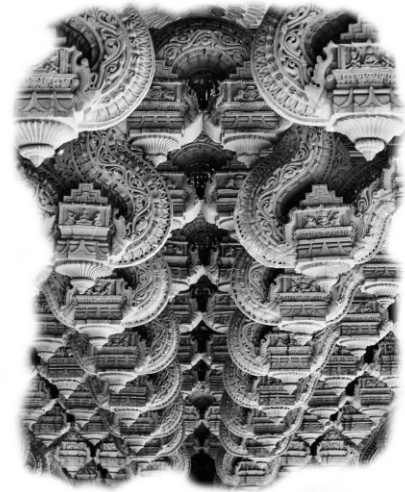
As you implement your own quicksort, you'll see that it's relatively easy to get a correct implementation with many checks here and there.

In sculpting metaphors, you can say that you have now identified the right kind of rock to work on.

Years of meticulous chipping away by many master artisans result in many of the masterpieces we see around us today.

One angle with which to polish your own quicksort is to look at minimizing the number of operations within the tight loops. This is likely to give you a solid intuition about when each element is accessed. This is also when you can start worrying about the actual machine cycles a method might consume. Some people who needlessly worry about machine cycles without a correct intuition about the algorithm don't know what they're talking about.

If all goes well you'll see benchmark results of your runs compared with reference runs and, where relevant, c++ library calls.



⁷⁸ If a conditional test can easily be moved up the code-tree hierarchy for no cost, it makes sense to check if you can factor the test out of the function it is in, into the function's invoker or even higher if possible.

Icing (Edit Jun 1, 2021)

Thanks to https://reddit.com/u/lane_johnson, I'm able to share the following really cool video he put together. It's at the end of the spec for a reason. Understand the concept first, try and fail many times, and then watch this video for maximum enjoyment. Lane is currently professor of CS at Foothill.



[Lane's Quicksort Vid](#)

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files⁷⁹ into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

Happy Questin',

&



⁷⁹ `Pivoting.h` and `Entry_Pass.cpp`

*off i flies to new hampshire
the state that screams live free or die here.*



No more munchies, or zombie sleepies. I'm free to soar the skies my dear

Code Heaps

In this quest, you get to ride a jolly good heap, and then customize it into a special heap.

Jolly Good Heap

Wee!!!! *Good* always needs to come before *special*. That's what I always say.

So before you get started on special heaps, you're gonna implement a good ol' heap, which is, like, a jolly good heap.



A jolly good heap is a regular binary min heap, implemented over an array in a pretty much standard way. So let's get it over with first.

Figure 1 shows a fuzzy pic of the Heap template class.

```
/ NOTES.
/ - T must implement operator<() via get_sentinel<T>() { return min possible }
/ - The max size of a heap = capacity - 1 because elem 0 is a sentinel
/ - Capacity is simply elems.size() (= max_size + 1)
template <typename T>
class Heap {
protected:
    vector<T> _elems;
    size_t _size;
    static const int INIT_HEAP_CAPACITY = 128;

    // private helpers
    bool _percolate_down(size_t pos);
    bool _heapify();

public:
    Heap();
    Heap(const vector<T>& vec);

    virtual bool is_empty() const { return _size == 0; }
    virtual bool insert(const T &elem);
    virtual bool delete_min();
    virtual const T &peek_min() const;

    virtual string to_string() const;

    friend class Tests; // Don't remove this line
```

Figure 1. The Heap Class

Specifics

Implement your binary min-heap by placing a sentinel at location 0 of your data array, `_elems`. This sentinel must be set in the constructor by issuing a call to a client supplied function `get_sentinel<T>()`. You will need to implement it to complete your own testing.

`get_sentinel<T>()` should simply return the lowest value an object of type `T` can have, which, in turn, is guaranteed not to occur elsewhere in the data. Write it and use it to your heart's content. But make sure your submitted code don't have it.

Once you have this sentinel in place, you can avoid a needless comparison of an index to 0 in your `insert()` method as you're bubbling elements into place. That is, traversing a chain of parents from a leaf towards the root. (Why?)

Here are some rewarding things to know about a jolly good heap:

`_percolate_down`

Let's say that an element that doesn't obey the heap ordering property in the data array is a problem element. Suppose that there is exactly one such element in the heap.

Given the index (`in _elems[]`) of this element, perform a series of moves in which the problem element is repeatedly swapped with its smallest child until it cannot be swapped any more. (Will the algorithm still work if you chose any child rather than the smallest child?)

Note that this can be implemented efficiently. You can avoid needless swaps by creating a *hole* and moving the hole to its final location before filling it with the problem element. In this location, the problem element will not be a problem element any more. (Why?)

`_heapify`

You call it when you want your heap's vector of elements to be reordered according to the binary heap property. Its running time is linear in the number of elements. (How?)

Done correctly, you will simply need to `percolate_down()` all elements from array index $n/2$ to 0, where n is the heap size. (Why is this enough?)

`insert`

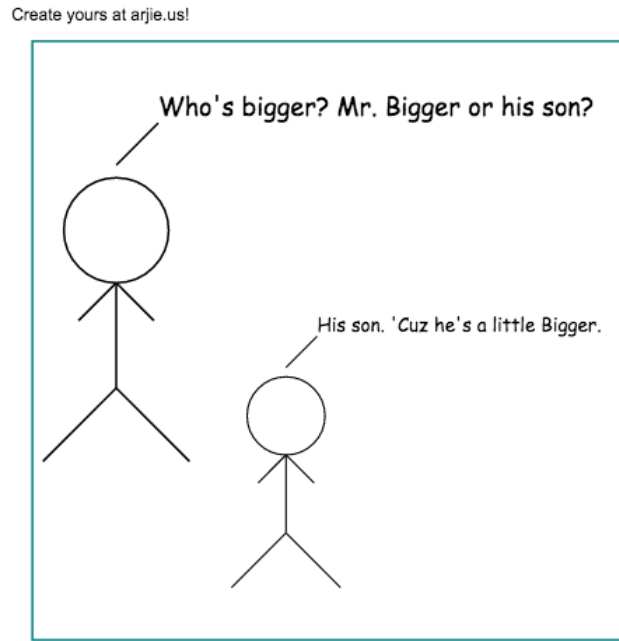
Insert the given `elem` at the very end of `_elems` (that is, at index `_size+1`). Then bubble it up into its correct place in the heap so that the heap property is restored. Double the vector size if you need to grow.

This method does the opposite of `_percolate_down()`, but it's only a few lines of code, so I didn't bother to separate it out into a private helper method. Feel free to do so if your sense of esthetics says otherwise.



To bubble an element up into its correct place, compare it to its parent. If the child is smaller than its parent, simply swap the parent and child and repeat the process at the child's new location until you can't move it up towards the root any more.

Again, note that this can be implemented efficiently by creating a *hole* and moving the hole to its final location before filling it with the element to be inserted.



I made this comic at Arjie.us, a site created by one of your senior questers (in 2018)

delete_min

Remove the element at the root of the heap. If the heap is empty, it must return false.

Deleting the root (min) element can be done as follows.

- Swap out the min element with the element at the end of the heap (the right-most element in the row of leaves in a pictorial representation of the heap).
- `_percolate_down()` this new root to its correct location in the heap.
- Adjust size accordingly.

peek_min

Return a const reference to the heap's min element in constant time. If the heap is empty, you would return the sentinel, conveniently located *you know where*. (Why do we need this method?)

to_string

No instructions, and no hints. Only an example. If you do it, all you have to do is make your string look like the one in Figure 2.

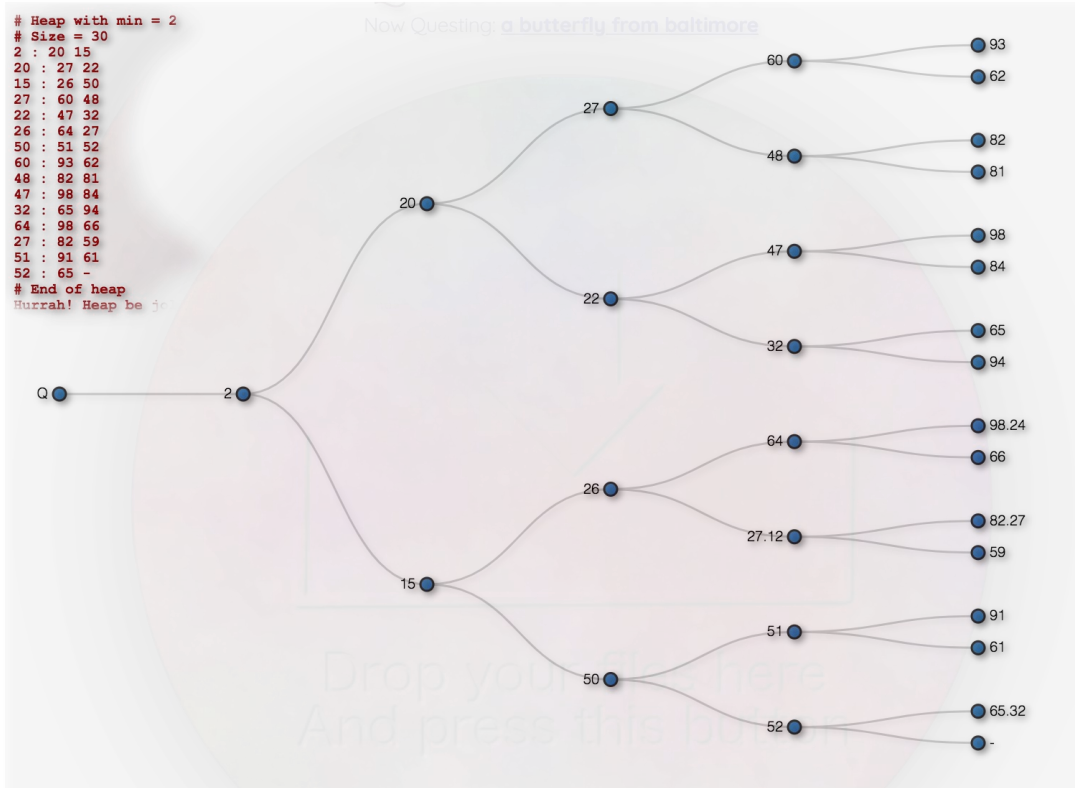


Figure 2. Figure to_string out from this Figure 2 - Ignore the fractional parts of the numbers in the depiction. I stuck 'em in there to help you distinguish between dupes.

If you want to silently skip this MQ, you can return the empty string. If you return anything else, the returned string must be legit. If not, it will result in an "EPIC Fail" which means you can't proceed (and your code/origin info was saved for review).

Remember

Users of your class are not required to furnish any more than the less-than comparison operator for the template parameter class. That is, if I specialize your template like:

```
Heap<Song_Entry> my_heap;
```

you cannot assume that `Song_Entry::operator>()` exists. You can, however, assume that `Song_Entry::operator<()` will exist.



The Special Heap

The `Special_Heap` is a subclass of `Heap`. It should behave just like a regular min heap except that it should support a new method:

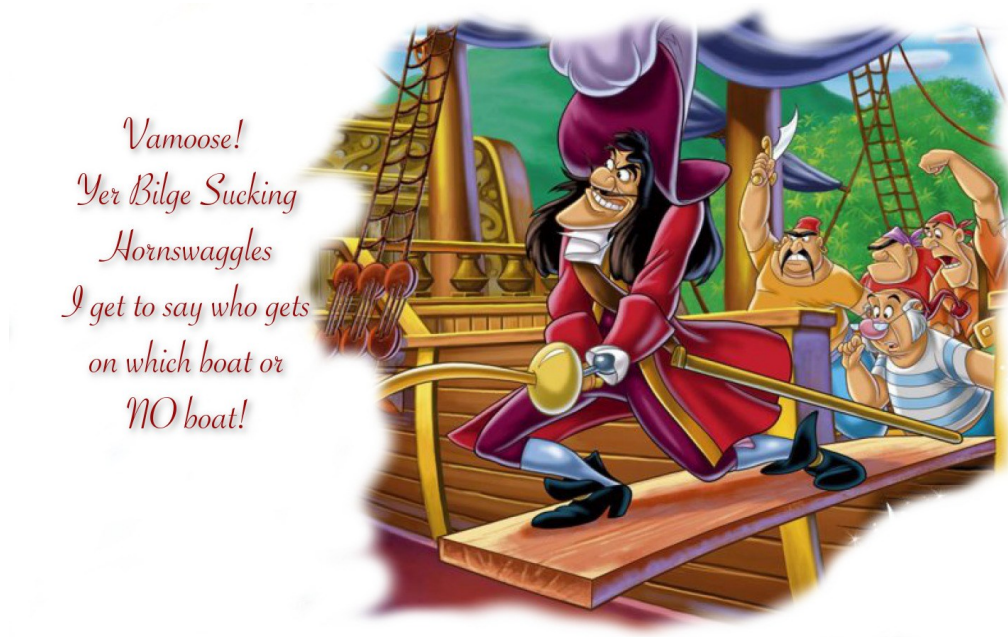
```
get_least_k(size_t k)
```

This method should return a const reference to `_elems` after permuting it such that its k greatest indices contain its k least elements - essentially the same as the partition problem from Sharkland with the added requirement that the elements must be in non-ascending (reverse-sorted) order.⁸⁰

Further,

- this new method must have a run-time complexity that is at worst $O(k \log N)$, and
- it must work in-place, within the heap's data array (using $O(1)$ space).

To make your task easy, you are allowed to destroy your data array's heapiness in the process. You may mark the heap as empty before returning from this method.



That time when Hook pulled the pecking order before scuttling the ship

⁸⁰ Sibeï Wan, Fall 2020, pointed this out.

Strategy

Assuming k starts at 0 it's clear that finding the 0-least element would take constant time. That's $O(\log N)$.

So far so good.

A min-heap (your base class) already satisfies that condition. The challenge is to generalize this property so it works for not just the $k=0$ 'th element but for any k .

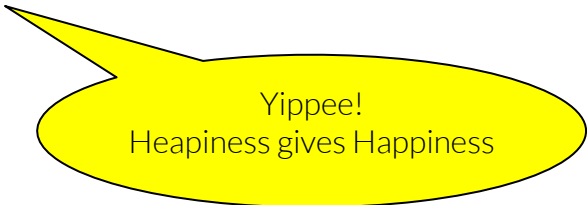
Since you're allowed to permute the internal data array when the call to this function arrives, you may be thinking that you could simply use the pivot-based partitioning strategy from your `shark` quest to return the least- k elements. But remember that its complexity is $O(N)$ on average.

Even if its worst-case quadratic complexity was deliberately ignored, it still wouldn't pass our test because we want the k -least elements in $O(k \log N)$ time, not $O(N)$.

The key insight here is to see that there is some ordering information already implicit in your heap that Quickselect fails to exploit. There may be several elements in the array that don't have to be examined at all.

Thus, by leveraging the heapiness of the array, you might be able to extract its k -least elements with a worst case of $O(k \log N)$. How? The presence of k in $O(k \log N)$ is a hint that you're allowed to do k $O(\log N)$ operations on the array.

Perform k `delete_min()` operations. Since each `delete_min()` takes at most $\log(N)$ time, you can be assured that your worst case behavior will be no worse than $O(k \log N)$.



Yippee!
Heapiness gives Happiness

All right. That solves the performance part of the problem. How about the space part? Can we make it happen in-place? (Ponder that before reading ahead).

Do it in-place

Welcome back (I hope).

Leverage another important insight: Every call to `delete_min()` is guaranteed to reduce the size of the heap by 1. Simply move each removed element into the just vacated spot, which is always the spot after the last occupied one.

If you perform k such `delete_min()`'s in a row, each time putting the deleted element into the newly created vacant spot, your k least elements will be in the k greatest indices of the array.

Now you can return a const reference to it.

get_least_k

If you have your min heap working properly this function should be a breeze to implement.

Simply get the min element, then delete it, then copy it into the just vacated cell of the heap which would have shrunk by one element upon the `delete_min()` operation. Repeat this step k times. At the end, mark the heap as empty (set size to 0), and return a const reference to your data array, `_elems`.

Suppose the heap is not large enough to complete the request. Then we have two options. Either we could throw an exception and complain, or you could simply return `_elems` as is, with no changes. Jack Morgan and I had [a conversation on this topic](#) in a previous quarter if you're interested. This method should respond to impossibly large requests by returning `_elems` as is.

One way to make sense of this behavior is to imagine that if you ask someone to do an impossible job, they refuse to touch it.

I don't think you need it, but just in case... Figure 3 shows a partial picture of the `Special_Heap` class definition.



```
// This is a sub-class of Heap that provides a single extra method called
// get_k_smallest(size_t k), which has a worst-case performance of O(k log N)
template <typename T>
class Special_Heap : public Heap<T> {
public:
    Special_Heap() : Heap<T>() {};
    Special_Heap(const vector<T> &vec) : Heap<T>(vec) {}

    const vector<T> &get_least_k(size_t k);

    friend class Tests; // Don't remove this line
};
```

Figure 3. The `Special_Heap` Class

Moar Discussion Points

Here are a few more discussion questions for the forum if the rest got picked off by your fellow questers before you got them:

- Why you would use Quickselect (or Quicksort) at all instead of simply using a special heap (or Heapsort)?
- Why not use a balanced binary search tree for your backing store? If you did, how does that affect the running times of the various public facing heap operations? What is the tradeoff, if any?



Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box
3. Drag and drop your source files⁸¹ into the button and press it

Wait for me to complete my tests and report back (usually a minute or less).

Don't worry. Be Heapy.

Heapy Questin',

&



⁸¹ Heap.h & Special_Heap.h

*this mousy mouse in her housie house has peony for her name
let peony show you how to be. you'll never be the same*



Just see. Don't kill yourself locating it. Best be. Just fill yourself placating it.

Graph Algos

In this quest you get to implement a simple `Graph` class and a bunch of cool algorithms on it. You will define the graph class in `Graph.h` and `.cpp`, and implement your `Graph` algorithms in a separate friend class called `Gx` in files called `Graph_Algorithms.h` and `.cpp`.

The `Graph` class is one of the simplest classes of RED. Those of you who met *Simplee* can simply use her code.

The others can refer to Figure 1 and hack up your own copy quickly. This is going to be the basis of all your mqs in this quest.



```
#include <vector>
#include <climits>
#include <cmath> // FLT_MAX

// The graph is represented as a vector of nodes in which each node has
// a vector of edges. Each edge is represented by a target node and the
// weight of that edge.
class Graph {
public:
    struct Edge {
        int dst;
        float wt;

        Edge(int tgt = -1, float w = 0) : dst(tgt), wt(w) {}
        bool operator==(const Edge &that) const { return dst == that.dst && wt == that.wt; }
        bool operator!=(const Edge &that) const { return dst != that.dst || wt != that.wt; }
    };

protected:
    static double constexpr FLOOR = 1e-6;
    std::vector<std::vector<Edge> > > _nodes;

public:
    size_t get_num_nodes() const { return _nodes.size(); }
    bool is_empty() const { return _nodes.empty(); }
    void clear() { _nodes.clear(); }
    std::string to_string() const;


    Graph &add_edge(int src, int tgt, float wt, bool replace=true);
    float find_edge_weight(int src, int tgt) const;

    friend class Gx;
    friend class Tests; // Don't remove this line.
};
```

Figure 1. The Graph Class

`Graph` is not a template class. Every node in this graph is identified by a unique non-negative integer.

Store it as a vector of a vector of `Edges`, where each edge is a structure containing a destination node number and a floating point weight.



Important: The graph is NOT a 2D matrix. It is a vector of nodes, where each node contains a vector of edges. Nodes that don't have edges leaving them will be represented by the cells that hold empty vectors. Put another way, the edge from node A to node B can be found in the vector located at `_nodes[A]`. But that edge is NOT `_nodes[A][B]`. Instead you must scan this inner vector to find the edge with the target of interest.


Some silent design decisions

- Node numbering starts at 0
- Self-loops are disallowed (and method behavior unspecified for graphs with self loops)
- Interpret edge weight as distance in distance problems, and capacity in flow problems.

Before getting started on the real MQs, first make sure your green level MQs are still functional. You even get rewards for it. You have to complete the implementation of the `Graph` class by supplying the following missing implementations in your `Graph.cpp`.

`add_edge(int src, int tgt, float wt, bool replace=true)`

After ensuring that your `_nodes` vector is large enough to index into either given node, insert an `Edge` object in the vector of edges leaving the source (`src`) pointing to the given destination (`tgt`) with the given weight (`wt`). Further helpful things about my reference implementation:

- 
- I'm not keeping the edge lists sorted
 - Every incoming `Edge` is compared to every existing one until a match is found or none can be found.
 - If a match to an existing `Edge` with the same destination is found, the boolean variable `replace` tells what to do. If it is true, the given weight replaces the existing weight. If it is false then the given weight is added to the existing weight.
 - If no match can be found, then a new `Edge` object with the given destination and weight is created and it is inserted at the *end* of the vector of edges leaving the given source.

`float find_edge_weight(int src, int tgt)`⁸²

Here is the cool thing about this method which should bring back memories of water birds.⁸³ If it's asked for the weight of an edge that doesn't exist, it returns `Graph::FLOOR`. This hints at the hidden assumption in any of our algorithms - A Graph is essentially *completely connected*. But unlike stilts which believe things they can't see, mice don't assume that invisible edges exist.

Graph methods should scan the vector of edges for the given source to find if requested edges are explicitly present with non-FLOOR weights even for simple connectivity tests.

⁸² If you get to choose the name of this method, would it start with `get_` or `find_`? Why?

⁸³ Why?

to_string

It's optional, and for your own benefit (like DDA said [here](#)). If you don't wanna do it, simply return the empty string. No complaints.

If you do wanna, I'm afraid all I have is an example. See if you can reverse-engineer the specs from Figure 2.

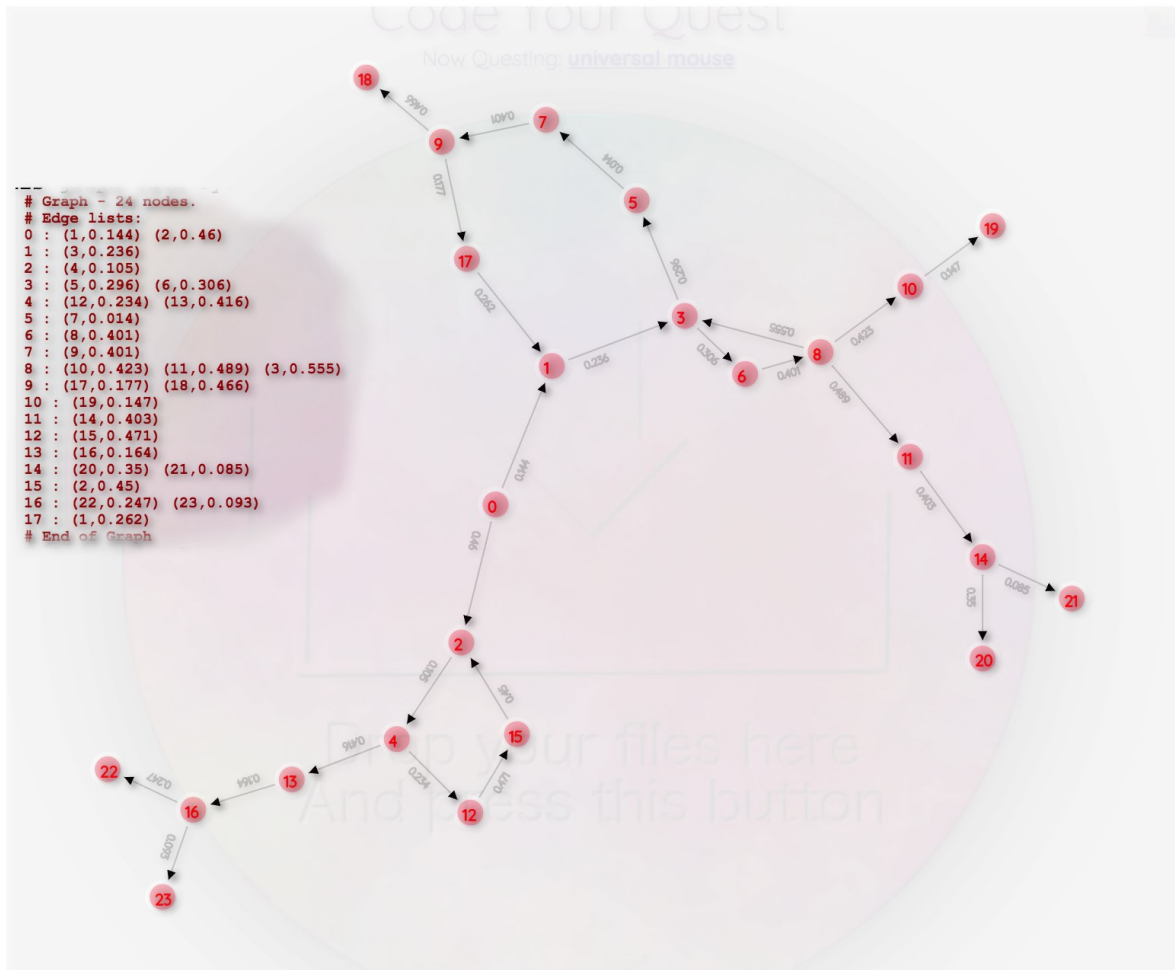
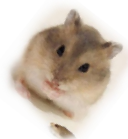


Figure 2. Example Graph::to_string() output



Now comes the interesting part. You get to implement all the cool algorithms in `Gx`. And some of them are worth a LOT of loot See Figure 3 for a picture of the `Gx` class.

```

class Gx {
private:
    struct NW { // package three co-used qtys
        int node, pred;
        float weight;
        NW(int n, int m, float wt) : node(n), pred(m), weight(wt) {}
        bool operator<(const NW &that) const { return this->weight > that.weight; }
        bool operator>(const NW &that) const { return this->weight < that.weight; }
    };

    static float _get_capacity_of_this_path(const Graph &g, const std::vector<int> &path);
    static float _get_max_capacity_path(const Graph &g, int src, int dst, std::vector<int> &path);
    static bool _is_cyclic(const Graph &g, size_t node, std::vector<bool> seen, std::vector<bool> & cycle_free);

public:
    static bool is_cyclic(const Graph &g);
    static bool prune_unreachables(Graph &g, int src);
    static size_t get_shortest_unweighted_path(const Graph &g, int src, int dst, std::vector<int> &path);
    static size_t get_shortest_weighted_path(const Graph &g, int src, int dst, std::vector<int> &path);

    static float get_max_flow(const Graph &g, int src, int dst);

    friend class Tests; // Don't remove
};

```

Figure 3. The Gx Class. Hmm... What does it take to get rid of the unsightly inversion of the comp ops?

Here are a few important things to know about the graphs you'll get:

In general, you can assume there are no self-loops in the graphs. You can safely ignore testing for them except where absolutely critical to make double-sure at low cost.

NW

This is like `std::pair`, except that it has 3 things in it that frequently ride together.

You'll find that you need to provide the comparison operators if you ever end up wanting to stick NWs in a heap. So it comes with two handy ones.

This is an optional (suggested) inner class. You may decide to package your variables differently, or not at all.

is_cyclic

If the given graph has one or more cycles, return true. Else return false.

There is no guarantee that the graph has only one entry point. That is, you cannot assume that node 0 is the only source without a parent.

Here is how I check for cyclicity. You don't have to do it this way: I iterate over the nodes and invoke a recursive helper method on each to tell if a graph rooted at the node is cyclic. The moment I find the first cycle, I return true all the way back immediately. I don't waste time trying to find all the cycles.



My private recursive helper _is_cyclic

I start a depth first descent from the given node. Every node I encounter must be a brand new node. If I hit a node I have already seen, I immediately return true.

To keep track of seen nodes I need a collection that I can look up quickly. A hash table comes to mind. But if you're thinking that it's too heavy for something as simple as what we need, you be thinkin like me.

Instead of a hash table, I exploit the fact that my nodes are indexed by integers. This lets me store a set of N node markers as a vector of N bools.

In my recursive helper, I have two set-of-node-markers parameters. One parameter, called `seen`, keeps track of the seen nodes in a particular recursive descent. The other, called `cycle_free`, keeps track of nodes whose acyclicity has been established, so they don't have to be checked again.

`cycle_free` is passed by reference, so it contributes negligible space overhead to the recursive call. But `seen` is passed in by copy (why?).⁸⁴ This means that each new level in a recursive call will create a fresh copy of the vector.⁸⁵

So suppose we do it that way, what's the worst that could happen? Always a good question, methinks.

The worst is if you end up with a degenerate graph which is simply one long linked list of many nodes. Then you'll end up recursing as deeply as the graph is *long*.

Just how bad is that? Suppose your input graph, rooted at 0, has 1K nodes, and that your graph is degenerate (a linked list of 1K nodes).

Then when you start your recursion at the root, you could end up recursing all the way to the leaf, 1000 nodes down, and thus 1000 levels deep. The call at each level would have its own copy of the `seen` vector. Since the graph has 1K nodes, each vector would take up 1K bytes (assuming 1 byte per bool). And therefore the total overhead due to this technique on the worst degenerate graph of 1000 nodes is at least $1K \times 1K = 1MB$.⁸⁶

To put this whole thing in perspective, you can assume that your process will get enough non-swap heap to handle graphs with degenerate filaments up to 15K nodes long (extremely unlikely).



prune_unreachables

Mark all nodes in the graph that are reachable from the given source node. Then sweep through the graph clearing out the edges of all OTHER nodes.

It's important that you don't delete the nodes themselves. Save yourself the hassle of renumbering the graph by leaving behind these edgeless nodes. Disconnected nodes can exist in a valid graph.

⁸⁴ Keven Yeh (2023) observed that it's possible to optimize further by passing `seen` by reference. How?

⁸⁵ YAWN (Yet another wacky nuance): The actual size of the vector passed by copy is not a serious consideration for running out of stack space in recursion. The vectors themselves will live on the heap, only their references will be on the program stack. How wacky cool is that?

⁸⁶ You can easily generalize this argument to show that the worst degenerate graph of N nodes will suffer a recursive space overhead of $O(N^2)$. Go for it.

One minor deviation from the normal behavior of this method is that it will simply return false when passed a non-existent node.

The alternative is to clear the edges of all the nodes in the graph.

Since that's pretty drastic, we'll simply provide a better named method called `clear()` and let this method fail with a more humble false return in this situation.⁸⁷



get_shortest_unweighted_path⁸⁸

Essentially, this is just a breadth-first search starting at your source node, ending when you find the destination. The found path must be written into the parameter `path` vector passed by reference, and the length of the path must be returned.⁸⁹ The first element of `path` should be the source, and its last should be the destination.

If there is no path from the source to the destination, you must return `string::npos`. If I request a path from a node to itself, you should return a path of length 1 (intuitively, it begins and ends at the source node).

Note that there may be multiple shortest unweighted paths. But only one of them is the lucky winner. No use sharing unlucky paths in the forums. 'Cuz they, like, change all of the time.

Better to figure out what is unlucky and stay away from it.

get_shortest_weighted_path

Once you have the unweighted path finder debugged, you'll find it easy to adapt it to find the shortest weighted path using Dijkstra's algorithm.

⁸⁷ The three levels of Questing Humility:

Mouse level: `(return false)` "I'm sorry, m'boss. I canna do it."

Duck level: `(throw exception)` "Are you outta your quackin mind? You stupid quack! If you **try** that again I'll extinctify you! Dumb quackin dodos..."

Fangs level: `(delete *; format *; end_world();)` "There! You sorry-ass m\$!%#. That'll teach you."

⁸⁸ If you get to choose the name of this method, would it start with `get_` or `find_`? Why? (Don't assume choices made in this quest are the best ones).

⁸⁹ Does it have to be BFS?

There are a few things to note here. If you work it out on paper, you'll see that this algorithm is really not all that different from its unweighted cousin. So think of it as enhancing what you already have, to account for edge weights.

The biggest difference is that instead of pushing and popping in a queue, you will push and pop in a min heap. Use the STL priority queue for this. Not the one you built in Baltimore.

Using a min heap guarantees that you process each of your neighbors in increasing order of their distance from you.

Why is it important to only process the nearest node each time? Intuitively, you can say that the nearest node is the ONLY node you can know for sure to not have a shorter path to it. How can you be sure that there is no shorter path from the source to your nearest node's nearest neighbor than the one through it (your nearest node)? Discuss.



Unsettling Settling

When you *settle* a node, you update the distances of all its neighbors. Some of these neighbors may already be in the min heap and need to be updated. How? As you probably know from Butterfly, updating particular elements in a heap is a right pain.

I know at least two less-than-ideal ways to handle this: One way is to hack the heap to let you hot-tweak element priorities.

The other way is to insert a new heap element with the now shorter distance. The old one will still be in the heap, but since it's a min heap, it won't be seen until after the new one has been seen. I think this is slightly better for our situation.

We'll go with this latter approach in this quest. But note that since your heap may collect a lot of junk as a consequence (why?), you need to have a way to remember nodes you've *settled* so you can silently skip them when they pop off the heap.

get_max_flow

This is like the Grand Ganondorf of Red Questopia. Get this one right and you're pretty much done.

Conceptually, the max flow problem is even easier to solve than the shortest path problem. But only once you've solved the shortest path problem. It needs to return a single floating point number equal to the maximum calculated flow from src to dst.

I found that many programmers who find maxflow challenging at first get confused at a particular point: They assume that an edge that has been used up is gone, and not part of the graph any more. Thus they fear that they may end up with a graph in which some flow cannot be extracted because lots of such "taken" edges are now missing, but critically needed - Yes?

Here is a different perspective of the problem that you won't find elsewhere. I tried this on a quester (in 2017) who said it helped:

Think of it this way: The moment some of the flow has been taken up in an edge, it immediately becomes capable of supporting exactly that much flow in the reverse direction. How? Just diminishing the forward flow is equivalent to increasing the reverse flow. See Figure 4.

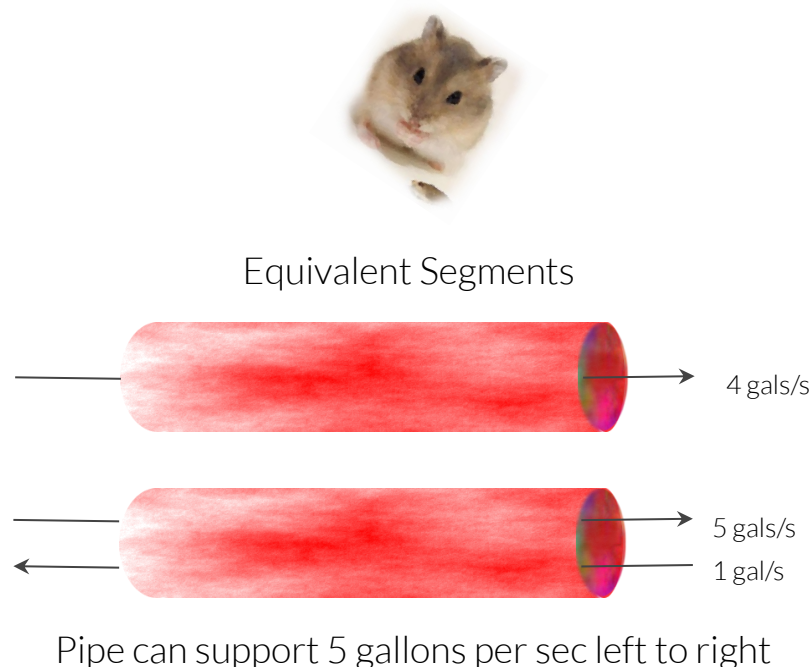


Figure 4. Another look at maxflow

So the trick to handling depleted edges (edges whose forward capacities you have used up) is not to remove them, but to add equal and opposite edges because those edges are still capable of conducting flow. Just not in the same direction.

I hope that helps you too.

As you implement `max_flow`, you may find you need the following two private helpers.

Private helper `_get_capacity_of_this_path`

Interpreting the edge weights of the input graph as flow capacities, return the capacity supported by the given path.

Yeah, something about a chain and its weakest link comes to mind here.

Private helper `_get_max_capacity_path`

This is almost identical to your `get_shortest_weighted_path` method, except for the fact that your heap compares in the opposite way.

Consider it a juicy opportunity to copy and refactor the code so that you need to rewrite as little as possible.⁹⁰



Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files⁹¹ into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

And then...

It's all entirely up to you now.

If you peeps were cormorants, I guess I might say *go forth and multiply*. I wish you the very best.

&

⁹⁰ Here's an engineering question: How do you re-signaturize your `get_shortest_weighted_path` method to be able to select between a minheap or a maxheap at run time? (Don't try it for this quest). Is it worth it?

⁹¹ `Graph.h, cpp` & `Graph_Algorithms.h, cpp`



Thank you for letting me serve you. With a pal like c++, ain't nobody no more to unnerve you⁹²

Well, dat be all folks

in yer questopic adventure with rare and wondrous creatures

i bet you loved c++, and all her arcane features

⁹² Weee!!!... I ate module zero for me brunch. These triple negs - they be me lunch.



With any luck, you're now ready to take on the best of 'em.

Are you a genius?

Ever wondered if there was a programmer in you?

Attend an intellectual retreat for pros — no prior knowledge of computer science required.

Friendly guides will walk with you until you're ready to start running with the other geniuses.

This book is packed with exciting quests for you to sink your teeth into.

Are you ready?

To solve the quests in a supportive group environment, enroll in the **Questopic Genius Bootcamp**:

quests.nonlinearmedia.org/genius

See questers in action:

reddit.com/r/cs2a

reddit.com/r/cs2b

reddit.com/r/cs2c

This edition can be downloaded for free at the official Genius site