CS I

By: Dr. Meade, Dr. Geist, Dr. Szumlanski

April 9, 2024

-3 Computer Science and C

These notes will cover a wide array of fundamental concepts in computer science, with an emphasis on the concept of order approximation, recursion, and complex data structures. We'll explore how various programs accomplishing the same task can be compared based on their memory and runtime usage.

The concepts will be presented through the lens of the C programming language, and here are several compelling reasons for utilizing C:

1. **Low-Level Nature:** C is a low-level language, lacking many built-in data structures. By constructing intricate data structures in C, you gain insights into how these structures might be implemented in languages that lack similar features.
2. **System Level Programming:** C is extensively employed in system-level programming, making the skills acquired in this language highly applicable across various classes.
3. **Precursor to C++:** C serves as the precursor to C++, a language widely used in object-oriented projects and throughout the industry. Mastery of C lays a solid foundation for delving into C++.
4. **Control Over Memory Management:** C affords a high degree of control over memory management. This aspect fosters a profound understanding of tasks typically handled by features like garbage collectors.

-3.1 Undefined Behavior: The Down Side.

Perhaps the biggest annoyance for students in computer science is dealing with Undefined Behavior (`UB`). Unlike Java, programs in C can do different things on different computers if implemented recklessly. `UB` can easily make a program seemingly pass all cases on your computer yet fail every case on the Teacher's/TA's. Most `UB` stems from using dangling pointers (discussed in great detail in the section on Dynamic Memory), indexing out of bounds, and using uninitialized memory.

-2 Compiler Installation

DO NOT USE REPLIT. Replit does not have the ability to let you use private repositories for free, and every semester at least one student has their code copied from Replit. Both the student that copies and the one that used Replit to write their code violate the course policy and are given a score as if they had cheated.

This section will help you install the proper tools to succeed in the CS1 homework programming assignments. While it is possible to write code for the assignments in any C compiler, doing so may cause you to lose points. C is considered a portable language; however, there are still differences between compilers and even compiler versions; if something works for you but does not compile for grading, that is still a zero just as if you submitted incorrect code.

It is highly recommend you learn how to install one of these on your own computer. However, if you are opposed to installing new software you can learn how to connect to the Eustis server, which already has a valid compiler installed.

CS1 uses GNU C (often referred to as `gcc`), a widely used open source compiler. The two other major C compilers in use today are `clang` (another open source compiler used on macOS) and Visual Studio (referred to here as VS, a proprietary compiler by Microsoft used on Windows). `clang` is meant to be `gcc` compatible; VS is not. VS considers many of the standard functions we will use in class to be deprecated for security reasons[1] and will clutter your output with warnings. More importantly, VS is more liberal with header file dependencies; there are many cases where your code will compile in VS but will in `gcc` due to header file differences.

Note that, confusingly, Visual Studio is a different Microsoft product from Visual Studio Code. The former is a combination development environment and a suite of compilers for developing applications for Windows. The latter is a lightweight code editor that works well with any compiler. As such, while Visual Studio is explicitly disallowed for homeworks, Visual Studio Code is encouraged.

You may write your code on any of the three major operating systems: Linux, Windows, or macOS; but you must set up `gcc` on your OS and use it.

-2.1 Compiler Installation

Software is typically installed via a system called a package manager. These sections will guide you through installing `gcc` on your platform. For all platforms, there are steps to be done in a terminal window; i.e. at the command line. Becoming familiar with the command line is a useful skill, especially for software development; later sections will discuss it in more depth.

-2.1.1 Linux

You can install `gcc` on Linux using the package manager for your distribution. It may even be preinstalled. From a terminal window, first check to see if it is by running `gcc`:

```
uname@hostname:~$ gcc
gcc: fatal error: no input files
compilation terminated.
```

If you see this output, then `gcc` is already installed. If not, and you see something like this:

```
username@hostname:~$ gcc

Command 'gcc' not found, did you mean:
...
```

[1] For good reason, in many cases, but the replacement functions Microsoft recommends are not all available on other platforms and as such we will not teach them in class.

then `gcc` is not installed. If you are running Ubuntu, you can use `apt` to install `gcc`. This operation must be done as the root user. Enter the command `sudo apt install gcc`:

```
username@hostname:~$ sudo apt install gcc
[sudo] password for username:
Reading package lists... Done
Building dependency tree
Reading state information... Done
...
```

At the password prompt, enter your login password. You will see `gcc` and some required additional packages being installed. When you get back to the prompt, try running `gcc` again as shown above to make sure it installed properly. Linux

-2.1.2 macOS

The development environment Apple provides for macOS is called Xcode. Xcode is a full IDE (integrated development environment) which means it contains an editor, a debugger, and tools for developing full applications with UI as well as the compiler. Xcode is very large; however, it is possible to install just the command line compiler. Open a terminal window (`Command+Space`) and try to run `gcc`. If the tools are not installed, a dialog will pop up asking to install them (Figure 1). Click to install and follow the prompts.

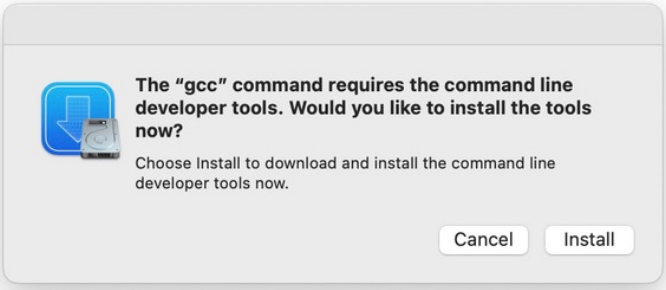


Figure 1: `gcc` install dialog on macOS

As previously mentioned, this will install `clang`, not `gcc`; however, as `clang` is a drop-in replacement for `gcc` it should be equivalent for the sake of the homeworks.

-2.1.3 Windows

There are two ways of getting `gcc` on Windows: running Linux in a virtual machine, or running a package manager directly on Windows. The former is recommended as all grading is done on Linux, and modern Windows versions make it easy to do. If you are running a modern version of Windows 10 or Windows 11, you can use WSL (Windows Subsystem for Linux). Open a PowerShell command prompt (Windows key to open the start menu, search for `powershell`) and enter:

```
PS C:\Users\username> wsl --install
Installing: Ubuntu GNU/Linux
Ubuntu GNU/Linux has been installed.
Launching Ubuntu GNU/Linux...
PS C:\Users\username>
```

Now, you can start a Linux command prompt by just running wsl:

```
PS C:\Users\username> wsl
username@hostname:/mnt/c/Users/username$
```

You can now follow the installation instructions for `gcc`.

If you want to go down the other path, using a native package manager, you can research cygwin and MinGW. Both of these collections include `gcc` compiled for Windows, but will require more steps to install than WSL.

Some students may have used CLion in a previous class. By default, CLion comes packaged with MinGW and thus uses `gcc`. If you choose to use CLion, you will not need to install a compiler manually; however, you have less direct control over how the compiler is run.

-2.2 Editor

There are many editors and IDE's that you can use to write your code. For the sake of consistency across platforms and ease of use, we recommend Visual Studio Code (usually abbreviated as VSCode). VSCode is free and can be installed from the app store on your platform (Microsoft Store on Windows, App Store on macOS, or the Ubuntu Software application on Ubuntu. Other Linux distros may have a similar portal, or a way to install VSCode directly via the package manager).

VSCode relies on extensions to add features for particular languages. For C code, we recommend Microsoft's own C/C++ tools. They will give syntax help and hilighting which makes your code easier to read while you are working on it. The first time you create a C file, VSCode will prompt you to install this extension; just clock to allow it to install.

-2.2.1 Starting VSCode

VSCode can be started either via the user interface like any other application, or from the command line. On Linux and Mac, either will work; on Windows, starting from the command line is easier because of WSL. Remember, we are running `gcc` in Linux under Windows, not on Windows itself, so VSCode needs to be connected to Linux. This is easy to do; just start a PowerShell prompt and enter `wsl code .` (That is not a typo, there is a literal period in the command to instruct VSCode to start in the current directory). This will start VSCode under a Linux session, and the following instructions on editing and compiling will all work without modification. See Figure 2.

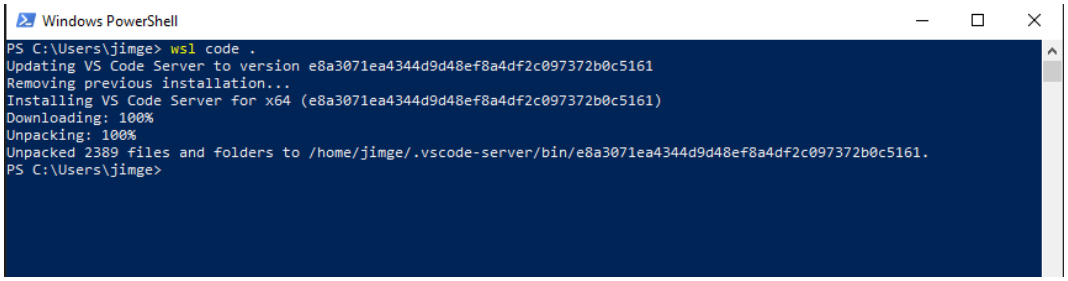


Figure 2: Starting VSCode on Windows

On starting VSCode from WSL, you may see the warning shown in Figure 3. It is safe to click `Don't Show Again` on this dialog. VSCode is suggesting storing files in the Linux environment because that is faster; however, we want to store them in Windows because it's easier to upload them to WebCourses from there.[2]

Figure 4 shows the canonical *Hello, World!* program in VSCode. The main parts of the app to note are:

[2]This does not mean Windows file access is slower than Linux. It is slower to access Windows files from inside WSL than to access files which are stored directly inside WSL. For the purposes of the assignments, the speed difference is not noticable.

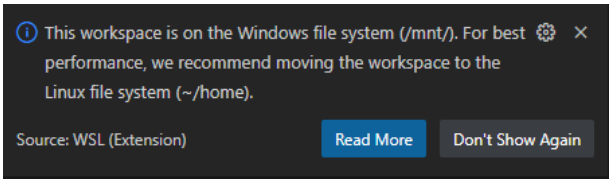


Figure 3: VSCode Linux file system warning

- The left icon bar shows the current mode. The top icon, the editor, is selected by default. Other modes allow more advanced searching, debugging, extensions, etc. These are all outside the scope of this document; we will concentrate on editing and running code.
- The left sidebar shows a list of open editors and everything in the current directory. In this figure, VSCode was started from a directory called `CS1` and contains just the source code and compiled version of the *Hello, World!* program. The sidebar can be hidden with `Ctrl+B`; typing `Ctrl+B` again will re-display it..
- The largest area is the editor itself, which contains the source code being edited.
- Below the editor is a terminal window where can compile and run our code. This window can be hidden and shown with `Ctrl+J`.

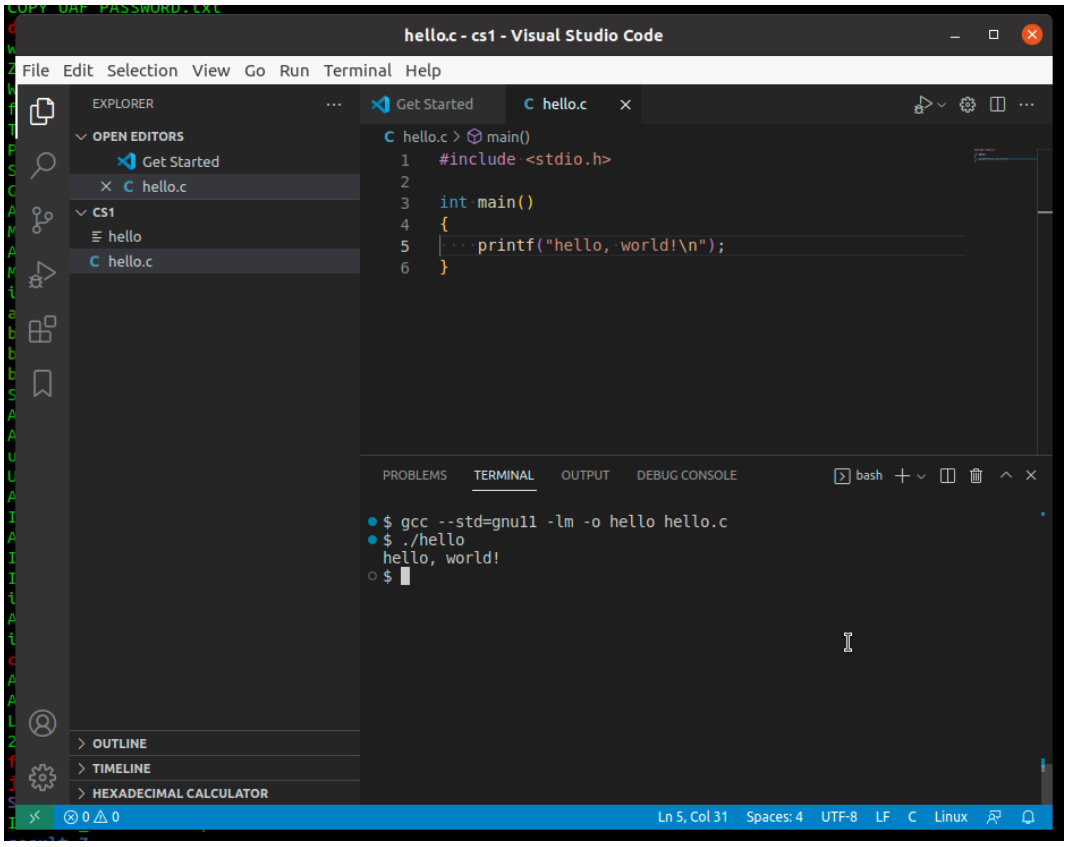


Figure 4: Hello, World! in VSCode

-2.2.2 Running a Program

The following steps will create and run the *Hello, World!* program. We recommend working through them to ensure that your environment is properly set up.

1. Create the source file by right clicking in the blank area of the file system sidebar, click on `New File...`, and name the file `hello.c`.
2. The file will be automatically opened in the editor. Type in the source code and hit `Ctrl+S` to save it.
3. If there isn't a terminal open, select `New Terminal` from the `Terminal menu`.
4. In the terminal, enter the `gcc` command as shown in the figure to compile the program. Note that the first argument is `gnu` followed by eleven, not two letter `l`'s. This will compile `hello.c` into an executable program called `hello`.
5. Run the program by entering the command `./hello`. The leading dot-slash is needed to run the program from the current directory.

The command used to compile the program is important, because this is exactly how your programs will be compiled when they are graded. The `-std=gnu11` argument tells the compiler what version of the C standard to use. Without this, your program may compile, but minor differences between the default standard for your compiler version may cause it to not compile for the TA's, so it is important to be explicit. `-lm` tells the compiler to include support for math functions like *sin*, *cos*, and *tan*; these will be needed in some assignments. Finally, `-o hello` is used to tell the compiler what to name your executable program.

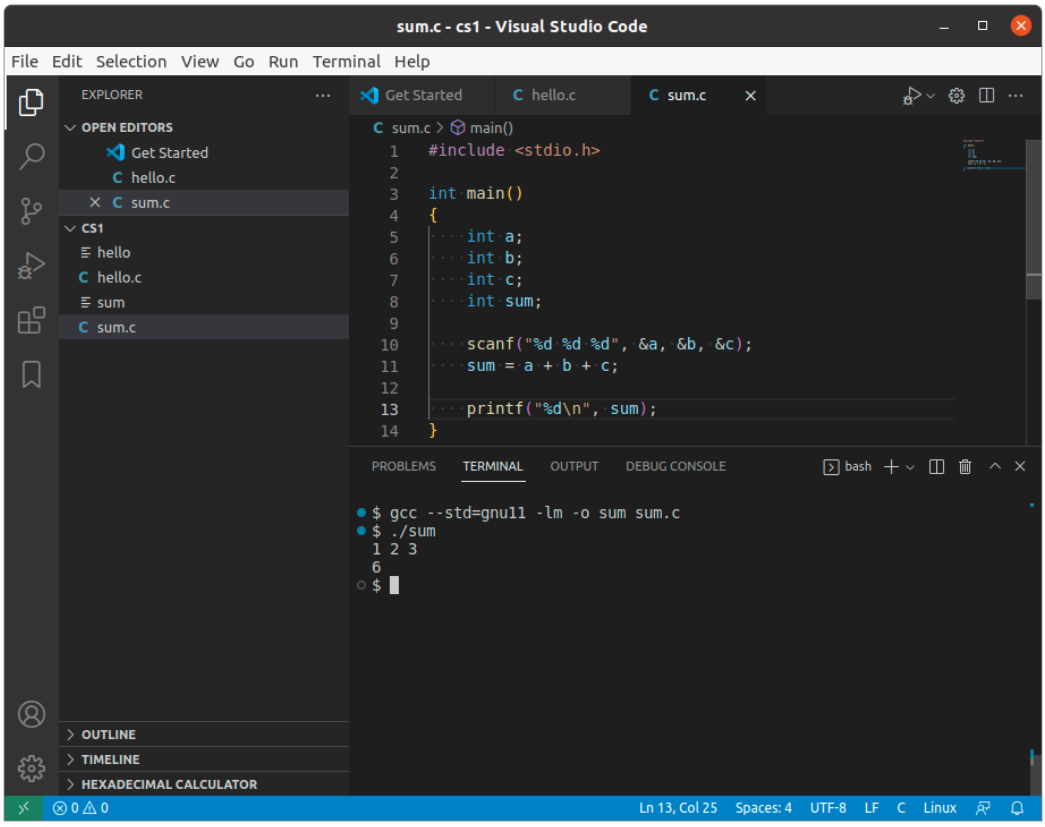


Figure 5: Reading Input from the Keyboard

-2.2.3 Reading Input

Most homework assignments will require you to read and process input. The assignments will give an exact expected input and output format; we will review how to do input and output in class. The assignments specify that your program should read its input from

`stdin`, which means that by default, the program will take input from the keyboard in the terminal window. However, this can become tedious when you are writing your code and running your program over and over. Figure 5 shows a program that adds three numbers. In the terminal window, when the program is run, it stops and waits for input. When the user enters the three numbers as shown, it will continue and print the sum.

The input to this example is simple; real homework input can be multiple lines of multiple strings or numbers. Thankfully, there is a better way. Figure 6 shows the same program, but taking input from a file.

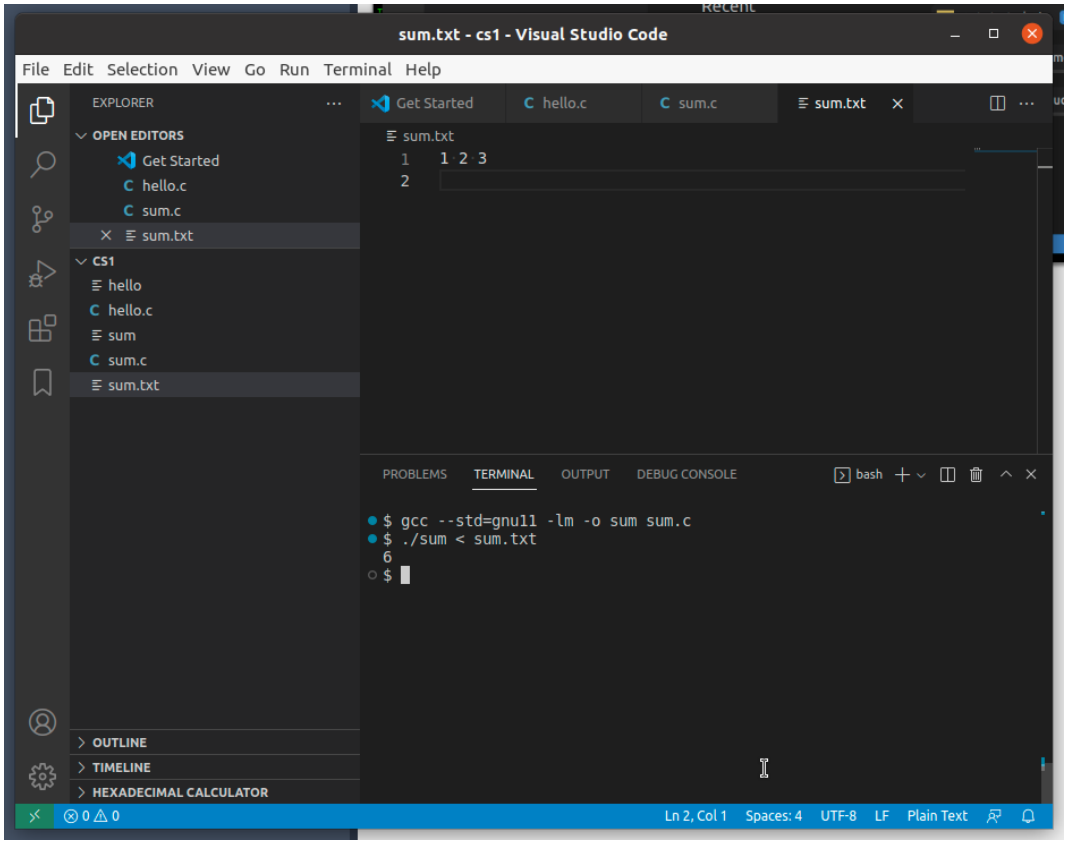


Figure 6: Reading Input from a file

In this screenshot, there is a second file with the program input called `sum.txt`. VSCode can create and edit any kind of text file, it isn't limited to source code. Note the slight difference when the program is being run: the command is now `./sum < sum.txt`. The less-than sign tells the terminal to *redirect* input[3]; instead of taking input from the keyboard, the program will now read input from `sum.txt`. Using this feature can save a lot of time when trying to get your assignment to work.

-2.2.4 Crashing

There will be many assignments that deal with pointers and dynamic memory. (If you don't know these topics well from *Intro to C*, don't worry; we will review them at the start of the semester.) When these features are used incorrectly, it is common for the program to crash; i.e. exit unexpectedly. Figure 7 shows a crashing program. Here, the program is intentionally performing an operation that is a common cause of crashes: a NULL pointer assignment. Note the text `Segmentation fault` after the output; this means the program has crashed. However, the program wrote the correct answer first.

[3]Really, it tells the command shell to redirect input. This syntax works on any modern shell, including `cmd` and `PowerShell` on Windows.

If your program does this, it will be graded incorrect. It is important to be aware of crashing as it is unacceptable program behavior. Even if the answer is correct, if the program does not exit cleanly, it is wrong.

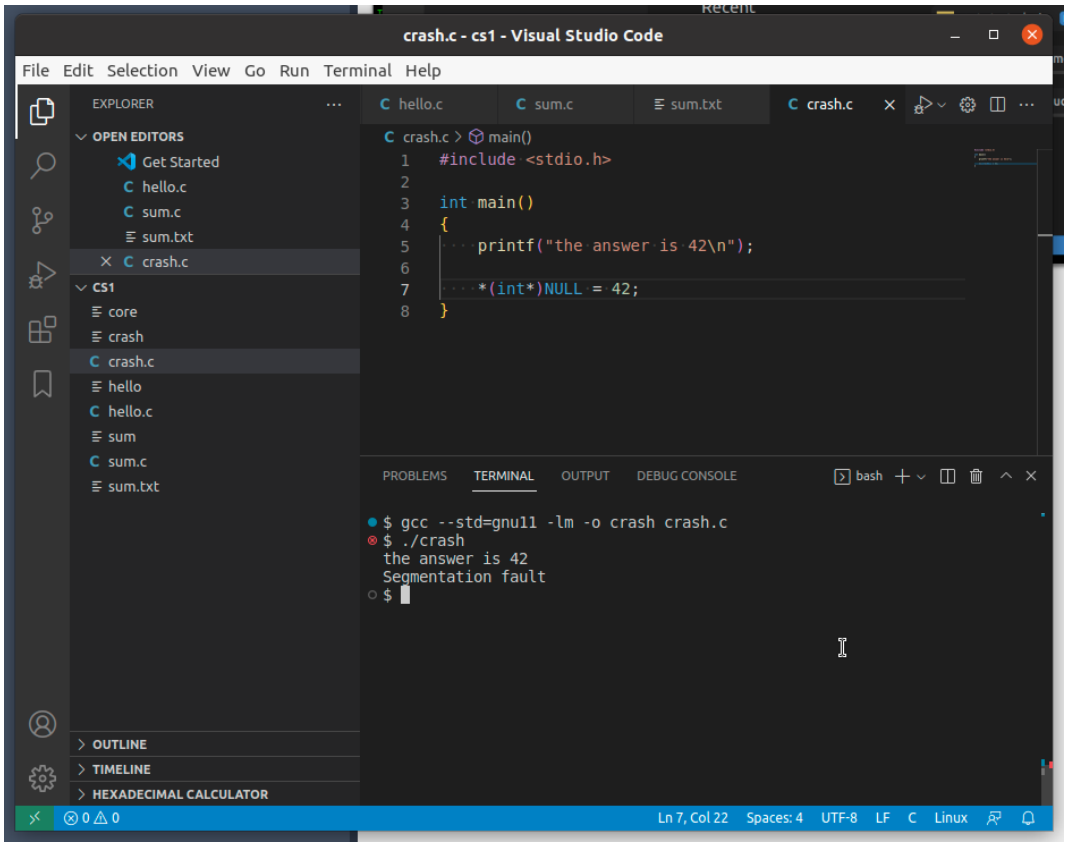


Figure 7: Crashing

-2.3 Tips for Homework Success

There are some things you can do to maximize your chances of performing well on the homework assignments. This list is compiled from common mistakes seen in previous semesters.

- Read the problem carefully. Your task in the assignments is to take a description of a problem and translate it into code; if you don't fully understand the problem, you will not be able to efficiently translate it into working code.
- Create more test data. Each assignment will come with several examples of inputs and correct outputs; however, the examples will not always cover every condition that can arise in the problem. Think about edge cases and create test data to cause them. The problem statement will also give information on how large the inputs can be; create test data that size. (Sometimes it's useful to write another program to do this). You will be given a time limit for how long your program will be allowed to run; this will help you understand if your program will run within that time limit.[4]
- Check for new test cases from others, as well. Sometimes people will share them on the class Discord channel, and sometimes the teaching staff will put new cases up on WebCourses.

[4]It's a good bet that if the assignment says there will be a maximum of 10 million inputs, at least one of the grading test cases will be that large.

- Pay attention to crashes. If you are crashing on exit and don't know why, try removing calls to free memory if you've added them; this is a common cause. Unless an assignment specifically says to free memory, you won't be counted off for not doing it. You should get in the habit for real world programming, but don't sacrifice points to it.
- If your program does not compile, you will get a zero. If your program compiles but does not work, you will get some points for things like good variable names and formatting. These are easy points; make sure your program at least compiles if you're up against the deadline.
- Pay attention to the output format. Assignments are graded in two parts: looking at the code for things like style and to see if the basics of the solution are present, and by running the assignment against a set of test cases. The latter is done by another program; output which is correct but improperly formatted will be marked wrong. There is no room for interpretation. (i.e. if the expected answer is "42", print "42", not "The answer is 42"). Don't print any other extraneous output such as prompts for input. If you added `printf` calls to debug your program, make sure they are removed or commented out before submitting.
- Download your assignment and make sure you have uploaded the correct version. This seems obvious, but every semester a few people lose points to it.
- Go to office hours. Many time, the TA's will give mini-lectures walking through how to solve the current homework. They won't write your code for you, but understanding the intended approach to solve a problem can help a lot.

I also recommend getting familiar with common Linux commands.

-1 Eustis

DO NOT USE REPLIT. Replit does not have the ability to let you use private repositories, and every semester at least one student has their code copied from Replit. Both the student that copies and the one that used Replit to write their code violate the course policy and are given a score as if they had cheated.

If you don't have the ability to install your own C compiler, then you should look to using Eustis. Eustis is a Linux server that UCF's CS department hosts that has a number of useful development tools. You can sign into Eustis using NID and NID password.

I won't reinvent the wheel here. Please use Dr. Szumlanski's Eustis user manual on webcourses or in the appendix section on Eustis to get Eustis working on your machine. Although Dr. Szumlanski has a list of useful Linux commands in the Eustis section, I have included an extended list on useful Linux commands in the appendix.

0 Background Knowledge

It is expected that you have all passed a prerequiste C programming class and have an understanding of concepts such as functions, structs, arrays, pointers, and dynamic memory. This section will provide a helpful refresher on the material covered in your C programming class. If you did not cover one or more of these topics or if you are not comfortable or unfamiliar with one or more of these topics, it is suggested that you spend the first week working on understanding the concept in detail. Each of these will be used frequently throughout the semester with little explanation on the fundamentals as to how the C language works.

0.1 Functions

Functions in C are used to prevent writing the same (or similar) lines of code that will be used in multiple places in the program. A classic example that will be looked into a lot in the Computer Science I course is sorting values. We may need to sort a set of values multiple times or in different ways, and if we created a single sorting segment of code it could reduce the amount of debugging needed if an error was created in the segment. We can also use functions to print values in a particular format, or perform some complex mathematics.

To make a function in C you (typically above the main function) list the return type of the function, the name of the function, the arguments (in parenthesis) to be given to the function the types of the arguments need to be specified in the, and then a block of code in curly brackets to be performed when the function is called. The code in the main it technically in the main function, and the main function should return the int type, which denotes to the system whether the function ran successfully (0 return code) or unsuccessfully (non-zero return code).

Below is an example of a function that swaps 2 integers using their addresses,

```c
void swap(int * a, int * b)
{
   int tmp = *a;  // Move a's value into a tmp variable
   *a = *b;       // Overwrite a with the value in b
   *b = tmp;      // Overwrite b with the value in the tmp variable
}
```

The return type is `void`, which means that no value will be returned. All non-`void` functions should return a value of the appropriate type before reaching the end of the function. Failure to do so results in `UB`. Below is an example of a non-`void` function that takes the absolute value of some given value.

```c
int abs(int value) // Function that returns an integer, the absolute value of value
{
   if (value < 0) // Check if the value is negative
   {
      return -value; // Return the negation of the value
   }
   return value; // Return the original value, because it was not negative
}
```

You can use (i.e. "call") the function by using the name of the functions followed by the required parameters in parenthesis. If we wanted to use that swap function in a program to swap the values of `x` and `y` we could do the following,

```c
void swap(int * a, int * b)
{
   int tmp = *a;  // Move a's value into a tmp variable
   *a = *b;       // Overwrite a with the value in b
   *b = tmp;      // Overwrite b with the value in the tmp variable
}

int main()
{
   int x = 5; // Create an integer x with value of 5
   int y = 7; // Create an integer y with value of 7

   swap(&x, &y); // Swap the values of x and y

   printf("x is %d\ty is %d\n", x, y); // Print the values stored in x and y

   return 0; // Exit program
}
```

Since the swap takes the address of integers and not values, we must pass in the address of the integers `x` and `y` to the function by using the ampersand (`&`). In order to change a value of a variable that is contained outside a function, the address of the variable is passed to the function. The reason for passing the address is that passing solely the value modifies only a copy of the value in the function and not the variable that is outside the function. Since arrays behave like pointers, changing an array in a function will change the array outside the function.

Below is an example of a function that does not actually swap the values in the main function,

```c
// THIS FUNCTION DOES NOT SWAP THE VALUES OUTSIDE THE FUNCTION
void swap(int a, int b)
{
   int tmp = a;  // Move a's value into a tmp variable
   a = b;       // Overwrite a with the value in b
   b = tmp;      // Overwrite b with the value in the tmp variable
}

int main()
{
   int x = 5; // Create an integer x with value of 5
   int y = 7; // Create an integer y with value of 7

   swap(x, y); // [DOES NOT] Swap the values of x and y

   printf("x is %d\ty is %d\n", x, y); // Print the values stored in x and y

   return 0; // Exit program
}
```

IMPORTANT: Do NOT specify the type when calling the function. In the `main` function we did not use the line of code "`swap(int x, int y);`"

In this class we will use a few times the concept of prototyping to help organize our code. When you prototype a function to tell your compiler that you will have a specifically named function, that has a certain return type, and certain types of arguments. However, when you prototype the function you do not need to specify the exact function only the type, name, and argument types. Instead of a block of code after the arguments a semicolon should be used instead. Below is an example of a program where the `swap` function has been prototyped and defined after its use in the `main` function,

```c
// Prototype the swap function here
void swap(int *, int *);

// The main function
int main()
{
   // Make an array of integers from 0 to 9 inclusive
   int arr[10];
   for (int i = 0; i < 10; i++)
   {
      arr[i] = i;
   }

   // Slide the value in spot 0 to the end
   for (int i = 0; i < 9; i++)
   {
      swap(&(arr[i]), &(arr[i+1]));
   }

   // Print the array
   for (int i = 0; i < 10; i++)
   {
      printf("%d\n", arr[i]);
   }

   return 0;
}

// The swap function
void swap(int * a, int * b)
{
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Without prototyping all your functions need to be explicitly defined before you call them in your code. When prototyping is used, you can define the functionality of your functions wherever you want in your code, which can be used to make your code more readable.

0.2 Structs

A `struct` in C gives programmers a way to create custom data type. It is a way to package information together. If you wanted to hold onto an array of points in 2 dimensional space without the use of a `struct`, then you could use 2 arrays—one of x-coordinates and one of y-coordinates. The technique of using multiple arrays where some index spot in some array corresponds to the same spot in a different array is known as parallel arrays. Parallel arrays can be difficult to manage and understand. When you want to change a point, you would need to consider modification of the x and the y array simultaneously, which means that if a function was in use, both arrays would need to be passed.

The alternative to parallel arrays is to use a `struct` to hold onto multiple pieces of information simultaneously.

0.2.1 `struct` Declaration

When making a `struct`, first the `struct` is denoted using the `struct` keyword. Next the name of the `struct` is specified. Following this is usually a block in curly braces of the types and name of each part (i.e. member) of the `struct`.

Below is an example of a `struct` that could store a 2 dimensional point,

```c
struct Point // The defintion of a struct type called Point
{
   int x; // An integer member called x
   int y; // An integer member called y
};
```

To instantiate a `Point struct`, the word `Point` would not be sufficient in the above example. With custom `struct` and no additional aliasing, to make an instance of the `struct` the fact that the data type is a `struct` needs to be specified. Specifying that the variable is a custom data structure is done by using the phrase `struct Point`.

Below is an example of instantiating a `Point` structure in a main function,

```c
struct Point // The defintion of a struct type called Point
{
   int x; // An integer member called x
   int y; // An integer member called y
};

int main() // The main function
{
   struct Point p; // Create an instance of the Point struct called "p"

   return 0; // Exit program
}
```

To change the data (i.e. members) of the `struct` we use the dot operators (`.`). The left hand side of the operator should be the name of the instance of the `struct` (e.g. `p`) and the right hand side of the operator should be the name of the member (e.g. `x`). Below is an example of making a point a location (3, 4),

```c
struct Point // Struct definition
{
   int x, y; // Both members defined in the same line
};

int main() // main function
{
   struct Point p; // Instantiation

   p.x = 3; // Initialize the x-coordinate
   p.y = 4; // Initialize the y-coordinate

   return 0; // Exit program
}
```

To make life easier C programmers can leverage something called `typedef` to alias the name of their `struct` type to something sorter. To use `typedef` start with `typedef`, then the name of the data type, followed by a one word alias, terminated with a semicolon. The scope of the `typedef` is important. If the `typedef` is used inside a block of code, then outside the block the alias no long works. A data type can have multiple aliases, (e.g. "`int`" could be called "`i32`" and "`number`" at the same time).

Below is an example of using aliasing via `typedef` on the 2 dimensional point,

```c
typedef struct Point Point; // Alias struct Point to Point

struct Point // Struct definition
{
   int x, y; // Both members defined in the same line
};

int main() // main function
{
   Point p; // Instantiation

   p.x = 3; // Initialize the x-coordinate
   p.y = 4; // Initialize the y-coordinate

   return 0; // Exit program
}
```

Some cringe programmers use the `typedef` at the same time as the `struct` definition. The simultaneous defintion and alias is done by starting the `typedef`, naming the `struct`, without giving the alias given the definition of the `struct`, and finally end with the alias and semicolon. It is legal, but disgusting. Below is an example of the terrible practice,

```c
typedef struct Point // Struct definition
{
   int x, y; // Both members defined in the same line
} Point; // Alias ends here.

int main() // main function
{
   Point p; // Instantiation

   p.x = 3; // Initialize the x-coordinate
   p.y = 4; // Initialize the y-coordinate

   return 0; // Exit program
}
```

0.2.2 `struct` Argument

Custom data structures can be used as arguments to functions. Note that you do need to specify which from which instance of the data structure you wish to access the member. A lot of programmers that are used to object oriented languages such as Java, make the mistake of assuming that the member will be used for some implicit instance of the custom structure. The language of C does not have methods. Without specifying the instance of the `struct` the compiler has no clue as to which instance of the `struct` should be used. On the next page is an example of some good and **BAD** usage of custom data structures as an argument to a function,

```c
typedef struct Point Point; // Alias
struct Point// Struct definition
{
   int x, y; // x and y member
};

void printPoint(Point p) // Type is okay be cause of Alias
{
   printf("(%d,", p.x); // Print the x coordinate of p just fine
   printf(" %d)\n", y); // DOES NOT COMPILE!!! The variable y does not exist
}
```

Below is a fixed version of the above code,

```c
typedef struct Point Point; // Alias
struct Point// Struct definition
{
   int x, y; // x and y member
};

void printPoint(Point p) // Type is okay be cause of Alias
{
   printf("(%d,", p.x); // Print the x coordinate of p just fine
   printf(" %d)\n", p.y); // Print the y coordinate of p just fine
}
```

The way the above functions have the custom data structures passed in is by value. In other words modifications to the `struct` in those functions would not reflect the changes in variables of the caller function. The `struct` can be passed by address using pointers, which would allow for the `struct` to be modified in called functions. Below is an example of a function that can and a function that cannot modify a `struct`,

```c
typedef struct Point Point; // Alias
struct Point// Struct definition
{
   int x, y; // x and y member
};

void modify(Point * p) // This function can modify point outside the function
{
   (*p).x += 1; // Derefernce Point p so the member x can be incremented
   (*p).y = 0; // Derefernce Point p so the member y can be assigned to 0
}

void noModify(Point p) // This function cannot modify the point outside the function
{
   p.x += 1; // Change the copy of the original point's x member
   p.y = 0; // Change the copy of the original point's y member
}
```

There is some syntactic sugar that can make using address of `struct` easier. The arrow operator (`->`) can be used to dereference the address of a `struct` and access the member in a more concise manner. The proper notation is that the left hand side of the operator is the address of (i.e. pointer to) the `struct` and the right hand side is the name of the member (e.g. "`x`". Below is an example of the arrow operator used with the `Point struct` used in earlier examples,

```c
typedef struct Point Point; // Alias
struct Point// Struct definition
{
   int x, y; // x and y member
};

void modify(Point * p) // This function can modify point outside the function
{
   p->x += 1; // Using arrow operator to dereference and access the x member of p
   p->y = 0; // Using arrow operator to dereference and access the y member of p
}
```

The arrow operator should only be used on the address of a structure, and not on a structure or the address of the address of a structure.

0.2.3 `struct` Return Type

Custom data structures (or an address to one using sec:DMA]Dynamic Memory) can also be the return value of a function. Just like how a function can have the return type specified, that return type can be a `struct`. Below is an example of a function that both uses a `struct` as arguments and uses a `struct` as a return type.

```c
typedef struct Point Point; // Alias
struct Point// Struct definition
{
   int x, y; // x and y member
};

Point add(Point lhs, Point rsh) // This function adds 2 points together
{
   Point result; // Instantiate the result

   // Store the sum of the x's from the 2 given points together in the result's x
   result.x = lhs.x + rhs.x;


   // Store the sum of the y's from the 2 given points together in the result's y
   result.y = lhs.y + rhs.y;

   return result; // Return the resulting Point
}
```

The result of the above function can be assigned into a Point that has already been instantiated. However, the address to memory of result should not be returned. If you need the address of a new point you should use Dynamic Memory.

0.3 Arrays

An array is a way of making a consecutive group of variables all of the same type. Arrays should be considered when the number of instances of a particular data type is quite large. Arrays are indexed by integer values, which means that loops can be used on arrays to reduce the number of operations.

0.3.1 Instantiation

To create a (static) array in C, the code is first the data type of the array, the name of the array, an open bracket ("`[`"), the number of instances of the needed data type, a closing bracket ("`]`"), and finally a semicolon. On the next page is an example of creating an array of 10 integers,

```
int array[10]; // Instantiate an array of 10 values
// The values in the array are not initialized
return 0; // Exit program
```

The values in that array are garbage. Using the values in the array would result in `UB`. In some systems the array might be all 0's, but in many systems the array will be filled with seemingly random numbers. Below is a way to ensure that the array is filled with all 0's,

```
int array[10] = {0}; // Instantiate an array of 10 values.
// The 0th value is going to be zero; all other values will be set to 0
return 0; // Exit program
```

You can also create an array of a variable size using an integer. Below is an example of code that creates a static array of a size depending on a input value,

```
int n;
scanf("%d", &n); // Read in a value n from the user
int array[n]; // Instantiate an array of n values
```

YOU HAVE TO READ IN THE VALUE `n` BEFORE INSTANTIATION OF `array`. When creating an array using a variable as the size, you cannot initialize the values using the same method for fixed sized arrays. In order to insure that arrays using a variable as the size have the correct initial values, an manual initialization—with a loop—is required. The following program that tries to initialize an array with variable size **DOES NOT COMPILE**,

```
int n;
scanf("%d", &n); // Read in a value n from the user
int array[n] = {0}; // Instantiate an array of n value.
// Tries and FAILS to initialize the array
```

Below is a proper way to make an array of `n` 0's,

```
int n;
scanf("%d", &n); // Read in a value n from the user
int array[n]; // Instantiate an array of n values
for (int i = 0; i < n; i++) // Loop through all the values
{
   arr[i] = 0; // Fill with 0s
}
```

You also cannot assign arrays to other arrays to copy values. The following is an example of an array initialization that **WILL NOT WORK**.

```
int array[10]; // Instantiate an array of 10 values
for (int i = 0; i < 10; i++) // Loop through all the values
{
   arr[i] = 0; // Fill with 0s
}
int array_two[10]; // Instantiate a second array of 10 values
array_two = arr; // Try (FAILS) to initialize the second array using the first
```

0.3.2 Array Arguments

Arrays are like pointers in that they can be represented as an address to the first spot of memory in a consecutive block. Due to an array's ability to behave like pointers arrays can be passed as pointers into functions. The ability to pass an address for an array also saves a lot in terms of both time—by not having to copy the array's values—and memory—by not having to store the values of the array twice. The down side to this is that array modifications in a

function will modify the array in the original function. **You cannot determine the length of an array passed as an argument to a function.** I highly recommend that you pass the length of the array into any function that is also taking in the array. Below is an example of how you can make a function that takes in an array as an argument,

```
// The array argument can be specified using open close brackets
void makeRandomArray(int array[], int length) // Function to populate an array
{                                             // with random values
   for (int i = 0; i < length; i++)
   {
      array[i] = rand() % 100; // Make the value random value in the
   }                           // range of 0 to 99 inclusive
}

// The array argument can be specified using the a pointer
int sumArray(int * array, int length) // Function to sum the values of the array
{
   int res = 0; // Instantiate and initialize a reult to 0
   for (int i = 0; i < length; i++) // Loop through the values of the array
   {
      res += array[i]; // Add the value of the array to the result
   }
   return res; // Return the sum of all the values
}

int main() // The main function
{
   int arr[10]; // Instantiate an array of 10 values
   makeRandomArray(arr, 10); // (Purposefully) Fill the array with random numbers
   int sum = sumArray(arr, 10); // Get the sum of the array
   printf("The array sum was %d.\n", sum); // Print the result
   return 0; // Exit the program
}
```

Arrays passed by arguments "can" be assigned to another array in the same function, but that assignment only changes what the array argument is pointing to not the actual values. You should read the following code and try to determine what will be printed,

```
void whatDoIDo(int array[], int length) {
   int array2[length];
   for (int i = 0; i < length; i++) {
      array2[i] = i;
   }
   array = array2;
}
int main() {
   int len;
   int arr[len = 10];
   whatDoIDo(arr, len);
   for (int i = 0; i < len; i++) {
      printf("%d\n", arr[i]);
   }
   return 0;
}
```

After you trace the code by hand, try running the code. Remember to include the Standard Input Output header file.

0.4 Pointers

Pointers are addresses to sections of memory that can be used to modify variables instantiated in functions that are not your own. Additionally, passing in the address of a large block of memory is faster than copying all the data of the block into a function. The pointer to a block of memory can be extracted using the ampersand (&). Using acquiring the address of a block of memory is sometimes referred to as acquiring a reference or referencing. To convert an address (reference) to a block of memory back into the data, a dereference operator—using the asterisk (*)— is used. Below is a swap example that was used earlier in functions,

```
void swap(int * a, int * b)
{
   int tmp = *a;  // Move a's value into a tmp variable
   *a = *b;       // Overwrite a with the value in b
   *b = tmp;      // Overwrite b with the value in the tmp variable
}

int main()
{
   int x = 5; // Create an integer x with value of 5
   int y = 7; // Create an integer y with value of 7

   swap(&x, &y); // Swap the values of x and y

   printf("x is %d\ty is %d\n", x, y); // Print the values stored in x and y

   return 0; // Exit program
}
```

As mentioned earlier, arrays are very much like pointers. An array can be treated as a pointer to a section of memory. You can chop of the beginning portion of an array and pass it into the function by using the address of a specific index into the array. Below is an example of passing in an array without the first 3 values into a function,

```
void fillWith(int array[], int length, int value) // function that fills an array
{                                                  // with a given value
   for (int i = 0; i < length; i++) // Loop through all the values in the array
   {
      array[i] = value; // Assign the value
   }
}

int main() // The main function
{
   int len = 10;
   int arr[len];
   fillWith(arr, len, 0); // Fill the full array
   fillWith(&arr[3], len - 3, 1); // Fill the value after the first 3

   for (int i = 0; i < len; i++) // Loop through the array
   {
      printf("%d\n", arr[i]); // Print all the values 0,0,0,1,1,...
   }

   return 0; // Exit the program
}
```

You can also use pointer arithmetic to acquire a pointer to a different location in an array. In the following code segment the 2 function calls perform the same role as the function calls in the one above,

```
  fillWith(arr, len, 0); // Fill the full array
  fillWith(arr + 3, len - 3, 1); // Fill the value after the first 3
```

You can also access values in an array using dereference operator the same way you dereference a pointer. Below is an example of how you can fill an array with the values 1 through `n` where

```
int main() // The main function
{
  int n;
  scanf("%d", &n); // Read in the length of the array

  int arr[n]; // Instantiate the array

  for (int i = 0; i < n; i++) // Loop through the array
  {
    *(arr + i) = i + 1; // Assign the i-th value of the array
  }

  return 0; // Exit the program
}
```

The bracket operator to access a value in an array is really syntactic sugar that is the same as an offset through pointer arithmetic followed by dereferencing. The following lines of code are all equivalent,

```
  arr[i] = 0;
  *(arr + i) = 0;
  *(i + arr) = 0;
  i[arr] = 0;
```

There is a special pointer that is never able to be acquired normally. The pointer is called `NULL`. Dereferencing the `NULL` pointer results in a segmentation fault without fail. The following program will crash without fail.

```
int main() // The main function
{
  int * ptr = NULL; // Point to NULL
  while (*ptr) // Crash the program
  {
    ptr++; // Will not be reached, but would move the pointer forward normally
  }
  return 0; // Exit program
}
```

0.5 Dynamic Memory Introduction

Dynamic memory—discussed more in detail later—allows for resizing memory, which is useful when not knowing how much memory will be needed at instantiation. Memory that is created static using—like the arrays created earlier—cannot be resized in the functions in which they are instantiated. Dynamic memory managed using a number of functions 4 of which we will discuss in more detail later are the following,

- `malloc` - Creates a section of dynamic memory in the heap
- `calloc` - Creates a section of dynamic memory in the heap and initialized to bytewise 0
- `realloc` - Changes the size of a section of dynamic memory that was in the heap
- `free` - Relinquishes a section of memory that was given by a dynamic memory back to the system

Hopefully you've seen all of these functions in your C programming class.

1 Strings

Strings are a sequence of values. Most of the time, they are thought of as a sequence of characters. For the purposes of implementation in C an array of characters is used, and expected by most libraries that take strings as input.

Since arrays are pointers in C. Passing a string to a string function does not require copying over all the memory, but instead it will pass in the pointer (the address). The address can be a lot smaller than an actual string, especially when dealing with genomes, which could contain billions of base pairs.

1.1 NULL terminator

Every string function needs to know how long the string is. In C an extra character (the NULL terminator) is appended to the end of a string to denote the end. Keep in mind that characters can be thought of as values in the range of 0 to 255 inclusive. The NULL terminator uses the value of 0. The NULL terminator can also be represented using the character literal `'\0'`.

Since the NULL terminator uses the value of 0, it is the only character that evaluates to false in C, which makes code shorter sometimes when checking for NULL terminators. This addition of the NULL terminator does mean that typically more memory than characters in the string are required when allocating the memory.

The string "Apple" which has 5 characters, must be stored in an array with at least 6 characters. Failure to do so can result in array out of bounds errors when trying to store "Apple" using built in string functions. My recommendation is that you always add 1 to the allocation for strings. Here is a picture of how the values in a string storing "Apple" would look.

Index	0	1	2	3	4	5	6
Value	`'A'`	`'p'`	`'p'`	`'l'`	`'e'`	`'\0'`	??

For example, if you need an array for string of up to length 20, then do the following,

```c
#define SIZE 20 // Hard coded max number of characters in the string
int main() // The main function
{
   char word[SIZE + 1]; // Instantiate the string
   // Read in the word and process it here...
   return 0; // Exit the program
}
```

This section will talk about string functions, ways to read in strings.

1.2 String Functions

There are several functions that are part of the C standard library. These functions can be included in your program by using the `<string.h>` header file. Since strings are arrays, and arrays are pointers, each string function deals with character pointers. Below are a list of some of the functions in `<string.h>` of which you should be aware.

- `strlen` (Important)
- `strcpy` (Important)
- `strcat` (Important)
- `strcmp` (Important)
- `strncmp` (Not on FE)
- `strdup` (Not on FE; Dynamic Memory)
- `strstr` (Not on FE)
- `strcascmp` (Not on FE)

1.2.1 `strlen`

Probably the most simple string function is the `strlen`. It takes as an argument the pointer to (the address of) the first character of the target string, and it returns the number of characters that occurs before the first NULL terminator. Usage of the `strlen` can be seen below,

```c
#include <string.h> // Include the string library

int main() // The main function
{
   int lenOfWord = strlen("Apple"); // Find the length of the word "Apple"
   // Note 5 will be stored in integer above

   return 0; // Exit the program
}
```

What is interesting about this is that the number of characters before the first NULL terminator is also the same as the index of the first NULL terminator. Below is an example of how `strlen` can be implemented.

```c
int myLen(char * str) // A function that finds the length of a string
{
   int index = 0; // Find the index of the first NULL terminator
   while (str[index] != '\0') // Loop until at NULL terminator
   {
      index++; // Increment to the next spot
   }

   return index; // Return the index of the first NULL terminator
}
```

Since any character in the array could be a NULL terminator, the `strlen` function needs to look at all possible characters up until the first NULL terminator. There is a common mistake beginning programmers make that causes some programs to be needlessly slow when processing strings that are not changing. The below example code is a program that is slow,

```
#define SIZE 1000000 // Hard code the max number of characters in a string

int main() // The main function
{
   char str[SIZE + 1]; // Make a string with A LOT of A/B's
   for (int i = 0; i < SIZE; i++)
      str[i] = 'A' + (rand()%2);
   str[SIZE] = '\0';

   int count = 0; // Count the number of 'A'
   for (int i = 0; i < strlen(str); i++)
      if (str[i] == 'A')
         count++;

   return 0; // Exit the program
}
```

Below is a faster version of the above program.

```
#define SIZE 1000000 // Hard code the max number of characters in a string

int main()
{
   char str[SIZE + 1]; // Make a string with A LOT of A/B's
   for (int i = 0; i < SIZE; i++)
      str[i] = 'A' + (rand()%2);
   str[SIZE] = '\0';

   int len = strlen(str); // Count the number of 'A'
   int count = 0;
   for (int i = 0; i < len; i++)
      if (str[i] == 'A')
         count++;

   return 0; // Exit the program
}
```

Rather than recomputing the string length everytime (which does not change). We compute it once and use the store value. Here is a third version that is arguably faster.

```
#define SIZE 1000000 // Hard code the max number of characters in a string

int main()
{
   char str[SIZE + 1]; // Make a string with A LOT of A/B's
   for (int i = 0; i < SIZE; i++)
      str[i] = 'A' + (rand()%2);
   str[SIZE] = '\0';

   int count = 0; // Count the number of 'A'
   for (int i = 0; str[i] != '\0'; i++)
      if (str[i] == 'A')
         count++;

   return 0; // Exit the program
}
```

1.2.2 strcpy

The next string function discussed in these notes is the string copy function. `strcpy` copies the contents of one string into another. One of the strings is overwritten with the new string. The function takes as input 2 string pointers. The function returns the pointer to the string written to.

Author Note: I almost never use the return value from this function

The input to the function is first the string being overwritten and second the string from which the characters come. Alternatively, you can think of it as the destination string before the source string. Mnemonically, I remember "`strcpy(a,b);`" as the string equivalent of "`a = b;`". Keep in mind that string assignment typically does not work the way you will probably intend in C. The string copy function is how string assignments should be performed. Below is an example of `strcpy`,

```c
#include <string.h> // Include the string header file

int main()
{
   char str[5 + 1]; // Store the word "Apple" into str
   strcpy(str, "Apple");

   return 0; // Exit the program
}
```

Author's Note: The src and dest should not overlap, otherwise UB. Beware of array out of bounds it also causes UB

Below is an example of how `strcpy` could be implemented.

```c
char * myCpy(char * dest, char * src)
{
   int index = 0; // Store the contents of the src string upto the NULL terminator
   while (src[index] != '\0') {
      dest[index] = src[index];
      index++;
   }

   dest[index] = '\0'; // NULL terminate the destination string

   return dest; // Return the location of the destination string
}
```

1.2.3 strcat

The cat in `strcat` is short for concatenate. The function like `strcpy` takes in the destination and source string pointers (in that order). The function also returns the resulting destination string. The contents of the second string are written to the end of the first string. Mnemonically, "`strcat(a, b);`" can be thought of as "`a += b;`". Again since strings are pointers. You can/should not use standard incremental operators (e.g. `+=`) on them. On the next page is an example of `strcat`,

```c
#include <string.h> // Include the string header file

int main() // The main fuction
{
   char filling[8 + 1]; // Store the word "Apple" into filling
   strcpy(filling, "Apple");

   char pastry[3 + 1]; // And store "Pie" into pastry
   strcpy(pastry, "Pie");

    strcat(filling, pastry); // Make an "ApplePie"

    return 0; // Exit the program
}
```

Author Notes: dest and src should not overlap (UB). Beware of array out of bounds (UB). Don't assume that uninitialized memory will be "empty strings" when using `strcat` (UB)

Below is an example of how `strcat` could be implemented,

```c
char * myCat(char * dest, char * src) // A custom concatination function
{
   char * newDest = dest; // Find the end of the destination string
   while (newDest[0] != '\0')
      newDest++; // Mild pointer abuse

   int index = 0; // Copy all the source to the end of the new dest
   while (src[index] != '\0')
   {
      newDest[index] = src[index];
      index++;
   }

   newDest[index] = '\0'; // Terminate the resulting string

   return dest; // Return the location of the destination string.
}
```

1.2.4 `strcmp`

The `strcmp` function is useful for finding the lexicographical ordering between 2 strings. Lexicographical is not the same as alphabetical. The value of the characters is used for sorting, which means that `'Z'` comes before `'a'`. The value returned from `strcmp` is guaranteed to be:

- 0 if the two strings are identical.
- A positive value if the first differing characters in the 2 strings is greater for the first argument.
- A negative value if the first differing characters in the 2 strings is greater for the second argument.

Do NOT check that the return is 1, 0, or -1, because it could be any positive value not just 1. Similarly the negative value might not be -1.

`strcmp` cannot directly be used to check if a string is a prefix of another string. A prefix of some original string is a string that is composed of some contiguous substring of characters starting at the beginning of the original string. A substring of some original string is a contiguous subsequence of characters that occurs in the original string. Mnemonically,

"`strcmp(a,b);`" is similar to "`a - b;`". Once again you should not try to actually subtract strings in C, instead use the `strcmp` method.

```c
#include <string.h> // Include the string header for the strcmp function

int main() // The main function
{
   strcmp("apple", "Banana"); // Some positive value (b/c 'a' > 'B')
   strcmp("apple", "Apple"); // Some positive value (b/c 'a' > 'A')
   strcmp("apple", "apple"); // 0 (b/c equal strings)
   strcmp("apple", "banana"); // Some negative value (b/c 'a' < 'b')
   strcmp("app", "apple"); // Some negative value (b/c '\0' < 'l')
   strcmp("apple", "app"); // Some positive value (b/c 'l' > '\0')

   return 0; // Exit the program
}
```

The function `strcmp` has no clue as to where the strings will be different, so the function needs to loop through all the characters comparing them. Below is a possible implementation of `strcmp`,

```c
int myCmp(char * first, char * second)
{
   int index = 0; // Find the first difference or the end
   while (first[index] == second[index] && first[index] != '\0')
      index++;

   if (first[index] == second[index]) // Check if the strings were equal
      return 0;

   return first[index] - second[index]; // Different strings; return the difference
}
```

Note in the function above the check if the strings were equal is unnecessary, since the difference of 2 equal values is 0.

1.2.5 `strncmp` (Not on FE)

The `strncmp` function is like string compare where only a limited number of characters is compared. The limited comparison can enable checking if a string is a prefix or check how long of a prefix 2 strings have in common. The arguments for the function are 2 strings and the number of characters to compare. The return value is similar to the return of `strcmp`, with the exception that no characters are checked beyond the specified limit. Below is an example of using `strncmp`.

```c
#include <string.h> // Include the strncmp function

int main() // The main function
{
   strncmp("app", "apple", 3); // 0 (b/c first 3 characters are ==)
   strncmp("app", "apple", 1); // 0 (b/c first character is ==)
   strncmp("app", "apple", 4); // negative value (b/c '\0' < 'l')
   strncmp("app", "apple", 20); // negative value (b/c '\0' < 'l')
   strncmp("apple", "apple", 20); // 0 (b/c strings are ==)

   return 0; // Exit the program
}
```

1.2.6 `strdup` (Not on FE)

When we get to dynamic memory `strdup` will make more sense. The `strdup` function is useful if,

- You want to reduce the memory used to store
- You need to create a copy of a string dynamically
- You did not know the size of the string at the time of creating the memory[5]

The function takes as an argument a character pointer, and returns a dynamically created copy (with a NULL terminator) that uses the exact size necessary. Strings created this way should be `free`'d, which means if you are using this function you need to also include `<stdlib.h>`. An example of `strdup` can be seen below,

```
#include <string.h> // Include the strdup function

int main() // The main function
{
   char * str = strdup("hello"); // Create a string called "hello"

   str[0] = 'H'; // Make the first letter capital H

   printf("%s\n", str); // Print the string

   free(str); // Clean up the dynamic memory allocated by strdup

   return 0; // Exit the program
}
```

Checking for a `NULL` pointer is a good idea when using `strdup`.

1.2.7 `strstr` (Not on FE)

The `strstr` function is useful to check if some string (needle) is contained in another (haystack). The 2 strings are passed as arguments: haystack first and needle second. The return value is the start of the first location that the needle is located in the haystack. If the needle is not contained in the haystack, `NULL` is returned instead. Below is an example of `strstr` being used.

```
#include <string.h> // Include the strstr function

int main() // The main function
{
   if (NULL != strstr("abacaba", "cab")) // Check if "cab" is contained in "abacaba"
      printf("cab is contained.\n");
   else
      printf("No cab is found.\n");

   return 0; // Exit the program
}
```

1.2.8 `strcasecmp` (Not on FE)

The strcasecmp is a useful function to compare strings, where you want words that differ by only case (uppercase/lowercase) to be treated as equal. If there is a difference that is not just by case, then the standard lexicographical comparison is used.

[5] Structs with strings would be a good example for this

1.3 Reading Strings

Reading strings can be intimidating. You could try to use some method to read a single character at a time to parse your string, but there are more functions that can help parsing depending on your needs.

1.3.1 scanf

In my opinion the simplest method to use is `scanf` using the "`%s`" format specifier. Using "`%s`" will read in a string that is delineated by any whitespace. This means that if you wanted to read "`Travis Meade`" into a string, you would need to use "`%s`" twice since there are two "strings" separated by a space.

The downside to this method is that it is hard to tell if the words are separated by a single space or a tab or newlines. It is for this reason that many programmers opt to use a method that will read a full line. Below is an example of reading strings using `scanf` and "`%s`".

```c
#include <string.h> // Include string functions
#include <stdio.h> // Include the io functions

int main() // The main function
{
   char filling[5 + 1]; // Instantiate the arrays that will hold the input.
   char pastry[3 + 1];
   char dish[5 + 1 + 3 + 1];

   scanf("%s%s", filling, pastry); // Read in "Apple" and "Pie"
   // The input strings should be separated by some whitespace when typed

   strcpy(dish, filling); // Create the dish ("Apple Pie") from the given input
   strcat(dish, " ");
   strcat(dish, pastry);

   printf("We made %s!\n", dish); // Print the resulting dish

   return 0; // Exit the program
}
```

1.4 gets

DEPRECATED From the description in the man pages **"Never use this function."** This function is the source of well known bugs in deployed code.

That being said the function will read into a string characters until a newline or End Of File (`EOF`) character is reached. The last character read into the buffer (newline/`EOF`) is replaced with a NULL terminator. The return value of the gets is the NULL pointer if the read fails (probably already at `EOF`) and the destination pointer if successful.

1.4.1 fgets

`fgets` more or less stands for file get string. It is not deprecated. It takes 3 arguments:

1. To where the string is being written (dest)
2. The size of the character array
3. The file point of the stream from where the word is coming (src)

The method reads a full line upto and including newline characters/`EOF`s. However, if reading that many characters would overfill the character array. Instead all but the last character will be filled with the contents from the specified file pointer, and the last character

would become the NULL terminator. Since `fgets` also writes into the string the newline character it is most of the time important to make the strings an additional character longer than necessary. Below is an example of the `fgets` function,

```c
#include <stdio.h> // Include the fgets function

int main() // The main function
{
   // We will read "Apple Pie" from the input using fgets
   // Make a string that is long enough
   // 5 for "Apple"
   // 1 for the space
   // 3 for "Pie"
   // 1 for the newline
   // 1 for the NULL terminator

   char line[5 + 1 + 3 + 1 + 1]; // Instantiate the array

   fgets(line, 5 + 1 + 3 + 1 + 1, stdin); // Read in the line

   printf("We read in \"%s\".\n", line); // Print the line read in

   return 0; // Exit the program
}
```

Author's Note: You should probably use MAX_LINE_SIZE + 2 when creating a character array to hold the information you will read in using `fgets`. It should hopefully be obvious that passing a larger value than the size of the buffer is `UB`. <u>DO NOT USE</u> `sizeof(str)` for determining how many characters are in the string.

1.5 String Practice

For practice try

- implementing your own verions of the other string functions such as strstr, strncpy, and strcasecmp
- reversing a string
- count the number of times a particular string appears in another as a substring
- replacing all instances of some substring with another
- flipping the capitalization of all letters
- breaks a string apart by spaces

2 Dynamic Memory Allocation

Suppose you are creating a roster of students that are part of a school. An array would be useful for storing the roster. The size of the array needs to be large enough to hold all the students.

How would we know the longest possible roster length? Additionally, if the longest roster length is in the hundreds of thousands, but only a few students are in the roster at the beginning there will be a lot of memory wasted.

For this reason C comes with the ability to change the size of an allocation using dynamic memory. Most variables you have created have been in a section of memory referred to as the stack. Even function calls put their information in the stack. The memory in the stack is created and removed in a fixed order, and to resize memory the stack cannot be used. The other section of memory (called the heap) is where dynamic memory is located. To access the heap special functions should be used.

In general Dynamic memory is good for

- Creating advanced data structures with complex topologies
- Expanding the size of an instantiation (unbounded collections)
- Cleaning up parts of memory no longer needed before leaving scope
- Returning memory created in functions

2.1 Built-in Functions

The functions that we will focus on for accessing the heap come from the `<stdlib.h>` header file. We will discuss 2 ways of "creating" a block of memory, a way to resize, and a way to recycle the memory allocated.

2.1.1 `malloc`

`malloc` is a simple function for getting a block of dynamic memory. It takes in a single argument: a `size_t` value representing the desired number of bytes in the resulting block of memory. To get the bytes of a particular data type[6] the `sizeof` unary operator should be used. The function returns the address to the block of memory "created". Since the block of memory could store any information the return type is actually a `void` pointer (`void *`). To convert the block to the desired data type, a cast should be used.

The resulting block of memory does not have the original values modified, so the initial values could be anything. It is a good idea to initialize whatever memory you created using `malloc` as quickly as possible. Below is an example of creating an array of integers using `malloc`,

```c
#include <stdlib.h> // Include the stdlib header file that contains malloc

int main() // The main function
{
   int * array = (int *) malloc(10 * sizeof(int)); // Create an array of 10 integers

   return 0; // Exit the program
} // Note: the memory held by array is leaked
```

The first `int *` instantiates the "`array`" pointer and represents the type of data that `array`—the variable—will contain. The `(int *)` after the equal sign is the cast that will convert the memory block returned from `malloc` into an `int` pointer. The argument in the

[6] `sizeof` can be used on variables, but `sizeof` returns the number of bytes used by the type that the variable represents. For example an int pointer, which you may intend to be an array, would return the size of the pointer and not the size of the bytes in the array itself. I recommend never using `sizeof` on variables.

`malloc` call (i.e. "`10 * sizeof(int)`" quotes for clarity) represents the number of bytes wanted in the returned memory block. The desire is to have enough bytes for 10 `int`s.

If `malloc` fails to allocate memory a `NULL` pointer is returned. Trying to access the value at address `NULL` (e.g. `NULL[0]`) will result in a segmentation fault. The below segment of code checks that an allocation succeeded before accessing the values,

```c
#include <stdlib.h> // Include the allocation functions
#include <stdio.h> // Include the Input Output functions

int main() // The main function
{
   int * array = (int *) malloc(10 * sizeof(int)); // Create an array of 10 integers

   if (array != NULL) // Check if malloc succeeded
   {
      for (int i = 0; i < 10; i++) // Since the array is valid fill it
      {
        array[i] = i;
      }
   }
   else
   {
      fprintf(stderr, "WARNING: Allocation failed.\n"); // Warn user of failure
   }

   return 0; // Exit the program
} // Note: the memory held by array is leaked if the allocation succeeded
```

In the above code we print the warning message to the standard error instead of the standard output. There are 2 standard streams of output in Unix systems. This was done so that when chaining commands problems encountered by a particular command could print the problems without feeding that warning as input into a command further down the chain. You can also use `fprintf` to print to standard output by using `fprintf(stdout, <format string>, <args>);`.

2.1.2 `calloc`

Like `malloc`, `calloc` will return a block of memory as a void pointer. Also like `malloc` if the allocation fails the NULL pointer is returned. `calloc` not only creates, but also initializes a block of data dynamically. The returned block of memory has each byte set to the 0-byte. For integers the values are all 0, and for character arrays, each character is the NULL terminator.

Also unlike `malloc`, `calloc` takes in 2 parameters: the number of items, and the size of each item. Requiring 2 parameters was useful when multiplication could overflow a `size_t`, but with modern systems `size_t` is typically large enough to fit very large products of values.

Below is an example that creates an array of 10 integers using `calloc`,

```c
#include <stdlib.h> // Include the calloc function

int main() // The main function
{
   int * array = (int *) calloc(10, sizeof(int)); // Create an array of 10 integers

   // Note: Assuming calloc is successful each number will be 0

   return 0; // Exit the program
   // Note: the memory held by array is leaked
}
```

2.1.3 realloc

realloc is a useful function when using dynamic memory in C. Dynamic memory can be resized into smaller or larger sizes by realloc. To resize a section of dynamic memory the address of the old section will need to be passed in as an argument. Additionally, realloc needs to know the new size of the block of memory. For such reasons the arguments for realloc are first the old pointer, and secondly the desired number of bytes for the resulting block of memory. Below is an example of realloc

```c
#include <stdlib.h> // Include the dynamic memory functions

int main()
{
    int * array = (int *) calloc(10, sizeof(int)); // Create an array of 10 integers

    array = (int *) realloc(array, 20 * sizeof(int)); // Make the array bigger

    return 0; // Exit the program
    // Note: the memory held by array is leaked (but only once)
}
```

If a NULL pointer is given as the first argument, then realloc behaves the same as malloc. There are three possible outcomes for realloc when given a non-NULL,

1. The address returned is the same as the address given as the first argument. In this situation, realloc succeeds and does not use significant time.
2. The address returned is non-NULL, but different from the address given as the first argument. In this situation, realloc succeeds but takes a significant amount of time to complete. The old address is free'd, and the new address given has been allocated.
3. The address returned is NULL. In this situation, realloc failed. The old memory is not free'd. Failing to free the old pointer of memory will result in a memory leak.

realloc can be "slow" if the new memory block is at a different location from the original. When the memory block is moved, the contents of the memory block is copied to the new location, which takes time directly related to the size of the memory block.

2.1.4 Free

Every time you create memory using malloc, calloc, or realloc your system needs to maintain that memory and prevent other programs or other parts of your program from using that memory. The memory protection can lead to issues when trying to allocate memory repeatedly. Allocating too much memory can cause your system to slow down or even deny giving out additional blocks of memory. To give back the memory to your computer you can use the free function that takes in a pointer to a block of memory that was given to your program through malloc, calloc, or realloc.

Freeing NULL is valid according to the C standard. The program will do nothing to the memory structure when freeing NULL. You don't need to avoid freeing NULL, but there is no reason to do so other than making code easier to write in some cases.

Recall that the realloc function will free the old pointer if the old address does not match the new address. On the next page is an example of memory being cleaned up by the free function.

```c
#include <stdlib.h> // Include allocation and free functions

int main() // The main function
{
   int * array = (int *) calloc(10, sizeof(int)); // Create an array of 10 integers

   array = (int *) realloc(array, 20 * sizeof(int)); // Make the array bigger

   free(array); // Try to free the created memory
   // Note this might not free the memory if realloc failed :(

   return 0; // Exit the program having given back all memory
}
```

2.2 Memory Errors

Using dynamic memory creates a large list of possible problems that will be discussed in this section.

2.2.1 Dereference `NULL`

The classic problem that you will encounter in C with respect to dynamic memory is dereferencing `NULL`. Dereferencing `NULL` causes the system to segmentation fault. Note segmentation fault is one of the better errors to induce when using dynamic memory; when some errors occur the program can silently fail, which can be frustrating when debugging. Below is a simple example of deferencing `NULL`,

```c
int main() // The main Function
{
   int * array = NULL; // An invalid array

   array[0] = 0; // seg fault here

   return 0; // Exit the program
}
```

Dereferencing NULL can occur when,

1. Indexing into an array that may be set to `NULL`
2. Using the arrow operator (->) of a pointer to a `struct` that was set to `NULL`
3. Trying to access the value of a pointer to some variable (e.g. pass by pointer to a function), where the pointer was `NULL`
4. Using unchecked pointer from a allocation function that failed

2.2.2 Memory Leaks

When dynamic memory that was allocated is not `free`'d prior to losing the reference the memory is "leaked". Your program will not be able to give back to the system the section of memory created. In older computers memory leaks could cause the operating system to fail, but most modern OS's can clean up whatever mess your program makes. However, it can cause the computer to behave poorly as it tries to clean up the memory.

On the next page is an example of a memory leak,

```c
#include <stdlib.h> // Include allocation functions

void leakMemory(int size) // Function that leaks memory
{
    int * array = (int *) calloc(size, sizeof(int)); // Allocate an array
    // On successful allocation the address lost after leaving the function's scope
}


int main() // The main function
{
   leakMemory(10); // Leak 10 integers worth of memory
   return 0; // Exit the program
}
```

Below is another example of a memory leak,

```c
#include <stdlib.h> // Include some allocation functions

int main() // The main function
{
   int * array = (int *) calloc(10, sizeof(int)); // create an array

   array = (int *) calloc(20, sizeof(int)); // Leak here

   free(array); // only frees the array created in the second calloc

   return 0; // Exit the program
}
```

2.2.3 Use After `free`

Use After Free is when memory which was initially protected by the computer using an allocation function, is dereferenced after the block was free'd. Use after free is `UB`; doing so can cause your program to have a segmentation fault, but more often than not the program will continue its execution, but the memory protections will become compromised, and variables can randomly change values. It is hard to detect and debug. Below is an example of use after free,

```c
#include <stdlib.h> // Include allocation functions

int main() // The main function
{
   int * array = (int *) malloc(10 * sizeof(int)); // Create an array of 10 integers

   free(array); // The array is free'd

   for (int i = 0; i < 10; i++) // Try to use the array
      array[i] = i; // Use after free here

   return 0; // Exit the program
}
```

A special type of use after free is called **Double Free**, where the same pointer to a block of memory is free'd twice, but allocated only once. Below is an example of a double free,

```c
#include <stdlib.h> // Include allocation functions

int main() // The main function
```

```c
{
   int * array = (int *) malloc(10 * sizeof(int)); // Create an array of 10 integers

   free(array); // The array is free'd

   free(array); // This is bad (UB)

   return 0; // Exit the program
}
```

2.2.4 Other UB

Dereferencing a **Dangling Pointer** is bad. A dangling pointer is a pointer to an unallocated block of memory, it could be a pointer that is uninitialized or a pointer that has had its memory free'd (i.e. Use After Free). Using a dangling pointer may lead to your program having its values changed unexpectedly or (if you're lucky) cause your program to crash.

Allocating a block of 0 bytes is also UB. DO NOT DO THIS.

2.2.5 NULL Pointer and Allocation Failure

It was mentioned earlier, but when any of the dynamic memory functions fails to obtain a protected block of memory a `NULL` pointer is returned. It is a good practice to check for `NULL` prior to using the dynamic memory your program receives from the system. However, `NULL` is actually a really good part of the C standard.

It is a good practice to assign your pointers to `NULL` when you allocate a pointer and after freeing dynamic memory. This can in many cases prevent the possibility of Use After Free or Double Free. It also cuts back on the number of dangling pointers still in your program. When you dereference `NULL` your program crashes, which will be noticed. Additionally, if you free one of these `NULL` pointers, nothing bad happens, since the C standard allows for it.

2.3 Dynamic Structs

Dynamic memory functions and `sizeof` can be used to create `struct`s dynamically. Since the values received from the dynamic memory function are pointers, rather than getting a `struct`, programs will receive a `struct` pointer. To access the members of `struct` pointers the `struct` must first be dereferenced this can be done with both the `*` and `.` operators. Since `struct` pointers tend to occur a lot in more advanced programs and data structures, a combined arrow operator (`->`) can be used instead as mentioned earlier. Below is an example of a `struct` created dynamically and used,

```c
#include <stdlib.h> // Include allocation functions

struct car // Struct definition
{
   int mileage, year; // Members of the struct
};

int main() // The main function
{
   struct car * myCar = (car *) malloc(sizeof(struct car)); // Create a car
   myCar->mileage = 113000; // Modify the mileage of the car
   (*myCar).year = 2023; // Modify the year of the car

   free(myCar); // Free the car
   return 0; // Exit the program
}
```

It should be noted that `typedef`s can be used to simplify the above code to become `sizeof(car)` instead of `sizeof(struct car)`.

Below is an example of creating and modifying an array of cars,

```c
#include <stdlib.h> // Include the dynamic memory function

typedef struct car car; // Alias

struct car // struct definition for a car
{
   int mileage, year; // The members of the car struct
};

int main() // The main function
{
   car * car_lot = (car *) malloc(3 * sizeof(car)); // Create 3 cars dynamically

   car_lot[0].mileage = 15000; // Modify the values of the first car
   car_lot[0].year = 2022;

   car_lot[1].mileage = 57000; // Modify the values of the second car
   car_lot[1].year = 2019;

   car_lot[2].mileage = 512; // Modify the values of the third car
   car_lot[2].year = 1985;

   free(car_lot); // Free the array

   return 0; // Exit the program
}
```

Note that in the above example the dot operator (`.`) was able to be used because although array was a pointer to a structure, the bracket operator (`[#]`) dereferenced the array and converted the pointer to the `struct`s into a normal `struct`.

You can also have `struct` with members that point to blocks of dynamically allocated memory. `struct` with pointer members will be used a lot in this class. On the next page is an example of a `struct` that holds 2 dynamically created sections of memory,

```c
#include <stdio.h> // Input Output header file
#include <stdlib.h> // Allocation functions
#include <string.h> // String functions

#define MAX_LEN 10000 // Max length of a name

typedef struct Person Person; // Alias for the person struct
struct Person // A person struct defintion
{
   char * first_name, * last_name; // Pointers to arrays of characters
}

int main() // The main function
{
   char tmp[MAX_LEN + 1]; // Instantiate the array for the names
   Person p; // Instantiate the person

   scanf("%s", tmp); // Read in the first name
   p.first = (char *) malloc(sizeof(char) * (strlen(tmp) + 1)); // Allocate first
   strcpy(p.first, tmp); // Copy the first name in

   scanf("%s", tmp); // Read in the last name
   p.lst = (char *) malloc(sizeof(char) * (strlen(tmp) + 1)); // Allocate last
   strcpy(p.last, tmp); // Copy the last name in

   free(p.first); // Clean up memory
   free(p.last);
   return 0; // Exit
}
```

2.4 Multidimensional Arrays

Dynamic memory functions can be used to create multidimensional arrays as well. A multidimensional array can be thought of as an array of arrays. Additionally arrays are similar to pointers. For these reasons a 2D array is a pointer to pointers. To create a multidimensional array the outermost dimension must be allocated first, but to avoid use after free when cleaning up memory the innermost dimension must be free'd first. Below is an example of dynamic multidimensional arrays, where each row of the array has a number of spots proportional to the index.

```c
#include <stdlib.h> // Include the allocation functions

int main() // The main function
{
   int ** grid = (int **) malloc(10 * sizeof(int *)); // Create the first dimension
   // Note each element of the grid is actually an int pointer

   for (int i = 0; i < 10; i++) // Create the "rows" of the grid dynamically
   {
      grid[i] = (int *) calloc(i + 1, sizeof(int));
   }

   // Use grid here

   for (int i = 0; i < 10; i++) // Free the ‘‘rows’’ of the grid
   {
      free(grid[i]);
   }

   free(grid); // Free the initial allocation

   return 0; // Exit the program
}
```

The allocation and the free must go in reverse order—allocate from first dimension to last and free from last dimension to first—to prevent a Use After Free when cleaning up memory. This jagged array (array with unequal allocations per index) would be very challenging to implement statically. The method of allocation demonstrated in the example above extends to higher dimensional arrays.

2.5 Dynamic Memory Practice

To practice your dynamic memory skills,

- Write a program that reads in an integer representing the number of names and proceeding this the names. Allocate your memory dynamically such that you only store as much memory as necessary to hold all the information. You can assume that the largest name has only 99 characters.
- Write a program that stores multiple menus, where each menu has a list of entrees that all have a price. Store all this information dynamically.

3 Array Lists

Consider the school problem of holding on to students in a roster. We would like to support the function of adding students, removing students, and accessing particular students record's. This all needs to be done with the expectation that the number of students is unbounded. That is, students can keep being added without a hard limit. To solve this problem we will leverage a data structure known as the `List`. A List should be an unbounded, ordered collection of values.

In most languages there are 2 mains ways to create a List:

1. **Array List** - a list backed using an array
2. **Linked List**

The most basic function a List can support is appending (adding to the end of the List). In this section efficient append to an Array List and iteration through an Array List will be demonstrated.

3.1 Array List Structure

To create an Array List in C a pointer (of a typed dependent on the desired function) will be needed. The pointer will point to the section of memory that will contain the collection of stored values. To add an element to the end of the collection, the number of values currently in the collection will also need to be known.

The most basic Array List will store both an array and a size.

3.1.1 Implementation details

In C, a heap pointer cannot naturally point to an endlessly large chunk of memory because each part of memory in the heap has a fixed size. Now, when you wonder about the size of an array, it is not always best to make it the same as the number of items in the array.

If you have an Array List where the size is both the number of elements and the available spots in the array and you add a new value, there won't be an "empty" spot for it, so a new spot needs to be created. This means the array has to be resized to have one more spot than before. The new value goes to the location based on the old size of the array. We use the old size as the new index, because of 0 indexing. For example, when the size is 0, the first index to insert a value is still 0.

Based on the above method described the append function for an array list of integers would look like the following code. It should be noted that a pointer to the list is passed into the function so the values of the list can be changed.

```c
#include <stdlib.h> // Allocation functions acquired

struct ArrayList // An Array List definition
{
   int size; // Number of values in the list
   int * array; // The pointer to the heap memory that has the values
};

void append(struct ArrayList * list, int value) // The append function
{
   list->array =  // Make the array 1 spot larger
      (int *) realloc(list->array, (list->size + 1) * sizeof(int));

   list->array[list->size] = value; // Store the new value at the old size

   list->size++; // Update the Array List size
}
```

The downside of using such an append is that many appends cause many `realloc` calls. Reallocating an array could involve (depending on the behavior of the memory manager) copying the old values into a new part of the heap—recall that `realloc` can be slow. All the copies added together gets **really** slow as the program executes.

Imagine the worst case where every append requires a copy. Let's compute the total number of values that are copied.

Append Iteration	Copies for Current Iteration	Total Copies
1	0	0
2	1	1
3	2	3
4	3	6
5	4	10
...	...	...
N	N-1	(N)(N-1) / 2

That expression can be found using some Analysis. If N was around a million, we would have around 500,000,000,000 copies. As of the time of writing these notes modern computers can easily perform about 100,000,000 of these copy operations per second, so a program would take a little over an hour to build the Array List. Hopefully we do not have to create too many lists of such a size.

3.1.2 Better Append/Structure

Since the copy can take longer time to complete as the execution of the program proceeds, a work around involves making the time between copies longer as execution of the program proceeds. How appends can avoid a copy is to not reallocate in each insertion; make the array of the `struct` a bit larger than necessary. The allocation size not equating to the number of elements in the array introduces a new problem "How does the program know when the array is full?" To solve this our structure will store an additional value: the capacity.

Without discussing the math in too much detail—Analysis will be covered later—a good rule of thumb is to double the capacity whenever a reallocation needs to be performed. Note the capacity should not start at 0, if the capacity will be doubled during each resize. Below is an example of a more efficient Array List,

```
#include <stdlib.h> // Dynamic memory functions get

struct ArrayList // Struct definition
{
   int size;    // The number of elements in the array
   int capacity; // The number of spots in the array
   int * array; // The pointer to the heap memory block
};

void append(struct ArrayList * list, int value) // Efficient append
{
   if (list->size == list->cap) // Check if the array in the list is full
   {
      list->capacity *= 2; // Double the capacity of the array

      list->array = // Update the heap memory block
         (int *) realloc(list->array, (list->capacity) * sizeof(int));
   }

   list->array[list->size] = value; // Store value at index based on old size

   list->size++; // Update Array List size, since 1 more stored value
}
```

In terms of overhead the above technique above in the worst case uses twice as much memory as necessary, but that is reasonable in modern systems. In the case where 1,000,000 values were stored, only around 2,000,000 needs to be allocated. Additionally the number of copies operations will be relatively low, again to be described in the Analysis Section.

3.2 Other Functions

You may depending on your purpose for your Array List may wish to create other functions. In this section we describe a few ones that are common to implement.

3.2.1 Initialization

When using Array Lists it is useful to make function for initializing (and sometimes creating or destroying) them. When an Array List is created it is empty. The number of elements contained (the size) would be zero. The capacity should be based on whatever default value you end up using. The larger the default values the less small reallocation needs to be done, but the more memory may be wasted. Below is an example of an Array List initialization function,

```c
#include <stdlib.h> // Dynamic Memory Functions

#define DEFAULT_CAPACITY 10 // Default capacity of the array list

struct ArrayList // Array List definition
{
   int size;    // The number of elements in the array
   int capacity; // The number of spots in the array
   int * array; // The pointer to the heap memory block
};

void initialize(struct ArrayList * list) // Initialize array list
{
   list->size = 0; // There are no values in the list

   list->capacity = DEFAULT_CAPACITY; // Set the capacity to the default value

   list->array = (int *) calloc(DEFAULT_CAP, sizeof(int)); // Create initial array
}
```

Note that using a default capacity of 0 would result in trying to allocate a block of memory that is 0 bytes. Allocation of a block of size 0 is `UB` and should be avoided. It would be better to set the array pointer to `NULL` if the default capacity is 0.

3.2.2 Iteration

Another popular function that can be implemented is iteration (looping over the contents). This could be important if values in the array need to be printed, modified, or checked. Iteration would require looping up to the size of the Array List and accessing the values in each index of the array member of the List. On the next page is an example that adds one to each value of the Array List,

```
struct ArrayList // Struct definition
{
   int size;    // The number of elements in the array
   int capacity; // The number of spots in the array
   int * array; // The pointer to the heap memory block
};

void incrementAll(struct ArrayList * list) // Function to increment all values
{
   for (int i = 0; i < list->size; i++) // Loop through all the values in the list
   {
      list->array[i]++; // Add to the i-th value in the array
   }
}
```

3.2.3 Removal

When removing a value there are several options available depending on what might need to be maintained about the list and the information regarding the removed element. The two main factors are

1. Is the removal by value or index?
2. Does the relative order of the remaining values need to be maintained?

The most efficient removal can be done if both the index is given and the order does not need to be maintained. When such a behavior is required, then removal can be done by swapping in the last value of the array into the index to remove from and decrementing the size of the list (see code below),

```
struct ArrayList // The definition of the struct
{
   int size;    // The number of elements in the array
   int capacity; // The number of spots in the array
   int * array; // The pointer to the heap memory block
};

void removeByIndex(struct ArrayList * list, int index) // Remove from an array list
{
    // Store the last value into the spot to remove
   list->array[index] = list->array[size - 1];

   list->size--; // Update the resulting number of list elements
}
```

If the relative order of the remaining value needs to be maintained, then a loop to shift the values down would need to be leveraged (see code below).

```c
struct ArrayList // Definition of structure
{
   int size;    // The number of elements in the array
   int capacity; // The number of spots in the array
   int * array; // The pointer to the heap memory block
};

// Function that removes and maintians order
void removeByIndex(struct ArrayList * list, int index)
{
   for (int i = index; i < list->size - 1; i++) // Move values down from index
   { // Note the loop ends at size - 1 to avoid Array OOB
      list->array[i] = list->array[i + 1];
   }

   list->size--; // Update the resulting number of list elements
}
```

The case could also be that the value to remove is given rather than the index. Such a function would require finding the value to remove first by looping through the values until the value to remove is found.

3.3 ArrayList of Data Structures

An Array List can be used to hold more than just integers; it could be used to store a List of strings or a List of `struct` (or `struct` pointers). Below is an example where an Array List holds only student `struct` (not pointers to students) along with several example function

```c
#include <stdlib.h> // Dynamic memory functions
#include <string.h> // String functions

struct Student // Student struct defintion
{
   char * name; // Dynamic string :O
   int creditHours; // Number of hours
};

struct ArrayList // Array List struct defintion
{
   int size; // The number of students in the list
   int capacity; // The number of spots in the array
   struct Student * array; // The array of students
};

void initializeList(struct ArrayList * list) // Initialize the values of a list
{
   list->size = 0; // Set up the size and capacity of the empty list
   list->capacity = DEFAULT_CAPACITY;

   list->array = (struct Student *) // Allocation the initial block of memory
      calloc(DEFAULT_CAPACITY, sizeof(struct Student));
}

void append(struct ArrayList * list, char * name, int credits) // Add to list end
{ // Assume name needs to be allocated in this function
   if (list->size == list->capacity) // Check if the array is full
   {
      list->capacity *= 2; // Double the capacity

      // Conintued on next page
```

```
      list->array = (struct student *) realloc(list->capacity,
         sizeof(struct Student)); // Reallocate the memory block
   }

   list->array[list->size].name = // Create the name dynamically
      (char *) calloc(strlen(name) + 1, sizeof(char));

   strcpy(list->array[list->size].name, name); // Copy the contents of the name

   list->array[i].creditHours = credits; // Copy the credit hours

   list->size++; // Update the resulting number of list elements
}

void cleanList() // Free the memory created from the list
{
   for (int i = 0; i < list->size; i++) // Loop and clean up all the names
      free(list->array[i].name);

   free(list->array); // Free the array
}
```

4 Basic Linked Lists

Data types can contain themselves. Users in a social networking sites might require their data to be stored in other Users. However, due to how structs are used by the compiler, a struct needs to be completely defined before we try to create a variable (or a field) with its type. For example the following segment of code won't compile when placed in a c program.

```
struct user
{
   char name[20 + 1];
   struct user friend;
};
```

The above sample code tries to use the user type in the user struct, but the type is not complete until the parser can reach the end of the struct definition. If the above code did compile, then creating an instance of the struct type listed above would require an unlimited amount of memory. A picture of an instance of the struct would look something like the following,

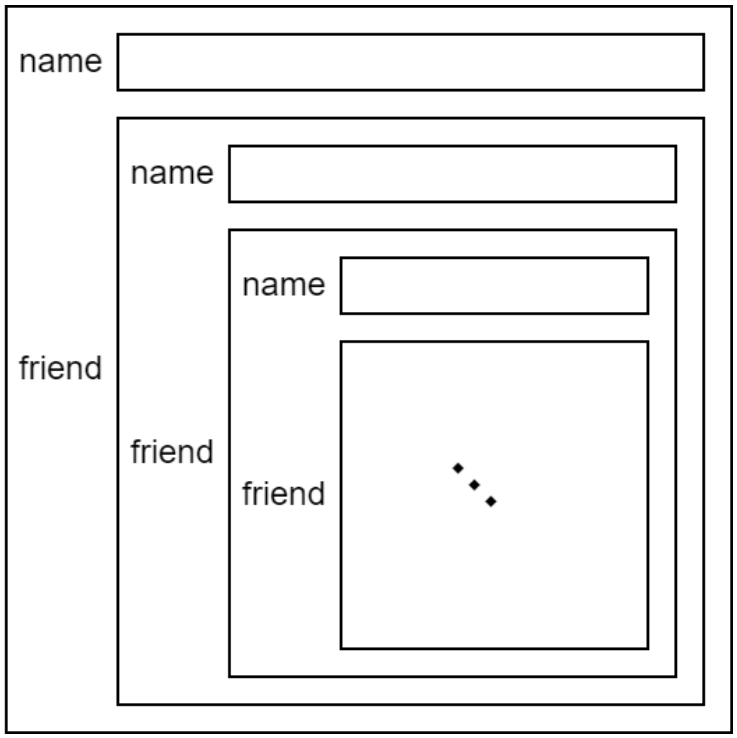


Figure 8: Example of the bad user `struct`.

But the repetition of the friend `struct` would never stop. To allow for a `struct` that could hold onto types that are like themselves, addresses should be used. This is possible since addresses should all be the same size.

4.1 Linked Memory

To enable a `struct` containing data corresponding to themselves, pointers are leveraged. A reference in a structure to a structure of the same type is known as a link. On the next page are a few examples of possible linked data structures,

```c
struct User // Struct definition
{
   char name[20 + 1]; // Some data
   struct User * friend; // A single link
};
```

```c
struct User // Struct definition
{
   char name[20 + 1]; // Some data
   int numFriends;
   struct User ** friends; // An array of links
};
```

For the second example listed above, if the friend member of the `struct` was a `User *` (an array of users) instead of a `User **` (an array of user references), then the `struct` would not be as useful. The problem with using an array of `User` (opposed to `User` references) is that the `User` that is immediately prior and immediately after a particular `User` will always be after and before that `User`. If multiple users want to use one particular `User` as a friend, then their arrays will have to have similar elements (other `User` instances), assuming that there is only one `struct User` per actual user.

With the assumption that a `User` could add an unlimited number of friends, then an Array List with some capacity within the `User struct` would be more efficient. Below is an example of how to create a more advanced user structure,

```c
#define DEFAULT_CAP 10 // The default capacity of the Array List in the user struct

typedef struct User User; // Alias
struct User // The user struct definition
{
   char name[20 + 1];
   int numFriends, friendCapacity; // Parts of an array list
   User ** friends; // The array part of the array list
};

User * createUser(char * name) // Function to create a user dynamically
{
   User * res = (User *) malloc(sizeof(User)); // Create a user dynamically

   strcpy(res->name, name); // Fill the name of the struct with the given name

   res->numFriends = 0; // Initialize the Array List of friends
   res->friendCapacity = DEFAULT_CAP;
   res->friends = (User **) calloc(DEFAULT_CAP, sizeof(User *));

   return res; // Exit the program
}
```

4.2 Linked Lists

Going a step further, these structures can be chained together. The only limit to how far the chain can go is based on the amount of memory your computer can have stored. This allows for a second manner of generating an unbounded sequence of data (a List). Since this method uses links instead of using an array this data structure is known as the **Linked List**. Each individual data snippet that comprises the Linked List is referred to as a Node. This terminology is taken from graph theory. For this same reason the links between the Nodes can be referred to as Edges. A example Node `struct` of a Linked List is on the next page,

```
struct Node // Struct defintion
{
   int data; // This can be whatever type your program needs
   struct Node * next; // The link
};
```

The last node of a Linked List is typically denoted by a Node that points to `NULL`. The first Node in the List is typically called the **head**, and the last Node in the list is called the **tail**. Below is a picture example of what a Linked List would look like if it contained the values 1 through 5 in order.

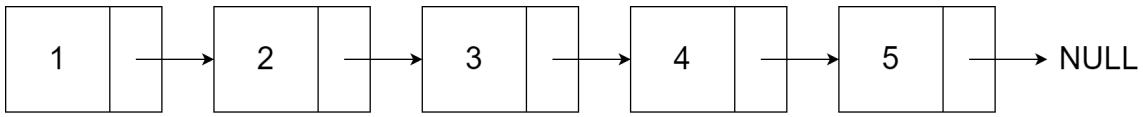


Figure 9: Example of linked list with 5 nodes.

To get access to the full Linked List only the address of the head is needed. Below is an example of a hard-coded Linked List,

```
typedef struct Node Node; // Struct Alias

struct Node // The Node struct definition
{
   int data; // This can be whatever type your program needs
   Node * next;
};

int main() // The main function
{
   Node node1, node2, node3, node4; // Declare all the nodes.

   node1.data = 8; // Fill in the data for Node 1
   node1.next = &node2;

   node2.data = 6; // Fill in the data for Node 2
   node2.next = &node3;

   node3.data = 7; // Fill in the data for Node 3
   node3.next = &node4;

   node4.data = 5; // Fill in the data for Node 4
   node4.next = NULL;

   return 0; // Exit program
}
```

4.3 Basic Linked List Functions

Several common Linked List functions exist. All the following functions will be implemented for a `struct Node` that looks like the following,

```
typedef struct Node Node; // Alias
struct Node // The Node struct definition
{
   int data; // This can be whatever type your program needs
   Node * next; // The address to the next node
};
```

The following functions will be covered,

- Create Node
- Insert Head
- Insert Tail
- Remove Head
- Remove Tail
- Print List
- Delete List

For the functions described above when a modification to the list is made, **the resulting head of the list will be returned**, since the head of the list describes the entire list. This is not required in all Linked List implementations. Returning the head is Dr. Meade's preference. In a later section an alternative will be discussed.

4.3.1 Create Node

If the Linked List is going to be created incrementally and dynamically, then a function to create a piece of the List (a single Node) would be helpful. A function to create a node would look like the following.

```
Node * createNode(int data) // Function to create a Node
{
   Node * res = (Node *) malloc(sizeof(Node)); // Create the node dynamically

   res->data = data; // Initialize the data in the node

   // Initialize the next pointer. It is unknown at this point what
   // the node will point to, but the node could be the only node in
   // the list. If the node is the only node, then the next pointer
   // should be NULL as this is the last node in the Linked List.
   res->next = NULL;

   return res; // Return the node and the data associated with it
}
```

4.3.2 Insert Head

A function to add values to a linked list would be useful in some scenarios. The Linked List could be passed in using the head since the head has all the nodes following it. When adding a value to the list it is important to know where the value should end up at. Does the value become the new head? Does the value become the new tail? Is the value going to come after some given value or node? Does the value go into the Linked List in a sorted order?

Below is an example of a function that will insert a value at the beginning of a Linked List,

```
Node * insertHead(Node * head, int data) // Insert at head of linked list
{
   Node * newNode = createNode(data); // Create the node dynamically
   newNode->next = head; // The old head will be the node that follows this new head
   return newNode; // Return the resulting head of the Linked List
}
```

4.3.3 Insert Tail

To insert a value at the end of a Linked List, a way to "walk" to the end of the Linked List would be required. The walk is necessary since the number of nodes in a Linked List is variable. Walking to the end of a Linked List could be achieved by using a loop. Below is an example of inserting a value at the end of a Linked List iteratively,

```
Node * insertTail(Node * head, int data) // Insert tail of linked list
{
   Node * newNode = createNode(data); // Create the node dynamically

   if (head == NULL) return newNode; // Handle the empty Linked List case

   Node * last = head; // Find the CURRENT last node of the list
   while (last->next != NULL)
   {
      last = last->next; // This last is not actual the last so move
   }
   last->next = newNode; // The old last will point to this new tail
   return head; // Return the resulting head of the Linked List
}
```

The above program is going to be slower than inserting at the head because of the loop.

4.3.4 Remove Head

In many applications it will be desirable to remove a node from a Linked List. Node removal could be (in some applications) at the beginning of the Linked List, at the end of the Linked List, or by some given value.

To remove the first value of the Linked List (the head), we only need to know what the head was. The resulting head of the Linked List would typically become the value after the old head (the head's next node). A function that returns the resulting head would free the old head and return whatever the old head's next value becomes. However, accessing the old head's next value after the free would result in a use after free. Due to the potential for use after free, a temporary pointer could be leveraged. Below is an example how removing the head can be done,

```
Node * removeHead(Node * head) // Remove the head of a linked list
{
   if (head == NULL) return NULL; // Handle the empty Linked List case

   Node * newHead = head->next; // Get the pointer to the resulting head of the list

   free(head); // Delete the old head
   return newHead; // Return the resulting head of the Linked List
}
```

4.3.5 Remove Tail

Tail removal is also possible. The function needs to find the last node. Note that an extra case of a single node needs to be handled since the head of the list would change. On the next page is an example of removing a node from the end of the Linked List,

```c
Node * removeTail(Node * head) // Remove the tail of a linked list
{
   if (head == NULL) return NULL; // Handle the empty Linked List case

   if (head->next == NULL) // Handle the single node case
   {
      free(head);
      return NULL; // No list is left over
   }

   Node * secondToLast = head; // Get the pointer to the node before the last node
   while (secondToLast->next != NULL) // Find the actual second to last node
   {
      secondToLast = secondToLast->next; // Not correct node... maybe the next one?
   }

   free(secondToLast->next); // Delete the old tail
   secondToLast->next = NULL; // Second to last node becomes the tail
   return newHead; // Return the resulting head of the Linked List
}
```

4.3.6 Print List

To print the list, we need to visit each node until we hit the end.

```c
void printList(Node * head) // Function to print a linked list
{
    while (head != NULL) // Loop over each node
    {
        printf("The number is %d\n", head->data); // Print contents
        head = head->next; // Move to the next node
    }
}
```

4.3.7 Delete List

To delete the list, we need to free each node. We already know how to delete the head of the list, so we can just repeatedly delete the head until the list is empty.

```c
void deleteList(Node * head) // Function to delete a linked list
{
   while (head != NULL) // While the list is not empty
   {
      head = removeNode(head); // Remove the current head
   }
}
```

4.4 Extra Practice

As extra practice consider making functions that do the following,

- Remove a node by value
- Reverses the list
- Sorts the list

List continued on next page...

- Breaks a list in 2 halves
- Moves the last value to the front
- Moves the front to the end.

5 Improved Linked List

To make linked list more efficient there are a few modifications that can be made. The way the tail functions were implemented both required loops. Loops take multiple operations to complete, and when analyzed it will become apparent that iterating to the end of the list each time does not work well with larger data sets.

This section will talk about 3 modifications to the basic linked list structure to provide varying degrees of performance improvements. The modifications can be summarized as follows,

1. Head Tail Storage
2. Circular Linked List
3. Doubly-Linked Linked List

5.1 Head Tail

One way to improve the a linked list it hold onto the information of the tail. Since the other nodes of the list to be accessed, the head should also be stored. A problem occurs when modifying the list. Sometimes both the head and the tail need to change in the same function. Modification of both pointers with a single return is tricky unless a complex return type with 2 values is returned—a `struct`. Alternatively, a reference to the head reference and tail reference could be passed so that what the addresses contained in the variables can change in the original scope. Both ways are not appealing.

The implementation incorporated in these notes is to pass in an address of addresses. The method is implemented by using a List `struct` to improve code readability. Below are the List `struct` and the Node `struct` that will be used,

```
typedef struct Node Node; // Alias to keep code shorter
typedef struct List List;

struct Node // Node struct defintion
{
   int value; // Place holder
   Node * next;
};

struct List // List struct definition
{
   Node * head, * tail; // Pointers to the head and tail
};
```

The key word for using this `struct` is **invariant**. An invariant is some high-level or low-level constraint that the program needs to abide by. In the basic Linked List there are several invariants such as,

- the address stored in the last node's next member is `NULL`,
- the first node, if it exists, needs to reach all other nodes in the list via next pointers,
- if the list is empty, then the address of head should be `NULL`.

In our modified version of the linked list are additional invariants required such as,

- the head and tail addresses need to agree on the,
- when the list is empty the address stored in tail should be `NULL`

This means that insertions and removal functions will need more pointer updates, and more corner cases may need to be considered. In this part of the notes you will find a few functions for linked list that contain both a head and a tail pointer. Example code for the following functions can be found below,

1. Create List
2. Delete List
3. Insert Head
4. Insert Tail
5. Remove Head
6. Remove Tail

5.1.1 Head Tail - Create List

The original create node function can be used because the node `struct` did not change. However, a new dynamic `struct` creation function can be implemented. The create list method is useful when the program needs to make multiple lists that could need to hold different values. Sometimes you might find yourself in need of a list of lists. Below is an example of how you could dynamically create a list in a function and return it to the calling function,

```
List * createList() // Function to instantiate dynamically and initialize a list
{
  List * res = (List *) malloc(sizeof(List)); // Create the list dynamically
  res->head = res->tail = NULL; // Make sure both pointers are to NULL
  return res; // Return the resulting list
}
```

There is some abuse of the **assignment operator** and associativity in the above statement. For as a refresher for some the assignment operator (`=`) is right associative in C. Which means that the right most assignment operator needs to execute first in the even of ambiguous ordering in an expression due to a lack of parenthesis. The right associativity means that the head and tail assignment line can accurately be represented using the following line of code.

```
  res->head = (res->tail = NULL); // Make sure both pointers are to NULL
```

The operator in the parenthesis will therefore assign the tail to `NULL`. You might be wondering what is the result of an assignment operator. The assignment operator returns the resulting value that was assigned. For the above example when the first assignment completes (the one in parenthesis) the line of code is effectively the following,

```
  res->head = NULL; // Make sure both pointers are to NULL
```

The result of the above line of code will be that head is also assigned to `NULL`. In essence the line in the original function assigns both pointers to `NULL` as the comment originally suggests.

5.1.2 Head Tail - Delete List

Since the possibility to create list dynamically exists, a function to clean up the memory allocated with them is useful for preventing memory leaks. Assume that the nodes are dynamically allocated but not values in the nodes are dynamically allocated (with the exception of the next node). The first function on the next page shows how Linked List deletion could be attempted but has a major flaw,

```
void deleteList(List * list) // Delete dynamically allocated linked list
{
   Node * cur = list->head; // Loop through all nodes of the list starting at head
   while (cur != NULL)
   {
      free(cur); // Remove the current node
      cur = cur->next // Move to the next node
   }

   free(list); // Free the list container
}
```

The above code has a Use After Free. The `cur` node reference is dereferenced to look at the next member after the `struct` has been `free`'d. Below is an example how the function could be fixed using a temporary pointer,

```
void deleteList(List * list) // Delete dynamically allocated linked list
{
   Node * cur = list->head; // Loop through all nodes of the list starting at head
   while (cur != NULL)
   {
      Node * tmp = cur->next; // Store the next node address in the list
      free(cur); // Remove the current node
      cur = tmp; // No more use after free since cur was derefered earlier
   }

   free(list); // Free the list container
}
```

The function may appear slow because we need a loop, but there is no getting around using a loop like behavior when every node needs to be deallocated.

5.1.3 Head Tail - Insert Head

In the basic version of the linked list there was a way to insert nodes to the front of the list. The same function can be created in this modified version of the linked with the need to pay extra attention to some corner cases. Below is an example of how insertion of a head of a linked list could be performed,

```
void insertHead(List * list, int value) // Insert a value to list front
{
   Node * newHead = createNode(value); // Create a list head

   if (list->tail == NULL) // Check if the list is empty
   {
      list->tail = newHead; // The tail should be this new node
   }

   newHead->next = list->head; // The rest of the list follows this new head

   list->head = newHead; // Put the new head into the list
}
```

The function does not need to return the head of the list since the list `struct` will store that information. The function's `return` type can now be `void`.

5.1.4 Head Tail - Insert Tail

Insertion to the tail can also be implemented in this modified version. However, in the basic linked list a loop up to the tail is required. In the version of the linked list where a tail reference is stored the loop is no longer required. Below is an example of a possible implementation,

```
void insertTail(List * list, int value) // Insert a value to list back
{
   Node * newTail = createNode(value); // Create a list head

   if (list->head == NULL) // Check if the list is empty
   {
      list->head = newTail; // This is the new head too
   }
   else // Consider the case where there is at least one node
   {
      list->tail->next = newTail; // The new tail is the node after the old tail
   }

   list->tail = newTail; // Update the tail pointer
}
```

5.1.5 Head Tail - Remove Head

Head removal is similar, but an extra case for handling when the list becomes empty is needed because of the invariant.

```
void removeHead(List * list) // Remove the front
{
   if (list->head == NULL) // Empty list?
   {
      return; // Do nothing...
   }

   Node * newHead = list->head->next; // Store the new head
   free(list->head); // Delete the head
   list->head = newHead; // Update the list head (NOT USE AFTER FREE)

   if (list->head == NULL) // Did the list get emptied?
   {
      list->tail = NULL; // invariant
   }
}
```

5.1.6 Head Tail - Remove Tail

However, for tail removal finding the resulting tail (after removal) is challenging. The resulting tail will be the node who's next pointer is the old tail. To find this value the list must be traversed. On the next page is an example of how the function could be implemented,

```
void removeTail(List * list) // Remove the last node
{
   if (list->head == NULL) // Empty list?
   {
      return; // Do nothing...
   }

   if (list->head == list->tail) // SPECIAL CASE: One node list?
   {
      free(list->head); // Free the only node
      list->head = list->tail = NULL; // Both NULL now
      return;
   }

   Node * newTail = list->head; // Find the new tail
   while (newTail->next != list->tail)
   {
      newTail = newTail->next;
   }

   free(list->tail); // Deallocate old tail
   list->tail = newTail; // Make the new tail the tail
   newTail->next = NULL;
}
```

5.2 Circular List

Some software developers that write their own Linked Lists with tail pointers might find using an extra `struct` annoying. I am one of those "developers". The information that the tail node contains is somewhat wasted. The tail's next in the above implementation points to `NULL`, but its next pointer could be leveraged to store more information. In some implementations the tail's next could point to the head instead. By pointing to the head (and storing the tail instead of the head of the Linked List) both the head and tail will be accessible by using a single pointer. Instead of returning the head of a Linked List, all the Linked List methods could return the resulting tail instead. Below is a picture of what the linked list would look like if drawn out,

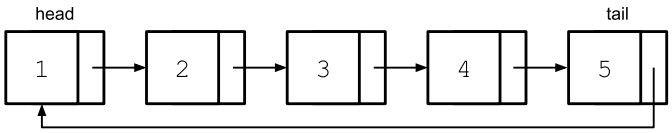


Figure 10: Example circular linked list.

All of the functions that use a circular linked list will take in a tail pointer and return a tail pointer if the list is modified. Below is an example of creating a new node,

```
Node * createNode(int value) // Function to dynamically create a node
{
   Node * res = (Node *) malloc(sizeof(Node));
   res->next = res; // Circle back to the resulting node
   res->value = value;

   return res; // Return the created and initialized node
}
```

Below is an example of how to insert at the head,

```
Node * insertHead(Node * tail, int value) // Head insertion function
{
   Node * newHead = createNode(value);

   if (tail == NULL) // Empty list case
   {
      return newHead;
   }

   newHead->next = tail->next; // Point to the old head

   tail->next = newHead; // Tail link to new head

   return tail; // Return resulting tail
}
```

Below is an example to create a new tail,

```
Node * insertTail(Node * tail, int value) // Tail insertion function
{
   return insertHead(tail, value)->next; // Magic
}
```

The above function works, because when either a head or a tail is inserted a node is created between the old tail and old head. However, the only difference is that a head insertion keeps the tail pointer the same and a tail insertion moves the tail pointer to the next node.

The downside to using a circular linked list is that iterating through requires tracking 2 nodes (and cannot be easily done using recursion). Below is an example of printing the contents of a circular linked list,

```
Node * printList(Node * tail) // Head insertion function
{
   Node * cur = tail->next; // Start at the head
   do
   {
      printf("%d -> ", cur->value);
   }
   while (cur != tail->next); // Loop until the head is seen

   printf("\n"); // Make the list on a single line
}
```

The circular linked list also requires a loop to remove the tail. The major benefit is that a List `struct` is not needed.

5.3 Doubly-Linked List

To avoid the need for a loop when removing the tail of a linked list the node before the tail needs to be known. To know the node before the tail, the address can be stored in the tail. In fact, every node should store the previous to make sure that after removal of tail the new previous node will be easy to find. Now every pair of adjacent nodes have 2 links between them. Because of the duplicity of the links the high-level version of the data structure is known as **doubly-linked linked list**. On the next page is an example of the node definition,

```
struct Node
{
   int value;
   Node * next; // Node towards the tail
   Node * prev; // Node towards the head
};
```

More invariants are required depending on details regarding the list structure. Either a circular linked list or a head tail linked list should be used in conjunction with the doubly-linked linked list to get quick access to the tail. The implementation used in these notes will be a circular list. In the circular list the head's previous should point to the tail, and the tail's next should point to the head. Either the head or tail can be used to store the full list since they that know the address of each other. In these notes the head will be stored. Below is an example of creating a node,

```
Node * createNode(int value) // Node creation function for Double-E linked list
{
   Node * res = (Node *) malloc(sizeof(Node));
   res->next = res->tail = res; // Circle back to self
   res->value = value;
   return res;
}
```

Below is an example of functions that insert to head and tail,

```
Node * insertHead(Node * head, int value) // Insert head function (return head)
{
   Node * newHead = createNode(value);

   if (head == NULL) // Empty list?
   {
      return newHead;
   }

   newHead->next = head; // Adjust the pointers from the new head first
   newHead->prev = head->prev;

   newHead->prev->next = newHead; // Connect the old nodes into the newHead
   newHead->next->prev = newHead;

   return newHead; // Return the new head of the list
}

Node * insertTail(Node * head, int value) // Insert tail function (return head)
{
    return insertHead(head, value)->next; // Magic
}
```

The insert tail function is able to leverage the insert head method the same way as was in the circular linked list. The most notable modification is the part of the code that connects the old nodes to the new head in the head insertion function. It should be apparent that when moving from some node to the `next` then back to `prev` the resulting location will be the original node. To ensure this behavior happens the address of the new head can be assigned to that pointer for using both `next`'s `prev` and `prev`'s `next`. The described pointer modification can be thought of as another invariant maintenance. On the next page are the example functions for head and tail removal,

```
Node * removeTail(Node * head) // Function to remove the tail
{
   if (head == NULL) // Empty list?
   {
      return NULL; // The list stays empty
   }

   if (head->next == head) // One node list?
   {
      free(head);
      return NULL; // The list becomes empty
   }

   Node * newTail = head->prev->prev; // Determine the new and old tails
   Node * oldTail = head->prev;

   newTail->next = head; // By pass old tail
   head->prev = newTail;

   free(oldTail); // This is done because we lost the reference

   return head; // Return resulting head
}

Node * removeHead(Node * head) // Function to remove the head
{
   if (head == NULL) // Empty list
   {
      return NULL;
   }

   return removeTail(head->next); // next is new head and head would be tail...
}
```

Note that in the above example no loops are needed. Due to the lack of loops, the head-/tail insertion/removal are all highly efficient at the sacrifice of some memory. The above described data structure can be used as an implementation of a Deque.

5.4 Extra Practice

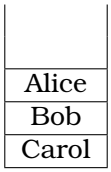
As extra practice consider making functions that do the following,

- Reverse a linked list
- Removes by value from any of the non-basic linked list
- Removes all nodes with a particular value
- Checks if values exists in the list
- Merges 2 lists together
- Sorts a list
- Moves the last value to the front
- Moves the first value to the back.

6 Stacks

When storing accessing values in a program frequently the most recently stored value will need to be accessed or removed. An example of such an access pattern is the **program stack**. The program stack will add new functions, and the most recently added function will be removed from the stack when returning. Additionally, the program will perform its execution within the most recent function added to the program stack. The described data structure is known as the **stack** and the behavior is Last In First Out (LIFO).

The stack is typically pictorially represented as a collection of data growing from the bottom up. The following would represent the possible contents of a stack,



The newest value in the collection would be Alice the oldest value in the stack would be Bob. The stack needs to be supported by some storage and a set of functions. The basic functions that are supported are the following,

- Push - Adds a value to the top of the stack
- Poll - Removes the top value from the stack
- Top - Access the top value on the stack

The stack is an Abstract Data Type (ADT), and can be implemented in a number of ways. In any implementation the stack needs to be unbounded. The 2 primary ways we could implement the stack would be using a Linked List or an Array List.

6.1 Stack Implementation (Array List)

The Array List needs to be able to support the 3 Stack functions. To support the **Push** operation the Array List can append values to the end quickly in most cases. Doing so would result in the top of the stack residing in the high index locations.

To support the **Pop** operation the highest index value would need to be removed since the newest values are in the high indices. To remove the last value in the Array List the memory associated with that content of the last value of the array should be cleaned up, after which the size of the Array List should be decremented. The time needed for the Pop is also realitively fast.

To support the **Top** operation, the last value of the Array List should be accessed. The runtime for accessing the last value of the Array List is also fast and does not depend on the number of values in the stack.

Below is a picture representing the contents of the stack in an Array List based on the contents described in the example earlier in this section,

Index	0	1	2	3	4
Value	Carol	Bob	Alice		

6.2 Stack Implementation (Linked List)

A Linked List could also be used to emulate a stack by implementing the 3 Stack functions. To support the **Push** operation, the Linked List could either insert to the head or insert to the tail of the Linked List. Depending on the implementation these functions could take either be fast without a loop or slow with a loop. If inserting to the head is used, then the head of the list would be the newest values (the top). If inserting to the tail is used, then the tail of the list would be the top instead.

To support the **Pop**, node removals would be needed. Whether the removal is from the head or the tail depends on where the top is located. If the top is at the head (from head insertion), then the Pop will remove the head of the list. Head insertion and Head removal is efficient in any linked list implementation (i.e. no loops), so using the head as the top of the Stack would be the most efficient implementation possible.

To support accessing the **Top** of the Stack as an operation, the Linked List needs to return the head of the Linked List based on the described implementation. Accessing the head of the list does not require a loop and the effort should not grow with respect to the number of nodes of the list (the number of values in the Stack).

Below is an example of how the Linked List would represent the stack described in the

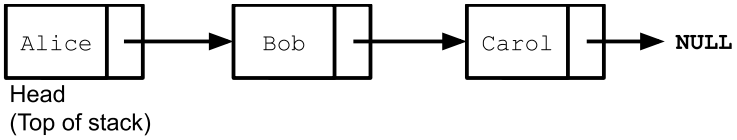


Figure 11: Example a Linked List based Stack.

The tail could be implemented as the top of the stack as well, but doing so would require an advanced Linked List implementation to ensure efficient functions.

6.3 Stack Application (Postfix Expression)

When writing programs you may often find yourself using complex mathematical expressions to assign values to variables. The mathematical operators should be performed in an order depending on the Order of Operations. There are a many ways in which the correct operator order can be found in mathematical expressions.

A naive approach would be to loop through the expression and find the highest precedence order. After the "first" operation is found, perform the operation and repeat. This approach would require nested loops and would be worse in terms of time than some how looping over the expression once or twice. Better methods for expression valuation exist. An example of a faster method leverages an expression format known as **Postfix** (AKA Reverse Polish) Notation.

In Postfix Notation the operators immediately after the operands (e.g. a number or another expression) executed on. Additionally, the order in which the operators should be executed are listed in order from left to right. The operator order matching the expression order ensures that parenthesis will not be necessary to distinguish execution order.

Below is an example of how to add 2 and 3 and multiply the result by 4,

```
2 3 + 4 *
```

It should be noted that because operand order matters for the subtraction and division, and expressions can involve performing operation execution in arbitrary orders there might not be a way to represent an expression with ALL the operations after the operands. For example consider the following standard (infix) notation expression.

```
(1 - 2) / (3 - 4)
```

The order of operations should be - - /

The first 2 operands operated on should be the 1 and 2; the second 2 operands should be the 3 and 4. Using the expression 1 2 3 4 - - is not correct because the 3 and 4 would be subtracted first and then the 2 and the result of 3 subtracted from 4 would be subtracted next. To handle this problem the first subtraction will be put inside the expression, but after the operands operated upon. The resulting postfix notation will be,

```
1 2 - 3 4 - /
```

There is always a way to create a postfix expression from an infix expression, where each operand occurs in their original order. Below is a method on how to perform such a conversion.

6.4 Infix Conversion to Postfix

The goal is to make a more efficient method for expression evaluation. For now our evaluator will work on converting from an infix expression into a Postfix expression. A single pass through the expression will be ideal. When examining a part (token) of the expression there will be 4 different cases the token can be in.

6.4.1 Case 1: Operand

If the token under observation is an operand (a number), then the token needs to come before its operation. The operand should appear before any future operands and the operator that is going to operate on it, and the token should appear after any previous operands. To ensure that the order of operands appears correctly, the current token can be printed out immediately.

6.4.2 Case 2: Operator

If the token is an operator it should be stored until the corresponding operands are printed. The operators need to come out in order based on their precedence. We can store the operators, but if some operator stored has a higher precedence, the higher precedence operator will need to be printed before the current token. A higher precedence operator should also occur before the next operand (the operand to the right of the current operator). The reason is that the earlier stored (higher precedence) operator will operate on operands strictly before the current lower precedence operator. See the below example of how a higher stored precedence operator (i.e. the multiply) could occur before some given token (i.e. the addition),

$$\text{Operand}_1 \text{ * } \text{Operand}_2 \text{ + } \text{Operand}_3$$

For the above reason all higher precedence operators should be printed at the point prior to storing the current operator. The order in which they should be printed is from highest to lowest precedence. The operators should be accessible from highest to lowest precedence. Additionally, once this current operator is stored it will be the highest precedence operator stored. The storage and accessing behavior is that of the last in (highest precedence) is the first out (highest precedence), AKA a stack.

Thus the operators are stored in an operator Stack. The procedure is while the top of the stack is higher precedence than the current token, print the top and pop. Once the top is no longer a higher precedence, the current token is printed.

6.4.3 Case 3: Open Parenthesis

When the token is an open parenthesis all future operators will have a higher precedence than all previous ones. Due to the change in precedence all the earlier operators in the stack should be ignored until the parenthesis is ended. There should be an entirely new stack until the parenthesis are ended.

Due to multiple groups nested parentheses many stacks would need to be created. The most recent stack is the one with the highest precedence. When a stack is removed (due to a closing parenthesis) the most recent stack should be removed. The situation requires a stack of stacks. The stacks of stacks can be implemented using just a single stack. To represent the bottom of a new stack a special character (such as an open parenthesis) can be inserted into the stack.

6.4.4 Case 4: Close Parenthesis

When a closing parenthesis is reached all the operators in the stack (until the next bottom stack character) should be removed from the stack since all the operators will execute before anything outside the parentheses and the next token should be some operator. Additionally the top most bottom stack character should be removed.

The resulting procedure is

- While the top of the stack is not a bottom stack character
  print the top of the stack and pop.
- Once the top of the stack is a bottom stack character pop the top of the stack

6.4.5 Code

Note that the above instructions does not handle the operators on the stack at the end of the expression. Those can be handled by printing the contents of the stack until it is emptied. Below is example of what the pseudo code for the function would look like,

```
Stack = empty stack
For token in expression:
  If token is an operand:
    Print token
  Else If token is an operator:
    While Top of Stack is higher precedence than token:
      Print Top of Stack
      Pop Stack
    Push token on Stack
  Else If token is an open parenthesis:
    Push token on Stack
  Else:
    While Top of Stack is not an open parenthesis:
      Print Top of Stack
      Pop Stack
    Pop Stack
While Stack is not empty:
  Print Top of Stack
  Pop Stack
```

Although the runtime might look like it would be worse than looping once through the expression due to the nested loops, each operator will be pushed onto the stack exactly once and therefore popped exactly once. Thus the number of times that the inner loop will execute is bounded by the length of the expression. Another helpful way to analyze the above function description is that each token will be printed at most once which is the instruction that can execute the most, due to the nested loops.

6.4.6 Conversion Example

Consider converting the following infix expression

```
(2 * (3 - 1) + 3) / (4 - 1 * 3)
```

Below is a trace of the stack as the Infix expression is parsed,

Token	(	2	*	(	3	-	1	)	+	3	)	/	(	4	-	1	*	3	)
Stack After Token	(	(	* (	((	((	- ((	- ((	* (	+ (	+ (		/	(/	(/	(/	- (/	- (/	* - (/	* - (/	/

Expression: `2 3 1 - * 3 + 4 1 3 * - /`

6.5 Postfix Evaluation to Result

Once the Postfix expression is formed, analysis can be performed more easily since the first operator found—from left to right—is the operator that should operate first. The operator should follow immediately after the operands for the respective operation. The last 2

operands are needed to evaluate the partial expression. To get access to the last 2 operands a Stack can be used, since the first element out is the last element in. The whole evaluation of a Postfix expression can be done with a single pass through the tokens handling one of 2 cases. After all tokens are processed the top of the stack should be the final evaluation of the expression.

6.5.1 Case 1: Operand

If the token is an operand, push the token onto the stack of operands. The token should be stored so that it can be evaluated when the corresponding operator is found.

6.5.2 Case 2: Operator

If the token is an operator, then the top 2 operands on the stack should be found and removed. The result of the operator on the 2 operands should be pushed onto the stack. Be careful when storing the operands on the top. The last operand in is the first one out. For this reason the second operand in the operation is the first out of the stack, and the first operand in the operation is the second out of the stack.

6.5.3 Code

Below is pseudocode for Postfix evaluation

```
Stack = empty stack
For token in expression:
   If token is an operand:
      Push token on Stack
   Else If token is an operator:
      Second = Top of Stack
      Pop Stack
      First = Top of Stack
      Pop Stack
      Push (First <token> Second) on Stack
Return Top of Stack
```

The function should need to pass through the expression at most one time. For this reason the code is better than nested loops.

6.5.4 Evaluation Example

Here is an example of evaluating the following expression,

2 3 1 - * 3 + 4 1 3 * - /

Below is the trace of the stack during parsing of the Postfix expression,

Token	2	3	1	-	*	3	+	4	1	3	*	-	/
Stack After Token										3			
			1						1	1	3		
		3	3	2		3		4	4	4	4	1	
	2	2	2	2	4	4	7	7	7	7	7	7	7

6.5.5 Combination

The 2 steps can be combined together into a single Infix Expression Evaluator. When an operator is printed from the conversion algorithm, evaluate the operator instead, and use the method from the evaluation algorithm. When an operand is printed in the conversion algorithm, instead of printing it, add it to the stack instead. On the next page is pseudo-code for combining 2 processes into one,

```
OperatorStack = empty stack
OperandStack = empty stack

# Loop through the expression
For token in expression:
   If token is an operand:             # CASE 1: Operand
      Push token on OperandStack
   Else If token is an operator:       # CASE 2: Operator
      While Top of Stack is higher precedence than token:
         # Evaluate using the high precedence operator
         Second = Top of OperandStack
         Pop OperandStack
         First = Top of OperandStack
         Pop OperandStack
         Push (First <Top of OperatorStack> Second) on OperandStack

         # Remove the high precedence operator from the stack
         Pop OperatorStack

      # Add the new Operator
      Push token on OperatorStack
   Else If token is an open parenthesis: # CASE 3: Open Paren
      Push token on OperatorStack
   Else:                                 # CASE 4: Close paren
      While Top of OperatorStack is not an open parenthesis:
         # Evaluate using the operator in the parenthesis
         Second = Top of OperandStack
         Pop OperandStack
         First = Top of OperandStack
         Pop OperandStack
         Push (First <Top of OperatorStack> Second) on OperandStack

         # Remove the operator from the inside the parenthesis
         Pop OperatorStack

      # Remove the parentheses
      Pop OperatorStack

# Handle all the lower precedence operators after the expression is
# exhausted
While OperatorStack is not empty:
   # Evaluate using the ending operator
   Second = Top of OperandStack
   Pop OperandStack
   First = Top of OperandStack
   Pop OperandStack
   Push (First <Top of OperatorStack> Second) on OperandStack

   # Remove the ending operator
   Pop OperatorStack

# Return the result
Return Top of OperandStack
```

6.6 Practice Questions

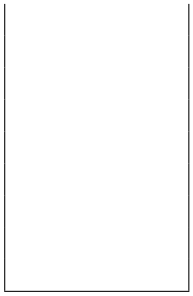
Try solving the following question from a previous test.

For the following infix expression, write the expression in postfix using the conversion method discussed in class. Write the contents of the stack after the 4th push.

```
5 * 3 + 2 * (4 + 2 / 1 - 6)
```

Postfix Expression _____

Stack



Answer

7 Queues

If we tried to create a data structure for processing users, such as holding orders for products (food, computers, shoes, etc.) or services (ticket purchasing, printing, etc.), a stack would not suffice. A developer would typically expect that the order in which processes are handled is the order in which they arrive, but stacks will remove/process elements in order of most recently added. The old process (the one to be first processed) is at the bottom of a stack and is inaccessible.

A new data structure is required. A data structure that behaves in a First In First Out (FIFO). The structure is commonly referred to as a **queue** and requires a few functions in order to work.

The queue like the stack needs to support functions and have ways to store the associated data. The functions implementations and data storage can vary depending on the implementation. That is the queue is also an Abstract Data Type (ADT). There are 2 common ways queues can efficiently be implemented.

There are 3 common functions that the queue should support,

- Enqueue - Adds a value to the back of the queue
- Dequeue - Removes the front[7] of the queue
- Front - Access the front of the queue

7.1 Implementation 1: Array List (Circular Buffer)

An array list can perform insertion into the high indices easily using append. However to remove from the opposite end, a typical array list implementation will need to slide elements down to fill in the empty spot created. The described behavior will allow for enqueueing without a loop (during most enqueues), but a dequeue would require a loop in the worst case. Below are pictures of enqueueing and dequeueing would look like in the implementation described so far,

Figure 12: Example a poor Array List based queue implementation.

To improve the implementation a modification to how the data is represented in the dynamic array is used. Instead of assuming all the values are at the front of the array list (starting at index 0), the array list will start at an arbitrary index, which will need to be stored in the array list data `struct`.

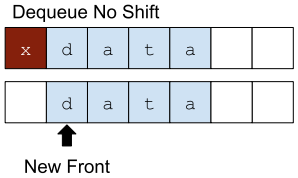


Figure 13: Example a better array list dequeue implementation.

On the next page is an example of the structure definition of the described,

[7] The queue needs to enqueue from the opposite end that the queue dequeues.

```
struct CircularBuffer
{
   int * array;
   int size;
   int cap;
   int frontIndex;
};
```

Rather than resizing anytime a value would be written off the end of the array, the index written to can wrap around the array back to the beginning. Below is a figure that depicts how the described queue would work if at the end of the array,

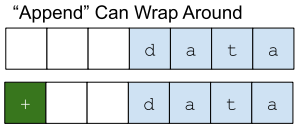


Figure 14: Example of enqueueing when at the end of the array.

The downside to the implementation is that expanding the array is tricker, since the data will need to be moved around. A single realloc won't suffice, the resulting data layout needs to account for the potential wrap around. Below is an example of how data will need to be reorganized when an expansion occurs,

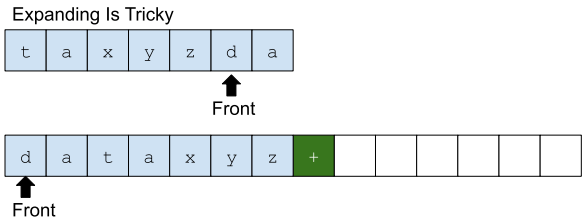


Figure 15: Enqueueing in a full array whose expansion requires moving the data.

The copying can be done using 1 or 2 for loops or 2 memcpy calls if you prefer. The above described implementation ensures that the enqueues and dequeues will behave in quickly over the course of the lifetime usage of the data structure.

7.2 Implementation 2: Linked List

Alternatively a linked list can be used. The enqueue could work by performing a head or tail insertion (depending on the implementation). The dequeue could work by performing a head or tail removal (depending on the implementation). It requires less effort to use the tail removal, so typically head removals are used. Since the enqueue and dequeue happen at opposite ends, the tail insertion is commonly used. The head removal and tail insertion only require a linked list with head and tail information or a circular linked list. The implementation allows for operations without loops.

Below is a picture of how the queue would look like in a linked list,

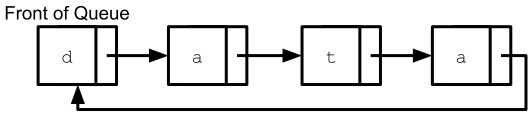


Figure 16: Representation of a Linked List queue implementation.

7.3 Deque

There is also a type of queue called the double ended queue (**deque**). The double ended queue is to handle enqueues and dequeues happening from either end of the data structure. The deque can be used to implement a stack by inserting and removing from the front only . The deque can be used to implement a queue by inserting to only one end and removing strictly from the other. The deque has applications in the algorithm called 0-1 Breadth First Search (BFS).

The implementation of a deque can be efficiently achieved using either a Circular Buffer as described earlier or a Doubly Linked Linked List.

7.4 Queue Application: Flood Fill

A common usage for a queue is the flood fill. Imagine if we wanted to know in a rectangular grid, with some cells blocked off all the cells reachable from a particular cell with a limited number of steps.

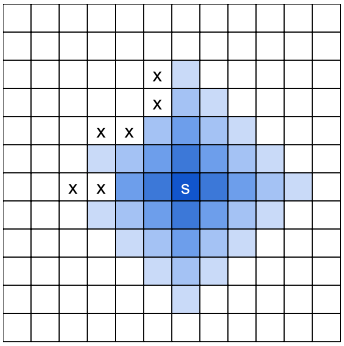


Figure 17: Example of water flowing from a start up to 4 spaces away in a grid where x's denote blocked cells

A naive implementation could involve generating all possible paths of the given number of steps. Generating all paths without recursion is tricky. Additionally, the number of paths is quite large, and there are many ways to visit the same node.

The way to improve this method is to not revisit a cell with a longer path length. To handle this properly all paths to cells should be processed in order of the length of the path, and a cell should only be processed once. The process can be thought of how water would flood a location. The water will spread out uniformly across a surface and (usually) not flow backwards. Once a spot is wet it does not become dry and they wet again using the same source of water.

Below is pseudo code of the method,

```
Queue = empty queue
Wet = 2D Boolean Grid
Enqueue Start to Queue
Start is Wet

While Queue is not Empty:
   Current = Front of Queue
   Dequeue Queue
   If Current is terminal length:
      Break
   For each Next location from Current:
      If Next is not Wet:
         Next is Wet
         Enqueue Next to Queue
```

8 Recursion

Functions and data structures can be **recursive**. Recursion is when something is self defined. If a function is executed by calling itself, then the function is recursive. If a struct has something akin to itself as a member, then it is also recursive.

8.1 Recursive Structures

Recursive data structures in C are not trivial to implement. Below is an example of an attempt of a recursive `struct` that does not compile,

```
// An attempt at a recursive struct
struct my_recursive_struct
{
    int value; // Some value in the struct

    // The same struct within the struct (DOES NOT WORK!!!)
    struct my_recursive_struct inside;
};
```

The above example does not compile because the type `struct my_recursive_struct` has not completed its definition prior to trying to use it inside the `struct` definition. The definition needs to be complete so the `struct` can be created with the appropriate size. This problem can be fixed by leveraging pointers, since the pointer size (the address size) is the same no matter what type the pointer is pointing to. Below is an example of a "recursive" `struct` that can compile,

```
// A 2nd attempt at a recursive struct
struct my_recursive_struct
{
    int value; // Some value in the struct

    struct my_recursive_struct * inside; // struct (as a pointer) within struct
};
```

The concept and usage of that structure was demonstrated in the section on Linked Lists.

8.2 Recursive Functions

A recursive function is a function that calls itself. At a high level a recursive function could look something like the following,

```
// An attempt at a recursive function
void function()
{
   // Some code
   function();
   // Some more code
}
```

The two required parts of a recursive function are,

1. The Recursive Call
2. The Base Case

8.2.1 Recursive Call

The part of the function that calls itself is known as the recursive call. The recursive call will usually pass in modified arguments when calling itself. Alternatively, some global memory

could be modified prior to the recursive call.

One problem that is frequently encountered when using recursion is that of the stack overflow. The program stack (the part of memory that stores the function calls and the local variables created statically in functions) is limited in size. When a function is returned from the memory on the stack is freed up. However, creating too many function calls prior to returning from any of them (recursing too deep) can cause the program to run out of memory. To avoid recursing too deep a second part of a recursive function is needed.

8.2.2 Base Case

The base case is the part of the recursive function that stops the recursive calls. It will enable returning from the function without calling the function any additional times. The structure of a recursive function could look like either of the following,

```
// A simple recursive function structure
type Function(args)
{
   // ==== BASE CASE PART ====
   // A base case to stop the recursive function
   if (args is a base case)
   {
      // Some code
      return some value;
   }

   // ==== RECURSIVE CALL PART ====
   // Some more code
   Function(modified args);
   // Even more code
   return some value;
}
```

```
// Another recursive function structure
type Function(args)
{
   // ==== RECURSIVE CALL PART ====
   // Check if the recursive call should me made
   if (args is in a recursive case)
   {
      // Some code
      Function(modified args);
   }

   // ==== BASE CASE PART ====
   // If we did not enter the above conditional, then
   // the arguments are in a base case
   // Some more code
   return some value;
}
```

There are other recursive functions structures that exists besides the above, but the ones above are common.

8.3 Examples

Here are a few examples of recursive functions.

8.3.1 Factorial

The factorial of a number is the product of all natural numbers up to the inputted value. For example the factorial of 4 is 4(3)(2)(1) = 24. Note that the factorial of 0 is 1. It should be noted that n factorial is also the same as n-1 factorial times n. Below is an example of a recursive function that computes the factorial of some non-negative integer.

```
// A recursive function that computes the value of n!
// Input: a non-negative integer, n
// Output: n factorial
int factorial(int n) {
   // Base case for 0 factorial
   if (n == 0) return 1;

   // Recursive call that computes n-1 factorial and multiplies by n
   return n * factorial(n - 1);
}
```

8.3.2 Towers of Hanoi

In a temple in a forest near a city called Hanoi are three stone towers lined up from East to West. On the Easternmost tower sits a set of 64 circular, golden disks. Since the beginning of time monks have tirelessly move the disks of gold with the intention of moving all 64 disks to the Westernmost tower. Due to the immense weight of the disks only 1 disk can be off a tower at a time, and due to the fragility of the disks a large disk can not be placed on top of a smaller disk. As it so happens once all the disks are on the Westernmost tower the world will end. The goal is to determine (assuming the monks move optimally) the number of disk movements required to end the world. Below is a picture example of the optimal moves required to move a tower of 3 disks from a starting tower to some ending tower,

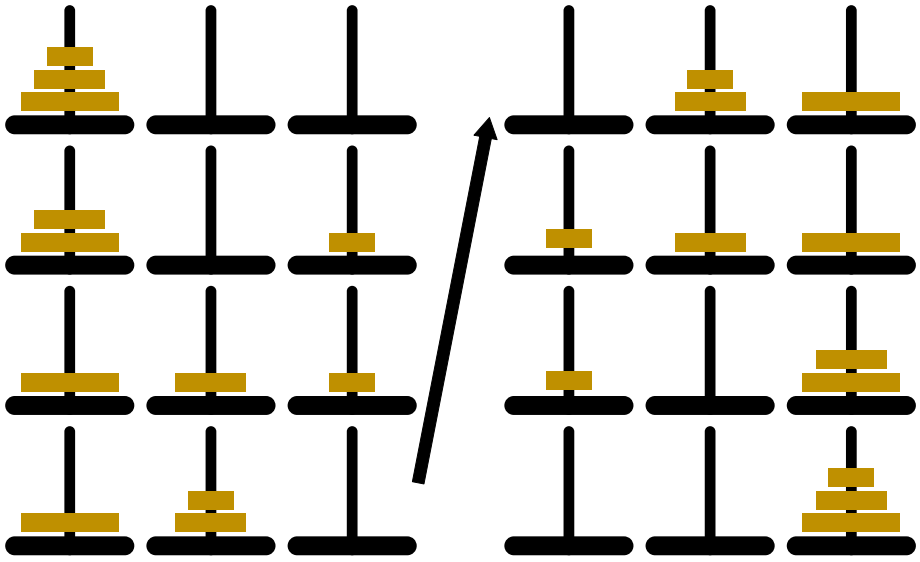


Figure 18: Example of solving 3-disk towers of Hanoi.

The first observation to make is that the bottom most disk should be moved exactly once. It will be moved from the East tower to the West tower. If the disk is moved from the East tower to the Center tower instead, then there must be no disks on top of the bottom disk and there must be no disk on the center tower. This would mean that all the other 63 disks would be on the Western tower, which would have required a set of moves to move the 63 disks from the East tower to the West tower. This same set of moves could have been

modified to move the 63 disks from the East tower to the Center tower. Allowing the bottom disk to be moved exactly once.

If moving a tower of 63 disks is known, then moving a tower of 64 disks can be done by moving the 63 disks, then moving the bottom disk, then moving the 63 disks again. The same method for moving a 64 disk tower can be applied to moving a 63 disk tower. In other words to move a 63 disk tower we need to move a 62 disk tower so the bottom disk can be moved, etc.

A recursive solution can be structured in the following manner,

```
// A recursive function that moves a n disk hanoi tower from
// start to end, using aux as the auxiliary tower
// Return the number of moves needed
int hanoi(int n, char * start, char * end, char * aux)
{
   // Base case for a tower with 0 disks
   if (n == 0) return 0;

   // Store the number of moves
   int moves = 0;

   // Move the top n-1 disks to the auxiliary tower
   moves += hanoi(n - 1, start, aux, end);

   // Move the n-th disk
   moves++;

   // Move the top n-1 disks from the auxiliary tower to the end
   moves += hanoi(n - 1, aux, end, start);

   // Return the moves needed
   return moves;
}
```

The number of moves required to solver a towers of Hanoi puzzle can be found using a formula, but the math required to solve this is a bit tricky. The analysis will be shown later.

8.4 Brute Force

Recursion can be used to try exploring a series of options that can be modeled using branching decisions. The 2 common explorations used by recursive brute forcers is that of **permutations** and **combinations**. In this section we will explore an application of permutations commonly referred to as Traveling Salesperson Problem (TSP). An example of an application of combinations will be demonstrated in a problem referred to as subset sum.

8.4.1 Traveling Salesperson

Suppose we are going to visit some towns, starting and ending at any desired town, and between the towns are roads that have some cost (time or toll) to use. We would ideally want to minimize the sum cost of visiting all the towns.

A naive approach could use loops. We could try all possible towns as the starting town (using a loop). Within that loop we could try all possible towns as a second town (using a loop). Within the inner loop we could try all possible towns as a third town (using a loop). And so on...

One challenge is that we also need to ensure that none of the towns are revisited. That could be done using an if statement. The code for the naive method using nested loops would look something like the following.

```
// A function that tries all town orders for only 5 towns
int sales5()
{
   for (int town1 = 0; town1 < 5; town1++) // Try all first towns
   {
      // Try all second towns
      for (int town2 = 0; town2 < 5; town2++)
      {
         // Ensure that the town has not already been visited
         if (town1 == town2)
         {
            continue;
         }

         // Try all third towns
         for (int town3 = 0; town3 < 5; town3++)
         {
            // Ensure that the third town has not already been
            // visited
            if (town1 == town3 || town2 == town3)
            {
               continue;
            }

            // Try all fourth towns
            for (int town4 = 0; town4 < 5; town4++)
            {
               // Ensure that the fourth town has not been visited
               if (town1 == town4 || town2 == town4 ||
                   town3 == town4)
               {
                  continue;
               }

               // Try all possible last towns
               for (int town5 = 0; town5 < 5; town5++)
               {
                  // Ensure that the last town has not been visited
                  if (town1 == town5 || town2 == town4 ||
                      town3 == town5 || town4 == town5)
                  {
                     continue;
                  }

                  // Compute the cost (USE ANOTHER LOOP)
               }
            }
         }
      }
   }
   return min_cost;
}
```

The method demonstrated on the previous page will not extend well. A loop will be needed for each town. If the number of towns increased we would have to manually add another loop. Recursion can be used to create nested loops without actually coding a loop. The below pseudo code demonstrates how recursion could be leveraged to make a variable number of nested loops,

```
// A function that emulates N nested loops
void NLoops(int N)
{
   If N is 0:
      // Handle the base case
      return
   Loop:
      // Since the code in in this loop, 1 less loop is needed
      NLoops(N-1)
}
```

The above method can be used to try all permutations (rearrangement) of values below is an example that can create a permutations of values 0 through N - 1.

```
// This function permutes the values 0 through N-1 given,
// * The value of N
// * How many values are currently in the permutation
// * An array that holds 0s and 1s representing if a value has bee
//   used (1) or not used yet (0)
// * The current permutation
// The permutations are built up from the beginning
void perm(int N, int index, int * used, int * perm)
{
   // Check if all N values are in the permutation
   if (index == N)
   {
      // Do something with the permutation
      return;
   }

   // Loop through all values
   for (int value = 0; value < N; value++)
   {
      // Check if the current value is used
      if (used[value] == 1) continue;

      // Put this value in the current location of the permutation
      used[value] = 1;
      perm[index] = value;

      // Fill in the rest of the permutation
      perm(N, index + 1, used, perm);

      // Remove the value from the permutation
      used[value] = 0;
      perm[index] = -1;
   }
}
```

8.4.2 Subset Sum

For the subset sum problem an array of values are given, and a target sum is given. The goal is to determine if a selection of the values in the array sum to the given target. The selection could be nothing, the whole array, or anything in between. The problem is reduced to determining for each value is it going to be in the sum or not. We can use a series of nested loops to determine the multiplicity of each value in the array. On the next page is an example of the code for 5 values,

```
int canSum5(int * arr, int target)
{
   for (int i1 = 0; i1 < 2; i1++)
   {
      for (int i2 = 0; i2 < 2; i2++)
      {
         for (int i3 = 0; i3 < 2; i3++)
         {
            for (int i4 = 0; i4 < 2; i4++)
            {
               for (int i5 = 0; i5 < 2; i5++)
               {
                  int sum = 0;
                  sum += i1 * arr[0];
                  sum += i2 * arr[1];
                  sum += i3 * arr[2];
                  sum += i4 * arr[3];
                  sum += i5 * arr[4];
                  if (target == sum)
                  {
                     return 1;
                  }
               }
            }
         }
      }
   }
   return 0;
}
```

The above code might be easier to code than the earlier TSP example, but it still requires a number of loops based on the length of the array. Changing the code length is not good. The arbitrary number of loops can be emulated using the recursion. On the next page is an example of how an arbitrary array length can be handled recusively,

```c
int canSum(int * arr, int n, int target)
{
   // Check if target is easy
   if (target == 0)
   {
      return 1;
   }

   // Check if no values can be used
   if (n == 0)
   {
      return 0;
   }

   // Try both possibilities (value in sum or not)
   for (int i = 0; i < 2; i++)
   {
      int * newArr = (arr + 1);
      int newN = n - 1;
      int newTarget = target - (i * arr[0]);

      // Check if the new target can be reached using the reduced array
      if (canSum(newArr, newN, newTarget))
      {
         return 1;
      }
   }

   // No solution found
   return 0;
}
```

8.5 Linked List Recursively

Many linked list functions can be done recursively if using the basic version of the linked list. Below is an example of how insertion into the tail of a linked list can be done recursively,

```c
Node * insertTail(Node * head, int value)
{
   if (!head) // Base Case: Empty list
   {
      return createNode(value);
   }

   // Insert into the rest of the list
   head->next = insertTail(head->next, value);

   // Return the head of the list
   return head;
}
```

8.6 Practice Problems

For additional practice try writing recursive functions to

- print the value of all the nodes in a linked list
- reverse a linked list
- remove all nodes of a particular value from the linked list
- counts the number of subsets that sum to a particular value
- counts the number of permutations of N values with exactly K inversions[8]

[8] An inversion is a pair of indices such that the values of the pair are out of order.

9 Analysis (EXTREMELY IMPORTANT)

Many different algorithms (or data structures) can be used to solve the same problem. Computer Scientists want to be able to compare the algorithms and order in terms of efficiency. The two primary ways to compare algorithms are,

1. **Space** - the memory required to store the information during execution
2. **Time** - could be represented as the number of simple instructions[9] required to execute the program on the given inputs.

We can't really use the actual clock cycles or runtime, because the same program can compile differently in different systems, and even in the same system. Background processing, or differences in values/branch prediction can cause varying behavior. Examples of simple operations includes,

- **Mathematical Operators/Incrementors** - For example `+`, `-`, `*`, `/`, `%`, `+=`, `-=`, `/=`, `++`, `-`, etc. It should be noted that `/` and `%` typically take more clock cycles than operators like `+`, `-`, `*`.
- **Binary Operators** - For example `>>`, `|`, `&`, `<<`, `^`, $\sim$. These operators are all very fast.
- **Pointer Dereference** - For example `arr[i]`, `myStruct->myMember`, `*x`. Practically speaking, getting values from an address can take a long time depending on cache performance. They are treated as constant.
- **Primitive Assignment** - For example `x = 1`, `str[i] = 'c'`, `float1 = float2`, etc. Assigning a `struct` to the value of another `struct` could require many operations depending on the size of the `struct`.
- **Simple Comparison** - For example x != 1, str[i] == 'c', float1 < float2, etc. String comparison can require many comparison in bad cases, and `struct` comparison could also require many operations depending on the size of the `struct`.
- **Branches** - For example `if`, `break`, `continue`, `return`, moving to the top of a `for` or `while` loop when at the end. Note that loops could require multiple branches, which means that a single for loop could easily result in multiple simple operations.

Due to the the inconsistencies of clocks on different hardware and different systems, runtimes are typically approximated by dropping constant factors. For example if $50N$ simple operations are expected for some input N, then the algorithm is on the Order of approximately N operations, since it is hard to predict how many total clock cycles all the operations will take, but it is expected to be CN for some constant C. When the order is CN we say that the algorithm is **linear**, since the number of operations is linear with respect to the input N. When the order is C (not dependent on N) we say that the algorithm is **constant**, since the number of operations is constant with respect to the input N.

Additionally, when talking about order, the input values of N are imagined to be quite large, so an order of N + 1 would mean that the value of 1 has no significant impact on the actual runtime. Thus the order of N + 1 would be converted to an order of N.

9.1 Order Functions

Rather than using the word/phrase "order of F(N)", there are symbols that denote this. There are 3 main types of limits we will consider for class, but there are others.

1. Big Omicron - Most common; referred to as Big-Oh
2. Big Omega - Almost never used

[9] Different simple instructions might require a different number of (constant) clock cycles.

3. Big Theta - Should be used whenever possible

Try to avoid using equality for order approximations (e.g. $f(x) = O(g(x))$). It is more formal to use either membership ($\in$) or the approximate symbol ($\approx$). Both $4N = O(N)$ and $N = O(N)$, which gives the appearance that $4N = N$, which could infer that N is 0 or 4 is 1.

9.1.1 Big Omicron (O)

The Big Omicron (Big-Oh) is an upper bound on the approximate number of operations. We use the capital letter O to denote Big-Oh. The following examples are ALL true,

- $3N \in O(N)$
- $N \in O(N^2)$ Because N^2 grows faster than N.
- $10^9 N \in O(N)$ theoretically

The following examples are NOT true.

- $3N \in O(1)$
- $N^2 \in O(N)$
- $N \in O(log(N))$

Calculus Definition Since we are considering the growth for large N there is a calculus way to define Big-Oh,

$$f(n) \in O(g(n)) \equiv \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Quantifier Definition A formal quantifier definition could look like the following

$$f(n) \in O(g(n)) \equiv \exists n_0, c \in \mathbb{R}^+ (\forall n > n_0 f(n) \leq cg(n))$$

In English there is some starting bound and constant such that g scaled by said constant is larger than f.

9.1.2 Big Omega

Big Omega (Ω) is how to denote a lower bound on growth. Its mostly used when establishing the exact bound denoted by Big Theta.

9.1.3 Big Theta

Big Theta (Θ) is used when the growth of a function can be expressed as growing both slower than and faster than the scale of another function. A function might not have a Big Theta approximation, but such functions are rare. $n + 1$ is in Big Theta of n.

If the exact number of operations is known, the theta bound can be found by

1. Dropping constant factors
2. Dropping non-dominating terms (as n gets large)

For the function $n^6 + 100n$, the value 100 is a constant factor and the term n is non-dominant. Thus the function $n^6 + 100n$ is in $\Theta(n^6)$.

9.1.4 Function Growth

When you have an order approximation you can determine for large input values what functions will be larger than other functions. Any function that is linear—$\Theta(n)$—will be less than any function that is **quadratic**—$\Theta(n^2)$—because $n \in O(n^2)$, but $n \notin \Theta(n^2)$. When $f(n) \in O(g(n))$, but $f(n) \notin \Theta(g(n))$, we sometimes says that $f(n) \in o(g(n))$, or $f(n)$ is in little omicron of $g(n)$. Little omicron will not be on the foundation exam, but it can be used to say that function "grows" strictly slower than other functions. In this section programs will express their runtimes as order approximation of the number of simple operations. Functions that grow slower will use less operations for larger input sizes. Programs that have slower growing runtimes will execute faster and be more desirable.

9.1.5 Logs

In order analysis, logs can be tricky. For instance when considering the function $\log_2(n)$ vs $\log_{10}(n)$. It appears that $\log_2$ grows faster than $\log_{10}$. However, the two functions approximate the same function. That is they have the same growth. The identical growth is a result of how log bases can be converted. Note

$$\log_{10}(n) = \log_2(n) \cdot (\log_2(10))$$

In the case above the $\log_2(10)$ is a constant factor that can be dropped for the sake of approximation. For similar reasons all logs have the same growth. The base of a log is not listed when writing the order (i.e. $O(\log(n))$ is used instead of $O(\log_2(n))$ or $O(\ln(n))$). Additionally $\log(n^2)$ has the same growth as $\log(n)$, because of the following log trick,

$$\log(n^2) = 2\log(n)$$

The last thing to note is that $\log(n)$ is in $O(n^c)$ for any $c > 0$. Which means that $\log(n)$ has a very slow theoretical growth.

9.1.6 Exponentials

A polynomial runtime is a runtime in $O(n^c)$ for some constant $c \geq 0$. Exponential runtime is $\Theta(c^n)$ for some constant $c > 1$. Exponential is always faster growing than polynomial.

Rule of Thumb In terms of growth log grows slower than polynomial which grows slower than exponential.

9.1.7 Example Questions

Question 1. Suppose we have the following loop,

```
for (int i = 0; i < n; i++)
{
   x += i;
}
```

How many operations are performed in the above loop with respect to n? Answer

Question 2. What is the simplified order approximation of $n^3 - 100n + 6$? Answer

Question 3. Order the following functions from slowest growth (left) to fastest growth (right).

$$n^{2.5} \quad n! \quad \sqrt{3n} \quad 10n \quad \log_4(n^4) \quad 2^n$$

Answer

9.2 Summations

Summations and summation properties are used to determine a closed form number of operations taken by code segment when using loops. Consider the following segment of code,

```
for (int i = 0; i < n; i++)
{
   for (int j = 0; j < i; j++)
   {
      ans += j * i;
   }
}
```

We can express the number of times `ans` is incremented by the following expression,

$$0 + 1 + 2 + 3 + ... + (n - 1)$$

The number of times `ans` is incremented will approximate the number of simple operations, since there is a constant number of operations happening per each `ans` incrementation. The addition of those terms—to discussed later in this section—becomes $\frac{n(n-1)}{2}$. When approximating using the functions described in the section above the result is $\Theta(n^2)$.

9.2.1 Notation

Writing out the summation with terms every time is annoying and informal. Notation is introduced to represent the sum of many simple terms. In mathematics summations are thought of as the sum of function evaluated at points between two integral values. In other words a summation should be able to be represented as,

$$f(a) + f(a + 1) + f(a + 2) + \cdots + f(b)$$

The Greek letter, capital Sigma (Σ) is used. The equation below the Σ is typically describing what variable will be incremented and at what value it starts (inclusive). Above the Σ denotes the last value at which the variable will be during the summation (inclusive). The expression to the right of the Σ denotes the function in which the value of the variable is substituted. Below is an equation demonstrating how Σ is used,

$$\sum_{i=a}^{b} f(i) = f(a) + f(a + 1) + f(a + 2) + \cdots + f(b)$$

9.2.2 Properties

There are a few summation rules that can make reduction of summations easier. This section will cover 5 popular rules. Each rule is an equation and can be used to substitute in either direction.

1. **Constant Factor.**

   Constant factors can be distributed into and out of summations. Below is an example of how this rule can be used in an arbitrary summation.

   $$\sum_{i=a}^{b} c \cdot f(i) = c \sum_{i=a}^{b} f(i)$$

   This rule can be shown using distribution. The value of c cannot depend on the variable of the summation for this technique to work. However the value of c could depend on values that are changing that do not directly or indirectly rely on the variable's value.

2. **Splitting Summation of Functions.**

   If the summation is composed of two functions added together, then each of the functions can be split into its own summation.

$$\sum_{i=a}^{b}(f(i)+g(i)) = \sum_{i=a}^{b} f(i) + \sum_{i=a}^{b} g(i)$$

   This rule can be shown using associativity and commutativity of addition. **This technique DOES NOT apply to product of functions.**

3. **Merging Bounds.**

   If one summation picks up on one end other the other the summations can be merged. Below is an example of what the technique looks like with arbitrary values.

$$\sum_{i=a}^{b} f(i) + \sum_{i=b+1}^{c} f(i) = \sum_{i=a}^{c} f(i)$$

   Because summations are inclusive on both bounds, the two summations cannot have the same middle bound. If the middle bound is shared, then this technique can be leveraged to remove one of the terms from either summation, then combine two of the remaining summations. Below is an example of what that might look like,

$$\begin{aligned}
\sum_{i=a}^{b} f(i) + \sum_{i=b}^{c} f(i) &= \sum_{i=a}^{b} f(i) + \left(\sum_{i=b}^{b} f(i) + \sum_{i=b+1}^{c} f(i)\right) \\
&= \sum_{i=a}^{b} f(i) + \left(\sum_{i=b+1}^{c} f(i) + \sum_{i=b}^{b} f(i)\right) \\
&= \left(\sum_{i=a}^{b} f(i) + \sum_{i=b+1}^{c} f(i)\right) + \sum_{i=b}^{b} f(i) \\
&= \sum_{i=a}^{c} f(i) + \sum_{i=b}^{b} f(i) \\
&= \left(\sum_{i=a}^{c} f(i)\right) + f(b)
\end{aligned}$$

4. **Bounding From 1.**

   The lower bounds can be changed to a smaller, different value, but the terms introduced to complete the summation must be removed. Effectively 0 is added to the summation where 0 is the addition of the missing terms and the negation of the missing terms. Below is an example of what this would look like,

$$\sum_{i=a}^{b} f(i) = \sum_{i=1}^{b} f(i) - \sum_{i=1}^{a-1} f(i)$$

   Because a is the lower bound in the left hand side, the value will be in sum due to the inclusion of the upper and lower bounds. The upper most value missing from the summation is the value when i is $a-1$. The rule for bounding starting at 1 is useful for many known closed forms and it can be derived from the merging bounds property discussed in the previous summation property.

5. **Bound Substitution.**

   The variable that is used to increment the summation can be changed to 1 without introducing an extra summation, **but** it requires modifying the function on the inside of the summation and the upper bound. To decrease the starting value of the variable we have to remove some value, that same value removed must be added back into the function, and removed from the top bound to ensure the value of the terms and the number of terms remains the same. Below is an example of how it can be applied,

$$\sum_{i=a}^{b} f(i) = \sum_{i=a-(a-1)}^{b-(a-1)} f(i+(a-1)) = \sum_{i=1}^{b-(a-1)} f(i+(a-1))$$

9.2.3 Summation Closed Form

When evaluating summations, if the bounds get large or based on variables, then computing the result of the summation can take a long time or be impossible to represent using a single number. Summations typically can be represented using a bounded (i.e. does not depend on a variable) amount of mathematical operations via some other expression. A **Closed Form** of a summation is an equation that uses a bounded number of operators to express it. There are many summations that are common enough to have known closed forms. Below are common **arithmetic summations** and their closed forms (which can be found on the Foundation Exam's formula sheet),

$$\sum_{i=1}^{n} 1 = n$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \frac{n(n+1)n(n+1)}{4}$$

The arithmetic summations have a constant power (i.e. not depending on i), and raise the incrementing variable (i.e. i) to that constant power.

There is also a common **geometric summation** closed form, and although it is not on the Foundation Exam's formula sheet, it does appear on the Foundation Exam questions from time to time. Below is the geometric summation for $a \neq 1$,

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$$

For example the closed form of

$$\sum_{i=1}^{n^2} (i+n)$$

can be found using the following logic,

$$\begin{aligned}
\sum_{i=1}^{n^2}(i+n) &= \sum_{i=1}^{n^2} i + \sum_{i=1}^{n^2} n \\
&= \tfrac{n^2(n^2+1)}{2} + \sum_{i=1}^{n^2} n(1) \\
&= \tfrac{n^2(n^2+1)}{2} + n\sum_{i=1}^{n^2} 1 \\
&= \tfrac{n^2(n^2+1)}{2} + n(n^2)
\end{aligned}$$

The order approximation of that summation is $\Theta(n^4)$

9.2.4 Nested Summations

The summation of a summation can also be evaluated just like loops with in loops are possible in programming. A nested summation is a summation that is contained as a term within a larger summation. When evaluating a nested summation the sum on the inside should be evaluated first. The value of the following summation can be evaluated in the following manner,

$$\sum_{i=1}^{3}\sum_{j=i}^{2i} i \cdot j$$

$$= \sum_{i=1}^{3} i \cdot \sum_{j=i}^{2i} j$$

$$= \sum_{i=1}^{3} i \cdot \left(\frac{2i(2i+1)}{2} - \frac{(i-1)(i)}{2}\right)$$

$$= \frac{2(3)}{2} - \frac{0(1)}{2} + 2\frac{4(5)}{2} - 2\frac{1(2)}{2} + 2\frac{6(7)}{2} - 2\frac{2(3)}{2}$$

9.2.5 Number of Operations as Summations

The purpose behind learning summations is to model the number of operations as a summation. By then using summation properties the number of operations can be converted to an equation expressed in terms of the input size. The goal is to find the a summation that properly approximates the number of operations. The trick is to usually express loops as summations, program functions as math functions, and every simple operation as a constant 1.

Example 1. Consider the following code segment that was introduced earlier in this section,

```
for (int i = 0; i < n; i++)
{
   for (int j = 0; j < i; j++)
   {
      ans += i * j;
   }
}
```

As mentioned when the problem was first introduced—the simple operations that we should most concern ourselves with is the `ans` increment. The number of times the `ans` is incremented can be expressed using a nested summation that looks like the following,

$$\sum_{i=0}^{n-1}\sum_{j=1}^{i} 1$$

Using the summation closed form on the inside we get,

$$\sum_{i=0}^{n-1} i$$

which by splitting the summation becomes,

$$\sum_{i=0} 0i + \sum_{i=1} n - 1i$$

By summation evaluation of the left sum and a summation closed form we get,

$$0 + \frac{(n-1)(n-1+1)}{2}$$

By mathematical manipulation,

$$\frac{1}{2}n^2 - \frac{1}{2}n$$

The order approximation of which is $\Theta(n^2)$.

Example 2. Below is another code segment to analyze,

```
for (int i = 0; i * i < N; i++)
{
   ans += i * i;
}
```

Note that the above segment does not stop when `i` is `N`, necessarily. The program stop when $\texttt{i}^2 \approx \texttt{N}$, or when `i`—the number of iterations—is approximately the $\sqrt{\texttt{N}}$. Thus the number of operation is roughly,

$$\sum_{i=0}^{\sqrt{N}} 1 = \sqrt{N} + 1 \in \Theta(\sqrt{N})$$

Example 3. Below is a third code segment to analyze,

```
for (int i = 1; i < N; i+=i)
{
   ans++;
}
```

The runtime of the above segment is $\Theta(\log(\texttt{N}))$. It might look like $\Theta(\texttt{N})$, but `i` doubles every iteration. Thus `i` can be thought of as 2^x where x is the number of iterations. When 2^x is N the loop stops. The number of iterations (i.e. x) is $\log_2(\texttt{N})$ when the loop stops.

9.2.6 Harmonic Series

The harmonic series is a special summation. The N-th value is the following,

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \cdots + \frac{1}{N-1} + \frac{1}{N}$$

To find the result we will generate a lower and upper bound. The **lower bound** is created by moving all fraction down. The fractions are moved down by increasing the denominators. The denominators become the next power of 2 (if not already a power of 2). The summation looks something like the following,

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \cdots + \frac{1}{2^x} + \frac{1}{2^x}$$

The first term is 1, but all the other terms when grouped with their identical value each sum to $frac12$. The number of $\frac{1}{2}$s that exist is x. The total is therefore 1 + $(\frac{1}{2})x$. To find the value of x use the fact that N should be approximately 2^x. Thus by taking the log of both sides, it can be shown that x = $\log_2(N)$. By using the simplification steps the sum can be shown to be in $\Omega(\log(N))$.

The **upper bound** is created by moving all fractions up. The fractions are moved up by decreasing the denominators. The denominators become the previous power of 2 (if not already a power of 2). The summation looks like the following,

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{2}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{8}+\frac{1}{8}+\cdots+\frac{1}{2^x}+\frac{1}{2^x}$$

All terms when grouped with their identical value each sum to 1. The number of 1s that exist is $x+1$. The total is therefore $x+1$. To find the value of x use the fact that N should be approximately 2^x. Thus by taking the log of both sides, it can be shown that $x = \log_2(N)$. By using the simplification steps the sum can be shown to be in O(log(N)).

Thus the sum is in Θ(log(N)).

9.2.7 Practice Problem

Question 4. What is the closed form for the following summation?

$$\sum_{i=n}^{3n}(i^2+2)$$

Answer

Question 5. What is the runtime of the following segment of code in terms of input value N?

```
int ans = 0;
for (int i = 0; i < N; i++)
{
   ans+=i;
}
```

Answer

Question 6. What is the runtime of the following segment of code in terms of input value N?

```
for (int i = 0; i < N; i++)
{
   for (int j = 0; j * j < i; j++)
   {
      ans++;
   }
}
```

Answer

Question 7. What is the runtime of the following segment of code in terms of input value N?

```
for (int i = 0; i * i < N; i++)
{
   for (int j = 0; j < i; j++)
   {
      ans++;
   }
}
```

Answer

=

Question 8. What is the runtime of the following segment of code in terms of input value N?

```
for (int i = 1; i < N; i++)
{
   for (int j = i; j < N; j+=i)
   {
      ans++;
   }
}
```

Answer

9.3 Recurrence Relations

A recurrence relation is used to express either the runtime or the memory usage of a recursive function/data structure. A recurrence relation is a function that is self-defined.

A recurrence relation could be used to express the runtime towers of Hanoi puzzle. The towers of Hanoi solution that moves an N disk tower has to perform the sequence of steps to move an $N-1$ disk tower, a single disk, and finally, a second $N-1$ disk tower. When the tower has exactly 1 disk a single disk movement is required. The number of steps/operations could be written using the following recursive math function,

$$H(N) = H(N-1) + 1 + H(N-1)$$
$$H(1) = 1$$

Notice that like a recursive programming function a recurrence relation has 2 main components,

1. Recursive Part
2. Base Case

A recurrence relation is unfortunate, because it cannot be easily put into an order approximation form. Typically, the function needs to be reduced such that there is no recursion in the function's definition. There are 2 main methods to simplify a recurrence relation

1. **Iterative method** - Either back substitution (what we will do) or forward substitution
2. **Master Theorem (AKA Master Method)** - A 3 case solution to a form of recurrence relation. A more general but more annoying to use method is the Akra-Bazzi theorem.

There are relations that cannot be solved using the master theorem (e.g. the Hanoi recurrence relation).

9.4 Iterative Method (Back Substitution)

The iterative substitution is done in typically 1 of 2 ways. The way these notes will cover it that of the backward substitution. The backward substitution starts at an arbitrary point in the recurrence relation and expresses the value in terms of smaller input values. The method consists of several steps,

1. Express the function (in this case $H(N)$) in terms of decreasing input sizes, until...
2. A general form is identified.
3. Find the terminating iteration of the general form
4. Plug in the terminating values into the general form and solve

For the towers of Hanoi example we would do the following,

$$H(N) = 1 + 2H(N-1) \text{ [Step 1: Back sub; write the original equation]}$$
$$H(N-1) = 1 + 2H(N-1-1) \text{ [plug } N-1 \text{ into original equation]}$$
$$H(N) = 1 + 2(1 + 2H(N-2)) \text{ [substitute } H(N-1) \text{ with result]}$$
$$H(N) = 1 + 2 + 4H(N-2) \text{ [Simplify]}$$
$$H(N-2) = 1 + 2H(N-2-1) \text{ [plug } N-2 \text{ into original equation]}$$
$$H(N) = 1 + 2 + 4(1 + H(N-3)) \text{ [substitute } H(N-2) \text{ with result]}$$
$$H(N) = 1 + 2 + 4 + 8H(N-3) \text{ [Simplify]}$$

$$H(N) = \sum_{i=0}^{k-1} 2^i + 2^k H(N-k) \text{ [Step 2: General Form (after } k \text{ steps) Acquired!!!]}$$

$$N - k = 1 \text{ [Step 3: look for terminating iteration]}$$
$$N = k + 1 \text{ [Some math]}$$
$$N - 1 = k \text{ [Found terminating iteration]}$$

$$H(N) = \sum_{i=0}^{N-1-1} 2^i + 2^{N-1} \text{ [Step 4: Plug in terminating values (removing k)]}$$
$$H(N) = \sum_{i=0}^{N-2} 2^i + 2^{N-1} \text{ [Reduction]}$$
$$H(N) = \frac{2^{N-2+1}-1}{2-1} + 2^{N-1} \text{ [Geometric closed form; now closed]}$$
$$H(N) = \frac{2^{N-1}-1}{1} + 2^{N-1} \text{ [Reduction]}$$
$$H(N) = 2(2^{N-1}) - 1 \text{ [Reduction]}$$
$$H(N) = 2^N - 1 \text{ [Reduction]}$$

Thus the order of the towers of Hanoi puzzle is exponential. The back substitution requires a fair bit of algebra work and observation skills.

9.4.1 Building Recurrence Relations

Recurrence relations can be used to model the number of simple operations taken by a recursive function. By using recurrence relation reduction methods such as back substitution, computer scientists can determine the closed form representation of the number of operations. With a closed form expression of the number of operations, the runtime of a recursive function can be compared to it's iterative counter part.

There are important factors when creating the recurrence relation that models number of operations,

1. Which variable(s) affect your number of recursive calls. Recursive call counts can be controlled by factors such as the length of the array, a position in the array of a particular size, or the number of nodes in the data structure.

2. The change in variables passed to the recursive call. A recursive function could cut the input variable by a constant factor, or by a fixed value.

3. The number of times the function the function is called recursively from a single call. The number of times may be dependent on the value of a variable (as with permutations).

4. The number of operations happening within a single function call. In many cases this depends on the variables passed to the function.

5. A what variable value does the function enter the base case.

On the next page is a recursive function in C that will be analyzed,

```
int fun(int N, int * arr)
{
   if (N == 1)
   {
      return 0;
   }
   int ans = fun(N / 2, arr);
   for (int i = 0; i < N - 1; i++)
   {
      ans += (arr[i] < arr[i+1]);
   }
   return ans;
}
```

The value of N is will determine how many recursive calls happen, because when N is 1 the function stops recursing.

$$T(N) = ???$$

$$T(1) = 1$$

The variable size is cut in half with each recursive call, because the the value of N that is passed into the recursive `fun` call is $N/2$,

$$T(N) = ??? \cdot T(N/2) + ???$$

$$T(1) = 1$$

The function is recursively called a single time with each function call; there is only one `fun` call in `fun`, and that call is not in a loop,

$$T(N) = 1 \cdot T(N/2) + ???$$

$$T(1) = 1$$

The number of operations per recursive call is O(N), because of the for loop goes to N-1 and steps by 1 each time,

$$T(N) = 1 \cdot T(N/2) + N$$

$$T(1) = 1$$

9.4.2 Practice Problems

Question 9. For the following recursive function determine the recurrence relation expressing a tight order approximation of the number of operations, AND solve for the closed form of the recurrence relations and express the closed form in an order approximation.

```
int fun2 (int N, int * arr)
{
   if (N <= 1)
   {
      int res = 0;
      for (int i = 0; i < N; i++)
      {
         res += arr[i];
      }
      return res;
   }
   int ans = fun2(N / 2, arr);
   ans += fun2(N - (N / 2), arr + (N / 2));
   return ans;
}
```

Answer

Question 10. For the following recursive function determine the recurrence relation expressing a tight order approximation of the number of operations, AND solve for the closed form of the recurrence relations and express the closed form in an order approximation.

```
int fun2 (int N, int * arr)
{
   int res = 0;
   for (int i = 0; i < N; i++)
   {
      res += arr[i];
   }
   if (N <= 1)
   {
      return res;
   }
   res = fun2(N / 2, arr);
   res += fun2(N - (N / 2), arr + (N / 2));
   return res ;
}
```

Answer

9.5 Best Worst and Average

Sometimes an algorithm can vary in time depending on the order or values of data entered (and not just the quantity). There are 3 situations that computer scientists focus on,

1. Best case
2. Average case
3. Worst case

9.5.1 Best Case

The best case is NOT when the input is 1. The best case is what values could give a good (the best) performance. Consider looking up values in an unsorted array. The best case is that the value you are looking for is in the first index. That does take $\Theta(1)$ regardless of how many elements are in the list.

If one were to look up L (potentially different) values in an unsorted list of V values, the best case if all L were the first. The best case runtime is $\Theta(L)$.

9.5.2 Average Case

The average case is possible when every possible outcome is weighted by its likelihood. For looking up words in an unsorted list the chance that a target word is at any given position is $\frac{1}{W}$, where W is the number of words. The number of comparisons the method takes given the word is in the first index is 1, the number of comparisons the method takes given the word is in the second index is 2, etc. The following summation is acquired when weighing each option by its probability,

$$\frac{1}{W} + \frac{2}{W} + \frac{3}{W} + \frac{4}{W} + ... + \frac{W}{W}$$

Expressed as a summation,

$$\sum_{i=1}^{W} \frac{i}{W}$$

By factoring out the constant we get,

$$\frac{1}{W}\sum_{i=1}^{W} i$$

Using the summation closed forms discussed earlier the sum can be reduced to

$$\frac{1}{W} \cdot \frac{W(W+1)}{2}$$

By some properties of fractions the result is

$$\frac{(W+1)}{2}$$

which means the value will be roughly in the middle of the list on average.

In terms of order approximations, $\Theta(W)$ is how many operations a linear search will take on average.

9.5.3 Worst Case

The worst case is when the input will be given such that the runtime behaves in the worst possible manner. Suppose a random search was executed. A random index is selected, and if the value is contained at the index the program exits. The worst case is the random number generator picks values that never finds the correct value. In this case the program would not halt. It can be shown that the average case for such an algorithm is roughly the length of the list, but this requires a recurrence relation, which has not been presented yet. The worst case for a linear search would be the length of the array.

9.5.4 Examples

Question 11. What would be the best and worst case for performing a string compare on two strings each containing N characters?

Answer

9.6 Amortization

Amortization is useful, when although the worst case runtime might be high for a single part of the programs execution, the total number of operations is low. It's a technique used for proving behavior of min queues, KMP, z-values, and array lists.

9.6.1 Array List Append

The array list amortizes nicely when considering the behavior incurred by doubling the capacity for resizing. Assume an array list is initialized with 1 capacity. The number of operations performed when not resizing the list is relatively constant. Resize requires a linear number of operations to move the contents to another section of memory. Consider the following example of how many simple operations that will be incurred based on the order of the inserted values,

Insertion Order	Resize Operations	Other Operations
1st	0	c
2nd	c	c
3rd	$2c$	c
4th	0	c
5th	$4c$	c
6th	0	c
7th	0	c
8th	0	c
9th	$8c$	c
10th	0	c

The sum of the second column is never more than twice the number of inserted elements times 2. In fact the sum is $2S-1$, where S is the number of resizes, and the number of resizes is around $\log_2$ of the number of inserted elements, since it takes twice as many insertions each time to resize the array.

The total number of operations to insert the first N values will be at most $2Nc + Nc$. The result is $\Theta(N)$. This means that although a single insertion could take $\Theta(N)$ operations over the course of the full execution, the worst case per insertion will be around $\Theta(1)$ on average over the course of the full use of the data structure.

9.6.2 Practice Problem

Question 12. What is the Order Approximation of the runtime of the following segment of code?

```
int i = 0, j = 0;
while (i < N)
{
   while (j < N)
   {
      ans+=i*j;
      j++;
   }
   i++;
}
```

Answer

9.7 Practical Applications

A good reason for using this order analysis is that runtimes can be estimated for the sake of determining how long a program needs to be run to solve some problems. Many problems that occur on the foundation exams are something akin to "Given a program that takes O($log(N)$) time takes 3 seconds to halt on N = 1,000 values, how long will the same program take on N = 1,000,000 values?"

9.7.1 Runtime Estimation

The example problem can be solved by assuming that the number of operations per second is constant and that the given order bound is what determines the total runtime. The following formula is really important,

$$Rate \cdot Operations = Time$$

In the example above we will end up with

$$R \cdot Clog(N) = 3s$$
$$R \cdot Clog(1000) = 3s$$

Sadly, R and C cannot be individually found, because there is not enough information, but their product can be found.

$$R \cdot C = \frac{3s}{log(1000)}$$
$$R \cdot C = \frac{3s}{log(10^3)}$$
$$R \cdot C = \frac{3s}{log(10^3)}$$

Now when solving the time for the bigger N the following formula is found,

$$R \cdot C \cdot log(100000) = Time$$
$$\frac{3s}{3log(10)} \cdot log(100000) = Time$$
$$\frac{3s}{3log(10)} \cdot 6log(10) = Time$$
$$3s \cdot 2 = Time$$
$$6s = Time$$

9.7.2 Input Size Estimation

The problem can be modified to use runtimes to compute other runtimes. Here is a sample problem "Given a program that takes O(N^2) time takes 10 seconds to halt on N = 1,000,000 values, how many values can be processed in 0.1 seconds?" Again we can use the rate formula to find the product of R and C.

$$Rate \cdot Operations = Time$$
$$R \cdot C(N^2) = Time$$
$$R \cdot C(1000000^2) = 10s$$
$$RC = \frac{10s}{10^{12}}$$

Then the original formula is used a second time, with the new values used as a substitution,

$$Rate \cdot Operations = Time$$
$$R \cdot C(N^2) = Time$$
$$\frac{10s}{10^{12}}(N^2) = 0.1s$$
$$N^2 = (.1s)\frac{10^{12}}{10s}$$
$$N^2 = 10^{10}$$
$$N = 10^5$$

9.8 Multiple Variables

Expressions with multiple variables can exist with multiple variables. The variables should normally be assumed to have no correlation. For example, suppose a program takes $5N$ + $3\log_2(M)$ operations. The resulting order is $\Theta(N + \log(M))$. It might be the case that N is relatively constant and M grows really fast as we scale up input. In such a situation $\log(M)$ would dominate, but it could be that the two values grow at the same rate and the term N would dominate in that situation. It cannot determine the behavior based on just the number of operations. The maximum of the 2 terms will dominate, and the sum of those 2 will always be at least as large as the maximum.

Similarly it cannot be said that NM is in $\Theta(N^2)$ nor is NM in $\Theta(M^2)$.

9.8.1 Practice Problems

Question 13. What is the runtime of the following segment of code?

```
for (int i = 1; i < N && i < M; i++)
{
   ans++;
}
```

Answer

Question 14. What is the runtime of the following segment of code?

```
for (int i = 1; i < N || i < M; i++)
{
   ans++;
}
```

Answer

10 Sorting

An array is sorted if the values are in their comparison order. From the discrete math point of view there is a relation on the elements of the array that is,

- Reflexive ($a \leq a$)
- Transitive (if $a \leq b$ and $b \leq c$, then $a \leq c$)
- Strongly Connected (at least $a \leq b$ or $b \leq a$)

Note that because equivalent elements could exist, the Antisymmetric property does not need to hold. Additionally, the elements of the array are in an order such that for any pair of indices `i` and `j` such that `i` $\leq$ `j`, then `array[i]` $\leq$ `array[j]`.

A sorted array of integers for this reason is defined to be in ascending order. There are a few ways to sort an array. This section of notes will discuss a few algorithms and analyze them in terms of best, average, and worst case for determining which sorting algorithm should be considered when implementation is required.

10.1 Stable Sorts

Before diving straight into the sorting algorithms it should be noted that sometimes a sorting method is required to be **stable**. A stable sort is an algorithm that forces elements that are equivalent under the given comparison to maintain the relative order from the given array in the resulting array.

To force an unstable sorting algorithm to be stable, an additional value can be stored in the original data structure that represents the original index, and ties of the original data type can be broken by comparing against the original index of the two elements in question.

10.2 Selection

The **selection sort** is a common first sorting algorithm to learn. The selection sort works by first finding the element that will come first in the array, and placing it in the first spot. After the first element is "removed" from the original array and "placed" in the resulting array, the next smallest element is found and placed in the next available spot. This process is furthermore repeated for each next smallest value, until the original array is empty.

The algorithm can use the original array as the resulting array by swapping the values into a **prefix** (a contiguous subarray starting at the beginning) of the original array. Below is a sample of how the algorithm would work on a possible input array.

Initial	4	2	1	5	3
First Pass	1	2	4	5	3
Second Pass	1	2	4	5	3
Third Pass	1	2	3	5	4
Fourth Pass	1	2	3	4	5

After the (`N`-1)-th pass the array will be sorted, where `N` represents the original array size.

It can be thought of as the picture day sort. When lining up students in order of height for a picture, we can look down the line and see the tallest person and remove them. It is easy for a person to compare everyone's heights for the tallest student, but it is harder for a computer to do the same.

10.2.1 Runtime

In code the program needs to pass through the entire array and keep track of the best value. Once the program has reached the end, then the program will be certain the smallest value is known. The pass on the first iteration takes `N`-1 comparisons. The subsequent passes will each take 1 less comparison (e.g. `N`-2, then `N`-3, `N`-4, etc.).

Thus the number of comparisons for the selection sort is (N-1) + (N-2) + (N-3) + $\cdots$ + 1, which becomes N(N-1)/2 or $\Theta(N^2)$. The selection sort takes the same number of comparisons regardless of the initial order of the array (unlike all the other sorting algorithms discussed in this set of notes).

10.2.2 Implementation

Below is an example of the pseudo code,

```
Selection(array, N):
  For i in the range of [0, N-1]:
     Find the index of the smallest value in the range of [i, N-1]
     Swap(index of the smallest value, i)
```

10.3 Insertion

The next sorting algorithm discussed here (insertion sort) is a little more efficient. The algorithm also builds the sorted array as a prefix. How the insertion sort differs is by which values are placed into the prefix. Rather than finding the value that will be in the first spot after the array is completely sorted, the insertion sort will use a sorted prefix of the values that happened to be first in the array.

The insertion sort will "insert" the values from the original array one by one to the end of the prefix. When the value is inserted into the prefix it might be in the wrong location. To fix this the insertion sort will swap the element with value that comes before it repeatedly until the value that comes before the newly inserted value is ordered correctly with respect to the new value.

For example if 5 was inserted at the end of the array 2, 4, 6, and 8, then 5 would swap twice (once with 8 and once with 6) such that the resulting array would be 2, 4, 5, 6, and 8. Below is an example of the insertion sort algorithm would work on one possible input array,

Initial	4	2	1	5	3
First Pass	2	4	1	5	3
Second Pass	1	2	4	5	3
Third Pass	1	2	4	5	3
Fourth Pass	1	2	3	4	5

Note that the first pass does not need to include just the first value since a single value is already sorted in the initial array.

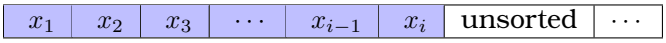
The algorithm is "better" than the selection sort in the best case. The best case number of comparisons needed for the insertion sort is N-1 (Θ(N)). The best case happens when the initial array is sorted. Only 1 comparison is needed for each value after the first to realize the value is already in the correct location.

The algorithm has the worst performance when the array is in the reverse sorted order. The worst case behavior is because with each insertion the inserted value will be compared against every value in the prefix. In other words the value in the second location of the initial array will do 1 comparison (against the first value). The value in the third location of the initial array will do 2 comparisons (against the first and second value). The value in the fourth location of the initial array will do 3 comparisons, and so on. The value in the last spot will perform N-1 comparisons. The total number of comparisons will be N(N-1)/2, which is still $\Theta(N^2)$.

10.3.1 Average Case

The algorithm's runtime on average is not that great either. To consider the average the linearity of expectations can be used. That is the sum of the expected number of comparisons needed for each value in the initial array will be the expected sum. The expected number of

comparisons for values after the first index is based on which index the value could go into. Consider the sorted prefix of length `i`,



A newly inserted value—x_{i+1}—could reside in `i`+1 possible spots. Either after any one of the initial `i` values or at the beginning of the prefix. Each spot is equally likely. The spot after x_i requires 1 comparison. The spot after x_{i-1} requires 2 comparisons. In general the spot after x_j requires $i-(j-1)$ comparisons. The front of the array requires i comparisons. For this reason the number of comparisons required on average is the total—$i+(i(i+1)/2)$—divided by the number of possibilities—$i+1$. To make the math work a little better an upper bound of $i+1+(i(i+1)/2)$ divided by $i+1$ and a lower bound of $(i(i+1)/2)$ divided by $i+1$ will both be considered.

Upper Bound The upper bound on the expected number of comparisons for the (`i`+1)-th term will therefore be $(i+1+(i(i+1)/2))/(i+1)$. By factoring out $(i+1)$ in the numerator the following fraction can be reached,

$$\frac{(i+1)(1+i/2)}{(i+1)}$$

By simplification the expression becomes,

$$(1+i/2)$$

Which if the sum of which is taken from $i=1$ to N, the following can logic can be found,

$$\sum_{i=1}^{N} 1+\frac{i}{2} = \sum_{i=1}^{N} 1 + \sum_{i=1}^{N} \frac{i}{2}$$

$$= N + \frac{1}{2}\sum_{i=1}^{N} i$$

$$= N + \frac{1}{2}\cdot\frac{N(N+1)}{2}$$

$$= \frac{1}{4}\cdot N^2 + \frac{5}{4}\cdot N$$

$$\in \mathrm{O}(N^2)$$

Alternatively, since the worst case is $\Theta(\mathtt{N}^2)$ and thus $\mathrm{O}(\mathtt{N}^2)$ and the worst case should be an upper bound of the average case, we can argue that the average case will also be $\mathrm{O}(\mathtt{N}^2)$.

Lower Bound The lower bound on the expected number of comparisons for the (`i`+1)-th term will be $(i(i+1)/2)$ over $(i+1)$, or $i/2$, comparisons. The sum of those terms from i+1=2 to i+1=N is therefore 1/2 + 2/2 + $\cdots$ + (N-1)/2. The sum is $(N)(N-1)/4$, which is still $\Omega(\mathtt{N}^2)$.

Since the upper and lower bound are both on the order of $\mathtt{N}^2$, the average is also $\Theta(\mathtt{N}^2)$.

10.3.2 Implementation

Below is an example of the pseudo code of the insertion sort algorithm,

```
Insertion(array, N):
  For i in the range of [1, N-1]:
     Store the current index as i
     While the value at current is smaller than the value before:
        Swap the values of those at current and the one before
        Move the current location to the spot before
```

10.4 Bubble

The third (and final iterative) sort discussed in these notes is the bubble sort algorithm. The bubble sort acquired its name due to how the bubbles in a glass of liquid grow in size as they move towards the top. The bubble sort moves through the array from the front to the back and the largest value seen in the array's prefix is kept at the current index of the sweep during the pass. To keep the largest value at the current index a swap between the largest value and the proceeding value is used if the next value is smaller than the largest. Below is an example of a single pass of the bubble sort algorithm.

Initial	4	2	1	5	3
After First Compare	2	4	1	5	3
After Second Compare	2	1	4	5	3
After Third Compare	2	1	4	5	3
After Fourth Compare	2	1	4	3	5

Multiple passes are potentially required to sort the array. Note that when the first pass of the bubble sort is finished the largest value will be in the correct location. That element will never move again. This means that the value can be ignored for the sake of the algorithm. Effectively the size of the array is reduced by 1. The next pass will therefore move the second largest value into the correct location. Every pass will move 1 more value to the correct location. The maximum number of passes required is N-1, where N is the number of elements in the array. If the passes are done efficiently, then the total number of comparisons in the worst case would be (N-1)+(N-2)+...+1 for a total of N(N-1)/2, which is $\Theta(N^2)$.

Below is an example of running the Bubble sort on some initial array

Initial	4	2	1	5	3
After First Pass	2	1	4	3	5
After Second Pass	1	2	3	4	5
After Third Pass	1	2	3	4	5
After Fourth Pass	1	2	3	4	5

The algorithm could keep track of the number of swaps done in a pass, and if the pass completes without performing a swap, then the algorithm can terminate before performing all N-1 passes. In the best case a single pass could determine that the array is sorted and the algorithm can terminate. The total number of comparison in this case would take N-1 or $\Theta(N)$.

To analyze the average case we can examine where the smallest value is located in the original array. The smallest value can be in N possible locations. The number of passes required to move the smallest value is based on the index the smallest value starts. Every pass will move the smallest value only 1 index close to the beginning. If each pass takes an average of (N-1)/2 comparisons. The average of all the comparisons for moving the first value to the front would be

$$(1/N)(1(N-1)/2 + 2(N-1)/2 + 3(N-1)/2 + \cdots + (N-1)(N-1)/2)$$

Totals is

$$(1/N)((N-1)(N)/2)((N-1)/2)$$

With a little reduction we get

$$((N-1)/2)((N-1)/2)$$

which is $\Theta(N^2)$.

10.5 Quick

The quicksort is the first recursive sorting algorithm covered in this set of notes. The quicksort works by reducing the array into two smaller pieces and recursively sorting each piece. The array is split by choosing some value in the array to become a pivot. The elements less than the pivot are moved into one side, and elements greater than the pivot are moved into the other. When an array is of some trivial size (e.g. 1 or less) the array can be returned without swapping any values.

The pseudo code looks like the following

```
Quicksort(array)
   If the array is small enough return the array
   Choose the pivot of the array
   Smaller = elements in array < pivot
   Larger = elements in array > pivot
   Quicksort(Smaller)
   Quicksort(Larger)
   Return Smaller + pivot + Larger
```

10.5.1 Quicksort Best Case

The best case for the quicksort is when the selected pivot is the median of the array (the middle in terms of value). The number of comparisons to split the array will be N-1, where N represents the number of values in the array. The runtime can be expressed as a recurrence relation.

$$Q(N) = Q\left(\frac{(N-1)}{2}\right) + Q\left(\frac{(N-1)}{2}\right) + N - 1$$
$$Q(1) = 1$$

We will use a slightly different form (shown below). Using back substitution the closed form can be found.

$$Q(N) = 2Q\left(\frac{N}{2}\right) + N$$

Find $Q(N/2)$

$$Q\left(\frac{N}{2}\right) = 2Q\left(\frac{N}{4}\right) + \frac{N}{2}$$

Substitute

$$Q(N) = 2\left(2Q\left(\frac{N}{4}\right) + \frac{N}{2}\right) + N$$
$$= 4Q\left(\frac{N}{4}\right) + N + N$$

Find $Q(N/4)$

$$Q\left(\frac{N}{4}\right) = 2Q\left(\frac{N}{8}\right) + \frac{N}{4}$$

Substitute

$$Q(N) = 4\left(2Q\left(\frac{N}{8}\right) + \frac{N}{4}\right) + N + N$$
$$= 8Q\left(\frac{N}{8}\right) + N + N + N$$

Find the general form (at step k),

$$Q(N) = 2^k Q(\frac{N}{2^k}) + kN$$

The recursion stops when $N/2^k = 1$; or $N = 2^k$; or $\log_2(N) = k$.
Thus

$$\begin{aligned} Q(N) &= NQ(1) + log_2(N)N \\ &= N1 + Nlog_2(N) \end{aligned}$$

The above is $\Theta(\mathtt{N}\log(\mathtt{N}))$.

10.5.2 Quicksort Worst Case

The worst case pivot is one of the extremes (either the minimum or the maximum). The pivot will force all the remaining N-1 values into one side (i.e. one recursive call). The number of comparisons is still N-1. The runtime can be expressed as the following recurrence relation.

$$\begin{aligned} Q(N) &= Q(N-1) + N - 1 \\ Q(1) &= 1 \end{aligned}$$

Solving the closed form using back substitution looks like the following,

$$Q(N) = Q(N-1) + N - 1$$

Find $Q(N-1)$,

$$\begin{aligned} Q(N-1) &= Q(N-1-1) + N - 1 \\ &= Q(N-2) + N - 1 \end{aligned}$$

Substitute into the original recurrence relation,

$$Q(N) = Q(N-2) + N - 1 + N$$

Find $Q(N-2)$,

$$\begin{aligned} Q(N-2) &= Q(N-2-1) + N - 2 \\ &= Q(N-3) + N - 2 \end{aligned}$$

Substitute into the recurrence relation,

$$Q(N) = Q(N-3) + N - 2 + N - 1 + N$$

The general form looks like the following.

$$\begin{aligned} Q(N) &= Q(N-k) + \sum_{i=0}^{k-1} N - i \\ &= Q(N-k) + \sum_{i=0} k - 1N - \sum_{i=0} k - 1i \\ &= Q(N-k) + kN - k(k-1)/2 \end{aligned}$$

The terminating conditions are the following

$$N - k = 1; N = k + 1; N - 1 = k$$

Plugging in the conditions gives the following closed form

$$\begin{aligned} Q(N) &= Q(1) + (N-1)N - (N-1)(N-1-1)/2 \\ &= 1 + N^2 - N - \frac{N^2}{2} + \frac{3N}{2} - 2 \\ &= \frac{N^2}{2} + \frac{N}{2} - 1 \end{aligned}$$

The term is in $\Theta(\mathtt{N}^2)$.

10.5.3 Quicksort Average Case

The quicksort's average case is notably harder to analyze. The pivot could be any value in the array. An array of size N has N possible locations where the pivot could be located in the sorted array. Each pivot is equally likely (i.e. 1/N chance for each). The number of operations for the recursive calls can be expressed in the table below.

Pivot Index	Left Recursion	Right Recursion
1	Q(0)	Q(N-1)
2	Q(1)	Q(N-2)
3	Q(2)	Q(N-3)
⋮	⋮	⋮
N-1	Q(N-2)	Q(1)
N	Q(N-1)	Q(0)

The recurrence relation can be written as the following,

$$Q(N) = \frac{1}{N}(Q(0) + Q(N-1)) + 1/N(Q(1) + Q(N-2)) + \cdots + 1/N(Q(N-1) + Q(0)) + N - 1$$

The recursive calls can be combined,

$$Q(N) = \frac{1}{N}(Q(0) + Q(N-1) + Q(1) + Q(N-2) + \cdots + Q(N-1) + Q(0)) + N - 1$$

Reordered,

$$Q(N) = \frac{1}{N}(Q(0) + Q(0) + Q(1) + Q(1) + \cdots + Q(N-1) + Q(N-1)) + N - 1$$

Combined again,

$$Q(N) = \frac{2}{N}(Q(0) + Q(1) + \cdots + Q(N-1)) + N - 1$$

To eliminate the long summation a trick is used where we can plug in a slightly smaller value than N into the function. First the function must get the summation without a scale. To fix the scale issue both sides are multiplied by $N/2$.

$$\frac{N}{2} \cdot Q(N) = (Q(0) + Q(1) + \cdots + Q(N-1)) + (N-1) \cdot \frac{N}{2} \tag{1}$$

Plugging in $N - 1$,

$$\frac{(N-1)}{2} \cdot Q(N-1) = (Q(0) + Q(1) + \cdots + Q(N-1-1)) + (N-1-1) \cdot \frac{(N-1)}{2} \tag{2}$$

Taking the differences of both sides of equations (1) and (2) we get the following,

$$\frac{N}{2} \cdot Q(N) - \frac{(N-1)}{2} \cdot Q(N-1) = Q(N-1) + (N-1) \cdot \frac{N}{2} - (N-1-1) \cdot \frac{(N-1)}{2}$$

By moving the non-$Q(N)$ terms to the other side we get the following equation,

$$\frac{N}{2} \cdot Q(N) = Q(N-1) + \frac{(N-1)}{2} \cdot Q(N-1) + (N-1) \cdot \frac{N}{2} - (N-2) \cdot \frac{(N-1)}{2}$$

$$\frac{N}{2} \cdot Q(N) = \frac{(N+1)}{2} \cdot Q(N-1) + (N-1) \cdot N2 - (N-2)\frac{(N-1)}{2}$$

$$\frac{N}{2} Q(N) = \frac{(N+1)}{2} \cdot Q(N-1) + \frac{(N-1) \cdot 2}{2}$$

$$\frac{N}{2} Q(N) = \frac{(N+1)}{2} \cdot Q(N-1) + (N-1)$$

Multiply both sides by $2/N$ to express the equation in terms of $Q(N)$ again.

$$Q(N) = \frac{(N+1)}{N} \cdot Q(N-1) + 2 - \frac{2}{N}$$

The $2 - 2/N$ term can be treated as a constant (in the range of 1 to 2).

$$Q(N) = \frac{(N+1)}{N} \cdot Q(N-1) + c$$

Now the recurrence relation is solved using iterative substitution.

$$Q(N) = \frac{(N+1)}{N} Q(N-1) + c$$

Find $Q(N-1)$,

$$Q(N-1) = \frac{(N-1+1)}{(N-1)} Q(N-1-1) + c$$

$$= \frac{(N)}{(N-1)} Q(N-2) + c$$

Substitute,

$$Q(N) = \frac{(N+1)}{N} \cdot \left(\frac{(N)}{(N-1)} \cdot Q(N-2) + c \right) + c$$

$$= \frac{(N+1)}{(N-1)} \cdot Q(N-2) + \frac{(N+1)}{N} \cdot c + c$$

Find $Q(N-2)$,

$$Q(N-2) = \frac{(N-2+1)}{(N-2)} \cdot Q(N-2-1) + c$$

$$= \frac{(N-1)}{(N-2)} \cdot Q(N-3) + c$$

Substitute,

$$Q(N) = \frac{(N+1)}{(N-1)} \left(\frac{(N-1)}{(N-2)} Q(N-3) + c \right) + \frac{(N+1)}{N} \cdot c + c$$

$$= \frac{(N+1)}{(N-2)} \cdot Q(N-3) + \frac{(N+1)}{(N-1)} \cdot c + \frac{(N+1)}{N} \cdot c + c$$

OR,

$$= \frac{(N+1)}{(N-2)} \cdot Q(N-3) + \frac{(N+1)}{(N-1)} \cdot c + \frac{(N+1)}{N} \cdot c + \frac{(N+1)}{(N+1)} c$$

Find the general form,

$$Q(N) = \frac{(N+1)}{(N-(k-1))} \cdot Q(N-k) + \sum_{i=0}^{k-1} \frac{(N+1)}{(N-(i-1))} \cdot c$$

Find the terminating conditions,

$$N - k = 1; N = k + 1; N - 1 = k$$

Substitute in the terminating conditions,

$$Q(N) = \frac{(N+1)}{(N-(N-1-1))} \cdot Q(1) + \sum_{i=0}^{N-1-1} \frac{(N+1)}{(N-(i-1))} \cdot c$$

$$Q(N) = \frac{(N+1)}{(2)} \cdot 1 + c \sum_{i=0}^{N-2} \frac{(N+1)}{(N-(i-1))}$$

Changing the sum variable—i—to some other variable—j. The goal is to convert the summation into something like the harmonic series. The harmonic series can be found more easily found if $N-(i-1)=j$. Thus $N-j=i-1$. By rewriting the expression in terms of i, the equation $N+1-j=i$ can be found. When $i=0$ then $j=N+1$, and when $i=N-2$ then $j=3$.

$$Q(N)=\frac{(N+1)}{(2)}\cdot 1+c\sum_{j=3}^{N+1}\frac{(N+1)}{j}$$

To complete the harmonic series some terms will be added,

$$Q(N)=\frac{(N+1)}{2}+c\cdot\left(\left(\sum_{j=3}^{N+1}\frac{(N+1)}{j}\right)+\frac{(N+1)}{2}+\frac{(N+1)}{1}-\left(\frac{(N+1)}{2}+\frac{(N+1)}{1}\right)\right)$$

$$=\frac{(N+1)}{2}+c\cdot\left(\sum_{j=1}N+1\frac{(N+1)}{j}\right)-\left(\frac{(N+1)}{2}+\frac{(N+1)}{1}\right)$$

$$=c\cdot(N+1)\cdot\left(\sum_{j=1}^{N+1}\frac{1}{j}\right)-(N+1)$$

The summation (and the $c(N+1)$ scale) is in $\Theta(\mathtt{N}\log(\mathtt{N}))$, and the term subtracted is non-dominant, so the result is $\Theta(\mathtt{N}\log(\mathtt{N}))$. The average case runtime is much better than the average case for the other iterative sorting algorithms, but the worst case leaves a lot to be desired.

10.6 Merge

The final sorting algorithm discussed in this document. The method uses recursion like the quicksort but splits the array exactly in half each recursive call. A pivot is not chosen to split the array; instead the array is split in half without rearranging the values first. These two halves are sorted recursively using this new sorting method. After the two half arrays are sorted they need to be combined. The two halves are merged by comparing the first values of each half. The smallest value of the array has to be the first value of one of the two halves. That value can be removed from the corresponding array, and the merge procedure can be repeated for the next smallest value and so on until one of the halves empties. When one of the halves is empty, the remaining values of the other half can be appended to the end of the resulting array

The pseudo code can be seen below.

```
Mergesort(array)
  If the array is small enough return the original array
  Mergesort(first half of array)
  Mergesort(second half of array)
  Merge the two halves of the array
```

10.6.1 Merge Analysis

The merge operation takes N-1 comparisons in the worst case and N/2 comparisons in the best case for an array with N values. Either number can be treated as a constant times N. The runtime can be expressed as the following recurrence relation

$$M(N)=2M(N/2)+cN$$

The recurrence relation is the same form as that of the best case for quick sort and can be solved in the same manner. The resulting runtime becomes $\Theta(\mathtt{N}\log(\mathtt{N}))$ in all three cases.

10.7 Runtime Recap

Here is a table with the runtimes for the discussed algorithms in terms of the array size–N,

Algorithm	Best Case	Average Case	Worst Case
Selection	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^2)$
Insertion	$\Theta(N)$	$\Theta(N^2)$	$\Theta(N^2)$
Bubble	$\Theta(N)$	$\Theta(N^2)$	$\Theta(N^2)$
Quick	$\Theta(N\log(N))$	$\Theta(N\log(N))$	$\Theta(N^2)$
Merge	$\Theta(N\log(N))$	$\Theta(N\log(N))$	$\Theta(N\log(N))$

11 Graphs Primer

The linked list was a linked data structure where accessing random elements was slow; accessing a particular node in a linked list requires traversing the full list from the head to the node of interest. In the section a formalization of a new data structure will be presented that can enable quicker arbitrary access by allowing for traversing different parts of the full data structure from each node. The described structure will create a more general linked data structure known as the **graph**.

11.1 General Terminology

A graph is a discrete data model composed of a set of Nodes and a set of relations on the Nodes called Edges. A **Node** is some piece of data. An **Edge** is a relation (connection) between (typically) 2 Nodes, such as the link in the original linked list. The nodes that are connected by an edge are sometimes referred to as the **Endpoints** of the edge. An edge is **Incident** to a node if the node is an endpoint of the edge. Figure 19 is an example of a graph with 5 nodes (labeled 1 through 5), and 7 edges. The endpoints of the bolded edge are 1 and 2.

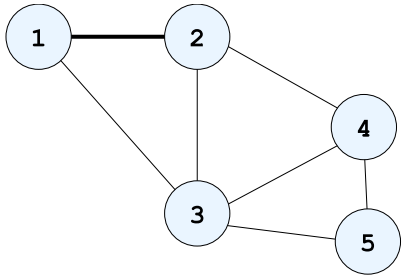


Figure 19: Picture example of a graph.

Sometimes edges will have directions. The origin node of the edge will be referred to as the **Start** in these notes, and the terminating node of the edge will be referred to as the **End**. Edges have their direction noted in pictures by using arrows. Figure 20 is an example of a graph with a single directed edge, where the start is the node labeled 2 and the end is labeled 1.

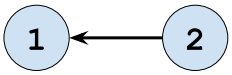


Figure 20: Picture example of a graph with a directed edge.

A **Path** in a graph is a series of distinct edges such that the edges can be directed in a manner, such that adjacent pairs of edges in the Path share an endpoint (the start of the later edge is the same node as the end of the earlier edge). Across all edges in the path any node can occur at most twice (either the between adjacent edge pairs) or the first edge and the last edge may share the same node. When the start of the first edge is the same as the end of the last edge the Path is known as a **Cycle**.

The graph in Figure 21 there is no path from Node 1 to Node 5, but there is a path from Node 1 to Node 2 (edge (1,2)), and there are 2 paths from Node 3 to Node 5 (e.g. (3,4) + (4,5); (3,5)). There is no cycle containing Node 1, but there is a cycle containing Node 3 ((3, 5) + (5, 4) + (4, 3)).

A graph is **Connected** when for all pairs of distinct Nodes there exists a path between them (the start of the first edge is one of the nodes and the end of the last end is the other node). The graph in Figure 19 is connected, but the graph in Figure 21 is disconnected.

A graph with no cycles is called **Acyclic**. The graph in Figure 20 is acyclic, but the graph in Figure 19 and 21 are not acyclic.

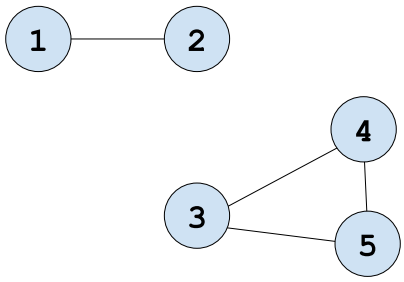


Figure 21: Picture example of a graph with a cycle.

11.2 Tree Terminology

A Tree is a special graph. A **Tree** is a graph that is connected and acyclic. Figure 22 is an example of a tree. The first graph pictures in this set of notes (Figure 19) was not a tree,

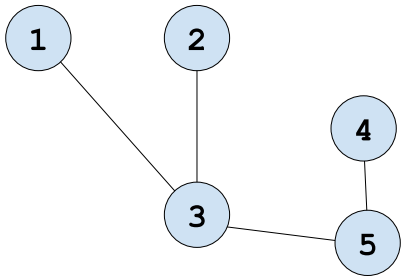


Figure 22: Picture example of a tree.

The number of edges and Nodes forms a very special relationship. If there is at least 1 node in a tree, the number of nodes (N) is equal to the number of edges (E) plus 1. In other words,

$$N = E + 1$$

The relationship can be proved using induction, but will not be covered in this set of notes. Here are a few more properties of trees,

- Between all pairs of distinct nodes there exists a unique path.
- If any edge is removed, the graph becomes disconnected (not connected).
- If any edge is added a cycle will be formed.

Nodes in a tree that are incident to only one edge are called **Leaves** (singular **Leaf**). In the graph pictured in Figure 22 example there are 3 leaves (Nodes 1, 2, and 4).

11.3 Rooted Tree

A special type of Tree exists that could be implemented as a data structure using linked memory. The special tree is called the Rooted Tree. **Rooted Trees** are trees where all edges are directed and there is a special node. The edges should be directed such that all paths from the special node have their edges point away from the special node. The special node is a sort of source of the tree. Each node can be reached using a path from the special node. The special node is therefore commonly referred to as the **Root**. Any node that has no edges pointing out of them are called **Leaves**.

Figure 23 has a picture of graph that is an example of a rooted tree where the root of the tree is Node 5. In that same figure the leaves are labeled 1, 2, and 4.

Figure 24 has a picture of a tree with edges directed in a way that makes it not a rooted tree, because no node can be the root.

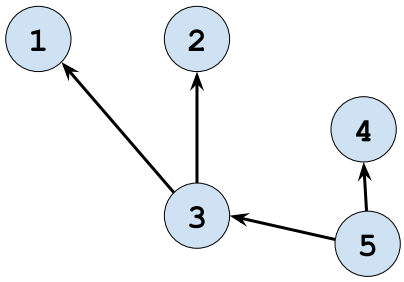


Figure 23: Picture example of a rooted tree, where Node 5 is the root.

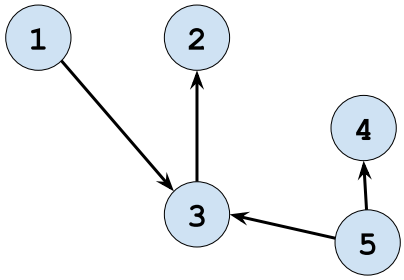


Figure 24: Picture example of a tree with no root.

11.4 Familial Terminology

Rooted trees tend to represent family trees (in some sense) for this reason family relationship terminologies apply to the nodes.

Consider an edge in the rooted tree. The start (node) of the edge is a **Parent** of the end (node) of the edge, and the end of the edge is a **Child** of the start of the edge. A **Grandparent** is a parent of a parent, and a **Grandchild** is a child of a child. A **Sibling** is a child of a parent that is not the original node. An **Uncle** is a sibling of a parent. A **Nephew** is a child of a sibling.

In Figure 23 the following relations can be observed,

- Node 3 is a child of Node 5
- Node 3 is a parent of Node 2
- Node 3 is a sibling of Node 4
- Node 1 is a grandchild of Node 5
- Node 5 is a grandparent of Node 2
- Node 1 is a nephew of Node 4
- Node 4 is an uncle of Node 2

An **Ancestor** of a node is any node along the path from the root to the given node. A **Descendant** is any node who has the given node as an ancestor. Nodes are descendants and ancestors of themselves, which is weird, but useful in advanced graph applications. The set of descendants of a node form a **Subtree**—a subset of nodes and edges that form a tree. The result of descendants form trees means that there is a recursive structure to rooted trees, which means that many functions on rooted trees can be implemented recursively!

A Tree is **binary** if every node has at most 2 children. If a tree can have an arbitrary number of children the tree is known as an **N-ary rooted tree**.

11.5 Traversals

Suppose we need to visit all nodes in a rooted tree. From the root all nodes can be reached. To visit all nodes we can use the following recursive code.

```
Visit(Root):
  For each Child of Root:
    Visit(Child)
```

The act of visiting all nodes in a tree is known as Traversing. Traversing trees can be used for several purposes such as,

- Cloning a tree
- Deleting a tree
- Printing the values of a tree
- Finding a value in a tree
- Inserting a child of a node in a tree
- Finding a particular node in a tree

Suppose we wanted to check if a value existed in our tree. We could check if we are the value and if not we could check if the value exists in any of our children's subtrees. If we find it great, but if the element is not in any children's subtree, the function would return that the value is not in the original given tree. We could use the following pseudocode.

```
Contains(Root, Value):
  If Value of the Root is Value:
    Return Contained
  For each Child of Root:
    If Contains(Child, Value) is Contained:
      Return Contained
  Return Not Contained
```

11.5.1 Traversals

There are 2 main types of traversal for N-ary rooted trees,

1. Preorder
2. Postorder

11.5.2 Preorder Traversals

The example "contains" code above is known as a preorder order traversal. The nodes are processed before processing the children. In the contains case function above the nodes are processed by checking their value. Figure 25 has a picture of a rooted tree. If a preorder traversal was performed on the tree, where the left most children are handled first, then the order in which the nodes are processed are,

```
1, 5, 7, 3, 6, 2, 3, 4
```

Below is an example of what the preorder traversal pseudocode could look like.

```
Preorder(Root):
  Process(Root)
  For Child in Children of Root:
    Preorder(Child)
```

Preorder traversals are useful for when a tree needs to be copied.

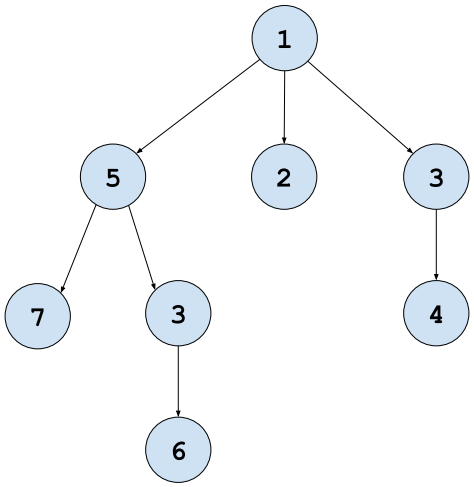


Figure 25: Picture example of a rooted tree with children going dowards.

11.5.3 Postorder Traversal

When a nodes are processed after processing the children, the traversal is known as a postorder. Perform a postorder traversal on the tree from Figure 25 has a picture of a rooted tree, where the left most children are handled first. If done properly the nodes will be processed in the following order,

```
7, 6, 3, 5, 2, 4, 3, 1
```

Below is an example of what the preorder traversal pseudocode could look like.

```
Postorder(Root):
   For Child in Children of Root:
      Postorder(Child)
   Process(Root)
```

Postorder traversals are useful for when a tree needs to be deleted.

11.6 C Implementation

Since these trees will be implemented in assignments in C, the remainder of this section will focus on implementations in C.

11.6.1 N-Ary Rooted Tree

N-ary rooted trees have an arbitrary number of children and can be implemented using an array list or linked list of nodes to hold onto the children, below is an example of a possible implementation of the node struct,

```c
typedef struct Node Node;
struct Node
{
   int value; // Data of the node
   Node ** children; // Array of pointers to nodes
   int size; // Number of children
   int cap; // Length of the children array
};
```

Below is an example of how a child could be added to a parent,

```c
// Function to add a child to a parents children array
void add(Node * par, Node * child)
{
   // Check if the children array is full
   if (par->cap == par->size)
   {
      // Expand
      par->cap *= 2;
      par->children = (Node **) realloc(par->children, par->cap * sizeof(Node *));
   }

   // Add to the array of children
   par->children[par->size++] = child;
}
```

Below is an example of how a full tree could be deleted using a post order traversal

```c
// Function to delete a root and all it's decendents
void delTree(Node * root)
{
   // Empty tree case
   if (root == NULL)
   {
      return;
   }

   // Delete children first
   for (int i = 0; i < root->size; i++)
   {
      delTree(root->children[i]);
   }

   // Delete the root after all children's trees are deleted
   free(root->children);
   free(root);
}
```

11.6.2 Binary Rooted Tree

A binary tree can avoid using a list structure because the number of children are limited to 2. Instead explicit pointers to the children can be made. Since Trees are typically drawn with the root at the top and the edges pointing in the downwards direction, the children end up on the left or right side of the node. Due to the direction of how the children are drawn the children are commonly labeled as either left or right. Below is an example of how the Node structure for a binary tree could look.

```c
typedef struct Node Node;
struct Node
{
   int value; // Data
   Node * left; // Holder of left child address
   Node * right; // Holder of right child address
};
```

When the child is not present—as with a leaf of the tree—the respective child pointer(s) of the node will typically point to `NULL`. On the next page is an example of a function that that creates a node dynamically and initializes the values.

```c
Node * createNode(int value)
{
   Node * res; // Instantiate

   res = (Node *) malloc(sizeof(Node)); // Allocate

   res->value = value; // Initialize
   res->right = NULL;
   res->left = NULL;

   return res; // Return
}
```

Binary Tree Question 1. Suppose a binary tree has K levels, what is the least and most number of nodes possible?
Answer

Inorder Traversal Binary trees have a third type of traversal—**inorder**—because there is the ability to process the node between the traversals of the 2 children. Below is a C code example of an inorder traversal of a binary tree.

```c
void inorder(Node * root)
{
   if (root != NULL)
   {
      inorder(root->left);
      process(root);
      inorder(root->right);
   }
}
```

11.7 Honorable Mentions

There are many things that can be done with graphs, but will not be covered in this class. If you want to get acquainted with more advanced graph concepts, please look into the following topics,

- Graph Structures
  - Edges List
  - Adjacency Matrix
  - Adjacency List
- Search Algorithms
  - Depth First Search
  - Breadth First Search
  - 0-1 BFS
- Topological Sorting
  - Khan's Algorithm
- Shortest Distance Algorithms
  - Floyd-Warshall's
  - Bellman Ford
  - Dijkstra's
- Component Algorithms
  - Kosaraju's
  - Tarjan's Low Link
- Bipartite Matching
- Hungarian Algorithm
- Flow
  - Ford Fulkerson
  - Edmond Karp
  - Dinitz

12 Binary Search Trees

Suppose a problem requires the ability to quickly look up a value in a large container of values (e.g. dictionary). A binary search could allow for fast look up assuming the values are sorted. If there is also the ability for the container to change over time then the lookup might be trickier. If values are added to the dictionary, then sorting the values after every add may be required to ensure that the binary search can find the result. There are methods that lazily do not sort the value after each insertion, but instead wait until the number of added values are at about the square root of the original size of the array. Such methods would linear search these unsorted values in addition to binary searching the sorted values. Square roots are frequently used in algorithm development.

However, better methods that take logarithmic time exists. A tree structure that can help efficiently solve this problem is the **Binary Search Tree** (BST). The raw binary search tree works well on average, but the worst case has some room for improvement.

12.1 Searchable

Without structure finding an element in a Binary Tree would required a traversal. Traversing a full tree looking for a value is as bad as performing a linear search. The key fast lookups in a BST is to ensure that the tree is searchable. The Binary Tree is used as the base structure. When looking for a particular node[10] the BST should "know" which direction[11] down the tree to go. The precognition is not magical it is acquired by ensuring that nodes with values less than a node's value are stored in one a fixed child's subtree, and nodes with values greater than a node's value are stored in the other child's subtree. Typically, the left subtree has nodes whose values are less than the root's value, and the right subtree has nodes whose values are greater than the root's value.

The above mentioned constraint is an invariant which means that modifications such as insertion and removals in the BST will have to adhere to maintaining this structure. Values cannot be inserted randomly into the tree. A deterministic process—that is a process whose behavior is discernible based on the structure and type of modification—is employed. An additional change is that when removal occurs knowing the parent of a node will be important. Because parent information is important, the parent pointer is typically stored in the node `struct` of the BST.

12.2 Insertion

Insertion will follow the following set of rules,

- If the tree is empty,

  the tree becomes a node with the value to insert

- If the value at the root of the tree is equal to the value inserted,

  the behavior will depend on the purpose of the tree[12]

- If the value at the root of the tree is less than the value inserted,

  The value is inserted into the subtree of the root's right child

- If the value at the root of the tree is greater than the value inserted,

  The value is inserted into the subtree of the root's left child

[10] by value; not address

[11] Left or right

[12] The insertion could fail; the root could have a value within it modified; an insertion could happen on either child's subtree.

Binary Search Tree Question 1. Try inserting the following values into an empty BST and write draw the structure of the final tree.

$$7,\ 8,\ 5,\ 9,\ 6,\ 2,\ 1,\ 4,\ 10$$

Answer

Below is an example of a BST insertion in C,

```c
typedef struct Node Node;
struct Node
{
   int data;
   Node * left;
   Node * right;
   Node * parent;
};
Node * createNode(int data)
{
   Node * res; // Instantiate pointer

   res = (Node *) malloc(sizeof(Node)); // Allocate

   res->data = data; // Initialize data
   res->left = NULL; // Initialize pointers
   res->right = NULL;
   res->parent = NULL;

   return res; // Return
}

Node * insertNode(Node * root, int data)
{
   if (root == NULL) // Empty tree case
   {
      return createNode(data);
   }

   if (root->data < data) // data is larger than root
   {                  // Go right
      root->right = insertNode(root->right, data);

      // Connect the child to the parent
      root->right->parent = root;
   }

   if (root->data > data) // data is smaller than root
   {                  // Go left
      root->left = insertNode(root->left, data);

      // Connect the child to the parent
      root->left->parent = root;
   }
}
```

12.2.1 Analysis

The order approximation for runtime for insertion into a BST—as described above—with N nodes can be determined for the best, average, and worst case. The tree structure for the best and worst cases are "similar" to each other.

The **worst case** is when every level has a single node, and insertion occurs on the last level. To reach the last level every level before it needs to have a node compared in the tree.

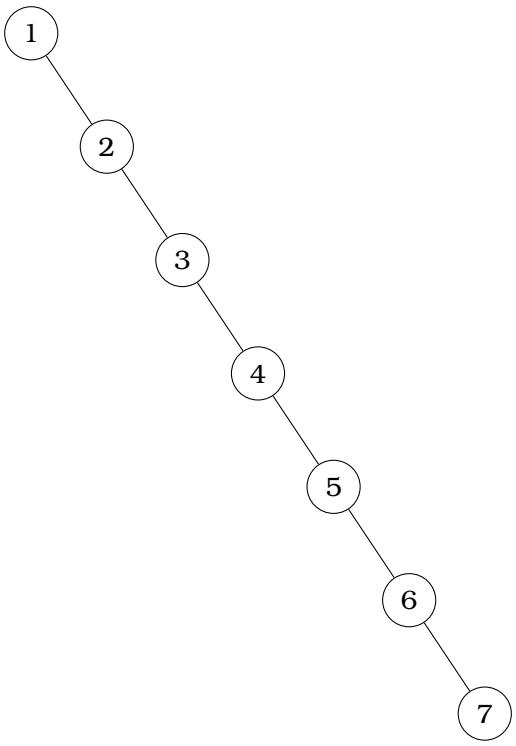


Figure 26: Bad BST structure.

The combination of these situations would result in comparing every node in the tree to the value inserted. The runtime is therefore $\Theta(N)$. An example tree structure that causes the worst case performance can be seen in Figure 26. Inserting the values greater than 7[13], would cause the worst case performance.

The **best case** for insertion occurs when there is an empty spot high up the tree[14]. In the exact same tree structure as worst case, Figure 26, we have the best case. Inserting values less than 1, e.g. 0, result in the best case insertion. The value of N does not impact the number of operations taken. The resulting runtime is $\Theta(1)$.

The **average case** is a lot better than the worst case, but not as good as the best case. To do the analysis a recurrence relation can be formed. It can be argued that the value inserted can uniformly exist between any pair of values in terms of their sorted order or after or before the last and first element respectively. For example if the values in the BST were the following,

1, 5, 8, 10, 14, 20

Then we would try to argue that a value less than 1 would be as likely as a value between 5 and 8, which is as likely as a value between 8 and 10. For the above example that doesn't seem to hold true, since more values are between 5 and 8 than between 8 and 10, but on average this would be the case, since we are just as likely to have had 8 inserted in the tree as opposed to 7.

With this in mind, it can be argued that the recurrence relation will have to consider N + 1 possible locations that our node could go. However, the root of the tree determines which direction the node will move. All N possible nodes could be the root with an equal probability. The root itself determines the possible sizes of the next tree, and where the node goes determine which subtree is used. With all of this combined the average case number of comparisons becomes the following recurrence relation,

[13] or values between 6 and 7
[14] near the root

$$T(N) = 1 + \frac{1}{N}\sum_{i=1}^{N} \frac{i}{N+1}T(i-1) + \frac{N+1-i}{N+1}T(N-i)$$

$$T(0) = 0$$

The $\frac{1}{N}$ on the outside of the summation is because each value can be the root. The fraction on the inside of the summation is the probability that the inserted value will be on either side of the root. The summation can be expanded to make analysis easier.

$$T(N) = 1 + \frac{1}{N}\left(\frac{1T(0)}{N+1} + \frac{NT(N-1)}{N+1} + \frac{2T(1)}{N+1} + \frac{(N-1)T(N-2)}{N+1} + \cdots + \frac{NT(N-1)}{N+1} + \frac{1T(0)}{N+1}\right)$$

By reordering the values in the summation the following equation can be achieved,

$$T(N) = 1 + \frac{1}{N}\left(2\frac{1T(0)}{N+1} + 2\frac{2T(1)}{N+1} + \cdots + 2\frac{NT(N-1)}{N+1}\right)$$

Pull the common factor out of the summation for the following equation,

$$T(N) = 1 + \frac{2}{N(N+1)}\Big(1T(0) + 2T(1) + \cdots + NT(N-1)\Big)$$

Multiply both sides by $N(N+1)$,

$$N(N+1)T(N) = N^2 + N + 2\Big(1T(0) + 2T(1) + \cdots + NT(N-1)\Big)$$

Substitute $N-1$ for N,

$$(N-1)(N)T(N-1) = N^2 - N + 2\Big(1T(0) + 2T(1) + \cdots + (N-1)T(N-2)\Big)$$

Take the difference of both sides for the 2 equations above,

$$N(N+1)T(N) - (N-1)NT(N-1) = 2N + 2\Big((N)T(N-1)\Big)$$

Eliminate a factor of N,

$$(N+1)T(N) - (N-1)T(N-1) = 2 + 2\Big(T(N-1)\Big)$$

Bring the $T(N-1)$ term to the same side,

$$(N+1)T(N) = 1 + 2T(N-1) + (N-1)T(N-1)$$

Reduce,

$$(N+1)T(N) = 1 + (N+1)T(N-1)$$

Divide,

$$T(N) = \frac{1}{N+1} + T(N-1)$$

Iteratively substitute until we find that,

$$T(N) = \left(\sum_{i=1}^{N} \frac{1}{i+1}\right) + T(0)$$

$$T(N) = \sum_{i=1}^{N} \frac{1}{i+1}$$

The above sum is a harmonic series. The series is in $\Theta(log(N))$. The finding of this result implies that the average depth of a `NULL` node is logarithmic with respect to the number of nodes in the tree.

12.3 Containment

One common problem that comes up in applications of computer science is to determine if a value is **contained** in a collection. In an unsorted array a linear search would be necessary, and as mentioned earlier in this section a sorted array although having the ability to find values using a binary search would require resorting if values were incrementally added to the collections. Luckily, due to their structure BSTs can determine containment efficiently on average. Below is a structure of how containment can be determined in a BST structure in C,

```c
// Return 1 if a value is contained in the tree
// Return 0 otherwise
int contains(Node * root, int data)
{
   if (root == NULL) // Empty Tree (not contained)
   {
      return 0;
   }

   if (root->data == data) // Root has the value
   {
      return 1;
   }

   if (root->data < data) // Bigger than root
   {
      return contains(root->right, data);
   }

   // Have to be smaller than root because all returns
   return contains(root->left, data);
}
```

12.3.1 Analysis

The analysis for contains is similar to the analysis for insertion. The main difference is that containment can additionally stop early if the value looked for is the root. Due to the similarities between contains and insertion, the runtimes for the 2 functions are identical in each case.

12.4 Removal

A final function that BSTs typically implement is that of **removal by value**. The remove function is the trickiest of the basic functions of the BST. The BST is a linked structure. Linked lists were another linked structure, but removal by value from a linked list is much easier, because the replacement node in all cases is obvious. In a linked list the replacement node is the next node of the removed node.

If the removed node from the BST has 0 or 1 child the replacement would be either NULL or the child respectively. For nodes that have 2 children there are 2 seemingly possible replacements. However, the replacements that can be considered are not necessarily the children. The reason is that those children might also have 2 children themselves, which means their sibling could be lost if used as the replacement. The node typically used as the replacement needs to have at most 1 child. A node with at most 1 child can be found in either child's subtree. The nodes that can be used are called either,

- predecessor - largest value node in left child's subtree; found by going as far right as possible in left child's subtree

- successor - smallest value node in right subtree; found by boind as far left as possible in right child's subtree

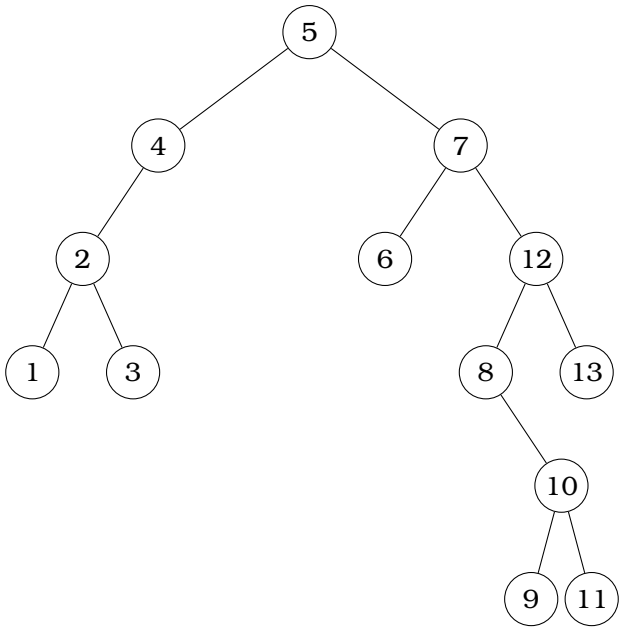
This means that the predecessor or successor needs to be removed from it's subtree to become the replacement. Removing the node from its respective subtree is doable, because it should have at most 1 child, so the removal is simple. Below is removal from a BST,

```
Node * bstRemove(Node * root, int data)
{
   if (root == NULL) // Value was not in tree??
   {
      return root;
   }

   if (root->data == data) // remove root
   {
      if (!root->right && !root->left) // 0 children case
      {
         free(root);
         return NULL;
      }

      if (!root->right) // 1 child: left child exists
      {
         Node * newRoot = root->left;
         newRoot->parent = root->parent;
         free(root);
         return newRoot;
      }

      if (!root->left) // 1 child: right child exists
      {
         Node * newRoot = root->right;
         newRoot->parent = root->parent;
         free(root);
         return newRoot;
      }

      Node * newRoot = successor(root); // 2 child case :(

      int tmp = root->data; // swap values
      noot->data = newRoot->data;
      newRoot->data = tmp;

      root->right = bstRemove(root->right, data); // Recursively remove

      return root; // Return root
   }

   if (root->data > data) // smaller than root: go left
   {
      root->left = bstRemove(root->left, data);
   }
   else // bigger than root: go right
   {
      root->right = bstRemove(root->right, data);
   }
}
```

12.4.1 Analysis

Remove requires effectively finding a `NULL` node to help with replacing the node. Because a `NULL` node needs to be found, the runtimes are identical to that of BST insertion.

Binary Search Tree Question 2. What does the following tree look like after removing nodes 1, 5, and 8 in that order? When necessary use the successor.



Answer

12.5 Analysis Summary

In summary we have the following runtimes for the 3 main BST functions for the data structures discussed in this section,

BST	Best	Average	Worst
Insert	$\Theta(1)$	$\Theta(log(N))$	$\Theta(N)$
Contains	$\Theta(1)$	$\Theta(log(N))$	$\Theta(N)$
Remove	$\Theta(1)$	$\Theta(log(N))$	$\Theta(N)$

Sorted Array	Best	Average	Worst
Insert	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$
Contains	$\Theta(1)$	$\Theta(log(N))$	$\Theta(log(N))$
Remove	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$

Unsorted Array	Best	Average	Worst
Insert	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Contains	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$
Remove	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$

Sorted Array SQRT Buffer	Best	Average	Worst
Insert	$\Theta(1)$	$\Theta(\sqrt{N})$	$\Theta(N)$; Amortized $\Theta(\sqrt{N})$
Contains (in)	$\Theta(1)$	$\Theta(log(N))$	$\Theta(\sqrt{N})$
Contains (not in)	$\Theta(\sqrt{N})$	$\Theta(\sqrt{N})$	$\Theta(\sqrt{N})$
Remove[15]	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$

[15] can be improved by marking values as non existent

Binary Search Tree Question 3. What would be the best and worst case runtime approximation for inserting N nodes into an initially empty BST? Answer

12.5.1 Inorder Traversal

The inorder traversal of a BST has a useful behavior. The nodes are processed by the order of the value in an inorder traversal. The smallest valued node will be processed first, and the largest valued node will be processed last.

13 AVL Trees

As we have noted in the analysis of the BST section the worst case runtime for the BST is expensive. When the tree forms something that resembles a linked list—e.g. Figure 27—the runtimes for insertion, removal, and containment can all take Linear time.

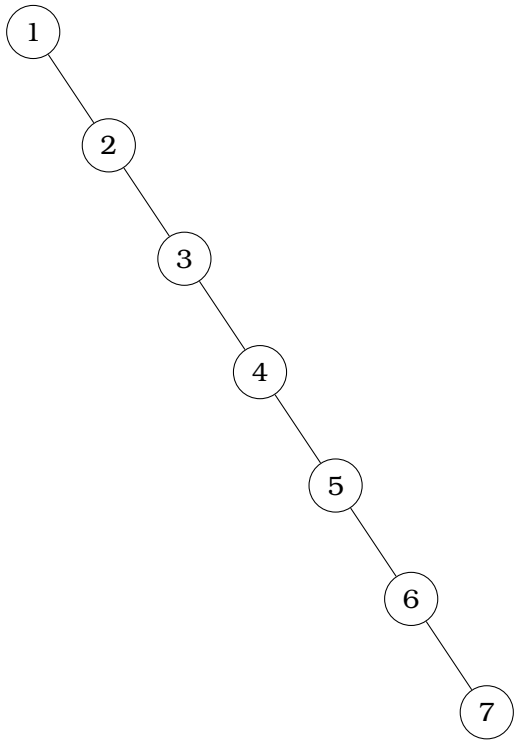


Figure 27: Bad BST structure presented originally in the BST section of the notes.

To generate the tree in Figure 27, the values need to be inserted in order. The problem is due to deep nodes, which cause many comparisons.

When performing a binary search on an array of values. There is no deep number of comparisons needed. The search can always split the search area in half each time. In a BST when a deep path exists there are nodes along that path the do not evenly split the search space in half. In fact many nodes along a long path in an tree with near linear runtimes split very close to the minimum or maximum value in the range.

The solution to making the tree have better worst case runtimes is to improve the balance of the structure. It cannot be the case the the root is always exactly the middle. Trying to maintain that is extremely costly with every manipulation. Most "balanced" search trees try to limit the imbalance at each node by creating additional invariants.

Adelson-Velsky and Landis created a search tree that aims to keep depths between nodes as close as feasible to prevent imbalance. There a few more terms that need to be addressed first,

- The **depth** of a node is the edge distance[16] from the root to that particular node.
- The **deepest leaf** is leaf in a node's subtree with the largest depth.
- The **height** of a node is the difference between a node's deepest leaf's depth and the node's depth[17].

[16] I prefer using node distance instead

[17] The height of a `NULL` node is -1. sometimes the height is defined to be the node distance between a node and it's deepest leaf; in such cases the height of `NULL` is 0

- The **balance factor** of a node is the difference between the height of the node's left and right child[18].

An **AVL Tree** is a BST, where the balance factor of every node in the tree has a magnitude no more than 1. In other words the only valid balance factors of an AVL tree is -1, 0, or 1.

Consider the tree in Figure 28.

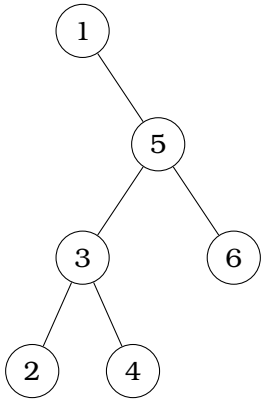


Figure 28: A Tree for the demonstration of balance factors.

The depth, height, and balance factors of the nodes are the following,

Node	1	2	3	4	5	6
Depth	0	3	2	3	1	2
Height	3	0	1	0	2	0
Balance Factor	-3	0	0	0	1	0

The tree is Figure 28 is not an AVL tree, because the node with value 1 has an invalid balance factor. Converting an arbitrary BST into an AVL tree is very difficult. Instead AVL trees should be maintained from insertion of the first node. To ensure trees after every tree modification—insertion and removal—previous function will have to be changed to incorporate both consideration of the existing balance of the tree and allow for node rebalances. Rebalances happen through small node relationship changes known as **rotations**.

13.1 Node Rotations

Rotations are modification that will maintain a inorder traversal behavior but modify the tree structure. Maintaining the inorder traversal is important, since the search property is only possible because the inorder traversal process the nodes in their comparison order. Any BST structure containing integer values with a standard comparison will inorder traverse the nodes in increasing order. The inorder traversal property follows from nodes with a smaller values always appearing in a node's left subtree and nodes with a greater value always appearing in a node's right subtree. The property holds recursively down each node's child's respective subtrees.

Rotations "swap" the relationship between a node and one of it's child. Since there are 2 possible children, there are 2 possible rotations. Nodes can be rotated left or right.

- The left rotation - moves a node down and towards the left. The right child of the rotated node replaces the node. If a right child does not exist, the rotation cannot (and should not) occur.
- The right rotation - moves a node down and towards the right. The left child of the rotated node replaces the node. If a left child does not exist, the rotation cannot (and should not) occur.

[18] `NULL` node's don't have a balance factor, but if `NULL` nodes did, the height would be 0

When the child (C) of the rotated node (R) replaces the R, R needs to be referred to by a node that is not its original parent (P). In a rotation the new parent of the R becomes the C. The structure goes from P references R, which references C to P reference C, which references R. If C originally had 2 children one of them becomes orphaned. Luckily R just lost a child so, a reference opens up for the orphaned child of C.

Figure 29 show an example of a right rotation on Node R

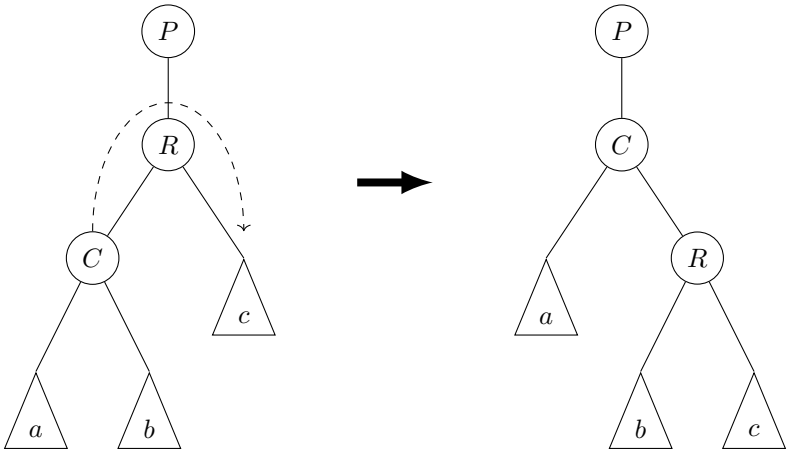


Figure 29: Demonstration of a right rotation on node R. C—the left child of R—starts with subtrees a and b as its children. R has a right subtree of c and a parent of P.

The left rotation is the mirror of the right rotation. Figure 30 shows a picture example of a left rotation,

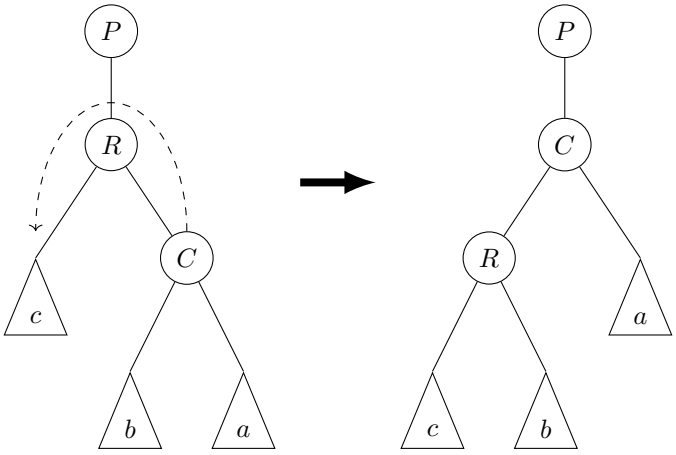


Figure 30: Demonstration of a left rotation on a node R.

In both of the rotations shown in Figure 29 and 30 the subtree b contains values that are between R and C, because in Figure 29 the values in b are greater than C and less than R, and in Figure 30 the values in b are greater than R and less than C. In both of the resulting trees the values of b end up between those 2 nodes in terms of their inorder traversal.

13.2 AVL Imbalance Cases

In AVL trees imbalances are fixed from the bottom of the tree up. Due to the bottom up nature of the checks, imbalanced nodes will not have children that are imbalanced. The rotations described above can be used to move a heavy (deep) children of imbalanced nodes up the tree. AVL trees work by breaking down an imbalanced node into 1 of 4 cases based on where the deepest grandchild lies,

1. Left-Left - The imbalanced node has a Balance Factor of 2, and the left child has a balance factor ≥ 0
2. Left-Right - The imbalanced node has a Balance Factor of 2, and the left child has a balance factor of -1
3. Right-Left - The imbalanced node has a Balance Factor of -2, and the left child has a balance factor of 1
4. Right-Right - The imbalanced node has a Balance Factor of -2, and the left child has a balance factor ≤ 0

13.2.1 Left-Left (Right-Right)

The cases that are easiest are the Left-Left and Right-Right cases. For the Left-Left case the tree is heavy on the left side. A right rotation will move the left (heavy) side up the tree. Figure 31 shows an example of a Left-Left case and the resulting tree after the right rotation that fixes it. After the rotation 3 is no longer imbalanced either, so the tree becomes a valid AVL tree.

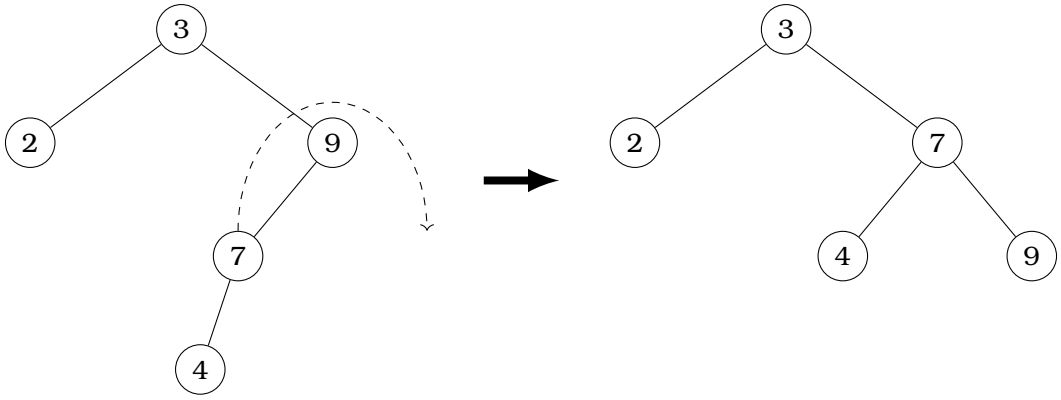


Figure 31: Demonstration of a Left-Left imbalance on 9, which is fixed using a single rotation to the right on 9.

In the original tree of Figure 31 the nodes 9 and 3 are both imbalanced at first. However, balance correction starts from the bottom. By working from the bottom up the first node encountered is 9. The grandchild with the deepest height is 4, additionally 9 has a balance factor of 2, and the left child of 9 has a balance factor of 1. For these reasons 9 is in a Left-Left imbalance. A single rotation to the right is all that is needed to rebalance 9.

In all Left-Left and Right-Right cases a single rotation away from the heavy side will correct the tree structure. Figure 32 shows a more general example of how Left-Left cases occur. The respective height of each subtree is labeled beneath it. The balance factor the the child and the rotated node are listed above. The resulting balance factor of the originally unbalanced node becomes 0 or 1, and the balance factor for the resulting root of the imbalanced node's subtree becomes 0. In summary the balance factors are fixed for the subtree. The downside is that the height of the full tree (originally $n+3$) has potentially shrunk (possibly $n+2$).

Since the Left-Left is a mirror of the Right-Right case a Left rotation on the imbalanced node in a Right-Right case will have the same effects as the Right rotation on the imbalanced node in a Left-Left case.

13.2.2 Left-Right (Right-Left)

The Left-Right case is a bit trickier. Consider the tree in Figure 33. The rotation to the right on the imbalanced node which can move the heavier child up the tree makes second imbalance that is a mirror the first. Trying to use the left rotation on the resulting imbalanced

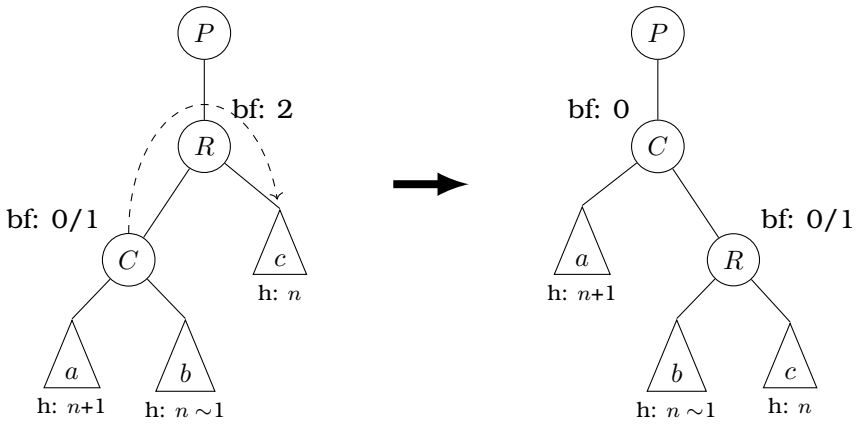


Figure 32: The figure demonstrates the 2 possible Left-Left cases. Where the $n \sim 1$ height of the b subtree means that the height could be n or $n + 1$.

node—node with value 4—would result the original tree. A single rotation in such cases will not suffice.

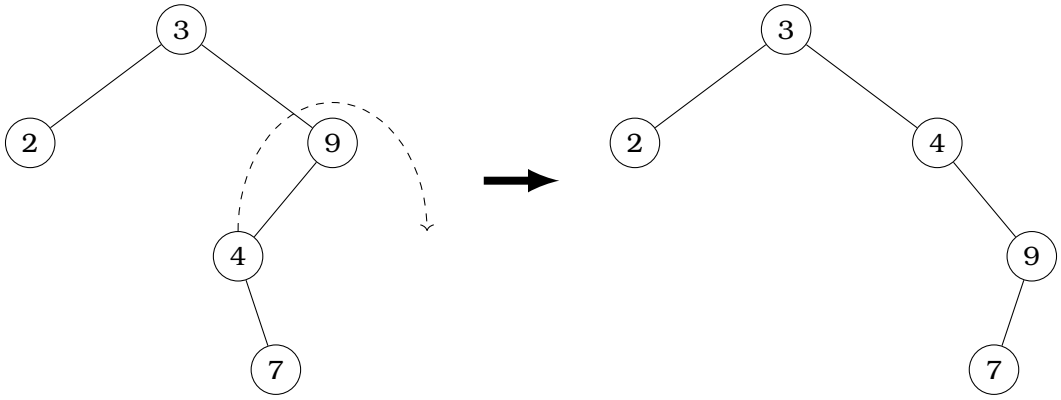


Figure 33: Demonstration of a Left-Right imbalance on 9, which is NOT fixed by a rotation to the right on 9.

The solution to solving this case is to reduce a Left-Right case into a Left-Left case by first performing a rotation on the child of the imbalanced node. A left rotation on the child of a Left-Right imbalanced node will convert the imbalance into a Left-Left case, see Figure 34.

The Right-Left case is a mirror of the Left-Right, so the mirror of the rotations on the mirrored nodes will fix the tree.

13.3 AVL Insertion

When inserting a node into an AVL tree the standard BST insertion can be used. However, nodes could have their balance factors modified and some of those updated balance factors could be invalid. The only nodes that could have their balance factors modified are the ones along the path from the root of the tree to the node that was inserted. Any node off the path will not have the inserted node in their subtree. As mentioned earlier imbalanced node checks are performed from the bottom of the tree towards the root. The combination of these two pieces of information means that from the inserted node to the root imbalances are checked for and mitigated.

Any rebalance performed on node during insertion will fix the remainder of the tree, because the height of the deepest child will be reduced by one regardless of the case. Any insertion will require at most 1 rebalance.

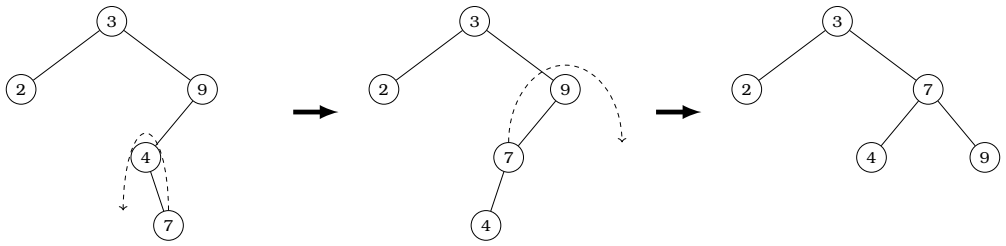


Figure 34: Demonstration of a Left-Right imbalance on 9, which is fixed by first a rotation on 9's left child to the left, followed by a rotation on 9 to the right.

The balance factor for nodes can be updated without storing the height of each node. As the insertion is propagating changes of balance factors up the tree there are 4 cases to consider

1. The child on the side of insertion was the inserted node.
2. The child on the side of insertion went from a non-zero balance factor to a zero balance factor.
3. The child on the side of insertion went from a zero balance factor to a non-zero balance factor.
4. The child on the side of insertion did not change its balance factor.

For case 1 and 2 the node must move the balance factor in the direction of the insertion. For case 3 and 4 no modifications to the balance factor is needed.

13.4 AVL Removal

When removing a node from an AVL tree the standard BST removal can be used. Again nodes could have balance factors modified and the resulting tree could become non-AVL. The nodes with possible modifications to their balance factors will be those along the path from the removed node to the root. Rebalances are performed from the bottom of the tree to the top. The resulting rebalance and reduce the height of the tree by 1 in some cases. Due to the possible reduction in height the path up the tree could still be imbalanced. More rebalances may be necessary. The number of possible rebalances is linear with respect to the depth of the node removed.

Like with insertion the balance factor for nodes can be updated without storing the height of each node. As the removal is propagating changes of balance factors up the tree there are 4 cases to consider

1. The child on the side of removal is NULL.
2. The child on the side of removal went from a non-zero balance factor to a zero balance factor.
3. The child on the side of removal went from a zero balance factor to a non-zero balance factor.
4. The child on the side of removal did not change its balance factor.

For case 1 and 3 the node must move the balance factor away from the side of removal. For case 2 and 4 no modifications to the balance factor is needed.

13.5 Analysis

Analysis of the AVL involves focusing on the worst case for the various functions. The worst case can be determined by the deepest node in the tree. Determining the most deep node for a particular number of nodes that could exists is difficult. Solving the inverse of the problem—finding the least number of nodes for a particular depth—is easier. Figure 35 shows examples of trees with the least number of nodes for the possible root heights.

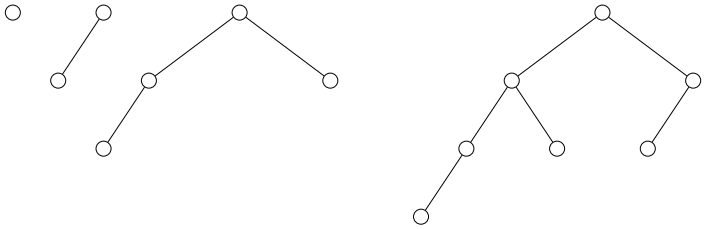


Figure 35: Example AVL structures with minimal number of nodes.

The next AVL tree is possible by adding the previous 2 trees together joined by a single node. The relationship forms the following recurrence relations defined by the height of the tree (H),

$$T(H) = T(H-1) + T(H-2) + 1$$

$$T(0) = 1$$

$$T(1) = 2$$

The relationship appears similar to the Fibonacci relationship. In fact it can be shown that,

$$T(x) = F_{x+3} - 1$$

The closed form of the Fibonacci is in $\Theta(\phi^n)$, where ϕ is the golden ratio. Thus $T(n) \in \Theta(\phi^n)$. By using the logarithm of both sides it can be shown that the deepest node (in the worst case) is at a depth of $\Theta(log(n))$, where n is the number of nodes. Thus the worst case runtimes for the basic BST functions are $\Theta(log(N))$.

The draw back is that finding the shallowest `NULL` which is required for best case of the remove and insertion is a lot deeper in the tree. It can be formally proved that that `NULL` will also be $\Theta(log(N))$ deep in the tree. Due to this constraint the best case runtimes for insertion and removal are worse than those in the normal BST. Below is the table of runtimes

AVL	Best	Average	Worst
Insert	$\Theta(log(N))$	$\Theta(log(N))$	$\Theta(log(N))$
Contains	$\Theta(1)$	$\Theta(log(N))$	$\Theta(log(N))$
Remove	$\Theta(log(N))$	$\Theta(log(N))$	$\Theta(log(N))$

13.6 Honorable Mentions

There are other types of balanced search trees. Here a few of Dr. Meade's favorites,

- Splay Trees
- Red Black Trees
- Treaps
- 2-4 Trees

14 Heaps

Some times when queuing elements rather than taking the element that has been in the queue the longest, the programmer may want to remove the most important element. These important elements have a higher **priority** than the other elements in the collection. An example would be that of a hospital triage. Each patient would have the severity of their condition assessed upon arrival, and the patient with the highest level of urgency should be seen over that of other patients that may have arrived earlier. Removing and processing elements based on their priority is a fundamental to an abstract data type called the **priority queue**. The fundamental functions of the priority queue includes,

- enqueue - add element with priority to the collection
- front - retrieve element with highest priority in the collection
- dequeue - remove element with highest priority from the collection

There are several low level data structures discussed in earlier sections that can allow for such behavior. Notably we could use any of the following,

- Array List (sorted or unsorted)
- Linked List (sorted or unsorted)
- BST
- AVL

The runtimes for Array List and Linked List would be very bad since at least one modification function for each data type would have an average of a linear runtime. The BST is better because all of the functions needed from the priority queue would be an average of logarithmic, but the worst case runtimes would be linear. The AVL would avoid any linear runtimes, but at the cost of an increased complexity in code and best case runtimes all taking logarithmic time. Below is a table of the runtimes of a priority queue implemented using an AVL,

Function	Best	Average	Worst
Enqueue	$\Theta(log(N))$	$\Theta(log(N))$	$\Theta(log(N))$
Front	$\Theta(log(N))$	$\Theta(log(N))$	$\Theta(log(N))$
Dequeue	$\Theta(log(N))$	$\Theta(log(N))$	$\Theta(log(N))$

A better low level data structure exists that can help create a priority queue with simple code, and better runtimes than the AVL. The better data structure is called the **heap**.

14.1 Heap

The heap's structure can be motivated by the problem the heap solves. The heap needs to have fast accesss to the element with the highest priority. The fast highest priority access can be done by storing the element with the highest priority near the entry point—the head of a linked list or the root of a rooted tree—of the data structure. A linked list would not be efficient due to the linear time traversals to insert values. A tree could give logarithmic behavior, so trees can be leveraged.

A heap will therefore be a rooted tree where the root is the highest priority. The root as the highest priority element can be used as the structure for all subtrees. A recursive structure ensures that any node will be a higher priority of all it's descendants, which means tree exploration is not necessary in many cases.

14.2 Priority Queue Restrictions

Further constraints will be put on the priority queue in this class to ensure decent runtimes. An N-ary rooted tree implementation can be difficult to structure and navigate. Due to the added complexity of having an aribtrary number of children, when making a priority queue in this class a binary tree—each node has at most 2 children—will be used.

Trees can grow very long an lead to linear runtimes when severely unbalanced. The priority queues used in this class will be complete trees. A **complete binary tree** is a tree where every level of nodes must be completely filled in (from left to right) for any nodes to appear on the next level.

Figure 36 is an example tree where the values in the nodes represent the order of creation.

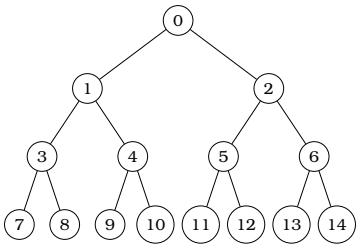


Figure 36: Example of a complete binary tree structure where the values in the nodes indicate insertion order.

The complete binary tree version of the heap will be referred to as a **binary heap**.

14.3 Binary Heap Insertion

To insert a value into the binary heap the following things must happen,

- a node must be created at the next spot near the bottom of the heap
- the new value has to exist in the binary heap
- the resulting structure should abide by the heap invariant—nodes are higher priorities than their descendants

The process usually involves initially placing the new value at the location of the new node to create—last incomplete layer on the left most side of the tree. The problem with placing the value at the new location is that the heap invariant might be violated. **Nodes should NOT be rotated to fix this problem**. The fix is not done by changing the structure of the tree, but by swapping values in the nodes. The problem with the described insertion is that a value at the bottom of the tree might have a higher priority than its parent.

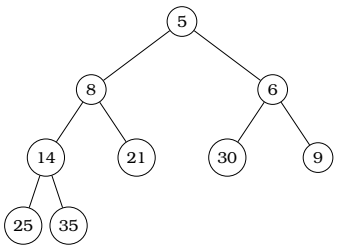
The swap will occur between the inserted node and its parent if the heap invariant does not hold for the parent of the inserted node. The swap will never move the parent of the inserted node into a bad spot; the parent was already higher than its subtree, so it moving down will still ensure that the parent is the highest element of its subtree. Similarly the node that was moved up will be fine as well. The parent was already the highest priority element of the subtree the node moved up must have a higher priority than that, so transitively the node move up will still have a higher priority.

The node that moves up during this process might still be in the wrong spot if the new parent has a lower priority. The heap invariant violation is again fixed by swapping the inserted node with its parent. The swaps keep happening until the inserted node is at the root or the parent of the inserted node has a higher priority than the inserted node.

The repeated swapping a node value up the tree is known as a **percolation up**. The pseudo code for percolation up and heap insertion can be seen on the next page,

```
PercolateUp(CurrentNode):
  While CurrentNode has Parent AND
       Priority of Parent of CurrentNode < Priority of CurrentNode:
    Swap(Parent of CurrentNode, CurrentNode)

Insert(Heap, Value):
  Create Node with Value
  Insert Node at bottom of Heap
  PercolateUp(Node)
```

Heap Question 1 What is the resulting heap after inserting the value 7 into the following heap, where smaller values have higher priority?



Answer

14.4 Binary Heap Removal

To remove a value from the heap the following needs to happen,

- the rightmost node (not value) on the last layer is removed
- the value (not node) in the root of the heap is no longer present
- the nodes maintain the heap invariant

Two nodes are modified. One loses the value, and the other loses the node. To ensure that the bottom value stays in the heap, the value is temporarily placed in the root, where the value is erased from. The problem with the modification, is that the heap invariant is potentially violated, since a bottom node is now the root. To fix the invariant violation the value moved to the top can swap with one of the children. The child the node should swap with is the child with the highest priority.

Moving the higher priority child into the higher position will ensure that the node that is the root of the subtree is the one with the highest priority, and the problem of the heap invariant violation will be passed down the tree to the grandchildren. The process of swapping the violating node with the higher priority child continues until there are no children with a higher priority for the swapped node.

The process of repeatedly swapping a with the higher priority child is know as **percolate down**. Below is an example of pseudocode from perolating down and heap removal,

```
PercolateUp(CurrentNode):
  While Priority of Highest Priority Child of CurrentNode > Priority of CurrentNode:
    Swap(HighestPriority Child of CurrentNode, CurrentNode)

Remove(Heap, Value):
  Swap(Root, Bottom of Heap)
  Remove(Bottom of Heap)
  PercolateDown(Root of Heap)
```

14.5 Analysis

Per usual the runtimes of the functions for the new data structure will be analyzed. The functions under consideration will be the enqueue, dequeue, and front function for the binary heap.

14.5.1 Best Cases

The best case for insertion is when a low priority value is inserted. A low priority value with respect to the values already in the queue means the value will not move much during percolation. Assuming finding the next spot in the heap to insert is $\Theta(1)$—covered in implementation section—the total number of operations will not depend on the size of the heap, and will therefore be $\Theta(1)$.

The best case of removal is when the value swapped with the root has a high priority and can stop percolation down very early in the tree. Since the value swapped is at the bottom of the heap, the early percolation stop can only occur when the child that swaps during percolation is on the opposite side from the subtree, but the children of the child swapped are a lower priority than the node percolating down the heap. The structure is not likely to happen. However, the described behavior allows for $\Theta(1)$ runtime.

The best case for accessing the highest priority is $\Theta(1)$, because the node is always at the top of the heap.

14.5.2 Worst Cases

The worst case for insertion is when the highest priority value is inserted. The node might have to percolate to the root. If this is the case, the runtime will depend on the number of layers of the heap. The number of nodes grows exponentially with respect to the layer, which by inversion means the number of layers grows logarithmic with respect to the number of nodes. For this reason the number of layers and subsequently the resulting runtime is $\Theta(log(N))$ in the worst case for insertion.

The worst case for removal is when the value swapped to the root needs to percolate to the bottom of the heap. Percolation to bottom happens when the priority of the node swapped is very low. Percolation to the bottom requires only a constant number of operations per layer. Since the number of layers is logarithmic with respect to the number of nodes the runtime becomes $\Theta(log(N))$ in the worst case.

The worst case for accessing the highest priority is $\Theta(1)$, because the node is always at the top of the heap.

14.5.3 Average Cases

The runtimes of the heap can be a little tricky for the average case.

For insertion into the heap the value is initially placed at the bottom. Figure 37 shows how many nodes are on each layer. Most of the nodes are on the bottom layer. These nodes have very low priority. On average the nodes at the bottom will be some of the lowest priority nodes. Informally these nodes are half of the nodes with the lowest priority. The chance that an inserted node would need to move up a layer is 50%. The chance that the node moves up another layer is 25%. A summation representing the average can be formed,

$$\sum_{i=1}^{log_2(N)} (i)\frac{1}{2^i}$$

$$= \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{log_2(N)}{N}$$

Define the above summation to be S. Consider $2S$,

$$2S = \frac{1}{1} + \frac{2}{2} + \frac{3}{4} + \cdots + \frac{log_2(N)}{N/2}$$

Consider $2S - S$,

$$2S - S = \frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{N/2} - \frac{log_2(N)}{N}$$

Simplification gives,

$$S = 2 - \frac{1}{N/2} - \frac{log_2(N)}{N}$$

$$S = 2 - \frac{log_2(N) + 2}{N}$$

That above value is approximated by $\Theta(1)$. Thus the runtime is constant for the average case insertion into a heap.

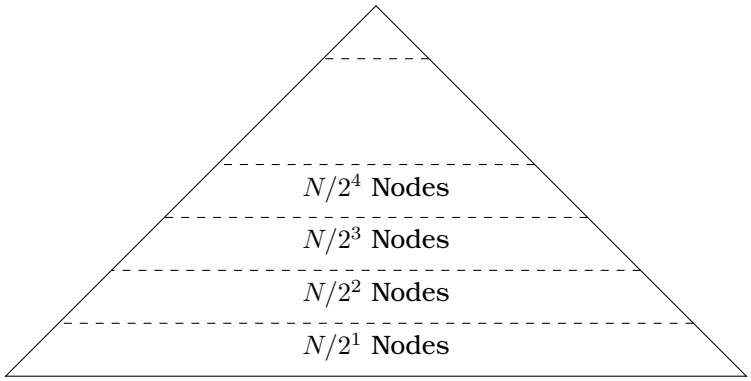


Figure 37: Number of nodes per layer in a heap

The removal on average is worse, because the value moved to the top will be a large priority on average. The chance that it moves to the bottom of the heap will be about 50%. The worst case could be expressed as $.5log_2(N) + .5(someothercase)$, but the other case is no worse than log, so the runtime would be approximated by $\Theta(log(N))$.

The average case for accessing the highest priority is $\Theta(1)$, because again the node with the highest priority is always at the root which has $\Theta(1)$ access time.

In summary the priority queue will have the following runtimes,

Function	Best	Average	Worst
Enqueue	$\Theta(1)$	$\Theta(1)$	$\Theta(log(N))$
Dequeue	$\Theta(1)$	$\Theta(log(N))$	$\Theta(log(N))$
Front	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

14.6 Implementation

Rather than using a linked data structure—like a series of nodes that combine to form a tree—typical binary heap implementations are created using an array. An array implementation is possible because the nodes are inserted and removed in a fixed order. When a node is created or removed the only factor that matters for determining the location of the modified node is the number of nodes that are currently in the tree.

Figure 38 shows the index for each node,

To find make the heap work efficiently the children and parent of each node need to be found quickly.

Below is a table representing the children and parent index for each node,

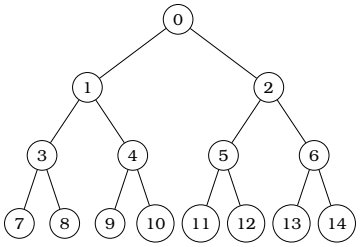


Figure 38: A tree where each value denotes the index in the array for the particular node.

Index	Parent	Left	Right
0		1	2
1	0	3	4
2	0	5	6
3	1	7	8
4	1	9	10
5	2	11	12
7	2	13	14
8	3	15	16
9	3	17	18

The pattern for finding the parent for index x is $\lfloor \frac{x-1}{2} \rfloor$. The pattern for the left child for index x is $2x + 1$. The pattern for the right child for index x is $2x + 2$.

14.7 Heapify

A well known algorithm is one that can convert a random unsorted array into a heap. The function that converts an array to a heap is known a heapify. The heapify algorithm typically refers to the linear time (worst case) version of the algorithm. An array can be converted by inserting values ones at a time into an initially empty heap. The average case would be constant per insertion for a total of $\Theta(N)$, but in the worst case, the runtime could be logarithmic for each insertion for a total of $\Theta(Nlog(N))$ time.

The slow version of the algorithm described above effectively works by percolating up each value of the array from index 0 to index N-1. A faster version exists by building from the bottom of the heap rather than the top of the heap—as described in the slow version. The faster version starts at index $N - 1$ and proceeds to index 0. Each value is percolated down the heap.

14.7.1 Correctness

Typically when a value is percolated down the resulting data structure is a valid heap, but fundamentally percolate down joins 2 heaps by taking some new value that will be in the heap and moving it into the heap that has the higher priority value. Effectively the input to percolate down is a node and 2 heaps, and the result is a single heap. The same behavior happens when percolating from the bottom of the heap to the top. Each value spot in the array become the root of a subheap as long as the 2 children are the root of subheaps. The 2 children are roots of subheaps, since they were percolated down first. The heap invariant is held true inductively.

14.7.2 Heapify Analysis

The percolate down method of the heapify algorithm is fast, because most of the elements are at the bottom. The top of the heap might move down a logarithmic number of spots, but the total number of movements is not too large. Half the values will move down 0 spots. Formally the worst case number of comparisons can be modeled using the following sum,

$$\sum_{i=1}^{log_2(N)} i(\frac{N}{2^i})$$

$$= N \sum_{i=1}^{log_2(N)} i(\frac{1}{2^i})$$

$$N(\frac{1/2}{+}\frac{2}{4} + \frac{3}{8} + \cdots \frac{log_2 N}{N})$$

Earlier it was shown that the expression in the parenthesis become $2 - \frac{log_2(N)+2}{N}$. Thus the total becomes,

$$N(2 - \frac{log_2(N) + 2}{N})$$

$$= 2N - log_2(N) - 2)$$

The result of which is $\Theta(N)$.

15 Tries

Balanced binary search trees are pretty efficient, but the algorithm is complex. Additionally, sometimes comparisons can take a very long time, and when inserting strings into a BST, there are potentially many comparisons that will look at the same data in the string to look up. If the strings stored in a BST where something like DNA, where many sequences would have long identical prefixes, then every comparison would needlessly look at the same unimportant characters.

A modification of the tree structure exists that ensures that every character is looked at exactly once creating a lookup runtime something similar to $\Theta(1)$ "comparison". This miracle data structure is a tree that supports retrieval of strings, and is frequently referred to as the **Trie**.

The trie works by "storing" the pieces of the look-up information along the edges of the tree instead of in the node itself. For a string the information for looking up the string would be the character sequence that composes the string. "alice" would be 'a', 'l', 'i', 'c', and 'e'. Figure 39 contains a trie, and the node with label E denotes the node the would contain the data for "alice". In that same tree the node that would contain the data for "ali" would be in the node labeled C. The nodes that would contain the data for "alex" or "bob" are not drawn.

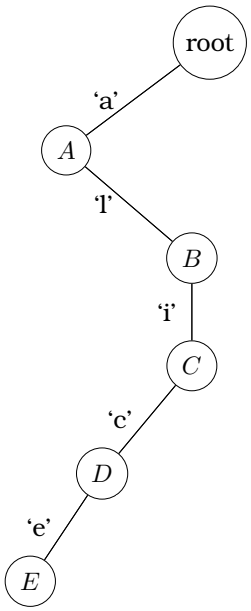


Figure 39: A trie showing the path that would be traversed to reach "alice"

15.1 Code

If a trie is used to implement a set, a flag value is stored in the node that represents membership. A 0-flag or `NULL` node represents that the corresponding data is absent from the set, and a flag of 1 would represent the value is contained in the set. Every node will also need pointers to the children of the node. Below is a possible definition of the structure in C for a node of a trie,

```
typedef struct Node Node; // Alias for the node
struct Node
{
   Node * children[26]; // 26 pointers to the possible children
   int flag; // flag to signal if a some sequence is in the set
};
```

15.1.1 Insertion

Below would be an example of inserting a word into a trie without recursion,

```c
// Function to insert a word into a trie
// The pointer to the root's address is passed so that if the tree is empty,
// a new root can be created
void insertNode(Node ** root, char * word)
{
   if (*root == NULL) // Check if the tree is empty
   {
      *root = (Node *) calloc(1, sizeof(Node)); // Create a new root
   }

   while (word[0] != '\0') // Loop to the end of the word
   {
      int index = word[0] - 'a'; // Get index based on letter
      word++; // Point to the next letter in the word
      root = &((*root)->children[index]); // Move to the next node

      if (*root == NULL) // Check if the child did not exist
      {
         *root = (Node *) calloc(1, sizeof(Node)); // Create a new root
      }
   }

   (*root)->flag = 1; // Add the word
}
```

15.1.2 Aggregation

Aggregation is when parts of some data come together to form a whole. An simple example of aggregation could be used to find the number of words that start with a particular prefix. Each node could store the count of the words in its subtrie. When a word is added or removed from the trie the nodes that are visited on the path can use a count from each of their child to know the count for its own subtree. To find the count for a particular node the time can be computer in $\Theta(1)$ by using the stored value, where otherwise the value would have to be looked up. Below is an example of how aggregate for a particular node

```c
typedef stuct Node Node;
struct Node
{
   int flag;
   int count; // Number of words in a subtrie
   Node * childen[26];
};

// Function to aggregate the children of a particular node
void aggregate(Node * cur)
{
   cur->count = cur->flag; // Add 1 to count if the flag is 1
   for (int i = 0; i < 26; i++) // Loop through all children
   {
      if (cur->children[i]) // If the child exists...
      {
         cur->count += cur->children[i]->count;
      }
   }
}
```

15.2 Pros-Cons

The algorithm is very fast—Θ(Length of String)—at looking up, inserting, and removing strings. The downside is that the data structure uses a lot of memory because of all the pointers.

15.3 Applications

The trie is used in many ways. Aside from allowing for quick look-up, tries are used in Huffman encoding trees and suffix trees. Certain automatas also use graphs with information stored on the edge. Aho-Corasick Algorithm works on such graphs.

16 Hashes

Balanced Binary Search trees—such as the AVL—has a log runtime for looking up values. Tries used a stupid amount of memory. Arrays have fast look ups, but if values were stored in it, finding it can be challenging, if we don't store values in a smart way. Hashes are that smart way of storing values.

16.1 Hash Function

A **Hash Function** generates a compressed value—hash—that represents a fingerprint of the original data. There are many options for the hash function such as,

- Random[19] - Use a built in (or custom) pseudo-random number generator
- Length - Use the length of the data
- Folding - Sum the values of the data together
- Radix - Sum the values of the data together, but weight each value of the data by a value that increase exponentially
- Secure Hash Algorithm (SHA-#) - A family of existing, "well known" hash algorithm
- Message Digest Algorithm (MD#)- A family of existing, "well known" hash algorithm

Not every function can be a hash function. Here are the properties of hash functions that are good to have,

- Deterministic - The same data will generate the same resulting hash value. This property is required of all hash functions.
- Well Distributed - All the possible hash values that are returned should be equally likely of being returned for random data.
- Non-invertable - It should be challenging to find a new value that generates a particular hash value, even if you know an old value that generates that hash. Aside from some SHA and MD algorithms, none of the previous ones mentioned are non-invertable. Additionally, SHA-1 can be inverted, and MD2, MD4, and MD5 are all invertable with some effort.[20]

16.1.1 Examples

On the next page are some examples of how some of the hash functions can be implemented for a string,

[19] You must seed your random number generator every time you use hash functions that leverage pseudo-random functions

[20] MD1 and MD3 are not published; MD1 is proprietary, and MD3 was never meant to see the light of day.

```c
// A "random" hash function
int randHash(char * data) {
   int seed = data[0]; // Compute a seed based on the data
   srand(data);
   return rand();
}

// A length based hash function
int lengthHash(char * data) {
   int res = 0;
   while (data[res]) {
      res++;
   }
   return res;
}

// A folding hash function
int foldingHash(char * data) {
   int res = 0;
   int index = 0;
   while (data[index]) {
      res += data[index];
      index++;
   }
   return res;
}

// A radix hash function
int radixHash(char * data) {
   int res = 0;
   int base = 2; // The amount the scale increases by exponentially
   for (int i = 0; data[i] != '\0'; i++) {
      res *= base; // Scale all the previously added values
      res += data[i]; // Added a new value
   }
   return res;
}
```

16.2 Collisions

When a hash function on 2 different known pieces of data produce the same hash value, a collision occurs. It should near impossible to generate collisions on cryptography secure hash algorithms. Collisions are the bad for hash function usage. The point of having a hash function is that by comparing a hash value rather than a full piece of data can help determine if some data may be equal without knowing the full piece of data. When a notable lack of collisions occurs, a hash function can be useful for checking passwords without storing them.

Hashes are also commonly used in structures called **Hash Tables**, which use a hash function under modulo to insert values into an array of a fixed size. The Hash Table is typically used to implement a set-like data structure. The problem is that with small array sizes collision are made even more common. The problem is referred to as the **birthday paradox**. By a well known *approximation*, when the number of values reaches about the square root of the capacity the probability that a collision will have occurred is substantially high[21].

For us normal programmers that don't want to make a cryptographically secure hash algorithm to store values or make an array that is roughly $\Theta(N^2)$ memory, a work around is

[21] there is also a Taylor series approximation that has a higher accuracy

needed. Several solutions to handling collisions will be presented such as,

- Double Hash
- Probing
- Chaining

16.2.1 Double Hash

Double hash is when a second, independent hash function is used to find an alternate location where a value could be placed if some value is already in the spot determined by the first hash value. The drawback is that the second location could also have a collision, which would motivate the need for a third hash function, which could also collide. A better collision handling method should be sought for arrays that have sizes close to the number of values in them.

16.2.2 Probing

Probing is when after a collision a "nearby" empty spot is selected to hold the value being inserted. There are 2 main types of probing,

1. Linear
2. Quadratic

Linear Probing increments the index by 1 until an empty spot is found. Another way of thinking about linear probing is to try to insert some `data` at `Hash(data)+collisions`, where `collisions` is the number of collisions encountered in the process of performing this insertion. Below is an example of how linear probing can be used to insert a value into a hash table.

```
// The value when a spot is unused
#define EMPTY -1
// Table struct definition
struct Table {
   int size, cap;
   int * values;
};
// Function to insert a value into a hash table
void insert(Table * tab, int value) {
   // Find the hash value modded by the table capacity to wrap around
   int index = hashfunction(value, tab->cap);

   // Loop until an empty spot is found
   while (tab->values[index] != EMPTY) {
      index++; // Move to the next location

      // Wrap around if necessary
      if (index == tab->cap) index = 0;
   }

   tab->values[index] = value; // Fill spot with passed in value

   tab->size++; // Update size
}
```

Quadratic probing instead of adding the collisions in a linear manner adds the number of collision in a quadratic manner; the location of insertion is $\texttt{Hash(data)+collisions}^2$. The upside of such an insertion technique is that values that cluster around the same hash value (but different values) will not collide very much. The downside is that not all indices

can be reached using this probing technique. Only about half the indices will be reachable from any starting point. The array can only guarantee a half capacity if the array is a prime size.

A modification to the quadratic probing method exists that can fully utilize an array. To fully utilized the capacity requires using alternating positive and negative probing values. The insertion location $\texttt{Hash(data)+(-1)}^{\texttt{collisions}}\texttt{collisions}^2$. Additionally the prime size needs to have a remainder of 3 when divided by 4. **DO NOT DO THIS MODIFICATION ON THE FE.**

Perhaps the biggest problem when using probing is handling removals. Removals can make a spot in the array empty. If probing was used to place a value by skipping over a spot that has been removed, an empty spot might falsely indicate a value is not in the array. To fix the false empty problem a removed flag can be associated with each spot in the array to denote if a value was removed from empty locations. When resizing the array these empty spots can be discarded.

16.2.3 Chaining

Perhaps the best way to handle collisions is to store all the values that have the same hash in one spot of the array by using an array of collections (e.g. Linked Lists[22]). When a value is inserted into the Hash Table the value is inserted into the Collection at the the index based on the Hash function. Below is a simple example how insertion can be performed in C,

```c
// Node struct definition
struct Node {
   struct Node * next;
   int value;
};
// Table struct definition
struct Table {
   int size, cap;
   struct Node ** heads;
};
// Function to insert a value into a hash table
void insert(Table * tab, int value) {
   // Find the hash value modded by the table capacity to wrap around
   int index = hashfunction(value, tab->cap);

   // If the value is not in the collection
   if (!contains(tab->heads[index], value)) {
      // Insert the value into this table.
      tab->heads[index] = insertHead(tab->heads[index], value);
   }
}
```

16.3 Analysis

Analysis on collision handling is important for determining runtimes. The best case is if no collision occurs, which gives an approximation of $\Theta(1)$. The worst case for these functions occurs when all the values collide at the same hash. The runtime for checking all these values is $\Theta(N)$. The average case can be found by creating a recurrence relation after making an important observation. It is ideal to not use too many values in the table. If the array is kept partly empty (i.e. at most some ratio—α—of the array is full), then the ability to find an empty spot is increased to a reasonable level. The recurrence relation can be approximated by an upper bound using the following,

$$T(N) = 1 + (\alpha)(T(N)) + (1 - \alpha)(0)$$

[22] AVLs could make runtimes even better

Solving for $T(N)$ results in $T(N) = \frac{1}{1-\alpha}$. If α is .5 (i.e. the array is at most half full), the runtime is bounded by 2 on average. It should be noted that the above runtime is considered for probing. The average case runtime for chaining is theoretically better, since collisions only happen with values that have identical hash values. Chaining has the added benefit that the array can be more than half full and still maintain the constant average case runtime. In summary using a hash table of N values using the collision handling methods results in the following runtimes for insertion, removal, and containment checks,

Function	Best	Average	Worst
Hash Table Functions	$\Theta(1)$	$\Theta(1)$	$\Theta(N)$

A Szumlanski's Eustis Guide

On the next page

A Guide to Eustis and the Linux Command Line

By Dr. Szumlanski

Updated: Fall 2021

Table of Contents

Introduction....2
Your Eustis Password....2
Looking for Help?....2
Connecting to Eustis While Using Campus Wifi....2
Establishing a UCF VPN Connection (Windows and Mac)....2
Connecting to Eustis from a Command Line Terminal (Mac and Linux)....3
Troubleshooting “Could not resolve hostname” Errors....3
Transferring Files to Eustis via Terminal (Mac and Linux)....4
Connecting to Eustis with MobaXTerm (All-in-One Solution for Windows)....4
Basic Linux/Mac Commands (Including Command Line Compilation)....4
Appendix A: Establishing a UCF VPN Connection (Manual Installation for Linux Users)....6

Introduction

Eustis is the server we'll use this semester as a common testing platform where all assignment submissions need to run and work in order to receive credit. Eustis accounts have already been created for everyone who was enrolled in this course by Friday, August 20. Accounts for all other students will be created by Sunday, August 29. You can log into Eustis from computers on campus, but to log into Eustis from home (or certain dorms on campus), you must first establish a UCF VPN connection. (See details below.)

Server (using an SSH client (not a web browser), as described below): eustis.eecs.ucf.edu
Username: your NID
Password: your NID password

Important note: When typing your password, Eustis will not print asterisks ('*') to the screen. It might look like the system is frozen, but don't worry; it's capturing your password as you type it.

Your Eustis Password

Eustis is integrated with UCF's NET domain, meaning that you will use your NID and NID password to log in to the system once your accounts are created. If you have trouble logging into Eustis or get locked out of your account, you can e-mail your NID to helpdesk@cecs.ucf.edu and ask for assistance.

Looking for Help?

For account issues (unrecognized username or invalid password), please e-mail helpdesk@cecs.ucf.edu. Be sure to include your NID, let them know you're enrolled in this course, and provide a brief description of the login problem you're encountering (e.g., the exact error message Eustis is giving you).

For help establishing a connection to Eustis, please see one of the TAs in office hours. Your fellow classmates will also be a great resource for help with connection issues!

Connecting to Eustis While Using Campus Wifi

If you're using campus wifi, please log into the wifi network with your NID and NID password (*not* as a guest) in order to gain access to Eustis. If you do that, you won't need to go through the steps of manually establishing a VPN connection (except, sadly, in some of the dorms on campus).

Establishing a UCF VPN Connection (Windows and Mac)

Skip this if you're using a campus computer or if you're connected to UCF's wifi with your NID/password (not as a guest). If you're using Linux, you might have to manually install the Cisco AnyConnect Client. See pg. 6 of this document for instructions on how to do so.

1. Open https://secure.vpn.ucf.edu in your web browser.
2. Sign in as a student using your NID and NID password.
3. Download some stuff for Cisco AnyConnect.
4. Open up Cisco AnyConnect and type in: https://secure.vpn.ucf.edu
5. Again, type in your NID and NID password. This creates a VPN connection, and you can now log into the Eustis server using one of the methods described below.

Connecting to Eustis from a Command Line Terminal (Mac and Linux)

1. Open a terminal window and connect to Eustis by typing the following:

   ```
   ssh YOUR_NID@eustis.eecs.ucf.edu
   ```

   Note: You need to replace "YOUR_NID" with your actual NID.

2. When prompted for your password, type your NID password. Note that Eustis does not print asterisks ('*') to the screen as you type your password. It might look like the system is frozen, but don't worry; it's capturing your password as you type it. After you type your password, hit enter.

   Note: If you get an error about IP spoofing or a "REMOTE HOST IDENTIFICATION HAS CHANGED" warning when connecting, you might need to remove your old ssh key, like so (and then try re-connecting):

   ```
   ssh-keygen -R eustis.eecs.ucf.edu
   ```

Troubleshooting "Could not resolve hostname" Errors

Here are the most likely causes of "Could not resolve hostname" errors when trying to connect to Eustis:

1. You're off campus, but you're not connected to the VPN. Connect to the VPN using the instructions in this document.
2. You're on campus, but you **are** connected to the VPN. Disconnect from the VPN and try logging in to Eustis again.
3. You're on campus, but you're connected to the UCF_GUEST wifi network. Connect to UCF_WPA2 instead.
4. You're on campus, you're connected to UCF_WPA2, and you're **not** connected to the VPN. You're doing the right thing there, but some pockets of the UCF_WPA2 network on campus just aren't authenticated in such a way that they allow connections to Eustis for some reason – especially if you're in a dorm. Connect to the VPN and try again.
5. You're using the bash shell on Windows (i.e., the Ubuntu-based Windows Subsystem for Linux). For some reason, that terminal sometimes has trouble connecting to Eustis by hostname and requires you to use an IP address instead, as shown below. This IP address is also listed in the section titled "Reference: Direct IP Address for Eustis" in the "Introduction to Eustis" page in Webcourses.

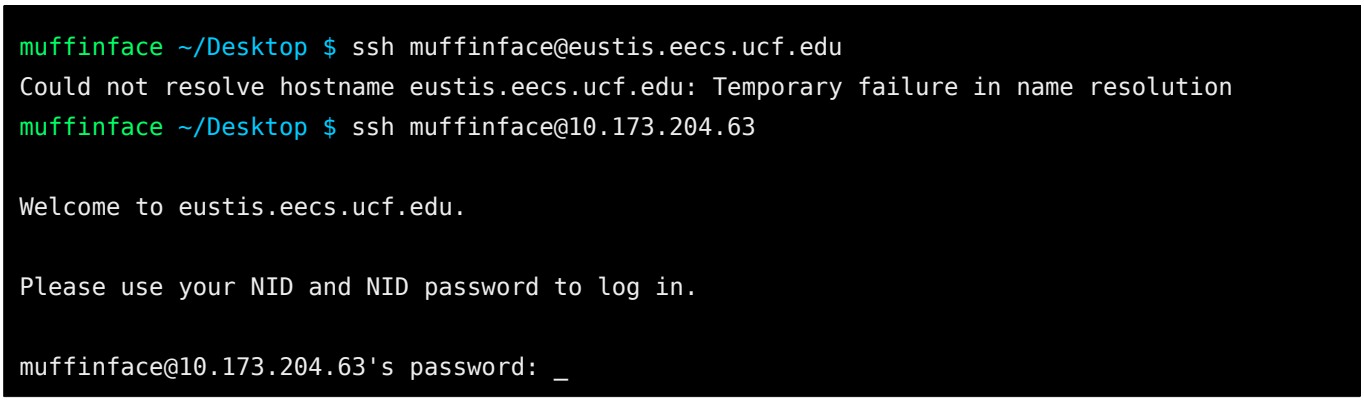


Figure 1: Connecting to Eustis through WSL (the Linux-based bash prompt in Windows) might require the use of Eustis's direct IP address rather than eustis.eecs.ucf.edu.

Continued on the following page...

Transferring Files to Eustis via Terminal (Mac and Linux)

1. Open a new terminal window.
2. Use the “cd” command to navigate to the directory containing the file(s) you want to transfer. E.g.:

```
cd Desktop/cop3502/program1
```

3. Type the following to transfer some file to Eustis. Note that you need the “:~/” at the end of this command:

```
scp some_file.txt YOUR_NID@eustis.eecs.ucf.edu:~/
```

4. When prompted, enter your NID password (and keep typing even though no asterisks appear as you do).
5. (*Super Handy Trick!*) To copy an entire folder to Eustis, use the *-r* flag, like so:

```
scp -r MyProjectFolder YOUR_NID@eustis.eecs.ucf.edu:~/
```

Connecting to Eustis with MobaXTerm (All-in-One Solution for Windows)

Please note that some versions of the SSH program included with Windows will not be able to connect to Eustis. You can instead download MobaXTerm – a free program that provides a beautiful, all-in-one solution for working with Eustis. MobaXTerm allows you to connect to Eustis, transfer files, and interact with the Eustis command line all in one program. It also allows you to edit files that are stored on Eustis (rather than editing them on your computer and then uploading). At the following link, click “GET MOBAXTERM NOW!” to download the free version: http://mobaxterm.mobatek.net/

To connect to Eustis with MobaXTerm, you’ll want to establish a new SSH session with the following settings:

Remote host: eustis.eecs.ucf.edu
Specify username: (*enter your NID in this field*)
Port: 22

Basic Linux/Mac Commands (Including Command Line Compilation)

Here’s a list of some useful commands that you’ll use throughout the semester as you work at the command line:

1. To compile a source file (.c file) into an executable:

```
gcc source.c
```

2. By default, the command in (1) will produce an executable file called *a.out*, which you can run by typing:

```
./a.out
```

3. To name the executable something else when you compile:

```
gcc source.c -o whatever
```

4. To run the program created in (3), which is called *whatever*:

```
./whatever
```

5. Running a program could potentially dump a lot of output to the screen. In Linux, if you want to redirect the output of your program to a text file (such as *output.txt*), it's easy. Just run the program using the following:

```
./a.out > output.txt
```

   This will create a file called *output.txt* that contains the output from your program.

6. Linux has a helpful command called *diff* for comparing the contents of two files, which is really helpful if you have one text file containing the output your program just produced and another file containing the solution your program *should* produce. You can see whether the contents of two files match exactly (character for character) by typing, e.g.:

```
diff output.txt solution.txt
```

   If the contents of *output.txt* and *solution.txt* are exactly the same, *diff* won't have any output. It will just look like this:

```
seansz@net1547:~$ diff output.txt solution.txt
seansz@net1547:~$ _
```

   If the files differ, *diff* will spit out some information about the lines that aren't the same. For example:

```
seansz@net1547:~$ diff output.txt solution.txt
1c1
< Hello, world!
---
> Hello world!
seansz@net1547:~$ _
```

7. To list all the files in your current directory:

```
ls
```

8. To delete a file, such as *output.txt*:

```
rm output.txt
```

9. To dump the contents of a file, such as *output.txt*, to the screen:

```
cat output.txt
```

10. To compile multiple source files into an executable, which you will have to do for some of your assignments this semester, simply list all of the source files you want to compile after the *gcc* command:

```
gcc source1.c source2.c source3.c
```

11. If you want to compile a program that includes *math.h* library functions, you need to link the math library by adding the *-lm* flag at the command line:

```
gcc source.c -lm
```

Appendix A:
Establishing a UCF VPN Connection (Manual Installation for Linux Users)

If you're using Linux, you might have to manually install the Cisco AnyConnect Client in order to connect to UCF's VPN. Here's how to do it.

1. Load up https://secure.vpn.ucf.edu in your web browser.
2. Sign in as a student using your NID and NID password.
3. If the client does not download right away and seems to be stuck at "Sun Java Applet started, this may take up to 60 seconds..." just be patient. It might take 2-3 minutes. Eventually, it should come up with this message:

```
Web-based installation was unsuccessful. If you wish to install the Cisco AnyConnect
Secure Mobility Client, you may download an installer package.

Install using the link below:

Linux x86_64
```

4. Right-click the link and save the file (`vpnsetup.sh`) to your desktop.
5. Open a terminal and enter the following commands:

```
# Go to your Desktop.
cd ~/Desktop

# Make the script you downloaded executable.
chmod +x vpnsetup.sh

# Run the script with administrative privileges. This will ask you for your Linux
# account password. That's okay. Type it in and hit [enter].
sudo ./vpnsetup.sh
```

6. Find and run the "Cisco AnyConnect Secure Mobile Client" in your Applications menu. (It was filed under "Other" on my system, but your mileage might vary.)
7. Enter https://secure.vpn.ucf.edu, click "Connect," and supply your NID and NID password.
8. You'll have to click "Accept" for some terms of service.
9. Open a terminal and connect to Eustis using your NID and NID password:

```
ssh YOUR_NID@eustis.eecs.ucf.edu
```

Note: You need to replace "YOUR_NID" with your actual NID.

Important note: Cisco AnyConnect sometimes produces the following error: "AnyConnect cannot confirm it is connected to your secure gateway. The local network may not be trustworthy. Please try another network." If that happens to you, you can either simply try switching to Firefox, or install OpenConnect instead.

B Basic Linux Commands

This section covers a few useful commands that can be used in Linux-based systems.

1. To compile C source files from command lines you can use the `gcc` (GNU C Compiler) command—assuming it has been installed. To properly use the `gcc` command you should type "`gcc`" followed by the name of the file you which to compile. Below is an example compiling a C file called "`source.c`"

   ```
   username@hostname:~$ gcc source.c
   ```

   If your program in unable to compile you might see something that looks like the following,

   ```
   username@hostname:~$ gcc source.c
   source.c: In function ‘main’:
   source.c:5:13: error: expected ‘;’ before ‘}’ token
      5 |     return 0
        |            ^
        |            ;
      6 | }
        | ~
   ```

   It is advised that you read these errors. The compiler will tell you roughly the error it encountered and will sometimes (depending on your compiler) suggest what you can do to fix the error.

2. If you compile from command line using `gcc` in a Linux-based system the compiler will generate a binary file—called `a.out`—that can be run by typeing a dot (`.`) and forward slash (`/`) before the binary file name. Below is an example of what running the executable should look like,

   ```
   username@hostname:~$ ./a.out
   Hello, World!
   username@hostname:~$
   ```

3. If you want to have more than one program compiled at once, you can change the name the binary using a special argument when using the `gcc` command. The name of the binary can is specified using a dash oh (`-o`) prior to the desired name all within the `gcc` command. Below is an example of making an executable called "`executable_name`",

   ```
   username@hostname:~$ gcc source.c -o executable_name
   ```

4. You can run custom named executables the same way you ran the `a.out` by using the dot slash before the name. Below is an example of running a custom named executable,

   ```
   username@hostname:~$ ./executable_name
   ```

5. Typing in the same input to your program will get very tiring through out the semester, especially when debugging. To make your job easier I suggest using standard input redirection. You can get your executables to take the contents of a file as if it was typed into the console after the executable has been run. Using the contents of a file as standard input is referred to as standard input redirection. Standard input redirection can bed done using the less than sign (`<`) followed by the name of the file you wish to treat as the standard input. Below is an example of standard input redirection,

   ```
   username@hostname:~$ ./a.out < input.txt
   ```

The output of your program can also be redirected to a file in a similar manner. Instead of using the less than sign, the greater than sign (`>`)

```
username@hostname:~$ ./a.out > output.txt
```

Input and output redirection can be used simultaneously. Below is an example of reading from "`input.txt`" and writing to "`output.txt`".

```
username@hostname:~$ ./a.out < input.txt > output.txt
```

6. If you have your program's output in a text file and the desired output in a second text file you can compare them automatically using the `diff` (difference) command. Below is an example of using `diff` on 2 files, "`output.txt`" and "`solution.txt`"

```
username@hostname:~$ diff output.txt solution.txt
```

When running the `diff` command, the 2 files are considered identical when no output is produced. Below is an example of what identical files should produced when run through `diff`.

```
username@hostname:~$ diff output.txt solution.txt
username@hostname:~$
```

When running the `diff` command, when the files have differences, there will be typically output produced that shows where the files are different. Below is an example of what your `diff` command might look like when the files are different.

```
username@hostname:~$ diff output.txt solution.txt
1c1
< answer
---
> Answer
username@hostname:~$
```

Sometimes one file might have extra invisible characters (whitespace) that can make the files appear to be different to your system. These extra characters can be produced if the files are produced on different systems, such as Dos (windows) vs Unix (Linux). These invisible characters can be ignored using some extra parameters (`-wb`) to the diff command. Below is an example of what 2 files with different whitespace might look like when run through `diff` at first without and then with the "`-wb`" flag,

```
username@hostname:~$ diff output_with_whitespace.txt solution.txt
1c1
< Answer
---
> Answer
username@hostname:~$ diff -wb output_with_whitespace.txt solution.txt
username@hostname:~$
```

7. The contents of a directory can be printed to the screen to list any files and directories that are located in your current working directory using the `ls` (list) command. There will typically be color coding on the output to distinguish between files, executables/binaries, and directories. Below is an example of what you may seen when running the `ls` command,

```
username@hostname:~$ ls
a.out  executable_name input.txt  output.txt  source.c  subfolder
```

8. The contents of a file can be output directly to the terminal using the `cat` (concatinate) command. You simply pass the name of the file to print as the sole argument to the command. Below is an example of the `cat` command,

```
username@hostname:~$ cat output.txt

#include <stdio.h>

int main() {
   printf("Answer\n");
   return 0;
}
username@hostname:~$
```

9. **Be very careful with this command. There is no recovering deleted files.** You *can* delete files. File deletion is done through the `rm` (remove) command. The files you wish to be deleted should follow the `rm`. Below is an example of removing the "`source.c`" file and listing the contents of the directory before and after the command,

```
username@hostname:~$ ls
a.out  executable_name input.txt  output.txt  source.c  subfolder
username@hostname:~$ rm source.c
username@hostname:~$ ls
a.out  executable_name input.txt  output.txt  subfolder
```

   DO NOT USE "`rm *`" unless you know what you are sure. The command "`rm *`" removes all files in your directory. Dr. Meade has done this several times on accident and lost hours of work each time.

10. A subdirectory (i.e. folder) can be created using the `mkdir` (make diretory) command. Using directories can keep your projects and source files organized, which can hopefully lead to less chances of submitting the wrong file. The name of the directory created should follow the "`mkdir`" command. Your directory should not have a space in the name or you will end up making multiple directories for each word you passed in. Below is an example of the command being used with the contents of the directory printed before and after,

```
username@hostname:~$ ls
a.out  executable_name input.txt  output.txt  source.c  subfolder
username@hostname:~$ mkdir new_dir
username@hostname:~$ ls
a.out  executable_name input.txt  new_dir  output.txt  source.c  subfolder
```

11. The location of where your terminal's commands originate is sometimes referred to as the `current working directory`. The `current working directory` is displayed after the colon and before the dollar sign in the `shell prompt` (the text before the dollar sign). To change your current working directory you can use the `cd` (change directory) command. You can change into a subdirectory by using the subdirectory name as the argument to the `cd` command. Below is an example of moving into a directory,

```
username@hostname:~$ cd my_dir
username@hostname:~/my_dir$
```

    Note that the `current working directory` in the `shell prompt` changes.

    If your current working diretory is in some subdirectory and you wish to go back a level you can use the `..` directory in `cd` to move to the parent directory. Below is an example of using the parent directory,

```
username@hostname:~/folder/subfolder$ cd ..
username@hostname:~/folder$
```

Note that the `current working directory` in the `shell prompt` changes, but the `current working directory` is not the `home directory` (aka `~`).

You can move directly to the home directory by using the tilde,"~", as the argument to `cd` or by using `cd` without any arguments at all. Below is an example of 2 ways you can move to the home directory from nested directories. In the example below the second `cd` is used to move 2 folders in with a single command,

```
username@hostname:~/folder/subfolder$ cd ~
username@hostname:~$ cd folder/subfolder
username@hostname:~/folder/subfolder$ cd
username@hostname:~$
```

12. You can print out your current working directory by using the `pwd` (print working directory) command. Below is an example of using the `pwd` command,

```
username@hostname:~$ pwd
/mnt/c/Users/username
username@hostname:~$
```

13. The names of files can be changed using the `mv` (move) command. WARNING: if a file with the new file name already existed, that file will be deleted when the move occurs. Below is an example of using the `mv` command,

```
username@hostname:~$ ls
old_file_name.txt
username@hostname:~$ mv old_file_name.txt new_file_name.txt
username@hostname:~$ ls
username@hostname:~$
new_file_name.txt
```

In the above example if a file name "`new_file_name.txt`" existed before hand that file's contents will be lost.

14. The a files contents can be copied using the `cp` (copy) command. WARNING: if a file with the new file name already existed, that file will be deleted when the move occurs. Below is an example of using the `cp` command,

```
username@hostname:~$ ls
old_file_name.txt
username@hostname:~$ cp old_file_name.txt new_file_name.txt
username@hostname:~$ ls
new_file_name.txt old_file_name.txt
username@hostname:~$
```

In the above example if a file name "`new_file_name.txt`" existed before hand that file's contents will be lost.

C Binary Search

Suppose we have a collection of people—their names—that are allowed to work on some top secret project. We need to create software that gives a cleared person access to the project. We could store the people in some array. If the collection can grow, then we should consider using an array list. For an individual how do we determine "membership"—are they contained in the collection? To make this problem easier we will assume that everyone has a unique name.

C.1 Linear Search

The simple solution to the problem is to loop through the list of people and use string compare to check if the given name matches the list name. The problem is that if the list of people is large[23], then looping through could take a long time. If the person is not in the list, then it

The same reason why this method would not be used in most implementations is the reason why humans would not read every word in a dictionary starting from the beginning to find a particular word. When looking for a word in a dictionary, humans use an interpolation style search. We flip roughly to the location we expect the word be and adjust the search method based on the values found on the page. Only once the range of words is small (a couple of pages) would one consider looking at a single word at a time for the target word.

The only reason why this search method works on a dictionary is because the list of words are sorted. If we first sort the list of people, or are given the list in sorted order, we can find a person faster than comparing a questionable name against all names in the list.

C.2 Binary Search

Assume the list of names is very long (possibly millions of names). If we looked at name that comes later in the list, for example the 100th name of the list, then there are 3 possibilities we could consider:

1. The questionable name the same as the name from the list
2. The questionable name is supposed to come after the current name in the list
3. The questionable name is supposed to come before the current name in the list

In the first case we can stop immediately, although that case is not very likely.

If we end up in the third case we know that the name would have to be in the first 100 names, assuming the name is in the list. We could look at all 100 of them and that would take a lot less time than looking at all potentially million names.

If we end up in the second case we don't need to look at the first 100 names. We save ourselves some time. This process of skipping 100 names could be used repeatedly and only 1% of the names would need to be looked at if the name we are targeting is near the end.

This is still not very efficient. We are able to eliminate the names on either side of the looked up name based on the location in which the name should exist. Case 3 requires potentially a lot less effort than case 2. To balance the cases out, the looked up name could be directly in the center. Looking directly at the center will eliminate half of the search space. This pruning of search space can then be repeated on the remaining set of names. On the next page is a sketch of the binary search algorithm,

[23] hopefully a top secret project does not have that many people

```
// Below is a binary search over a sorted array of n values.
// The goal is to determine whether the target is contained in the
// array.
// Return 1 if the value is contained
// Return 0 otherwise
BinarySearch(array, n, target)
    // Store the start and end of the range of possible locations
    // of the target
    LowIndex = 0
    HighIndex = n - 1

    // Keep pruning while the range of values is too big
    While (HighIndex - LowIndex > SMALL_RANGE)
        // Find the middle index of the search range.
        // Use the midpoint formula
        MidIndex = (HighIndex + LowIndex) / 2

        // Compare the target against the middle value
        If (array[MidIndex] == target)
            Return 1
        If (array[MidIndex] < target)
            // The middle is too small.
            // Target must be later
            LowIndex = MidIndex + 1
        If (array[MidIndex] > target)
            // The middle is too big.
            // Target must be earlier
            HighIndex = MidIndex - 1
```

C.2.1 Function Inversion

The binary search can also be used to find the "inverse" of functions that are hard to invert. Consider the situation in which the square root of a value needs to be found but no built in square root function exists (perhaps in some simple non-C language).

To find the square root, y, of some value, x, it must be the case that such that $y = x^{.5}$, or $y^2 = x$. Finding the value of the square root is hard. The inverse of the square root function, the quadratic function y^2, is easy to compute. For this problem the value y can be "searched" for by using the Continuous Value Theorem (CVT), because the function, $x^{\cdot}5$, is continuous. To do so 2 y values will be stored. The first stored value will be y_{low}, which will evaluate to less than the square root of x; that is $x \times x > y_{low}$. Additionally, y_{high} will evaluate to a value above the square root of x; $x \times x < y_{high}$. The CVT says that a y value between the stored y values will evaluate to the square root of x.

The midpoint between the 2 y values can be evaluated. If the resulting x value of the y-midpoint has the given desired x, then we found the correct y (square root of x). If the y value is too low or too high, then the corresponding y bound can be updated such that the CVT can be reapplied.

Checking the midpoint can be repeated until the answer is within some tolerance. There is a chance (due to floating precision) that an exact value for y will not be found.

C.2.2 Turning Point Finding

Some problems will require finding a minimum/maximum value that satisfies some given criteria.

For example, we have a bus that picks up people from the airport. The bus can arrive at whichever time we choose, but there are a limited number of times the bus can go to the airport, M. The bus needs to be at the airport at N specific times for the people. The bus will stay there for a fixed time, T, (determined by us) for each time the bus goes to the

airport. However, the longer the time the bus stays there the more we pay the drivers. Find the maximum profits by determining the least value of T such that all people will be picked up from the airport.

We can know for a specific value of T whether the bus can pick up all the people. We do that by placing the first bus as late as possible. The bus will then pick up as many people as possible. If the bus requires more than M trips to pick up all N people, then the value of T is too small. Any T value over the minimum will always enable the bus to pick up all N individuals using this strategy.

The values of T creates the following "function".

TODO

This location of the answer can be search for by trying possible values. When a T is too small the low index can be moved up. When a T value is too large (it's possible to pick up all N people), then the high index can be moved up. This strategy can be applied to find the point at which any indication function transitions exactly once.

C.3 Ternary Search

Some problems require find the the minimum of a cup shaped or maximum of a cap shaped function. A ternary search can be used for such problems as long as the function is continuous and has only at most 1 time in which the function changes the sign of the slope (e.g. positive to negative or negative to positive).

The ternary search breaks the search space into 3 sections using 2 midpoints (whereas the binary search breaks the search space into 2 sections using 1 midpoint). The search area can be broken into 3 equal areas by using (low + low +high) / 3 as the first midpoint and (low + high + high) / 3 as the second midpoint. The function is then evaluated at these points. The values of the function at the midpoints can create a rough estimate of the slope of the function between the 2 midpoints (and using some calculus we know that some point in between those 2 points will have that slope.

If a minimum is looked for and the "slope" is positive, then the minimum could not be in the highest third of the search area, (since the line is increasing by the point of the second midpoint. If the "slope" is negative the minimum could not be in the lowest third of the search area.

TODO

This search will cut the search area by 2 thirds each iteration. The search area will decrease exponentially, which means that time it takes to reach a particular error window will be logarithmic. However, this method does converge slower than a binary search. Modifications on the width of the outer search areas could help the search converge faster.

C.4 Interpolation Search

The interpolation search is useful when the distribution of the data is known. The interpolation search is used by estimating the location of a target value using the known distribution. Then that location is used as the "midpoint" for splitting the search space. The interpolation search space has a good chance of finding the target value in only a few iterations, since the guess should be close to the target.

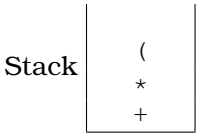
The easiest time for using interpolation search is when the data is uniformly distributed. When the data is uniformly distributed, a linear interpolation can be used. In the following picture the black line could represent the function to search on. The read dashed line shows the estimated function. The black dotted line shows the target value looked for. The red dotted line represent the guess (the intersection of the target and the estimated function).

TODO

D Answers

D.1 Stack Practice Question

Postfix Expression `5 3 * 2 4 2 1 / + 6 - * +`



Back to Question

D.2 Analysis Practice Questions

Question 1. Suppose we have the following loop,

```
for (int i = 0; i < n; i++)
{
   x += i;
}
```

How many operations are performed in the above loop with respect to n?

Every assignment, increment, comparison, and branch is a simple operation. The number of operations is about $4n + 2$. However, it would be approximated to $O(n)$ or more appropriately $\Theta(n)$.

Back to Question

Question 2. What is the simplified order approximation of $n^3 - 100n + 6$?

$\Theta(n^3)$

Back to Question

Question 3. Order the following functions from slowest growth (left) to fastest growth (right).

$$n^{2.5} \quad n! \quad \sqrt{3n} \quad 10n \quad \log_4(n^4) \quad 2^n$$

The order from slowest growth to fastest growth is the following,

Slowest					Fastest
$\log_4(n^4)$	$\sqrt{3n}$	$10n$	$n^{2.5}$	2^n	$n!$

Back to Question

Question 4. What is the closed form for the following summation?

$$\sum_{i=n}^{3n}(i^2+2)$$

$$=\sum_{i=n}^{3n}(i^2)+\sum_{i=n}^{3n}(2)$$

$$=\sum_{i=1}^{3n}(i^2)-\sum_{i=1}^{n-1}(i^2)+\sum_{i=1}^{3n}(2)-\sum_{i=1}^{n-1}(2)$$

$$=\sum_{i=1}^{3n}(i^2)-\sum_{i=1}^{n-1}(i^2)+2\sum_{i=1}^{3n}(1)-2\sum_{i=1}^{n-1}(1)$$

$$=\sum_{i=1}^{3n}(i^2)-\frac{(n-1)(n-1+1)(2(n-1)+1)}{6}+2(3n)-2(n-1)$$

$$=\sum_{i=1}^{3n}(i^2)-\frac{(n-1)(n)(2n-1)}{6}+2(3n)-2(n-1)$$

$$=\frac{(3n)(3n+1)(2(3n)+1)}{6}-\frac{(n-1)(n)(2n-1)}{6}+2(3n)-2(n-1)$$

Back To Question

Question 5. What is the runtime of the following segment of code in terms of input value N?

```
int ans = 0;
for (int i = 0; i < N; i++)
{
   ans+=i;
}
```

The runtime of the above segment is $\Theta(N)$, since the loop executes N times. There are N times a constant number of simple operations. In each loop there are only 3 operations performed,

1. `i < N` comparison
2. `i++` increment
3. `ans += i` increment

Back To Question

Question 6. What is the runtime of the following segment of code in terms of input value N?

```
for (int i = 0; i < N; i++)
{
   for (int j = 0; j * j < i; j++)
   {
      ans++;
   }
}
```

The runtime can be approximated using the following summation

$$\sum_{i=1}^{N}\sum_{j=1}^{\sqrt{i}}1$$

$$=\sum_{i=1}^{N}\sqrt{i}$$

There is not a closed form for the above summation. However, an upper and lower bound for that summation can be approximated.

The **lower bound** is approximated using smaller values. One way is to make the first $\frac{N}{2}$ values all 0, and the last $\frac{N}{2}$ values all $\sqrt{\frac{N}{2}}$. These work as lower bounds, because the square root function is strictly increasing. The resulting summation is,

$$\sum_{i=1}\frac{N}{2}0+\sum i=\frac{N}{2}N\sqrt{\frac{N}{2}}$$

$$=0\sum_{i=1}\frac{N}{2}1+\sqrt{\frac{N}{2}}\sum i=\frac{N}{2}N1$$

$$=0+\sqrt{\frac{N}{2}}\sum i=1\frac{N}{2}1$$

$$=\sqrt{\frac{N}{2}}\frac{N}{2}$$

$$\in\Omega(N\sqrt{N})$$

The **upper bound** is approximated using larger values. One way is to make all the values $\sqrt{N}$. Again this works because the square root function is strictly increasing. The resulting summation is,

$$\sum_{i=1}N\sqrt{N}$$

$$=\sqrt{N}\sum_{i=1}N1$$

$$=\sqrt{N}N$$

$$\in \mathrm{O}(N\sqrt{N})$$

The code's runtime is bounded above and below by the same approximation function—$N\sqrt{N}$—the code's runtime is in $\Theta(N\sqrt{N})$.

Back To Question

Question 7. What is the runtime of the following segment of code in terms of input value N?

```
for (int i = 0; i * i < N; i++)
{
   for (int j = 0; j < i; j++)
   {
      ans++;
   }
}
```

The runtime can be approximated using the following summation

$$\sum_{i=1}^{\sqrt{N}} \sum_{j=1}^{i} 1$$

$$= \sum_{i=1}^{\sqrt{N}} i$$

$$= \frac{\sqrt{N}(\sqrt{N}+1)}{2}$$

$$= \frac{\sqrt{N}^2 + \sqrt{N}}{2}$$

$$= \frac{N + \sqrt{N}}{2}$$

$$\in \Theta(N)$$

Much easier than dumb Question 6.
Back To Question

Question 8. What is the runtime of the following segment of code in terms of input value N?

```
for (int i = 1; i < N; i++)
{
   for (int j = i; j < N; j+=i)
   {
      ans++;
   }
}
```

The runtime of the above segment is $\Theta(N\log(N))$. It might look like N^2, but the number of times `j` executes is `N/i`, because `j` is increased by `i` in each loop. Thus the number of operations is on the order of the following summation

$$\frac{N}{1}+\frac{N}{2}+\frac{N}{3}+\frac{N}{4}+\frac{N}{5}+\frac{N}{6}+\frac{N}{7}+\frac{N}{8}+\frac{N}{9}+...+\frac{N}{N-1}+\frac{N}{N}$$

By pulling out a factor of N, the result is N times the N-th number of the Harmonic series. It was shown earlier that the Harmonic Series is logarithmic.

Back To Question

Question 9. For the following recursive function determine the recurrence relation expressing a tight order approximation of the number of operations, AND solve for the closed form of the recurrence relations and express the closed form in an order approximation.

```
int fun2 (int N, int * arr)
{
   if (N <= 1)
   {
      int res = 0;
      for (int i = 0; i < N; i++)
      {
         res += arr[i];
      }
      return res;
   }
   int ans = fun2(N / 2, arr);
   ans += fun2(N - (N / 2), arr + (N / 2));
   return ans;
}
```

The recurrence relations could be built in the following manner. The value of N is will determine how many recursive calls happen, because when N is 1 the function stops recursing. Even though there is a loop in the base case the loop is up to 1 which is a constant, so the base case runtime can be treated as a constant,

$$T(N) = ???$$
$$T(1) = 1$$

Note that $N - N/2$ is roughly $N/2$. The variable size is cut in half with each recursive call, because the the value of N that is passed into the recursive `fun` call is $N/2$,

$$T(N) = ??? \cdot T(N/2) + ???$$
$$T(1) = 1$$

The function is recursively called twice each recursion; there is only two `fun` call in `fun`, and that call is not in a loop,

$$T(N) = 2 \cdot T(N/2) + ???$$
$$T(1) = 1$$

The number of operations per recursive call is O(1), because there is no loop,

$$T(N) = 2 \cdot T(N/2) + 1$$
$$T(1) = 1$$

Below is an example of how back substitution can be used to solve for the closed form of the recurrence relation,
The original equation,

$$T(N) = 2 \cdot T(N/2) + 1$$

Substitute the adjusted parameter as N into the original recursive function and simplify,

$$T(N/2) = 2 \cdot T(N/2/2) + 1$$

$$T(N/2) = 2 \cdot T(N/4) + 1$$

Substitute $T(N/2)$ in the original recursive function with the and simplify a little bit,

$$T(N) = 2 \cdot (2 \cdot T(N/4) + 1) + 1$$

$$T(N) = 4 \cdot T(N/4) + 2 + 1$$

Substitute the new adjusted parameter as N into the original recursive function and simplify,

$$T(N/4) = 2 \cdot T(N/4/2) + 1$$

$$T(N/4) = 2 \cdot T(N/8) + 1$$

Substitute $T(N/4)$ into the original equation,

$$T(N) = 4 \cdot (2 \cdot T(N/8) + 1) + 2 + 1$$

$$T(N) = 8 \cdot T(N/8) + 4 + 2 + 1$$

Create a general form,

$$T(N) = 2^k \cdot T(\frac{N}{2^k}) + \sum_{i=0}^{k-1} 2^i$$

Find the terminating conditions

$$\frac{N}{2^k} = 1$$

$$N = 2^k$$

$$log_2(N) = k$$

Substitute in the terminating conditions

$$T(N) = N \cdot T(1) + \sum_{i=0}^{log_2(N)-1} 2^i$$

Simplify

$$T(N) = N \cdot 1 + \frac{2^{log_2(N)-1+1} - 1}{2 - 1}$$

$$T(N) = N + \frac{2^{log_2(N)} - 1}{1}$$

$$T(N) = N + 2^{log_2(N)} - 1$$

$$T(N) = N + N - 1$$

$$T(N) = 2N - 1$$

The order approximation is O(N).
Back To Question

Question 10. For the following recursive function determine the recurrence relation expressing a tight order approximation of the number of operations, AND solve for the closed form of the recurrence relations and express the closed form in an order approximation.

```
int fun2 (int N, int * arr)
{
   int res = 0;
   for (int i = 0; i < N; i++)
   {
      res += arr[i];
   }
   if (N <= 1)
   {
      return res;
   }
   res = fun2(N / 2, arr);
   res += fun2(N - (N / 2), arr + (N / 2));
   return res;
}
```

The recurrence relations could be built in the following manner. The value of N is will determine how many recursive calls happen, because when N is 1 the function stops recursing,

$$T(N) = ???$$
$$T(1) = 1$$

Note that $N - N/2$ is roughly $N/2$. The variable size is cut in half with each recursive call, because the the value of N that is passed into the recursive `fun` call is $N/2$,

$$T(N) = ??? \cdot T(N/2) + ???$$
$$T(1) = 1$$

The function is recursively called twice each recursion; there is only two `fun` call in `fun`, and that call is not in a loop,

$$T(N) = 2 \cdot T(N/2) + ???$$
$$T(1) = 1$$

The number of operations per recursive call is O(N), because there is a loop that executes N times,

$$T(N) = 2 \cdot T(N/2) + N$$
$$T(1) = 1$$

Below is an example of how back substitution can be used to solve for the closed form of the recurrence relation,
The original equation,

$$T(N) = 2 \cdot T(N/2) + N$$

Substitute the adjusted parameter as N into the original recursive function and simplify,

$$T(N/2) = 2 \cdot T(N/2/2) + \frac{N}{2}$$

$$T(N/2) = 2 \cdot T(N/4) + \frac{N}{2}$$

Substitute $T(N/2)$ in the original recursive function with the and simplify a little bit,

$$T(N) = 2 \cdot (2 \cdot T(N/4) + \frac{N}{2}) + N$$

$$T(N) = 4 \cdot T(N/4) + N + N$$

Substitute the new adjusted parameter as N into the original recursive function and simplify,

$$T(N/4) = 2 \cdot T(N/4/2) + \frac{N}{4}$$

$$T(N/4) = 2 \cdot T(N/8) + \frac{N}{4}$$

Substitute $T(N/4)$ into the original equation,

$$T(N) = 4 \cdot (2 \cdot T(N/8) + \frac{N}{4}) + N + N$$

$$T(N) = 8 \cdot T(N/8) + N + N + N$$

Create a general form,

$$T(N) = 2^k \cdot T(\frac{N}{2^k}) + \sum_{i=0}^{k-1} N$$

Find the terminating conditions

$$\frac{N}{2^k} = 1$$

$$N = 2^k$$

$$log_2(N) = k$$

Substitute in the terminating conditions

$$T(N) = N \cdot T(1) + \sum_{i=0}^{log_2(N)-1} N$$

Simplify

$$T(N) = N \cdot 1 + N \cdot \sum_{i=0}^{log_2(N)-1} 1$$

$$T(N) = N + N \cdot (log_2(N) - 1)$$

$$T(N) = N + N \cdot log_2(N) - N$$

$$T(N) = N \cdot log_2(N)$$

The order approximation is O($Nlog(N)$).
Back To Question

Question 11. What would be the best and worst case for performing a string compare on two strings each containing N characters?

- **Worst case** is O(N), because the strings could be equal, which requires looping through every character to check.
- **Best case** is O(1), because the first letter of the strings could be different, which requires looking at a single character, regardless of the value of N.

Back To Question

Question 12. What is the Order Approximation of the runtime of the following segment of code?

```
int i = 0, j = 0;
while (i < N)
{
   while (j < N)
   {
      ans+=i*j;
      j++;
   }
   i++;
}
```

The runtime is $\Theta(N)$, because the `j` is never reset in the loop. Once `j` reaches the value of `N` in the first iteration of `i`, the loop will never execute again.

Back To Question

Question 13. What is the runtime of the following segment of code?

```
for (int i = 1; i < N && i < M; i++)
{
   ans++;
}
```

The runtime for the above code snippet is $\Theta(\min(N, M))$.
Back To Question

Question 14. What is the runtime of the following segment of code?

```
for (int i = 1; i < N || i < M; i++)
{
   ans++;
}
```

The runtime for the above code snippet is $\Theta(\max(N, M))$ or $\Theta(N + M)$. A proof of why the two are equivalent is left as an exercise to the reader.

Back To Question

Binary Tree Question 1. Suppose a binary tree has K levels, what is the least and most number of nodes possible?

Maximum: First determine how many nodes will be on each level. The top (first) level will contain a single node—the root. To maximize the number of nodes every node of each level, but the last will have 2 children each. Thus there will be twice as many nodes on a particular level as the level above it.

$$Nodes(x) = 2Nodes(x-1)$$
$$Nodes(1) = 1$$

This is a recurrence relation with a relatively simple solution,

$$Nodes(x) = 2^{x-1}$$

The total number of nodes is

$$\sum_{i=1}^{K} Nodes(i)$$

By change of the incrementing variable we can get the following,

$$= \sum_{i=1-1}^{K-1} 2^{i+1-1}$$

$$= \sum_{i=0}^{K-1} 2^{i}$$

Using the geometric summation closed form

$$= \frac{2^{K-1+1} - 1}{2-1}$$

$$= \frac{2^{K} - 1}{1}$$

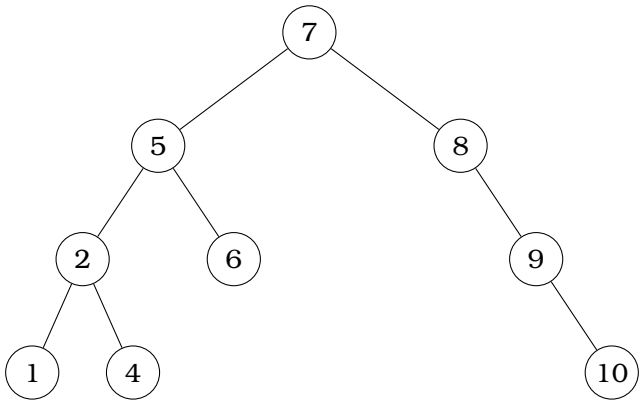
$$= 2^{K} - 1$$

Minimum: The least number of nodes per level would be one assuming we need exactly K levels. Since every level needs to be pointed to by a node on the previous level. The tree in question needs K levels, so there will be K nodes.

Back To Question

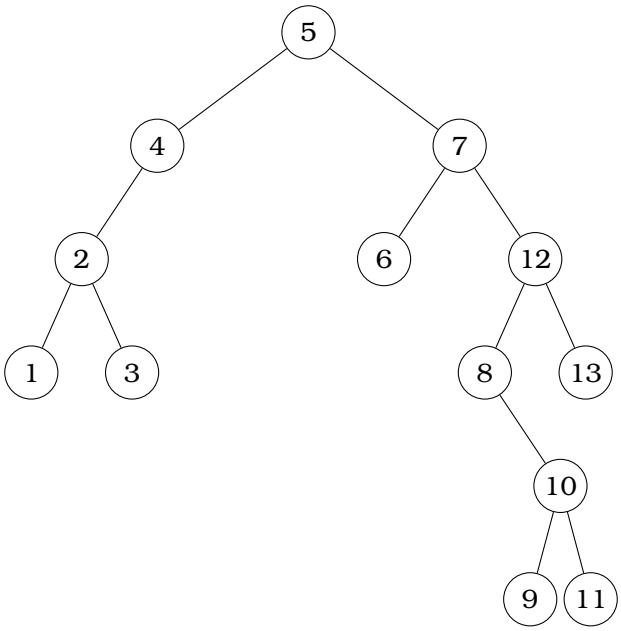
Binary Search Tree Question 1. Try inserting the following values into an empty BST and write draw the structure of the final tree.

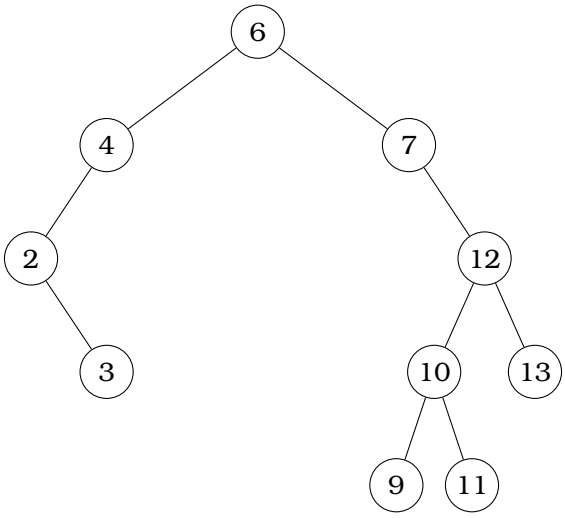
7, 8, 5, 9, 6, 2, 1, 4, 10



Back To Question

Binary Search Tree Question 2. What does the following tree look like after removing nodes 1, 5, and 8 in that order? When necessary use the successor.





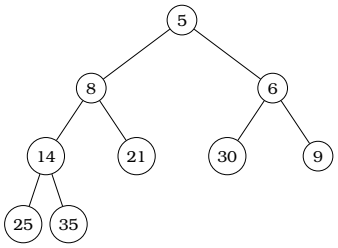
Back To Question

Binary Search Tree Question 3. What would be the best and worst case runtime approximation for inserting N nodes into an initially empty BST?

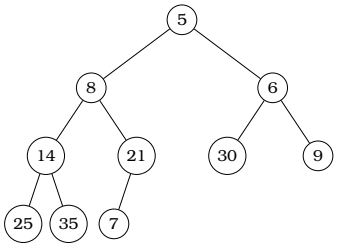
BST	Best	Worst
Insert N	$\Theta(Nlog(N))$	$\Theta(N^2)$

Back To Question

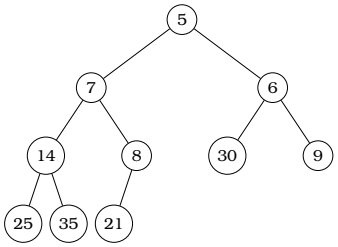
Heap Question 1 What is the resulting heap after inserting the value 7 into the following heap, where smaller values have higher priority?



After the initial insertion—but before the percolation—the tree should resemble the following,



After percolation the tree should look like the following,



Back To Question