Intro to Python

This course will introduce you to the Python programming language. Python is a high level programming language that is growing in popularity. As such, it is useful to know python. There are many versions of pythons. For our purposes, we will be using python version 3.8

Local installation

Read the first chapter in this book and get python installed on your computer based on your system. This book is available online from Seneca Library:

https://senecacollege.primo.exlibrisgroup.com/permalink/01SENC_INST/iq0dqm/alma997269205503226

For some assignments/labs we will make use of github's codespace which will run a virtual machine in the browser. No setup will be needed for you when you make use of codespace. All you will need is a modern web browser. You may wish to have a way to run code spaces anyways.

The Basics

This section looks at the basics of Python programming.

Interpreted vs Compiled

Python is both interpreted and compiled but these terms are complicated. JavaScript is an example of an interpreted language, where the source gets turned into machine language as it runs. C/C++ are examples of languages that are compiled. When you run a c++ compiler, the program doesn't actually run, instead it translates your c++ code into executable that you can then run.

However Python is a little different. You can run a Python program without ever compiling it. You can also compile a Python program but what you get isn't an executable in the C/C++ sense which is compiled for your machine. Instead, you get the program translated into bytecode. When you run the compiled program, that byte code is translated into your computer's machine language.

The Python Shell

The Python shell allows you to run python code without saving it to a file. It can be a quick way for you to test out some snippet of code. You can spin up a python shell by typing python at the command prompt

Comments

Use # at start of line to create a comment

```
# this is a comment
```

Basic Types and Variables

In C, when you declare a variable you have to state the data type of that variable. You think about the variable as a name for an area of memory that can hold stuff.

Unlike C, it is better to think of variables in python as labels. All variables have a value association with it, you don't just go around declaring storage like you do in C.

Naming rules:

Here are the naming rules:

- you can use letters, numbers and underscores in names
- variables names begin with letters and underscores, never numbers
- spacing is not allowed in variable names
- avoid python keywords for variable names

Naming Conventions:

- use snake case `this_is_a_variable_name`
- use lower case for variable names
- use ALLCAPS for constants.
- capitalize first word of class names without using _ between ThisIsAClassName

Strings

Strings are created by using either double quotes or single quotes:

```
"this is a string"
'this is also a string'
```

Since both are strings, the one you use could depend on what you want to say. For example, if your statement involves an apostrophes, use double quotes

```
"Python's string declarations are neato!"
```

Similarly you if you need to use double quotes in your string, you can use single quotes to declare it

```
'"Coffee is a way of stealing time that should by rights belong
to your older self" - Terry Pratchett'
```

If you need both.. you will need to use the escape \ character

```
"\"It's still magic even if you know how it's done\" - Terry
Pratchett"
```

Numbers

Just like C/C++ you can have integers or floating point values. Unlike C/C++ you don't specify the number of bits used to represent the numbers.

To create an integer, write a number without decimal points

```
x = 5
```

x is an integer

To create a floating point add a decimal point

```
y = 5.0
```

y is a floating point

Numeric Operators

The following mathematical operators are available

- **+** - addtion
- **-** - subtraction
- ***** - multiplication
- **/** - division
- **//** - integer division
- **%** - modulus
- ****** - exponent

Mathematical expressions

Mathematical expressions follow most of the same rules as C/C++ but there are some differences:

- when using dividing 2 integers, C/C++ truncates the decimal points 14/5 = 2 for example. Python results in the floating point result of the division.

14/5 = 2.8

- the modulo operation % following proper mathematical mod rules when dealing with negative values. In C/C++ it just negates the results. thus -9 % 5 = -4. In python, -9 % 5 = 1, because -9/5 = -1.8, taking the floor of this gives -2. -2 * 5 + 1 = -9
- you can use _ to make it easier to specify the number `x = 1_000_000_000` for example to specify 1 billion.

Selection

In general python selection is done using if/else type of statements. Pre python 3.10 there are no equivalency of a switch statement. It is recommended that you keep to the if/else type of construct for selection.

In Python, indentation really matters. there are no brackets. If you want something to be part of an if block, make sure they are indented the same way inside the if statement.

For example, in C if you wrote:

```
x=5;
if(x == 3)
    printf("bananas\n");
    printf("oranges\n");
```

the result output would be:

```
oranges
```

the indentation doesn't matter. You would normally wrap it all up using {} if

you wanted to have both statements to be done when x == 3.

In python there are no brackets. Instead, if you indent both statements then they are part of the if block. If you don't indent the second statement then it is not.

Boolean expressions

Boolean expressions are thing that evaluate to True or False. Values of basic types can be compared using ==, !=, <, <=, >, >=. For strings, the comparison is alphabetical ordering, thus "apple" < "banana", but "Banana" < "apple" (because "B" comes before "a")

Boolean operators:

In C/C++ the boolean operators &&, ||, ! are replaced by **and**, **or** , **not**

zero vs non-zero

zero is false, everything else is true. This is same as C/C++

Lazy Evaluation

Similar to C/C++, the boolean operators are lazily tested. This means that for a boolean expression that uses an **and** expression, if the left operand is false, the right operand will not be evaluated because the entire boolean expression can never be true

Similarly when given an expression that uses an **or** expression, if the left operand is true, the right operand will not be evaluated because the entire boolean expression will never be false

```
def truthy():
```

results in:

```
true or false result:
it's true!

false or true result:
it's false!
it's true!

true and false result:
it's true!
it's false!

false and true result:
it's false!
```

Syntax:

An if statement

```
if <boolean expression>:
    statement1
    statement2
    ...
```

if/else statement

```
if <boolean expression>:
    statement1
    statement2
    ...
else:
    statement3
```

if/else if/else if/...else

```python
if <boolean expression>:
    statement1
    statement2
    ...
elif <boolean expression>:
    statement3
    statement4
    ...
elif <boolean expression>:
    statement5
    statement6
    ...
...
else:
    statement7
    statement8
    ...
```

Lists

This is similar in nature to C++ arrays but also quite different at the same time.

Unlike C++ arrays, when you declare a list, you do not specify a capacity. As you add items, the list will grow as needed.

To use a list, declare a variable and initialize it with a list of values:

```python
my_list = [0, 2 , 3]
```

The other major difference between lists in Python and arrays in C++ is that items in Python lists can be of different types. For example, in C++, you cannot store integers and strings in the different elements of the same array. Every element must be of the same type.

In Python, every element in the list can have a different type.

```python
my_list = [1 , 2 , "hello", "world", 1.5]
```

To access a particular element, you index into the list. Indexing starts at 0 just like in C

```python
my_list = [1 , 2 , "hello", "world", 1.5]
print(my_list[3])
```

This above code will output:

```
world
```

list assignments

When you assign a variable list to another list variable, duplicates are not made, instead they are simply different names refering to the same list. This is similar to how we might think about array assignments in C/C++

```python
list1 = [1,2,3,4]
list2 = list1
list2[1] = 50
print(list1)  # output: [1, 50, 3, 4]
```

both list1 and list2 refer to the same list so any changes made through one list, the other list is also modified

Dictionaries

A dictionary is a set of key-value pairs. To create a dictionary. Syntactically it resembles the definition of JavaScript objects. The keys can be any basic types, strings or numbers

```python
my_dictionary = {<key1>:<value>, <key2>:<value>...}
print(my_dictionary)
```

A dictionary lets you find values based on keys quickly.

```python
my_dictionary = {"key1": 5, "key2":6, 7:15}
print(my_dictionary["key1"])  # output 5
```

You can add to a dictionary by assignment

```python
my_dictionary = {"key1": 5, "key2":6}
my_dictionary["key3"] = 7
print(my_dictionary["key3"])  # output 7
```

Iteration

For Loops

A for loop is a counting loop in python.

```python
for variable in range(start,end):
    ...
```

The above for loop is equivalent to the following in C/C++

```c
for(int variable = start;variable < end; variable++){
    ...
}
```

Furthermore, you can also use it to iterate through lists:

```python
my_list=[5, 4, 3, 2, 1]

for number in my_list:
    print(number)
```

While Loops

```python
i = 1

while i < 10:
  print(i)
```

A while loop can be very useful when there is no clear end… for example, if you were trying to do interactive input until some criteria is met.

Functions

To declare a function in python use the following syntax

```python
def function_name(argument_list):
    ...
```

You can call the function by using the function name and supplying arguments to the function

Classes

Classes allow you to create objects in python. There are however some subtle differences in python

In python classes there is a special function that is used to initialize and define the data within the class. So looking at code below, you will notice that the data members are effectively declared in a special function called `__init__`

The function is named init with two underscores before and after the word. The keyword self is the first argument followed by an argument list.

```python
class ClassName:
    def __init__(self, argument_list):
        self.attribute1 = ...   # this is the equivalent of data members
        self.attribute2 = ...   # in c++
```

You can create an inherited class in python by passing the base class into the class declaration of the derived class.

```python
class BaseClass:
    def __init__(self, argument_list):
        self.attribute1 = ...
        self.attribute2 = ...

    def function1(self, arguments):
        ...

class DerivedClass(BaseClass):
    def __init__(self, argument_list):

super().__init__(arguments_for_attribute1_and_attribute2)    # notice you don't use self in this call
        self.attribute3 = ...

    def function1(self, arguments):
        ... # overrides base class function

    def function2(self, arguments):
        ... # new function exclusive to derived class
```

Algorithms Analysis

What is Algorithms Analysis

When you write a program or subprogram you should be concerned about the resource needs of the program. The two main resources to consider are time and memory. These are separate resources and depending on the situation, you may end up choosing an algorithm that uses more of one resource in order to use less of the other. Understanding this will allow you to produce better code. The resource to optimize for depends on the application and the computing system. Does the program need to finish execution within a restricted amount of time? Does the system have a limited amount of memory? There may not be one correct choice. It is important to understand the pros and cons of each algorithm and data structure for the application at hand.

The amount of resources consumed often depends on the amount of data you have. Intuitively, it makes sense that if you have more data you will need more space to store the data. It will also take more time for an algorithm to run.

> 💡 **TIP**
>
> Algorithms Anaylsis does not answer the question "How much of a resource is consumed to process *n* pieces of data"... the real question it answers is "How much **more** of the same resource will it consume to process n+1 pieces of data"

In other words what we really care about is the growth rate of resource consumption with respect to the data size.

And with this in mind, let us now consider the growth rates of certain functions.

Measuring Resource Consumption

Analysis of algorithms is about measuring the growth of resource consumption as data size increases. Resources can be anything that has a limited supply. The top two are time and memory but these are not the only ones.

Time Resource

Each operation that your program performs requires a small slice of time... time to load the instruction, time to perform the operation, time to store the result... The more operations you do, the more time it will take. Thus, one way that we can measure the amount of time required by an algorithm is to measure how many operations it performs.

When doing this, we make the assumption that every operation has the same time cost. Given this assumption, addition and division would take the same amount of time. This is not the case in reality but it is good enough for analysis.

Memory Resource

The other big resource is memory. This is not calculated by operation count because the number of operations doesn't always increase memory consumption. However, we can still make a calculation on this based on variable declarations, dynamically allocated memory etc.

Measuring Resources

To perform an analysis, the first thing we do is figure out mathematically how much resources we consume. However, as stated in the introduction, this is not a number... it is a mathematical expression that typically involves a variable for the amount of data involved. This is because we can expect that the more data we have the more resources get consumed... it takes more time to process more data, it takes more memory to store more data. The real question is really how much more.

Once we have this expression, we can then look at how the resource consumption grows with respect to the size of the data.

Growth Rates

Growth Rates

Algorithms analysis is all about understanding growth rates. It is about understanding the growth in resource consumption as the amount of data increases. Typically, we describe the resource consumption growth rate of a piece of code in terms of a function (a "curve"). To help understand the implications, this section will look at graphs for some common growth rates from most efficient to least efficient.

Constant

A constant resource need is one where the resource need does not grow. That is processing 1 piece of data takes the same amount of resource as processing 1 million pieces of data. The graph of such a growth rate looks like a horizontal line

Constant Curve (y = 1)

Resource Consumption

Data Size (n)
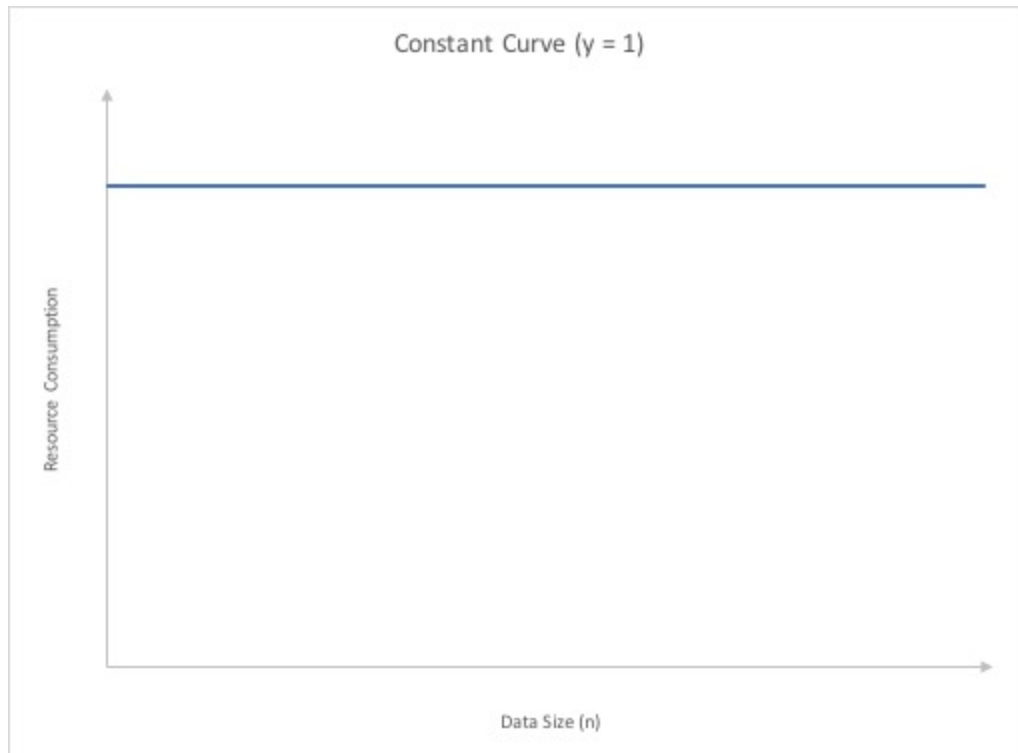
Logarithmic

A logrithmic growth rate is a growth rate where the resource needs grows by one unit each time the data is doubled. This effectively means that as the amount of data gets bigger, the curve describing the growth rate gets flatter (closer to horizontal but never reaching it). The following graph shows what a curve of this nature would look like.

Logarithmic Curve (y = logn)

Resource Consumption

Data Size (n)
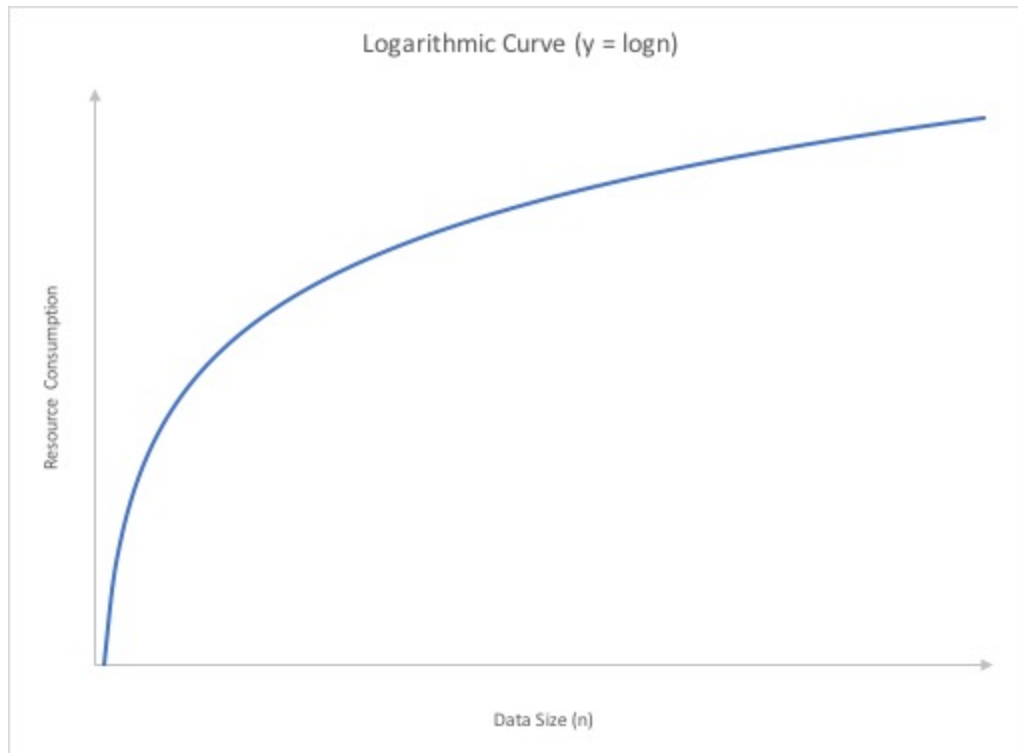
Linear

A linear growth rate is a growth rate where the resource needs and the amount of data is directly proportional to each other. That is the growth rate can be described as a straight line that is not horizontal.

Linear Curve (y = n)
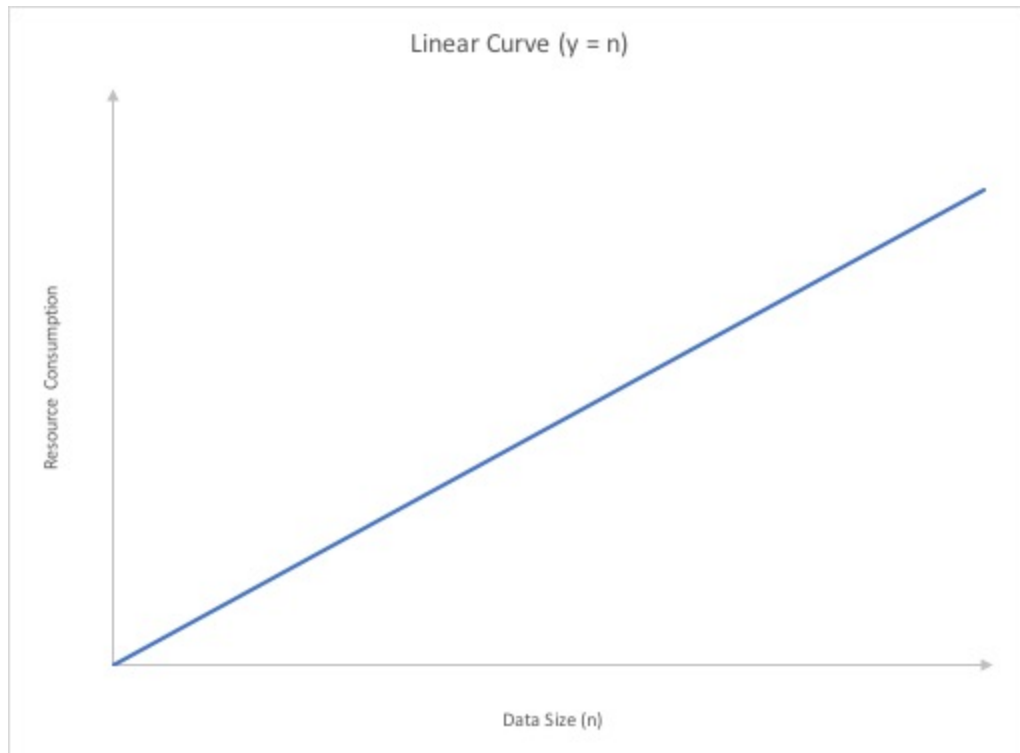
Loglinear

A loglinear growth rate is a slightly curved line. the curve is more pronounced for lower values than higher ones

Loglinear Curve (y = n log n)
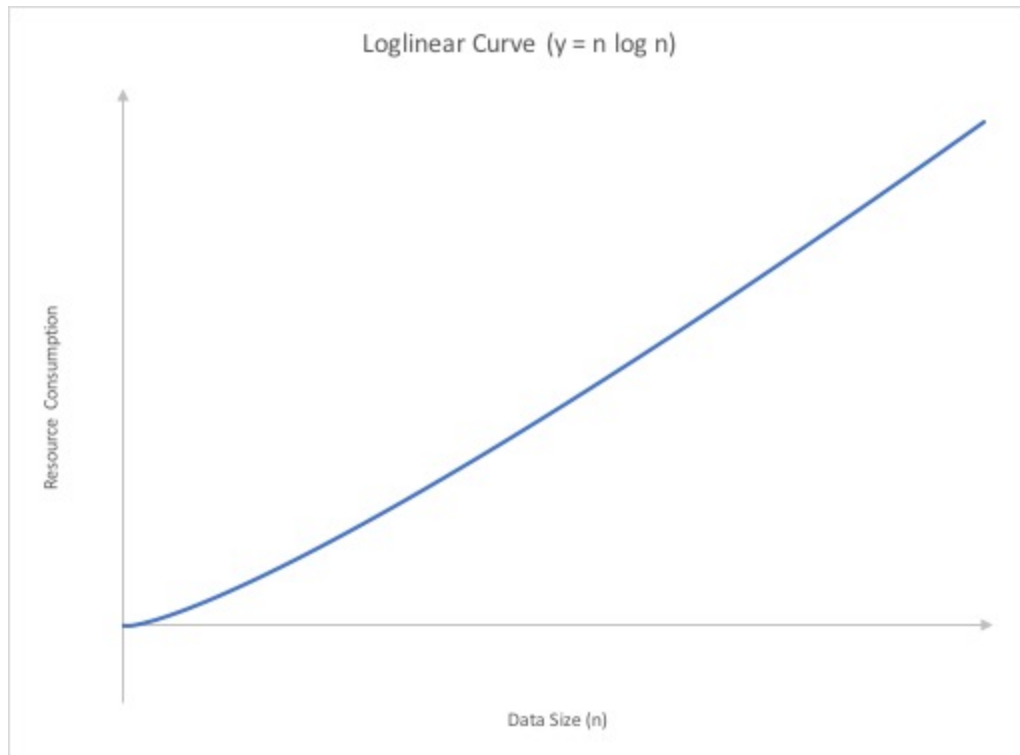
Quadratic

A quadratic growth rate is one that can be described by a parabola.

Quadratic Curve ($y = n^2$)
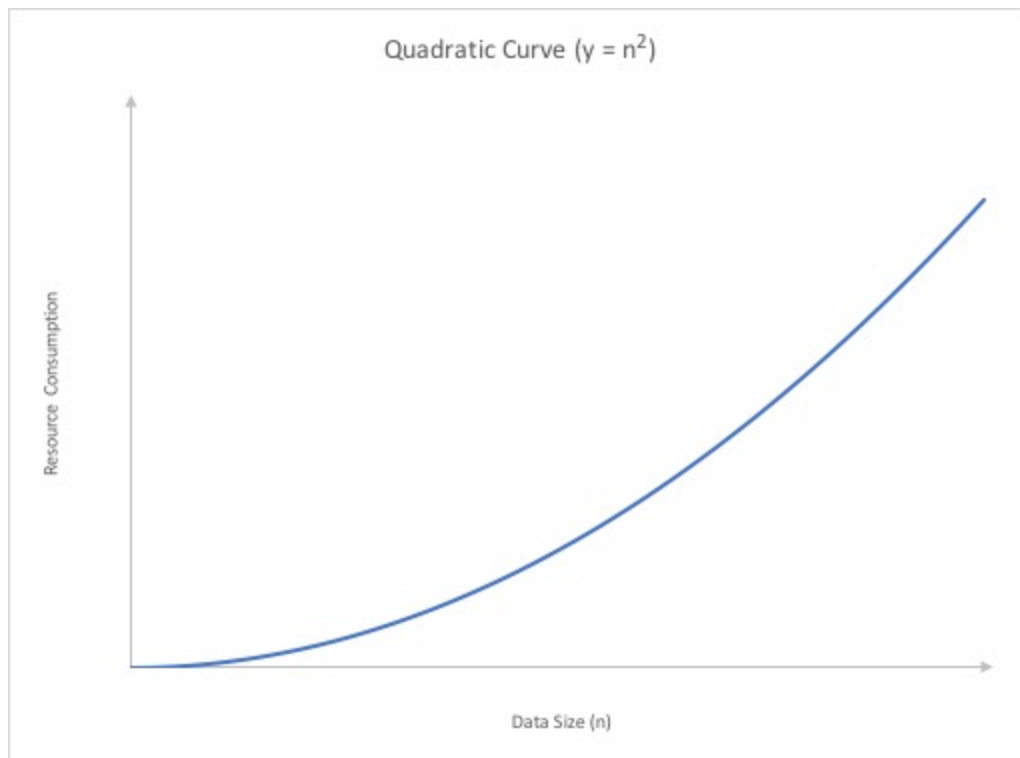
Cubic

While this may look very similar to the quadratic curve, it grows significantly faster

Cubic Curve $(y = n^3)$

Resource Consumption

Data Size (n)
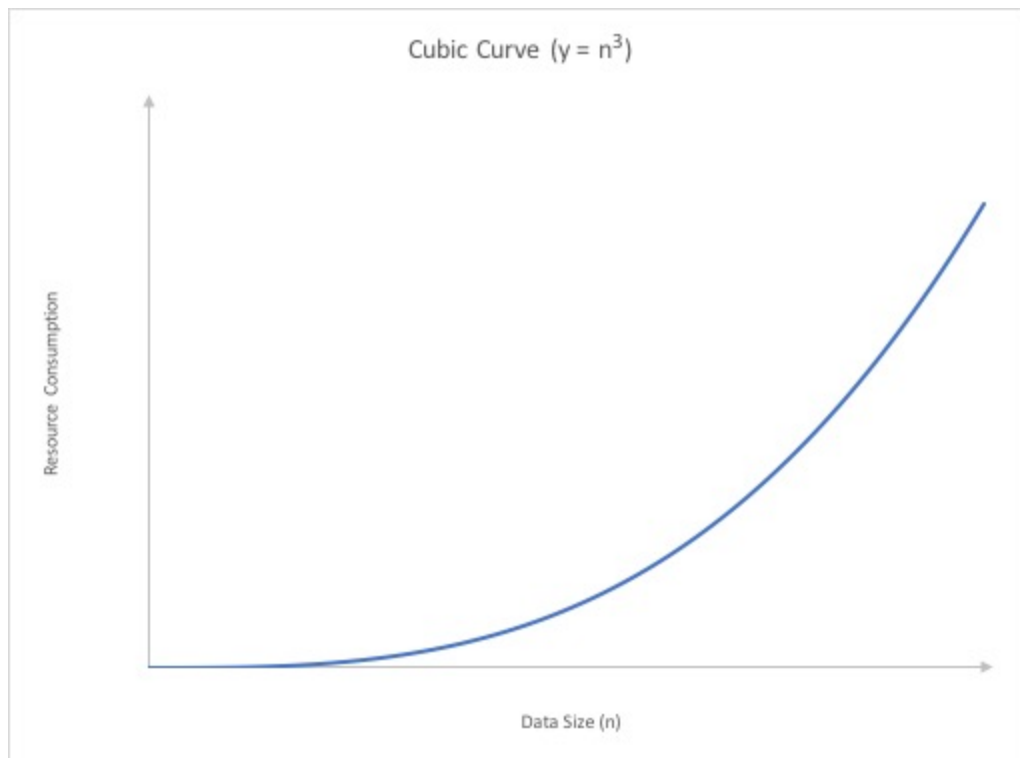
Exponential

An exponential growth rate is one where each extra unit of data requires a doubling of resource. As you can see the growth rate starts off looking like it is flat but quickly shoots up to near vertical (note that it can't actually be vertical)

Exponential Curve ($y = 2^n$)

Resource Consumption

Data Size (n)
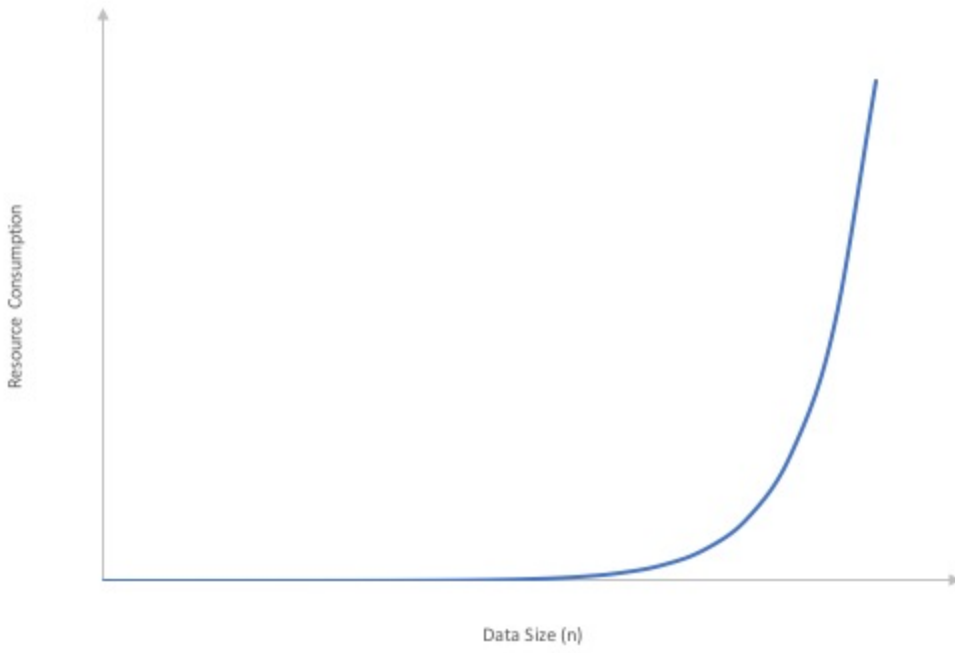
Asymptotic Notation

Asymptotic notation are formal notational methods for stating the upper and lower bounds of a function. They are the notations used when describing resource needs. These are: $O(f(n))$, $o(f(n))$, $\Omega(f(n))$, and $\Theta(f(n))$ (Pronounced, Big-O, Little-O, Omega and Theta respectively). The f(n) inside each of these notations is a description of a curve like we saw in the previous section. They describe the shape of a line.

Formally

"$T(n)$ is $O(f(n))$" iff for some constants c and n_0, $T(n) \leq cf(n)$ for all $n \geq n_0$

"$T(n)$ is $\Omega(f(n))$" iff for some constants c and n_0, $T(n) \geq cf(n)$ for all $n \geq n_0$

"$T(n)$ is $\Theta(f(n))$" iff $T(n)$ is $O(f(n))$ and $T(n)$ is $\Omega(f(n))$

"$T(n)$ is $o(f(n))$" iff $T(n)$ is $O(f(n))$ and $T(n)$ is NOT $\Theta(f(n))$

> 💡 **TIP**
>
> iff is the mathematical shorthand for "if and only if"

Informally

"$T(n)$ is $O(f(n))$" basically means that the function $f(n)$ describes the upper bound for $T(n)$

"$T(n)$ is $\Omega(f(n))$" basically means that the function $f(n)$ describes the lower

bound for $T(n)$

"$T(n)$ is $\Theta(f(n))$" basically means that the function $f(n)$ describes the exact bound for $T(n)$

"$T(n)$ is $o(f(n))$" basically means that the function $f(n)$ describes the upper bound for $T(n)$ where $f(n)$ will never actually be reached
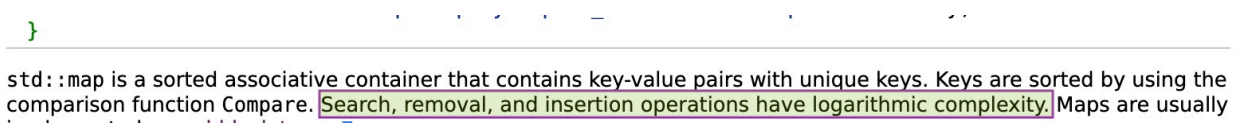
Worst case, Best Case, Average Case

The asymptotic notation system for bounds is often confused with the idea of worst case, best case and average case. They are actually very different things. Big-O does not describe a worst case, Omega does not describe a best case. Big-O describes an upper bound on each of these cases. Similarly Omega describes a lower bound on each of these cases. We should not mix up this idea.

In practice, most of the time we deal with either worst case or average case. We rarely ever consider best case. Each case can still have an upper bound(big-O) or lower bound (Omega).

An easy way to think about Asymptotic Notation

The math in algorithms analysis can be intimidating. One of the simplest ways to think about algorithms analysis is that it is simply a way to apply a rating system for your algorithms. It is like a audience rating for movies (G, PG, AA etc.). In movies these ratings tell you whether or not the movie was intended to for audieces of varying ages without you having to watch the movie first.

Similarly the resource consumption functions tells you the expected resource consumption rate without you having to look at the actual code. It tells you the kind of resource needs you can expect the algorithm to exhibit as your data gets bigger and bigger. From lowest resource growth to highest resource growth, the rankings are: $O(1)$, $O(\log n)$, $O(n)$, $O(n\log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$. References will sometimes state the expected resource consumption. For example, here is a screenshot of the reference page on c++ maps:

```
    }
```

std::map is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison function Compare. Search, removal, and insertion operations have logarithmic complexity. Maps are usually

Notice the highlighted section in the above image. "...Search, removal, and insertion operations have logarithmic complexity....". This statement means the as you add more and more data to a map, the amount of time it takes to perform search/remove/insert will grow in a logarithmic manner. That is, it will take a doubling of the amount of data stored to increase the resource consumption by 1. Effectively if we were to somehow calculate the time it takes to do these operations as data gets added to the map, we will find that it will grow like the curve below:

Logarithmic Curve (y = logn)
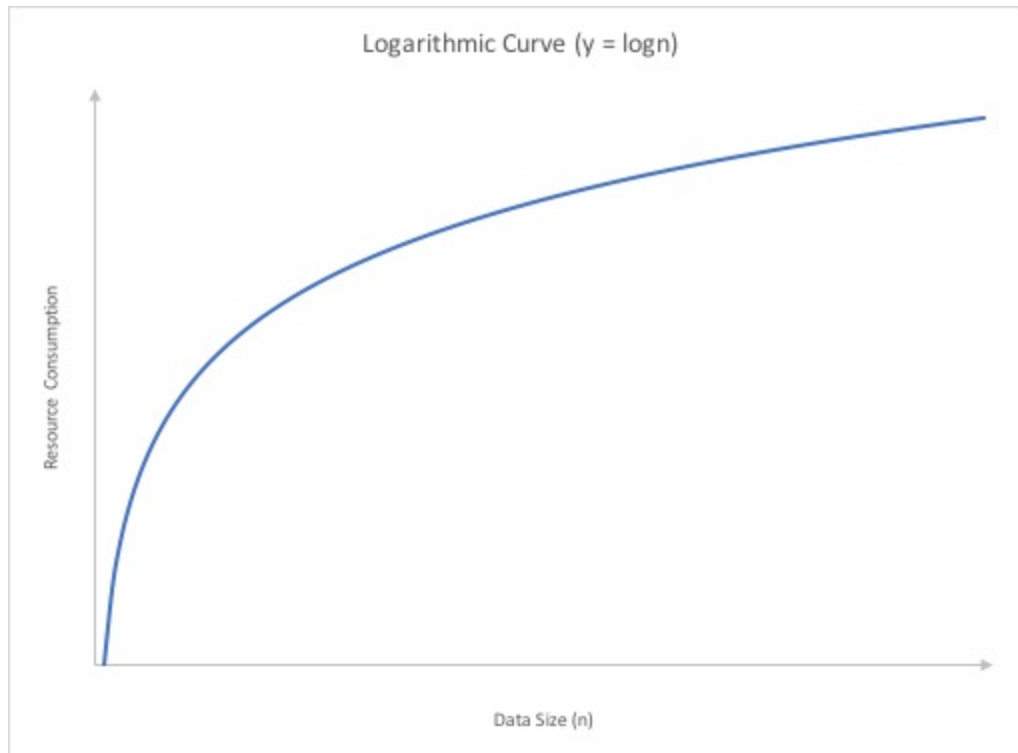
Think about the graphs in the grow rate section. The way each curve looks. Recognizing the impact that this will have is the most important thing to learn about analyzing. That is the most important thing to understand about algorithms analysis.

A closer look at the definition of big-O

Mathematical definitions are typically exact. If you say that a < b then a is always smaller than b... there isn't any room for anything else. What asymptotic notations used in algorithms analysis do is relax this exactness. It allows us to get an idea of resource usage without having to be completely exact about it.

Let's take a closer look a the formal definition for big-O analysis

"$T(n)$ is $O(f(n))$" iff for some constants c and n_0, $T(n) \leq cf(n)$ for all $n \geq n_0$

Start with the first portion of the statement

The above is portion is stating what we are defining. In other words, that part says: In order for $T(n)$ to be $O(f(n))$ it must meet the criteria set out in the rest of the definition. It also gives you an idea of what pieces are involved with the definition

- **n** is the size of the data set.
- **T(n)** is a function that describes the usage of a resource with respect to **n**
- **f(n)** is a function that describes a curve...$logn$, n, n^2, etc.
- **O(f(n))** means that the curve described by **f(n)** is an upper bound for the resource needs of a function **T(n)**

The next part of the definition:

c and n_0 refer to two values that are both constants. constants are values that do not change with respect to n. No matter what n is, c and n_0 must stay the same. The exact values of c and n_0 are not defined, but in order for the definition to be true, we must be able to find a value for each of these constants such that the rest of the statement is true. Note that it doesn't matter what they are.. you can pick any value you want for these constants... as long as the statement holds using them, you are fine.

This means that if we were to plot $T(n)$ on a graph, the $T(n)$ curve would fall underneath some stretched (scaled by a constant) version of the $f(n)$ curve. What this means is that $T(n)$ does not have to be under the specific curve described by $f(n)$. It is allowed to fall under a constantly scaled version of the $f(n)$. For example instead of having to be under an exact curve like n^2 , it can be under the $10n^2$ curve or even the $200n^2$ curve. In fact it can be any constant, as long as it is a constant. A constant is simply a number that does not change with n . So as n gets bigger, you cannot change what the value for c . The actual value of the constant does not matter.

The last portion of the definition is:

This is another relaxation of the mathematical definition. It means part of the $T(n)$ curve is allowed to actually be above the $cf(n)$ curve. $T(n)$ simply fall under the $cf(n)$ curve at some value of n (we call this point n_0). and once it falls under the $cf(n)$ curve, it cannot go back above the curve for any value of n greater than n_0

In summary, when we are looking at big-O, we are in essence looking for a description of the growth rate of the resource increase. The exact numbers do not actually matter in the end.

Example 1: Analysis of Linear Search

To look at how to perform analysis, we will start with a performance analysis of the following C++ function for a linear search:

Python   C++

```python
def linear_search(my_list, key):
    for i in range(0, len(my_list)):
        if my_list[i] == key:
            return i

    return -1
```

```cpp
template <class TYPE>
int linearSearch(const vector<TYPE>& arr, const TYPE& key){
    int rc=-1;
    for(int i=0;i<arr.size()&& rc==-1;i++){
        if(arr[i]==key){
            rc=i;
        }
    }
    return rc;
}
```

Assumptions

Looking at the C++ code (the Python one has been described a bit differently

but has similar functionality,) we can see that there is a loop. The loop stops on one of two conditions, it reaches the end of the array or rc becomes something other than -1. rc changes only if key is found in the array. So, what are the possibilities when I run this program?

1) key is not in the array... in that case loop stops when it gets to the end of the array

2) key is in the array... but where? If it is at the front, then loop stops right away, if it is in the back then it needs to go through all the elements... clearly these will both affect how many operations we carry out.

The worst case scenario is that of an unsuccessful search... the loop will need to run more times for that. If the search was successful however, where will the key be? To perform an average case scenario, we need to consider the likelihood of finding the key in each element. Now, making the assumption that the key is equally likely to be in any element of the array, we should expect to go through half the array on average to find key.

We will look at each of these in turn below

Analysis of an Unsuccessful Search

Let n represent the size of the array arr.
Let $T(n)$ represent the number of operations necessary to perform linear search on an array of n items.

Next we go through the code and count how many operations are done. We can do this by simply adding a comment to each line with the **total** operation count per line. That is we count how many operations are on that line **AND** the number of times that particularly line is done.

```python
def linear_search(my_list, key):
    for i in range(0, len(my_list)):     # n + 2
        if my_list[i] == key:            # n
            return i                     # 0

    return -1                            # 1
```

> **⚠ CAUTION**
>
> In python3, range() and len() are both constant. Further, range() calls are done only once to determine the range() of values used for looping. This is not the same as testing the continuation condition in C/C++ which happens at top of every loop. Here, you loop through the entire range of values

Where do the counts come from?

- Firstly, n is the length of the list.

```python
def linear_search(my_list, key):
    for i in range(0, len(my_list)):    # n + 2 - reason:  you
change the value of i each time you go through loop.
                                        # This change is done for
every value between from 0 to n-1 inclusive.
                                        # Thus, i is changed n
times in total. len() function call is constant,
                                        # range() in python3 is
also constant.  Thus, we count those items as 1
                                        # each.  range() is only
called once to get all the values loop must
```

Come up with the expression for T(n)

Recall that T(n) represents the total number of operations needed for an unsuccessful linear search. As such, we get T(n) by summing up the number of operations. Thus:

$$T(n) = n + 2 + n + 1$$
$$= 2n + 3$$

The $2n$ comes from the 2 operations that we do each time through the loop (note that we have *n* elements in the list/array.) The $+3$ comes from the 3 operations that we always have to do no matter what (the loop runs n times and causes $2n + 2$ operations.)

Thus the expression for the number of operations that is performed is:

$$T(n) = 2n + 3$$

Now, imagine n becoming a very very large number. As n gets bigger and bigger, the 3 matters less and less. Thus, in the end we only care about $2n$. $2n$ is the **_dominating term_** of the polynomial. This is similar to this idea. If you had 2 dollars, then adding 3 dollars is a significant increase to the amount of money you have. However, if you had 2 million dollars, then 3 dollars matter very little. Similarly, in order to find something in an array with 2 million elements by comparing against each of them, doing just 3 ops we need to do to find the range, length of list, and return matters very little.

Using the dominating term, we can say that the linear search algorithm has a run time that never exceeds the curve for n. In other words, linear search is $O(n)$

Now... can we actually prove this?

According to the formal definition for big-O

" $T(n)$ is $O(f(n))$ " if for some constants c and n_0, $T(n) \leq cf(n)$ for all $n \geq n_0$

In other words... to prove that $T(n) = 2n + 3$ is $O(n)$ we must find 2 constants c and n_0 such that the statement $T(n) \leq cn$ is true for all $n \geq n_0$

The important part about this... is to realize that we can pick any constant we want as long as it meets the criteria. For c , I can pick any number > 2. I will pick 3 (there isn't a good reason for this other than 3>2), thus c = 3. I will also pick 3 for n_0. The reason is that starting at n == 3, $2n + 3 \leq 3n$

We can see this in the graph below. Here, we see $T(n) = 2n + 3$ (blue) as well as the $cf(n) = 3n$ (orange). At n==3, the line for $2n + 3$ falls under $3n$ and it never gets above it again as n gets bigger and bigger
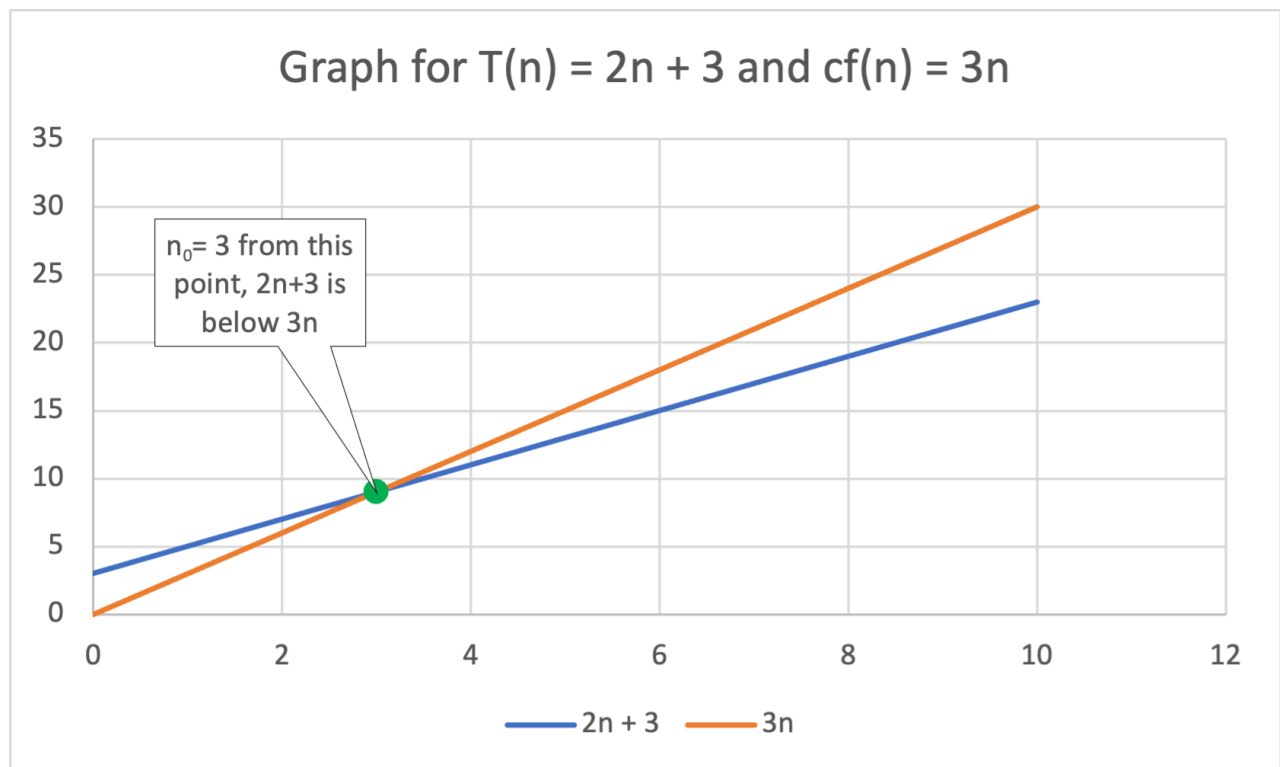
Graph for T(n) = 2n + 3 and cf(n) = 3n

Thus, we have proven that the function is $O(n)$ because we were able to find the two constants c and n_0 needed for the $T(n)$ to be $O(n)$

This is also the reason why it did not matter if we counted the [i] operator. If we counted that operator, our function would be : $T(n) = 3n + 3$.

The dominating term would be 3n. We would need to pick a different c that is greater than 3 and a different n_0 but we would still be able to find these and thus, T(n) is still O(n).

Analysis of a Successful Search

In the previous analysis, we assumed we wouldn't find the data and thus the worst possible case of searching through the entire array was used to do the analysis. However, what if the item was in the array?

Assuming that key is equally likely to be in any element, we would have to search through n elements at worst (finding it at the last element) and $\frac{n}{2}$ elements on average.

Now since we will at some point find the item, the statement inside the if will also be performed while the return -1 will not.

If we assume that the worst case occurs and we do not find the item until we get to the last element our operation count would be:

```
def linear_search(my_list, key):
    for i in range(0, len(my_list)):    # n + 2
        if my_list[i] == key:           # n
            return i                     # 1

    return -1                            # 0
```

When we add up our counts, we would end up with the same function.

If we assume we will only search half the list.. then what would we get?

```python
def linear_search(my_list, key):
    for i in range(0, len(my_list)):    # n/2 + 2
        if my_list[i] == key:           # n/2
            return i                    # 1

    return -1                           # 0
```

$T(n) = n/2 + 2 + n/2 + 1 = n + 3$

In this case, the dominating term is n. and thus T(n) is still O(n)

```cpp
template <class TYPE>
int linearSearch(const vector<TYPE>& arr, const TYPE& key){
    int rc=-1;                                  //1
    for(int i=0;i<arr.size()&& rc==-1;i++){     //1 + 5n
        if(arr[i]==key){                        //n
            rc=i;                               //0
        }
    }
    return rc;                                  //1
}
```

> ⚠️ **CAUTION**
>
> In the above code we are treating arr.size() function call as if its a constant value or variable. This can be done only because the size() function in vector class is constant. You can check this out here: https://en.cppreference.com/w/cpp/container/vector/size under the heading complexity. If the complexity was something else, we would need

Where do the counts come from?

- `int rc=-1;` is only ever done once per function call, there is one operator on that line, so the count is 1
- `for(int i=0;i<arr.size()&& rc==-1;i++)` int i=0 is only done once per function call, so it is counted as 1, the other operations <, . (dot), &&, == and ++ happen for each loop iterations and since the loop happens n times, each of those 5 operations are multiplied by n
- `if(arr[i]==key)` this is done every time we are in loop, loop runs n times, so count as 1
- `rc=i` it is worth noting that because we are assuming our search is unsuccessful, the if statement never evaluates to true, thus, this line of code never runs. Thus, it is not counted
- `return rc;` like the first initialization statement this return statement only happens once, so we count it as 1

Come up with the expression for T(n)

Recall that T(n) represents the total number of operations needed for an unsuccessful linear search. As such, we get T(n) by summing up the number of operations. Thus:

$$T(n) = 1 + 1 + 5n + n + 1$$
$$= 6n + 3$$

The $6n$ comes from the 6 operations that we do each time through the loop. We have n elements in the array. Thus, the loop must run $6n$ times.

The $+3$ comes from the 3 operations that we always have to do no matter what.

Thus the expression for the nu-mber of operations that is performed is:

$$T(n) = 6n + 3$$

Now, imagine n becoming a very very large number. As n gets bigger and bigger, the 3 matters less and less. Thus, in the end we only care about $6n$. $6n$ is the **dominating term** of the polynomial. This is similar to this idea. If you had 6 dollars, then adding 3 dollars is a significant increase to the amount of money you have. However, if you had 6 million dollars, then 3 dollars matter very little. Similarly if you had to find something in an array with 6 million elements

Using the dominating term, we can say that the linear search algorithm has a run time that never exceeds the curve for n. In other words, linear search is $O(n)$

Now... can we actually prove this?

According to the formal definition for big-O

" $T(n)$ is $O(f(n))$" if for some constants c and n_0, $T(n) \leq cf(n)$ for all $n \geq n_0$

In other words... to prove that $T(n) = 6n + 3$ is $O(n)$ we must find 2 constants c and n_0 such that the statement $T(n) \leq cn$ is true for all $n \geq n_0$

The important part about this... is to realize that we can pick any constant we want as long as it meets the criteria. For c , I can pick any number > 6. I will pick 10 (there isn't a good reason for this other than its bigger than 6, and the math is easy to do in our head for the purposes of presenting this) c = 10. I will also pick 1 for n_0 ,

The following graph shows both $T(n) = 6n + 3$ (blue) as well as the $cf(n) = 10n$ (orange). At some point, the line for $6n + 3$ falls under $10n$ and it never gets above it after that point.
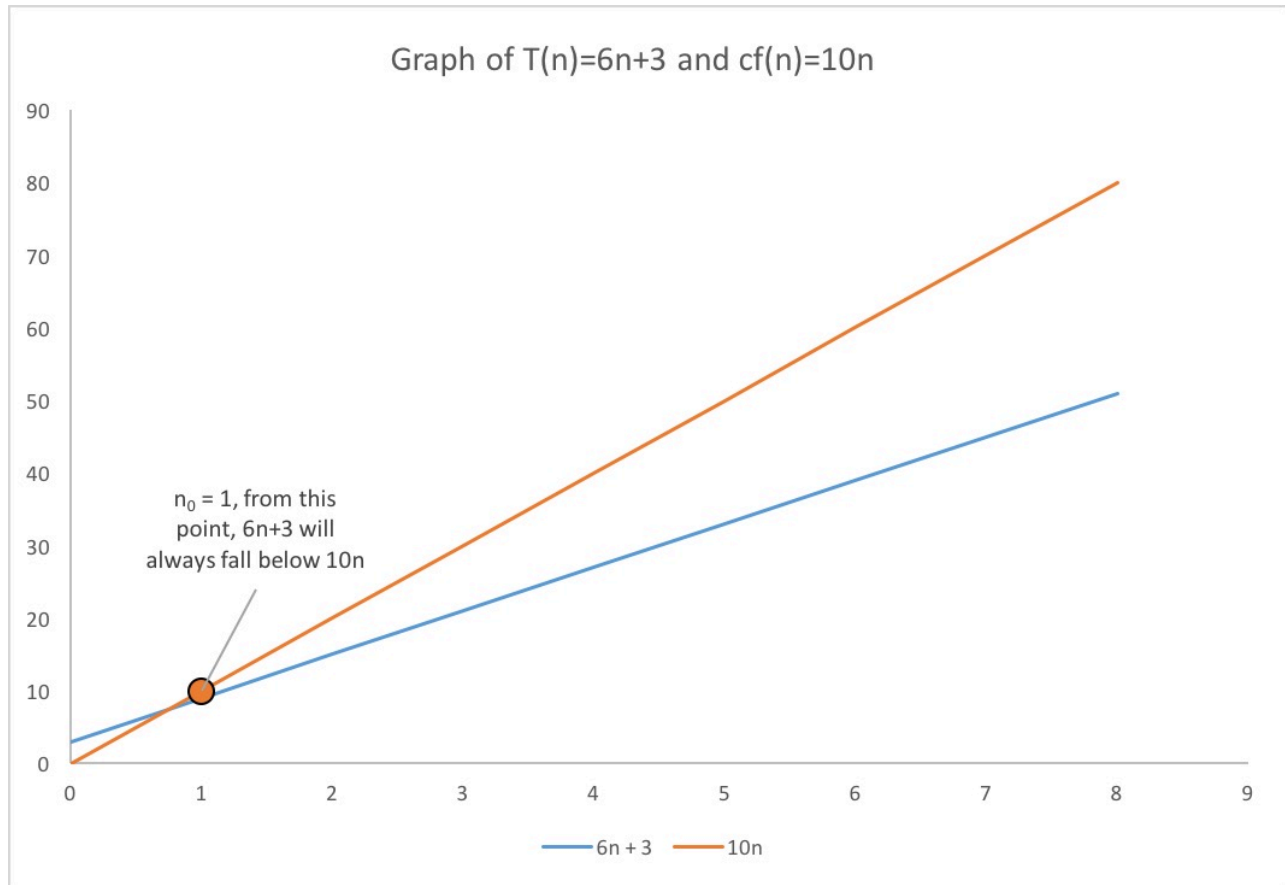
Graph of T(n)=6n+3 and cf(n)=10n

$n_0 = 1$, from this point, $6n+3$ will always fall below $10n$

— 6n + 3   — 10n

Thus, we have proven that the function is $O(n)$ because we were able to find the two constants c and n_0 needed for the $T(n)$ to be $O(n)$

This is also the reason why it did not matter if we counted the operations for the . operator or the [i] operator. If we counted both of them,

$$T(n) = 8n + 3$$

The dominating term would be 8n but this is still O(n).

The proof would go exactly the same way except that we can't use $n_0 = 1$ as the statement $T(n) \leq cn$ is not true when $n = 1$. However, when $n = 2$, $T(n) = 19$ and cn would be 20. Thus, it would be true and stays true for all values of $n > 2$

Analysis of a Successful Search

In the previous analysis, we assumed we wouldn't find the data and thus the worst possible case of searching through the entire array was used to do the analysis. However, what if the item was in the array?

Assuming that key is equally likely to be in any element, we would have to search through n elements at worst and $\frac{n}{2}$ elements on average.

Now since we will at some point find the item, the statement inside the if will also be performed. Thus, we will have 4 operations that we must run through once.

```
int rc = -1
int i = 0
rc=i
return rc;
```

These other operations will also run for each iteration of the loop:

```
i<arr.size() && rc == -1    --> 3 operations
i++                         --> 1 operation
if(arr[i]==key)             --> 2 operations
```

The difference is we do not expect that the loop would have to run through the entire array. On average we can say that it will go through half of the array.

Thus:

$$T(n) = 6 * 0.5n + 4 = 3n + 4$$

Now... as we said before... the constant of the dominating term does not actually matter. $3n$ is still n . Our proof would go exactly as before for a search where the key would not found.

> ⚠ **CAUTION**
>
> One thing to note. If you flip between the python code and the C/C++ code, it would seem that the number steps is higher in the C/C++ version of the code. However, this does not mean that the C/C++ version of the code is slower. In fact generally speaking, you will find C/C++ code to run faster than python due to the way it is optimized and the way it runs. This is why any application where speed truly matters in real time is often written in C/C++. So you cannot compare the two in this manner! Algorithms analysis gives you performance only as a function of resource consumption. it does not translate directly into wall clock time!

Example 2: Analysis of Binary Search

The analysis of a binary search is not the same as that of linear search because the loop of a binary search does not follow the pattern of going from the start of the array all the way to the end. It starts in the middle of an array and jump around. Not every element will be considered during the search process so this will be a bit different.

> ⊙ **INFO**
>
> A **binary search** can only be done to lists if two criteria are true:
>
> 1. the list must be sorted
> 2. access to any element given its position must have a constant run time.

Python C++

```python
def binary_search(my_list, key):
    rc = -1
    low = 0
    high = len(my_list) - 1

    while rc == -1 and low <= high:
        mid = (low + high) // 2
        if key < my_list[mid]:
            high = mid - 1
        elif key > my_list[mid]:
```

In the above code, there are two variables low and high. These represent the first and last index of the elements where key may be located. In other words, the only possible places key may exist is somewhere between my_list[low] and my_list[high] inclusive. In each iteration of the while loop we check the middle element between them. If key is smaller than the middle element, we move the high marker so that it is to one element before the middle one. If key is bigger, we move the low marker one element after the middle one. In other words we eliminate half of the remaining possibilities of key's location after each iteration of the while loop. We stop when we either find it or we have eliminated all possible locations for key.

Analysis of an unsuccessful search

Similar to a linear search we should analyze the run time for what happens in its worst case. In this case, it is also performing an unsuccessful search

Let n represent the size of the array

Let T(n) represent the number of operations needed to find key within the array

In a binary search the following operators are done exactly one time:

```
rc = -1
low = 0
high = len(my_list) - 1   # 1 operator here + 1 function call

return rc
```

The following are done for each iteration within the loop:

```
    rc == -1 and low <= high    # 3 ops per iteration of loop
    mid = (low + high) // 2     # 3 ops per iteration of loop
```

Within the loop there is also an if/else if/else statement. there are 3 paths through this statement. The path that would would lead the the highest operation count would be to follow the else-if. The reason is that we would end up doing both checks. and there are more operations in the else-if than the else statement. Thus, we should use that pathway

```
if key < my_list[mid]:              # <---check this (1 op)
    high = mid - 1                  # <---skip, check fails
elif key > my_list[mid]:            # <---check (1 op)
    low = mid +1                    # <---do this (2 ops)
else:                              # <---skip
    rc = mid                       # <---skip
```

Thus what we can say is that each time the loop runs, it will perform 10 operations at worst

```
template <class TYPE>
int binarySearch(const vector<TYPE>& arr, const TYPE& key){
    int rc=-1;
    int low=0;
    int high=arr.size()-1;
    int mid;
    while(low<=high && rc==-1){
        mid=(low+high)/2;
        if(key < arr[mid])
            high=mid-1;
        else if(key > arr[mid] )
            low=mid+1;
        else
            rc=mid;
```

In the above code, there are two variables low and high. These represent the first and last index of the elements where key may be located. In other words, the only possible places key may exist is somewhere between arr[low] and arr[high] inclusive. In each iteration of the while loop we check the middle element between them. If key is smaller than the middle element, we move the high marker so that it is to one element before the middle one. If key is bigger, we move the low marker one element after the middle one. In other words we eliminate half of the remaining possibilities of key's location after each iteration of the while loop. We stop when we either find it or we have eliminated all possible locations for key.

Analysis of an unsuccessful search

Similar to a linear search we should analyze the run time for what happens in its worst case. In this case, it is also performing an unsuccessful search

Let n represent the size of the array

Let T(n) represent the number of operations needed to find key within the array

In a binary search the following operators are done exactly one time:

```
int rc=-1;
int low=0;
int high=arr.size()-1;
int mid;
return rc;
```

The following are done for each iteration within the loop:

```
    low<=high && rc==-1   //<-- 3 ops
    mid=(low+high)/2;     //<-- 3 ops
```

Within the loop there is also an if/else if/else statement. there are 3 paths through this statement. The path that would would lead the the highest operation count would be to follow the else-if. The reason is that we would end up doing both checks. and there are more operations in the else-if than the else statement. Thus, we should use that pathway

```
if(arr[mid] > key)          //<---check this (1 op)
    high=mid-1;             //<---skip, check fails
else if(arr[mid] < key)    //<---check (1 op)
    low=mid+1;             //<---do this (2 ops)
else                        //<---skip
    rc=mid;                //<---skip
```

Thus what we can say is that each time the loop runs, it will perform 10 operations at worst

Number of Loop Iterations

The next question then is to figure out how many times this loop will run. We are of course assuming that the search will fail. Thus, the only way for this loop to stop is for low<=high to become untrue.

When we started off we have n elements to search. After one iteration through the loop we have approximately half of the original number of elements. After one more iteration we eliminate another half of the remaining elements.

the last iteration would involve only 1 element because low == high == mid, after that we would end up with low > high

Lets look at the pattern of the relationship between number elements within the search set and the iteration number.

Loop iteration	Number of elements between low and high inclusive (approx)	Denominator
1	$n = \frac{n}{1}$	$1 = 2^0$
2	$\frac{n}{2}$	$2 = 2^1$
3	$\frac{n}{2}/2 = \frac{n}{4}$	$4 = 2^2$
4	$\frac{n}{4}/2 = \frac{n}{8}$	$8 = 2^3$
5	$\frac{n}{8}/2 = \frac{n}{16}$	$16 = 2^4$
...	...	...
??	$1 = \frac{n}{n}$	$n = 2^?$

Now look at the exponent 2 was raised to in the denominator and the relationship with its loop iteration. If we can find what we need to raise 2 to in order to get n we will be able to find the number of iterations the loop will run.

By definition:

$b^x = a$ iff $log_b a = x$

In other words, x is the exponent we need to raise b to in order to get a . That value is $log_b a$. Applied to our situation, the exponent we must raise 2 to in order to get n would be $log_2 n$.

Our loop runs one time more than the exponent... thus our loop will run $1 +$ $logn$ time.

Putting these ideas together, we can summarize the operation counts as follows:

Python   C++

```python
def binary_search(my_list, key):
    rc = -1                             # 1
    low = 0                             # 1
    high = len(my_list) - 1             # 2

    while rc == -1 and low <= high:     # 3 * (1 + log_n)
        mid = (low + high) // 2         # 3 * (1 + log_n)
        if key < my_list[mid]:          # (1 + log_n)
            high = mid - 1
        elif key > my_list[mid]:        # (1 + log_n)
            low = mid + 1               # 2 * (1 + log_n)
        else:
            rc = mid

    return rc                           # 1
```

Adding it all up we find the following expression for $T(n)$

$$T(n) = 10(1 + logn) + 5$$
$$= 10 + 10logn + 5$$
$$= (10(logn)) + 15$$

The dominating term is $10(logn)$. Thus, $T(n)$ is $O(logn)$

```cpp
template <class TYPE>
```

Adding it all up we find the following expression for $T(n)$

$$
\begin{aligned}
T(n) &= 10(1 + logn) + 6 \\
&= 10 + 10logn + 6 \\
&= (10(logn)) + 16
\end{aligned}
$$

The dominating term is $10(logn)$. Thus, $T(n)$ is $O(logn)$

How to do an analysis in 5 steps

Doing mathematical analysis can be intimidating. However, what you are really doing is simply explaining a thought process. This can be done in a fairly consistent step by step manner. This section looks at the steps you should take to do this and break down how to approach an analysis.

Step 0: It is good to start by putting your code in.

> ⚠ **INFO**
>
> If you are writing out an analysis for a lab or assignment, it is a good idea to start by providing the snippet of the code you are analyzing so that you have a reference.

Python   C++

```python
def factorial (number):
    rc = 1

    # multiplying 1*x gives x, so we can skip it
    for i in range(2,n+1):
        rc = rc * i
    return rc
```

```cpp
unsigned int factorial (unsigned int n){
    unsigned int rc = 1;

    // multiplying 1*x gives x, so we can skip it
    for (unsigned int i=2;i<=n;i++){
        rc=rc*i;
    }
    return rc;
}
```

> 💡 **TIP**
>
> If you are copying code into a markdown page, you can use surround the code block with ``` to do a block quote.
>
> To format according to a language provide the name of the language after the third tick that precedes the block. For example:
>
> ``` cpp
>
> put your code here. code will be c++ syntax formatted
>
> ```
>
> ``` python
>
> put your code here. code will be python syntax formatted
>
> ```

Step 1: Establish variables and

functions (mathematical ones):

> ⚠ **INFO**
>
> Similar to writing a program you want to declare your variables and function prototypes first. You need to do this to establish context... variables need a meaning, functions need a meaning. You need to state what they represent first. **These statements need to be placed above the code block**

Let n represent the value we are finding the factorial for

Let $T(n)$ represent number of operations needed to find n! using the code below:

Step 2: Count your operations

> ⚠ **INFO**
>
> In this step you are explaining how it is that you come up with your final equation. Essentially, you are establishing the logic behind the mathematical statement you will make in step 3

Put the operation count in a comment beside each line of the code blurb. The loop runs n-1 times (i goes from 2 to n inclusive). Thus any operations that occur in the loop is counted as n-1.

Python   C++

```python
def factorial (number):
    rc = 1                              # 1

    for i in range(2,n+1):              # (n-1) + 1 + 1
                                        # (n-1) loop
iterations
                                        # 1 more for +
operator
                                        # 1 more for call
to range function
        rc = rc * i                     # 2 (n-1)
                                        # 2 as there are
two operators
                                        # (n-1) as loop
runs n-1 times
    return rc                           # 1
```

Step 3: Establish the Mathematical Expression for T(n)

> **(!) INFO**
>
> Based on the counts from step 2, put together an expression for T(n) you can even express the condition for when the expression is true... for example

We add up the operation counts in the comments

$$T(n) = 1 + (n - 1) + 1 + 1 + 2(n - 1) + 1$$

Step 4: Simplify your Equation

$$T(n) = 1 + (n - 1) + 1 + 1 + 2(n - 1) + 1$$
$$= 3(n - 1) + 4$$
$$= 3n - 3 + 4$$
$$= 3n + 1$$

Step 5: State your final result.

Therefore, $T(n)$ is $O(n)$

```
unsigned int factorial (unsigned int n){
    unsigned int rc = 1;                      // 1
    for (unsigned int i=2;i<=n;i++){          // 1 + (n-1) + (n-1)
                                              // 1 for i = 2
```

Step 3: Establish the Mathematical Expression for T(n)

> ⓘ **INFO**
>
> Based on the counts from step 2, put together an expression for T(n) you can even express the condition for when the expression is true... for example

We add up the operation counts in the comments

$$T(n) = 1 + 1 + (n - 1) + (n - 1) + 2(n - 1) + 1$$

Step 4: Simplify your Equation

> ⓘ **INFO**
>
> When you first write out your expression in step 3, the formula isn't always a polynomial. You should start by simplifying your equation. You want to establish a mathematical equation of the form:
>
> $$T(n) = a_x n^x + a_{x-1} n^{n-1} + a_{x-2} n^{n-2} ... + a_1 x + 1$$

$$
\begin{aligned}
T(n) &= 1 + 1 + (n - 1) + (n - 1) + 2(n - 1) + 1 \\
&= 4(n - 1) + 3 \\
&= 4n - 4 + 3 \\
&= 4n - 1
\end{aligned}
$$

Step 5: State your final result.

Therefore, $T(n)$ is $O(n)$

Recursion

Introduction

Recursion is one of those things that some of you may or may not have heard of / attempted. This section of the notes will introduce to you what it is if you do not already know and how it works. Some of this will be review some will not. Take your time to try and understand this process.

It is important to know at least a little recursion because some algorithms are most easily written recursively. The text book sometimes only provide the recursive version of an algorithm so in order for you to understand it, you will need to understand how recursion works.

The Run-time Stack

The run-time stack is basically the way your programs store and handle your local non-static variables. Think of the run-time stack as a stack of plates. With each function call, a new "plate" is placed onto the stacks. local variables and parameters are placed onto this plate. Variables from the calling function are no longer accessible (because they are on the plate below). When a function terminates, the local variables and parameters are removed from the stack. Control is then given back to the calling function. Understanding the workings of the run-time stack is key to understanding how and why recursion works.

Writing Recursive Functions

Recursive functions are sometimes hard to write because we are not used to thinking about problems recursively. However, if we try to keep the following in mind, it will make writing recursive functions a lot simpler. There are two things you should do:

1. state the base case (aka easy case). what argument values will lead to a solution so simple that you can simply return that value (or do a very simple calculation and return that value?

2. state the recursive case. if you are given something other than the base case how can you use the function itself to get to the base case

3. Recursion is about stating a problem in terms of itself. The solution to the current problem consists of taking a small step and restating the problem.

Example: The factorial function.

Write a function that is given a single argument n and returns n!. n! = n*(n-1)*(n-2) ...2* 1. For example 4! = 4 * 3 * 2 * 1 , by definition, 0! is 1

How would you do it recursively?

Lets start with base case. What value of n is it easy to come up with the result of n!?

0 is easy.. by definition 0! is 1. 1 is also easy because 1! = 1

Thus, we can establish that any value 1 or less is a base case with result of 1.

Now, the recursive case... how can we state factorial in terms of itself?

Let's consider this example:
$$5! = (5)(4)(3)(2)(1)$$
$$4! = (4)(3)(2)(1)$$

but consider another way to look at $5!$

$$5! = (5)\underbrace{(4)(3)(2)(1)}_{4!}$$

Thus, we can express above as:

$$5! = (5)(4!)$$

And in gneral:

$$n! = n(n-1)!$$

```
unsigned int factorial(unsigned int n) {
    unsigned int rc = 1 ;          //base case result
    if(n > 1) {                    //if n > 1 we  have the
recursive case
        rc = n * factorial(n - 1);  //rc is n * (n - 1)!
    }
    return rc;
}
```

💡 **TIP**

Recursion works by magic... ok it really doesn't but it might be easier to think of it that way. Essentially what recursion does is it starts with the idea that you already have a working function. (ie factorial already works). The base case actually ensures it does work at least for some values of n. The recursive case is simply using a working function to solve the

problem. Don't think about it too much and it will be a lot easier to code.

How Does Recursion Work?

To understand how recursion works, we need to look at the behaviour of the run time stack as we make the function calls.

> **① INFO**
>
> The runtime stack is a structure that keeps track of function calls and local variables as the program runs. When a program begins, the main() function is placed on the run time stack along with all variables local to main(). Each time a function is called, it gets added to the top of the runtime stack along with variables and parameters local to that function. Variables below it become inaccessible. When a function returns, the function along with its local variables are popped off the stack allowing access to its caller and its callers variables.

Suppose we have the following program:

```
unsigned int factorial(unsigned int n) {
    unsigned int rc = 1 ;          //base case result
    if(n > 1) {                    //if n > 1 we  have the
recursive case
        rc = n * factorial(n - 1);  //rc is n * (n - 1)!
    }
    return rc;
}


int main(void){
```

Lets trace what happens to it with respect to the run time stack:

Program begins, main() function is placed on stack along with local variable x.
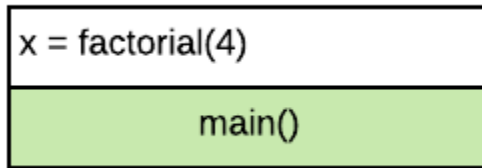
x = factorial(4)
main()

factorial(4) is called, so we push factorial(4) onto the stack along with local variables n and rc.

rc = n * factorial(n-1) n = 4
factorial(4)
x = factorial (4)
main()
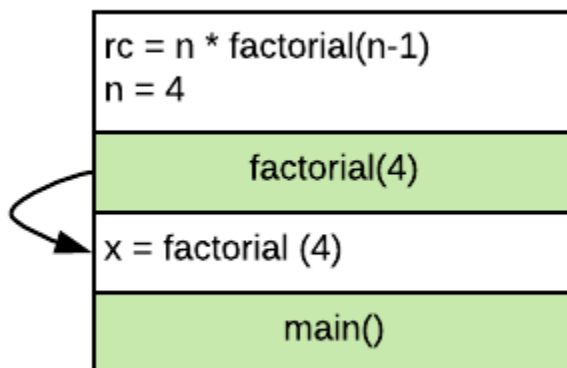
n is 4, and thus the if statement in line 3 is true, thus we need to call factorial(3) to complete line 4.

```
rc = n * factorial(n-1)
n = 3
```
factorial(3)
```
rc = 4 * factorial(3)
n = 4
```
factorial(4)
```
x = factorial (4)
```
main()
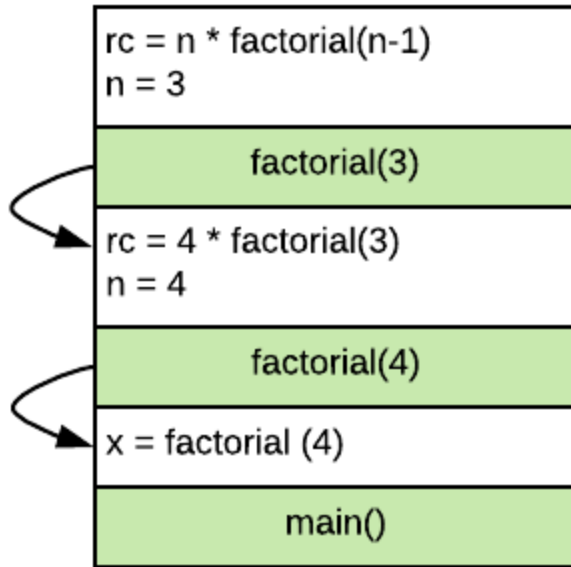
> ⓘ **INFO**
>
> Note that in each function call to factorial on stack there is a value for n and rc. Each n and rc are separate variables so changing it in one function call on stack will have no effect on values held in other factorial function calls on stack.

Now, the n is 3 which makes the if statement true. Thus, we make a call to factorial(2).

```
rc =  n* factorial( n-1)
n = 2

          factorial(2)

rc = 3 * factorial(2)
n = 3

          factorial(3)

rc = 4 * factorial(3)
n = 4

          factorial(4)

x = factorial (4)

          main()
```
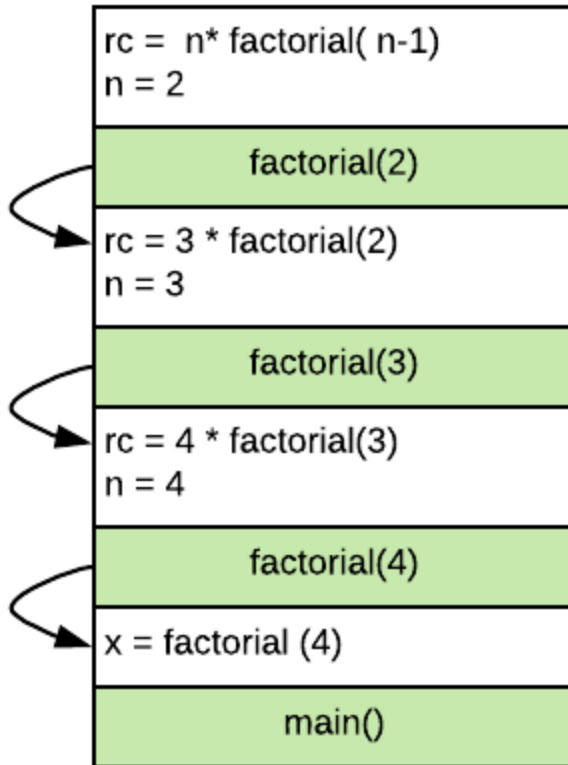
Once again, the if statement is true, and thus, we need to call factorial(1).

```
rc = 1
n = 1
```
factorial(1)
```
rc = 2* factorial(1)
n = 2
```
factorial(2)
```
rc = 3 * factorial(2)
n = 3
```
factorial(3)
```
rc = 4 * factorial(3)
n = 4
```
factorial(4)
```
x = factorial (4)
```
main()
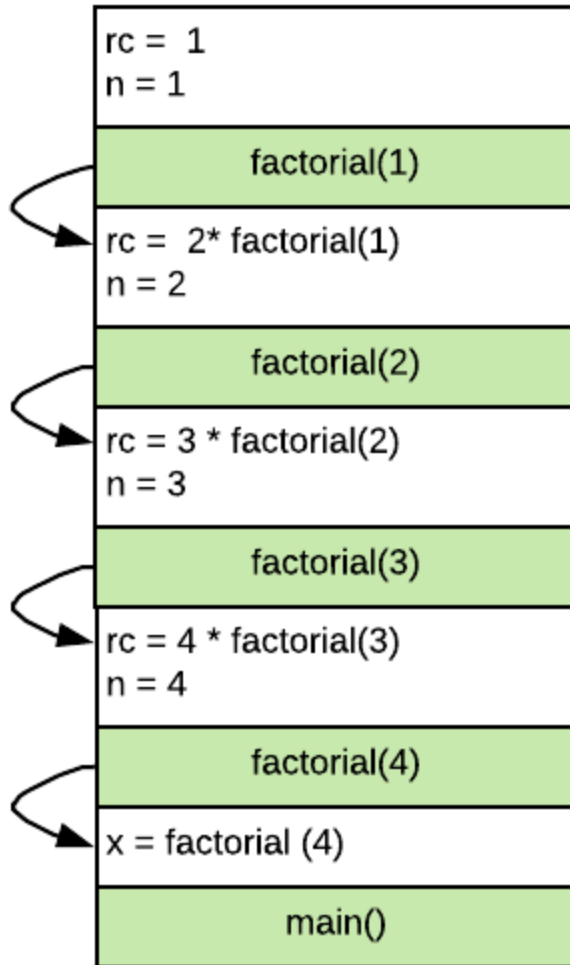
Once we make this call though, our if statment is false. Thus, we return rc popping the stack

```
rc =  2*1 = 2
n = 2
```
factorial(2)
```
rc = 3 * factorial(2)
n = 3
```
factorial(3)
```
rc = 4 * factorial(3)
n = 4
```
factorial(4)

x = factorial (4)

main()
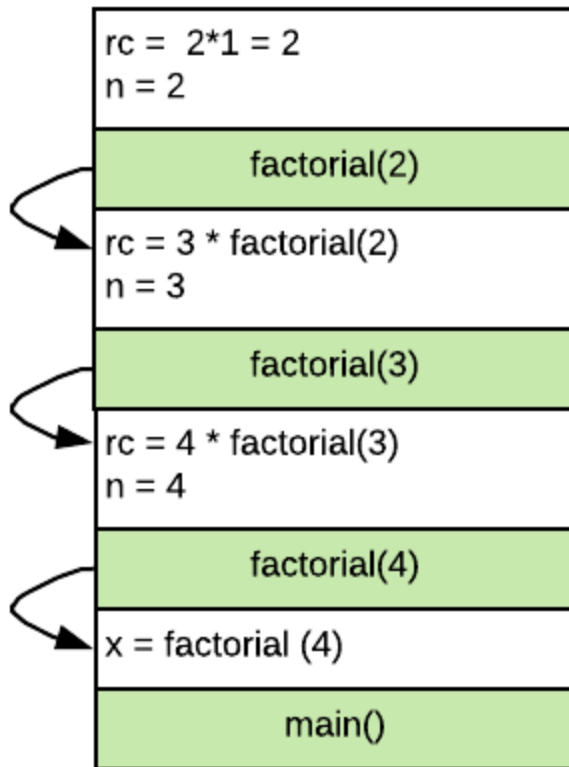
Once it is returned we can complete our calculation for rc = 2. This is returned,

```
rc = 3 * 2 = 6
n = 3
```
factorial(3)
```
rc = 4 * factorial(3)
n = 4
```
factorial(4)

x = factorial (4)

main()
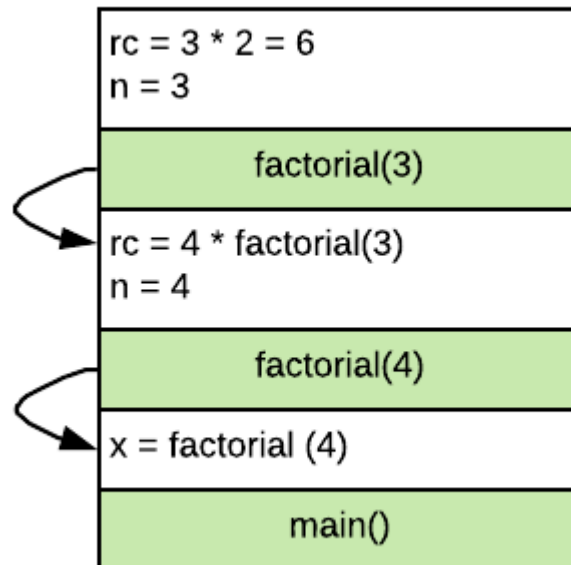
popping the stack

Once it is returned we can complete our calculation for rc = 6. This is returned,

rc = 4 * 6 = 24 n = 4
factorial(4)
x = factorial (4)
main()
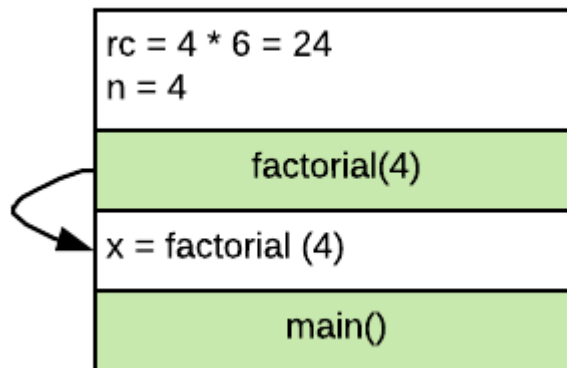
popping the stack

Once it is returned we can complete our calculation for rc = 24. This is

x = 24
main()
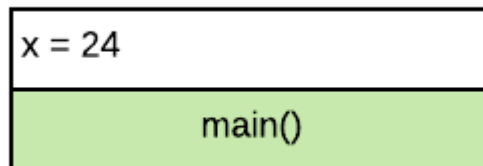
returned, popping the stack

Analysis of a Recursive Function

Performing an analysis of a recursive function is not all that different from performing an analysis of a non-recursive function. We are still going to use the same methodology to find a formula that will represent the number of operations required for a given data size. We will then use that formula to find a curve that best describes the resource usage growth rate.

Analysis of Factorial function

How would we perform an analysis on the following piece of code?

```
unsigned int factorial(unsigned int n){
    unsigned int rc = 1 ;              //base case result
    if(n > 1) {                  //if n > 1 we  have the recursive
case
        rc= n * factorial(n-1);  //rc is n * (n-1)!
    }
    return rc;
}
```

Start by declaring our mathematical variables and functions

Let n represent the number we are finding the factorial of

Let $T(n)$ represent the number of operations it takes to find n! using our recursive function

Similar to iterative version of our code, we will simply count our operations:

```
unsigned int factorial(unsigned int n){
    unsigned int rc = 1 ;        // 1
    if(n > 1) {                  // 1

        rc= n * factorial(n-1);  // 3 + number of ops done by
factorial(n-1)

                                 // There are 3 operators, but it
also
                                 // calls factorial(n-1) so we
must count
                                 // not just 3, but also all ops
done by
                                 //factorial(n-1)

    }
    return rc;                    //1
}
```

Now... by definition $T(n)$ represents the number of ops our function (factorial) requires/does when given n as its argument. But this also means that $T(n-1)$ would represent the number of ops needed by our function to find (n-1)! (ie number of operations performed by our function when n-1 is passed in as the argument.

Thus, we can actually rewrite our count summary as follows:

```
unsigned int factorial(unsigned int n){
    unsigned int rc = 1 ;        // 1
    if(n > 1) {                  // 1

        rc= n * factorial(n-1);  // 3 + T(n-1)
    }
    return rc;                    //1
}
```

Now... how do we solve this though? So, lets start by looking at our base cases. If n <= 1, we do exactly 3 operations:

```
unsigned int rc = 1 ;
n > 1
return rc;
```

In otherwords, we can say the following:

$$T(0) = 3$$
$$T(1) = 3$$

Remember that T is a function that represents how many operations our function requires to find factorial of the argument n. So if argument was 0, it takes 3 ops, if argument was 1, it takes 3 ops also. Notice how the analysis of this coincides with the base case of the recursive function

For n >= 2, we do the following 6 operations + we need count number of ops in the recursive call:

```
unsigned int rc = 1 ;      <--- 1 op
n > 1                      <--- 1 op
return rc;                 <--- 1 op
rc= n * factorial(n-1)  <--- 3 operations, =, * and - ... also
need to count
                           number of ops done by factorial(n-1)
```

Thus, we can write the expression as following for the general $T(n)$ for n >=2

$$T(n) = 6 + T(n - 1)$$

Now... by definition T(n) represents the number of ops needed to find n! using our function. But this also means that T(n-1) would represent the number of

ops needed to find (n-1) factorial using our function. Thus,

$$T(n) = 6 + T(n-1)$$

by the same token, we can say that

$$T(n-1) = 6 + T(n-2)$$
$$T(n-2) = 6 + T(n-3)$$

etc...

In other words T(n-1) = 6 + number of operations done by factorial(n-2). Similarly T(n-2) = 6 + number of operations done by factorial(n-3)

The above statement is true for all values of n >= 2. However, we also know that:

$$T(1) = 3 \text{ and } T(0) = 3$$

Thus, what we have is:

$$T(n) = 6 + T(n-1)$$
$$= 6 + 6 + T(n-2)$$
$$= 6 + 6 + 6 + T(n-3)$$
...
$$T(n) = 6 + 6 + 6 + ...6 + 3$$

There are a total of (n-1) 6's.We know this because we need to reach T(1) to get the 3.

Thus:

$$T(n) = 6(n-1) + 3 = 6n - 3$$

Thus, $T(n)$ is $O(n)$

Drawbacks of Recursion and Caution

Recursion isn't the best way of writing code. If you are writing code recursively, you are probably putting on extra overhead. For example the factorial function could be easily written using a simple for loop. If the code is straight forward an iterative solution is likely faster. In some cases, recursive solutions are much slower. You should use recursion if and only if:

1. the problem is naturally recursive (you can state it in terms of itself)
2. a relatively straight forward iterative solution is not available.

Even if both conditions above are true, you still might want to consider alternatives. The reason is that recursion makes use of the run time stack. If you don't write code properly, your program can easily run out of stack space. You can also run out of stack space if you have a lot of data. You may wish to write it another way that doesn't involve recursion so that this doesn't happen.
.

Searching and Sorting

Introduction

Searching is the process of finding particular piece of data (the key) within a data structure. Sorting is the process of taking a set of data and creating an ordering based on some criteria. In this section of the notes, we will be looking at sorting a list of data which will help support certain methods of searching for data within the list.

> ⚠️ **CAUTION**
>
> In this section of the notes, we will be refering to the data structure we are searching and sorting as a "list". With some of these algorithms, we specifically require the list to be array based/like as those algorithms require you to have *fast random access* to any element given its index. For simplicity, you can assume that the list we are talking about is array based/like.
>
> Types of lists that are "array based/like":
>
> - Python lists.
> - Vectors from the C++ standard library
> - C++ arrays
>
> List that are not "array based/like":
>
> - list from C++ standard library

Python provides a number of built-in functions to perform searching and

sorting. However, a programmer should know how these functions work, not just be able to call them.

This section of the notes will look at 2 different searching, and 5 different sorting algorithms. A 6th sorting algorithm will be presented later in the course.

Searching

Searching is the process of finding particular piece of data (the key) within a data structure. Some data structures will support faster searching by the way that the data is stored and organized.

In this section of the notes we are concerned with how to search a list. In particular we are looking at lists that are array like. For example, Python lists are "array like", as are vectors from the C++ standard library. The list from C++ standard library however is not array based so this is not the type of list we are considering in this chapter. What we want are data structures that provide fast random access to any element given its index.

Linear Search

The linear search algorithm is a pretty standard algorithm that you have probably written at some point in your previous programming courses. You may not have called it linear search, but it is likely that you have written it. Here we will formalize what the function does.

The linear search algorithm is given a list of values and a key. It returns the first index of where the key is found or -1 if the key is not part of the list. We use -1 to indicate the state of not finding the value because 0 is a perfectly valid index into the list.

The code for this is pretty straight forward. Start at the beginning of the list and search until you either find the item or you reach the end of the list.

Python   C++

```python
def linear_search(my_list, key):
    for i in range(0, len(my_list)):
        if my_list[i] == key:
            return i

    return -1
```

```cpp
template <class TYPE>
int linearSearch(const vector<TYPE>& arr, const TYPE& key){
    int rc=-1;
    for(int i=0;i<arr.size()&& rc==-1;i++){
        if(arr[i]==key){
            rc=i;
        }
    }
    return rc;
}
```

Performance of a linear search

A linear search has a run time of O(n). You saw this analysis in the algorithms analysis part of the notes.

If you were to sort the array and then, perform a linear search (where you would stop when you either find the key or find a value in the list that is bigger than key,) it's not worth it as your overall cost will be more than O(n) (beacues of sorting first!). Assuming that your key can be found anywhere in the array (i.e. equal chance of being smallest/biggest/second smallest/etc.), finding a value, on average, still requires that you go through half the array, and at the worst case, the entire array. The search part of the process is still, therefore, linear.

Binary Search

Binary search is a search that can only be performed on lists that are sorted AND have random access to each of its elements.

Consider two different number guessing games:

In game 1 you are asked to guess a number between the values 1 to 1000. After every guess you are told whether you are correct or incorrect. That is it, no other information other than whether you are right or wrong.

In game 2 you are asked to guess a number between the values 1 to 1000. After every guess you are told that the guess is either correct, too high, or too low.

Which game is it easier to find the correct number?

In the first game, you are only told whether your guess was right or wrong. Thus, any guess that you make can eliminate only one number from consideration. Every other number is still possible. Initially you had 1000 options to choose from. After making a guess, if you are wrong, you have 999 options left to choose from. After two guesses, 998 possibilities. This is linear search. After you look at one element, you eliminate the possibility that the value is there... but it is possible for value to be in every other element.

In the second game, you are told that not only whether the guess is correct or in correct but also whether the value is too high or too low. This allows you to eliminate many more numbers. If you guess 500 as your first guess, then even if the number is not 500, you can immediate either eliminate all numbers bigger than 500 or all numbers smaller than 500. Repeatedly doing this, always guessing at the middle of the remaining valid range will allow you to

continue to half the data set with each guess.

This is why the second game is much easier... you are zeroing in on the value you want.

Preconditions of a binary search

To perform a binary search requires two things:

- the list must be sorted. This allows us to split the array into two pieces, one where the key might be found and the other where the key definitely can't be found.
- the second requirement is the list must allow fast random access (ie be array-like). That is getting to any element of the list takes the same amount of time. In the list from the C++ standard library this is not the case.

The algorithm

The binary search algorithm goes like this: *Track the range of possible indexes by storing the first/last index where key may be found* initially this is 0 and (length of list) - 1. *Calculate the mid point index between those first/last indexes* look at the value at the middle element *if we found it we are done* if the key is smaller than middle element, then key can only be found between first index and element before middle element. Thus, set last index to middle index - 1. * if key is greater than middle element then key can only be found after that middle element, thus set first index to middle index + 1

Python   C++

```python
def binary_search(my_list, key):
```

```cpp
template <class TYPE>
int binarySearch(const vector<TYPE>& arr, const TYPE& key){
    int rc=-1;
    int low=0;
    int high=arr.size()-1;
    int mid;
    while(low<=high && rc==-1){
        mid=(low+high)/2;
        if(key < arr[mid])
            high=mid-1;
        else if(key > arr[mid] )
            low=mid+1;
        else
            rc=mid;
    }/*while*/
    return rc;
}
```

Sorting

Sorting is the process of taking a set of data and creating an ordering based on some criteria. The criteria must allow for items to be compared and used to determine what should come first and second. For example you can sort numbers. But even then, there is an ascending order (smaller numbers come before the bigger ones) or a descending order (bigger numbers come before the smaller ones). For other data, you can still have this kind of ordering. For example for strings, you can have alphabetic ordering ("apple" comes before "banana"). Anything that allows you to compare and say item A comes before item B allows for sorting.

Regardless of what it is that we are basing the sorting on, the algorithms used for sorting are the same. The only difference is the comparison operation. With the comparison all we are doing is asking which item should come first.

In this section of the notes we will cover 5 sorting algorithms.

Simple Sorts

The first 3 sorting algorithms are fairly straight forward. They are all O(n^2). They are also the easiest to understand. We will give a brief description of each below.

Bubble Sort

Bubble sort is called bubble sort because the algorithm repeatedly bubbles the largest item within the list to the back/end of the list.

Algorithm

1. start with first pair of numbers
2. compare them and if they are not correctly ordered, swap them
3. go through every pair in list doing the above 2 steps
4. repeat the above 3 steps n-1 times (ie go through entire array n-1 times) and you will sort the array

Animation

http://cathyatseneca.github.io/DSAnim/web/bubble.html

Implementation

Python   C++

```python
def bubble_sort(my_list):
    n = len(my_list)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if my_list[j] > my_list[j + 1]:
                my_list[j], my_list[j + 1] = my_list[j + 1],
my_list[j]
```

```c
void bubbleSort(int arr[],int size){
    int tmp;                              /*used for swapping*/
    int i;
    int j;
    for (i=0; i<size-1; i++) {
        for (j=0; j<size-1-i; j++){
            int next = j+1;
            if (arr[next] < arr[j]) {        /* compare the two
neighbors */
                tmp = arr[j];                    /* swap a[j] and
a[j+1]      */
                arr[j] = arr[next];
                arr[next] = tmp;
            }
        }
    }
}
```

Selection Sort

Selection sort works by selecting the smallest value out of the unsorted part of the array and placing it at the back of the sorted part of the array.

Algorithm

1. find the smallest number in the between index 0 and n-1
2. swap it with the value at index 0
3. repeat above 2 steps each time, starting at one index higher than previous and swapping into that position

Animation

http://cathyatseneca.github.io/DSAnim/web/selection.html

Implementation

Python   C++

```python
def selection_sort(my_list):

    n = len(my_list)
    for i in range(n - 1):
        min_idx = i   # record the index of the smallest value,
                      # initialized with where the smallest value
may be found
        for j in range(i + 1, n):              # go through list,
            if my_list[j] < my_list[min_idx]:  # and every time
we find a smaller value,
                min_idx = j                    # record its index
(note how nothing has moved at this point.)
```

```
void selectionSort(int arr[],int size){
    int i,j;
    int min;   //index of smallest value in the unsorted array

    for(int i=0;i<size-1;i++){
        min=i;
        for(int j=i+1;j<size;j++){
            if(arr[j] < arr[min]){
                min=j;
            }
        }
        if(min!=i){
            int tmp=arr[min];
            arr[min]=arr[i];
            arr[i]=tmp;
        }
    }
}
```

Insertion Sort

Insertion sort is called insertion sort because the algorithm repeatedly inserts a value into the part of the array that is already sorted. It essentially chops the array into two pieces. The first piece is sorted, the second is not. We repeatedly take a value/number from the second piece and insert it into the already sorted first piece of the array.

Algorithm

1. start with the second value in the list (note that the first piece/portion of the list contains just the first value initially.)
2. put that value into a temporary variable. This makes it possible to move

items into the spot the second value occupies at the time and this spot is now considered to be an empty spot in the sorted piece/portion of the array.

3. if, when compared with the last value in the first piece/portion of the list, the value in the temporary variable should go into the empty spot, put it in.
4. otherwise, move the last value in the sorted piece/portion part into the empty spot.
5. repeat steps 3 to 4 until the value in the tempprary variable is placed somewhere into the first sorted piece/portion of the array.
6. repeat steps 2 to 5 for every vakue in the second piece/portion/part of the array until all values are placed in the first part.

Animation

http://cathyatseneca.github.io/DSAnim/web/insertion.html

Implementation

Python   C++

```python
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        curr = my_list[i]  # store the first number in the
unsorted part of
                            # of array into curr
        j = i
        while j > 0 and my_list[j - 1] > curr:      # this loop
shifts value within sorted part of array
```

```c
void insertionSort(int arr[],int size){
    int curr;
    int i, j;
    for(i=1;i<size;i++){
        curr=arr[i];
        for(j=i;j>0 && arr[j-1] > curr;j--){
            arr[j]=arr[j-1];
        }
        arr[j]=curr;
    }
}
```

Merge Sort

The merge sort works on the idea of merging two already sorted lists.

If there existed two already sorted list, merging the two together into a single sorted list can be accomplished in O(n) time where n is the combined length of the two lists.

Thus, if we are able to take our original list, split it into two pieces and then somehow get them into a sorted state, we can then use the merge algorithm to put these two pieces back together.

The algorithm of exactly how we can do this is described below.

Algorithm:

Merge algorithm

Understanding the merge algorithm is an important step of understanding how merge sort works.

The algorithm to merge already sorted list works like this

- create an empty merged list
- have a way to "point" at the first element of each of the two list
- compare the values being pointed at and pick the smaller of the two
- copy the smaller to the end of the merged list, and advance the "pointer" of the list that the value came from
- continue until one of the list is completely copied then copy over remainder of the rest of the list

Merge Sort Algorithm

The merge algorithm is an O(n) algorithm as we pick n items between two lists. However, it depends on having two lists that are already sorted...So we must first get a list into that state.

1. Start with the entire list
2. Split the list into two halfs
3. mergesort each half - this will result in two sorted lists
4. merge them together

Now... the above is clearly a recursive algorithm. The base case for this are lists of size 0 or size 1. If you have an list that is 0 elements it doesn't exist so we don't have to sort it. If a list is 1 element then that list is sorted.

Animation

http://cathyatseneca.github.io/DSAnim/web/merge.html

Implementation

The implementation of merge sort is split into 3 pieces, each piece is detailed below.

mergeSort()

This is the mergeSort() function that the program calls. The recursive merge sort has a complicated prototype, we hide the complexity by writing a function

that will make the initial call. We also need a second list to store the merged list. The sorting algorithm creates a temporary list once and passes that into a recursive function to do all the work. The idea here is that we don't want to allocate memory on each merge... instead we can just use one list that stays around for the entire process.

Python   **C++**

```python
def merge_sort(mylist):
    empty_list = [0] * len(mylist)  # create an empty list for merging.  doing it once
                                    # is more efficient than repeatedly creating it when merging

    recursive_merge_sort(mylist, 0, len(mylist) - 1, empty_list)  # call recursive mergesort
```

```cpp
void mergeSort(int arr[],int size){
  int* tmp=new int[size];
  mergeSort(arr,tmp,0,size-1);
  delete [] tmp;
}
```

recursive merge sort

The recursive mergesort splits the list into two pieces, mergesorts each piece then merge them together. While we speak about splitting up the list, the reality is that the split is more conceptual than actual. We will use indexes to indicate where one list begins and one list ends. Thus this function is not only given the list but also the indexes to indicate where it begins and ends. The function is also passed an empty list that we pass to merge function. This is

done for efficiency.

Python   C++

```python
def recursive_merge_sort(mylist, first_index, last_index,
empty_list):
    # recursive merge sort.  the base case occurs when
first_index >= last_index
    # that situation will only occur if the portion of the list
is size 0 or 1.
    # in either case, do nothing and exit function.

    if first_index < last_index:
        mid_index = (first_index + last_index) // 2
        recursive_merge_sort(mylist, first_index, mid_index,
empty_list)
        recursive_merge_sort(mylist, mid_index + 1, last_index,
empty_list)
        merge(mylist, first_index, mid_index + 1, last_index,
empty_list)
```

```cpp
void mergeSort(int arr[],int tmp[],int start,int end){
  //if the array is more than one element big
  if(start<end){
    int mid=(start+end)/2;
    mergeSort(arr,tmp,start,mid);
    mergeSort(arr,tmp,mid+1,end);
    merge(arr,tmp,start,mid+1,end);
  }
}
```

The merge function

The merge function takes two sorted list and merge them together. However, because we are merging together two sorted pieces of a list, rather than being given individual lists, the function prototype must take in indexes to indicating starting/ending points of these lists within the original list

Python   C++

```python
# this function will merge two sorted pieces of an array into a
single sorted segment.
# We will refer to the two smaller sorted pieces of lists as a
and b.
# These are not true lists but rather pieces of mylist
# list a starts at a_first_index and ends at b_first_index - 1
(inclusive)
# list b starts at b_first_index and ends at b_last_index
(inclusive)
# this function will merge them together using empty_list as
temporary storage
# once it is merged together, it is copied back into mylist,
creating a single
# sorted segment that starts at a_first_index to b_last_index
(inclusive)

def merge(mylist, a_first_index, b_first_index, b_last_index,
empty_list):
    a_ptr = a_first_index  # used to track value from a
    b_ptr = b_first_index
    empty_list_index = a_ptr

    while (a_ptr < b_first_index) and (b_ptr <= b_last_index):
        if mylist[a_ptr] <= mylist[b_ptr]:
```

```c
/*function merges the two halfs of a sorted array together.
   The arrays are defined from arr[startA]to arr[startB-1] and
arr[startB]
to arr[endB]*/

void merge(int arr[],int tmp[],int startA,int startB,int endB){
  int aptr=startA;
  int bptr=startB;
  int idx=startA;
  while(aptr < startB && bptr < endB+1){
    if(arr[aptr] < arr[bptr]){
       tmp[idx]=arr[aptr];
       idx++;
       aptr++;
    }
    else{
       tmp[idx++]=arr[bptr];
       bptr++;
    }
  }
  while(aptr<startB){
    tmp[idx]=arr[aptr];
    idx++;
    aptr++;
  }

  while(bptr < endB+1){
    tmp[idx++]=arr[bptr];
    bptr++;
  }

  //copy back into arr
  for(int i=startA;i<=endB;i++){
    arr[i]=tmp[i];
  }
}
```

Quick Sort

This sort is fast and does not have the extra memory requirements of MergeSort. On average its run time is $O(n\ log\ n)$ but it does have a worst case run time of $O(n^2)$

The quick sort algorithm is a divide and conquer strategy where it will partition an array into 3 parts: - all values smaller than a pivot value (more on this later), the pivot, and all values larger than the pivot value

This partitioning algorithm is very fast and has an O(n) run time where n is the size of the original partition.

By partitioning the array in this manner we are splitting a large array into 3 pieces. The pivots are correctly placed in the array. All values smaller than the pivot are in the front part of the array. All values larger than pivot come after the pivot.

This partitioning algorithm is then applied to the parts that are smaller/bigger than pivot. The pivot is in the right place so no need to move it.

Once the partition is small enough, insertion sort is used to sort the very small arrays. The point at which the partition becomes "small enough" is dependent on the compiler but it is somewhere around the 16 to 32 element element range.

Algorithm:

Partitioning Algorithm

1. Select a pivot. This pivot must be part of the partition. The pivot is to be

chosen randomly from the the values within the partition.

2. Move the pivot out of the way by swapping it with the value at the end of the partition.

3. Go through list starting at the beginning. track the "end" of the smaller partition. This is initially set to being 1 before start index of the partition

4. Every time you come across a value that is smaller than the pivot you advance the "end" of the smaller partition by 1 and swap the value with whatever is there. This will effectively place all values smaller than pivot at the front of the array

Quick Sort Algorithm

1. if list is small, perform insertion sort. This is the base case

2. if the list is not too small, partition the list using the partitioning algorithm described above. This will split the list into 3 pieces, all values smaller than the pivot, the pivot, values greater than pivot

3. quick sort the piece that is smaller than pivot and the piece that is larger than the pivot.

Implementation

The implementation of quick sort has 4 components. Each of these are detailed below

quick sort

This function is similar in nature to the merge sort function. Its job is to provide a simple interface to the user. The actual work is done in the recursive quicksort function

```python
def quick_sort(mylist):
    recursive_quick_sort(mylist, 0, len(mylist) - 1)  # call
recursive quicksort
```

```cpp
void quickSort(int arr[],int size){
   quickSort(arr,0,size-1);
}
```

Recursive Quicksort

This is the recursive quick sort algorithm. THRESHOLD is a constant. When a partition's size falls below the THRESHOLD value, insertion sort is performed. This serves as the base case for recursive quick sort.

If the partition size is larger than the threshold, we pick partition the array, returning the location of the pivot. After this, we quick sort the remaining parts of the array.

```python
def recursive_quick_sort(mylist, left, right, THRESHOLD=32):
    if right - left <= THRESHOLD:
        insertion_sort(mylist, left, right)
    else:
        pivot_position = partition(mylist, left, right)
        recursive_quick_sort(mylist, left, pivot_position - 1)
        recursive_quick_sort(mylist, pivot_position + 1, right)
```

```cpp
void quickSort(int arr[], int left, int right){
    if(right-left <= THRESHOLD){
        insertionSort(arr,left,right);
    }
    else{
        int i = partition(arr,left,right);
        quickSort(arr,left,i-1);
        quickSort(arr,i+1,right);
    }
}
```

insertion sort

This is a modified version of the insertion sort algorithm. Instead of sorting the entire list, it sorts a piece of the list as defined by the indices passed to the function

Python   C++

```python
def insertion_sort(mylist, left, right):
    for i in range(left + 1, right + 1):
        curr = mylist[i]  # store the first number in the
unsorted part of
                          # of array into curr
        j = i
        # this loop shifts value within sorted part of array to
open a spot for curr
        while j > left and mylist[j - 1] > curr:
            mylist[j] = mylist[j - 1]
            j = j - 1
        mylist[j] = curr
```

```c
/*performs the insertion sort algorithm on array from index left
to index right inclusive.*/
void insertionSort(int arr[],int left,int right){
    int curr;
    int i, j;
    for(i=left+1;i<=right;i++){
        curr=arr[i];
        for(j=i;j>left && arr[j-1] > curr;j--){
            arr[j]=arr[j-1];
        }
        arr[j]=curr;
    }
}
```

Partitioning Function

This is the fast partitioning algorithm that splits up the array according to a pivot value

Python   C++

```python
import random

def partition(mylist, left, right):
    # choose a random index between left and right inclusive
    pivot_location = random.randint(left, right)

    # get the pivot
    pivot = mylist[pivot_location]

    # move the pivot out of the way by swapping with
    # last value of partition.  This step is crucial as pivot will
    # end up "moving" if we don't get it out of the way which will
```

```c
int partition(int array[],int left, int right){

  //choose a random index between left and right inclusive
    int location = left + (rand()%(right-left+1));

    //get the pivot
    int pivot = array[location];

    //move the pivot out of the way by swapping with
    //last value of partition
    array[location] = array[right];
    array[right] = pivot;             //move the pivot out of the
way


    int i=left-1;
    for(int j=left;j<right;j++){
       if(array[j]<=pivot){
          i=i+1;
          int tmp=array[i];
          array[i]=array[j];
          array[j]=tmp;
       }
    }

    //restore pivot so that it is just after i
    int tmp=array[i+1];
    array[i+1]=array[right];
    array[right]=tmp;
    return i+1;
}
```

Lists

A list is an **sequence of values**. It may have properties such as being sorted/unsorted, having duplicate values or being unique. The most important part about list structures is that the data has an ordering (which is not the same as being sorted). Ordering simply means that there is an idea that there is a "first" item, a "second" item and so on. The position within the list where a piece of data is stored may be as important a piece of information as the data itself.

It should be noted at this point that the list we are speaking of here is not the same as a list in Python or a list from the C++ STL. A list here is much more general.

Lists typically have a subset of the following operations:

- initialize
- add an item to the list
- remove an item from the list
- search
- sort
- iterate through all items
- and more...

A list may implement only a subset of the above functionality. The description of a list is very general and it can be implemented in a number of different ways.

Two general implementation methods are the array method or the linked list method. We will look at each in turn.

What are Linked Lists?

This section provides you with a high level introduction to the linked list data structure.

If you have ever used a list from the C++ STL, that is implemented as a linked list... so while you may not have actually implemented it, you might have used it. However there are definitely some important issues to consider when using a linked list. This section will provide a brief introduction to linked list and its concepts. Note that Python list is not linked list based.

Introduction

A linked list is a data structure that stores a collection of data objects in a sequential manner. Each data object in list is stored in a **node**. Each node consists of a single data object along with at least one pointer to another node, typically the next node within the list. The linked list object itself must contain a pointer to a node within the list that will allow access to every other node by following the pointers in the node. Usually this is a pointer to the first node in the linked list.

If you wanted to store an array of 5 numbers in an array this is what the array would look like:

| 2.5 | 1.3 | 6.4 | 3.6 | 5.1 |

A simple linked list that stores the same 5 numbers in the same sequence

would look like this
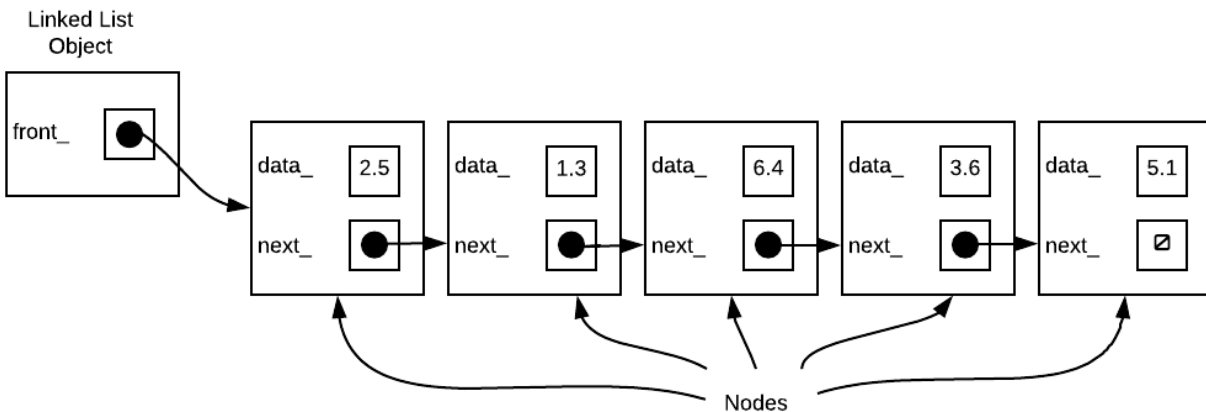
The above is a **_singly linked list_**. It is called this because there is a single pointer out of each node to another node.

> ⚠️ **CAUTION**
>
> Note that while the above diagram only has a single pointer out of the linked list object itself, it is not part of the definition of a singly linked list. The single link only refers to the link from one node to another, the fact there is only one link to the set of nodes.

Due to the way they are implemented, it is very fast to reach the first node in the linked list. However, to get to any other node, we must iterate through the list from node to node. Thus to get to the 5th node, we have to start at the first and follow the next pointers until we reach the 5th one.

This is different from an array. When you have an array, if you wanted to access the 5th item in the array, you would simply access array[4] which can be done in constant time because under that array is a mathematical calculation that is not affected by the array size.

Typical Operations

To look at how to implement a linked list, the operations a linked list can do should be considered. A linked list can typically support some subset of the following operations.

- push_front - add an item to the front of the linked list
- push_back - add an item to the back of the linked list
- pop_front - remove the frontmost item from the linked list
- pop_back - remove the backmost item from the linked list
- insert - given a point within the list insert an item just before that point
- erase - remove a node at a specific point within the list
- erase(a,b) - erases all nodes between a and b
- traversals - some operation that applies to every node in the list

Enhancements

To support the above operations, a linked list typically will store more information than the basic list. We will detail these in the implementation section

List Implementation

There are two typical ways to implement a list. The first is to use an array like data structure, the second is to use a linked list data structure (more on this below). The python builtin list structure is array based, while the C++ STL list is linked list based.

There are advantages and disadvantages to each implementations. Depending on the use case, there may be advantages with one implementation vs another.

Array based Implementation

Advantages

- items are stored in memory consecutively and you can have direct access to any particular item through the use of its index in constant time.
- When sorted, the list can be searched using binary search, providing a speedy search through large data sets.

Drawbacks

- space is often wasted as arrays are typically created bigger than needed.
- Insertion into anywhere other than the very end of the array is an expensive operation as it requires the shifting of all values from the point of insertion to the end of the array.
- Removal of any value anywhere other than the very last item is also expensive as it also requires a shift in all items from the point of removal to the very end of the list.

- growing the array can be amortized (average out the linear cost of single grow() spread out over all list operations) to $O(1)$, but when it does occur, that one single call to grow at that moment is linear, this may need to be considered if list is used in an application that has latency restrictions.

Linked list Implementation

Advantages

A linked list is an implementation that stores data in nodes. These nodes are linked together one after another.
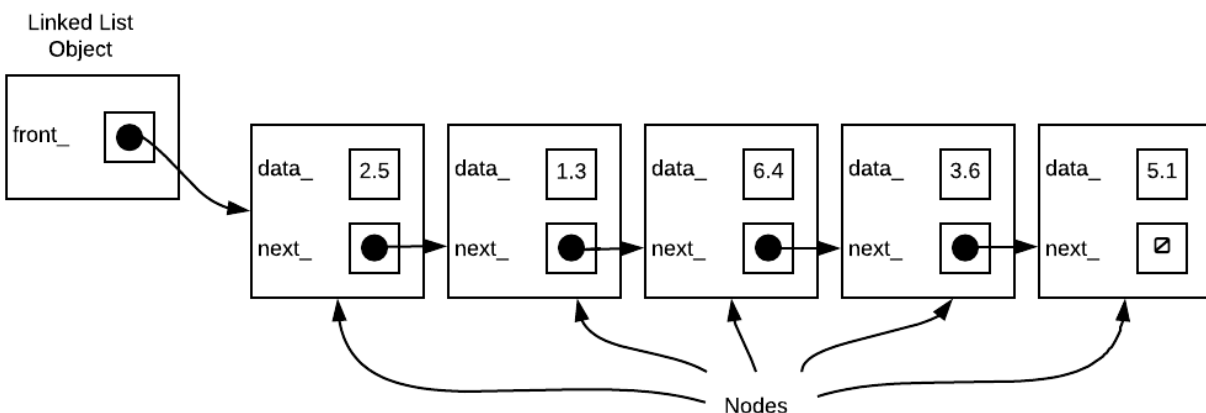
- Linked lists are very easy to grow and shrink as nodes only exist if there is data stored in them. When you grow the linked list, old nodes do not need to be duplicated as part of the grow process.
- Nodes are only created if there is data to store. No need to preallocate extra space
- Data is not stored in consecutive memory locations so a large block of contiguous memory is not required even for storing large amounts of data.
- Both insertion and removal of any node in the list (assuming that the

position of the insertion/removal is known) can be very efficient and runs in constant, O(1) time as it would only require a change of a few pointers. Exactly how long it takes depends on the type of linked list and the exact operations being performed. The key however is that when values are added or removed from a linked list, other values in the list are not moved around.

Drawbacks

- Each piece of data requires the storage of at least one extra pointer. When an array is full, it uses less memory than a linked list of the same size. Furthermore, if the data being stored in each node doesn't require much memory, then the pointer cost relative to the data stored can be significant. For example, an integer takes just as much room to store as a pointer. If you have a singly linked list of integers, the storage of each piece of data is double that of storing it into an array. Thus any array that is more than 50% full is storing more integers with same amount of storage as a singly linked list with the same amount of data.

- A linked list cannot be searched using binary search as direct access to nodes are not available.

- Data is not necessarily stored consecutively, this will mean that hardware advantages such as caching won't apply. If your program requires you to do something with all the data in the list, this can have significant impact on performance.

Memory Requirements

To implement a list using arrays, we allocate more space than is necessary. The array is used until it is full. Once an array is full, either all new insertions fail until an item is removed or the array must be reallocated. The reallocation

typically involves creating a larger array, copying over the old data, and making this the array. In order to amortize the cost of the grow operation over many list operations, implement the grow() by doubling the size of the array.

> ⚠️ **CAUTION**
>
> Growing an array can be an expensive operation which may require the duplication of all n elements of the array. Growing by some constant number of elements ensures poor performance. Always double.

To implement a list using a linked list like structure, each value is put into its own data node. Each node in the list is then linked with the next node by storing the address of the next node in the list. The memory usage in this case is at least one extra pointer for each piece of data. However, nodes are created on an as needed basis. There are never have unused nodes. Thus, the amount of memory used to store data in a linked list structure is typically lower than an array until the array is nearly full.

Implementation Concepts

To understand how to implement a linked list, we will want to look at the parts we will need to implement in order to create one.

Nodes

The basic unit of storage for a linked list is a **node**. A node stores data and one or more pointers to other nodes. At its most basic, a linked list node consists of data plus a pointer to the next node in the linked list. It is possible for a node to also store a pointer to the previous node in the linked list.

A linked lists where each node contains only a single pointer to the next node is called a **singly linked list**.

A linked list where each node contains two pointers one to the next node and one to the previous node is called a **doubly linked list.**

> 💡 **TIP**
>
> The idea of storing data in a node is not unique to linked lists. Other data structures such as trees also store data in nodes

Iterators

Iterators allows you to traverse a container class without actually knowing what the underlying container actually is or how that container may be

implemented. You can observe their usage in container classes within the C++ Standard Library such as <vector> and <list>. You can also see it in Python list and dictionaries. A linked list is a container class. Thus, our implementation of a linked list should also include the concept of iterators. Iterators also allow us to traverse the list in various ways (backwards and forwards for example). The idea is simply to separate the thing that allows us to go through a list with the underlying data structure

Iterators should support the following functionalities at minimum. Note that these are just general ideas. They are not language specific:

- ***first*** - set iterator to refer to the first item in a container
- ***next*** - sets iterator to the next item in the container
- ***isDone*** - returns true if iterator is not refering to anything
- ***currentItem*** - returns the current piece of data

When thinking about iterators, the important concept is that an iterator lets us go through a container one data item at a time. It provides a uniform method to go through the container and access the data. While you may view it as being similar to a node pointer, it is not the same. To the user of the list, there should be no such thing as a node. They only have iterators. This is really important to remember.

> ⚠ **CAUTION**
>
> An iterator is NOT a node, a linked list or a pointer to these. An iterator essentially hides the fact that there are even nodes within our linked list. It provides access to a set of data stored within a container in a uniform manner. It limits the type of access allowed to only some operations. We use iterators so that there is a familiarity to how we would access our custom container classes. That is we make them work in a similar manner to those containers that are part of the language.

Linked List

And finally we have the concept of the linked list itself. This particular object holds very little data within itself. The only data that is absolutely required is the location of a single node in the list, the node that allows you to follow links to every other node. Usually this is the first node in the list but it does not have to be the first node. This largely depends on the type of list we are coding and what we are trying to achieve.

Implementation - List and Nodes

A linked list is a container class. This means that the operations of a linked list are not affected by the types of data it may store. In C/C++ we would implement this using templates. In Python, we really don't have to worry about this. Our implementation will be a partial implementation of a doubly linked list with iterators. This section should be read as a guide on how to implement a linked list. It will start with ideas and diagrams then move on to show the implementation of the idea.

Linked List

A linked list object (we will call ours DList as it will be a doubly linked list), must store at least one pointer to the first node within the linked list. However, to speed up the implementation of certain functions, it is advantageous to store a pointer to the last node also.

Python   C++
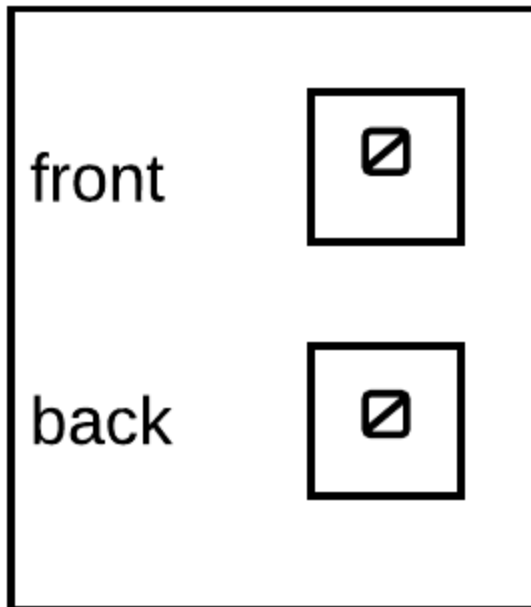
```python
class LinkedList:
    def __init__(self):
        self.front = None
        self.back = None
```
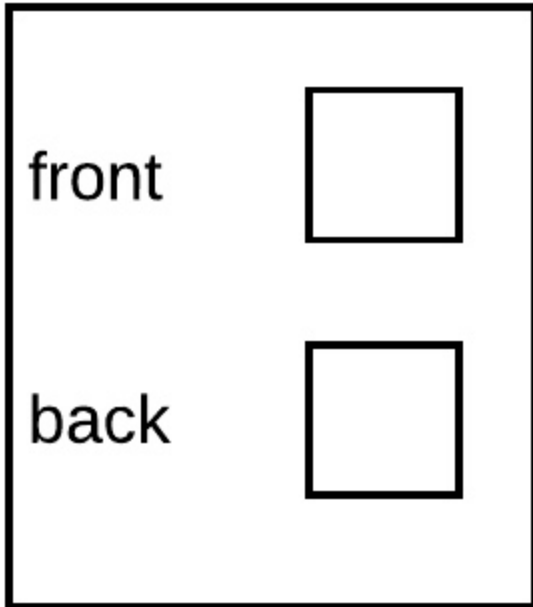
```
template <typename T>
class DList{

    Node* front;
    Node* back;
...
};
```

Nodes

A node is the object that stores the data. An instance of a linked list does not hold any nodes itself. It instead points to a set of nodes that are linked together. Each node object contains a piece of data and two pointers.
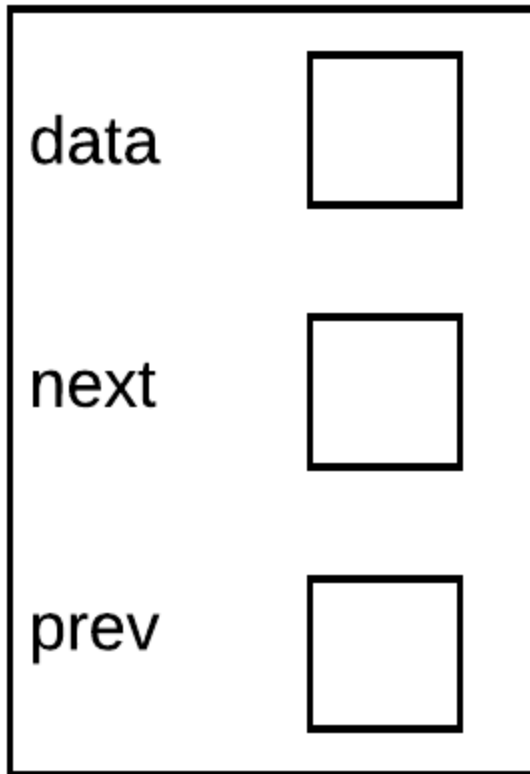
The idea of nodes are not unique to linked lists. Other container classes such as trees can also have nodes. As the lists may actually be used in conjunction with other container classes, we will want to hide our nodes. This is not just hiding the data within the node but rather hide the idea that nodes even exist at all. To do this, we will declare the node within the DList class itself. Aside from declaring data members, it is also a good idea to provide a constructor with defaults to simplify the creation of Nodes.

Python   C++

```python
class LinkedList:
    class Node:
        def __init__(self, data, next=None, prev=None):
            self.data = data
            self.next = next
            self.prev = prev

    def __init__(self):
        self.front = None
        self.back = None
```

> ⓘ **INFO**
>
> Above, you see the use of **None** which is python equivalent to a nullptr in C++. The **None** is a value to indicate that the reference refers to nothing. Any reference should always either refer to some object or it should be **None**. Checking to see if a reference is **None** means you are checking to see if it refers to nothing. **None** is considered to be False equivalent. That is if a reference is set to **None** and you check that reference, it will result in False.
>
> ```python
> ref = None
> if(ref):
>     print("reference refers to something")
> else:
>     print("reference refers to nothing (None)")
> ```

In C++ we declare the Node so that it is private to DList. As the Node is private to only DList, we will not need to make the Node a class. A struct will be fine and provide simpler access. This private declaration of the Node struct will also mean that outside of the DList class, there is no knowledge of the Node which is what we want.

```cpp
template <typename T>
class DList{
    struct Node{
        T data;
        Node* prev;
        Node* next;
        Node(const T& dat=T{},Node* nx=nullptr,Node* pr=nullptr){
            data=dat;
            prev=pr;
            next=nx;
        }
    };
    Node* front;
    Node* back;
...
};
```

> ⚠ **INFO**
>
> Above, you see the use of **nullptr** which is short for null pointer. The **nullptr** is a value to indicate that the pointer points to nothing. Any pointer should always either point to some object of its data type or to **nullptr**. Checking to see if a pointer is **nullptr** means you are checking to see if it points to nothing. **nullptr** is considered to be false equivalent. That is if a pointer is set to **nullptr** and you check that pointer, it will return false.

```
ptr=nullptr;
if(ptr){
    cout << "pointer points at something" << endl;
}
else{
    cout << "pointer points at nothing (nullptr)" << endl;
}
```

Linked List Constructor

The constructor of the linked list should set up the linked list object in a manner that indicates that there are no nodes within the linked list. This is fairly simple to implement and only requires that we initialize front_ and back_ to *nullptr*
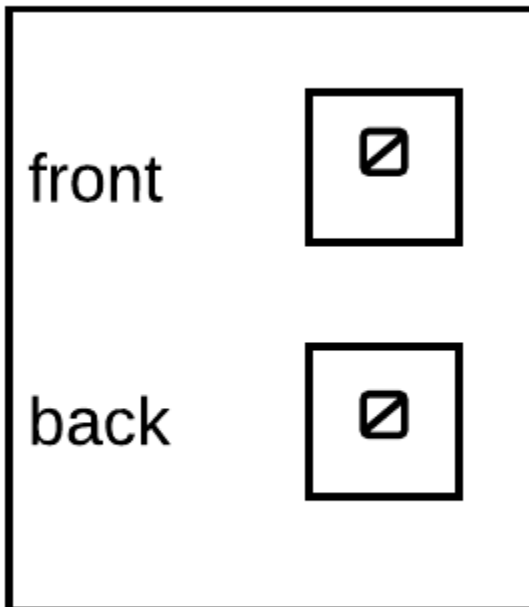
```cpp
template <typename T>
class DList{
...
public:
    DList(){
        front=nullptr;
        back=nullptr;
    }
...
}
```

push_front and pop_front

Implementation - push_front(), pop_front()

This section of the notes walks through the implementation of push_front() and pop_front()

Python   C++

push_front(data)

The push_front function adds a node to the front of the linked list. As with all insertion of nodes, the process can be generally described in the following manner:

1. create a new node with the appropriate data
2. link the new node up with the rest of the list

Let us start by considering how this would work in a general linked list. The steps are:

Step 1: Create new node, next node is the current front of list, there is no previous node which means that next node's prev should be set to a nullptr:
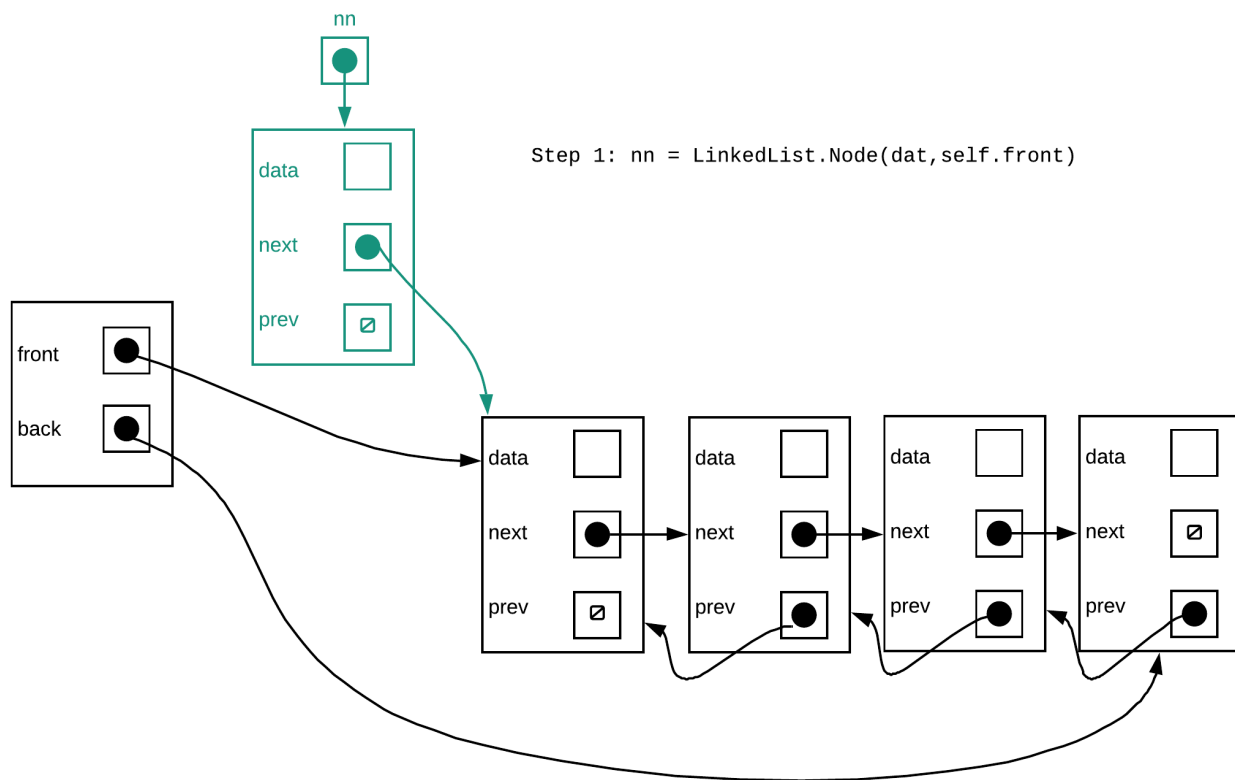
nn

data

next

prev

Step 1: nn = LinkedList.Node(dat,self.front)

front

back

data

next

prev

data

next

prev

data

next

prev

data

next

prev

Step 2: Make the previous node of the current front of list point to the new node

nn

data

next

prev

step 2: self.front.prev = nn

front

back

data

next

prev

data

next

prev

data

next

prev

data

next

prev
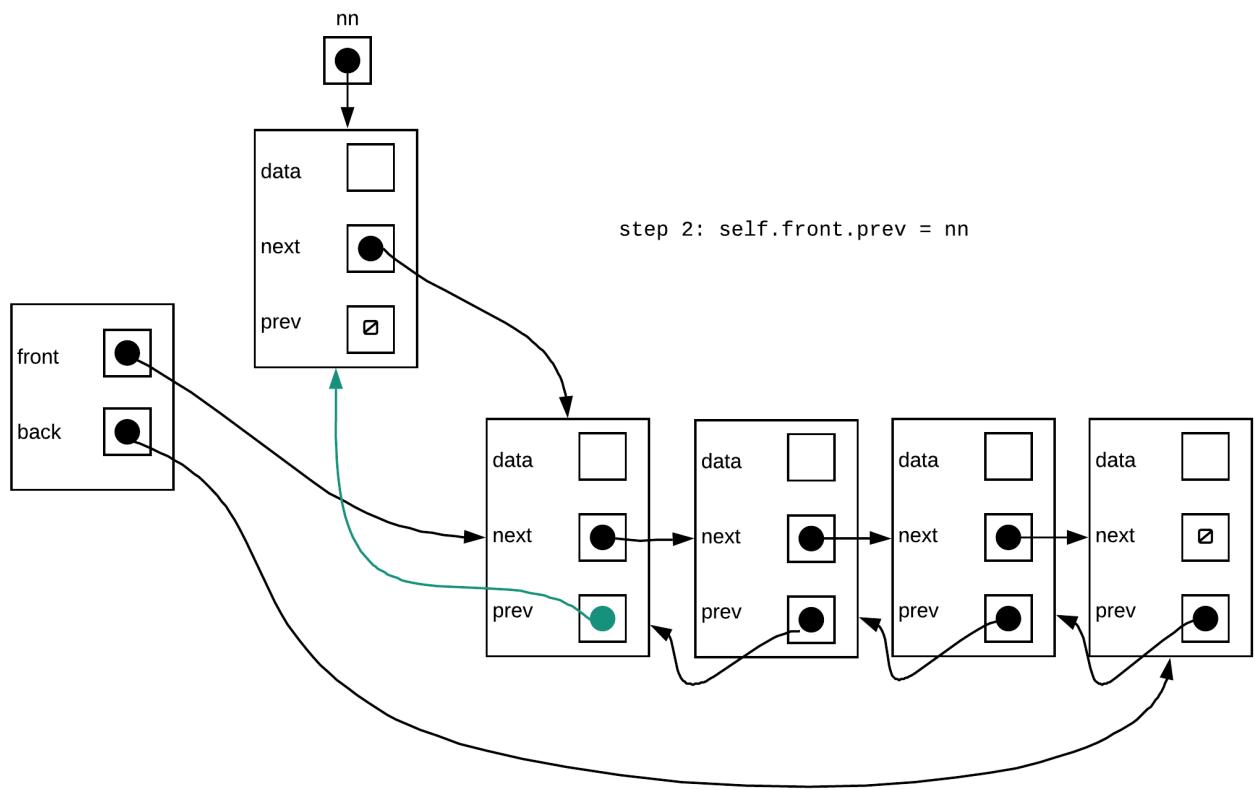
Step 3: Make the front pointer point to the new node
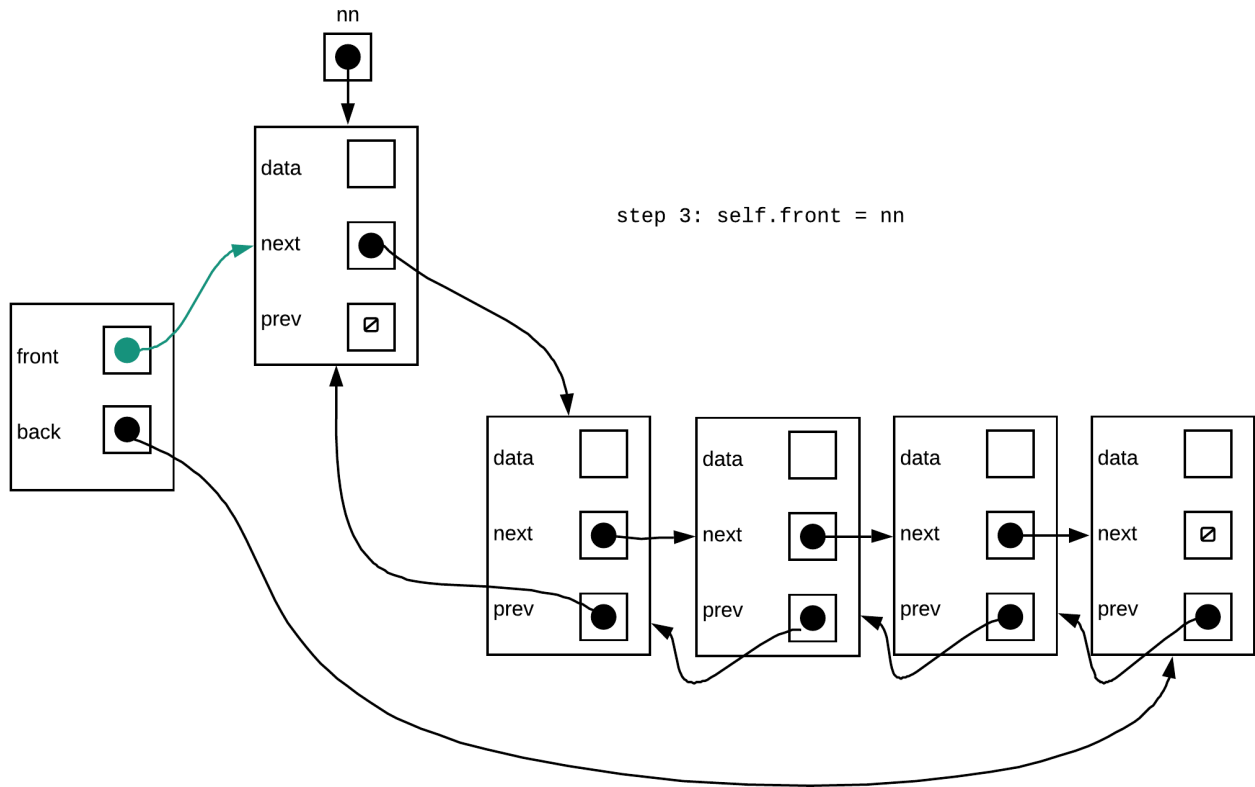
step 3: self.front = nn

Putting this together in code:

```
nn = self.Node(data, self.front)
self.front.prev = nn
self.front = nn
```

Does above always work?

Now lets consider if the following would work if we started off with a linked list that is empty.

Step 1:

nn
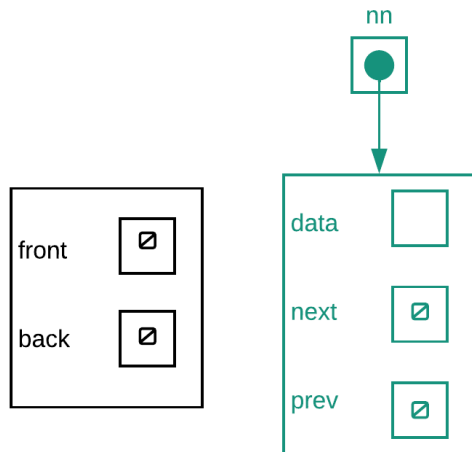
Step 1: nn = LinkedList.Node(data,self.front)

Step one will run without problems

Step 2:

nn
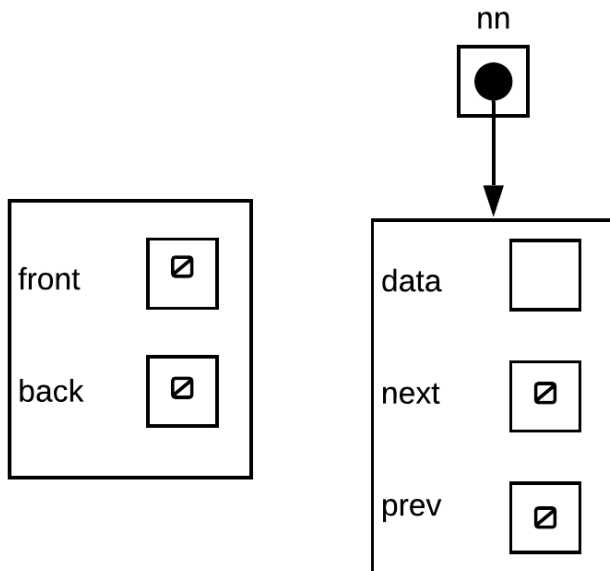
Step 2: self.front.prev = nn

self.front.prev would cause a crash because front is None

self.front is None, so self.front.prev will crash. Thus, it looks like we need to skip this step or do something different if we have an empty list

Step 3:

step 3: self.front = nn

The above will work. However, if we were to simply skip step 2, our linked list would not be valid as back would not be correctly set. The node we just added is not only the first node in the linked list but also the last. To make this work, we should add code to make back point to nn.

Putting all this together, The final function would look like the following:

```python
def push_front(self, data):
    nn = self.Node(data, self.front)
    if self.front is None:
        self.back = nn
    else:
        self.front.prev= nn
    self.front = nn
```

pop_front()

The pop_front() function removes the first node from the linked list. The following are the general steps to remove a node:

1. check to make sure that the list isn't empty (or that the node to be removed actually exist).
2. unlink the node to be remove from the list (ensure other nodes are not lost in the process)
3. deallocate the memory for the node

Given the above steps, let us consider how this would work in the general case. The steps for performing a pop_front are:

Step 1: Check to make sure list isn't empty. If it is do nothing. Otherwise continue to next steps

Step 2: Make a local pointer point to the first node in the list (hold this node so we don't lose it by accident)

Step 2: rm = self.front

Step 3: Make the front pointer point to the second node

Step 3: self.front = self.front.next

Step 4: Make the new front node's previous pointer a nullptr

Step 4: self.front.prev = None
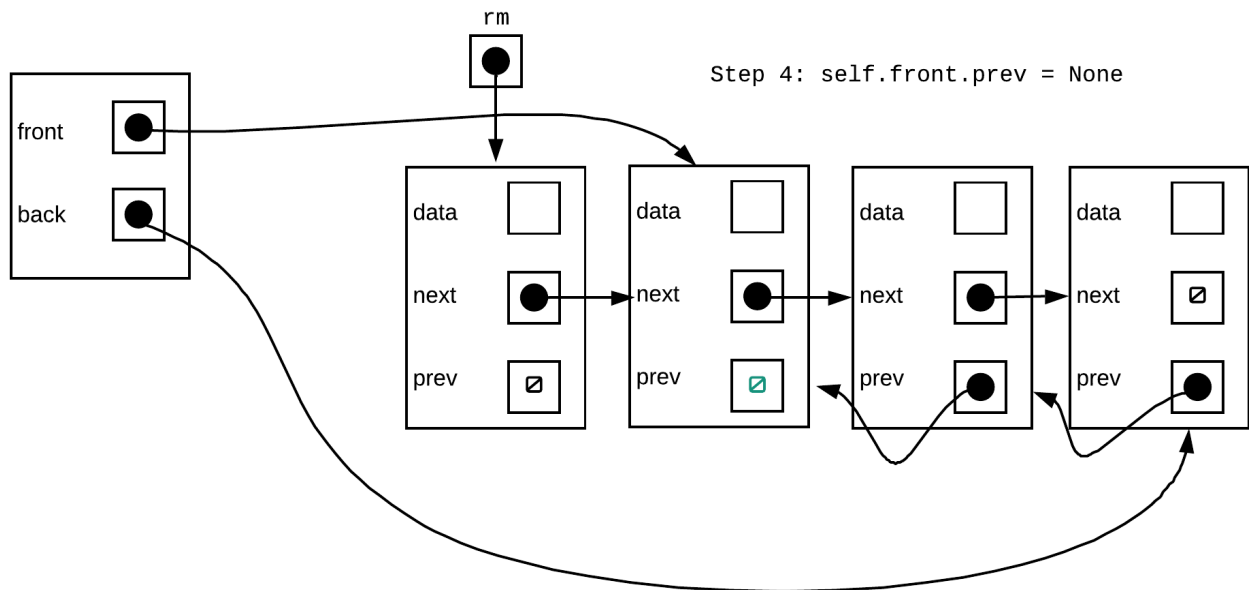
The code snippet to do this is:

```python
if self.front is not None:
```

Does the above always work?

The snippet of code above works for the general linked list and empty linked lists. However, does it always work? Let us consider the following situation. If we only had one node left in the list, would the above snippet still work?
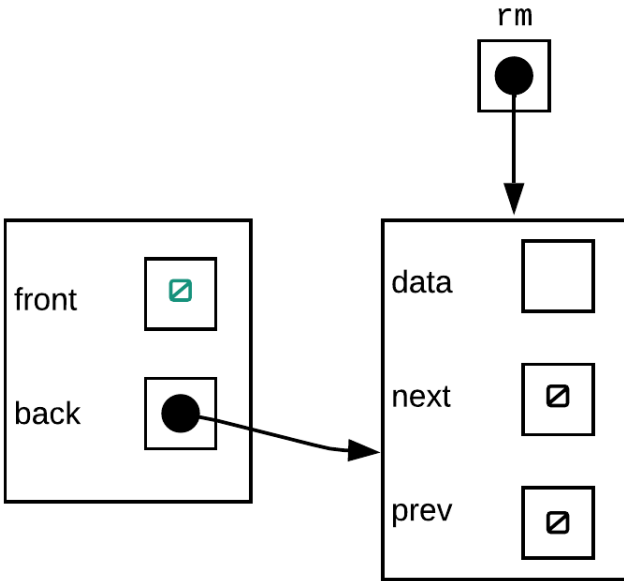
If we perform the four steps inside the if statement from the above snippet, this is what we will see:

Step 2:

Step 2: rm = self.front

Above looks fine.

Step 3:

rm

Step 3: self.front = self.front.next

front   ∅

back   ●

data

next   ∅

prev   ∅

This also looks correct

Step 4:

rm
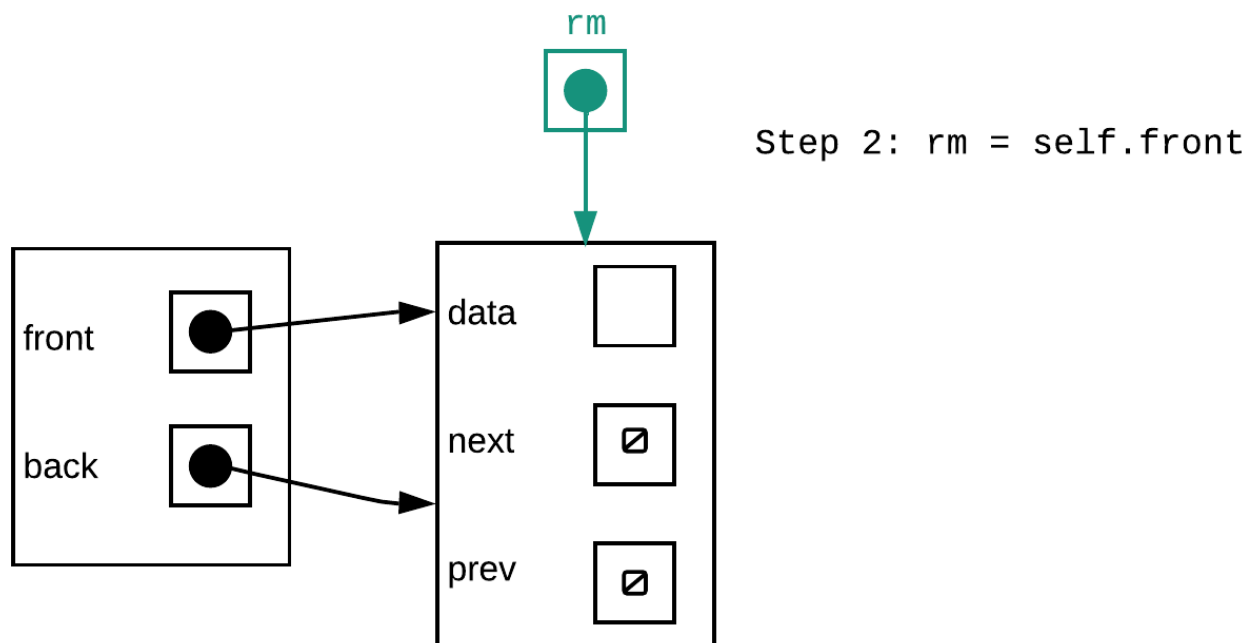
Step 4: front.prev = None
front.prev would crash because
front is None

front   ∅

back   ●

data

next   ∅

prev   ∅

If we were to try to do the above at this point we would end up crashing the program as front is currently nullptr. This step will either need to be skipped for lists with just one node or something differnt will need to be done

The list is not in a valid state. We have a back pointer that points to the

memory location of the node that should be deallocated. This pointer is therefore invalid. For lists with just one node, we must also adjust the back pointer to a nullptr/None (as the list should get empty after removing its only node.)

Putting all this together our pop_front() function should look something like this:

```python
def pop_front(self):
    if self.front is not None:
        rm = self.front
        self.front = self.front.next
        if self.front is None:
            self.back = None
        else:
            self.front.prev = None
        del rm
```

push_front(data)

The push_front function adds a node to the front of the linked list. As with all insertion of nodes, the process can be generally described in the following manner:

1. create a new node with the appropriate data
2. link the new node up with the rest of the list

Let us start by considering how this would work in a general linked list. The steps are:

Step 1: Create new node, next node is the current front of list, there is no previous node which should be set to a nullptr
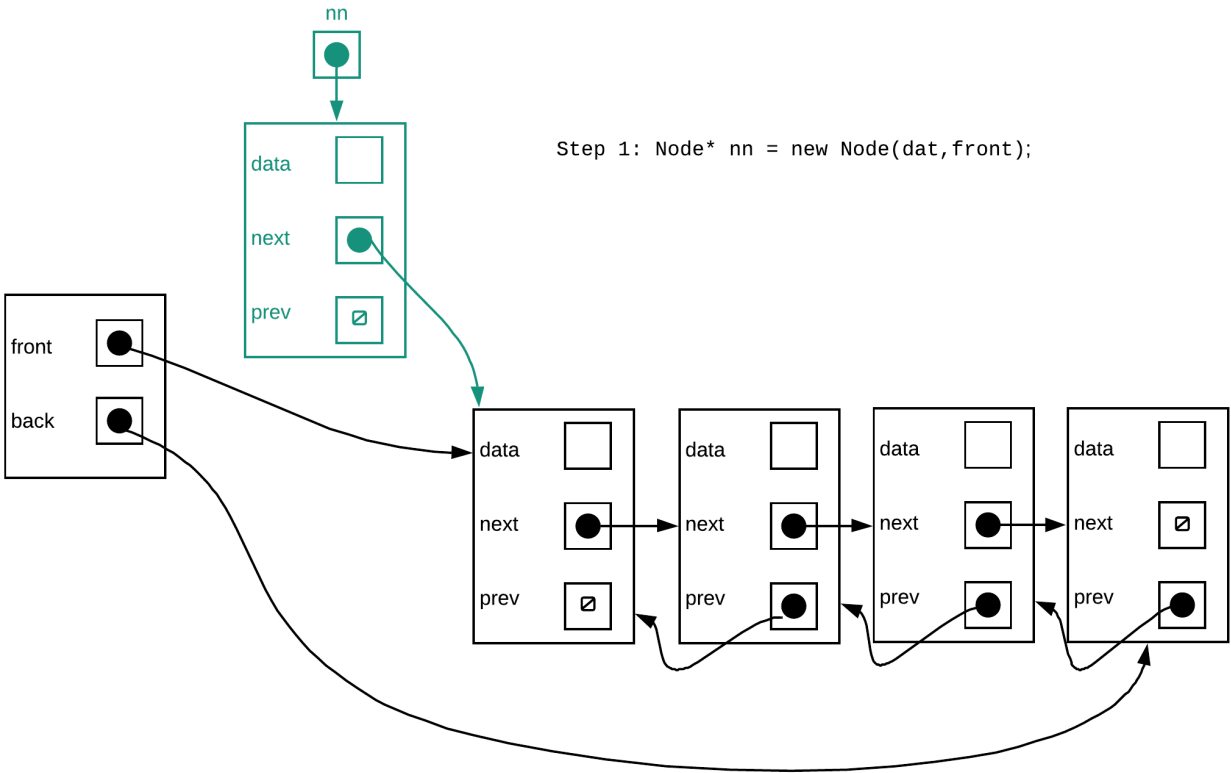
Step 1: `Node* nn = new Node(dat,front);`

nn

data

next

prev

front

back

data

next

prev

data

next

prev

data

next

prev

data

next

prev

Step 2: Make the previous node of the current front of list point to the new node

nn

data

next

prev

step 2: front->prev = nn;

front

back

data

next

prev

data

next

prev

data

next

prev

data

next
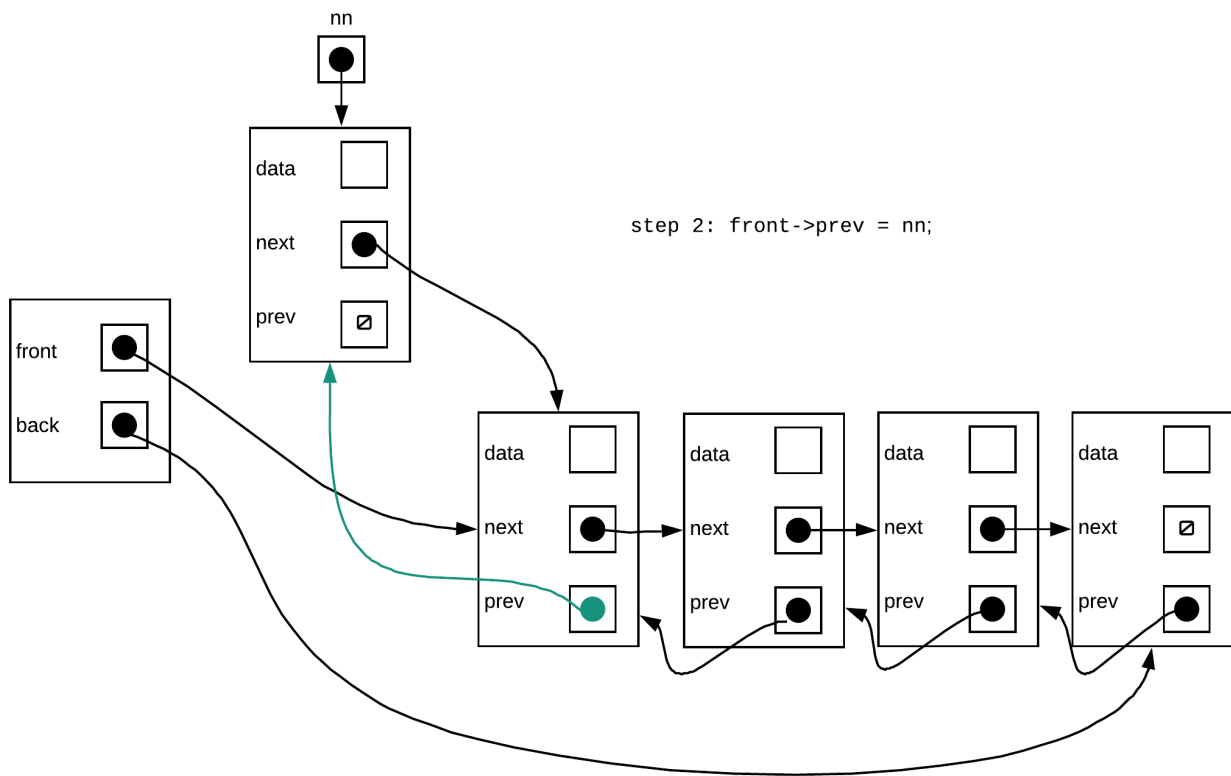
prev

Step 3: Make the front pointer point to the new node

step 3: front = nn;
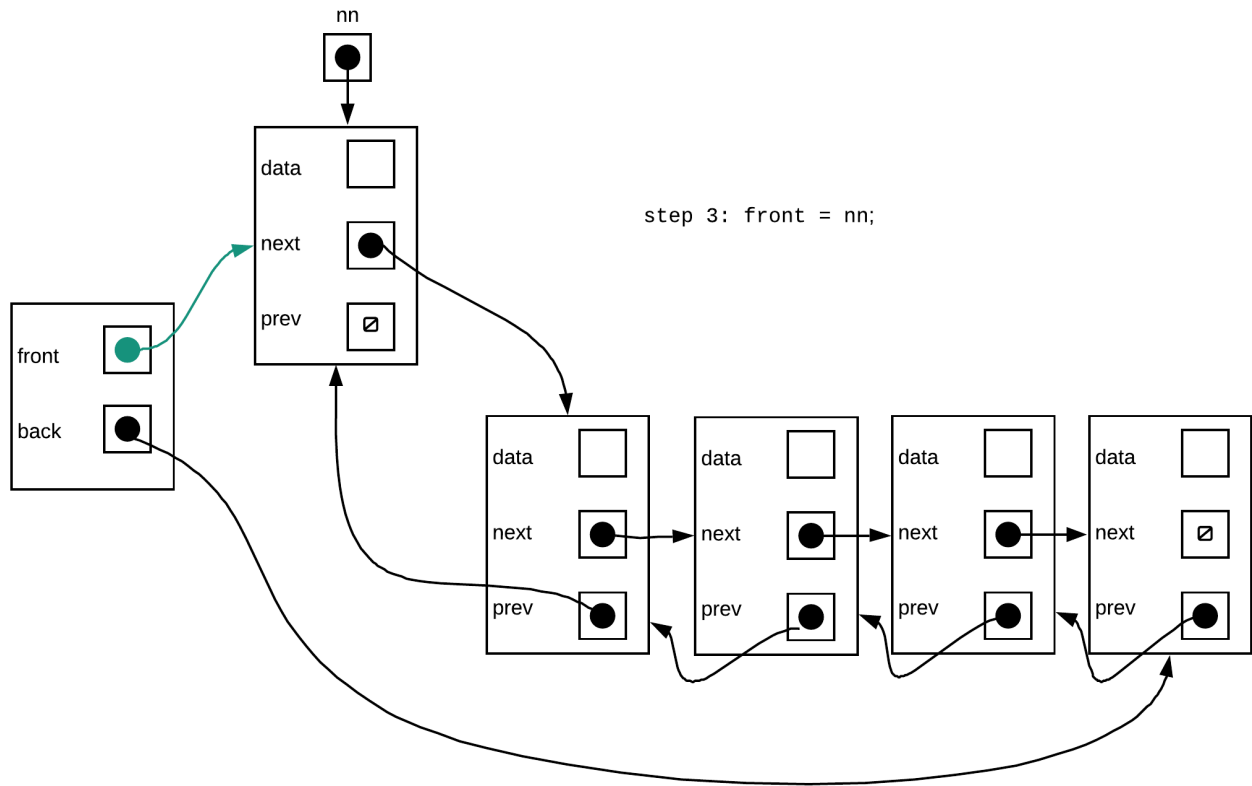
Putting this together in code:

```
Node* nn = new Node(data,front);
front->prev=nn;
front=nn;
```

Does above always work?

Now lets consider if the following would work if we started off with a linked list that is empty.
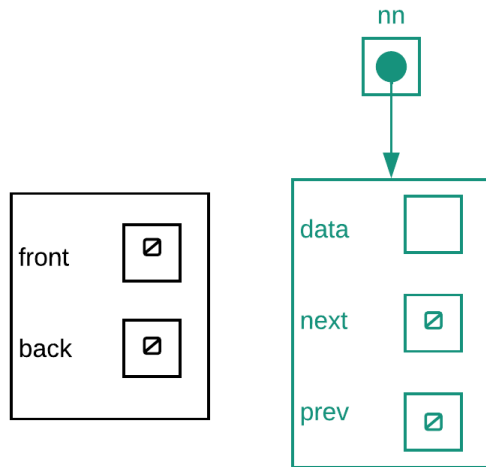
Step 1:

Step 1: Node* nn = new Node(data,front);

Step one will run without problems
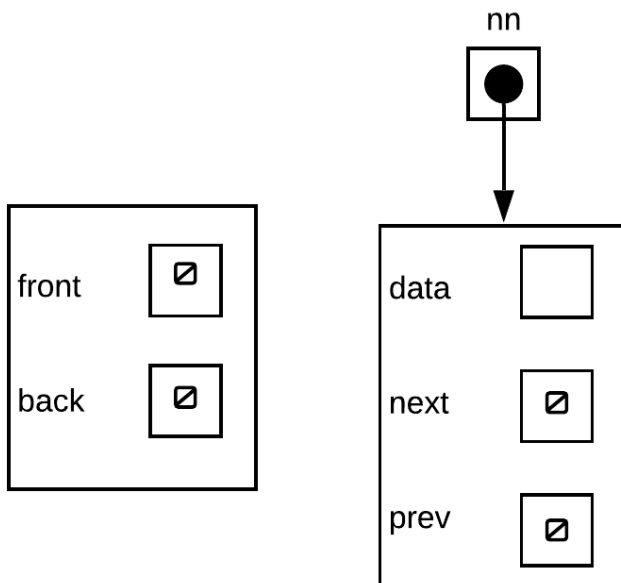
Step 2:

Step 2: front->prev = nn;

front->prev would cause a crash because front is nullptr

front is nullptr, so front->prev will crash. Thus, it looks like we need to skip this step or do something different if we have an empty list

Step 3:

step 3: front = nn;

The above will work. However, if we were to simply skip step 2, our linked list would not be valid as back would not be correctly set. The node we just added is not only the first node in the linked list but also the last. To make this work, we should add code to make back point to nn.

Putting all this together, The final function would look like the following:

```cpp
void push_front(const T& data){
    Node* nn = new Node(data,front);
    if(front){
        front->prev=nn;
    }
    else{
        back=nn;
    }
    front=nn;
}
```

pop_front()

The pop_front() function removes the first node from the linked list. The following are the general steps to remove a node:

1. check to make sure that the list isn't empty (or that the node to be removed actually exist).
2. unlink the node to be removed from the list (ensure other nodes are not lost in the process)
3. deallocate the memory for the node

Given the above steps, let us consider how this would work in the general case. The steps for performing a pop_front are:

Step 1: Check to make sure list isn't empty. If it is do nothing. Otherwise continue to next steps

Step 2: Make a local pointer point to the first node in the list (hold this node so we don't lose it by accident)

Step 3: Make the front pointer point to the second node

rm

Step 3: front = front->next

front

back

data

next

prev

data

next

prev

data

next

prev

data

next

prev

Step 4: Make the new front node's previous pointer a nullptr

rm

Step 4: front->prev = nullptr

front

back

data

next

prev

data

next

prev

data

next

prev

data

next

prev
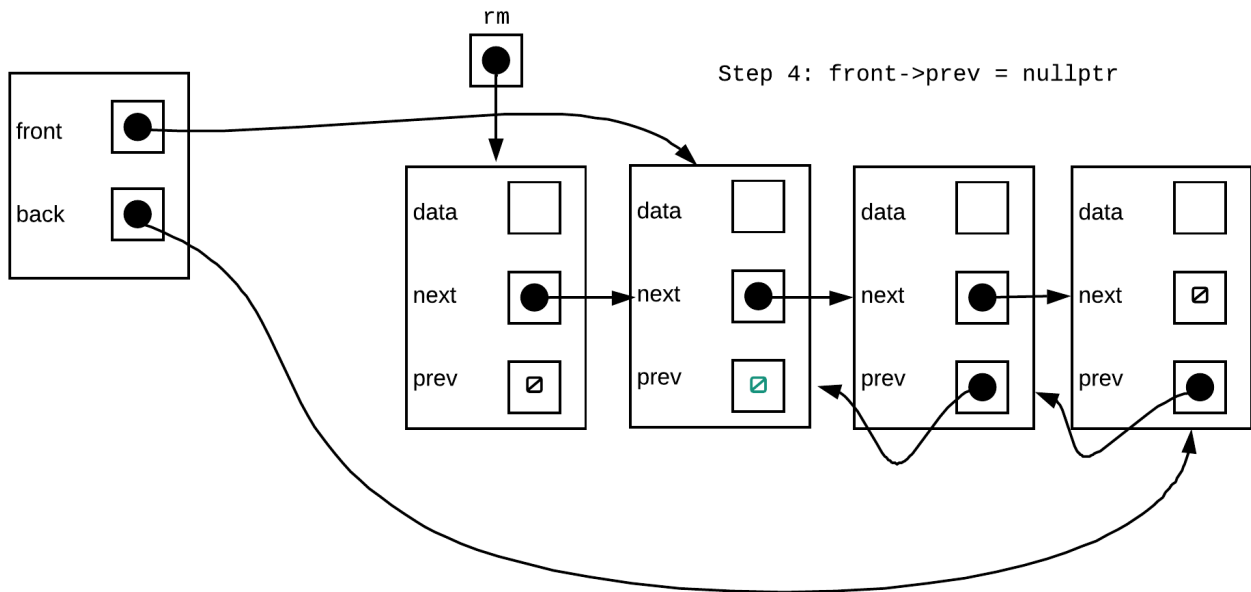
Step 5: deallocate the node we are removing

Step 5: delete rm;
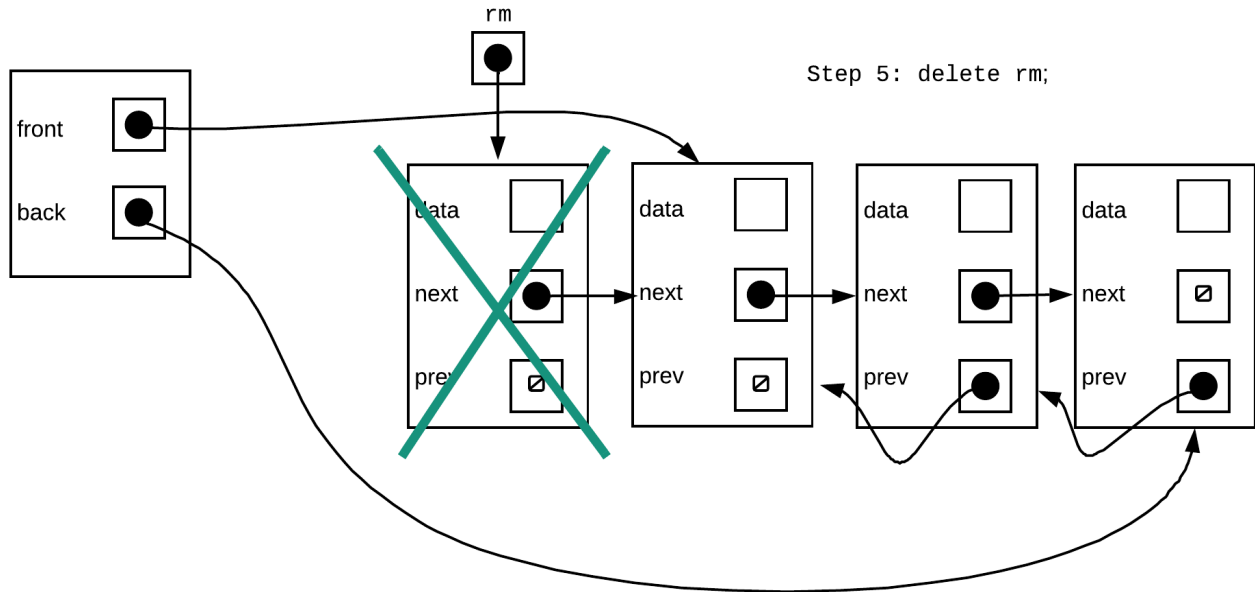
> **💡 TIP**
>
> **delete p;** does not deallocate the pointer **p** itself. Rather it deallocates what the pointer points at. Thus, **delete rm**; deallocates the node rm points at.

The code snippet to do this is:

```cpp
if(front){
    Node* rm = front;
    front=front->next;
    front->prev=nullptr;
    delete rm;
}
```

> **💡 TIP**
>
> A common misconception is that when you declare a pointer, you must

Does the above always work?

The snippet of code above works for the general linked list and empty linked lists. However, does it always work? Let us consider the following situation. If we only had one node left in the list, would the above snippet still work?

If we perform the four steps inside the if statement from the above snippet, this is what we will see:

Step 2:

Above looks fine.

Step 3:

rm

Step 3: front = front->next

front ☑

back ●────→

data

next ∅

prev ∅

This also looks correct

Step 4:

rm
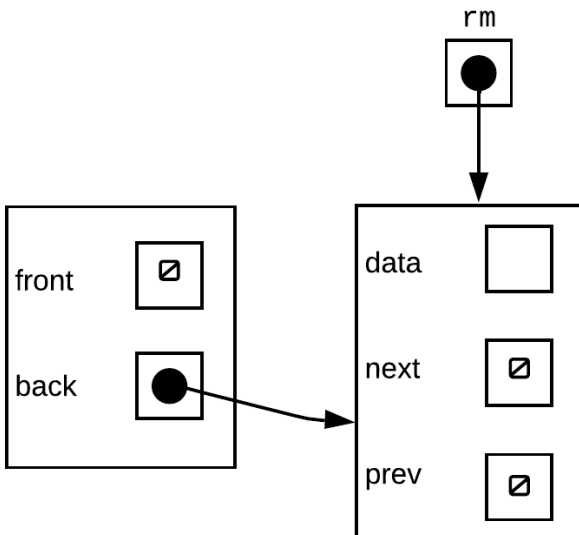
Step 4: front->prev = nullptr
front->prev would crash because
front is nullptr

front ∅

back ●────→

data

next ∅

prev ∅

If we were to try to do the above at this point we would end up crashing the program as front is currently nullptr. This step will either need to be skipped for lists with just one node or something differnt will need to be done

Step 5:

Step 5: delete rm;
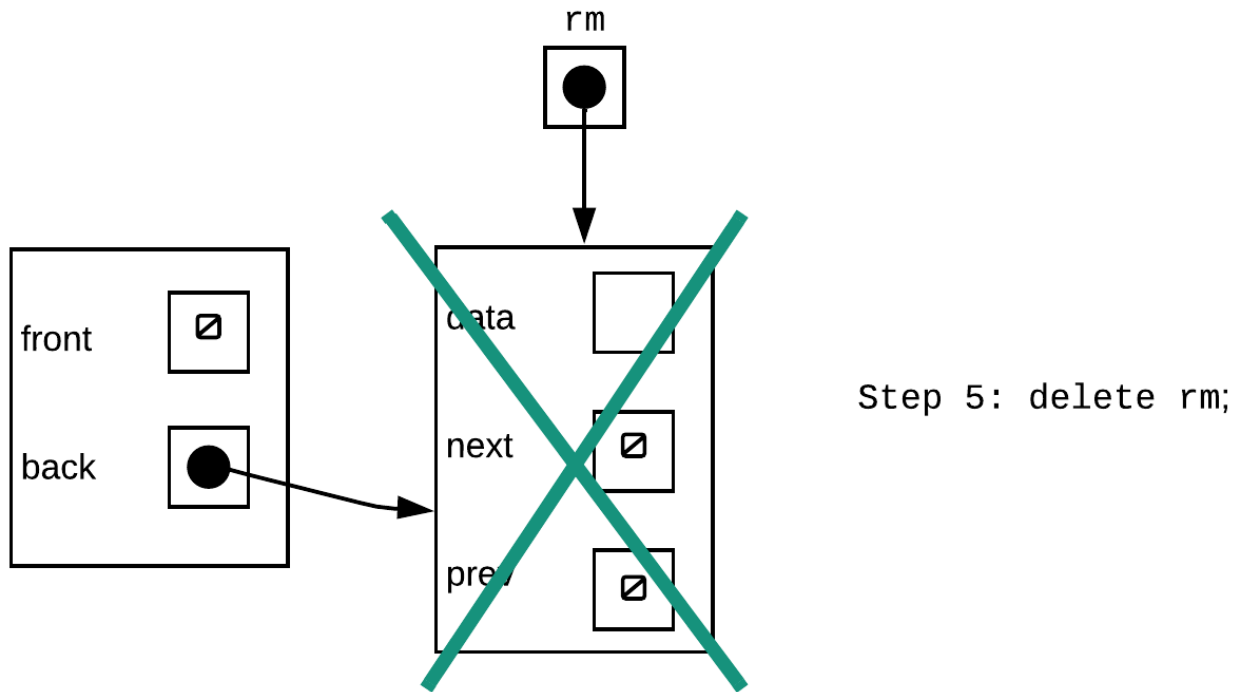
The final line is to delete the node, which won't crash. However, the list is not in a valid state. We have a back pointer that points to the memory location of the node that has now been deallocated. This pointer is therefore invalid. For lists with just one node, we must also adjust the back pointer to a nullptr as the list is now empty

Putting all this together our pop_front() function should look something like this:

```cpp
void pop_front(){
    if(front_){
        Node* rm = front;
        front=front->next;
        if(front==nullptr){  //if only one node exists
            back=nullptr;
        }
        else{
            front->prev=nullptr;
        }
        delete rm;
```

Modification - Sentinel Nodes

Consider the push_front()/pop_front() example we did early. Both functions required us to check for a special case and essentially write 2 different versions of the same function for those special cases (inserting into an empty list, removing from a list with only one node). One modification we can make to our linked list in order to simplify the functions is to add the idea of sentinel nodes.

Sentinel nodes are nodes that exist at the front and back of a linked list. These nodes always exist from the time the linked list is created to the time it is destroyed. They do not hold any data. The purpose for their existence is to eliminate most of the special cases when writing functions.

Most of the special cases in our implementations involve checking whether the front_/back_ pointers or next_/prev_ pointers are nullptr/None at the time or not. Sentinel nodes can help us dealing with these situations more easily by preventing these from happening, and let us have our code more simplified. In this section we will look at what it means to have sentinel nodes.

Linked List Constructor - With Sentinels

The constructor of the linked list should set up an empty linked list. With sentinels, this effectively means we create two nodes whose data is undefined and point them at each other. One is the front sentinel the other the back sentinel. These nodes do not hold any data. Their purpose is to reduce special cases.
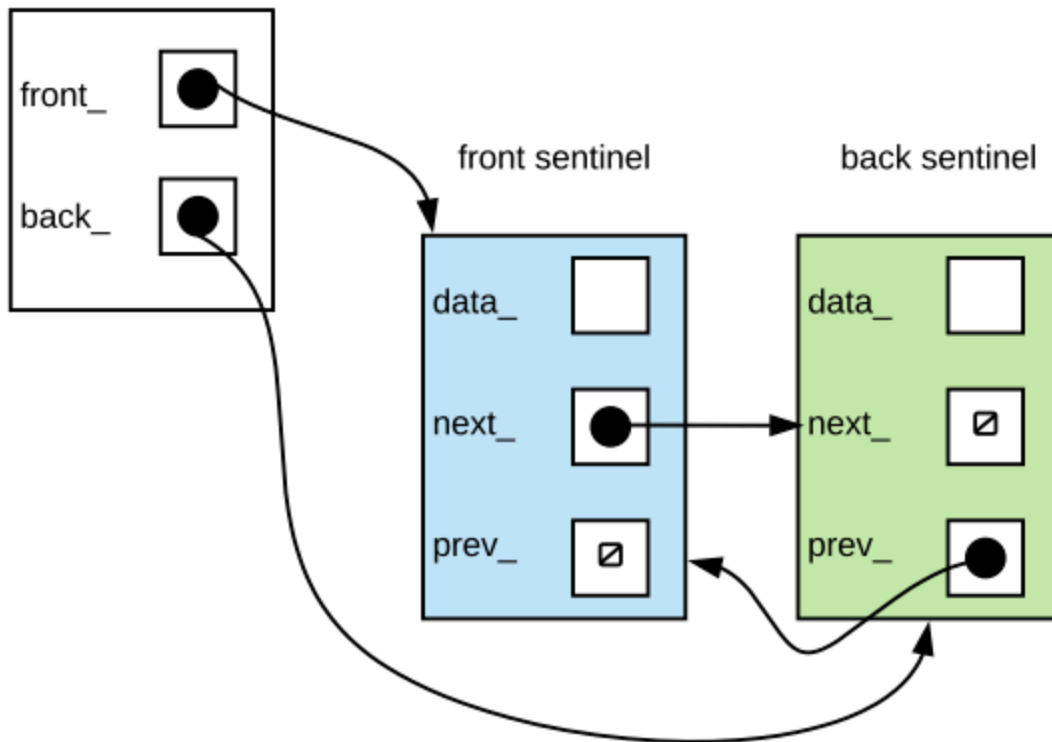
Empty linked lists for linked lists with sentinels will have 2 nodes. The front sentinel and the back sentinel. These are created on list construction.

Python   C++

```python
class LinkedList:
    class Node:
        def __init__(self, data, next=None, prev=None):
            self.data = data
            self.next = next
            self.prev = prev

    def __init__(self):
        self.front = self.Node(None)
```
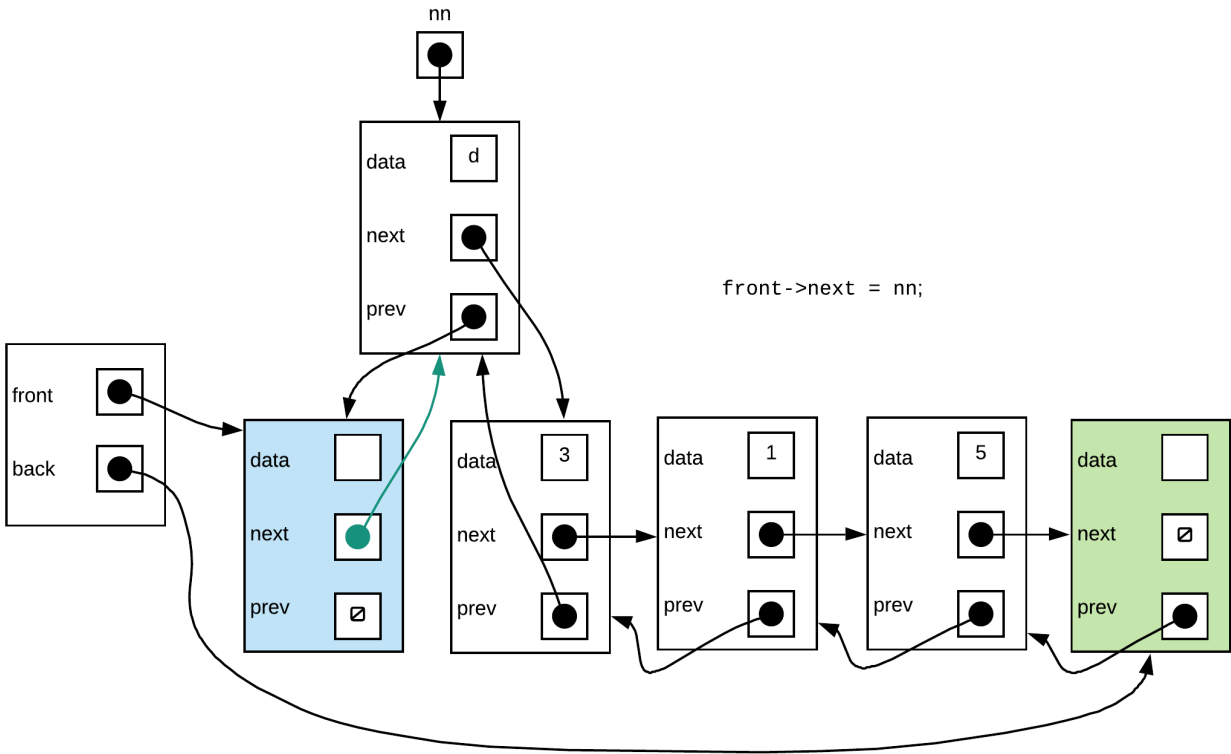
push_front()

How will push_front change with sentinels?

Step 1: Create new node, next node is the node that follows the front sentinel. The previous node is the front sentinel.


```
nn = LinkedList.Node(data,self.front.next,self.front)
```
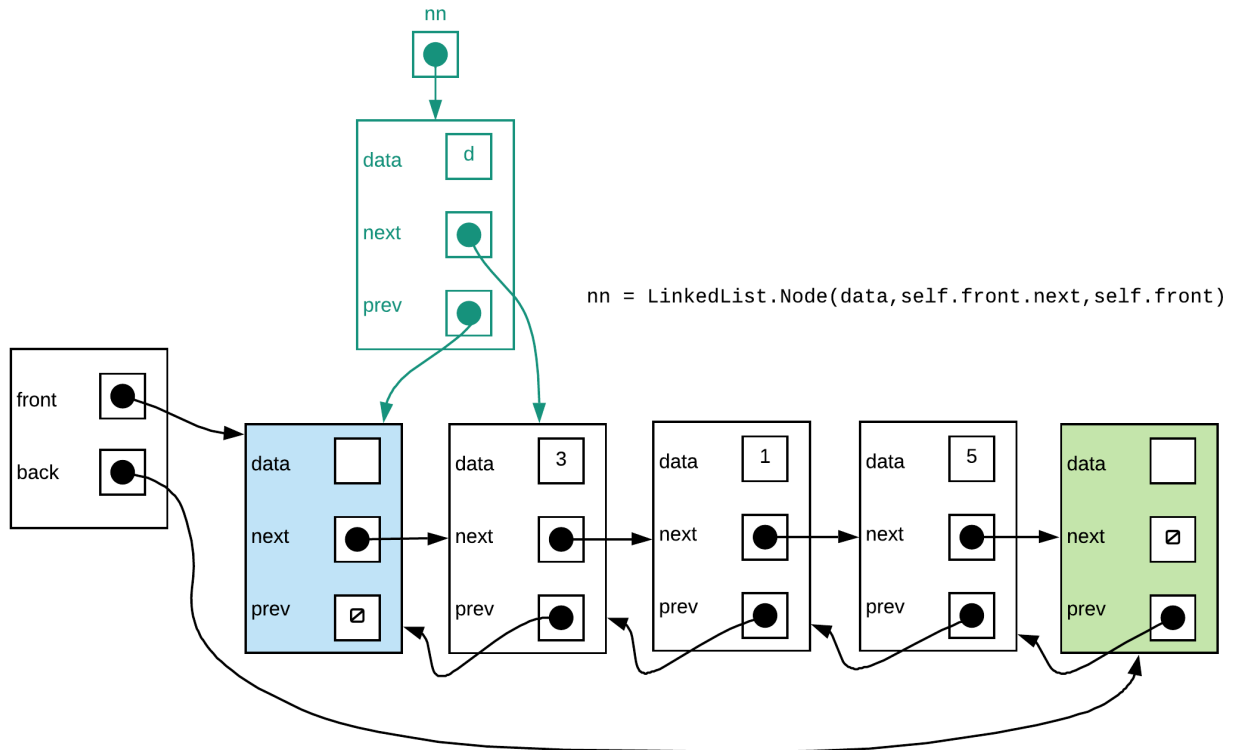
Step 2: Make the previous pointer of the node that follows the front sentinel point to the new node
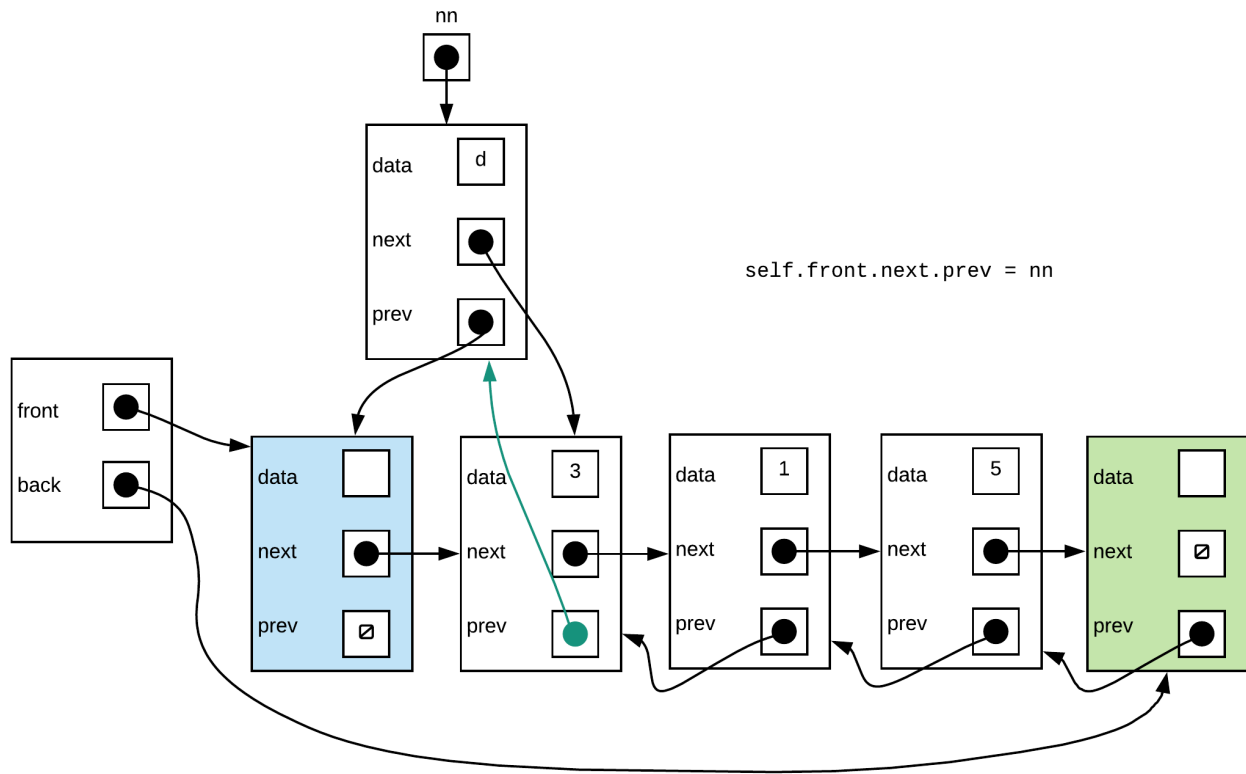
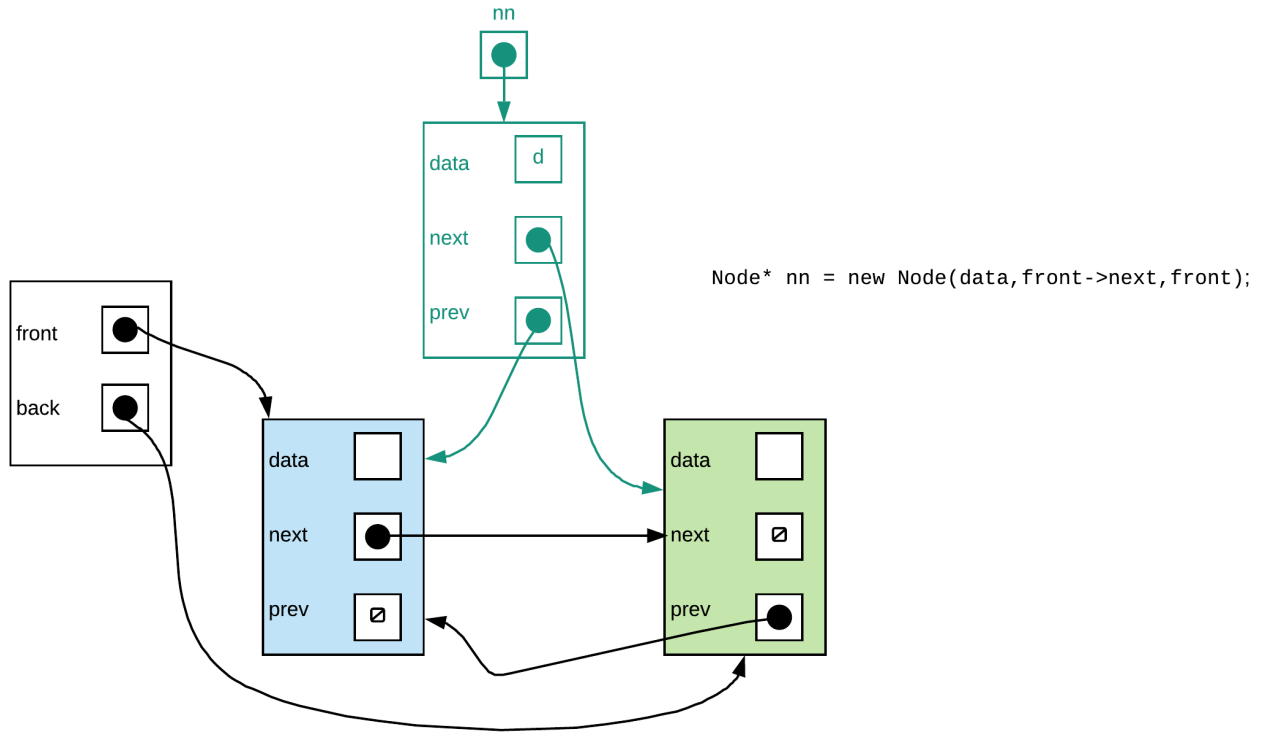Step 3: Set the next pointer of the front sentinel to the new node.
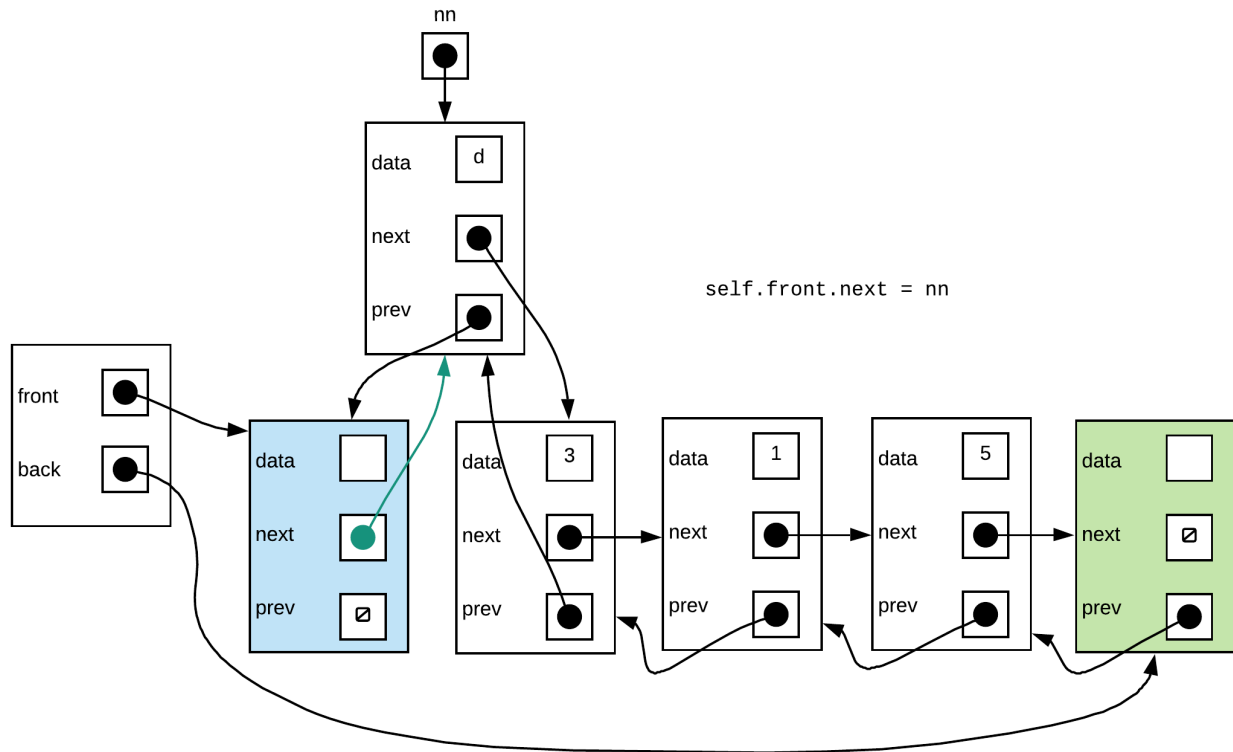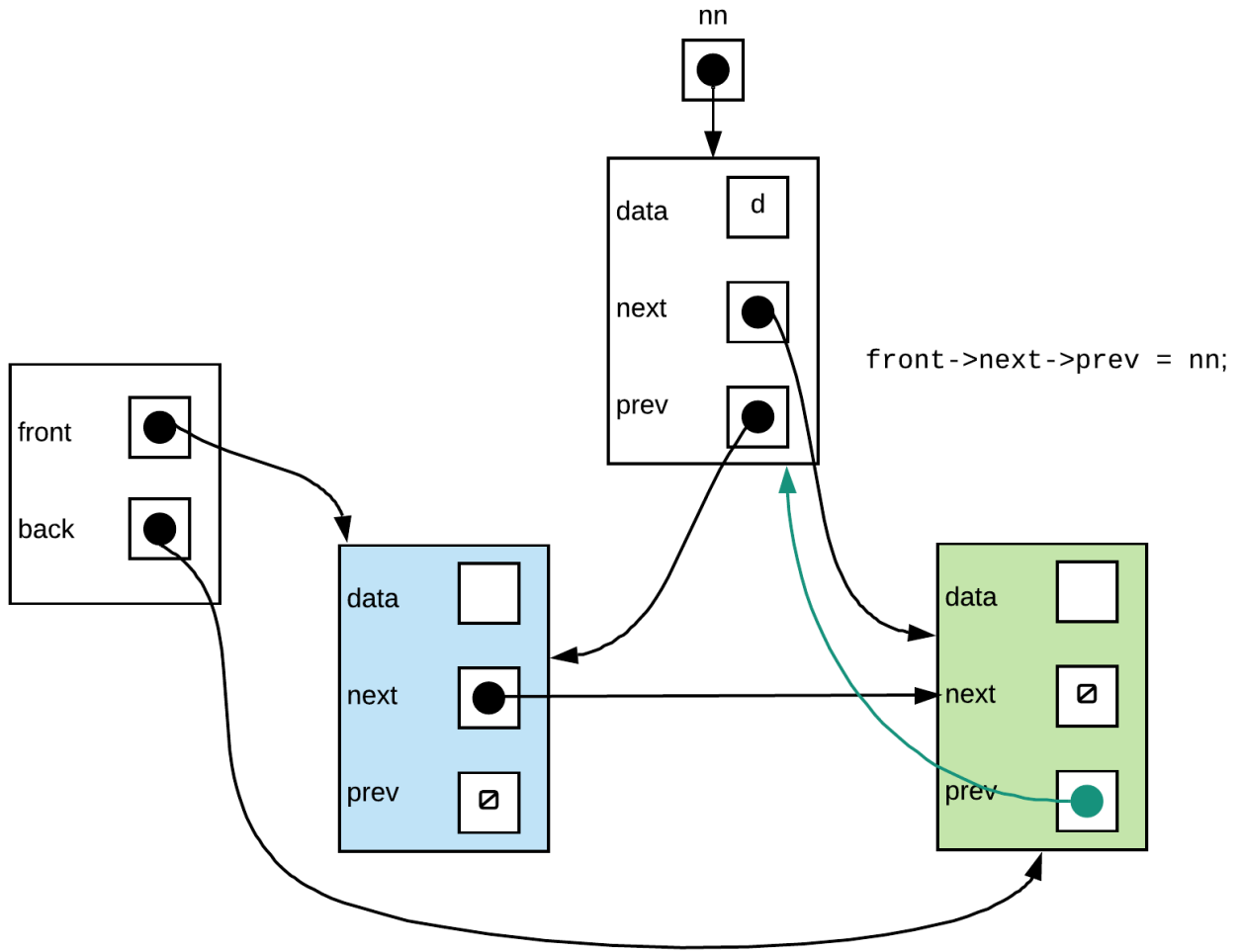
Does the above always work?

In the non-sentinel version, we know that the function fails for empty linked lists. Does the above function work if you had an linked list that has sentinel nodes?

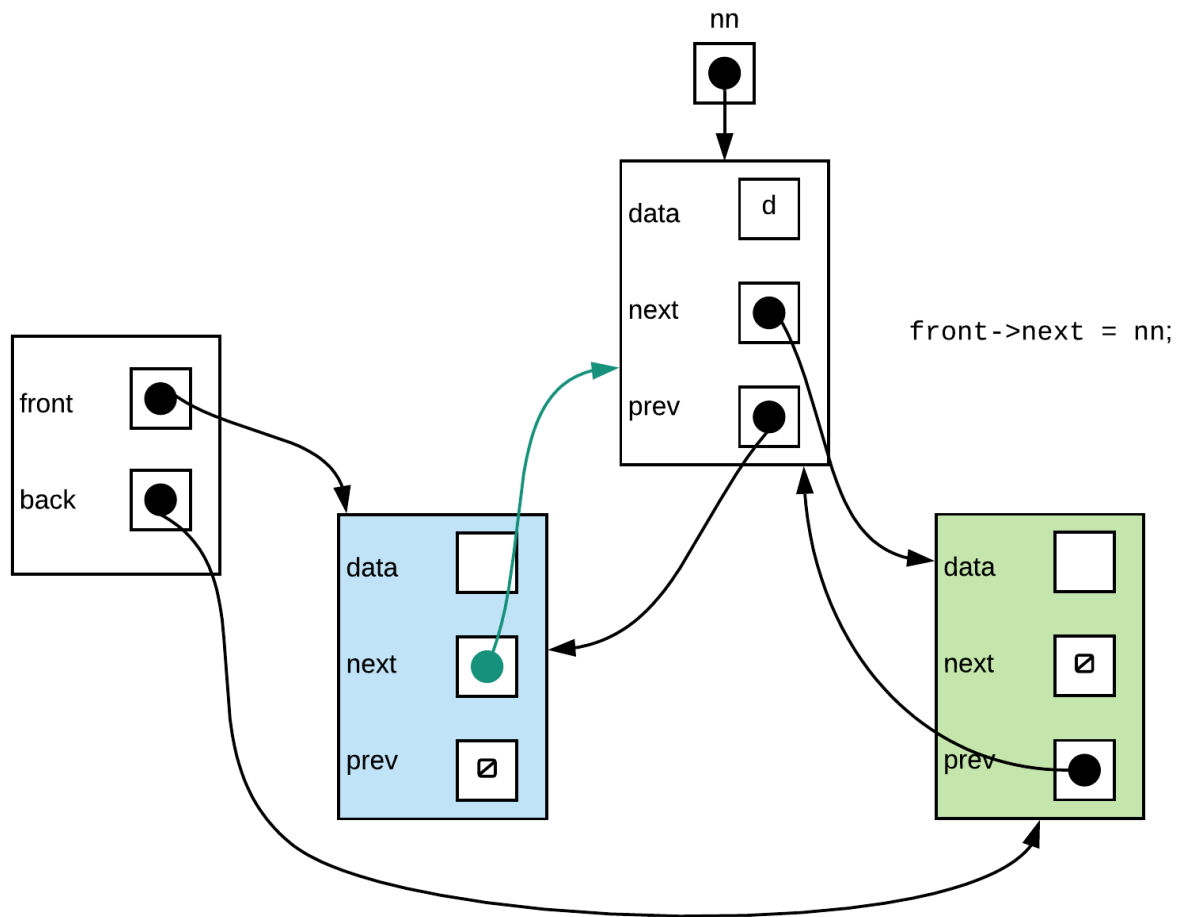We begin with an empty linked list with sentinel nodes

Step 1: looks good so far

nn

data d

next

prev

nn = LinkedList.Node(data,self.front.next,self.front)

front

back

data

next

prev

data

next

prev

Step 2: No problem here

nn

data | d

next | ●

prev | ●

self.front.next.prev = nn

front | ●

back | ●

data |

next | ●

prev | ⊘

data |

next | ⊘

prev | ●

Step 3: and done... a valid linked list

self.front.next = nn
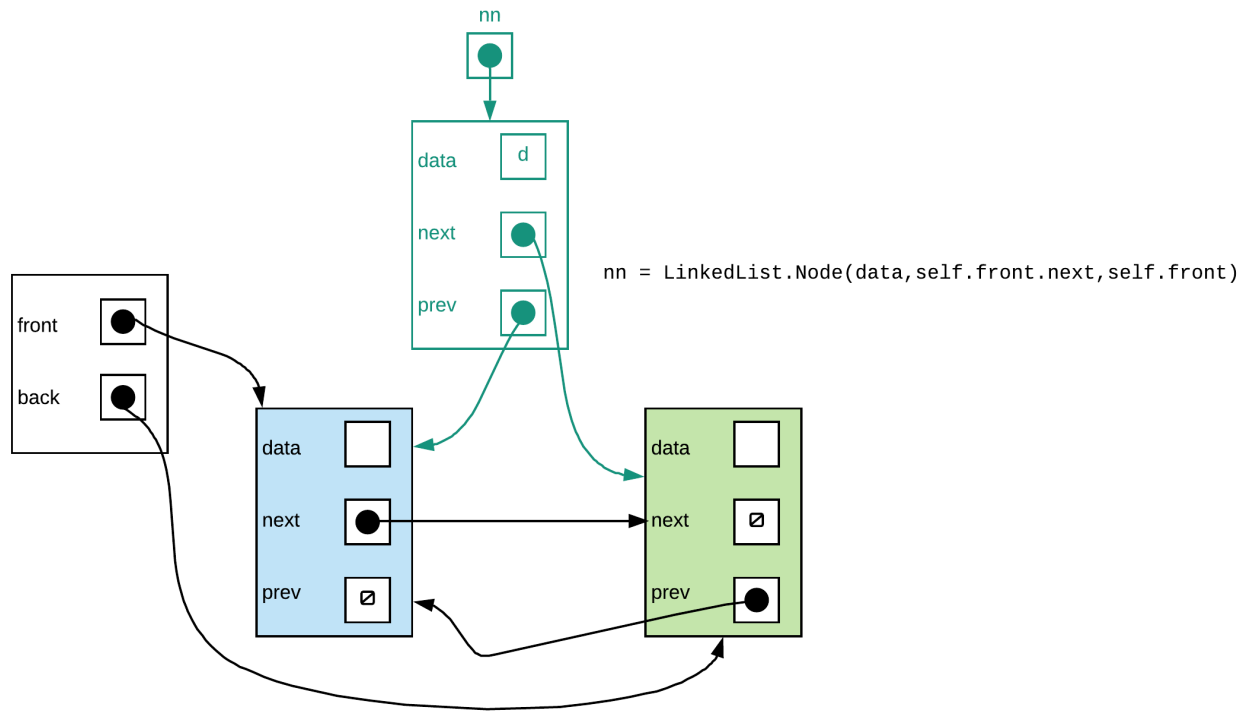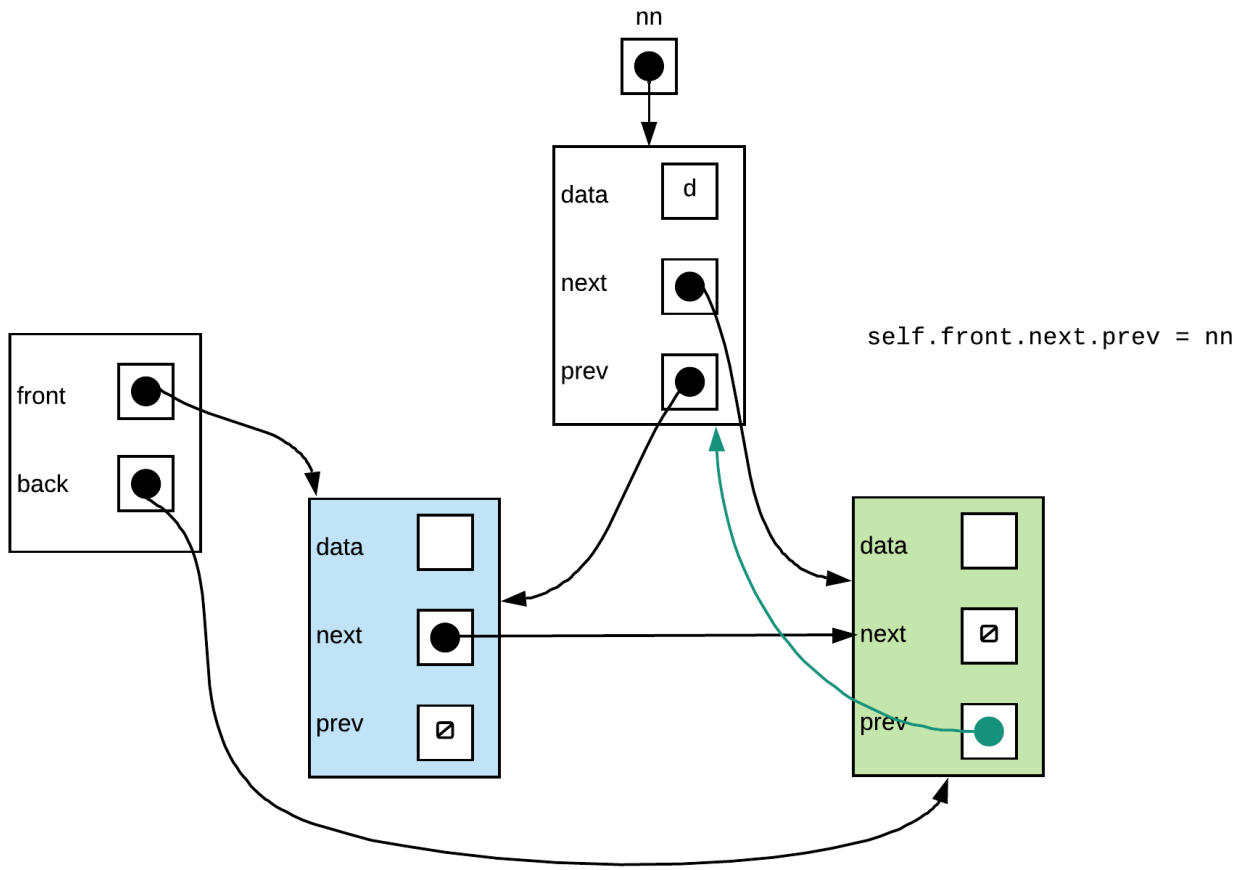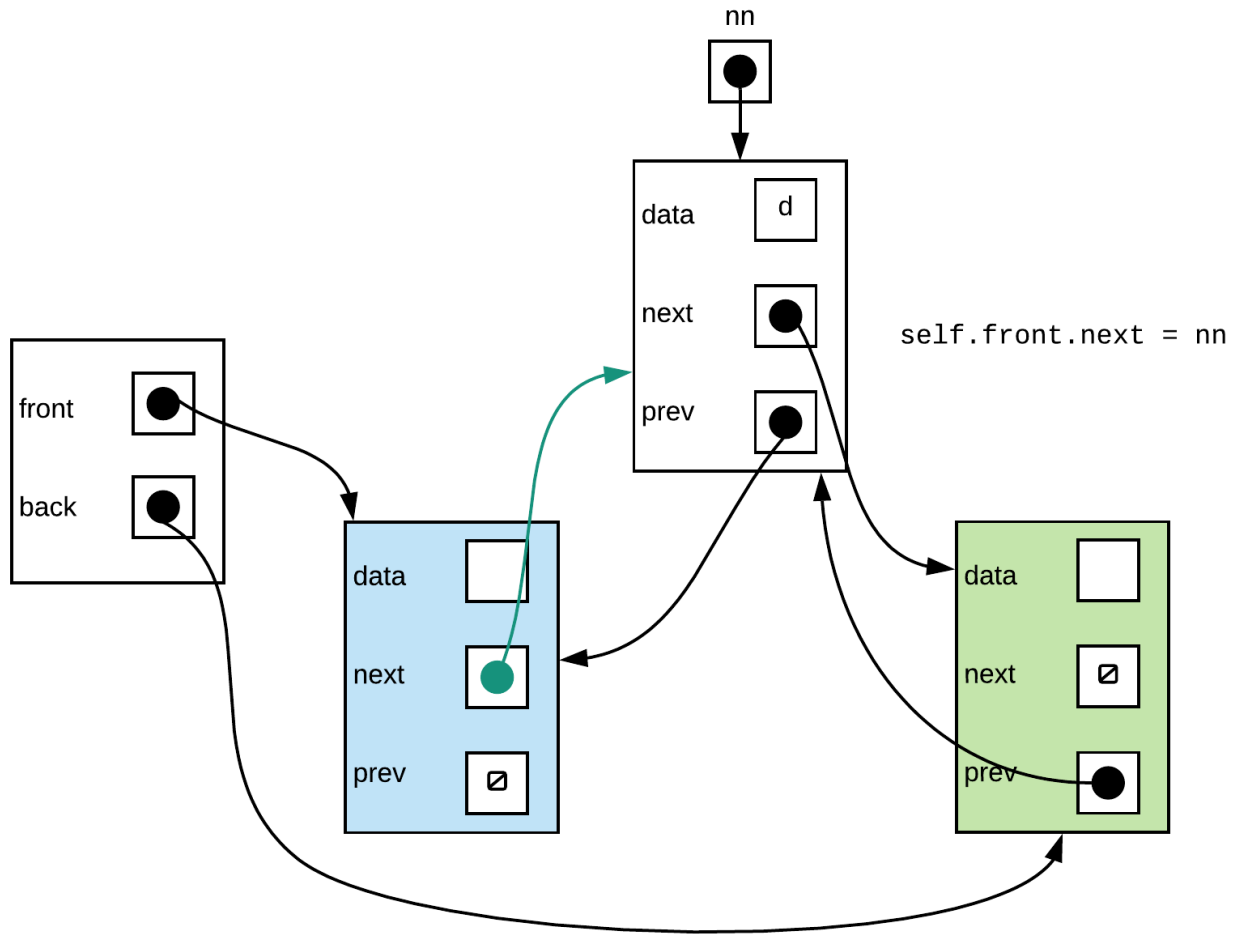
As you can see from above, the same set of steps applied to the general case as well as empty list would end up with a proper linked list. Thus we only need one version of push_front function (3 steps, no special case checks)
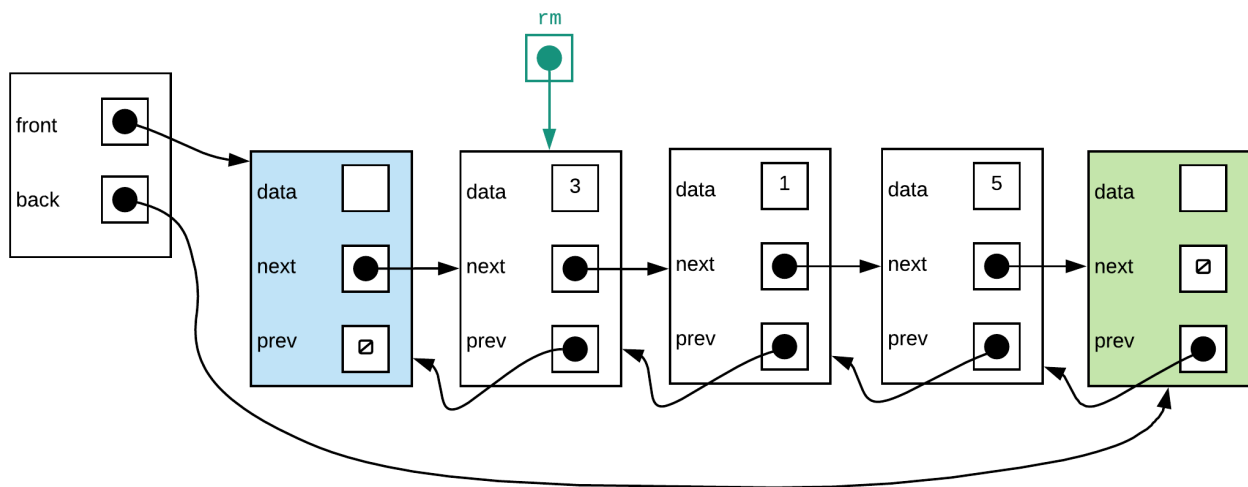
```python
def push_front(self, data):
    nn = self.Node(data, self.front.next, self.front)
    self.front.next.prev = nn
    self.front.next = nn
```
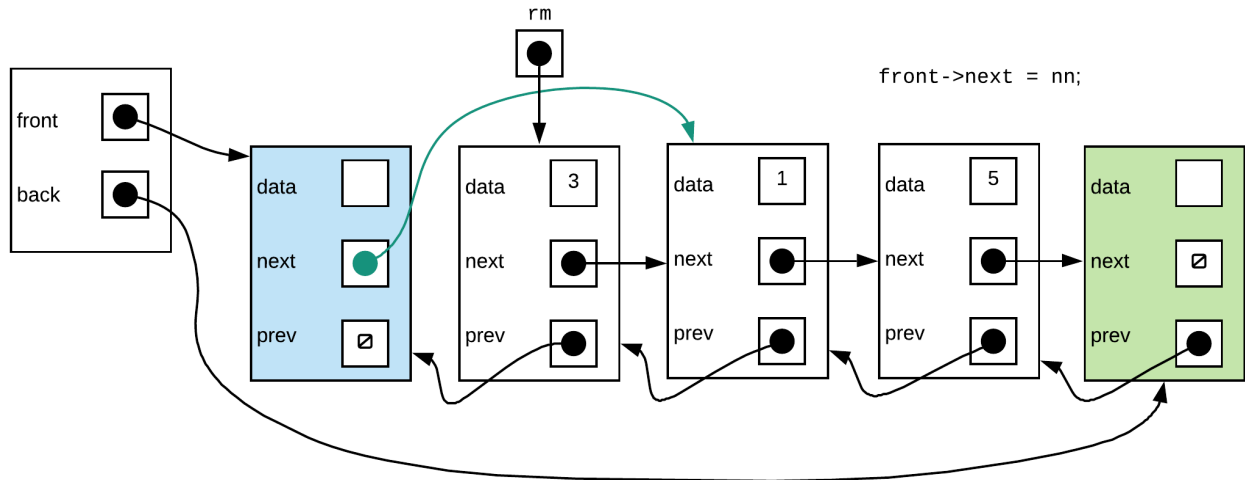
pop_front()

How will we change pop_front() now that we have sentinels?

Step 1: Check to make sure list isn't empty. If it is do nothing. Otherwise continue to next steps. Remember that with sentinels, an empty list still has two nodes (the front and back sentinels). Our empty check is therefore going to look at whether those are the only nodes that exist. We can do this by checking if front sentinel's next_ pointer points to the back sentinel
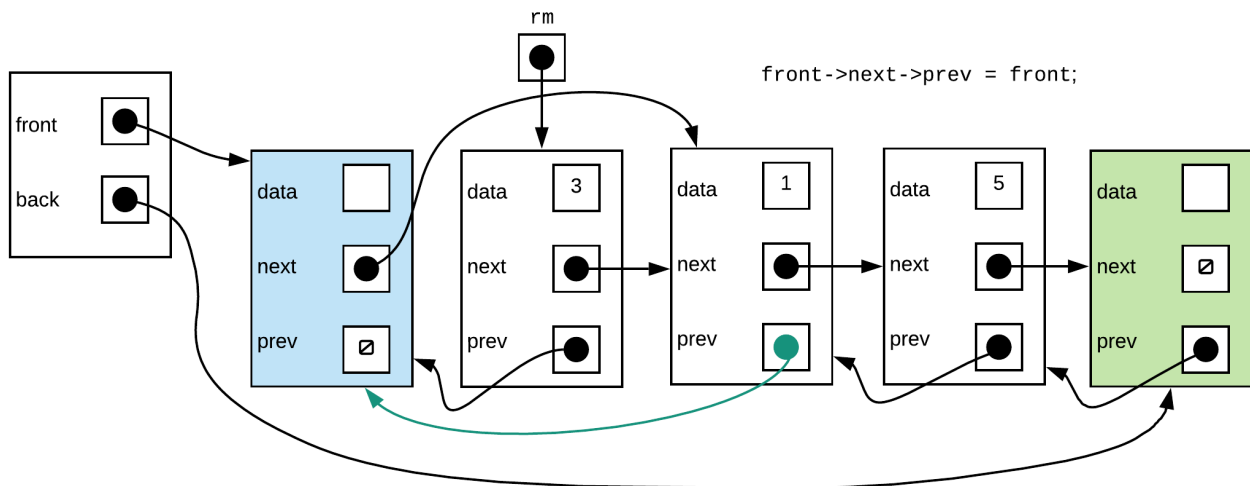
Step 2: Make a local pointer point to the Node we want to remove. This will be the node that follows the front sentinel as it is the first node with real data. (hold this node so we don't lose it by accident)
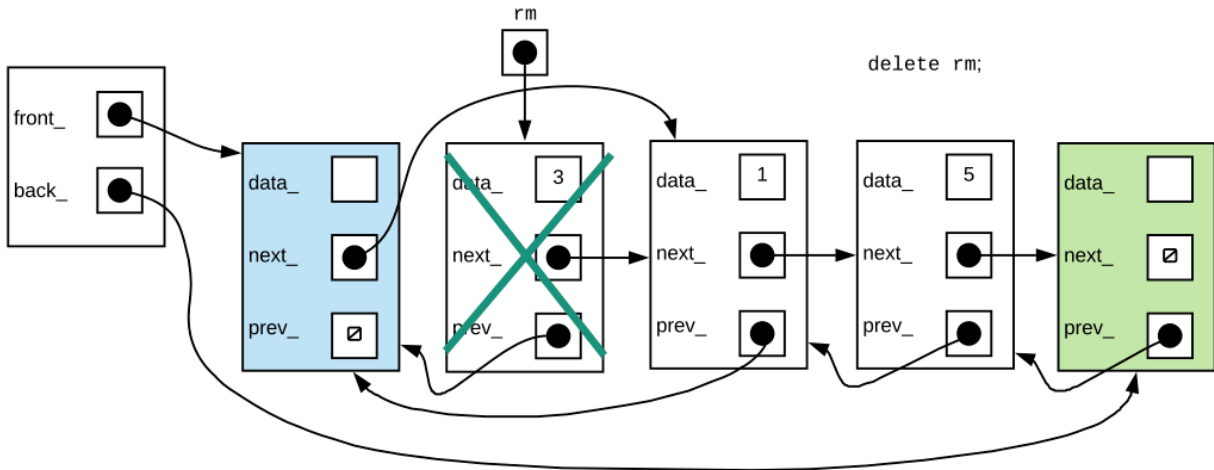

Step 3: Make the front sentinel's next pointer point to second data node (the one that follows the one we want to remove

Step 4: Make the previous pointer of the node that now follows the front sentinel point back to the front sentinel

The code snippet to do this is:

```python
if self.front.next is not self.back:
    rm = self.front.next
    rm.next.prev = rm.prev
    rm.prev.next = rm.next
    del rm
```
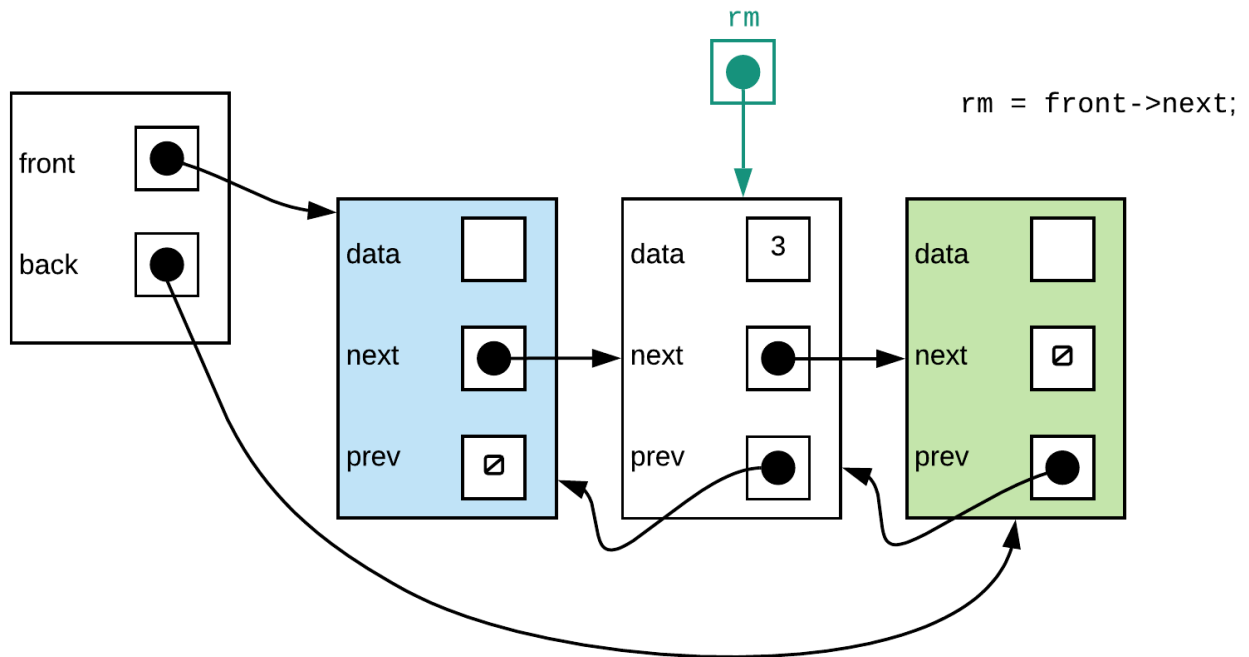
Does the above always work?

When we looked at the linked list that did not use sentinels we saw that the general solution did not work when the node being removed was the only node left. Question is, will it work now if the node we want to remove is the only one left.
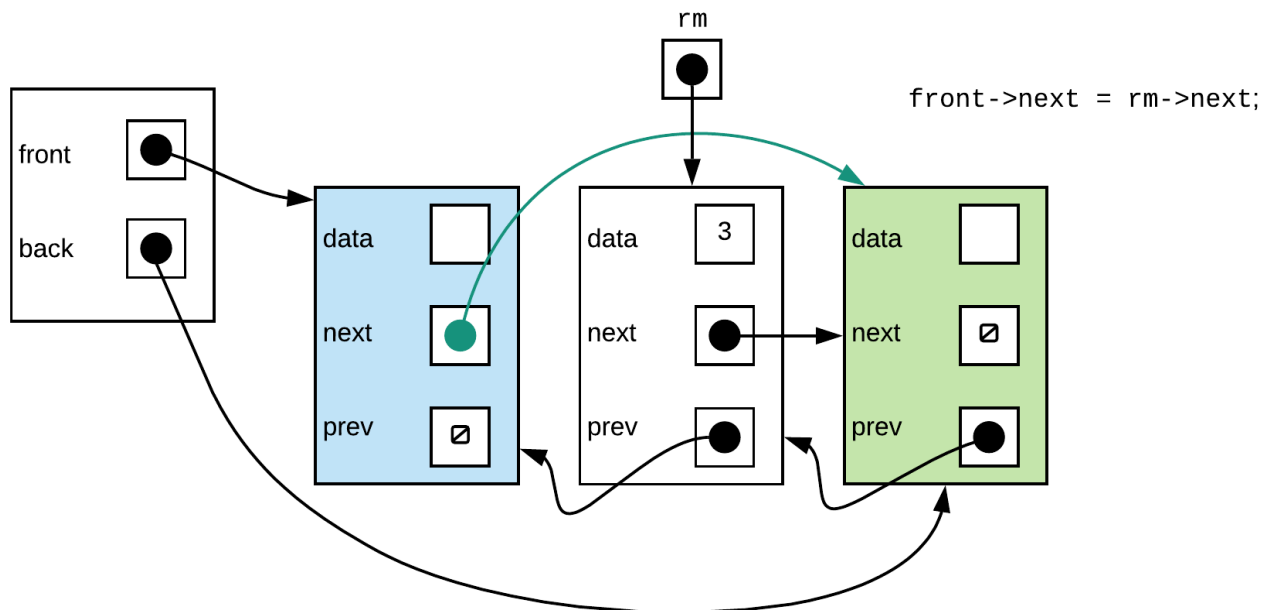
If we perform the four steps inside the if statement from the above snippet, this is what we will see:

Step 2:
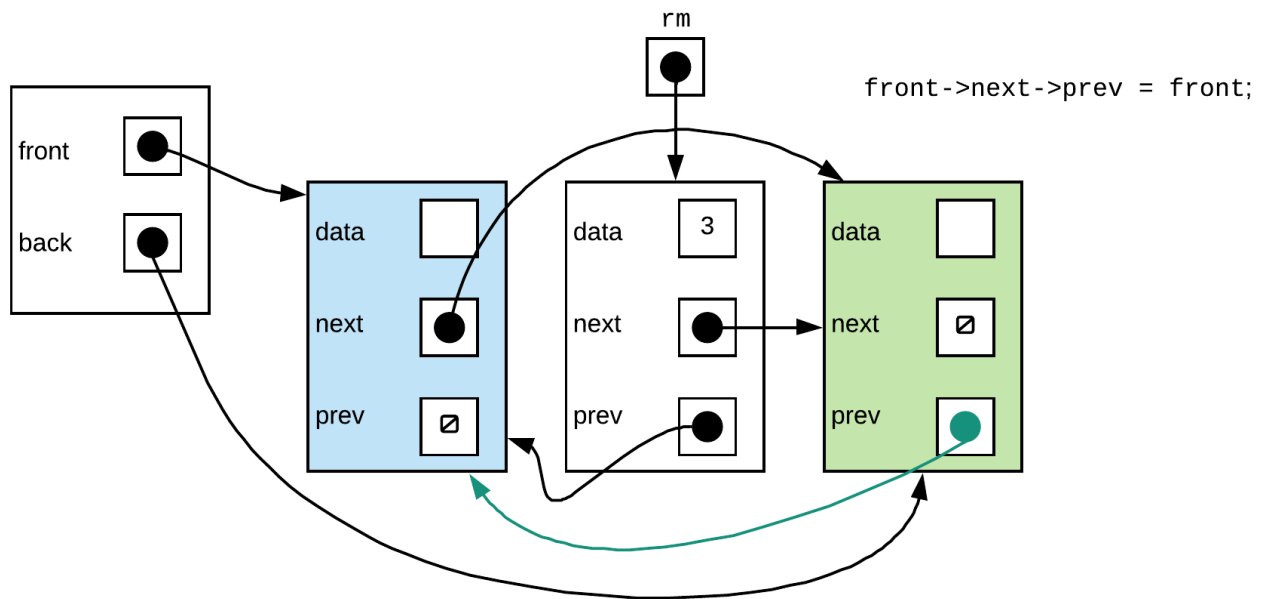


```
rm = self.front.next
```
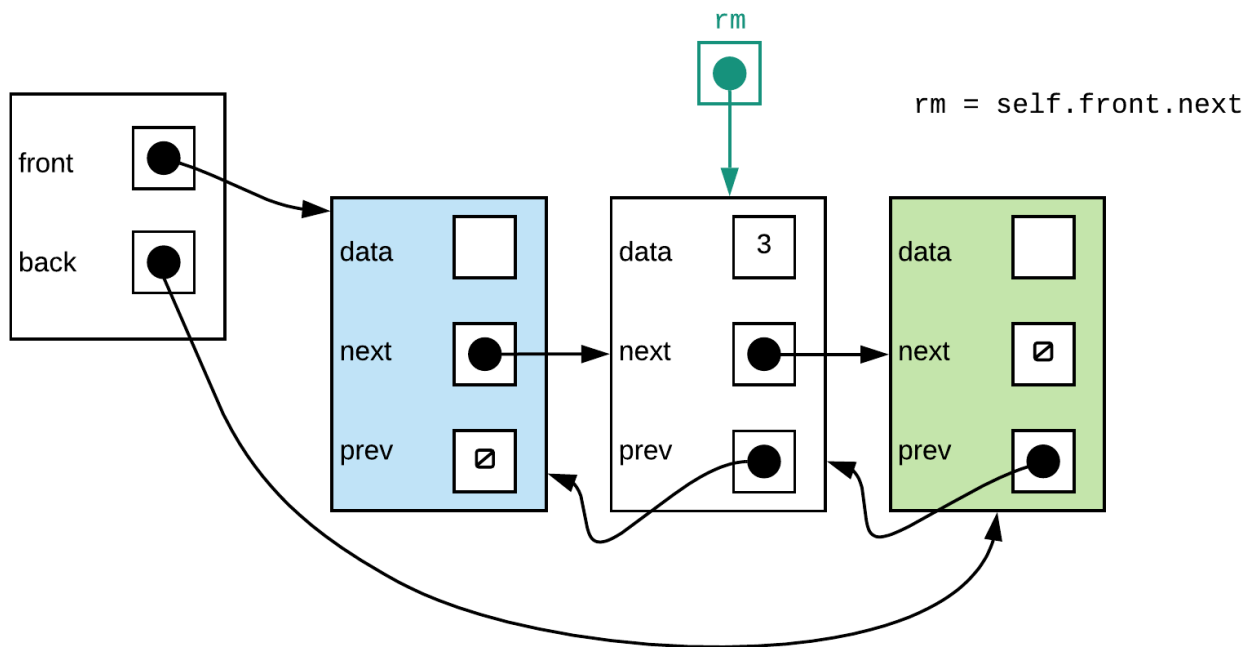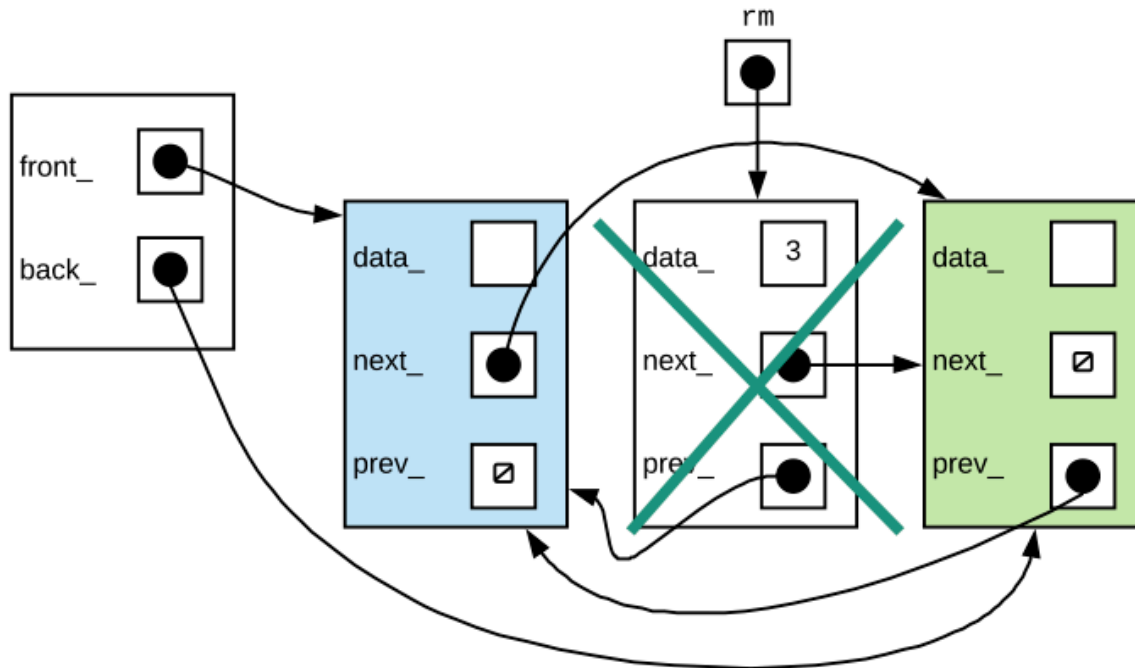
Above looks fine.

Step 3:

This also looks correct
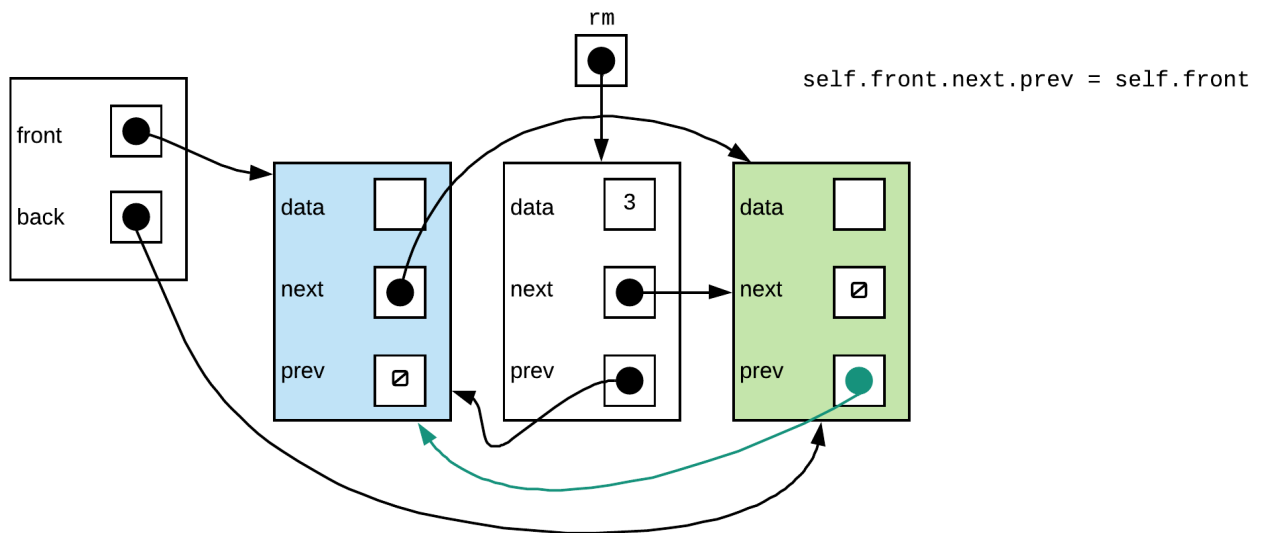
Step 4:

This was a problem in the non-sentinel version but we can see that there isn't a problem here. No nullptr access.

Thus, our function works for both the general and special case.

Putting all this together our pop_front() function should look something like this:

```python
def pop_front(self):
    if self.front.next is not self.back:
        rm = self.front.next
        rm.next.prev = rm.prev
        rm.prev.next = rm.next
        del rm
```

```cpp
template <typename T>
class DList{
    struct Node{
        T data;
        Node* next;
        Node* prev;
        Node(const T& dat=T{}, Node* nx=nullptr, Node* pr=nullptr){
            data=dat;
            next=nx;
            prev=pr;
        }
    };
    Node* front;
    Node* back;
public:
    DList(){
        front = new Node();
        back = new Node();
        front->next=back;
        back->prev=front;
    }
};
```

push_front()

How will push_front change with sentinels?

Step 1: Create new node, next node is the node that follows the front sentinel. The previous node is the front sentinel.

```
Node* nn = new Node(data,front->next,front);
```

Step 2: Make the previous pointer of the node that follows the front sentinel point to the new node

nn

data d

next ●

prev ●

front->next->prev = nn;

front ●

back ●

data [ ]

next ●

prev ⌀

data 3

next ●

prev ●

data 1

next ●

prev ●

data 5

next ●

prev ●

data [ ]

next ⌀

prev ●

Step 3: Set the next pointer of the front sentinel to the new node.

nn

data   d

next

prev

front->next = nn;

front

back

data

next

prev

data   3

next

prev

data   1

next

prev

data   5

next

prev

data

next

prev

Does the above always work?

In the non-sentinel version, we know that the function fails for empty linked lists. Does the above function work if you had an linked list that has sentinel nodes?

We begin with an empty linked list with sentinel nodes

Step 1: looks good so far

nn

data | d

next

prev

Node* nn = new Node(data,front->next,front);

front

back

data

next

prev

data

next

prev

Step 2: No problem here

nn

data   d

next   ●

prev   ●

`front->next->prev = nn;`

front   ●

back   ●

data   ☐

next   ●

prev   ⊘

data   ☐

next   ⊘

prev   ●

Step 3: and done... a valid linked list

nn

data d

next

prev

front->next = nn;

front

back

data

next

prev

data

next

prev

As you can see from above, the same set of steps applied to the general case as well as empty list would end up with a proper linked list. Thus we only need one version of push_front function (3 steps, no special case checks)

```cpp
void push_front(const T& data){
    Node* nn=new Node(data,front->next,front);
    front->next->prev = nn;
    front->next = nn;
}
```

pop_front()

How will we change pop_front() now that we have sentinels?

Step 1: Check to make sure list isn't empty. If it is do nothing. Otherwise continue to next steps. Remember that with sentinels, an empty list still has two nodes (the front and back sentinels). Our empty check is therefore going to look at whether those are the only nodes that exist. We can do this by checking if front sentinel's next_ pointer points to the back sentinel

Step 2: Make a local pointer point to the Node we want to remove. This will be the node that follows the front sentinel as it is the first node with real data. (hold this node so we don't lose it by accident)

Step 3: Make the front sentinel's next pointer point to second data node (the one that follows the one we want to remove

Step 4: Make the previous pointer of the node that now follows the front sentinel point back to the front sentinel

`front->next = nn;`

Step 5: deallocate the node we are removing

`front->next->prev = front;`

The code snippet to do this is:

```cpp
if(front->next != back_){  //check for empty list
    Node* rm = front->next;
    front->next=rm->next;
    front->next->prev=front;
    delete rm;
}
```

Does the above always work?

When we looked at the linked list that did not use sentinels we saw that the general solution did not work when the node being removed was the only node left. Question is, will it work now if the node we want to remove is the only one left.

If we perform the four steps inside the if statement from the above snippet, this is what we will see:

Step 2:

rm

rm = front->next;

front

back

data

next

prev

data 3

next

prev

data

next

prev

Above looks fine.

Step 3:

rm

front->next = rm->next;

front

back

data

next

prev

data 3

next

prev

data

next

prev

This also looks correct

Step 4:

This was a problem in the non-sentinel version but we can see that there isn't a problem here. No nullptr access.

Step 5:

The final line is to delete the node. Notice now, that our list is essentially an empty linked list (with just the two sentinels). Thus, our function works for both the general and special case.

Putting all this together our pop_front() function should look something like this:

```
void pop_front(){
    if(front->next != back){  //check for empty list
        Node* rm = front->next;
        front->next = rm->next;
        front->next->prev = front;
        delete rm;
    }
}
```

Stacks and Queues

A **stack** is a kind of list where items are always added to the front and removed from the front. Thus, a stack is a **First In, Last Out** (FILO) structure. A stack can be thought of a structure that resembles a stack of trays. Each time a new piece of data is added to the stack, it is placed on the top. Each time a piece of data is removed it also must be removed from the top. Typically only the top item is visible. You cannot remove something from the middle.

Queues like stacks are a special kind of list. In the case of a queue, items are added to the back and removed from the front. In other words a queue is a First In First Out (FIFO) structure. A queue is essentially a line up.

> ⚠ **CAUTION**
>
> The idea of **front, back**, and **top** in stacks and queues have absolutely nothing to do with their positions within the data structures used to implement them. It is up to the implementer to make that decision on where to put the frontmost, topmost and backmost items and maintain order.

Operations

Typically stacks and queues have the ability to do the following:

- add an item
- remove an item
- access the "next" item to be removed

There are various names that we attribute to these operations for these:

Operation	Stack	Queue
add an item	push	enqueue
remove an item	pop	dequeue
access the "next" item to be removed	top	front

The most important thing to remember about stacks and queues is that its all about the ordering. This absolutely must be tracked and maintained. When you add an item it must be positioned as the newest item. When you remove an item the one you remove is always the newest item for stacks or the oldest item for queues. You can't put something into the middle of a stack or queue. When you remove there is only one item that you can remove. Access is typically only granted to the item that is at the top/front of the stack/queue.

> ⊘ **INFO**
>
> Stacks and Queues are NOT for general storage. They are used to track ordering. Any other features other than the 3 above must be secondary.

Applications

Applications of stacks and queues typically involve tracking the ordering of a set of values. The question is whether the application is interested in the data in the same order as it was received or reverse. Note that this is not as straight forward as it may seem. We are not talking about encountering all the data in one go. It will typically involve continuous adding and removing of data to the Stack or Queue.

Some examples:

- bracket checking (stack)
- breadthfirst tree traversals (queue)
- infix to postfix expression (stack)
- postfix expression calculation (stack)

Stack Implementation

There are two general ways to implement a stack. As a stack is essentially a list with a restriction on the operations of a list, we can use either an array or a linked list to implement a stack. The key to understanding how to do this efficiently is to understand the nature of a stack.

A stack is a FILO (first in last out) structure. Thus, the most important thing to remember about it is that when you remove an item from the stack, you want to remove the newest item, where ever that may be. How it is stored internally (which end of the list do you insert into for example) does not matter as long as you always remove the newest item. Thus, the question of how to implement a stack comes down to choosing an algorithm such that the operations can be completed as quickly as possible.

Recall that the operations are as follows:

- **push** - add a new item to the stack
- **pop** - removes top item from the stack
- **initialize** - create an empty stack
- **isEmpty** - tests for whether or not stack is empty
- **isFull** - tests to see if stack is full and cannot grow (not always needed)
- **top** - looks at value of the top item but do not remove it

Array Implementation

With a list that implemented as an array we typically start by creating an array that is bigger than what we need. To add a value to the end of an array is a constant time operation as long as we track where the end is. If we were to do that, the most recently added item will be at the back of the array. Removing that item simply involves decreasing the end of array tracker by one.

Check out this animation for details:

http://cathyatseneca.github.io/DSAnim/web/arraystack.html

Linked List Implementation

To implement a stack using a linked list, we have to consider the type of linked list we would use and which end of the list we would want to insert to.

The simplest linked list is a singly list. If this linked list was implemented with just a pointer to the first node, insertion would be O(1) at front of list, O(n) at back of list. removal is O(1) at front of list and O(n) at back of list. If we added an end pointer to the list, then insertion to back of list can be O(1) also, however removing from the back of the list will still be O(n).

If we were to insert always at front of list, then the most recently added item would be at the front of the list. Thus, removal must occur from the front as we always remove the most recently added item. Since we can do both quickly with a simple singularly linked list, that is all we will need to do.

Check out this animation for details:

http://cathyatseneca.github.io/DSAnim/web/llstack.html

Queue Implementation

This section will look at how to efficiently implement a queue using both an array and a linked list. Like a stack, a queue is also a special type of list. While a queue is a "line up" the ideas of front and back are not meant to be taken literally. The key is to understand that a Queue is a FIFO (first in first out) structure. That is the item to be removed is the oldest item in the list. as long as this is true, it doesn't matter where exactly the items get put into the queue or where it is removed.

Linked List implementation

To implement a queue using a linked list, we have to consider the type of linked list we would use and which end of the list we would want to insert to.

The simplest linked list is a singly list. If this linked list was implemented with just a pointer to the first node, insertion would be O(1) at front of list, O(n) at back of list. removal is O(1) at front of list and O(n) at back of list. If we added an end pointer to the list, then insertion to back of list can be O(1) also, however removing from the back of the list will still be O(n).

Now, if we perform enqueue by inserting at the front of the list, the oldest item would be at the back of the list and thus, dequeue would have to be done there. enqueue would be quick O(1) but dequeue would be slow O(n). However, if we enqueue by inserting to the end of the list using a linked list that tracks the last node, then the oldest item would be at the front of the linked list. This will allow us to perform enqueue and dequeue in O(1) time.

Checkout this animation for details:

http://cathyatseneca.github.io/DSAnim/web/llqueue.html

Array Implementation

Implementing a queue with an array is a bit more complicated than implementing a stack using an array.

Consider the typical way we implement a list using an array. To insert a value to the front of the list, we would move all existing values to an element that is one index higher in the array then add the new value to the first element. To remove a value we would move all values in the array to element one index lower, overwriting the first element. Both the removal and insertion are O(n) operations.

Thus, if we were to use this list, and performed enqueues at the front of the list, the enqueue function would be O(n). Enqueuing in this manner would mean that the oldest item would be at the back of the array. Removing from back of list is fast so we can accomplish this in O(1) time as long as we track where the last item is.

Now... if we were to enqueue to the end of the list, the oldest item would be at the front, and thus removal would have to be done to the front of the list. In this case, enqueue would be fast O(1) but dequeue would be slow O(n).

So clearly we need to come up with a better way to handle this.

One way that we can handle this is to track two indices instead of one. The first is the index of the element at the front of the list. The second is the index of the element at the back of the list. When you insert, insert to the index of the back element and increment the index for back. When you remove, remove by incrementing the index of the front. The second part of the implementation is that we need to treat the array as if it is a ring. That is, the next element of the last element is the first element and the previous element of the first element is the last element. If we do not, we will quickly create a list with lots of unused

space at the front of the list and run out of space.

Check out this animation for details:

http://cathyatseneca.github.io/DSAnim/web/arrayqueue.html

Implementation - Iterators

Iterators are objects that are used to traverse and access data within a container class without exposing the internal structure of the container.

Consider the following typical usage for an iterator for a container (like list) in the C++ Standard library:

```cpp
std::list<int> ll;
...
std::list<int>::const_iterator it;
for(it = ll.begin();it !=ll.end();it++){
    std::cout << *it << std::endl;
}
```

The for loop in the above code sample prints out every value within ll. There is no need for the user to actually know what ll is. We could have changed list to vector (or some other container class) and the for loop for printing every value would not look different. We don't need to know that list is basically a linked list or that vector is essentially an array... the iterators define a way for us to access every piece of data within the container class.

In this section we will look at how to implement an iterator for our linked lists.

const_iterator vs iterator

Firstly, we need to be aware that we have two different iterators that we will need to implement. a const_iterator and an iterator. We need to understand

what the difference is. A const_iterator does not allow the modification of the data within the container while an iterator does. When you pass containers to functions, you typically either pass its address or you pass by reference. To make it safe, you may make the pointer or reference const. You would not be able to use a iterator to go through this sort of container but you can use a const_iterator. Furthermore you must support the assignment of iterators into const_iterators but not the other way around.

Iterator Functionality

The iterators we will write will support the following functionality:

- operator ++(makes iterator point to next piece of data. two versions, post and pre-fix)
- operator -- (makes iterator point at previous piece of data. two versions, post and pre-fix)
- operator * (dereference operator)
- operator == (compares two iterators, returns true if they refer to the same piece of data)
- operator != (compares two iterators, returns true if they do not refer to the same piece of data

We also need to add functions to the linked list class so that it will return iterators to the first piece of data within the list as well as one past the last item the list (begin() and end())

To allow assignment of iterators to const_iterators (but not the other way around) we will create a heirarchy where const_iterator is the base class and iterator is the derived class. Furthermore, we will also declare these iterators into the linked list class itself (publically).

Default Constructors

Both the iterator and const_iterator class has a default public constructor that sets the iterator to safe state.

```cpp
template <typename T>
class DList{
...
public:
    class const_iterator{

    public:
        const_iterator(){...}
        const_iterator operator++(){...}      //prefix
        const_iterator operator++(int){...}   //postfix
        const_iterator operator--(){...}      //prefix
        const_iterator operator--(int){...}   //postfix
        const T& operator*() const{...}
        bool operator==(const_iterator rhs) const{...}
        bool operator!=(const_iterator rhs) const{...}
    };
    class iterator:public const_iterator{
    public:
        iterator(){...}
        iterator operator++(){...}      //prefix
        iterator operator++(int){...}   //postfix
        iterator operator--(){...}      //prefix
        iterator operator--(int){...}   //postfix
        T& operator*(){...}
        const T& operator() const{...}
    };
    const_iterator cbegin() const{...}
    const_iterator cend() const{...}
    iterator begin(){...}
```

Friends

Even though the iterator classes are part of DList, DList will not have access to their private members. To allow access, we must declare that DList is a friend of each of the iterators.

begin()/cbegin() and end()/cend()

begin()/cbegin() returns an iterator/const_iterator that refers to the first piece of data in the container. In this case this is the node pointed to by front_. **end()/cend()** returns an iterator/const_iterator to the item that follows the last piece of data in the container. In this case, it would be a nullptr.

{% hint style="warning" %} **end()** does not return iterator to last item. It returns an iterator to whatever is **AFTER** the last item {% endhint %}

Dereference (*)

The dereference operator is how we access the data within each node. There are two versions. One returns a reference to the data the other a const reference to the data. Only the iterator (not const_iterator) can return the non-const reference to the data.

Increment and Decrement (++ and --)

Both the ++ and -- operators have a prefix and postfix version. The ++

operator makes the operand point to the "next" item (one closer to the end) in the list. The -- operator makes the operand point to the "previous" item (one closer to the beginning of the list. Both the prefix and postfix versions of the operators do the same thing to the operand. The difference is in what is returned. In the prefix version of the operator, the operator returns an iterator that refers to the same object as the operand. In the postfix versions of the operator, the operator returns an iterator to the object the operand pointed at before it was altered.

{% hint style="info" %} prefix and postfix refers to the position of the operator with respect to the operand

pre_fix - operator comes before (pre) the operand (++x or **--x)*
post_fix-operator comes after (post) the operand (x++ or x--) {% endhint %}

What data must we have?

It is probably pretty obvious that we will need to store a Node* in the iterator object. It can be used to refer to a node and access the data within the node. However, that will not be quite enough. Consider the following:

According to specs, if you did the following ++x followed by --x, x should end up back where it started. Suppose initially x was referring to the last node in the list. ++x would make x refer to end(). if you then did --x you should be able to get back to the last node. To support this in your implementation you must store enough information to get back to the last node after you hit end(). Now, if all you store is a Node*, then end() would simply set that pointer to nullptr. There is no way for you to get back to any node within the list as there isn't any reference to the list or any nodes within it.

Thus, in order to allow the above to occur we will not only need a Node* but we will also need to store a pointer the DList object itself. This way, we can use the

DList pointer to find its last node.

Private Constructors

It would be advantageous to create private constructors to iterator. These would only be accessible from members of DList and simplify construction.

Putting all this together

```cpp
template <typename T>
class DList{
...
public:
    class const_iterator{
        friend class DList;
        const DList* myList_;
        Node* curr_;
        const_iterator(Node* curr, const DList* theList){
            curr_ = curr;
            myList_=theList;
        }
    public:
        const_iterator(){
            myList_=nullptr;
            curr_=nullptr;
        }
        //prefix
        const_iterator operator++(){
            curr_=curr_->next_;
            return *this;
        }
        //postfix
        const_iterator operator++(int){
            const_iterator old=*this;
```

Tables

A Table is an ***unordered collection of records***. Each record consists of a key-value pair. Within the same table, keys are unique. That is only one record in the table may have a certain key. Values do not have to be unique.

Table operations

A table supports a subset of the following operators (though sometimes it may be combined in design)

- initialize - Table is initialized to an empty table
- isEmpty - tests if table is empty
- isFull - tests if table is full
- insert - add a new item with key:value to the table
- delete - given a key remove the record with a matching key
- update - given a key:value pair change the record in table with matching key to the value
- find - given a key find the record
- enumerate - process/list/count all items in the table

A Simple Implementation

Like lists, a ***table*** is an abstract data type. That is, it describes what it is but not how to implement one. The underlying data structure needed to create a table can vary widely from arrays sorted by keys to hash tables and even trees.

A simple implementation is to simply use an array sorted by keys.

Insertion/Update

To add to this table, we must first find the spot where the item will go. This can be done by modifying a binary search algorithm. Once the location is found, if a record with the same key is not already in the table, we will need to shift every item over to make room for the new record. If a record with a matching key already exists, the old record can be replaced with the new.

The run time for performing the search is O(log n). If the record already exists, and we are just updating it, the run time would be O(log n). However, inserting a brand new record with a different key will require shifting on average 50% of the list, and thus we are looking at a run time of O(n) for that operation.

Remove

To remove a record, we can start with a binary search algorithm to find the record according to the key. If such a record exists, removal will involve shifting the records down.

The run time for search is O(log n). However to remove, we must shift all the

records down. This process is O(n). Thus the remove operation is O(n)

Search

As the array is sorted by key, all searching can be done using a binary search. This has a run time of O(log n)

Drawbacks

This implementation clearly has a lot of drawbacks. While the search() function has an acceptable run time of $logn$, the other functions are generally slow. The only part that is fast is search. Ideally it would be faster than this.

Hash Tables

A hash table uses the key of each record to determine the location in an array structure. To do this, the key is passed into a hash function which will then return a numeric value based on the key.

Hash Functions

A hash function must be designed so that given a certain key it will always return the same numeric value. Furthermore, the hash function will ideally distribute all possible keys in the keyspace uniformly over all possible locations.

For example suppose that we wanted to create a table for storing customer information at store. For the key, a customer's telephone number is used. The table can hold up to 10,000 records and thus valid indexes for an array of that size would be [0 - 9999]

Telephone numbers are 10 digits (###) ###-####

The first 3 of which is an area code.

Now, if your hash function was: use the first 4 digits of the phone number (area code + first digit of number) that hash function would not be very good because most people in the same area would have the same area code. Most people in the Toronto for example have area code of 416 or 647... so there would be very little variation in the records. However the last 4 digits of a phone number is much more likely to be different between users (though certainly not unique).

Generally speaking a good hash function should be:

- uniform (all indices are equally likely for the given set of possible keys)
- random (not predictable)

Load Factor

The load factor denoted by the symbol λ (pronounced lambda) measures the fullness of the hash table. It is calculated by the formula:

$$\lambda = \frac{\text{number of records in table}}{\text{number of locations}}$$

Collisions

The pigeon hole principle

Suppose you had n mailboxes and m letters where m > n (more letters than mailboxes). If you were to place all the letters into the available mailboxes, there would be at least one mailbox with at least 2 letters in it. This is the pigeon hole principle.

This is effectively what the situation is with our hash function and keys. The number of mailboxes we have is n (capacity of array). The total number of possible keys is m. Typically, the total number of possible keys is bigger than the capacity of the array. (usually significantly bigger). Based on the pigeon hole principle, it means that we will have situations where at two keys will get hashed into the same hash index.

When two keys have the same hash index you have a ***collision***. Generally speaking, collisions are unavoidable. The key is to have a method of resolving them when they do happen. The rest of this chapter look at different ways to deal with collisions.

Chaining

At every location (hash index) in your hash table store a linked list of items. You only use as many nodes as necessary. Using Chaining the Table will not overflow as you can have as long a chain as you need. However, You will still need to conduct a short linear search of the linked list but if your hash function uniformly distributes the items, the list should not be very long.

For chaining, the runtimes depends on the load factor (λ) The average length of each chain is λ . λ is the number of expected probes needed for either an insertion or an unsuccessful search. For a successful search it is $1 + \frac{\lambda}{2}$ probes.

While it is possible for $\lambda > 1$, it is generally not a great idea to be too much over. Your goal isn't to search long chains as that is very slow. The ability to have long chains is more of a safety feature... you should try to still have as short a chain as possible.

Chaining is a simple way of handling collisions. Instead of storing the key-value pair (k, v) into the array (with capacity m) directly, chaining creates an array of linked lists, initially all empty. For each operation involving key k

- calculate $i = hashIndex(k, m)$
- perform operation (insert/delete/search) on the linked list at array[i]

Example

Suppose we were to have 6 keys $(k1, k2, k3, k4, k5, k6)$. The hash function returns as follows for these keys:

- $hashIndex(k1, m) == 0$

- $hashIndex(k2, m) == m - 3$
- $hashIndex(k3, m) == m - 1$
- $hashIndex(k4, m) == m - 3$
- $hashIndex(k5, m) == 0$
- $hashIndex(k6, m) == m - 3$

A table created using chaining would store records as follows (note that only key's are shown in diagram for brevity)
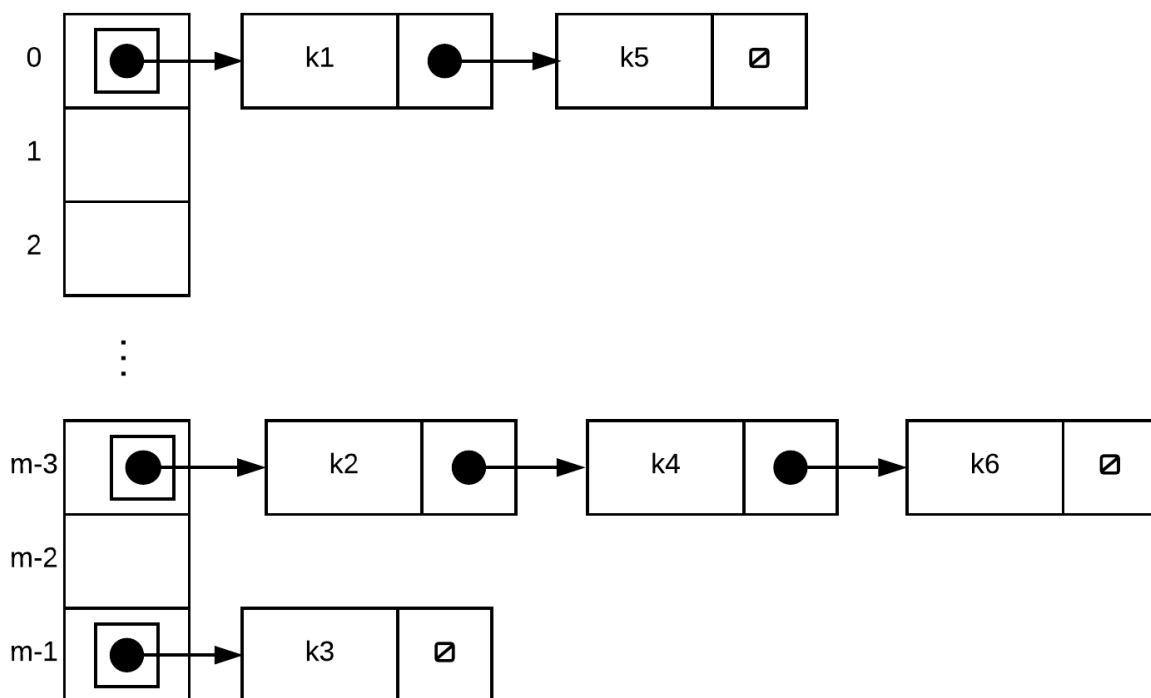
Worst case run time

insert(k,v) - cost to find the correct linked list + cost to search for k within the linked list, + cost to add new node or modify existing if k is found

search(k) - cost to find the correct linked list + cost to search for k within the

linked list

delete(k) - cost to find the correct linked list + cost to search for k within the linked list + cost to remove a node from list

In each of the above cases, cost of to find the correct linked list is $\theta(1)$ assuming that the cost of calculating hash is constant relative to number keys. We simply need to calculate the hash index, then go to that location

The cost to add a node, modify a node or delete a node (once node has been found) is $\theta(1)$ as that is the cost to remove/insert into linked list given pointers to appropriate nodes

The cost to search through linked list depends on number of nodes in the linked list. At worst, every single key hashes into exactly the same index. If that is the case, the run time would be $\theta(n)$

Thus, the worst case run time is $\theta(n)$. In reality of course, the performance is signficantly better than this and you typically don't encounter this worst case behaviour. Notice that the part that is slow is the search along the linked list. If our linked list is relatively short then the cost to search it would also not take very long.

Average case run time

We begin by making an assumption called Simple Uniform Hash Assumption (SUHA). This is the assumption that any key is equally likely to hash to any slot. The question then becomes how long are our linked lists? This largely depends on the load factor $\lambda = n/m$ where n is the number of items stored in the linked list and m is the number of slots. The average run time is $\theta(1 + \lambda)$

Linear Probing

Chaining essentially makes use of a second dimension to handle collisions. Chaining is an example of a **closed addressing**. With closed addressing collision resolution methods use the hash function to specify the exact index of where the item is found. We may have multiple items at the index but you are looking at just that one index.

This is not the case for linear probing. Linear Probing only allows one item at each element. There is no second dimension to look. Linear probing is an example of open addressing. Open addressing collision resolution methods allow an item to be placed at a different spot other than what the hash function dictates. Aside from linear probing, other open addressing methods include quadratic probing and double hashing.

With hash tables where collision resolution is handled via open addressing, each record actually has a set of hash indexes where they can go. If the first location at the first hash index is occupied, it goes to the second, if that is occupied it goes to the third etc. The way this set of hash indexes is calculated depends on the probing method used (and in implementation we may not actually generate the full set but simply apply the probing algorithm to determine where the "next" spot should be).

Linear probing is the simplest method of defining "next" index for open address hash tables. Suppose hash(k) = i, then the next index is simply i+1, i+2, i+3, etc. You should also treat the entire table as if its round (front of array follows the back). Suppose that m represents the number of slots in the table, We can thus describe our probing sequence as: $\{hash(k), (hash(k) + 1)\%m, (hash(k) + 2)\%m, (hash(k) + 3)\%m, \ldots\}$

Method 1:

Insertion

The insertion algorithm is as follows:

- use hash function to find index for a record
- If that spot is already in use, we use next available spot in a "higher" index.
- Treat the hash table as if it is round, if you hit the end of the hash table, go back to the front

Each contiguous group of records (groups of record in adjacent indices without any empty spots) in the table is called a cluster.

Searching

The search algorithm is as follows:

- use hash function to find index of where an item should be.
- If it isn't there search records that records after that hash location (remember to treat table as cicular) until either it found, or until an empty record is found. If there is an empty spot in the table before record is found, it means that the the record is not there.
- NOTE: it is important not to search the whole array till you get back to the starting index. As soon as you see an empty spot, your search needs to stop. If you don't, your search will be incredibly slow

Removal

The removal algorithm is a bit trickier because after an object is removed, records in same cluster with a higher index than the removed object has to be adjusted. Otherwise the empty spot left by the removal will cause valid searches to fail.

The algorithm is as follows:

- find record and remove it making the spot empty
- For all records that follow it in the cluster, do the following:
    - determine the hash index of the record
    - determine if empty spot is between current location of record and the hash index.
    - move record to empty spot if it is, the record's location is now the empty spot.

Example:

Suppose we the following 7 keys and their associated hash indices. Let us then insert these 5 keys from k1 to k5 in that order.

Key	Hash Index
k1	8
k2	7
k3	9

Key	Hash Index
k4	1
k5	8
k6	9
k7	8

Insert keys k1 to k4

All four keys have the different hash indexes and thus, no collisions occur, they are simply placed in their hash position.

0	1	2	3	4	5	6	7	8	9
	k4						k2	k1	k3

Insert k5. probe sequence of k5 is $\{(8+0)\%10, (8+1)\%10, (8+2)\%10, (8+3)\%10, (8+4)\%10, (8+5)\%10, \dots\} = \{8, 9, 0, 1, 2, 3 \dots\}$. Thus, we place k5 into index 0 because 8, 9 and 0 are all occupied

0	1	2	3	4	5	6	7	8	9
k5	k4						k2	k1	k3

Suppose we then decided to do a **search for k6**. k6 does not exist, so the question is when can we stop. k6's probe sequence is: $\{(9+0)\%10, (9+1)\%10, (9+2)\%10, (9+3)\%10, (9+4)\%10, (9+5)\%10, \dots\} = \{9, 0, 1, 2, 3, 4 \dots\}$. We begin looking at the first probe index 9. We proceed until we get to index 2. Since index 2 is empty, we can stop searching

0	1	2	3	4	5	6	7	8	9
k5	k4						k2	k1	k3
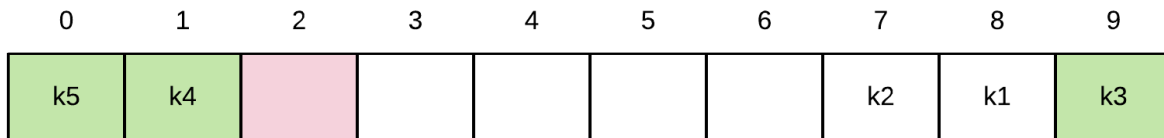
Search for k5. If we were to search for something that is there, this is what would happen. Probe sequence for k5 is $\{8, 9, 0, 1, 2, 3 \dots\}$. Thus, we would start search at 8, we would look at indices 8,9,0, and 1. At index 1 we find k5 so we stop

0	1	2	3	4	5	6	7	8	9
k5	k4						k2	k1	k3
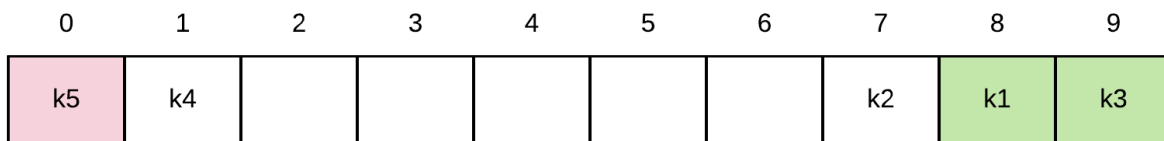
Now, lets remove a node.

A **cluster** is a group of records without any empty spots. Thus, any search begins with a hashindex within a cluster searches to the end of the cluster.

Currently there is one big cluster from index to 7 to index 1 inclusive.

Suppose we **removed k3.** If we did this, our one big cluster would be split into two smaller clusters. This is actually a good thing as search stops on first empty spot. So the only question really is whether each record in the group

that follows the removed records are in the correct cluster (the groups before the removed record is always in the correct spot.

	0	1	2	3	4	5	6	7	8	9
	k5	k4						k2	k1	
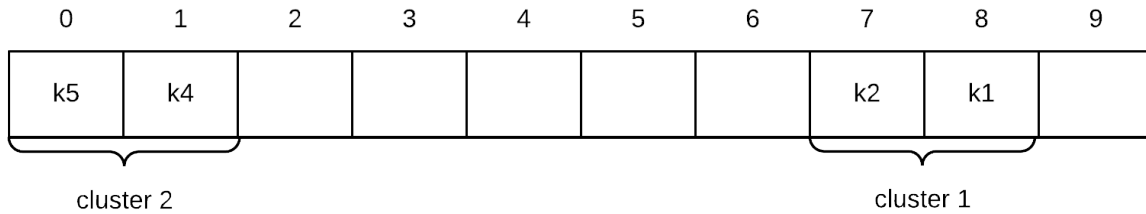
cluster 2   cluster 1

So we go through the remaining records in the cluster and use the hashindex of each key to determine if its in the correct cluster. If it is in the wrong cluster we move. it

Continuing with example, we look at k5. k5 should actually go in 8, so the record is in the wrong side of the empty spot, so what we do is move the record into the empty spot, make k5's spot the empty spot and continue.

	0	1	2	3	4	5	6	7	8	9
	k5	k4						k2	k1	
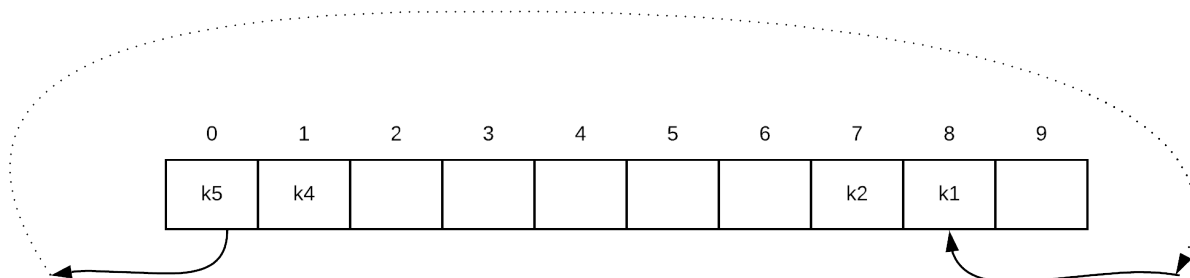
We test k4 and its in the correct side of the empty spot so we leave it alone
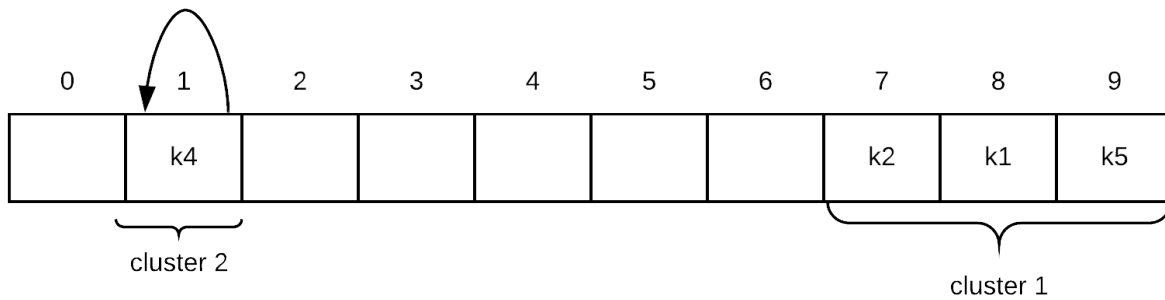
We finish processing the records within the cluster so we are done.

Method 2: Tombstoning

Tombstoning is a method that is fairly easy to implement. It requires each element to not only store the record, but also a status of element. There are three statuses:

- Empty - nothing has ever been inserted into this spot
- Used/Occuppied - a record is stored here
- Deleted - something was here but it has been deleted. Note this is NOT exactly the same as empty. You will see why in a moment.

Insertion

- If it is not there, start looking for the first "open" spot. An open spot is the first probe index that is either deleted or empty.

Searching

The search algorithm is as follows:

- use hash function to find index of where an item should be.
- Check to see if the item's key matches that of key we are looking for
- If it isn't there search for key-value pairs in the "next" slot according to the probing method until either it found, or until an we hit an empty slot.
- NOTE: it is important **not** to search the whole array till you get back to the starting index. As soon as you see an empty slot, your search needs to stop. If you don't, your search will be incredibly slow for any item that doesn't exist.

> 🔥 **DANGER**
>
> Note that only empty slots stop searching not deleted slots

Removal

The removal algorithm is as follows:

- search for the record with matching key.
- If you find it, mark the spot as deleted

Suppose we the following 6 keys and their associated hash indices (these are picked so that collisions will definitely occur). Let us then insert these 5 keys from k1 to k5 in that order.

Key	Hash index
k1	8
k2	7
k3	9
k4	7
k5	8
k6	9

Insert k1 to k3 (no collisions, so they go to their hash indices)

0	1	2	3	4	5	6	7	8	9
Empty	Empty	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
							k2	k1	K3

Insert k4. probe sequence of k4 is $\{(7+0)\%10, (7+1)\%10, (7+2)\%10, (7+3)\%10, (7+4)\%10, (7+5)\%10, \dots\} = \{7, 8, 9, 0, 1, 2 \dots\}$. Thus, we place k4 into index 0 because 7, 8 and 9 are all occupied

	0	1	2	3	4	5	6	7	8	9
	Used	Empty	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
	k4							k2	k1	K3

Insert k5. probe sequence of k5 is $\{(8+0)\%10, (8+1)\%10, (8+2)\%10, (8+3)\%10, (8+4)\%10, (8+5)\%10, \dots\} = \{8, 9, 0, 1, 2, 3 \dots\}$. Thus, we place k5 into index 1 because 8, 9 and 0 are all occupied

	0	1	2	3	4	5	6	7	8	9
	Used	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
	k4	k5						k2	k1	K3

Suppose we then decided to do a search. First lets search for something that isn't there, k6. k6's probe sequence is: $\{(9+0)\%10, (9+1)\%10, (9+2)\%10, (9+3)\%10, (9+4)\%10, (9+5)\%10, \dots\} = \{9, 0, 1, 2, 3, 4 \dots\}$. We begin looking at the first probe index. We proceed until we get to index 2. Since index 2 is empty, we can stop searching

	0	1	2	3	4	5	6	7	8	9
	Used	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
	k4	k5						k2	k1	K3

If we were to search for something that is there (k5 for example), here is what

we would do. Probe sequence for k5 is $\{8, 9, 0, 1, 2, 3 \dots\}$. Thus, we would start search at 8, we would look at indices 8,9,0, and 1. At index 1 we find k5 so we stop

0	1	2	3	4	5	6	7	8	9
Used	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
k4	k5						k2	k1	K3

Suppose we delete k3. All we need to do is find it, and mark the spot as deleted

0	1	2	3	4	5	6	7	8	9
Used	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Deleted
k4	k5						k2	k1	

When a spot is deleted, we still continue when we search... thus if we were to look for k5, we do not stop on deleted, we must keep going.

0	1	2	3	4	5	6	7	8	9
Used	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Deleted
k4	k5						k2	k1	

Quadratic Probing

With linear probing everytime two records get placed beside each other in adjacent slots, we create a higher probability that a third record will result in a collision (think of it as a target that got bigger). One way to avoid this is to use a different probing method so that records are placed further away instead of immediately next to the first spot. In quadratic probing, instead of using the next spot, we use a quadratic formula in the probing sequence. The general form of this algorithm for probe sequence i is: $hash(k) + c_1 i + c_2 i^2$. At it's simplest we can use $hash(k) + i^2$. Thus, we can use: $\{hash(k), (hash(k) + 1)\%m, (hash(k) + 4)\%m, (hash(k) + 9)\%m, \ldots\}$

Key	Hash index
k1	8
k2	7
k3	9
k4	7
k5	8
k6	9

Quadratic Probing Example

Insert k1 to k3 (no collisions, so they go to their hash indices)

	0	1	2	3	4	5	6	7	8	9
	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
								k2	k1	k3

Insert k4. probe sequence of k4 is $\{(7 + 0^2)\%10, (7 + 1^2)\%10, (7 + 2^2)\%10, (7 + 3^2)\%10, (7 + 4^2)\%10, (7 + 5^2)\%10, \ldots\} = \{7, 8, 1, 6, 3, 2 \ldots\}$. Thus, we place k4 into index 1 because 7 and 8 are both occupied

	0	1	2	3	4	5	6	7	8	9
	Empty	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
		k4						k2	k1	k3

Insert k5. probe sequence of k5 is $\{(8 + 0^2)\%10, (8 + 1^2)\%10, (8 + 2^2)\%10, (8 + 3^2)\%10, (8 + 4^2)\%10, (8 + 5^2)\%10, \ldots\} = \{8, 9, 2, 7, 4, 3 \ldots\}$. Thus, we place k5 into index 2 because 8 and 9 are both occupied

	0	1	2	3	4	5	6	7	8	9
	Empty	Used	Used	Empty	Empty	Empty	Empty	Used	Used	Used
		k4	k5					k2	k1	k3

Likewise searching involves probing along its quadratic probing sequence. Thus, searching for k6 involves the probe sequence $\{(9 + 0^2)\%10, (9 + 1^2)\%10, (9 + 2^2)\%10, (9 + 3^2)\%10, (9 + 4^2)\%10, (9 + 5^2)\%10, \ldots\} =$

$\{9, 0, 3, 8, 5, 4 \ldots\}$. We search index 9, then index 0. We can stop at this point as index 0 is empty

0	1	2	3	4	5	6	7	8	9
Empty	Used	Used	Empty	Empty	Empty	Empty	Used	Used	Used
	k4	k5					k2	k1	k3

Double Hashing

Double hashing addresses the same problem as quadratic probing. Instead of using a quadratic sequence to determine the next empty spot, we have 2 different hash functions, $hash_1(k)$ and $hash_2(k)$. We use the first hash function to determine its general position, then use the second to calculate an offset for probes. Thus the probe sequence is calculated as: $\{hash_1(k), (hash_1(k) + hash_2(k))\%m, (hash_1(k) + 2hash_2(k))\%m, (hash_1(k) + 3hash_2(k))\%m, \ldots\}$

Key	Hash1(key)	Hash2(key)
k1	8	6
k2	7	2
k3	9	5
k4	7	4

Key	Hash1(key)	Hash2(key)
k5	8	3
k6	9	2

Double uses a second hash function to calculating a probing offset. Thus, the first hash function locates the record (initial probe index)... should there be a collision, the next probe sequence is a hash2(key) away.

Again we start off with hashing k1, k2 and k3 which do not have any collisions

0	1	2	3	4	5	6	7	8	9
Empty	Empty	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
							k2	k1	K3

Insert k4. probe sequence of k4 is $\{(7+0(4))\%10, (7+1(4))\%10, (7+2(4))\%10, (7+3(4))\%10, \ldots\} = \{7, 1, 5, 9, \ldots\}$. Thus, we place k4 into index 1 because 7 was occupied

0	1	2	3	4	5	6	7	8	9
Empty	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
	k4						k2	k1	k3

Insert k5. probe sequence of k5 is $\{(8+0(3))\%10, (8+1(3))\%10, (8+2(3))\%10, (8+3(3))\%10, \ldots\} = \{8, 1, 4, 7, \ldots\}$. Thus, we place k5 into index 4 because 8 and 1 were occupied

0	1	2	3	4	5	6	7	8	9
Empty	Used	Empty	Empty	Used	Empty	Empty	Used	Used	Used
	k4			k5			k2	k1	k3

Graphs

A graph are made up of a set of vertices and edges that form connections between vertices. If the edges are directed, the graph is sometimes called a digraph. Graphs can be used to model data where we are interested in connections and relationships between data.

```
* flight routes between cities
* social media follower counts
* and more...
```

In this section of the notes we will define what a graph is and how the graph can be represented programmatically.

Graph Definition

The formal definition of a graph is as follows:

A Graph $G = (V, E)$ is made up made of a set of vertices $V = v_1, v_2, v_3...$ and edges $E = e_1, e_2, e_3....$ Each edge in E defines a connection between (u, v), where $u, v \in V$

Basically this means that a graph is made up of vertices and edges. Each edge connects two of the vertices together. A graph is represented visually as follows:
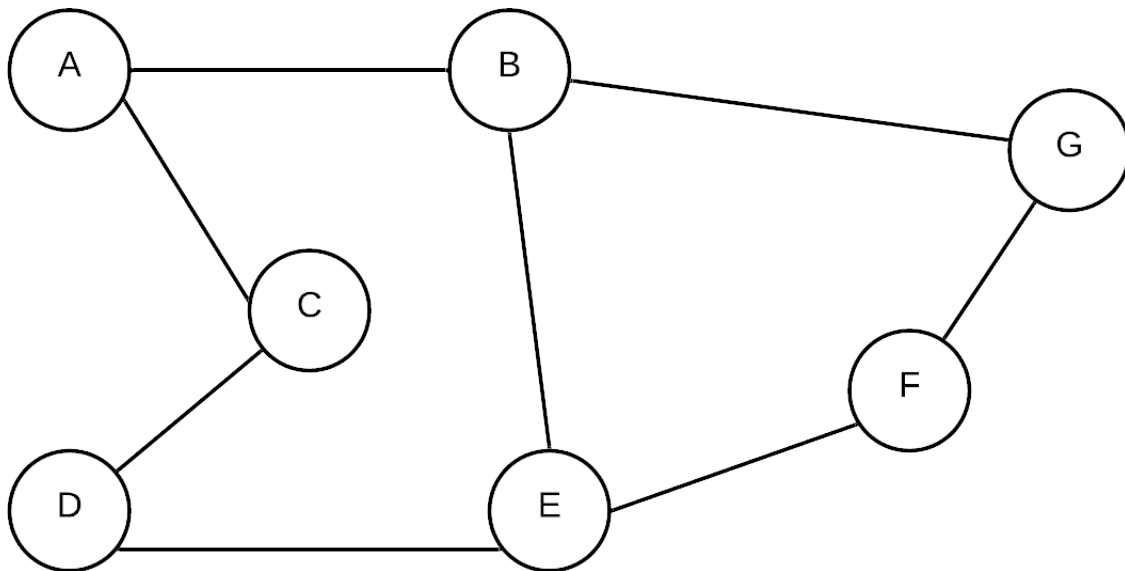
Vertices

A vertex in a graph represents some type of object. The full graph is about

relationships between these objects. For example, suppose you wanted to represent the flights between various airports for an airline. the airports would each be represented by a vertex.

When we look at a graph, we typically label each vertex. For example:
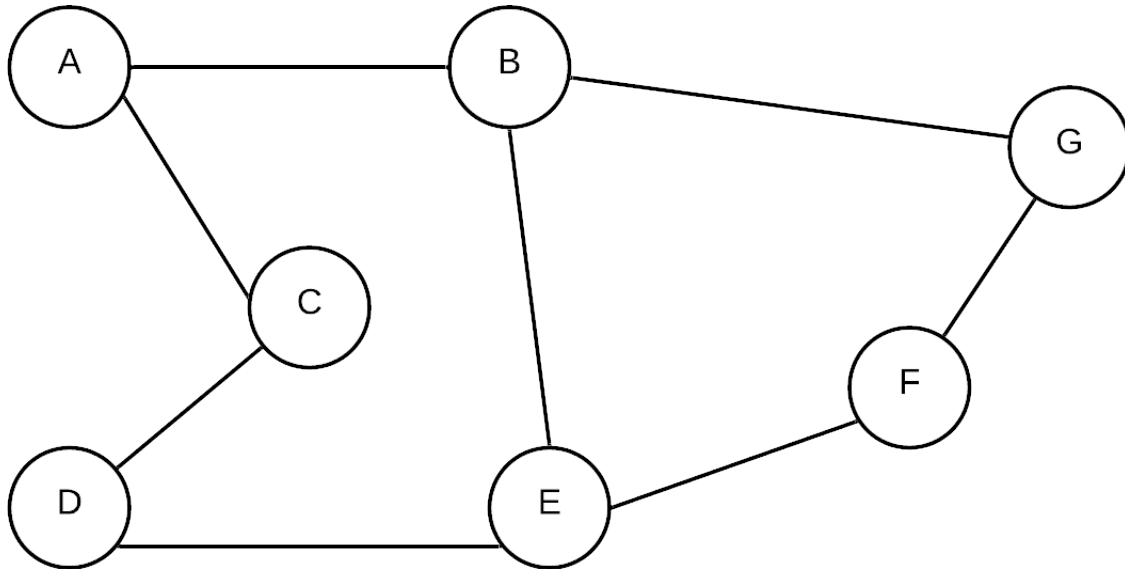
This graph has vertices A, B, C, D, E, F and G. If we were to represent the objects involved with a specific problem, we would store information about each object into records and store them into a table.

::: info

Note that the vertex labels A,B,C etc. are just labels. These labels are used to refer to the vertices in the diagram... in reality we don't necessarily store/name labels in this manner. Often vertices are just identified by a number or has some other identifying feature.

:::

Edges

Edges represent a connection between two vertices.

Aside from identifying a connection between vertices, edges can also have weights. For example:
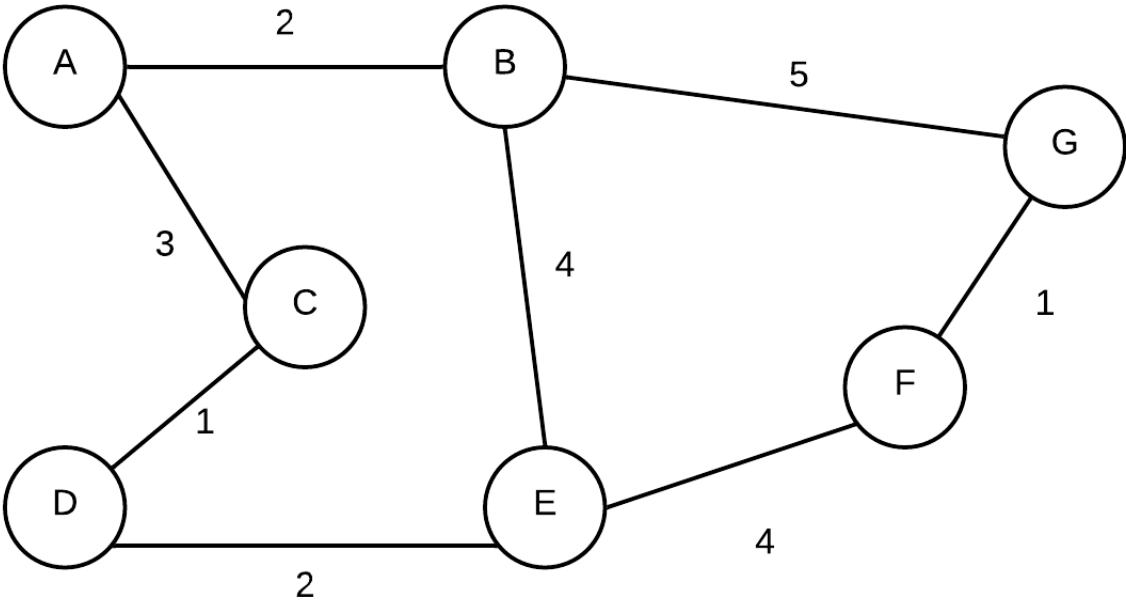
The weights on an edge can act in a way to serve as information about the nature of the connection between two vertices. For example, each vertex can represent a city and the weights, the distance between the cities.

Aside for weights, an edge may also include a direction. Graphs where edges have directions are also called a digraph. The following graph has both weights and direction:
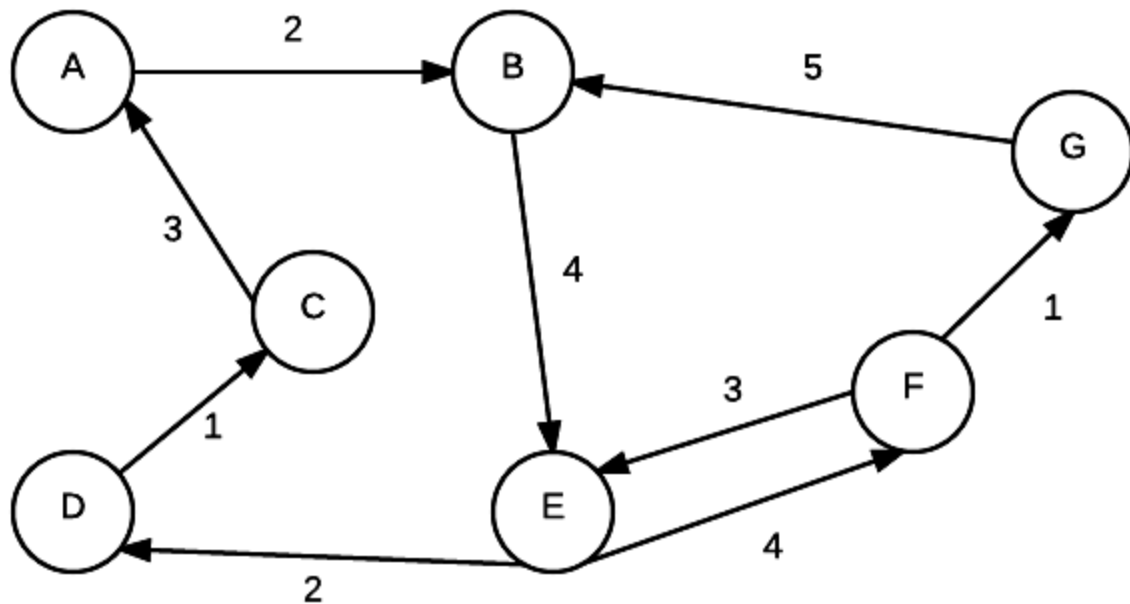
Other Definitions

- adjacent - Given two nodes A and B. B is adjacent to A if there is a connection from A to B. In a digraph if B is adjacent to A, it doesn't mean that A is automatically adjacent to B.
- edge weight/edge cost - a value associated with a connection between two nodes
- path - a ordered sequence of vertices where a connection must exist between consecutive pairs in the sequence.
- simplepath - every vertex in path is distinct
- path length - the number of the edges in a path.
- cycle - a path where the starting and ending node is the same
- strongly connected - If there exists some path from every vertex to every other vertex, the graph is strongly connected.
- weakly connected - if we take away the direction of the edges and there

exists a path from every node to every other node, the digraph is weakly connected.

Representation

To store the info about a graph, there are two general approaches. We will use the following digraph in for examples in each of the following sections.
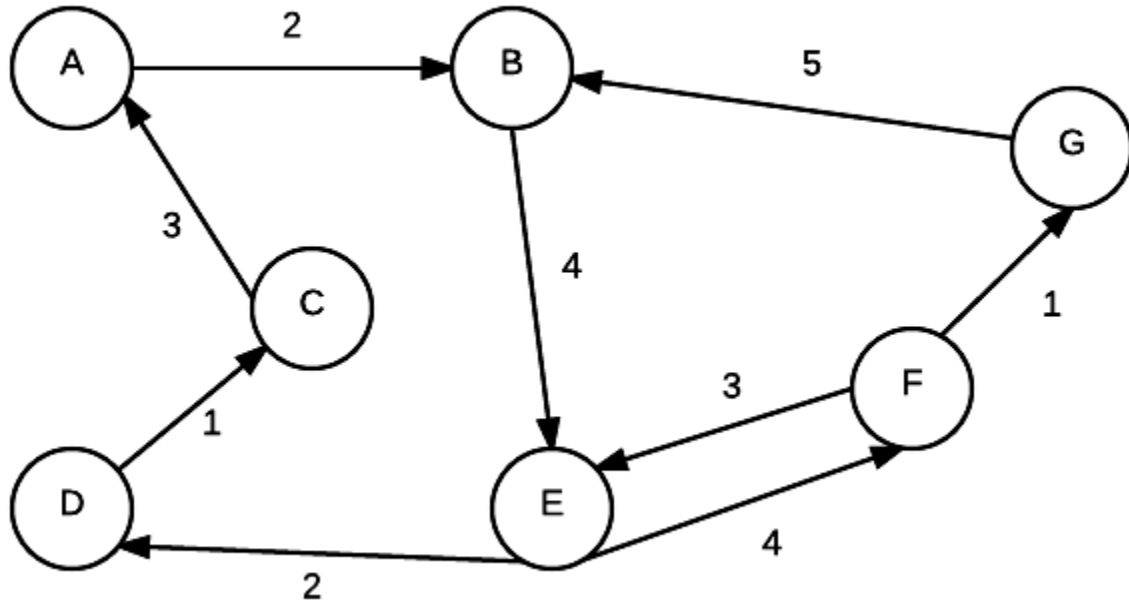
Adjacency Matrix

An adjacency matrix is in essence a 2 dimensional array. Each index value represents a node. When given 2 nodes, you can find out whether or not they are connected by simply checking if the value in corresponding array element is 0 or not. For graphs without weights, 1 represents a connection. 0 represents a non-connection.

The graph adjancey matrix is a good represenation if the graph is dense. It is not good if the graph is sparse as many of the values in the array will be 0.

An adjacency matrix representation can provide an O(1) run time response to the question of whether two given vertices are connected (just look up in table)

	A-0	B-1	C-2	D-3	E-4	F-5	G-6
A-0	0	2	0	0	0	0	0
B-1	0	0	0	0	4	0	0
C-2	3	0	0	0	0	0	0
D-3	0	0	1	0	0	0	0
E-4	0	0	0	2	0	4	0
F-5	0	0	0	0	3	0	1
G-6	0	5	0	0	0	0	0

Adjacency List

An adjacency list uses an array of linked lists to represent a graph. Each element represents a vertex and for each (other) vertex it is connected to, a node is added to its linked list. For graphs with weights each node also stores the weight of the connection to the node. Adjacency lists are much better if the graph is sparse. It takes longer to answer the question of two given vertices are connected (Must go through list to check each node) but it can better answer the question given a single vertex, which vertices are directly reachable from that vertex.

```
┌─────────┐        ┌─────────┐
│         │        │    B    │
│    A    │──────▶ │    2    │
│         │        │         │
├─────────┤        └─────────┘
│         │        ┌─────────┐
│    B    │        │    E    │
│         │──────▶ │    4    │
│         │        │         │
├─────────┤        └─────────┘
│         │        ┌─────────┐
│    C    │        │    A    │
│         │──────▶ │    3    │
│         │        │         │
├─────────┤        └─────────┘
│         │        ┌─────────┐
│    D    │        │    C    │
│         │──────▶ │    1    │
│         │        │         │
├─────────┤        └─────────┘
│         │        ┌─────────┐        ┌─────────┐
│    E    │        │    D    │        │    F    │
│         │──────▶ │    2    │──────▶ │    4    │
│         │        │         │        │         │
├─────────┤        └─────────┘        └─────────┘
│         │        ┌─────────┐        ┌─────────┐
│    F    │        │    E    │        │    G    │
│         │──────▶ │    3    │──────▶ │    1    │
│         │        │         │        │         │
├─────────┤        └─────────┘        └─────────┘
│         │        ┌─────────┐
│    G    │        │    B    │
│         │──────▶ │    5    │
│         │        │         │
└─────────┘        └─────────┘
```
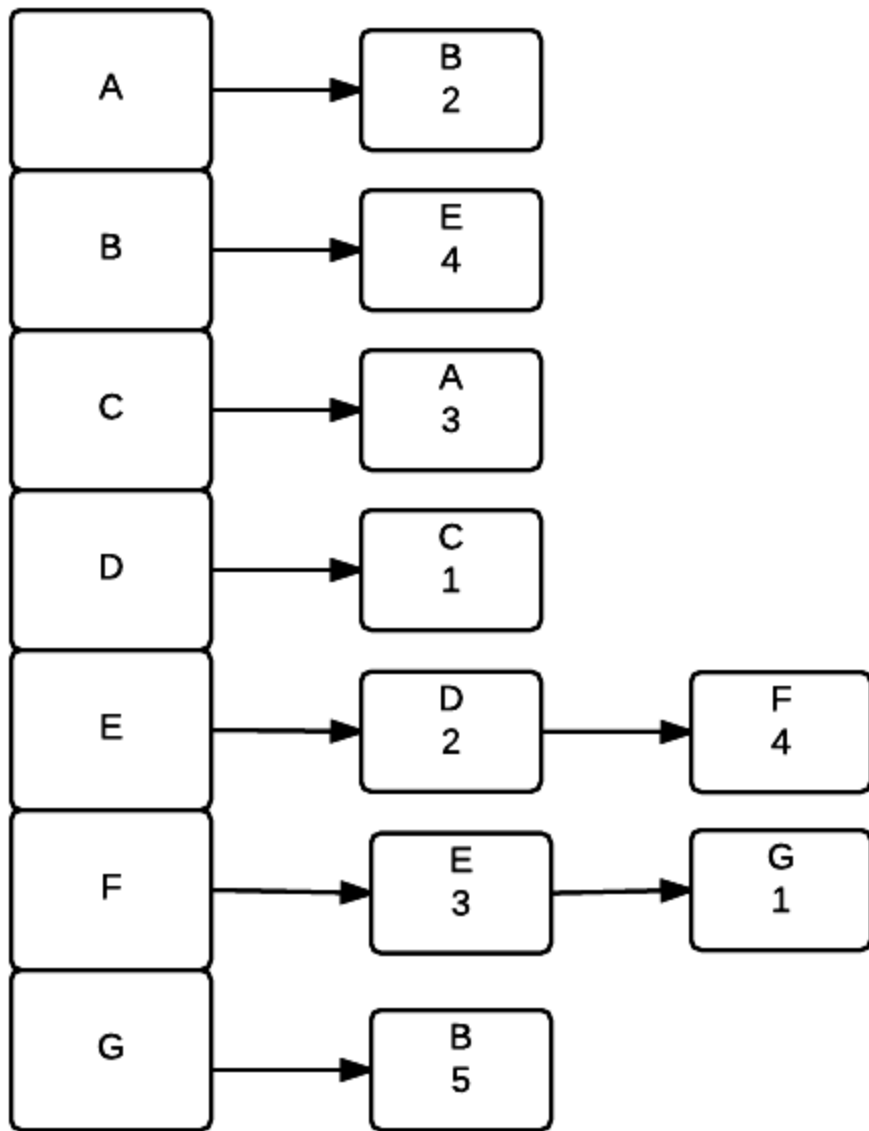
Examples

A graph with no weights or directions
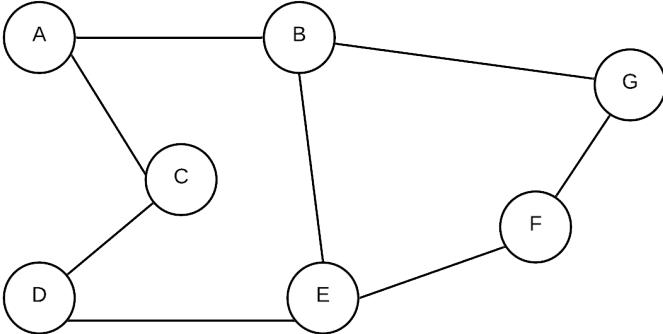
	(A) 0	(B) 1	(C) 2	(D) 3	(E) 4	(F) 5	(G) 6
(A) 0	0	1	1	0	0	0	0
(B) 1	1	0	0	0	1	0	1
(C) 2	1	0	0	1	0	0	0
(D) 3	0	0	1	0	1	0	0
(E) 4	0	1	0	1	0	1	0
(F) 5	0	0	0	0	1	0	1
(G) 6	0	1	0	0	0	1	0
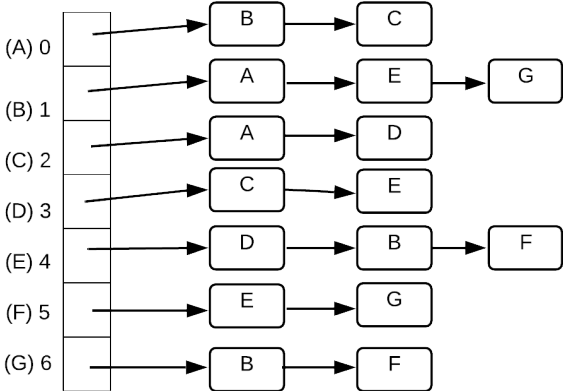
A graph with weights but no directions
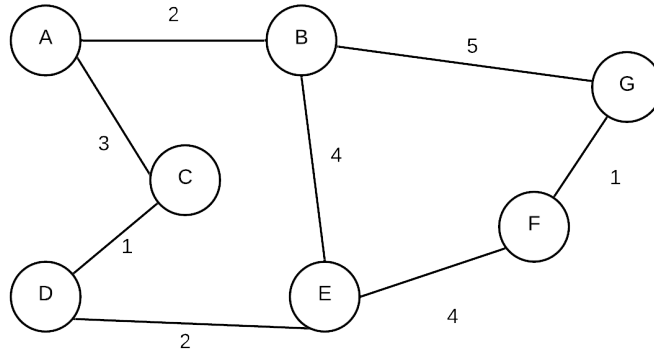
	(A) 0	(B) 1	(C) 2	(D) 3	(E) 4	(F) 5	(G) 6
(A) 0	0	2	3	0	0	0	0
(B) 1	2	0	0	0	4	0	5
(C) 2	3	0	0	1	0	0	0
(D) 3	0	0	1	0	2	0	0
(E) 4	0	4	0	2	0	4	0
(F) 5	0	0	0	0	4	0	1
(G) 6	0	5	0	0	0	1	0
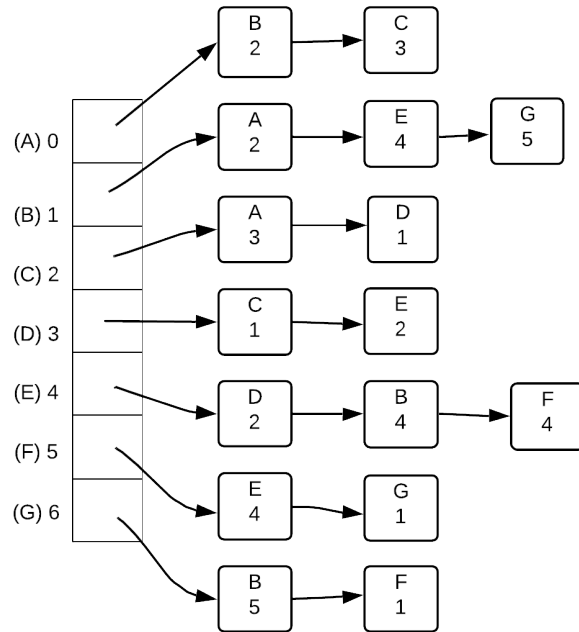
A graph with weights and directions
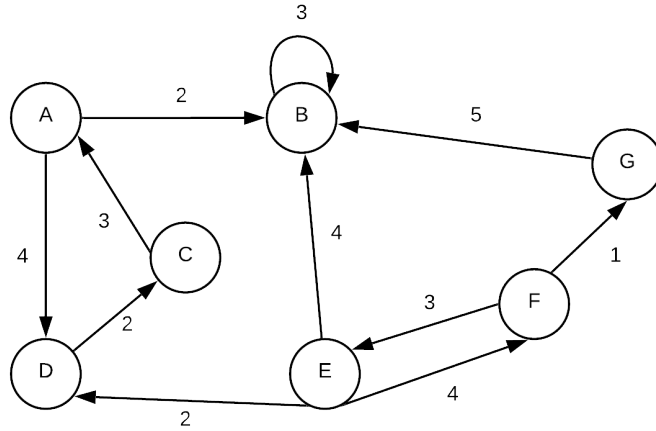
to: from	(A) 0	(B) 1	(C) 2	(D) 3	(E) 4	(F) 5	(G) 6
(A) 0	0	2	0	4	0	0	0
(B) 1	0	3	0	0	0	0	0
(C) 2	3	0	0	0	0	0	0
(D) 3	0	0	2	0	0	0	0
(E) 4	0	4	0	2	0	4	0
(F) 5	0	0	0	0	3	0	1
(G) 6	0	5	0	0	0	0	0
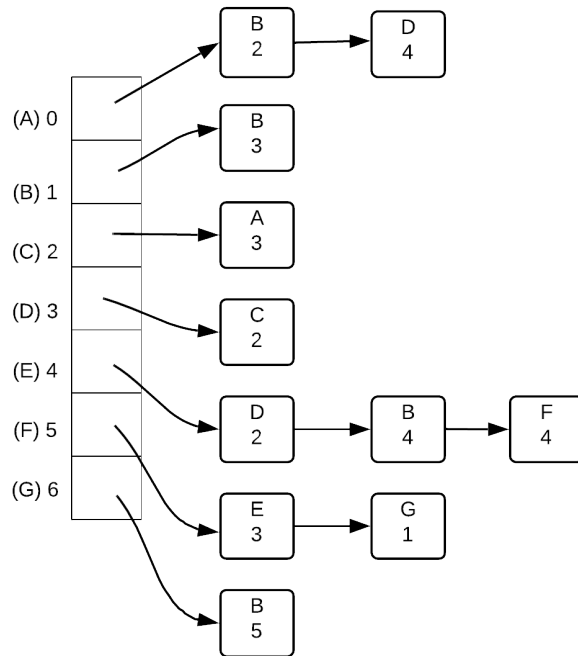
Dijkstra's Algorithm

Dijkstra's Algorithm is an algorithm for finding the shortest path from one vertex to every other vertex. This algorithm is an example of a ***greedy algorithm.*** Greedy algorithms are algorithms that find a solution by picking the best solution encountered thus far and expand on the solution. Dijkstra's Algorithm was first conceived by Edsger W. Dijkstra.

The general algorithm can be described as follows:

1. Start at the chosen vertex (we'll call it v1).
2. Store the cost to travel to each vertex that can be reached directly from v1.
3. Next look at the vertex that is least costly to reach from v1 (we'll call this vertex v2 for this example).
4. For each vertex that is reachable from v2, find the total cost to that vertex starting from v1 (we want the total cost, not just the lowest cost from v2). Update the cost of travel to these vertices if it is lower than the current known cost.
5. Pick the next lowest cost and continue.

Example

Consider the following graph (non-directed)
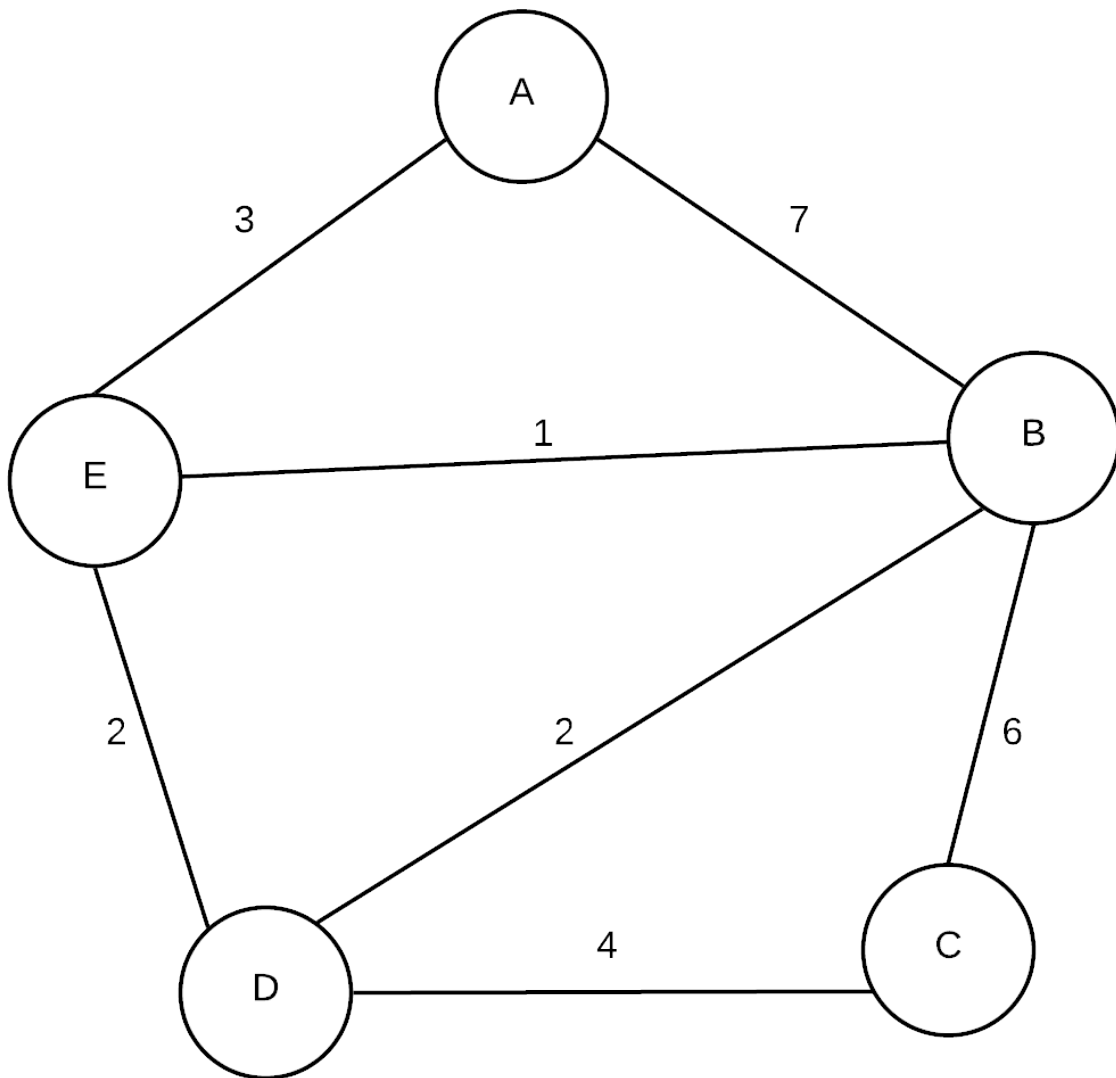
Initial state

We will use Dijkstra's algorithm to find the shortest distance from **A** to every other vertex. To track how we are doing, we will create a table that will track the shortest distances from A we have encountered so far as well as the "previous" node in the path (so if we follow all previous nodes, we end up back at **A**). We also track whether we "know" about the shortest path from **A** to each

node... that is, have we already been able to find the shortest path to that node from **A**. Currently (at the beginning,) we do not know which nodes are reachable from A so we store infinity into every spot other than A. The cost of getting to A from A is 0. Initially the table looks like this:

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		false
B	∞		false
C	∞		false
D	∞		false
E	∞		false

Expand A

Now, starting from **A**, there are two vertices that reachable, **B** and **E**

We store the total distance from **A** into the table for both these vertices if it is less than the shortest distance so far. At this point, **A** is now known as we have considered all its neighbours

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		**true**

Vertex	Shortest Distance from A	Previous Vertex	Known
B	7	A	false
C	∞		false
D	∞		false
E	3	A	false

Expand E

Now, we continue from here by considering the unknown vertex that has the shortest distance to **A**. In this case it is 3 (smallest in distance table with a false value in known). In implementation, we actually would use a priority queue (heap) to store the vertex with its distance and simply dequeue the smallest distance and check that the vertex was still unknown.

In any case, we are going to now consider **E**. From **E** the unknown neighbours are **B** and **D**. The distance to **D** (from A) is 5 (3 + 2). The distance to **B** is 4 (3 + 1). Now, The value stored previously for D was ∞ so we replace that with 5. The distance to **B** was 7, so we replace that also. We also mark that E is now known

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		true
B	4	E	false

Vertex	Shortest Distance from A	Previous Vertex	Known
C	∞		false
D	**5**	**E**	false
E	3	A	**true**

Expand B

The next vertex we consider is the one with the shortest distance to A that is still not known. This is now B.

From B, we can reach unknown neighbours C and D. The distance to C (from A) is 10 (4+6). The distance to D (from A) is 6 (4 + 2). The value stored in the table for C was ∞ so we replace that. The value stored in D was 5, so that entry is not replaced because this new distance is NOT smaller.

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		true
B	4	E	**true**
C	**10**	**B**	false
D	5	E	false
E	3	A	true

Expand D

The next vertex that with the smallest distance to A that is unknown is D. From D, there is only one unknown vertex (C). The distance to C is 9 (5 + 4). As this is smaller than what is stored, we will update the table

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		true
B	4	E	true
C	**9**	**D**	false
D	5	E	**true**
E	3	A	true

Expand C

The only vertex left is C. From C, there are no unknown neighbours so we mark C as known and we are done.

Vertex	Shortest Distance from A	Previous Vertex	Known
A	0		true
B	4	E	true

Vertex	Shortest Distance from A	Previous Vertex	Known
C	9	D	true
D	5	E	true
E	3	A	true

Shortest path

To find the shortest distance to A, we simply need to look it up in the final table.

However, if we want to find the shortest path it isn't so simple as a look up. We actually need to work away backwards by looking up the previous vertices. For example, to find shortest path from A to C, we see that C's previous was D. D's previous was E. E's previous was A. Thus, the path is:

A --> E --> D --> C

Minimum Spanning Trees

A spanning tree is a connected, acyclic subgraph of a graph $G = (V, E)$. That is it is the subset of edges that are connected and acyclic. If G itself is not connected, then we can generalize this to a spanning forest.

A minimum spanning tree (MST) of a graph $G = (V, E)$ with weight function $w(e)$ for each $e \in E$, is a spanning tree that has the smallest total weight.

How to find a minimum spanning tree

To find an MST, we can either start with every edge in the graph and remove edges till we get an MST or we can select edges till we form an MST.

For the removal method, we can start out with every edge in the graph and start eliminating the edges with the highest weights. An edge can be removed as long as we don't disconnect some vertex (ie if that is the only edge to a vertex, we can't remove it). We keep removing edges until we have an MST. However, this is not practical. If there are n nodes in a graph, a spanning tree has n-1 edges. potentially there are $\frac{(n)(n-1)}{2}$ edges in total. This means that any algorithm that finds an MST in this manner would end up with worst case $\Omega(n^2)$

Instead of removing edges, we can find the MST by building the MST instead. The general generic algorithm for doing this is as follows:

```
greedyMST(G=(V,E)){
    T=[];  //empty set of edges
    while(T is not an MST){
        find an edge e from E that is safe for T
        add e to T
    }
}
```

Now... this is a very generic algorithm and there are parts not yet defined. Lets start with looking for an e being safe for T. An edge e is safe for T iff adding e to T makes T subset of some MST of G. Of course... problem is how do we know if e is part of some MST of G when that is exactly what this function is suppose to find?

> **⚠ INFO**
>
> Theorem: If G is a connected, undirected weighted graph and T is a subset of some MST of G and e is any edge of minimum weight that are in different connected components of T, then adding e is safe for T (See text for proof)

Kruskal's Algorithm

Kruskal's algorithm starts by sorting edges according to weights. It then picks the smallest edge out and adds it to T only if the end points are in different connected components. If we use something like a BFS or DFS to find connected components, it will be very slow. Instead, what we can do is use a disjoint set

```
KruskalsMST(G=(V,E)){
    T=[];  //set of edges that form MST, initially empty
```
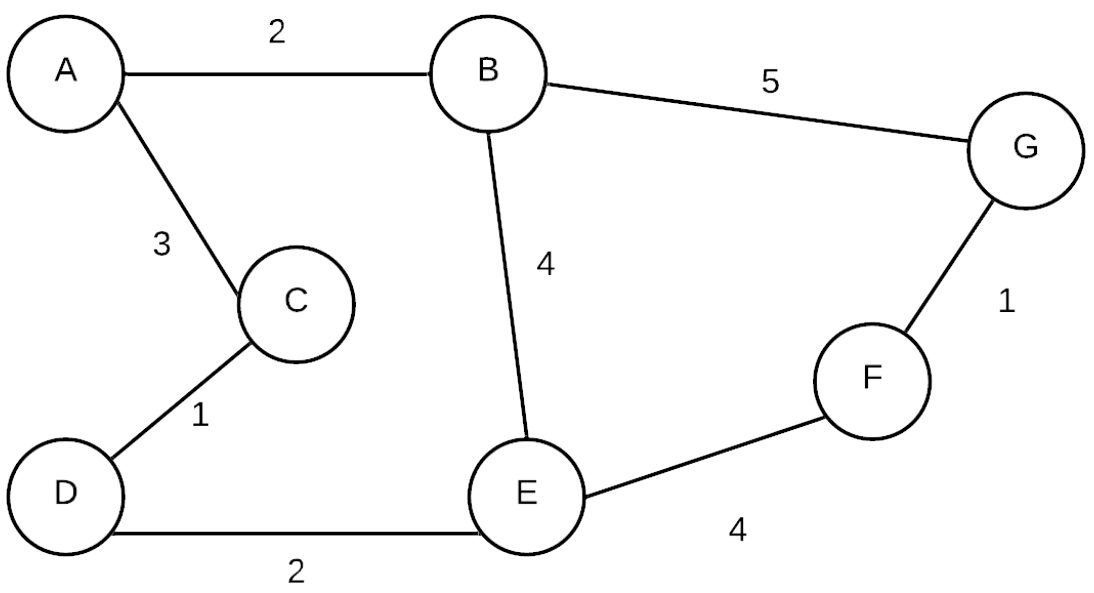
Example
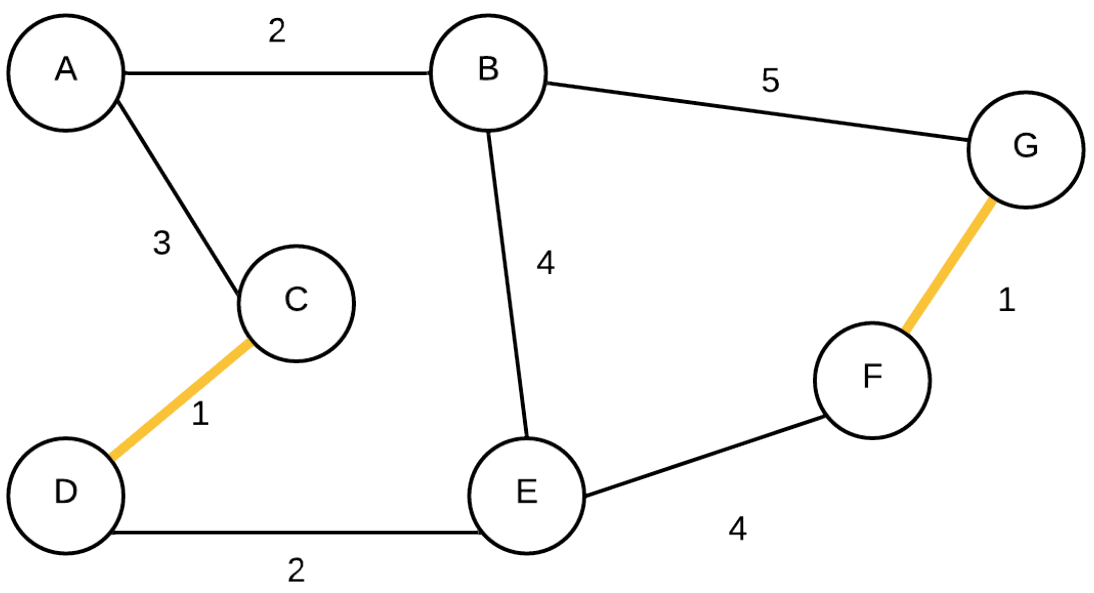
So let us consider the following graph:
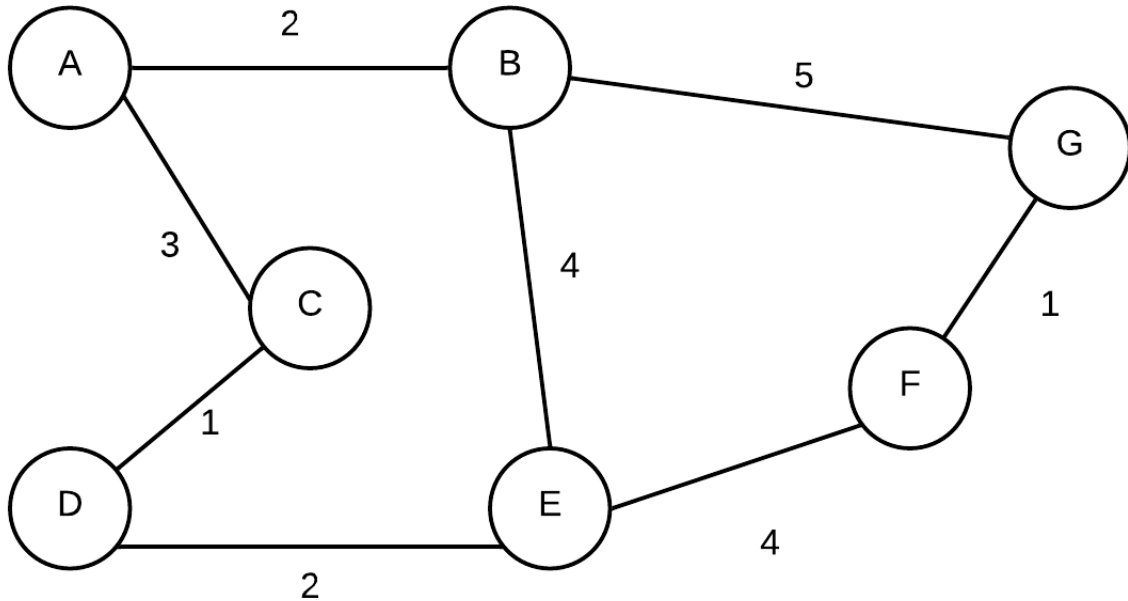
Given the graph above, the edges sorted in non-descending order by weight are: (C,D), (F,G), (A,B),(D,E),(A,C), (B,E), (E,F),(B,G). We will exam the edges in this order in the example below

Step #	graph, T shown in yellow	
Initial	 T= {}	{

Step #	graph, T shown in yellow
1	

graph, T shown in yellow

T={(C,D)}

Step #	graph, T shown in yellow
2	

T={(C,D), (F,G)}
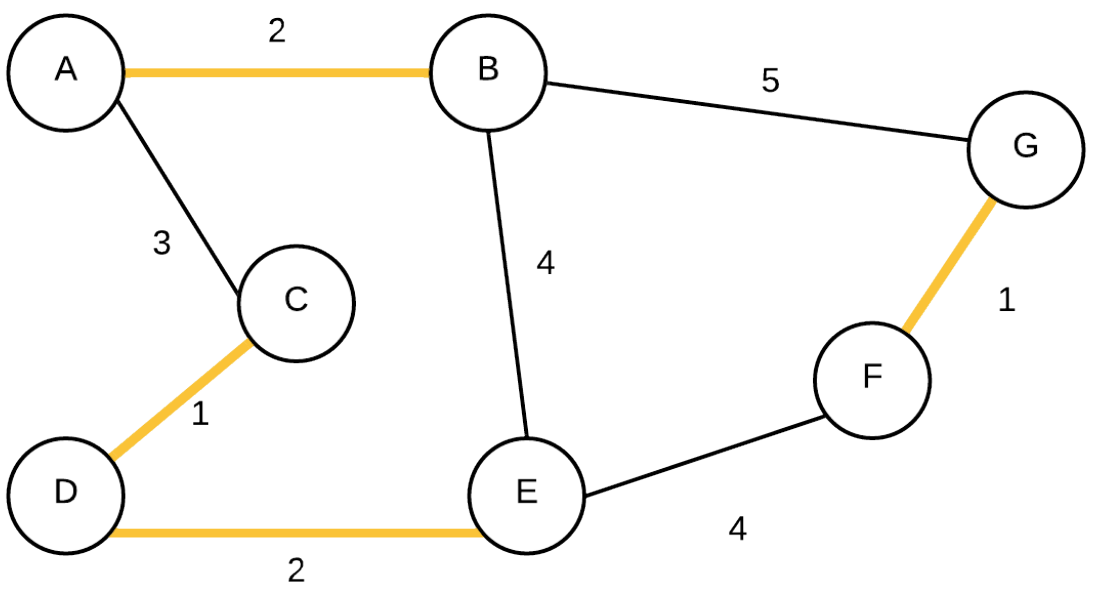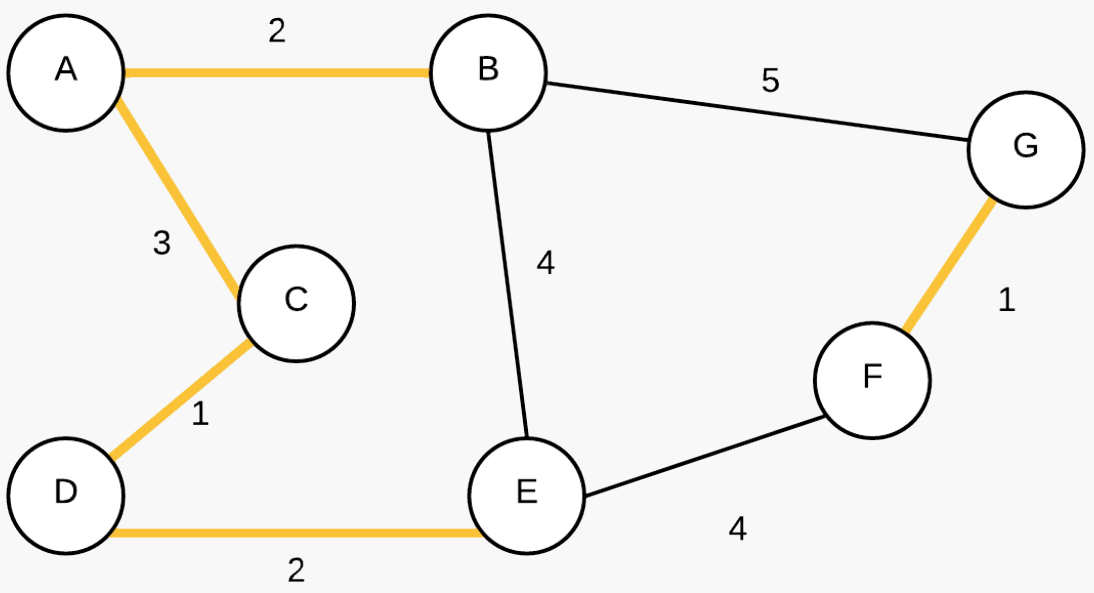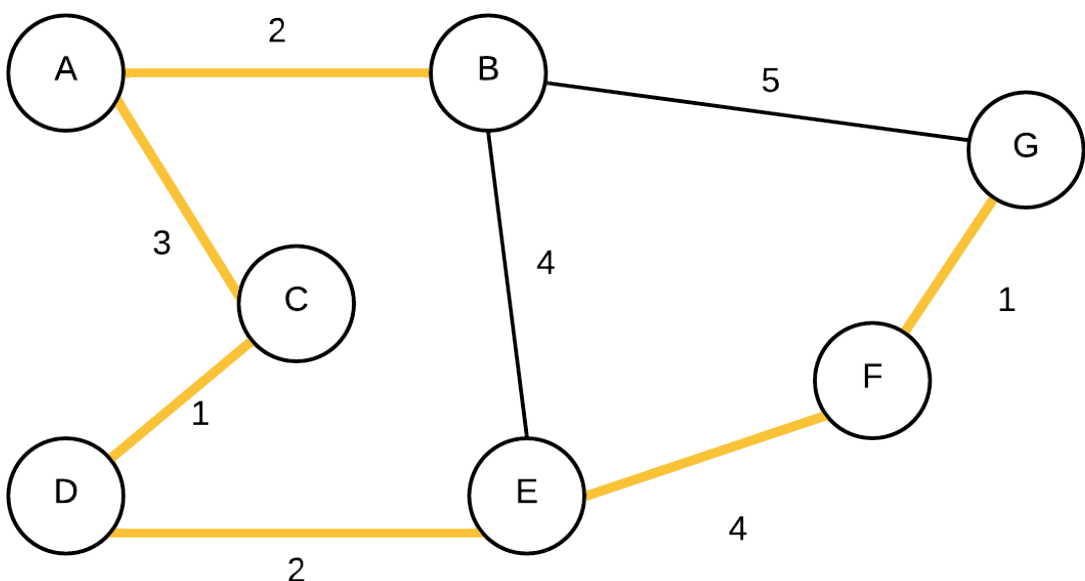
Step #	graph, T shown in yellow
3
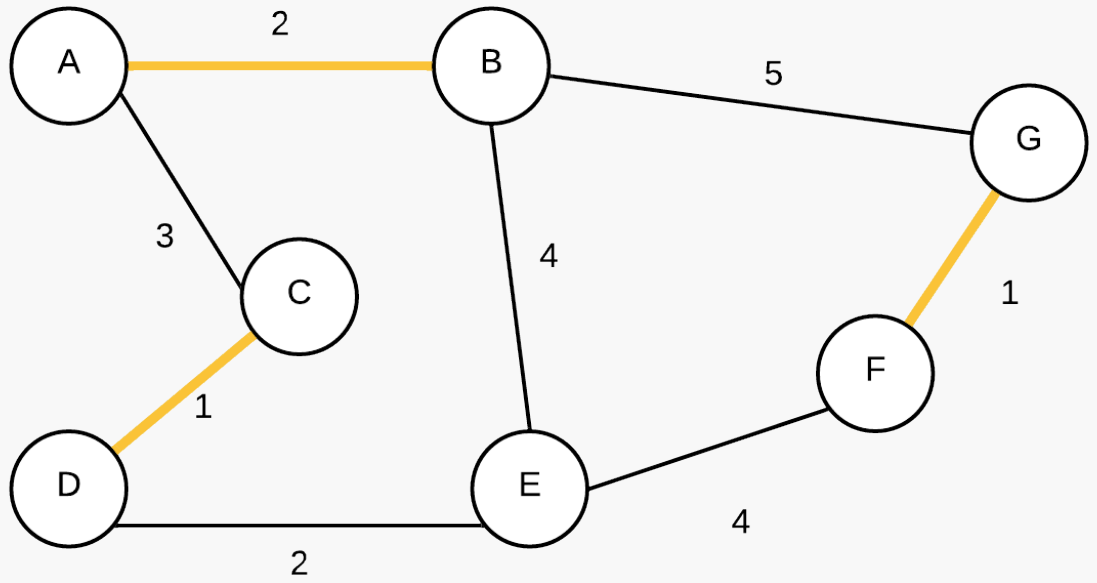
T={(C,D), (F,G),(A,B)} |

Step #	graph, T shown in yellow
4	

T={(C,D), (F,G),(A,B),(D,E)}

Step #	graph, T shown in yellow
5	 T={(C,D), (F,G),(A,B),(D,E),(A,C)}

Step #	graph, T shown in yellow
6	 T={(C,D), (F,G),(A,B),(D,E),(A,C)}
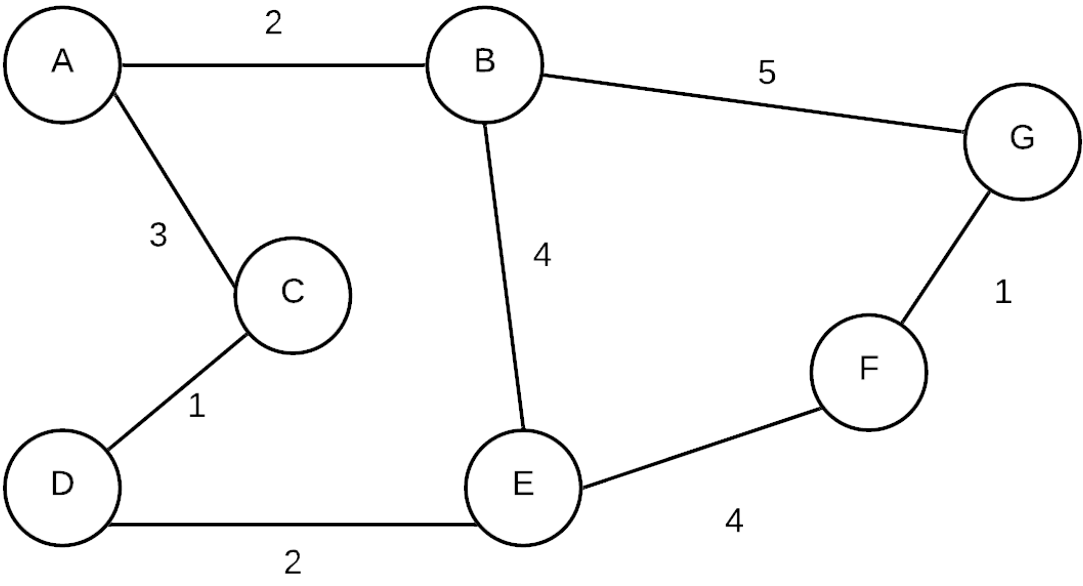
```
    We next consider (E,F) and as they are in different sets we
connect them.
    (B,G) are in same disjoint set so we are now done |
```

Prim's Algorithm

We pick a vertex $v \in V$ to be the "root" of the MST. After that we simply grow the tree by joining isolated vertices one at a time. An isolated vertex is any vertex that isn't part of the MST yet picking the smallest edge weight. To support this, we will use a MinHeap. We queue into this heap edges that will

connect an isolated vertex with the current MST. We use infinity if there is no direct edge yet to any vertex in the MST

Step	Graph	
Initial		
Initial		

Step	Graph	
state. pick A as root		
1		
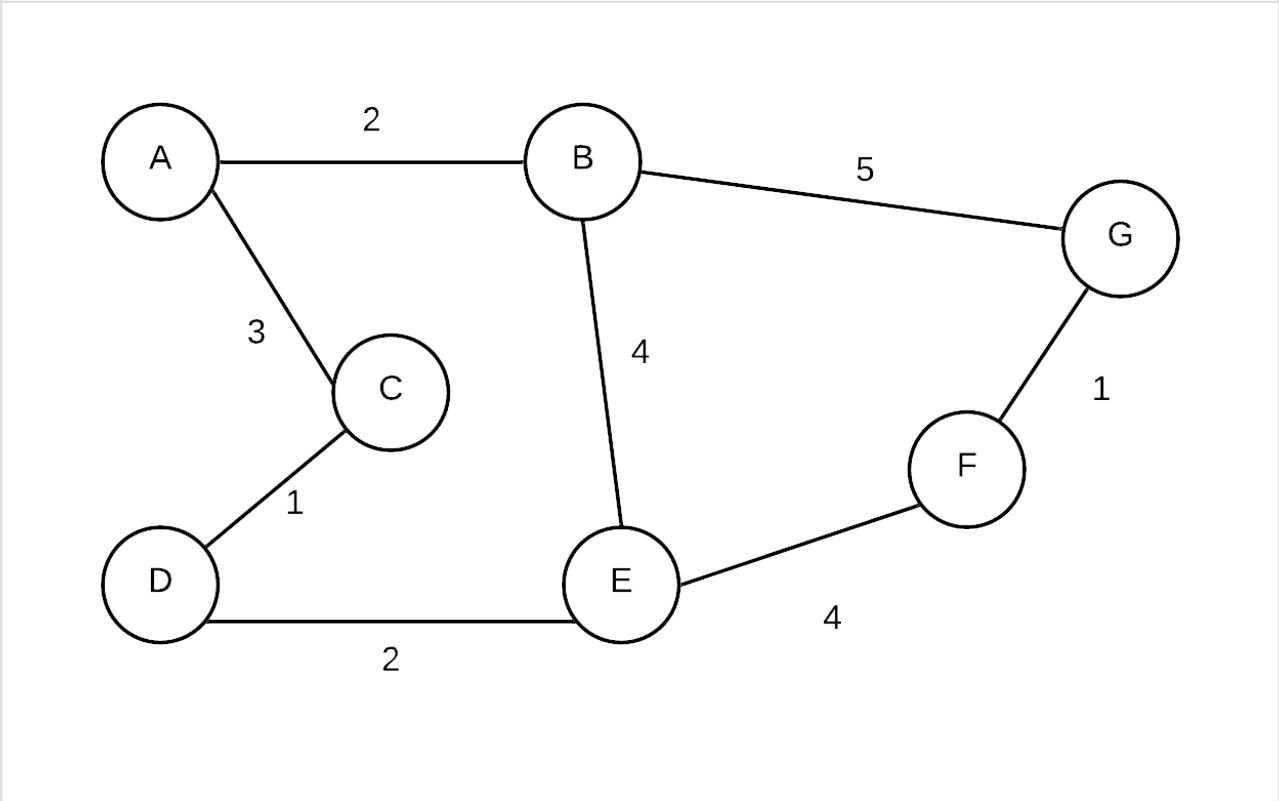
Step	Graph	
	T={}	
2	 T={(A,B)}	
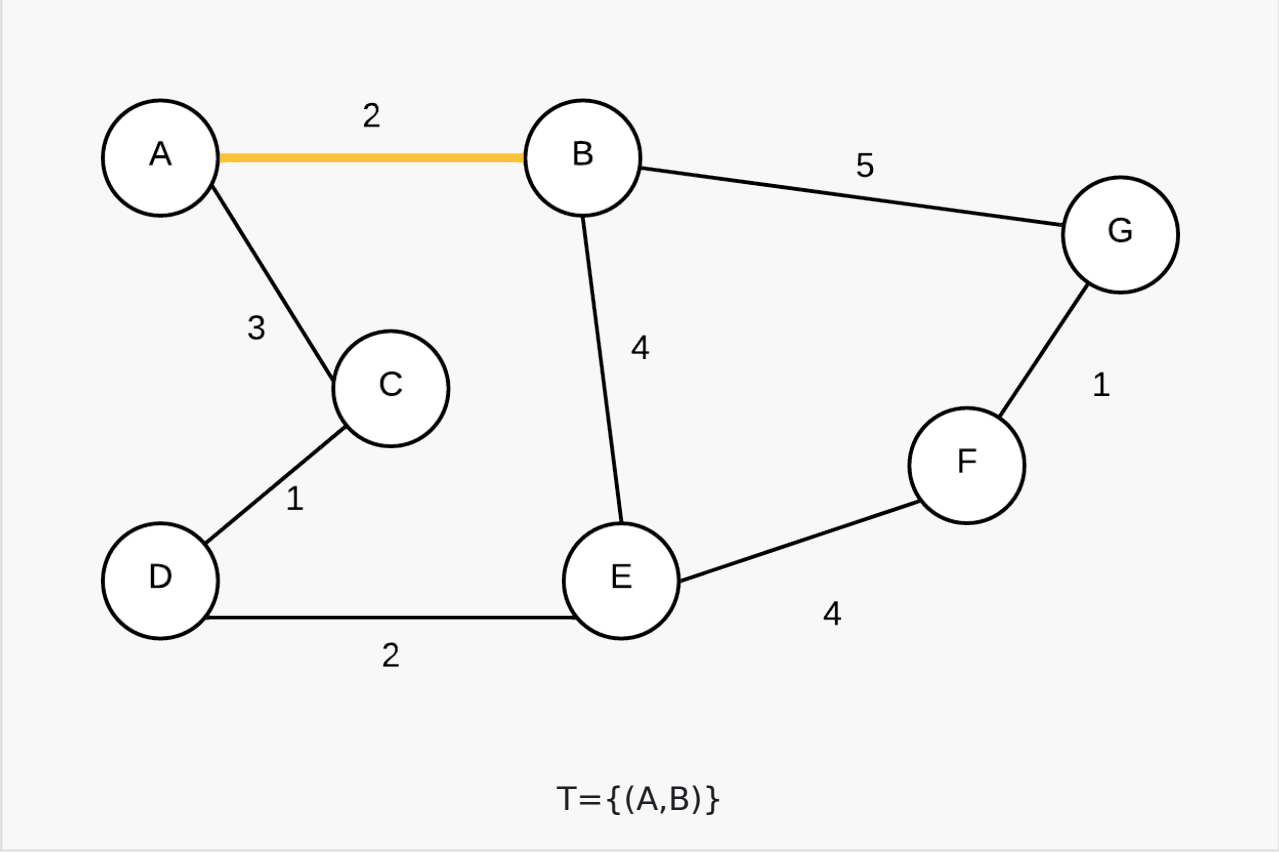
Step	Graph	
	Graph	
3	T={(A,B), (A,C)}	
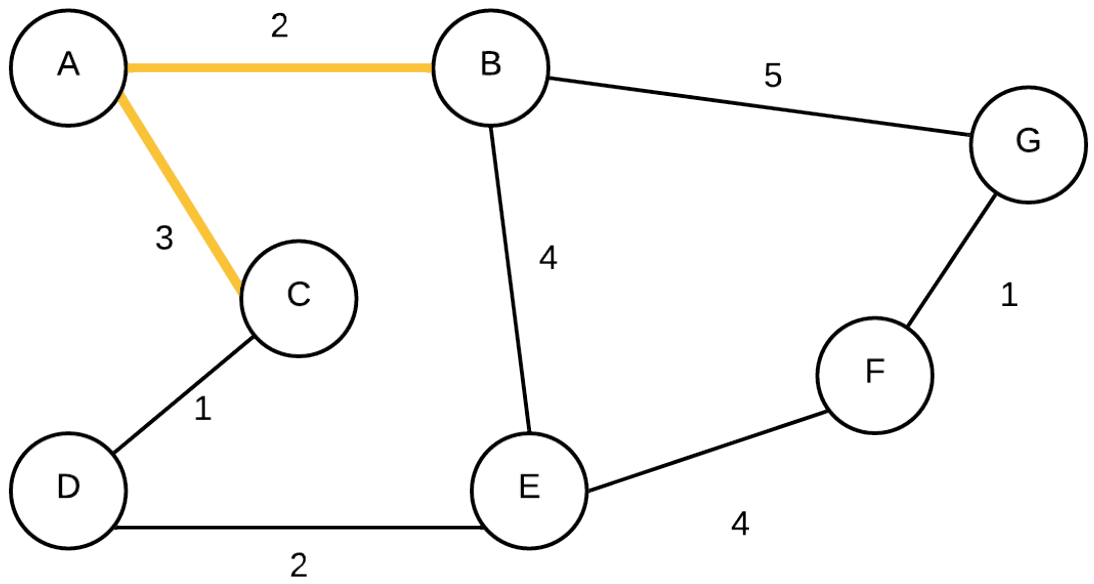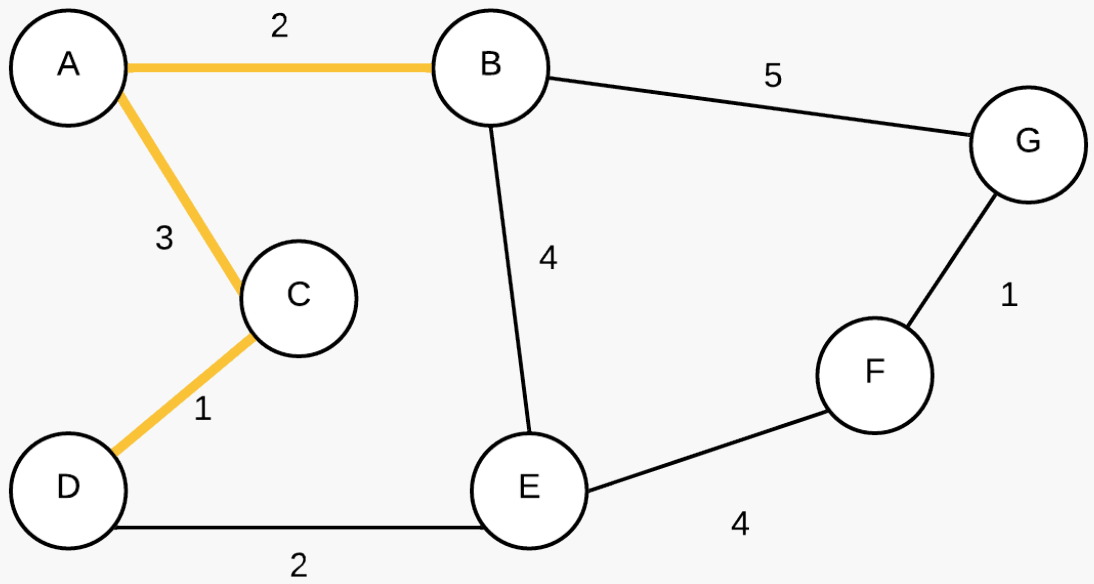
Step	Graph	
	Graph	
4	T={(A,B),(A,C),(C,D)}	
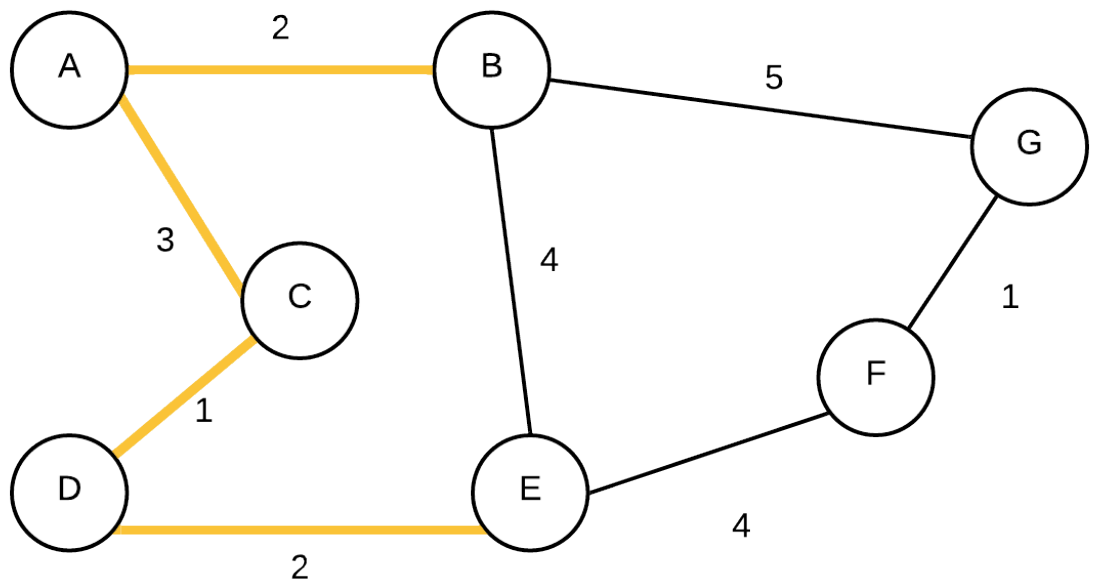
Step	Graph	
	Graph	
5	T={(A,B),(A,C),(C,D),(D,E)}	
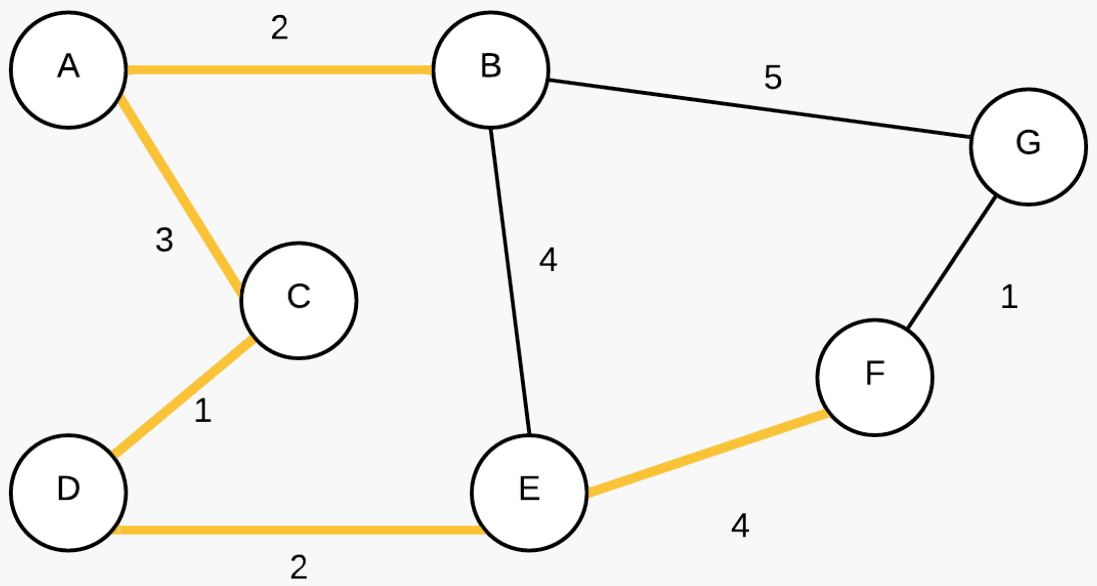
Step	Graph	
	Graph	
6	T={(A,B),(A,C),(C,D),(D,E),(E,F)}	
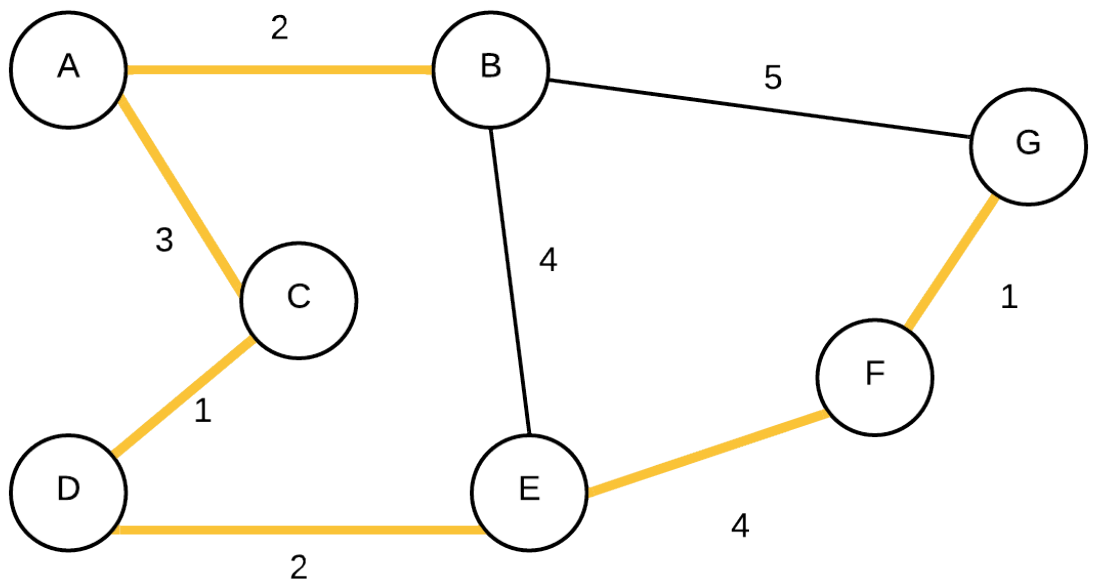
Step	Graph	
	Graph	
7	T={(A,B),(A,C),(C,D),(D,E),(E,F),(F,G)}	

Trees

A tree is a very important data structure. It has the ability to classify data and separate it reducing the overhead of search. It is used in many areas of computing. This section deals will introduce you to the idea of trees and some basic algorithms surrounding them.

Definitions

This section will include some basic terminology used when describing trees. To help you understand the terminology, use the following diagram:
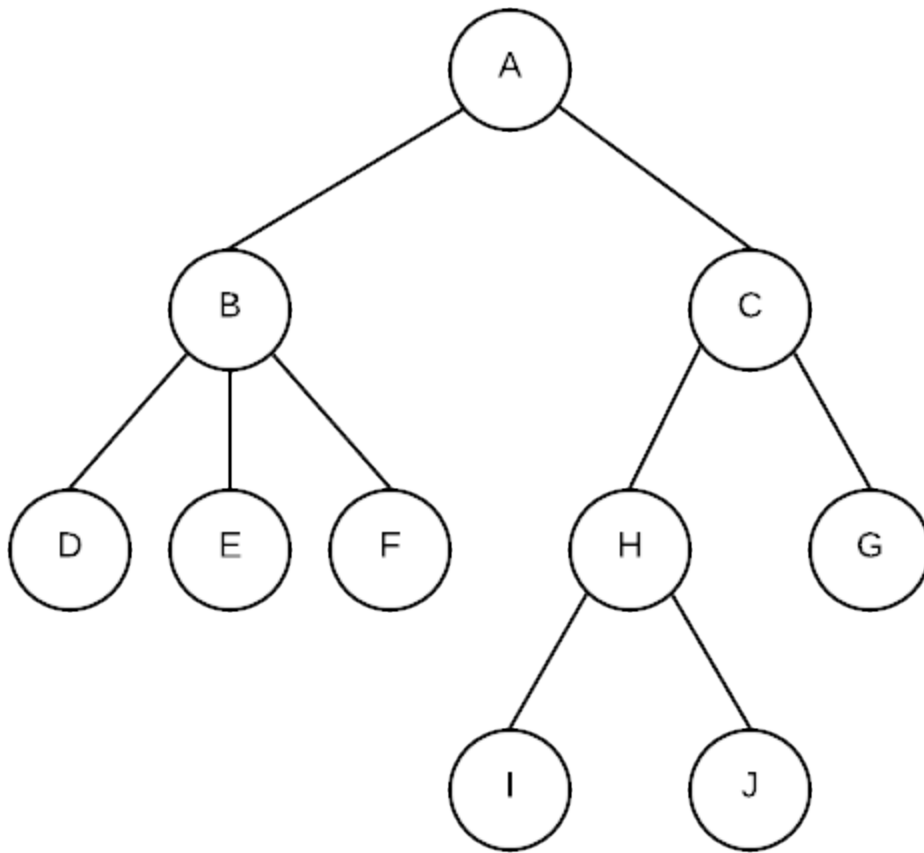
Node: the thing that store the data within the tree . (each circle in the above diagram is a node)

Root Node: the top most node from which all other nodes come from. A is the root node of the tree.

Subtree: Some portion of the entire tree, includes a node (the root of the subtree) and every node that goes downwards from there. A is the root of the entire tree. B is the root of the subtree containing B,D,E and F

Empty trees: A tree with no nodes

Leaf Node: A node with only empty subtrees (no children) Ex. D,E,F,I,J,and G are all leaf nodes

Children: the nodes that is directly 1 link down from a node is that node's child node. Ex. B is the child of A. I is the child of H

Parent the node that is directly 1 link up from a node. Ex. A is parent of B. H is the parent of I

Sibling: All nodes that have the same parent node are siblings Ex. E and F are siblings of D but H is not

Ancestor: All nodes that can be reached by moving only in an upward direction in the tree. Ex. C, A and H are all ancestors of I but G and B are not.

Descendants or **Successors** of a node are nodes that can be reached by only going down in the tree. Ex. Descendants of C are G,H,I and J

Depth: Distance from root node of tree. Root node is at depth 0. B and C are at depth 1. Nodes at depth 2 are D,E,F,G and H. Nodes at depth 3 are I and J

Height: Total number of nodes from root to furthest leaf. Our tree has a height of 4.

Path: Set of branches taken to connect an ancestor of a node to the node. Usually described by the set of nodes encountered along the path.

Binary tree: A binary tree is a tree where every node has 2 subtrees that are also binary trees. The subtrees may be empty. Each node has a left child and a right child. Our tree is NOT a binary tree because B has 3 children.

The following are NOT trees
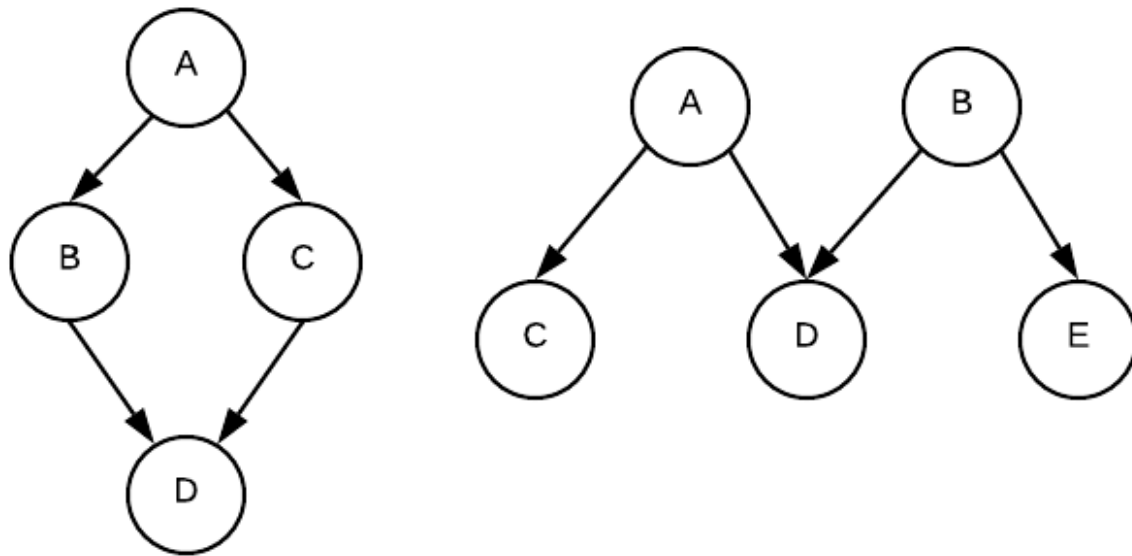
Tree Implementation

By its nature, trees are typically implemented using a Node/link data structure like that of a linked list. In a linked list, each node has data and it has a next (and sometimes also a previous) pointer. In a tree each node has data and a number of node pointers that point to the node's children.

However, a linked structure is not the only way to implement a tree. We will see this when in the next pages use an array/a list to represent a binary heap. In this case, the root will be stored at index 0 and for any node stored at index i, we will calculate the left subtree's root, the right subtree's root, and the parent node's indexes with formulas. However, if you do this for non-complete trees, you would end up with many unused spaces in the array/list.

Heap and Heap Sort

Heap Sort is a sort based on the building and destroying of a binary heap. The binary heap is an implementation of a **Priority Queue.** Normally, when you have a queue, the first value in is the first value out. However with a priority Queue the value that comes out of the queue next depends on the priority of that value.

Basic operations on the binary Heap include:

- insert - add an item to the binary heap
- delete - removes the item with the highest priority in the binary heap.

The idea for the Heap sort is this: put every value in the array into the binary heap using its value as the priority, then repeatedly call delete to remove the smallest/largest value and put it back into the array.

Priority Queues using Binary Heaps

A priority queue is an abstract data type. Like queues and stacks, it is an idea, the underlying storage and access is decoupled from the functionality of a priority queue. A priority queue is similar to a queue in that you use it for ordering data. However, instead of ordering based on when something was added, you order it based on the priority value of the item. An item at the front of the priority queue (and the item that will be removed if an item is to be dequeued) is the item with the highest priority. This is the type of queues you might find in a hospital emergency room. The thing that matters more is the severity of the illness as opposed to who got there first.

If you were to implement a priority queue using a regular list, you would essentially have to maintain a sorted list (sorted by priority). This would mean the following:

1. insertions would be O(n) for sure as you would need to go through an already sorted list trying to find the right place then doing the insertion.
2. removal is from the "front" of the queue, so depending on how you do your implementation, this can be potentially O(n) also.

This is not very efficient. If we were to create our priority queue in this manner the run time for both operations would not be very good. If we then base a sort on such a data structure, its run time would be no better than one of the simple sorts. Thus, clearly this isn't how we should implement a priority queue.

The interesting thing about a priority queue is that we really don't care where any value other than the one with the highest priority is. We care about where the highest priority item is and we want to be able to find it and remove it

quickly but all other values can be essentially anywhere.

A binary heap is a data structure that can help us achieve this.

Binary Heap - Basic Definitions

Before we begin with the definitions below, and since they involve trees, it may be a good idea to look at the tree definitions first so that the definitions with respect to heaps make more sense.

See: Tree Definitions

Binary Heap - A binary heap is a **complete binary tree** where the **heap order property** is always maintained.

Binary Tree - A binary tree is either a) empty (no nodes), or b) contains a root node with two children which are both binary trees.

Complete Binary Tree - A binary tree where there are no missing nodes in all except at the bottom level. At the bottom level the missing nodes must be to the right of all other nodes

These binary trees are complete.
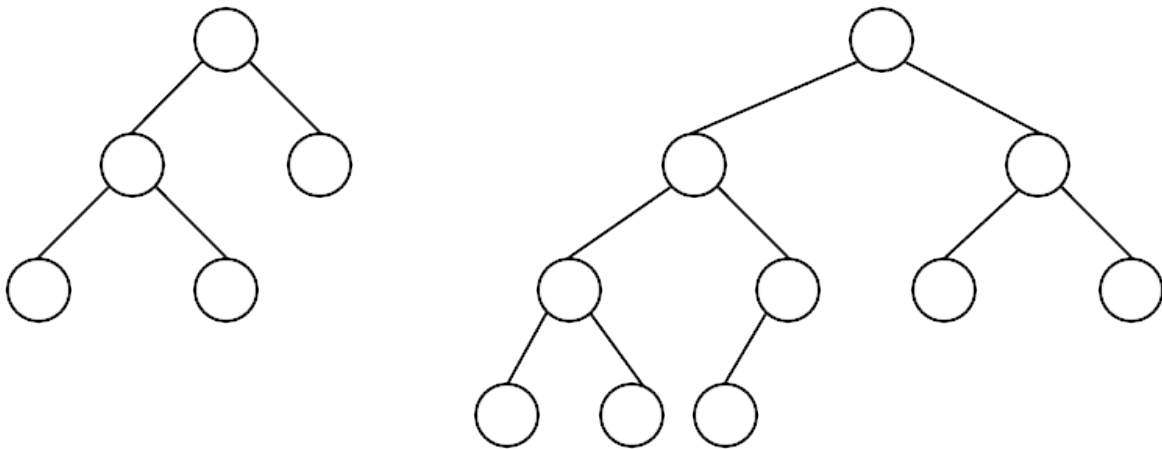
These binary trees are not complete. The first one is missing a node one level higher than leaves, the second is missing further left in the tree than an existing node
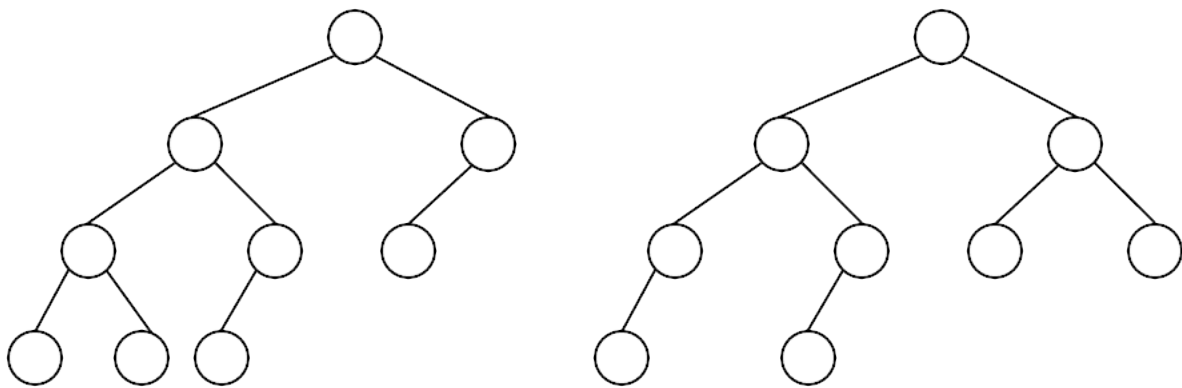
Heap Order Property: For each node, the parent of the node must have a higher priority, while its children must have a lower priority. There is no ordering of priority other than this rule. Thus, the highest priority item will be at the root of the tree. Below is a heap where we define the smaller value as having higher priority:
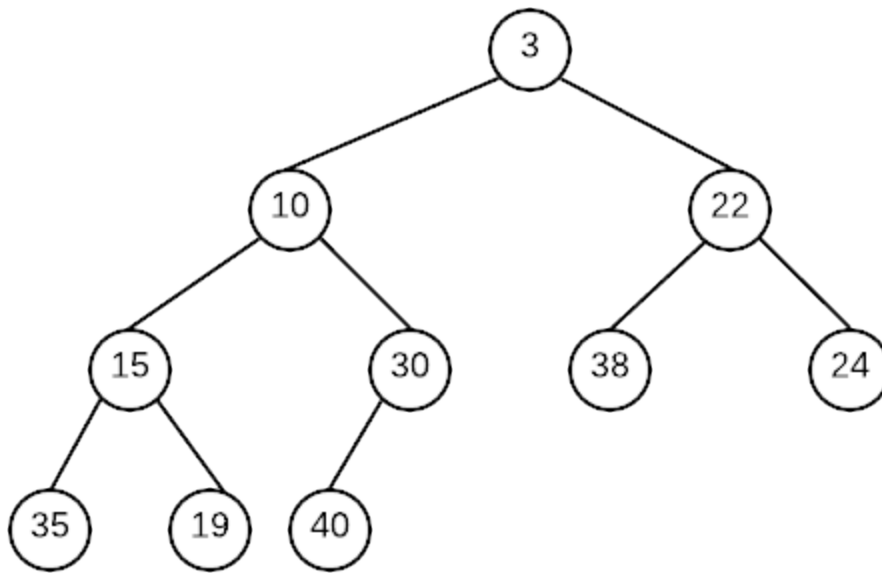
Insertion

Insertion into a heap must maintain both the complete binary tree structure and the heap order property. To do this what we do is the following.

- create a new empty node in the left most open spot at the bottom level of the tree
- If value can be placed into node without violating heap order property put it in
- otherwise pull the value from parent into the empty node
- repeat the previous two steps until the value can be placed

This process effectively creates an empty node starting at the bottom of the tree. The empty node moves up until it is in the correct position and the value can be placed inside the empty node. This process of moving the empty node towards the root is called **_percolate up_**

Example

Create a min heap (meanning that a node with a smaller value has more priority over another node with a larger value) by inserting these number in the order given: 10, 6, 20, 5, 16, 17, 13, 2

Insert 10 :

create empty node

put number in

Insert 6:

create empty node

could not be placed
pull down parent

put number in

Insert 20:

create empty node → put number in

Insert 5

create empty node → could not be placed pull down parent → could not be placed pull down parent → put number in

Insert 16

create empty node → put number in

Insert 17

create empty node

could not be placed
pull down parent

put number in

Insert 13

create empty node

could not be placed
pull down parent

put number in

Insert 2

create empty node → could not be placed pull down parent → could not be placed pull down parent → could not be placed pull down parent → put number in

Removal

The highest priority value is always the value that is removed from the binary heap. The way that the Heap is set up, the node with the highest priority is at the root. Finding it is easy but the removal process must ensure that both the complete binary tree structure along with the heap order property is maintained. Therefore, just removing the root would be a bad idea.

In order for the complete binary tree property to be maintained we will be removing the right most node at the bottom level. Note that a complete binary tree with n nodes can only have 1 shape, so the shape is pretty much determined by the fact that removing a value creates a tree with one fewer

node.

The empty spot that had been created by the removal of the value at root must be filled and the value that had been in the rightmost node must go back into the heap. We can accomplish this by doing the following:

- If the value could be placed into the empty node (remember, this starts at root) without violating the Heap Order Property, put it in and we are done
- otherwise move the child with the higher priority up (the empty spot moves down).
- Repeat until value is placed

The process of moving the empty spot down the heap is called **_percolate down_**

Example: (Note, removal always removes value from root)

Original Heap

10 | remove bottom, rightmost node, store value into a temp

10 | Empty out the root

Try to put value from temp into the node, doesn't fit in empty node

pull up higher priority child

pull up higher priority child

put number in
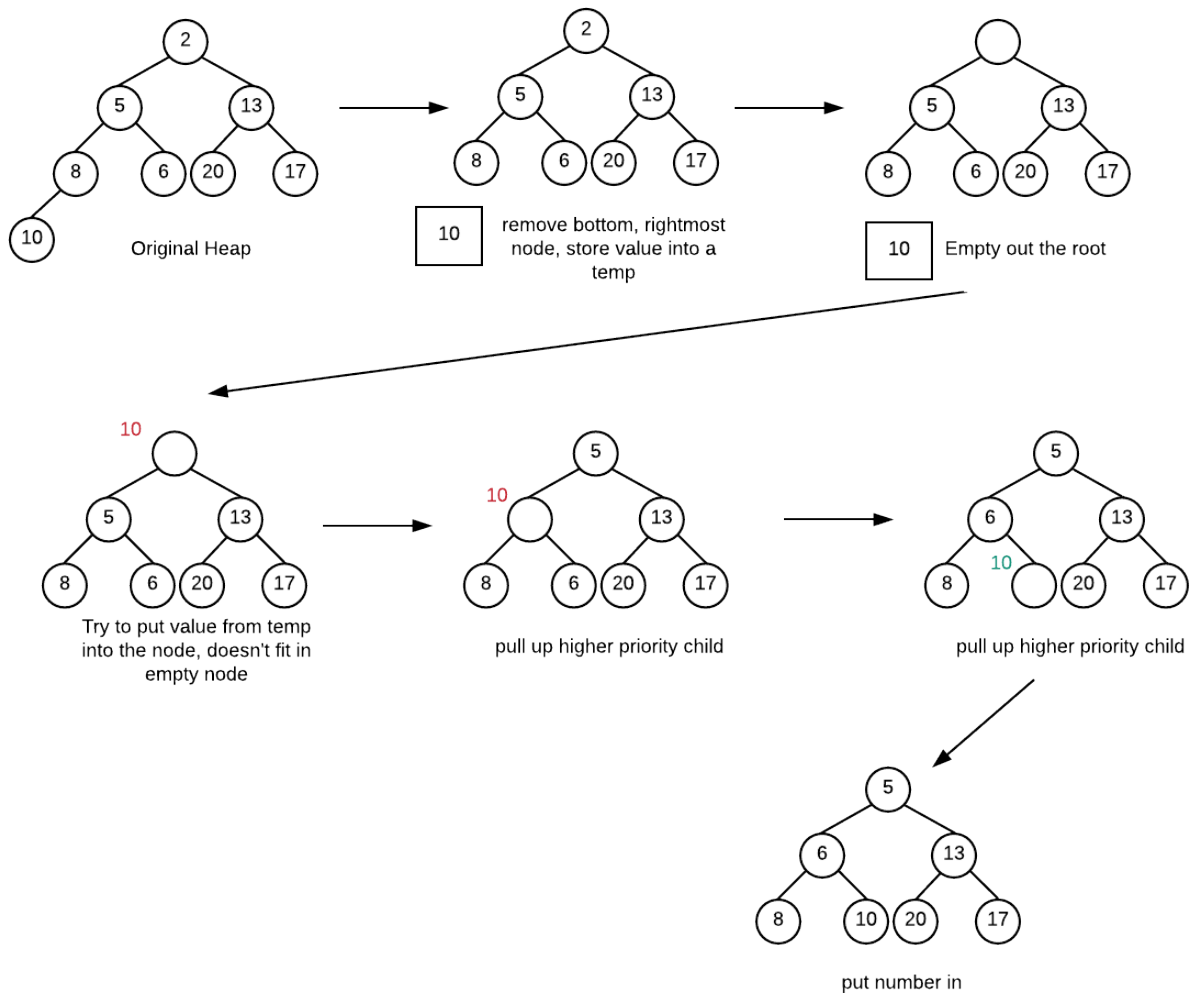
Implementation

Clearly we want to implement the heap in as simple a manner as possible. Although we can use a link structure to represent each node and their children, this structure is actually not that good for a heap. One reason is that each node needs two extra pointers so for small data the memory needed for the pointers

could easily exceed that of the data. Also, we really don't need the linking structure because our tree is a complete binary tree. This makes it very easy for us to use an array to represent our tree.

Idea is this. Store the data in successive array elements. Use the index values to find the child and parent of any node.

Suppose data was stored in element i. The left child of that node is in element 2i+1 The right child of the node is in element 2i+2 The parent of the node is stored in (i-1)/2 (for all but root node which has no parent)
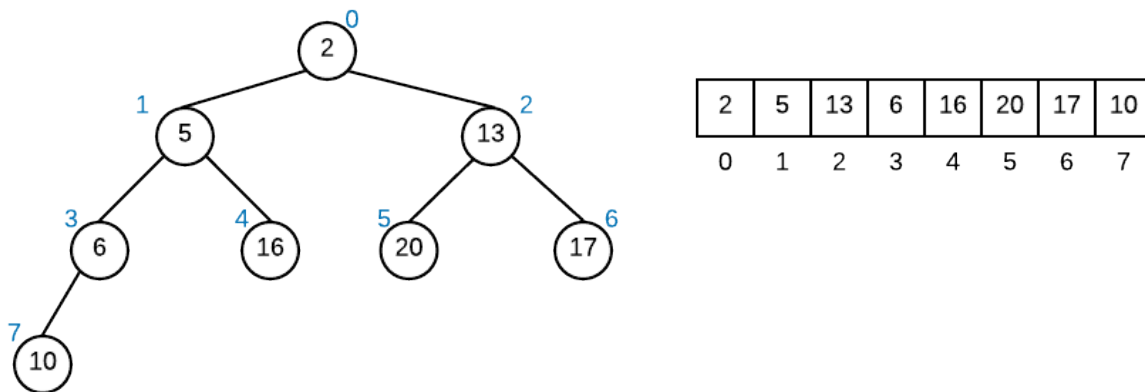
This representation is very convenient because when we add values to the end of the array it is like we are adding a node to the leftmost available spot at the bottom level.

We do not have empty elements in the array/list simply because of the definition of the heap (that it is a complete binary tree)

We also eliminate any need for pointers or the overhead of allocating nodes as we go.

Heapify and Heap Sort

Recall that Heap Sort basically operates by building a heap with *n* values then destroying the heap.

A complete binary tree with n nodes means that at most there are log n nodes from the root (top) to a leaf (a node at the bottom of the tree)

Insertion may require the percolate up process. The number of times a node needs to percolate up can be no more than the number of nodes from the root to the leaf. Therefore, it is pretty easy to see that this percolate up process is O(log n) for a tree with n nodes. This means that to add a value to a Heap is a process that is O(log n). In order for us to build a heap we need to insert into it n times. Therefore, the worst case runtime for this process is O(n log n)

The deletion process may require a percolate down process. Like the percolate up process this also is O(log n). Thus, to remove a value from the heap is O(log n). We need to remove n values so this process is O(n log n).

To heap sort we build heap O(n log n) then destroy the heap O(n log n). Therefore the whole sorting algorithm has a runtime of O(n log n)

Our heap can be represented with an array. The simplest implementation of heap sort would be to create a heap object:

> ⚠ **CAUTION**
>
> A simple but not good implementation of heap sort
>
> **Python**   **C++**
> _____

```python
def heap_sort(mylist):
    the_heap=Heap()      # we assume we have created some Heap
Object
    size = len(mylist)
    for i in range(0,size):
        the_heap.insert(mylist[i])

    for i in range(0,size):
        mylist[i]=the_heap.front()
        the_heap.delete()
```

```cpp
//simple but not correct way to heap sort an array
void heapSort(int data[],int size){
    Heap theheap;
    for(int i=0;i<size;i++){
        theheap.insert(data[i]);
    }
    for(int i=0;i<size;i++){
        data[i]=theheap.front();
        theheap.delete();
    }
}
```

but this is not actually a good implementation. The above would create a heap as it goes meaning that we need to actually double the data storage as the heap itself would be an array that is as big as data. Thus, the function would have the same drawback as merge sort. Furthermore building a heap by multiple insertions is actually slower than making a heap in place. Thus, instead of building an actual heap with a heap object, we will implement heap sort by turning the array into a heap in place then removing from it in place.

One way we can do this by borrowing the idea from insertion sort... in that algorithm we start off by hiving off the first element and saying that it is a sorted array. We then "insert" successive elements into this sorted array so that it stays sorted.

Remember that a heap uses every element of the array without any sort of gaps. If you have a heap with n elements, you need n element array to represent the heap. If you took a value out, then the array size decreases by 1. After doing a removal, you effectively end up with an open spot at the back of the array. Thus, we sort by filling those open spots with the value being removed, forming a sorted list as the values being removed will come out based on their priority ordering.

Thus heap sort works like this:

1. we build a heap in place (turn the array of unsorted numbers into a heap)
2. we remove a value from the heap and place that value at the end of the array.

One key thing we have to change though is the priority. We actually need to ensure that items that should go towards the end of the array is considered highest priority for the heap. Thus if we were sorting an array in ascending order (small to big) we should give larger values a higher priority.

Heapify

Since our heap is actually implemented with an array, it would be good to have a way to actually create a heap in place starting with an array that isn't a heap and ending with an array that is heap. While it is possible to simply "insert" values into the heap repeatedly, the faster way to perform this task is an algorithm called Heapify.

In the Heapify Algorithm, works like this:

Given a node within the heap where both its left and right children are proper heaps (maintains proper heap order) , do the following:

- If the node has higher priority than both children, we are done, the entire heap is a proper heap
- Otherwise
  - swap current node with higher priority child
  - heapify() that subtree

Effectively what is happening is that we already know that both the left and right children are proper heaps. If the current node is higher priority than both children then the entire heap must be proper

However if its not then we do a swap, this means that the current node's value has now gone down into one of the subtrees. This could cause that subtree to no longer be a heap because the root of that subtree now has a value of lower priority than it use to have. so we heapify the subtree.

Building a maxheap in place

This example will start with an array that is not heap. It will perform heapify() on it to form a max heap (bigger values have higher priority). In each of the diagrams below, the argument to heapify() is the index corresponding to the node
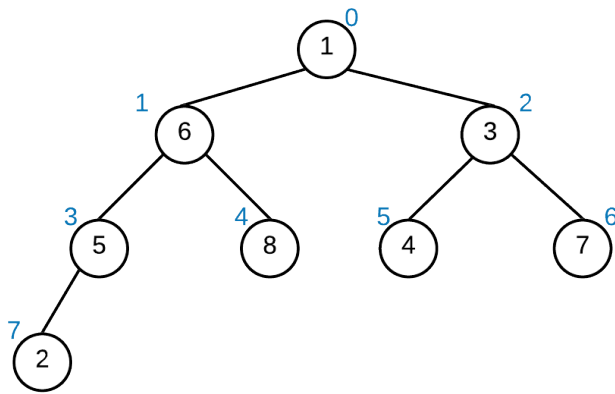
1	6	3	5	8	4	7	2
0	1	2	3	4	5	6	7

Firstly, all leaf nodes are valid heaps. Since they have no subtree, we don't need to deal with those nodes. They are highlighted in green in this next picture. Our algorithm therefore starts at the first non-leaf node from the bottom. This node is at index (n-2)/2 where n is the total number of values in our heap.
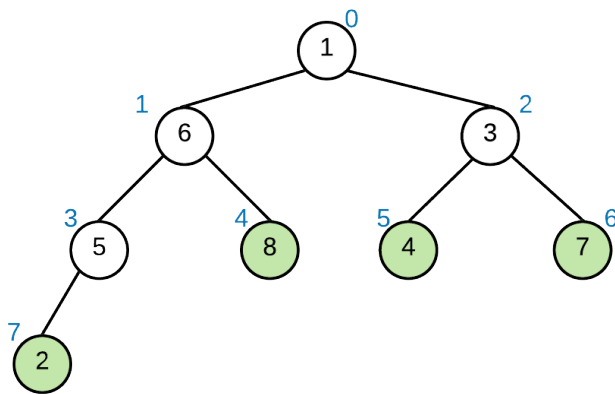
1	6	3	5	8	4	7	2
0	1	2	3	4	5	6	7

The heapify function takes the **index** of the root of the heapify routine (ie we know that nodes children are heaps, and we are looking at it from that node down.

heapify(3)

First node to consider is the node with 5 (index 3). the left subtree is lower priority and the right subtree doesn't exist so we do nothing.
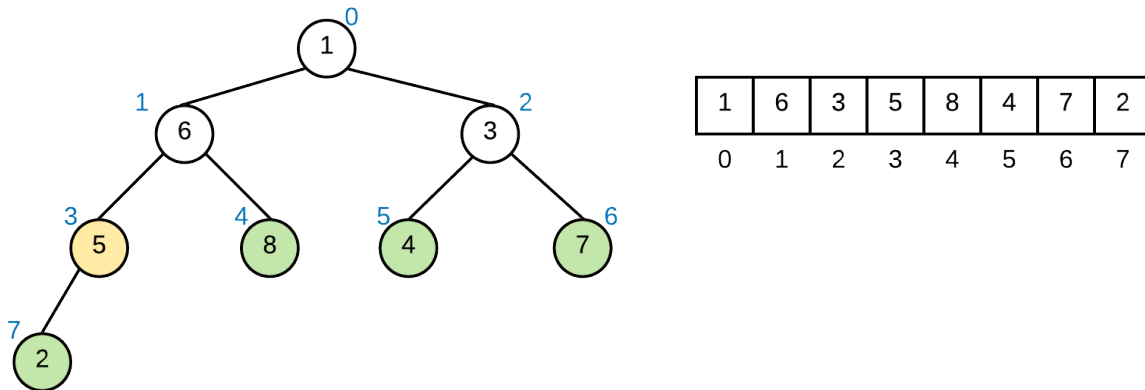
heapify(2)

Second node to consider is 3 (index 2). both 4 and 7 are bigger than 3 and thus have higher priority. However, 7 is higher priority than 4, so we swap these two values. 3 is now in a leaf node and thus we can stop.
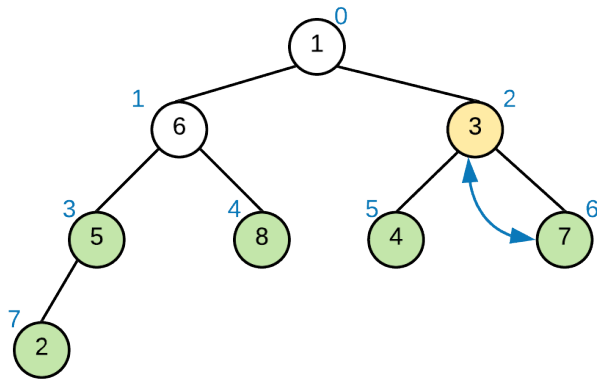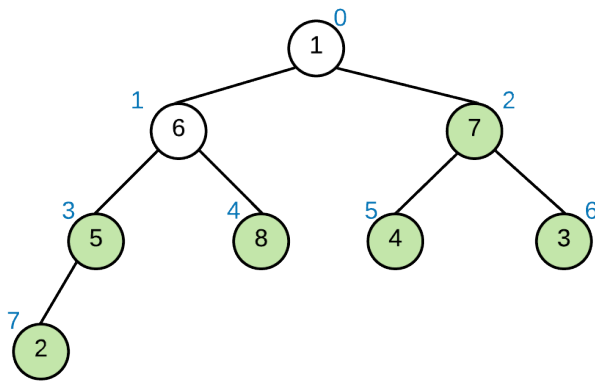
heapify(1)

Next consider the node with 6. 6 is higher priority than 5 but not higher priority than 8, so we swap them. 6 is now in a leaf node and thus we can stop.
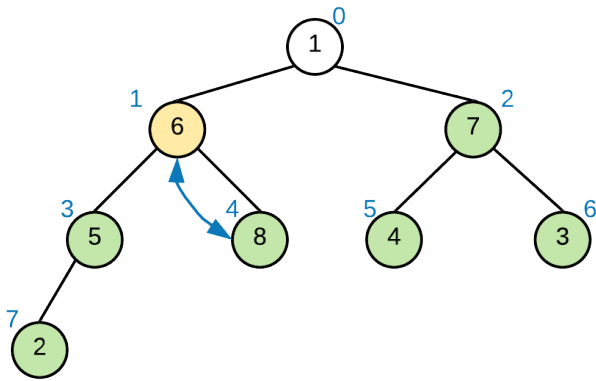
1	6	7	5	8	4	3	2
0	1	2	3	4	5	6	7
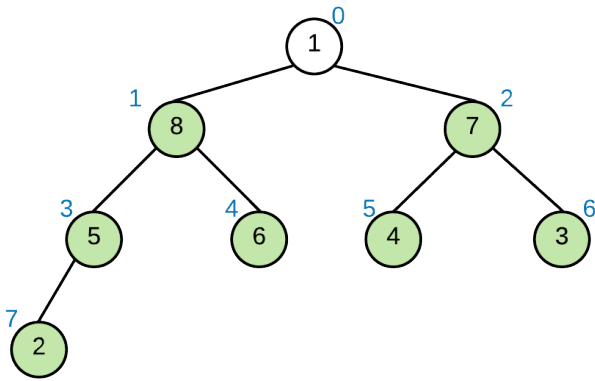
1	8	7	5	6	4	3	2
0	1	2	3	4	5	6	7

heapify(0)

Lastly we consider the node 1. both 8 and 7 have higher priority than 1, but 8 is higher priority than 7 and thus we swap 8 and 1 first.
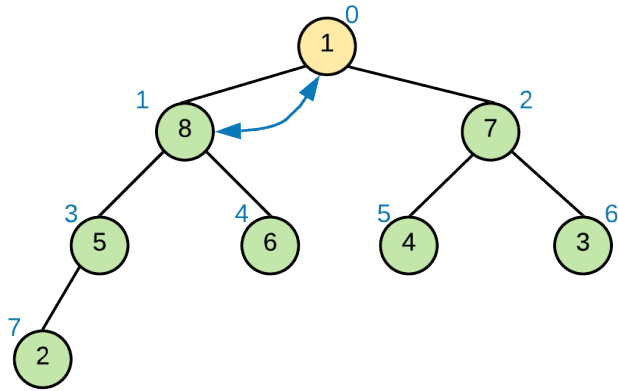
1	8	7	5	6	4	3	2
0	1	2	3	4	5	6	7

At this point we need to heapify the subtree that now has 1 as it's root (index 1). We swap it with the higher priority child (6 in this case). Notice that because we won't need to do anything to the right subtree with 7 as root, as that subtree was not modified.
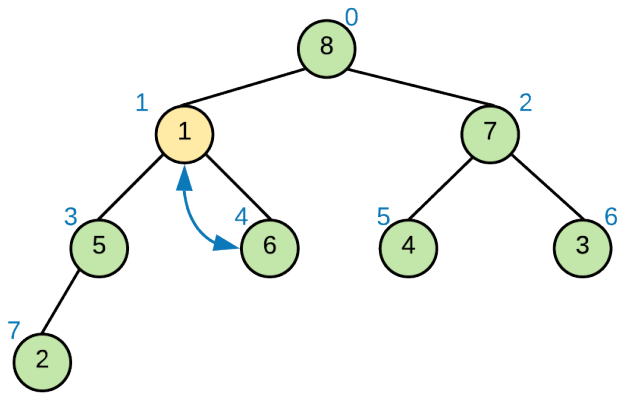
8	1	7	5	6	4	3	2
0	1	2	3	4	5	6	7

After the final swap we have a proper max heap
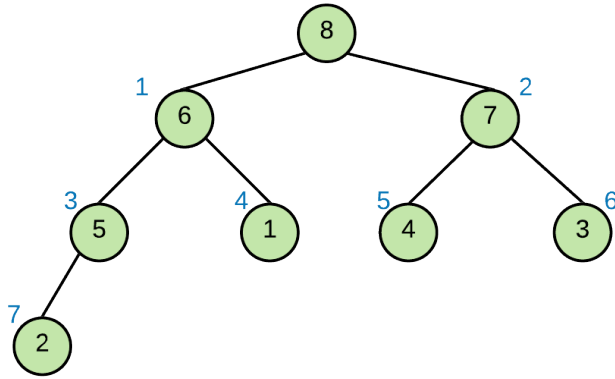
8	6	7	5	1	4	3	2
0	1	2	3	4	5	6	7

Once the heap is built, we continuously remove the highest priority value from the heap and place it at the back of the array.

Binary Search Trees

Binary search trees (BST) are binary trees where values are placed in a way that supports efficient searching. In a BST, all values in the left subtree value in current node < all values in the right subtree. This rule must hold for EVERY subtree, ie every subtree must be a binary search tree if the whole tree is to be a binary tree. The following is a binary search tree:
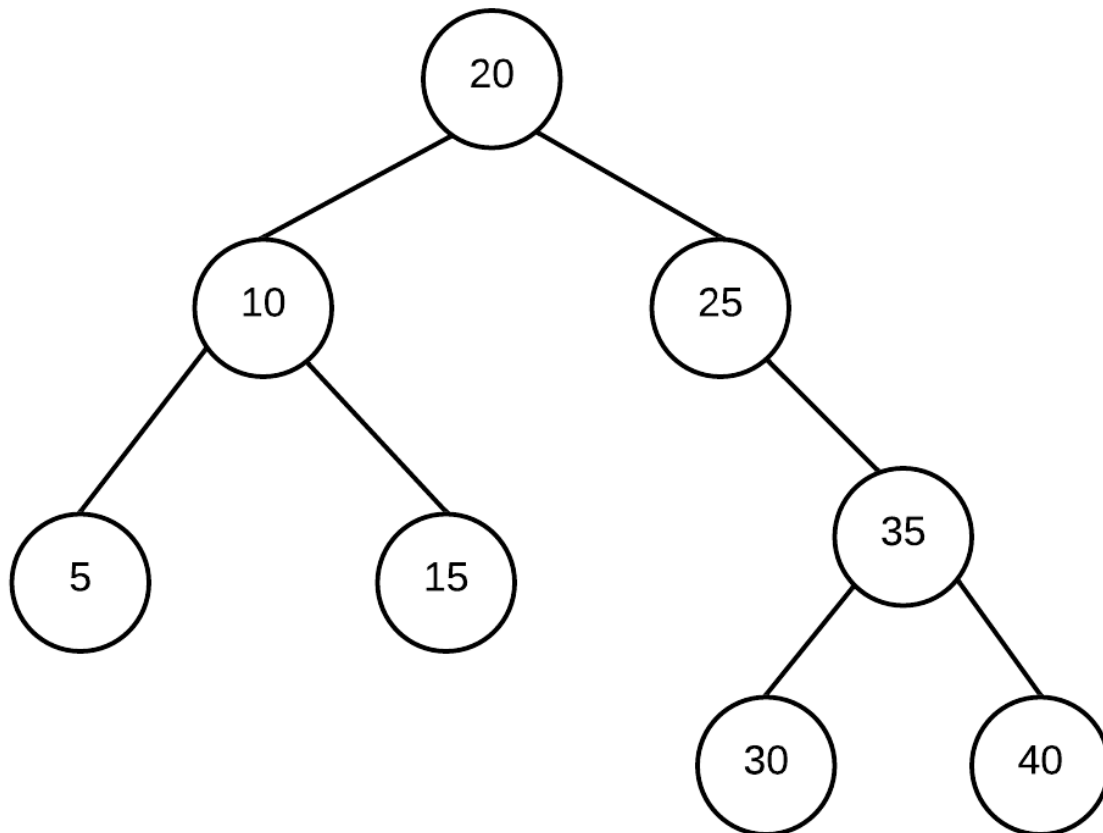
The following is NOT a binary search tree as the values are not correctly ordered:
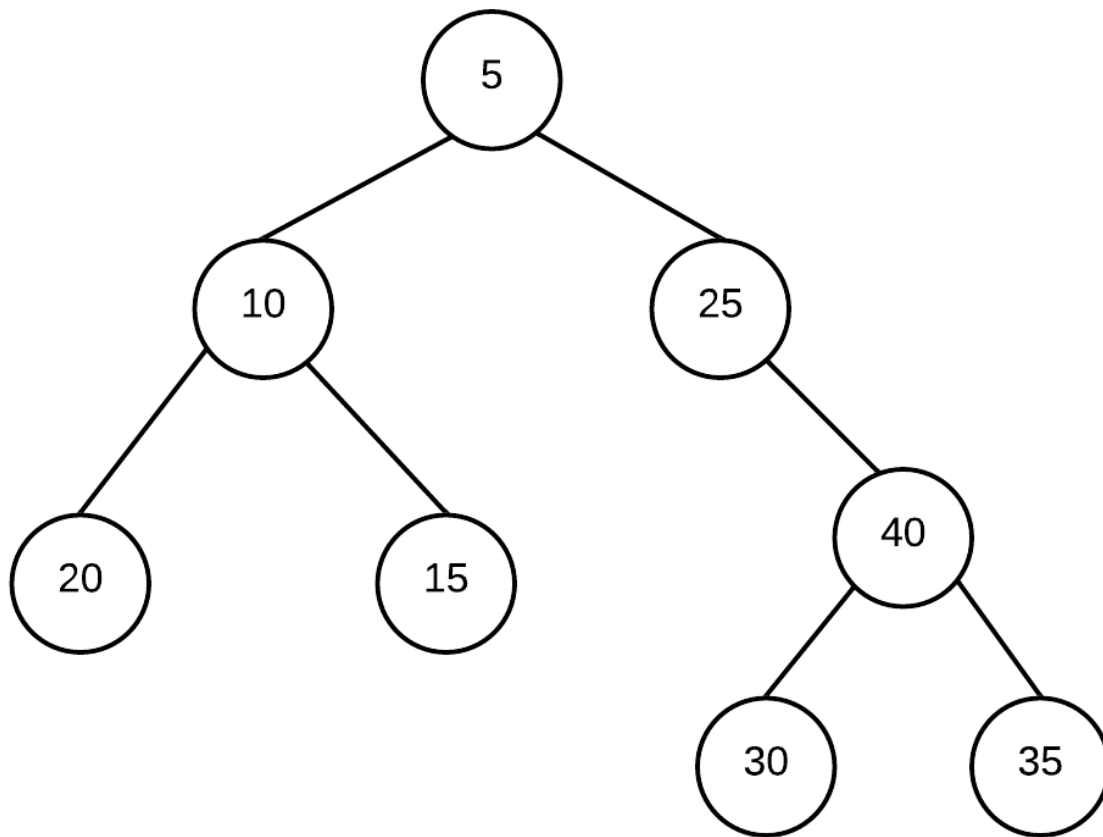
A binary search tree allows us to quickly perform a search on a linking structure (under certain conditions which we will explore later). To find a value, we simply start at the root and look at the value. If our key is less than the value, search left subtree. If key is greater than value search right subtree. It provides a way for us to do a "binary search" on a linked structure which is not possible with a linear linked list. During the search, we will never have to search the subtrees we eliminate in the search process... thus at worst, searching for a value in a binary search tree is equivalent to going through all the nodes from the root to the furthest leaf in the tree.

Insertion

To insert into a binary search tree, we must maintain the nodes in sorted order. There is only one place an item can go. Example Insert the numbers 4,2,3,5,1, and 6 in to an initially empty binary search tree in the order listed. Insertions always occur by inserting into the first available empty subtree along the search path for the value:

Insert 4:
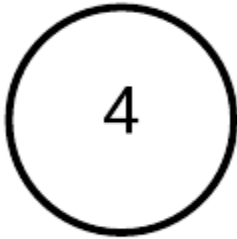
Insert 2: 2 is < 4 so it goes left
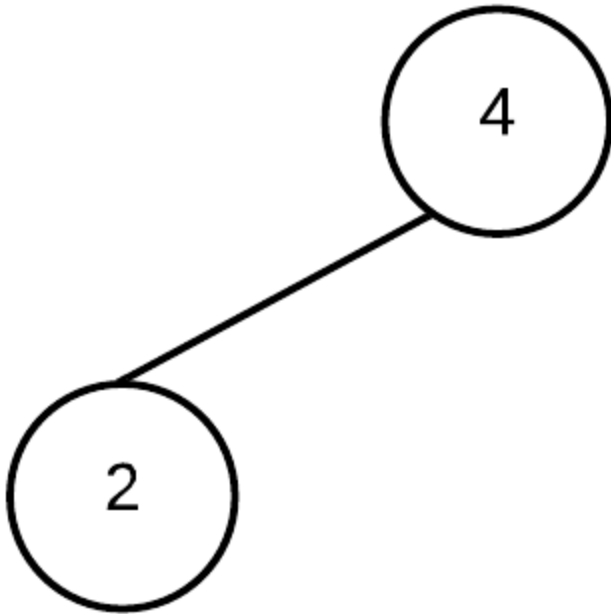
Insert 3: 3 is less than 4 but more than 2 so it goes to left of 4, but right of 2
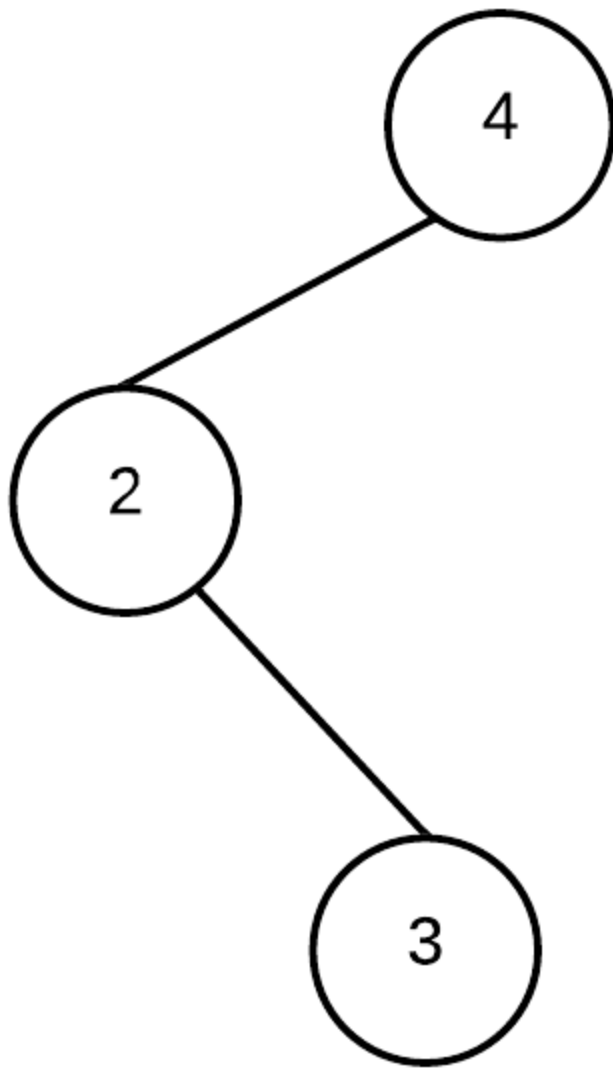
Insert 5: 5 is > 4 so it goes right of 4
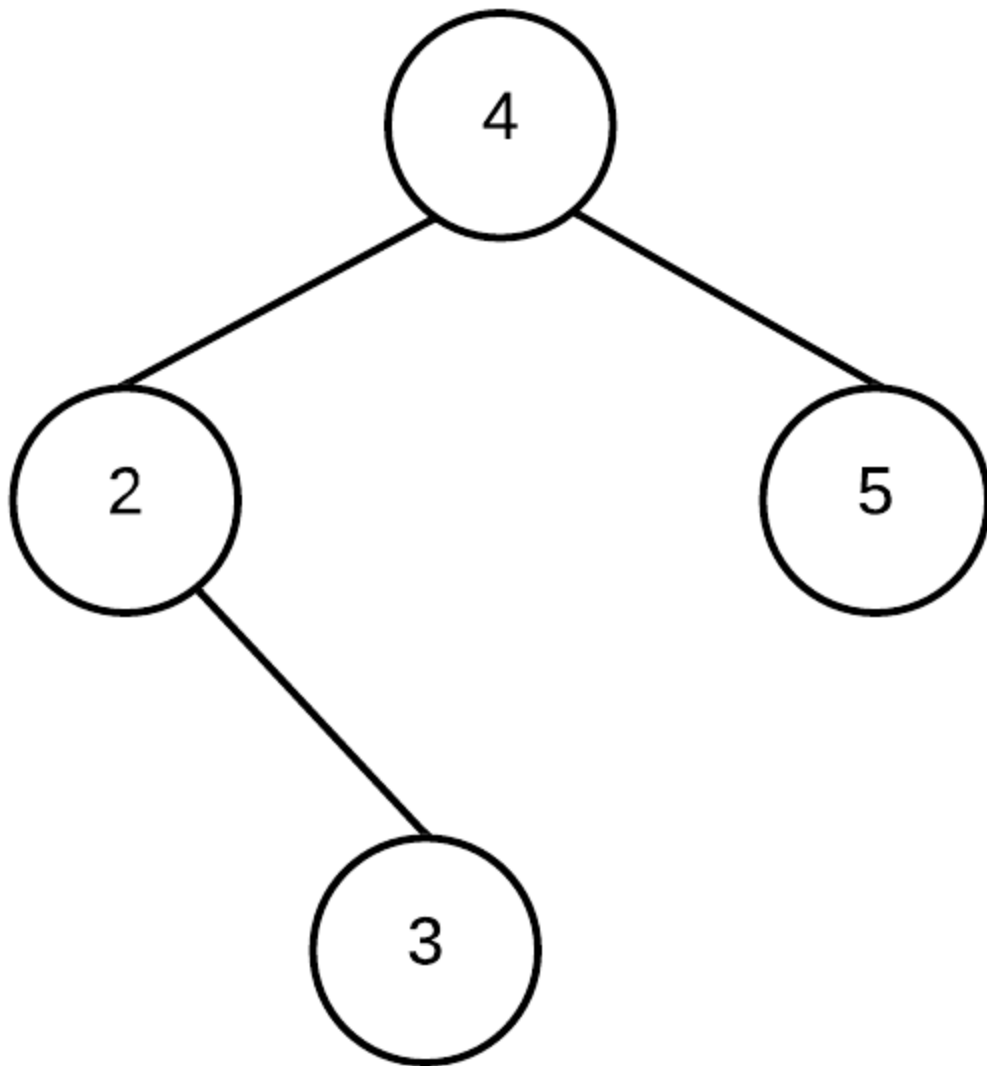
Insert 1: 1 is < 4 so it goes to left of 4. 1 is also < 2 so it goes to left of 2
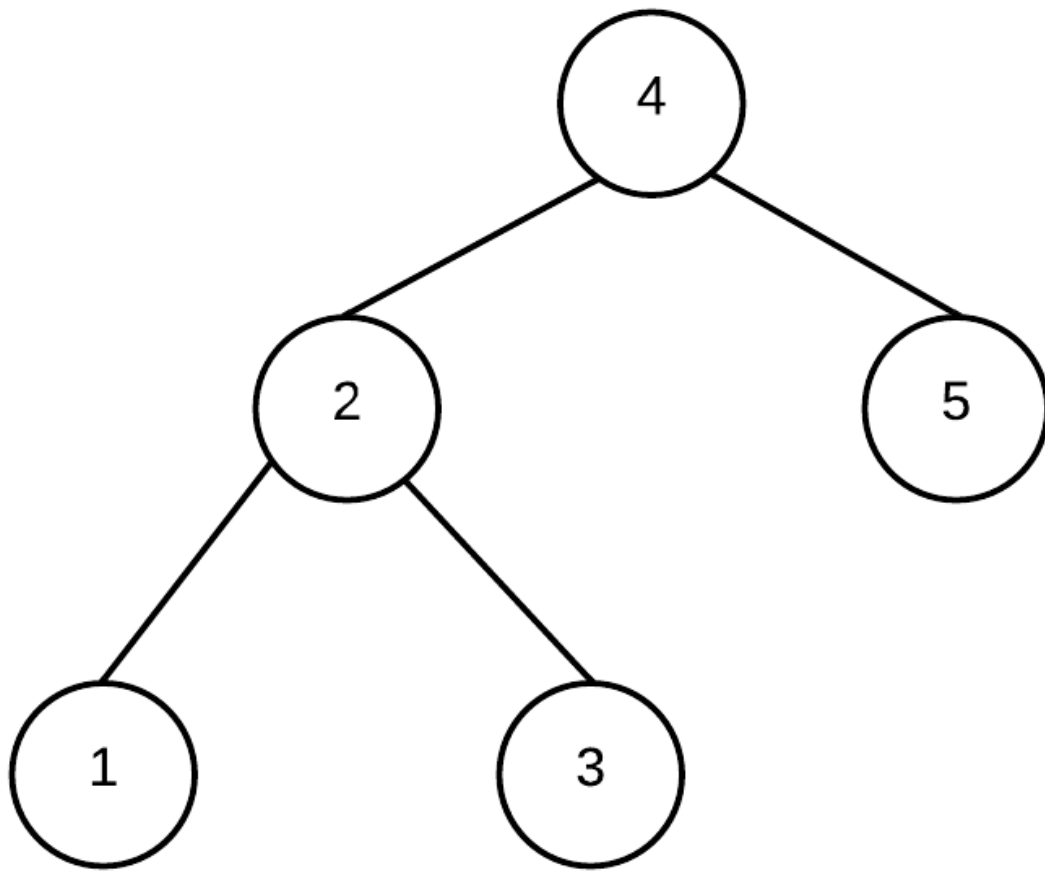
Insert 6: 6 is > 4 so it goes to right of 4. 6 is also > 5 so it goes to right of 5
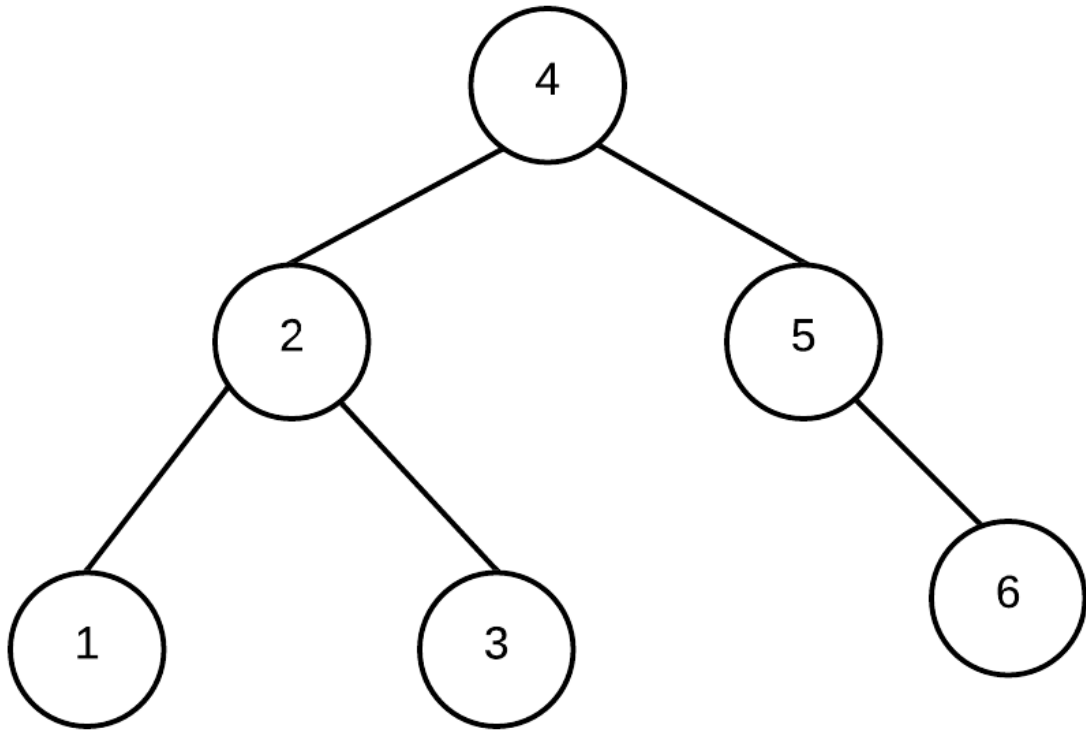
Removal

In order to delete a node, we must be sure to link up the subtree(s) of the node properly. Let us consider the following situations.

Suppose we start with the following binary search tree:
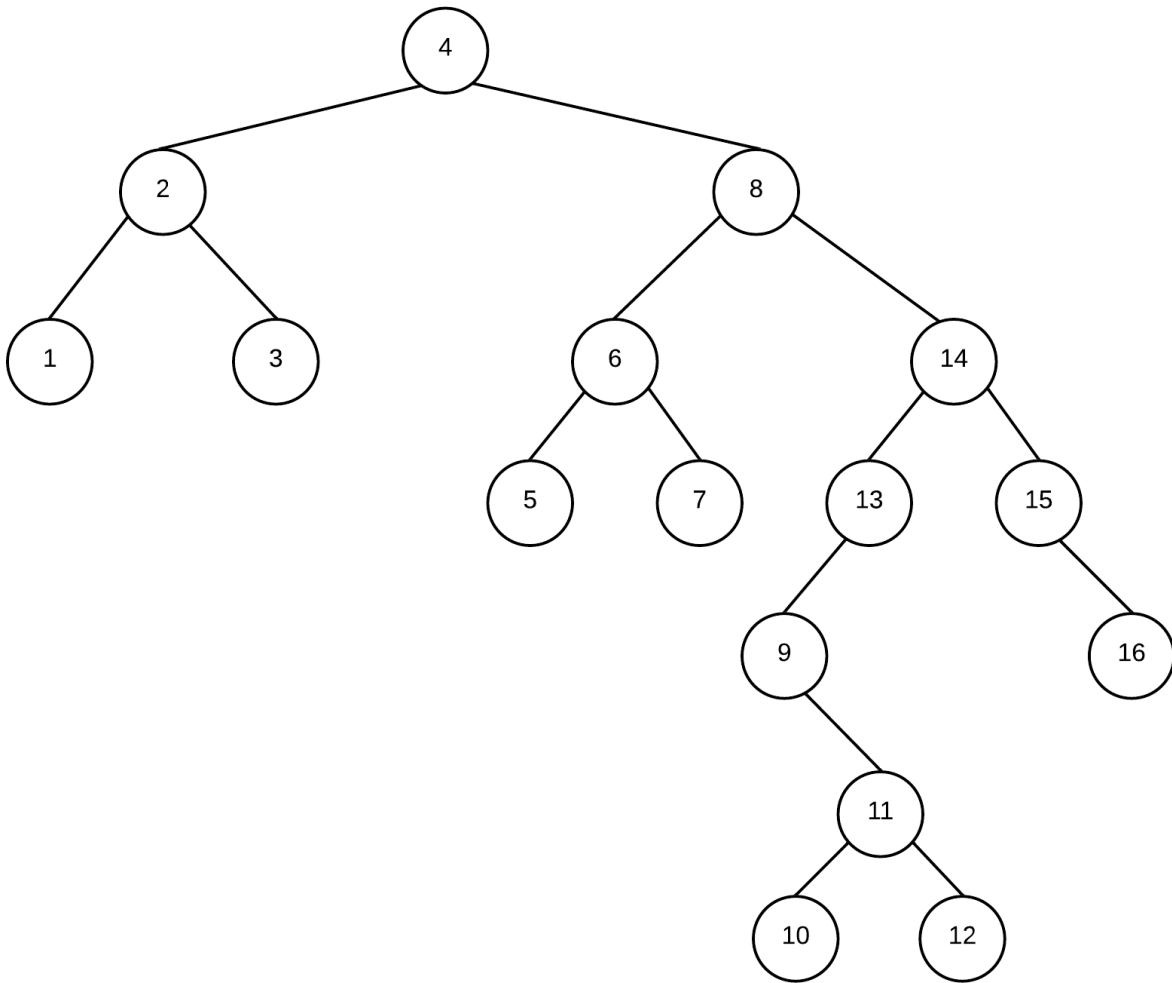
Suppose we were to remove 7 from the tree. Removing this node is relatively simple, we simply have to ensure that the pointer that points to it from node 6 is set to nullptr and delete the node:
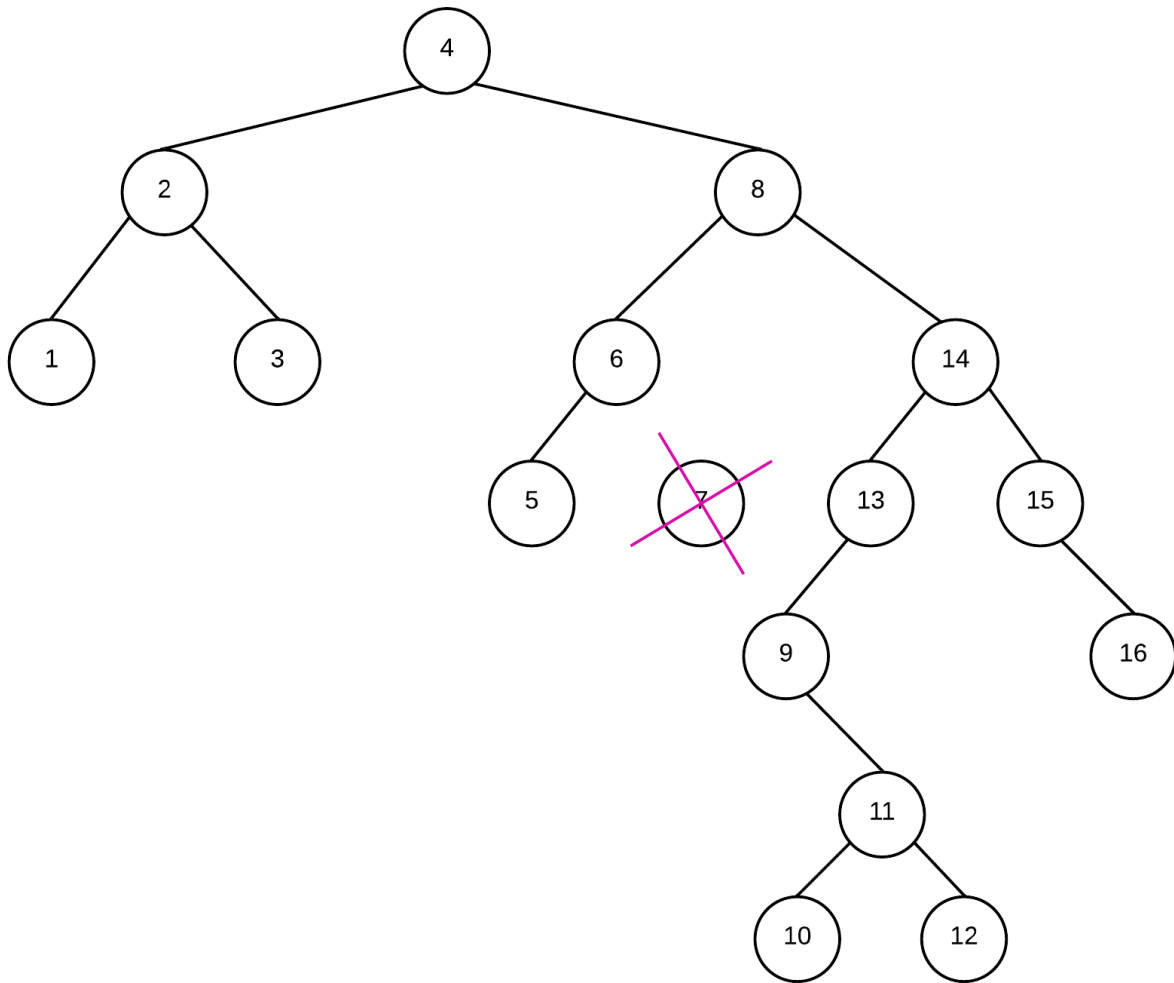
Now, Lets remove the node 6 which has only a left child but no right child. This is also easy. all we need to do is make the pointer from the parent node point to the left child.
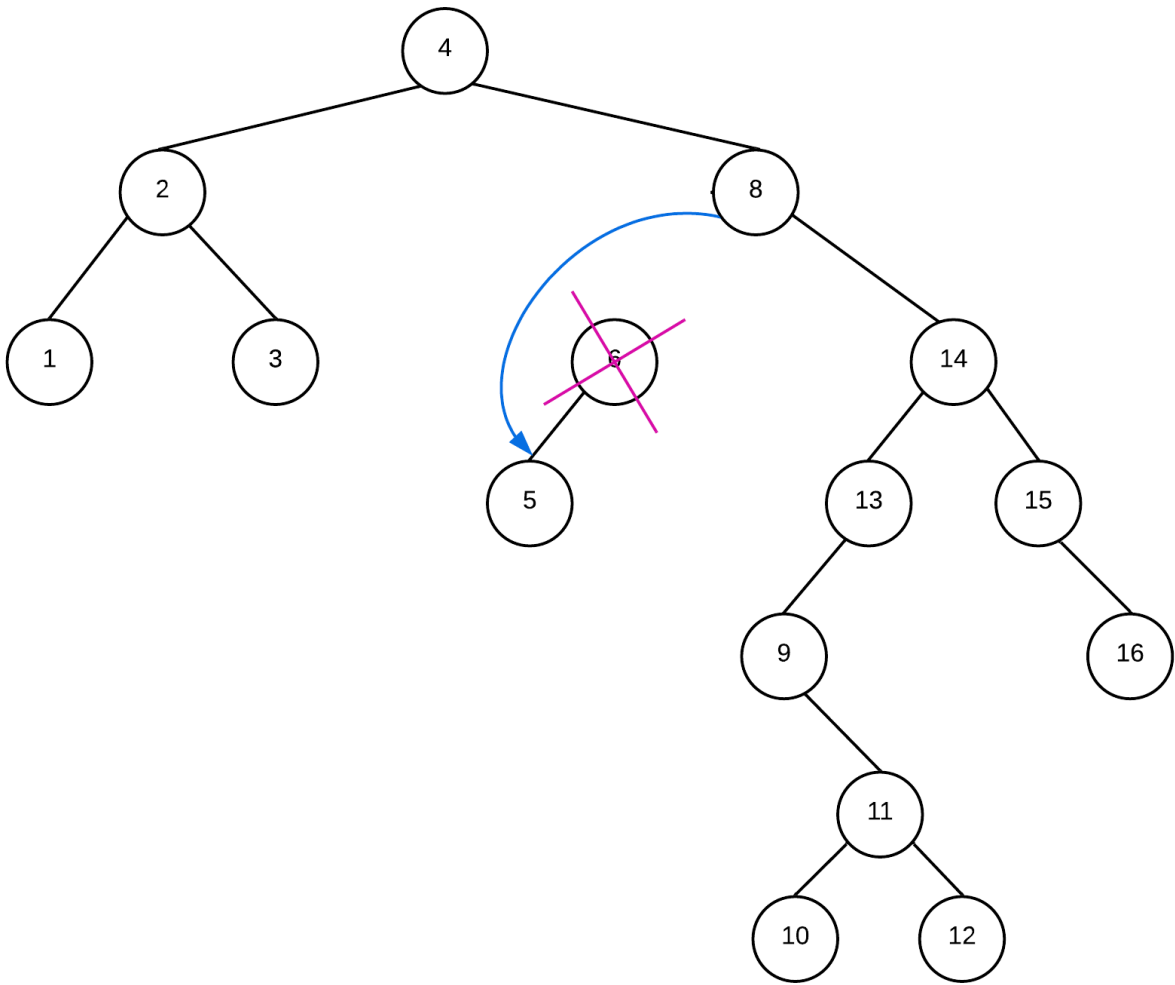
Thus our tree is now:
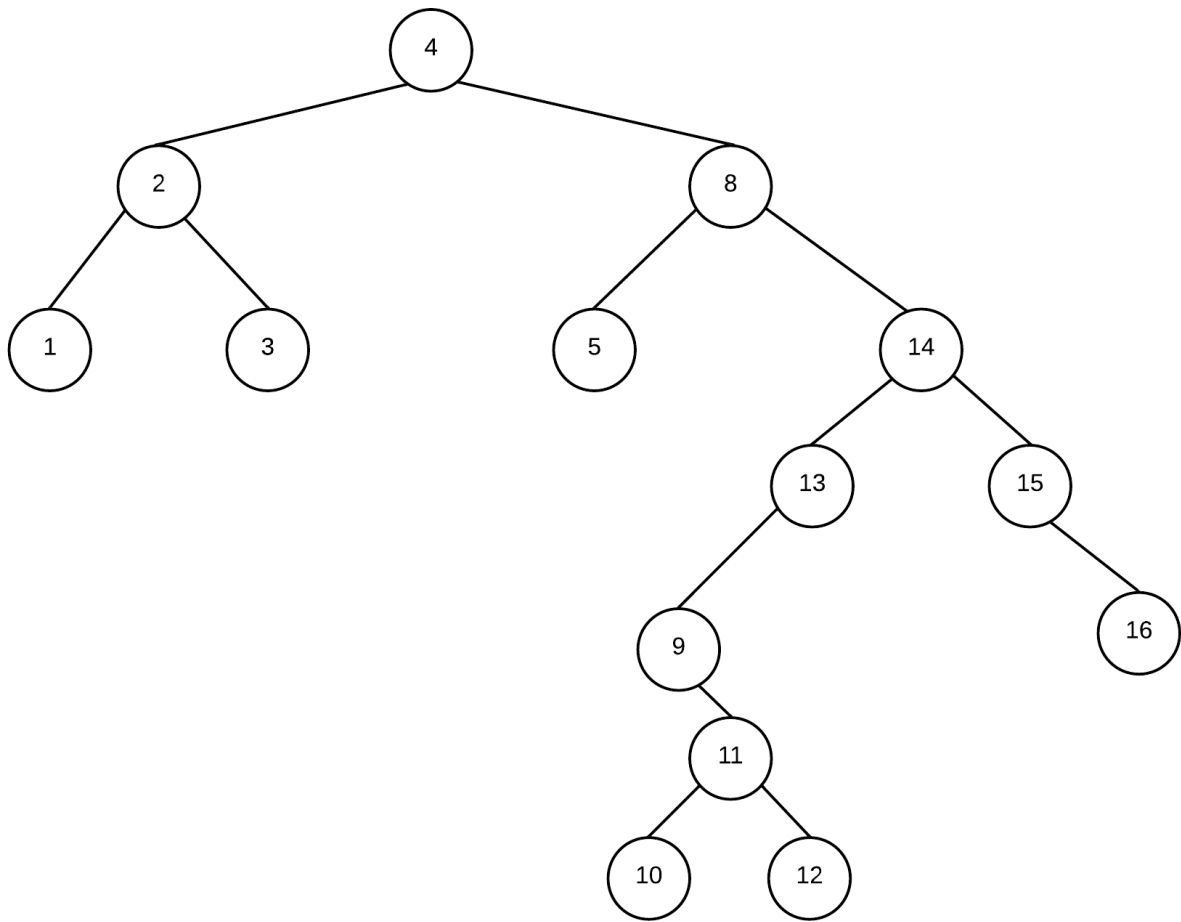
Now suppose we wanted to remove a node like 8. We cannot simply make 4 point to 8's child as there are 2 children. While it is possible to do something like make parent point to right child then attach left child to the left most subtree of the right right child, doing so would cause the tree to be bigger and is not a good solution.

Instead, we should find the inorder successor (next biggest descendent) to 8 and promote it so that it replaces 8.

The inorder successor can be found by going to the right child of 8 then going as far left as possible. In this case, 9 is the inorder sucessor to 8:
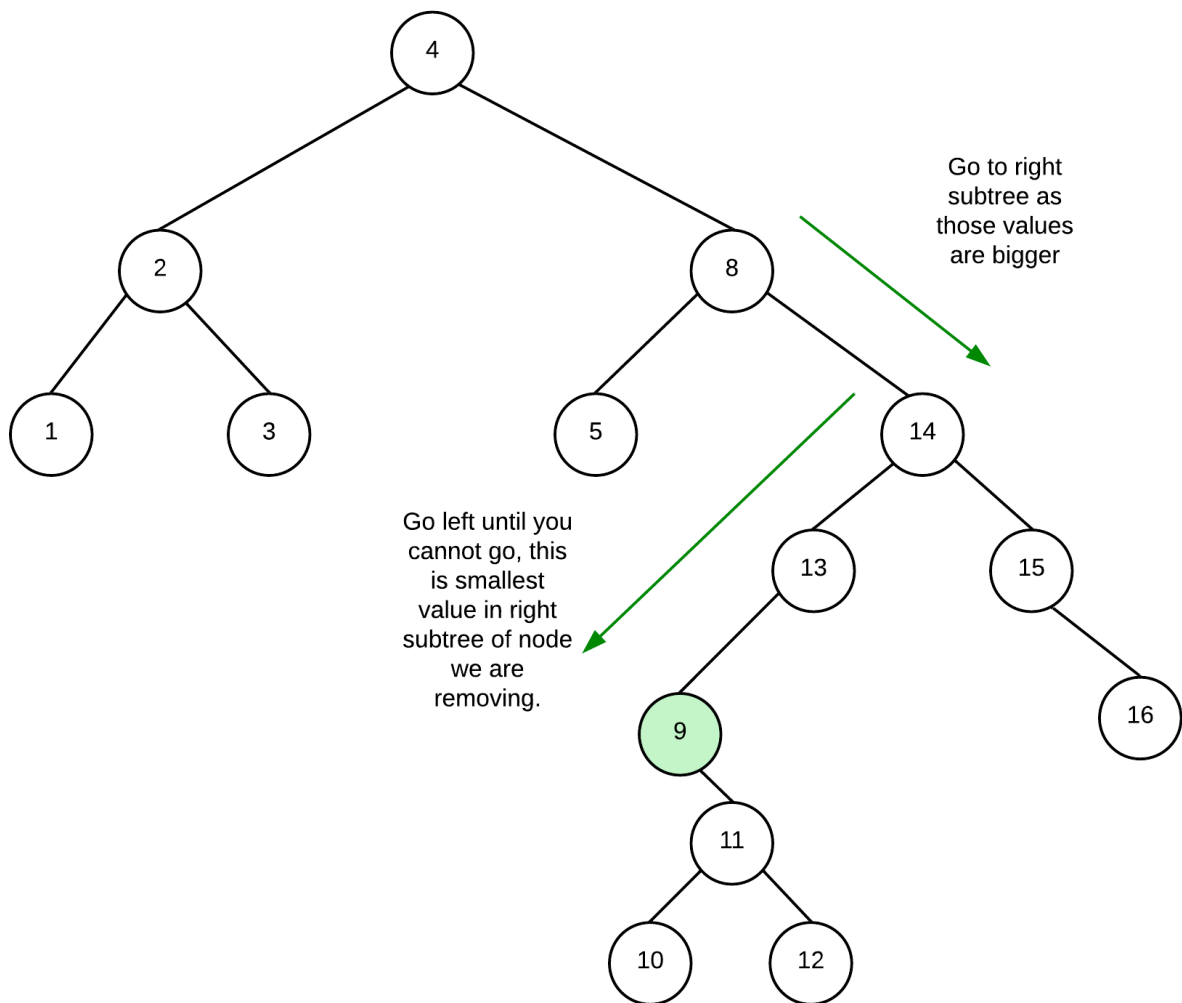
The inorder successor will either be a leaf node or it will have a right child only. It will never have a left child because we found it by going as far left as possible.

Once found, we can promote the inorder successor to take over the place of the node we are removing. The parent of inorder successor must link to the right child of the inorder succesor.

REPLACE 8 WITH ITS' INORDER SUCCESSOR
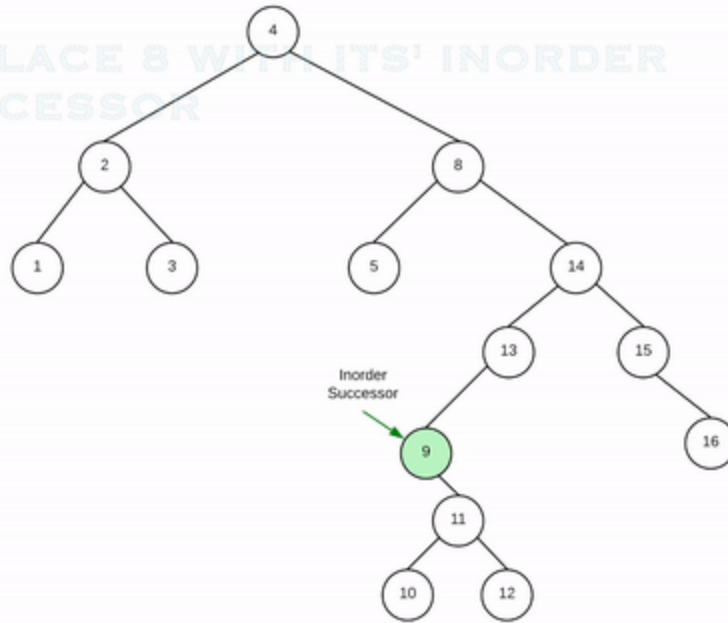
In the end we have:
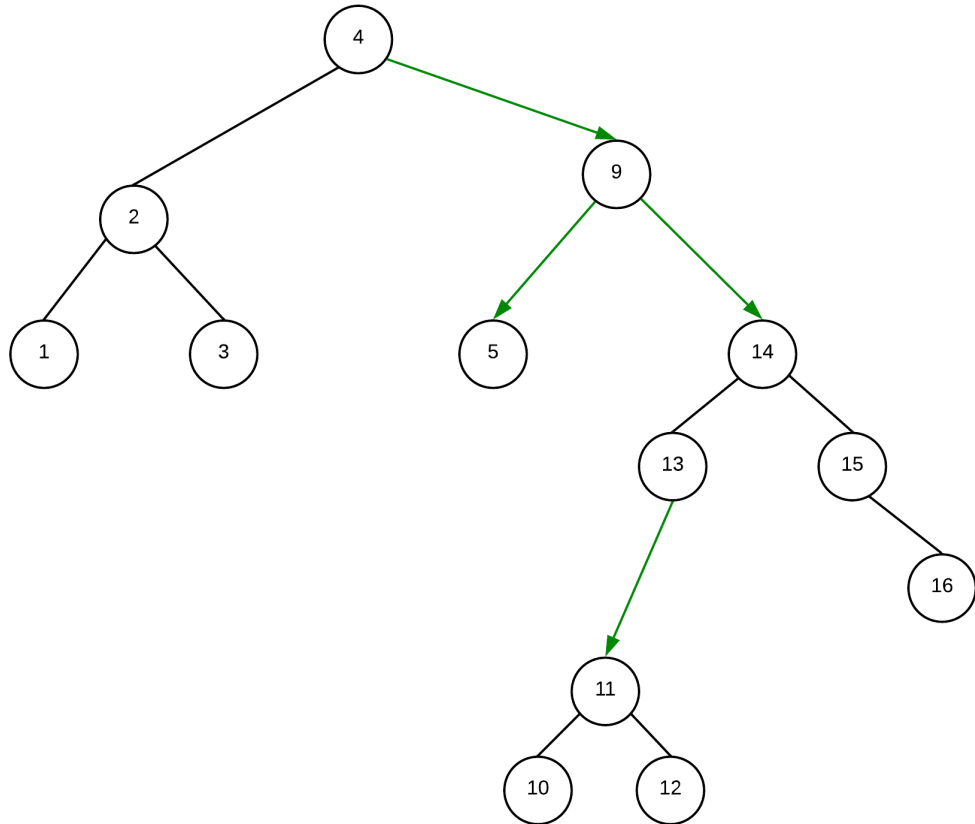
Traversals

There are a number of functions that can be written for a tree that involves iterating through all nodes of the tree exactly once. As it goes through each node exactly 1 time, the runtime should not exceed O(n)

These include functions such as print, copy, even the code for destroying the structure is a type of traversal.

Traversals can be done either depth first (follow a branch as far as it will go before backtracking to take another) or breadfirst, go through all nodes at one level before going to the next.

Depth First Traversals

There are generally three ordering methods for depth first traversals. They are:

- preorder
- inorder
- postorder

In each of the following section, we will use this tree to describe the order that the nodes are "visited". A visit to the node, processes that node in some way. It could be as simple as printing the value of the node:
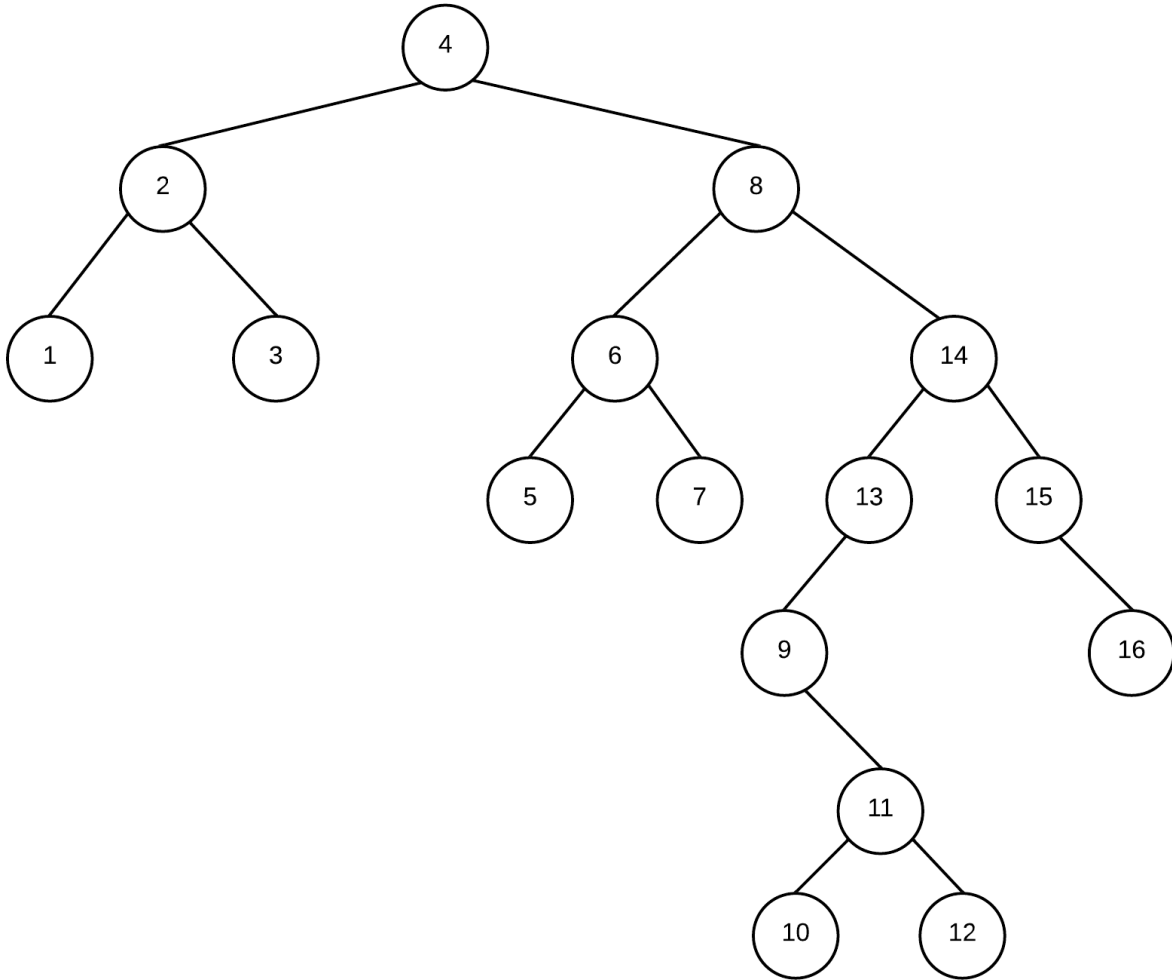
Preorder traversals

- visit a node
- visit its left subtree
- visit its right subtree

4, 2, 1, 3, 8, 6, 5, 7, 14, 13, 9, 11, 10, 12, 15, 16

Inorder traversals:

- visit its left subtree
- visit a node

- visit its right subtree

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

Notice that this type of traversal results in values being listed in its sorted order

Postorder traversals:

- visit its left subtree
- visit its right subtree
- visit a node

1, 3, 2, 5, 7, 6, 10, 12, 11, 9, 13, 16, 15, 14, 8, 4

This is the type of traversal you would use to destroy a tree.

Breadth-First Traversal

A breadfirst traversal involves going through all nodes starting at the root, then all its children then all of its children's children, etc. In otherwords we go level by level.

Given the following tree:
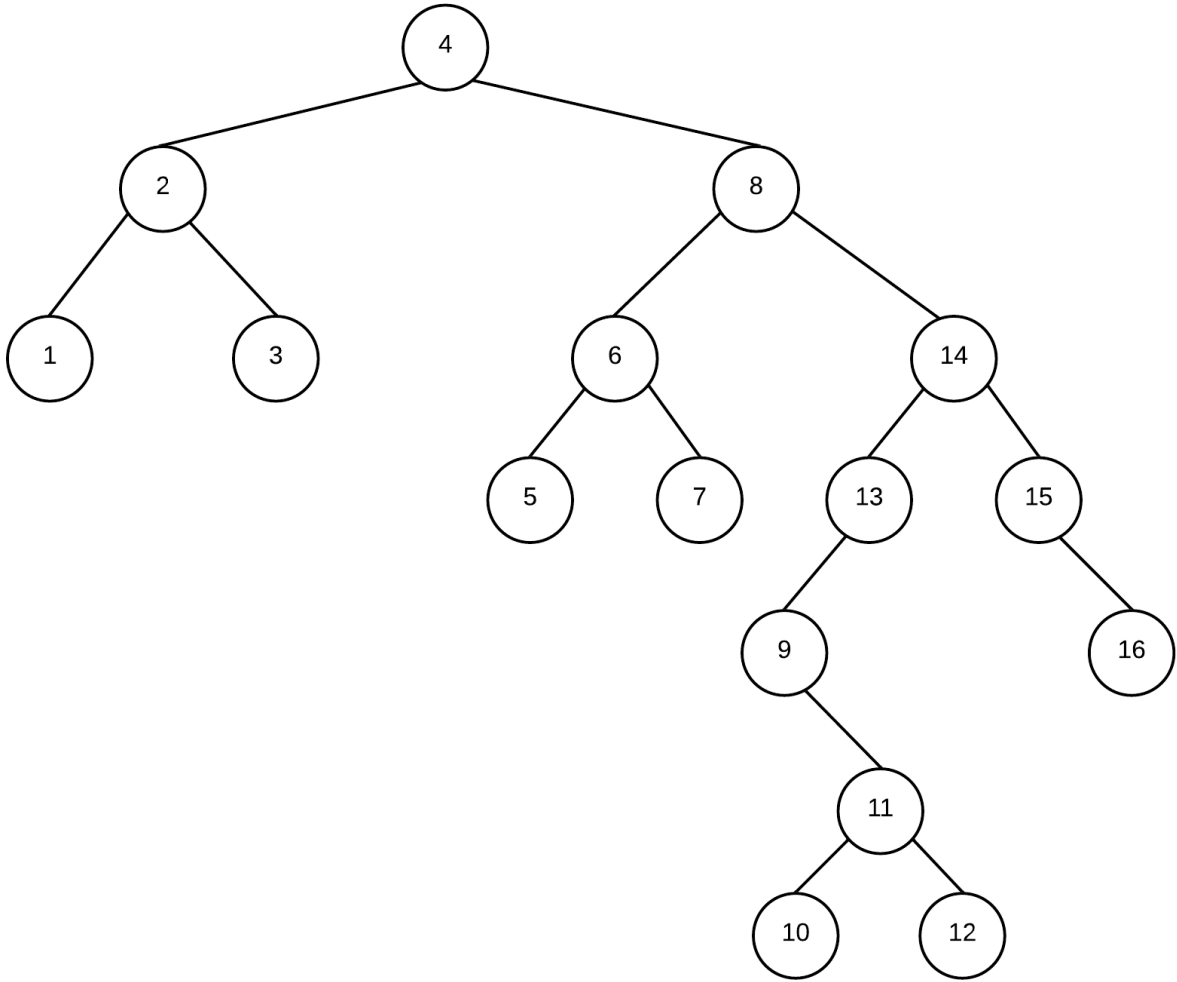
A breadthfirst traversal would result in:

4, 2, 8, 1, 3, 6, 14, 5, 7, 13, 15, 9, 16, 11, 10, 12

BST Implemenation

To implement a binary search tree, we are going to borrow some concepts from linked lists as there are some parts that are very similar. In these notes we will look a few of the functions and leave the rest as an exercise.

Similar to a linked list, A binary search tree is made up of nodes. Each node can have a left or right child, both of which could be empty trees. Empty trees are represented as nullptrs. The binary search tree object itself only stores a single pointer that points at the root of the entire tree. The data stored within the nodes must be of some type that is comparable. We will thus begin our binary search tree class declaration in a similar manner to that of a linked list. The code for each of the functions will be filled in later in this chapter.

Python   C++

```python
class BST:
    class Node:
        # Node's init function
        def __init__(self,data=None,left=None,right=None):
            self.data = data
            self.left = left
            self.right = right

    #BST's init function
    def __init__(self):
        self.root = None

    def insert(self, data):
        ...
```

```cpp
template <typename T>
class BST{
    struct Node{
        T data_;
        Node* left_;
        Node* right_;
        Node(const T& data, Node* left=nullptr, Node*
right=nullptr){
            data_=data;
            left_=left;
            right_=right;
        }
    };
    //single data member pointing to root of tree
    Node* root_;
public:
    BST(){...}
    void insert(const T& data){...}
    bool search(const T& data) const {...}
    void breadthFirstPrint() const {...}
    void inOrderPrint() const {...}
    void preOrderPrint() const {...}

    ~BST(){...}
};
```

If you rename the data members above you actually see that its pretty similar to that of a doubly linked list... The key to the operations of a BST lies not in what data is declared, but rather how we organize the nodes. The next 2 sections of the notes we will look at the implementation of the functions listed above. In some cases a function may be written both iteratively and recursively and both versions will be looked at

Constructor (C++)

When we create our tree, we are going to start with an empty tree. Thus, our constructor simply needs to initialize the data member to nullptr.

```cpp
template <typename T>
class BST{
    struct Node{
        T data_;
        Node* left_;
        Node* right_;
        Node(const T& data, Node* left=nullptr, Node* right=nullptr){
            data_=data;
            left_=left;
            right_=right;
        }
    };
    //single data member pointing to root of tree
    Node* root_;
public:
    BST(){
        root_=nullptr;
    }
...
};
```

Iterative Methods

This section looks at the functions that are implemented iteratively (or the iterative version of the functions)

Insert - Iterative version

This function will insert data into the tree. There are two ways to implement this function, either iteratively or recursively. We will start by looking at the iterative solution. In this situation, we begin by taking care of the empty tree situation. If the tree is empty we simply create a node and make root_ point to that only node. Otherwise, we need to go down our tree until we find the place where we must do the insertion and then create the node.

Python   C++

```python
class BST:
    class Node:
        # Node's init function
        def __init__(self, data=None, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    # BST's init function
    def __init__(self):
        self.root = None

    def insert(self, data):
        if self.root is None:
```

```cpp
template <typename T>
class BST{
    struct Node{
        T data_;
        Node* left_;
        Node* right_;
        Node(const T& data, Node* left=nullptr, Node*
right=nullptr){
            data_=data;
            left_=left;
            right_=right;
        }
    };
    //single data member pointing to root of tree
    Node* root_;
public:
...
    void insert(const T& data){
        if(root_==nullptr){
            root_=new Node(data);
        }
        else{
            bool isInserted=false;   //set to true when once we
insert the node
            Node* curr=root_;        //used to iterate through
nodes
            while(!isInserted){
                if(data < curr->data_){
                    //data belongs in left subtree because it is
                    //smaller than current node
                    if(curr->left_){
                        //there is a node to the left so go left
                        curr=curr->left_;
                    }
                    else{
                        //there isn't a node to left
```

Search - Iterative version

The key operation that is supported by a binary search tree is search. For our purposes we will simply look at finding out whether or not a value is in the tree or not. The search operation should never look at the entire tree. The whole point of the binary search tree is to make this operation fast. We should search it so that we can eliminate a portion of the tree with every search operation.

To do this we start at the root and compare that node's data against what we want. If it matches, we have found it. If not, we go either left or right depending on how data relates to the current node. If at any point we have an empty tree (ie the pointer we are using for iterating through the tree becomes nullptr) we stop the search and return false. If we find a node that matches we stop and return true.

Python   C++

```python
class BST:
    class Node:
        # Node's init function
        def __init__(self, data=None, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    # BST's init function
    def __init__(self):
        self.root = None

    def search(self, data):
```

```cpp
template <typename T>
class BST{
    struct Node{
        T data_;
        Node* left_;
        Node* right_;
        Node(const T& data, Node* left=nullptr, Node*
right=nullptr){
            data_=data;
            left_=left;
            right_=right;
        }
    };
    //single data member pointing to root of tree
    Node* root_;
public:
    bool search(const T& data) const {
        Node* curr=root_;      //used  to iterate through tree
        bool found=false;     //true if we find it false if we
haven't yet

        //loop until we either find it or we have no more tree
        while(!found && curr){
            if(data==curr->data_){
                found=true;
            }
            else if(data < curr->data_){
                curr=curr->left_;
            }
            else{
                curr=curr->right_;
            }
        }
        return  found;
    }
};
```

Breadth First Print

Writing a breadth-first traversal involves using the queue data structure to order what nodes to deal with next. You want to deal with the nodes from top to bottom left to right, and thus you use the queue to order the nodes. Here is an example of how we will do this.

We begin by declaring a queue (initially empty)

The Queue (stores pointers to indicated nodes)
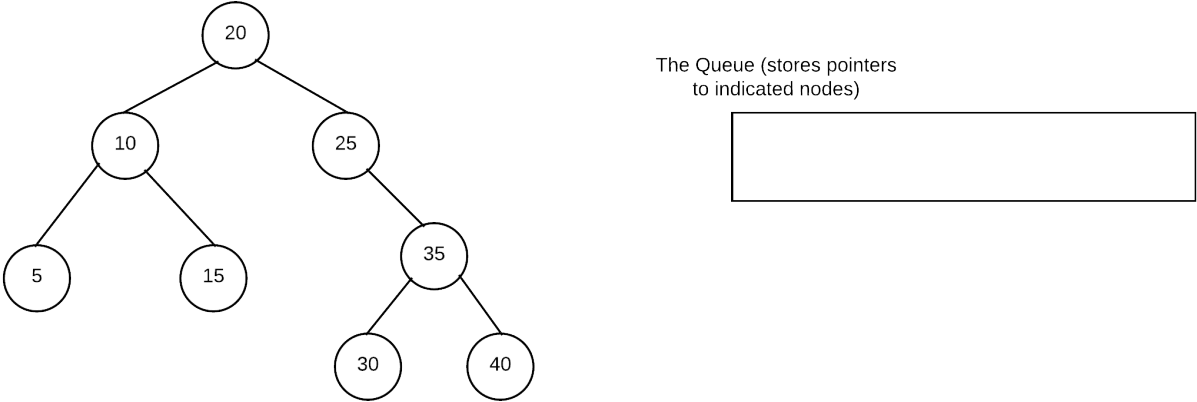
Prime the Queue

We start prime the queue by putting the root into the queue. In this example, we always check to ensure no nullptrs are ever added to the queue. Alternatively we allow the addition of nullptrs and decide how to deal with them when we dequeue.
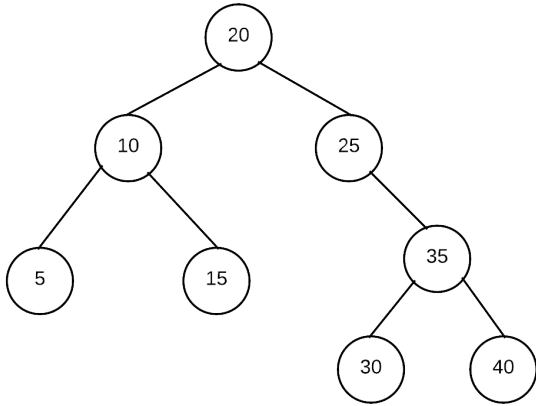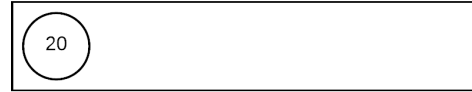
Dequeue front of node and process it by adding its non-nullptr children into the queue, print the node
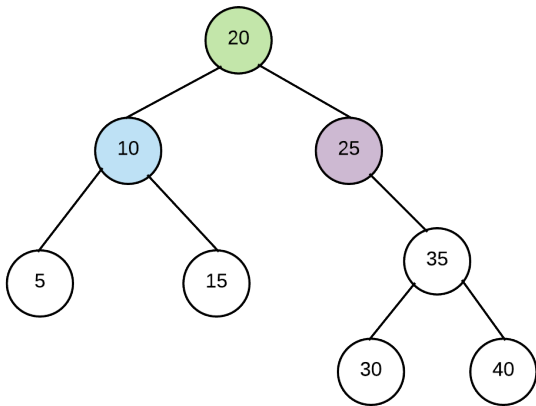
Current node

Output: **20**

Continue by dequeuing front node and processing it in same manner
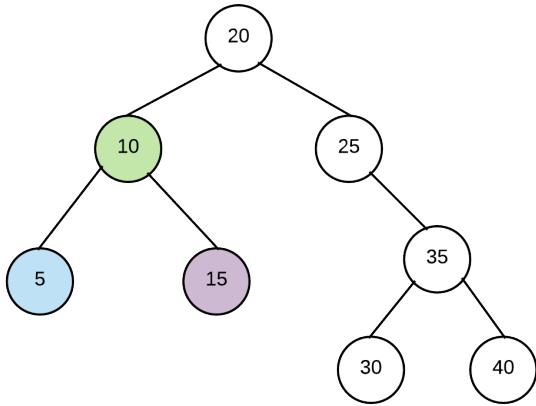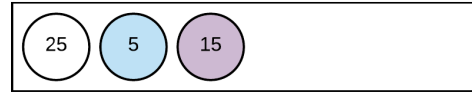
Output: 20 **10**

Repeat again as 25 only has a right child only it is added
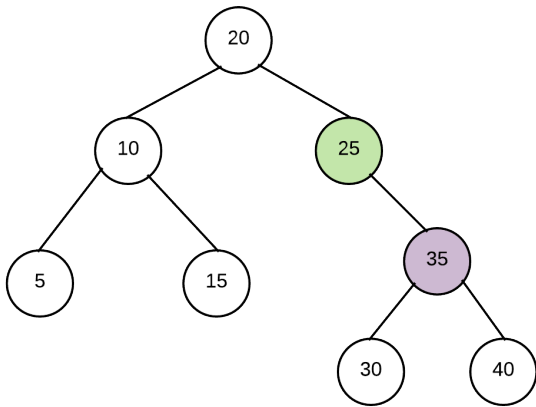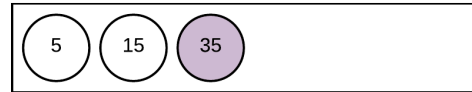
Output: 20 10 **25**

Repeat once again with 5 which has no children thus nothing is added to queue

The Queue (stores pointers
to indicated nodes)

| 15 | 35 |
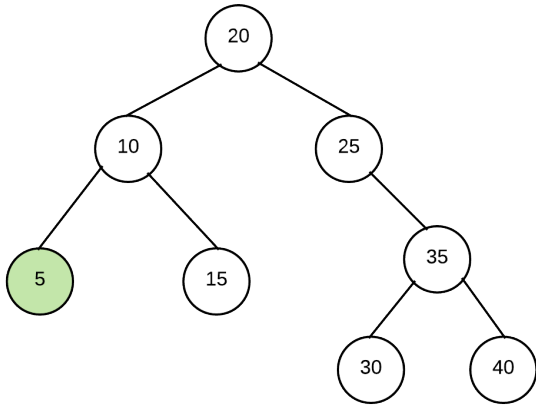
front of queue

5 Current node

Output: 20 10 25 **5**

Repeat again with 15 (also no children)

The Queue (stores pointers
to indicated nodes)

| 35 |

front of queue

15 Current node
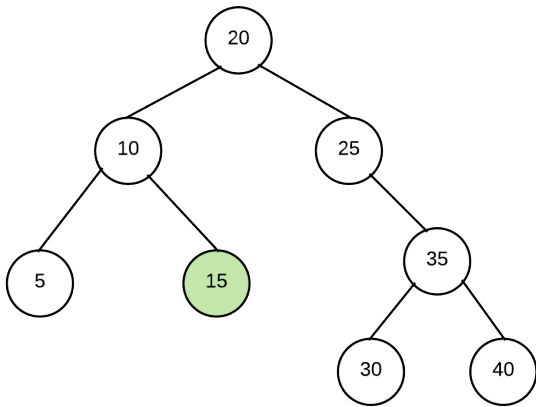
Output: 20 10 25 5 **15**

Repeat with 35 and add its children

The Queue (stores pointers
to indicated nodes)

front of queue

35 Current node
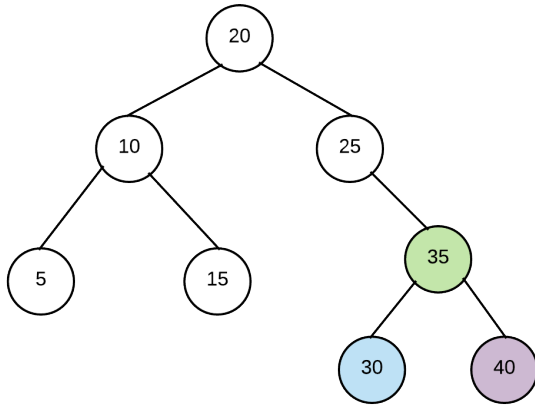
Output: 20 10 25 5 15 **35**

Continue by removing 30

The Queue (stores pointers
to indicated nodes)

front of queue

30 Current node
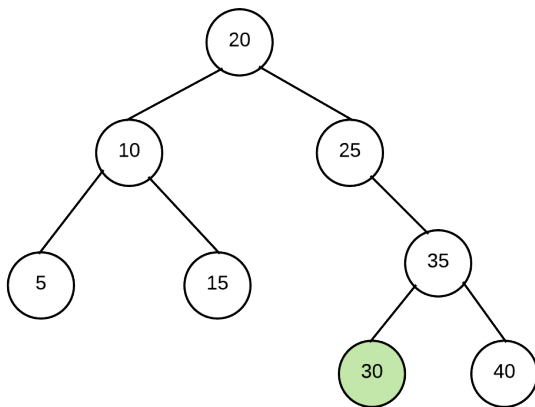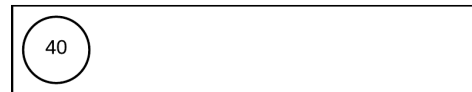
Output: 20 10 25 5 15 35 **30**

And one more time with 40

The Queue (stores pointers
to indicated nodes)

front of queue

40   Current node
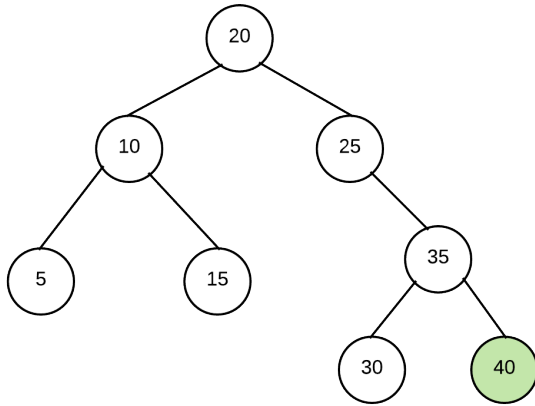
Output: 20 10 25   5   15   35   30 40

At this point queue is empty and thus, we have completed our breadthfirst print of the tree.

In code form, this is how we will write our code (we assume we have a template queue available:

Python   **C++**

```python
import queue

class BST:
    class Node:
        # Node's init function
        def __init__(self, data=None, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    # BST's init function
```

```cpp
template <typename T>
class BST{
    struct Node{
        T data_;
        Node* left_;
        Node* right_;
        Node(const T& data, Node* left=nullptr, Node*
right=nullptr){
            data_=data;
            left_=left;
            right_=right;
        }
    };
    //single data member pointing to root of tree
    Node* root_;
public:
...
    void breadthFirstPrint() const{
        Queue<Node*> theNodes;  //we assume the queue class has
these functions
                                //enqueue(), dequeue(), front(),
isEmpty()
        //prime queue
        if(root_){
            theNodes.enqueue(root_);
        }
        //while we have nodes to deal with
        while(!theNodes.isEmpty()){
            //deal with first node and remove it from queue
            Node* curr=theNodes.front();
            theNodes.dequeue();
            if(curr->left_){
                //if the current node has a left child add it to
queue
                theNodes.enqueue(curr->left_);
            }
            if(curr->right_){
```

Recursive Methods

One of the way to look at a binary search tree is to define the tree recursively. A binary search tree is either empty or consists of a root node who has two children each of which is a binary search tree. This recursive nature makes certain functions easy to write recursively. This portion of the implementation will look at how to write member functions recursively.

For all recursive functions involving a binary search tree, you typically need at least one argument that gives you access to the root of a subtree. The public interfaces for these recursive functions will typically start it off by passing in the root as the first argument.

When writing these functions, we look at the problem in terms of the subtree. Often the base case will involve dealing with an empty subtree (or doing nothing with an empty subtree). This section of the notes will deal with some details about how this can be accomplished and some typical ways of dealing with recursive solutions.

search - recursive function

This is the same function as the iterative version of the search (does the same thing). Only difference is that it is written recursively.

As stated earlier, recursive functions for our trees typically involve looking at it in terms of what to do with a tree (defined as root of the tree).

For the recursive search() function, we will need to write a recursive function and call it from the public search() function. The recursive function will not only have data for the argument but also the root of the subtree where we will be

trying to find the data. Note that we cannot use the data member root because we will lose the tree if we do. We must actually pass in the argument so that it can change in each call without causing the root to be modified. Thus, our recursive function has the following prototype:

```python
def r_search(self, data, subtree)
```

```cpp
bool search(const T& data, const Node* subtree) const
```

The above function returns true if data is in the tree who's root is pointed to by subtree, false otherwise.

As with all recursive cases we always want to start by figuring out the base case of the recursive function

Under what circumstances would it be so easy to find the answer we can confidently return the result immediately? So, given a tree (where all we see is the address of the root node) what situations would allow us to know that we have a result.

In our case there are two base cases:

- the tree is empty. If the tree is empty, the value can't be in the tree, so we can return false immediately.
- the tree is not empty but the value we want is in the root of the tree. If that is the case we don't need to look at the rest of the tree. We already know that its there.

> **⚠ CAUTION**
>
> a tree being empty is usually a base case for recursive functions involving a binary search tree. If this is not a base case you are considering, you should ask why and what would happen if you got an empty tree and whether or not you have made a mistake.

So... what if its not a base case? Well if its not a base case, then we know the following:

- there is a tree with at least a root node
- the data isn't in the root node

Thus if the value exists, its either in the left subtree or the right subtree of the root. As the BST places data so that values in right subtree is bigger than value in current node and value in left is smaller, which subtree we look at depends on how the data compares against the root. If the data is smaller than value in root, then it could only be in left subtree if it there at all. Similarly if it is bigger, it could only be in right subtree. Therefore to find if value is in tree we simply make the recursive call to search() using either subtree's left or subtree's right.

> **ⓘ INFO**
>
> For any function that involves looking for a value in the tree it is never more correct to look at both left and right. The difference will be O(log n) vs O(n)

Python   C++

```
class BST:
```

```cpp
template <typename T>
class BST{
    struct Node{
        T data_;
        Node* left_;
        Node* right_;
        Node(const T& data, Node* left=nullptr, Node*
right=nullptr){
            data_=data;
            left_=left;
            right_=right;
        }
    };
    //single data member pointing to root of tree
    Node* root_;
    //recursive search() function.  This function returns
    //true if data is found in tree with root subtree,
    //false otherwise
    bool search(const T& data, const Node* subtree) const{
        bool rc=false;
        //if it tree is empty, the if is skipped and we return
false
        if(subtree != nullptr){
            if(data == subtree->data_){
                //base case 2: we find it in the root of subtree
                rc=true;
            }
            else if(data < subtree->data_){
                //data is smaller than that stored in root.  If
we find it,
                //it will be in left subtree, so we call search
to see if its
                //there and return the result
                rc=search(data,subtree->left_);
            }
            else{
```

Insert - recursive version

Similar to search, we must write a separate private recursive insert() function that is called from the public insert function. Similar to search(), the recursive insert function also requires a pointer to the node we are trying to insert the data into. We think about this as inserting data into subtree. In python, there isn't really a pass by reference concept. As such, we will need to utilize the return statement in order to correctly attach our new tree.

Python   **C++**

```python
def insert(self, data, subtree):
```

Similar to search, we must write a separate private recursive insert() function that is called from the public insert function. Similar to search(), the recursive insert function also requires a pointer to the node we are trying to insert the data into. We think about this as inserting data into subtree. However, we actually want to modify the pointer being passed in so we are going to pass in a reference to the pointer instead of just the pointer:

```cpp
void insert(const T& data, Node*& subtree);
```

As with search, we want to start by figuring the base and recursive cases. For the base case, if the tree was empty we will simply make the current node the root of the tree. If not, we will want to insert the value either into the left or right subtree depending on how it compares to the data in the root of the subtree.

```python
class BST:
    class Node:
        # Node's init function
        def __init__(self,data=None,left=None,right=None):
            self.data = data
            self.left = left
            self.right = right

    #BST's init function
    def __init__(self):
        self.root = None

    def ins(self, data,  subtree):
        if(subtree==None):
            return BST.Node(data)
        elif(data < subtree.data):
            subtree.left = self.ins(data,subtree.left)
            return subtree
        else:
            subtree.right = self.ins(data, subtree.right)
            return subtree

    def recursive_insert(self,data):
        self.root = self.ins(data,self.root)
```

```cpp
class BST{
    struct Node{
        T data_;
        Node* left_;
        Node* right_;
        Node(const T& data, Node* left=nullptr, Node*
right=nullptr){
```

InOrder Print function

To print all values in the tree from smallest to biggest, we can write the function recursively. This is an example of an inorder tree traversal. That is we will visit every node in the tree exactly one time and process it exactly one time.

This function is most easily written recursively. As with all other recursive function for bst we will pass in the root of the subtree. This function will be called from the public inOrder print() function.

The base case for this function is simply that of an empty tree. If the tree is empty, there is nothing to print so we do nothing and exit function.

if tree isn't empty we want to start by printing all values smaller than the current node (stored in left subtree), then the current node then all the values bigger than the current node (stored in right subtree). if its a tree, we'll use the inOrderPrint() function to print those so that they will be printed in order also.

Python   C++

```python
class BST:
    class Node:
        # Node's init function
        def __init__(self,data=None,left=None,right=None):
            self.data = data
            self.left = left
            self.right = right

    #BST's init function
    def __init__(self):
        self.root = None


    def print_inorder(self, subtree):
        if(subtree != None):
            self.print_inorder(subtree.left)
            print(subtree.data, end = " ")
            self.print_inorder(subtree.right)

    def print(self):
        self.print_inorder(self.root)
        print("")
```

```cpp
    class BST{
    struct Node{
        T data_;
        Node* left_;
        Node* right_;
        Node(const T& data, Node* left=nullptr, Node*
right=nullptr){
            data_=data;
            left_=left;
            right_=right;
        }
    };
```

Pre Order Print

The pre-order print function is similar to the inorder print function except that
we print the current node before printing its subtrees. This is the ordering that
would be used if we were to do something like listing the contents of a file
system.

The code for this is pretty much identical to inOrderPrint(). The only difference
is in when we print the current node. What we want to do is print the current
node, then its left and right subtrees

Python   **C++**

```python
class BST:
    class Node:
        # Node's init function
        def __init__(self,data=None,left=None,right=None):
            self.data = data
            self.left = left
            self.right = right

    #BST's init function
```

```cpp
    class BST{
    struct Node{
        T data_;
        Node* left_;
        Node* right_;
        Node(const T& data, Node* left=nullptr, Node*
right=nullptr){
            data_=data;
            left_=left;
            right_=right;
        }
    };
    Node* root_;
    //prints the values of the tree who's root is stored in
subtree
    //by printing the value in a node then the values in its left
subtree
    //followed by the values in the right subtree
    void preOrderPrint(const Node* subtree) const{
        //base case is we have an empty tree... in that case we
do nothing
        //and exit the function
        if(subtree!=nullptr){

            //print value in current node
            std::cout << subtree->data_ << std::endl;

            //print left subtree
            preOrderPrint(subtree->left_);

            //print right subtree
            preOrderPrint(subtree->right_);
        }
    }
public:
...
```

Destructor

A destructor must deallocate every node in the tree. There are different ways that this could be done but the simplest is to write a post-order tree traversal. The reason its a traversal is simply because we need to visit every node in the tree and destroy it. The reason it has to be done in a post order manner is because we must first destroy both subtrees attached to a node before deleting the node or we will lose access to those subtrees.

Similar to the print functions, the destructor will be written by private recursive function that accepts a the root of the subtree we are trying to deallocate. We will call this function destroy:

```cpp
void destroy(Node* subtree);
```

Note that because we actually don't care about changing the value of the pointers themselves we don't need a & after the Node* part. Just go through and deallocate every node

As with all the other traversals, we have a base case where if the tree is empty we do nothing. Otherwise we will simply first begin by destroying the left subtree, then the right subtree then we can deallocate the root node of the current subtree (pointed to by the argument subtree)

```cpp
class BST{
struct Node{
    T data_;
    Node* left_;
    Node* right_;
    Node(const T& data, Node* left=nullptr, Node*
right=nullptr){
        data_=data;
```

Augmented Data Structures

An augmented data structure is a data structure where we add an extra piece of information so that we can acheive better performance. From a user's perspective the augmented data structure works no differently than the non-augmented version.

We will be looking at two augmented data structures in this section and they both help to solve a problem with the basic version of the BST.

Consider the following set of numbers:

5, 3, 7, 2, 9, 4, 6, 1, 8

If we were to insert each of these values in order into a BST using the algorithm we currently have studied, our tree would look like this:
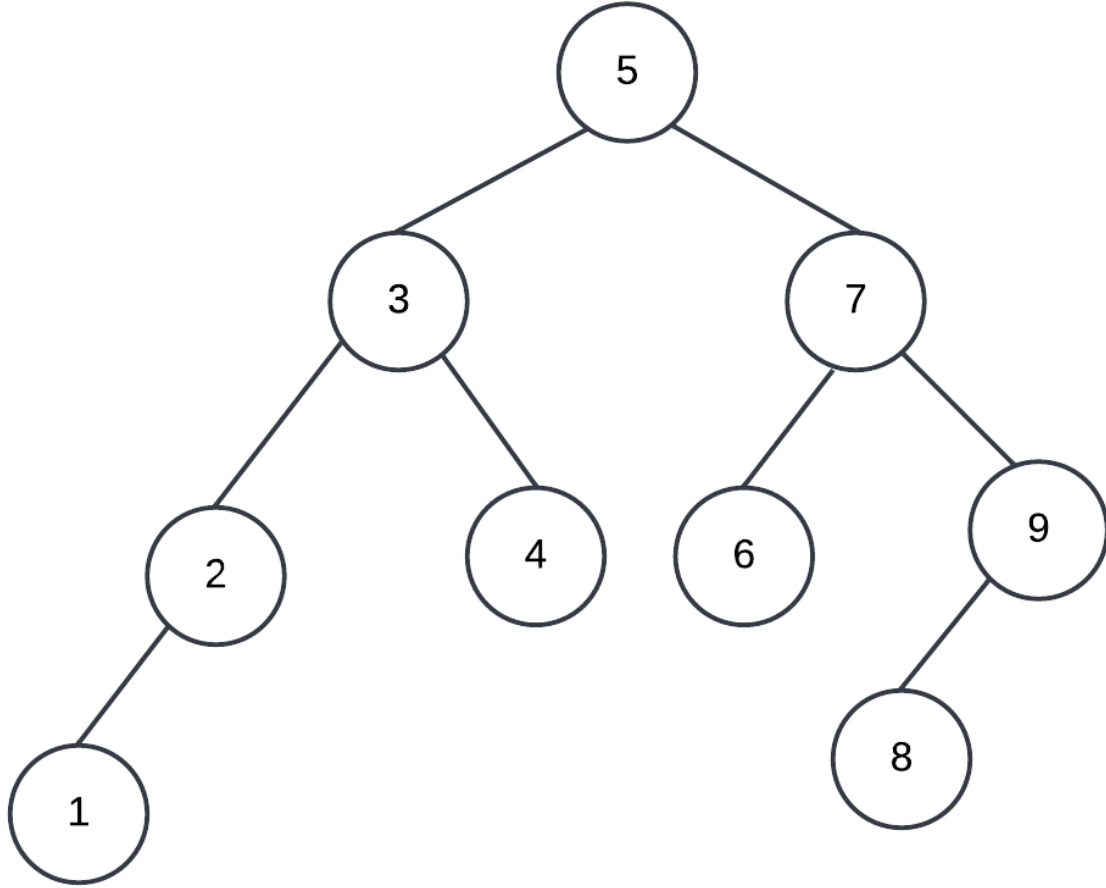
However, suppose we were to created a BST with exactly the same numbers but in the following order instead:

1, 2, 3, 4, 5, 6, 7, 8, 9

The tree created by the insertion algorithm would look like this:
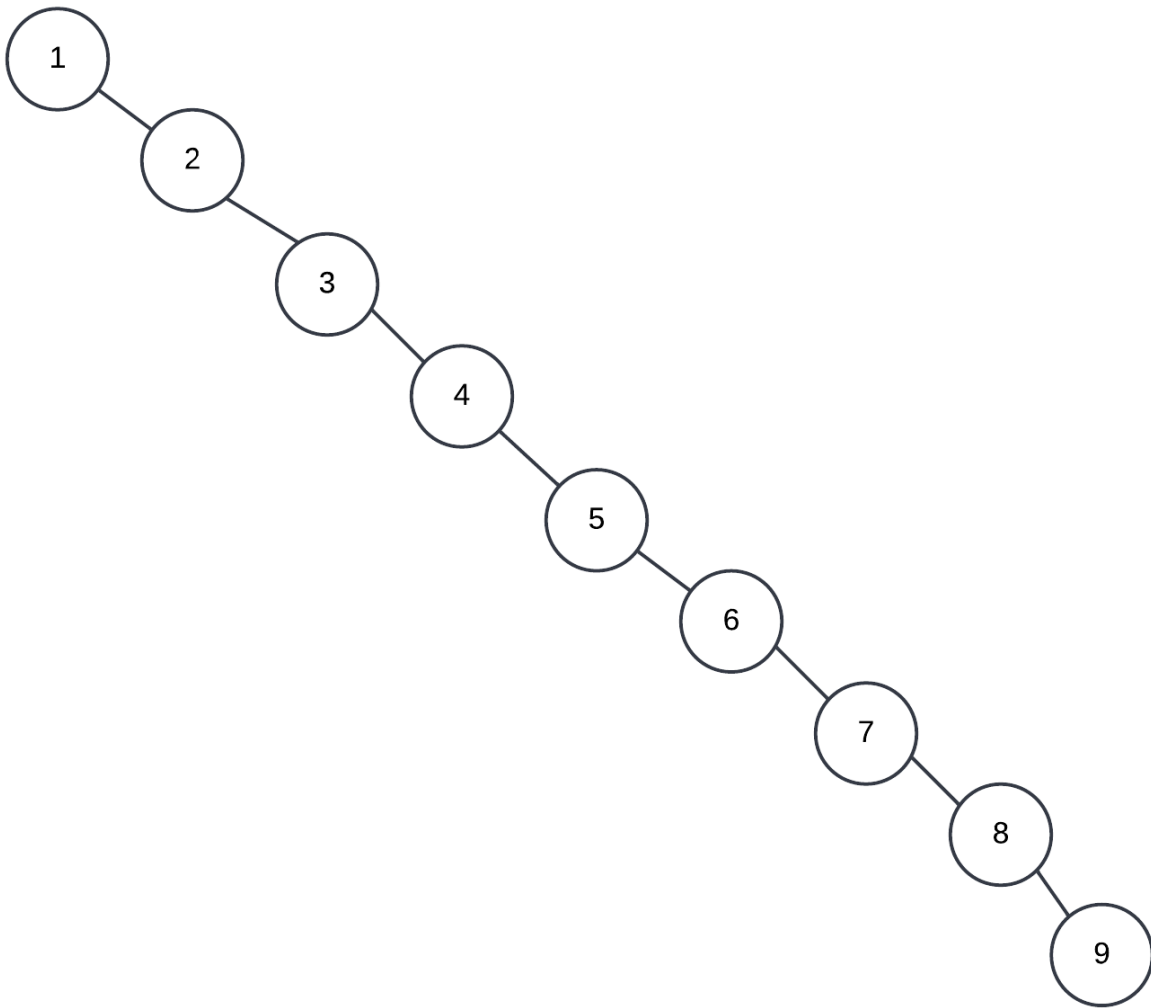
In the first case, the tree looks like tree. In the second case it looks like a stick. The problem of a tree that looks like a stick is that its run time for searching is not log n. its basically a linked list and its performance will match that of a linked list and thus search will perform in O(n) time.

As we cannot know how our data structure will be used, or prevent someone from using our data structure with sorted data, it would be good if we can ensure that no matter how the data comes in, the tree will be tree shaped.

In this chapter we will look at how we can add a little bit of extra information to

our tree and use that to help ensure our tree does not turn into a stick no matter how we receive our data.

Tree Balance

Before we consider how to look make our tree behave properly, we should consider what it means for a tree to be "tree shaped". We want something a little more formal than just "tree shaped". In this section we look into these definitions:

Perfectly Balanced Binary Trees

A binary tree is perfectly balanced if for every node within the tree, the number of nodes in its right and left subtrees differ by no more than one.

Note how this definition says that this is true for every node... not just the root.

So if we take any subtree within a perfectly balanced tree, that subtree will be perfectly balanced.

Here are a few trees that are perfectly balanced.

Here are a few that are not perfectly balanced and why they are not.

Notice that with this definition, a complete binary tree (which we said was a good tree) may not be pefectly balanced. For example this tree is not perfectly balanced even though it is complete.

What this tells you is that the perfectly balanced tree description is too restrictive and will exclude many trees that actually have same performance as that of a perfectly balanced tree. Getting a tree to be perfectly balanced is also an expensive process. Thus, we typically will allow for a more relaxed definition of what a good "tree shape" is

Height Balanced Binary Trees

A binary tree is height balanced if for every node within the tree, the height of its right and left subtrees differ by no more than one.

Notice how this definition is almost the same as that of perfectly balanced trees. The only difference is that we measure the height of the tree as opposed to the number of nodes in the tree. This is useful as the height measures the maximum number of nodes we will consider.

Similar to perfectly balanced trees, the height balanced definition is recursively applied. Thus every subtree of a height balanced tree is also height balanced.

Here are some height balanced binary trees

Here are some non-height balanced binary trees

By ensuring that the trees on either side of a node are never significantly taller than each other, we can ensure that any operation done will never visit more nodes regardless of the value of the operand. This is how AVL trees work. By storing the height of a node's subtree in each node, a node balance can easily be calculated and used to maintain height balance

Branches

A branch is defined as all the nodes from the root of the tree to a leaf inclusive of both root and leaf. Another way of looking at whether a tree is "tree shaped" is to consider the differences between the length of the the branches. We can look at whether or not a tree is good by looking at the differences in the length of the branches. This is how red-black trees work. They use a set of node colouring rules to ensure that no branch is more than twice the length of any

other branch in the tree.

AVL Trees

An AVL tree creates and maintains a **_height balanced_** binary search tree. A height balanced tree is either empty or the height of the left and right subtrees differ by no more than 1. A height balanced tree is at most 44% taller than a perfectly balanced tree and thus, a search through a height balanced tree is O(log n). Insert and delete can also be done in O(log n) time.

Height Balance

AVL trees work by ensuring that the tree is height balanced after an operation. It does this by storing the height of every subtree in it's root. The storing of this height is the augmentation... we add this extra bit of info to every node and ensure its correctness after each operation that might modify a tree. We can then use this information to ensure that our tree is height balanced. By knowing how tall each subtree is, we can tell whether or not any subtree is too tall by calculating the balance from the height.

The balance of any node is calculated as:

```
Node's balance = height of right - height of left
    of node         subtree                subtree
```

When a tree is height balanced the balance of every node is either -1, 0 or +1

In an AVL tree, whenever we do any operation that may modify the tree, we ensure that no node's balance ends up as anything other than -1, 0 or +1.

Note that storing of the height in each node is essential. We cannot calculate the height as finding the height of a tree with n nodes is O(n)

The height balance of a node is calculated as follows:

Height Balance Implication

Recall that the following is the node's balance formula:

```
Node's balance = height of right - height of left
    of node        subtree                subtree
```

The above formula means that if the right subtree is taller, the height balance of the node will be positive. If the left subtree is taller, the balance of the node will be negative.
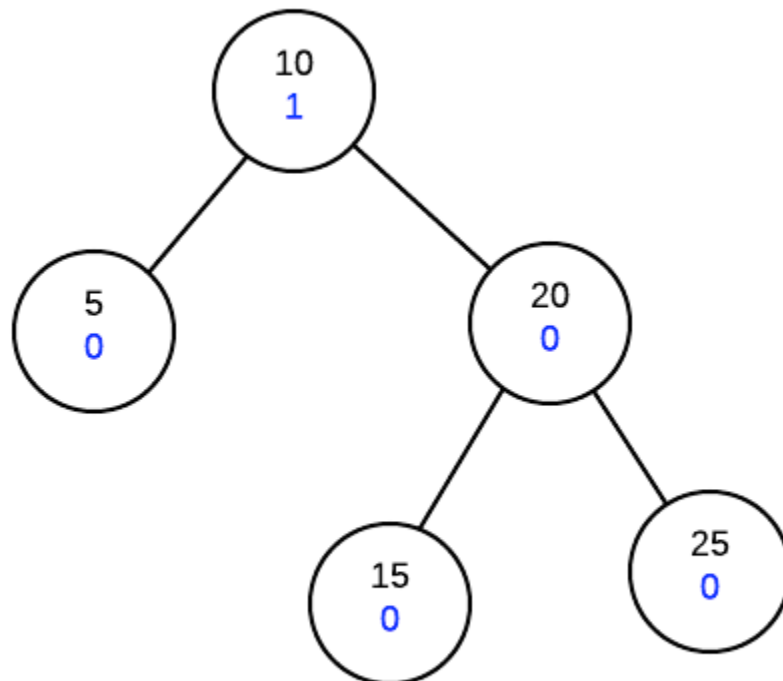
Insertion

Insertion in AVL tree is starts out similar to regular binary search trees. That is we do the following:

- Find the appropriate empty subtree where new value should go by comparing with values in the tree.
- Create a new node at that empty subtree.
- New node is a leaf and thus will have a height balance of 0
- go back to the parent and adjust the height balance.
- If the height balance of a node is ever more than 1 or less than -1, the subtree at that node will have to go through a rotation in order to fix the height balance. The process continues until we are back to the root.
- **NOTE: The adjustment must happen from the bottom up**

Example

Suppose we have start with the following tree (value on top is the value, value on bottom is the height balance
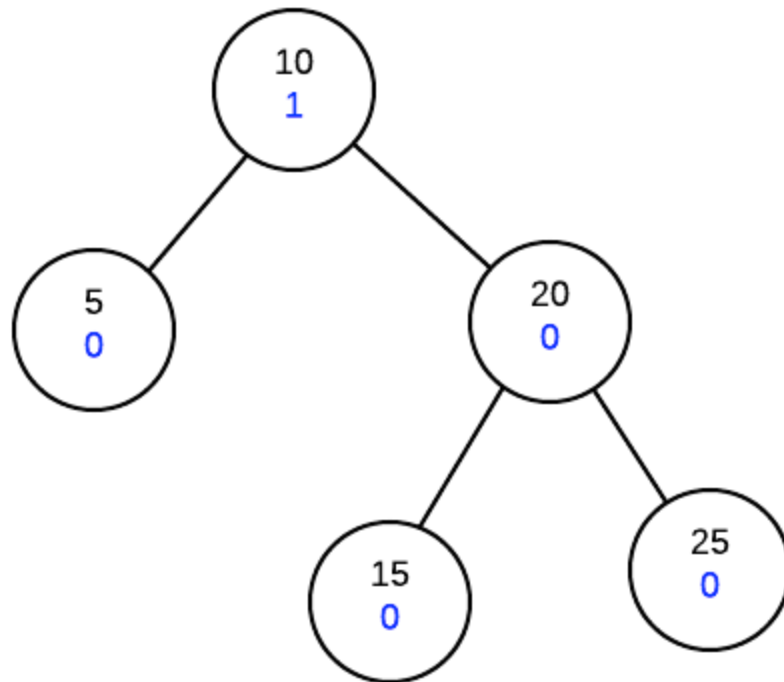
Insert 30

At this point, we have adjusted all the height balances along the insertion path and we note that the root node has a height balance of 2 which means the tree is not height balanced at the root.

In order to fix our tree, we will need to perform a rotation
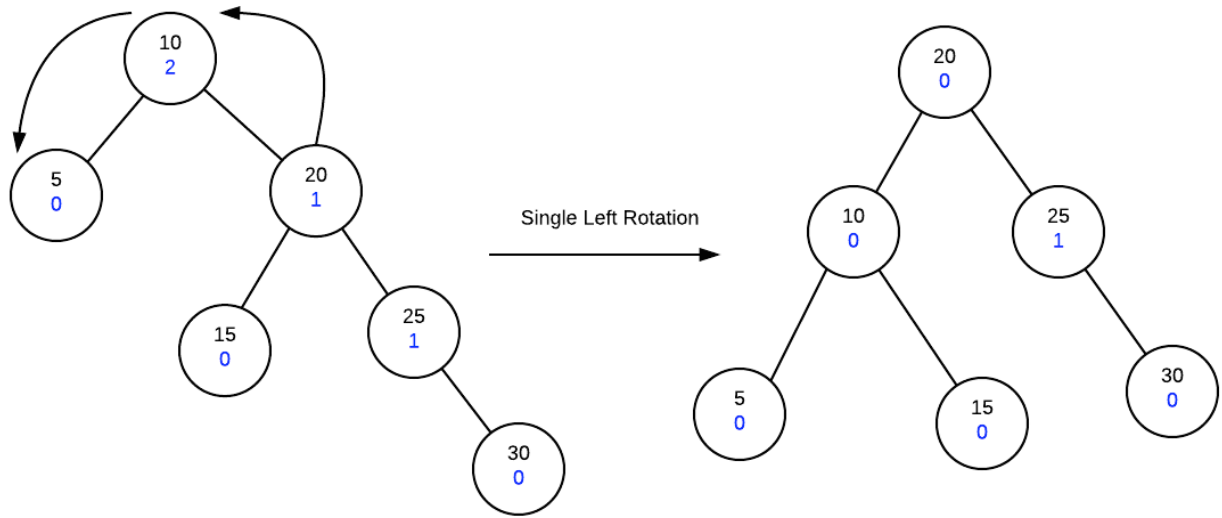
Insert 27

We start with our tree:
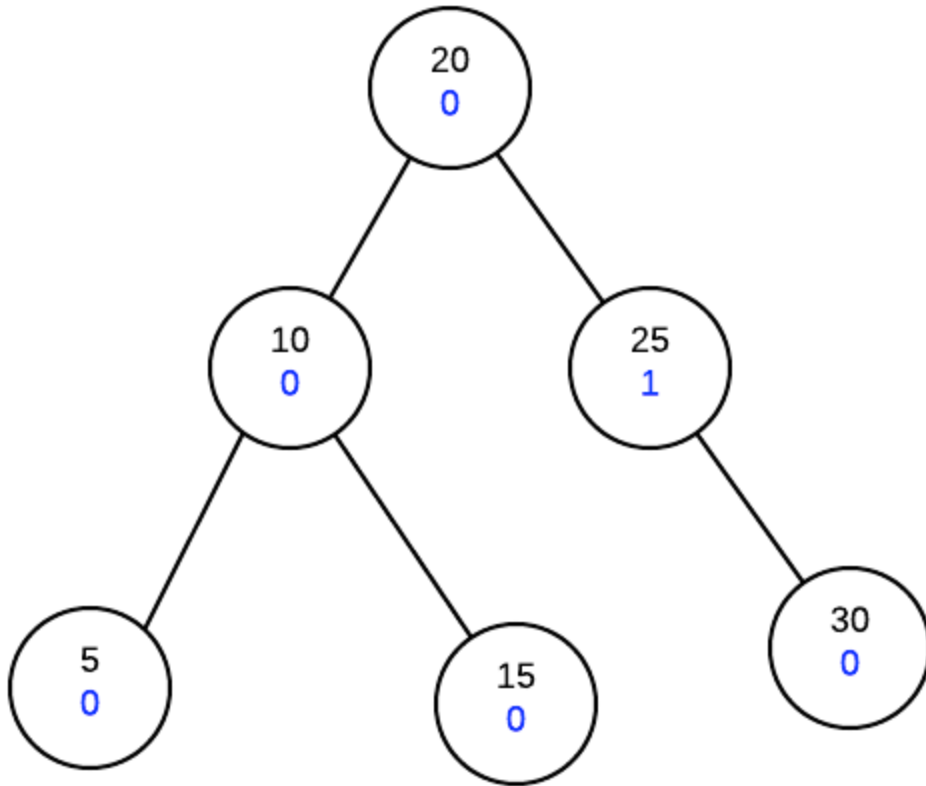
Now we find the correct place in the tree and insert the new node and fix the
height balances
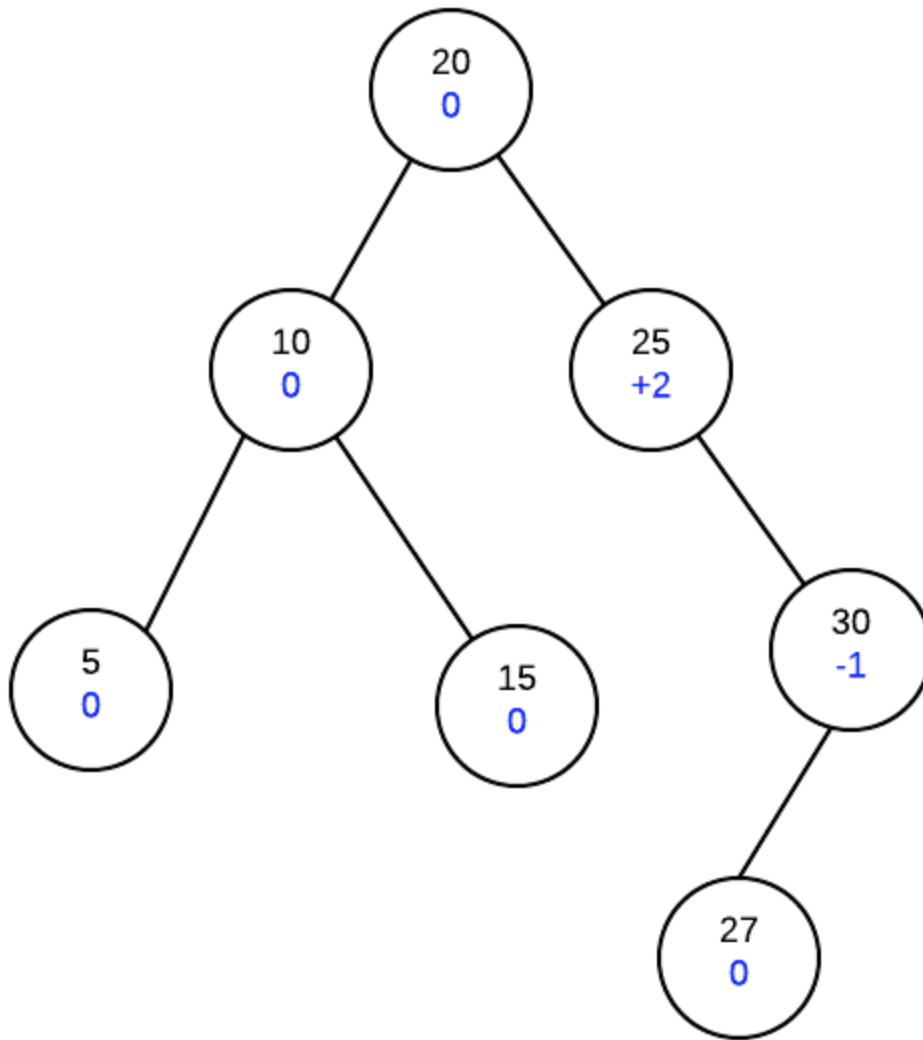
Now, as we reach node 25 and see that it the height balance is +2. If we then look at it's child's height balance we find that it is -1. As the signs are different, it indicates that we need a double rotation. Different signs indicate that the unbalance is in different directions so we need to do a rotation to make it the same direction then another to fix the unbalance.

double rotation part 1

Double Rotation part 2
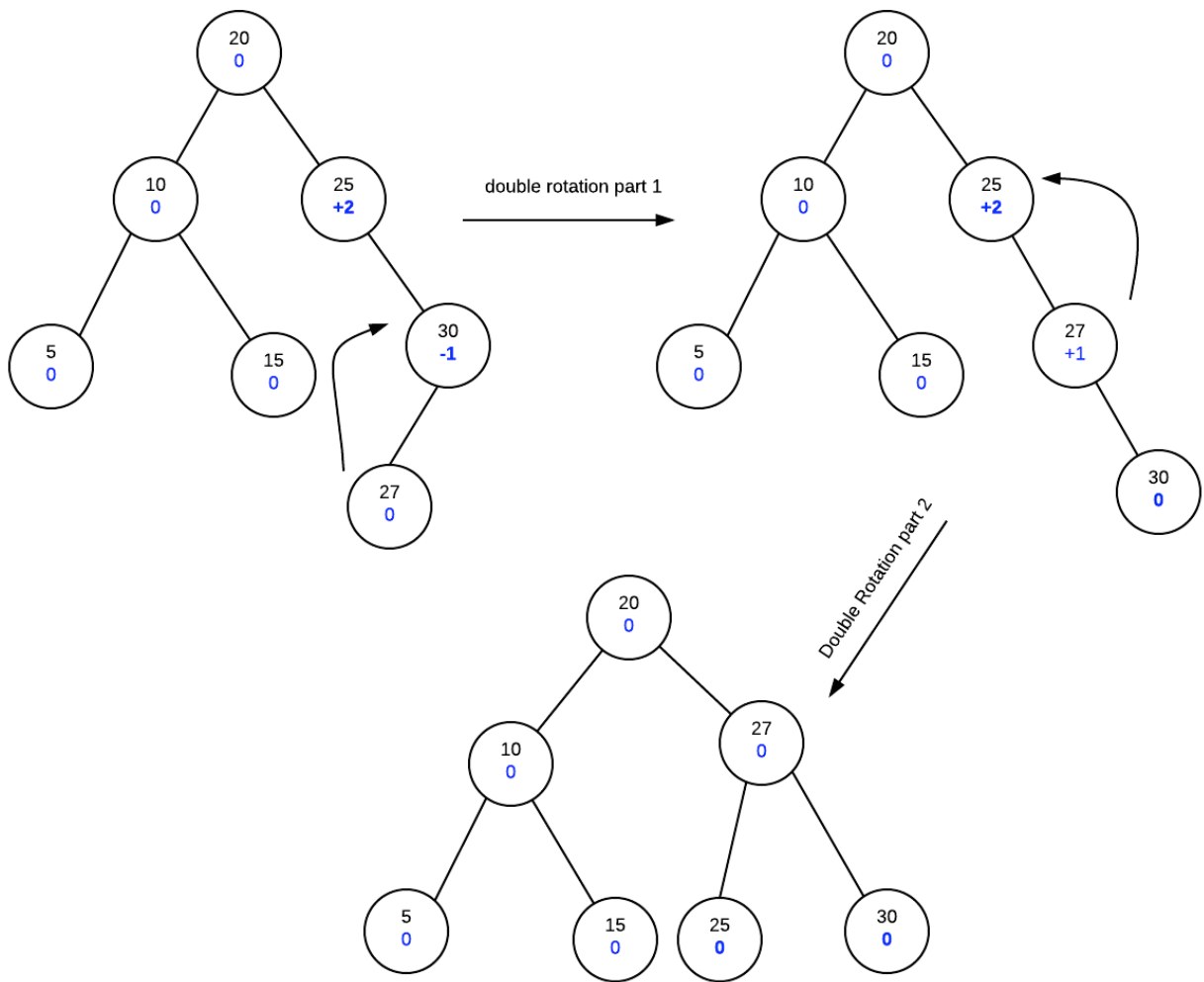
Why does this work?

Single Rotation

We always fix nodes starting from the insertion point back to the root. Thus, any node in the insertion path further towards leaf nodes must already be fixed.

Consider the following idea of what an avl tree looks like:
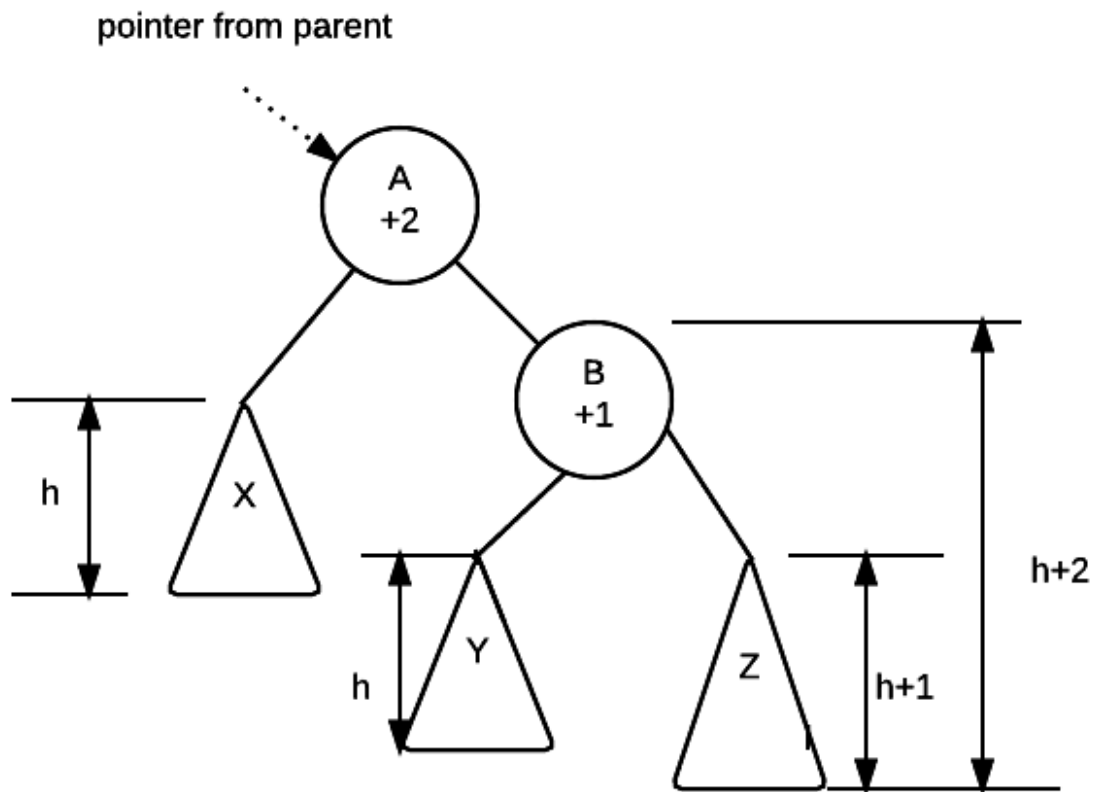
In this diagram, we have two nodes A and B and we see their height balance. We know that the subtrees X, Y and Z are valid avl trees because they would have been fixed as we process our tree back to the root.

From here we also know that:

```
all values  <  A  <  all values  <  B  <  all values
in X                      in Y                 in Z
```

While we don't know how tall X, Y and Z are, we know their relative heights because we know the height balance.

Thus, if X has a height of h, B must be h+2 tall because the height balance at A is +2. This means that the right subtree at B is 2 taller than A.

Continuing on, we know that Z is the taller of the two subtrees of B because the height balance is +1, and thus Z must be h+1 tall while Y must be h tall.

A rotation repositions B as the root of the tree, makes node A the left child of B. Make Y the right child of A.

Given that X, Y and Z are unchanged, the height balance at A and B will become 0.
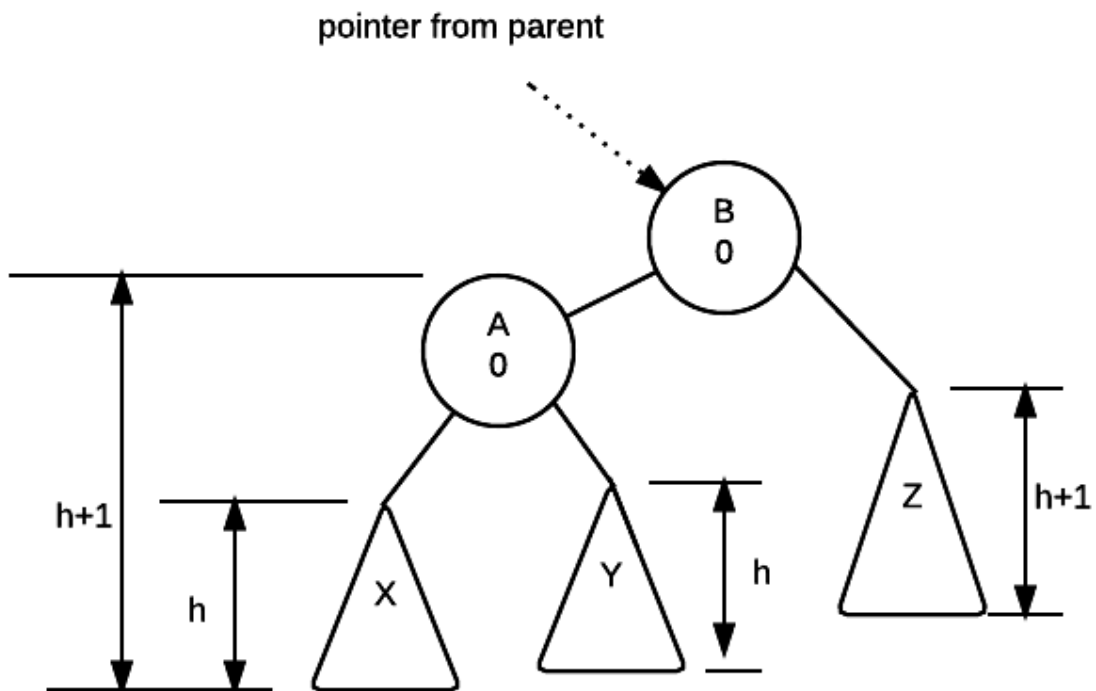
The mirror of the above is true for single left rotation

Double Rotations

A similar explanation of why double rotations work can be reasoned out by looking at the following tree:
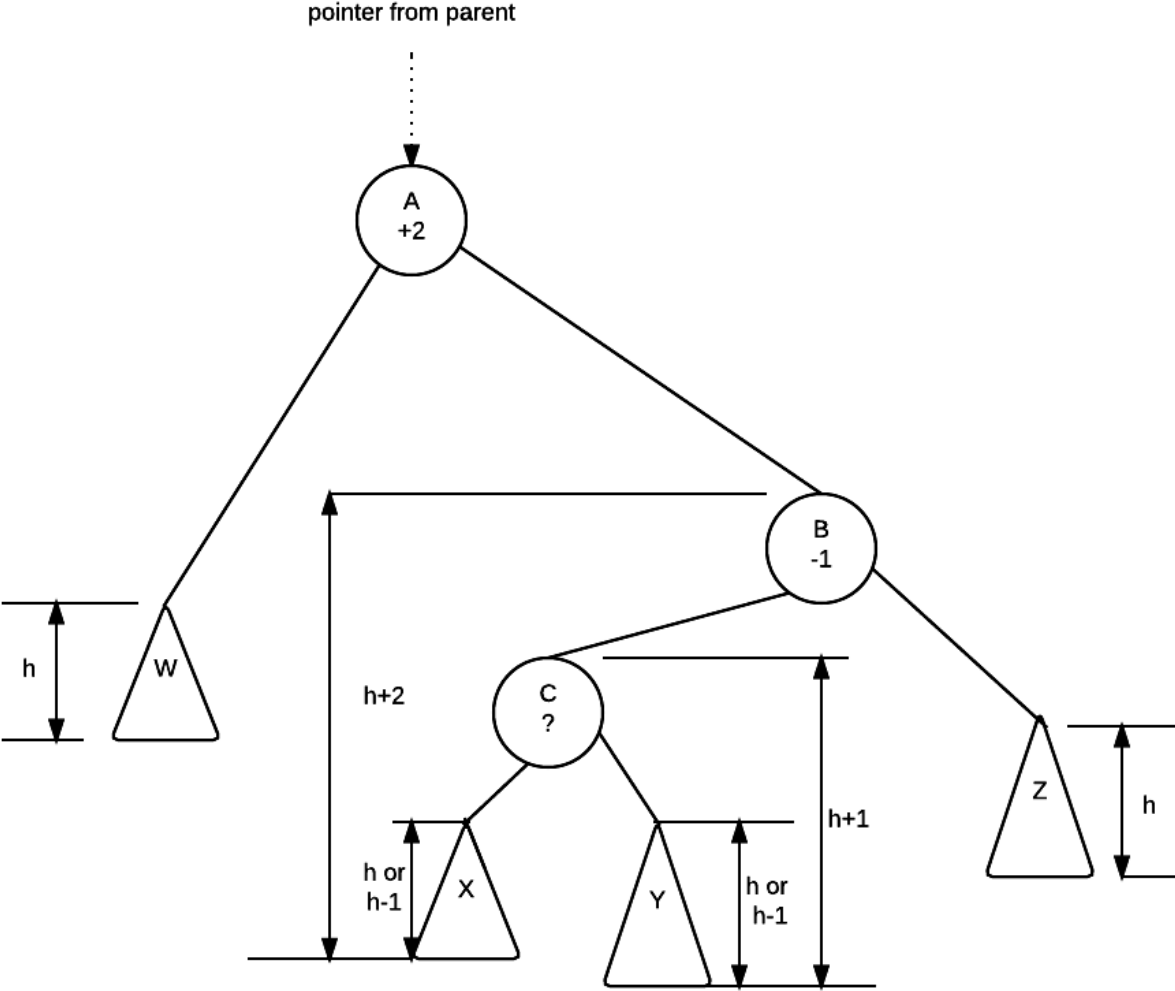
We know that W, X, Y and Z are all valid avl trees. We also know the following about the values in each of the trees and nodes

```
all                 all                 all             all
values  <  A  <  values  <  C  <  values < B <  values
 in W               in X                in Y             in Z
```

Now... we know the height balance of A and B (off balance node and child) we do not know the exact height balance of C. However, we do know that it is a valid avl tree, so C's height balance must be either -1, 0 or +1.

So, if C's height balance is 0, then both x and y will have height of h.

if C's height balance is +1 then y will be h and x would be h-1

if C's height balance is -1 then x would be h and y would h-1

Perform a double rotation:

double rotation part 1

Double Rotation part 2
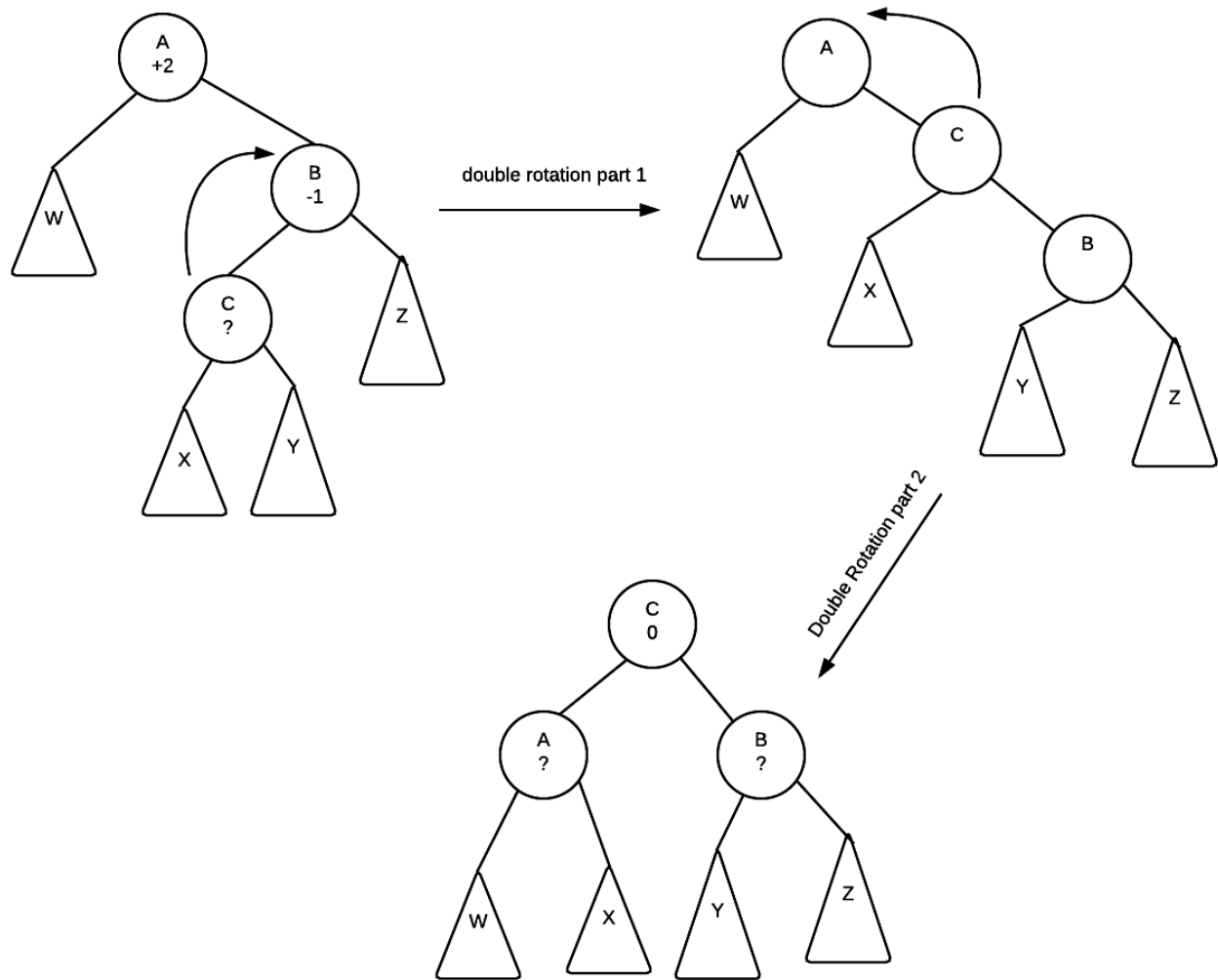
After the rotation is completed, notice that the position of the subtrees W,X, Y and Z along with the nodes A, B and C are placed in a way where their ordering is properly maintained.

Furthermore, The height balance of C becomes 0 regardless of what it was initially. The final height balance of A and B depends on what the original height balance of C was:

original height balance of C	height of X	height of Y	final height balance of A	final height balance of B
-1	h	h-1	0	+1
0	h	h	0	0
+1	h-1	h	-1	0

Deletion

The deletion algorithm for AVL trees must also keep the tree height balanced. This section will look at how deletion works.

The deletion algorithm does the following:

- In general follow BST deletion rules.
  - if node is leaf delete node, set pointer from parent to nullptr
  - if node has one child, have parent point to only child
  - if node has two children replace deleted node with inorder successor
- Because a deletion could potentially shorten a tree, we might need to adjust the tree. Starting with the deleted node (which may not actually be the node that contains the value we are getting rid of... more on this later) work our way back up to root, fix the height/balance info and rotate as needed.

The rest of this section will look at how it works

Deletion always removes a leaf node

Deletion always removes a leaf node. The previous statement may seem a bit odd because it seems like it can't be true, but it is. Note that node being remove isn't necessarily the node where value being removed was originally found. Lets consider the three deletion cases to explain why it is true:

Value is found in a leaf node

In this case, clearly the node to be removed is a leaf as we found it there. Thus, we get rid of it, adjust or height/balance values going back up the tree

Value is in a node with exactly one child

A node with only one child means that the only child is a leaf. The reason for this is because our tree is height balanced to start with. If the only child had a child, the node wouldn't have been height balanced as illustrated in the following diagram:

From parent

A
+1

B
0

A is to be deleted, it has
one child. A must have had
balance of +/- 1

From parent

A
+2

B
0

C

In order for B to not be a
leaf, A 's balance would
have had to be invalid
(which isn't allowed)
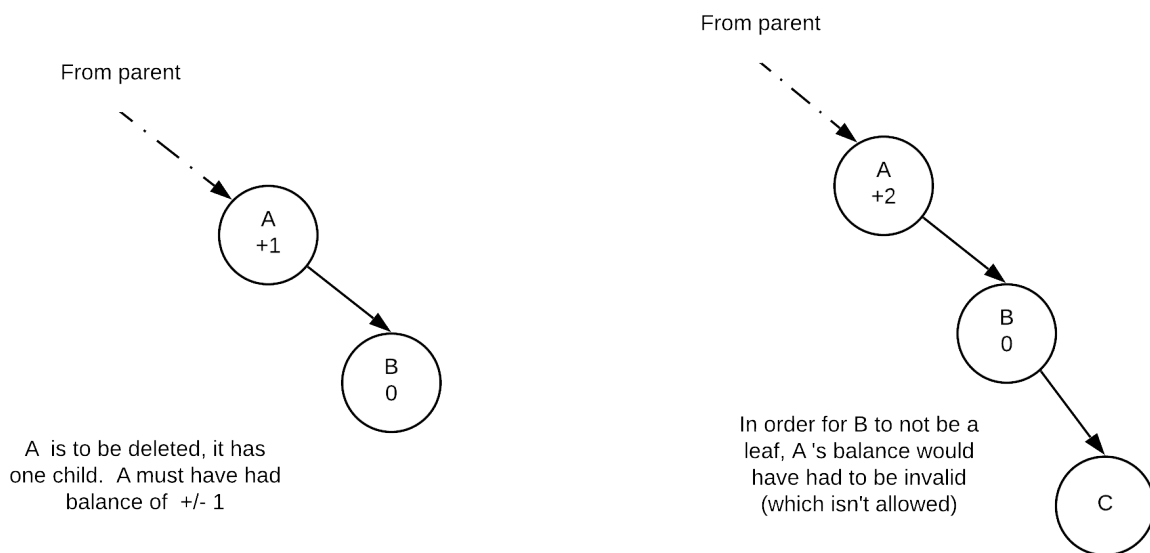
Thus, what effectively happens is that we end up with a tree that is shaped like we were removing B

Value is in a node with exactly two children

The algorithm to remove a node with two children is to replace the node with its inorder successor node. This node is found by going one node to the right then going left until you can't. The inorder successor itself can only have a right child (if it had a left child it wouldn't be the inorder successor). As with the case of nodes with only one child, the inorder sucessor's right child must be a leaf because the inorder successor must have been height balanced before the deletion operation.

From parent

A

B

D
+1

E
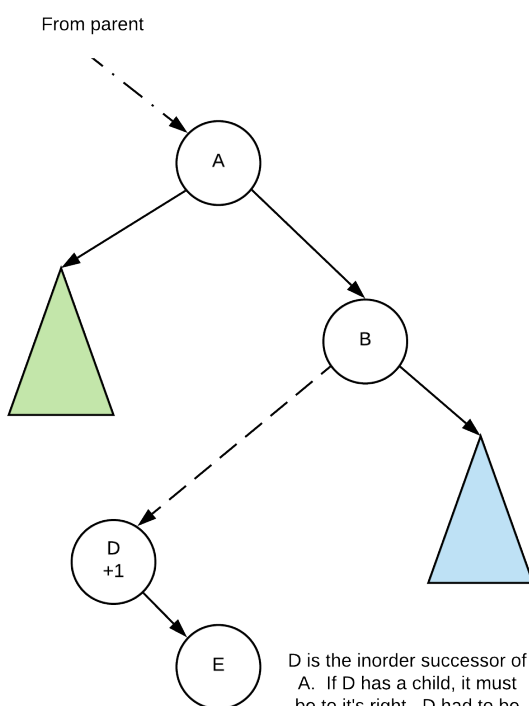
D is the inorder successor of A. If D has a child, it must be to it's right. D had to be height balance to begin with thus E must be leaf (same reason why nodes with one child had to be a leaf

From parent

D

B

E
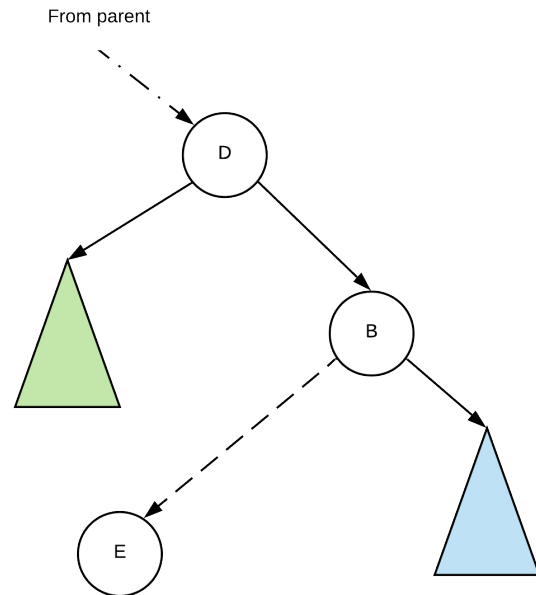
Deletion of A, involves promoting D to take the place of A and having E take D's old spot Effectively the node being removed is the one where E was

Fixing the tree

When will we need to fix the tree?

Clearly if the deletion doesn't shorten its subtree, there is nothing that needs to be done.

Thus, we only need to consider what happens if deleting the node causes the tree to shorten. In the examples below we only consider those situations.

Let us consider the following. In each case, B is the node being removed, A is some node along the path to B

Balance of A was initially 0

Suppose that A initially had a balance of 0. Removing B, shortened the blue subtree. However, as the balance was 0 at A, A's balance will simply go either to +/- 1. No rotation needed. Also not only was no rotation needed, the height of the tree at A is actually exactly the same as it was before the deletion and thus, we can stop fixing the tree as nodes further up will not be affected
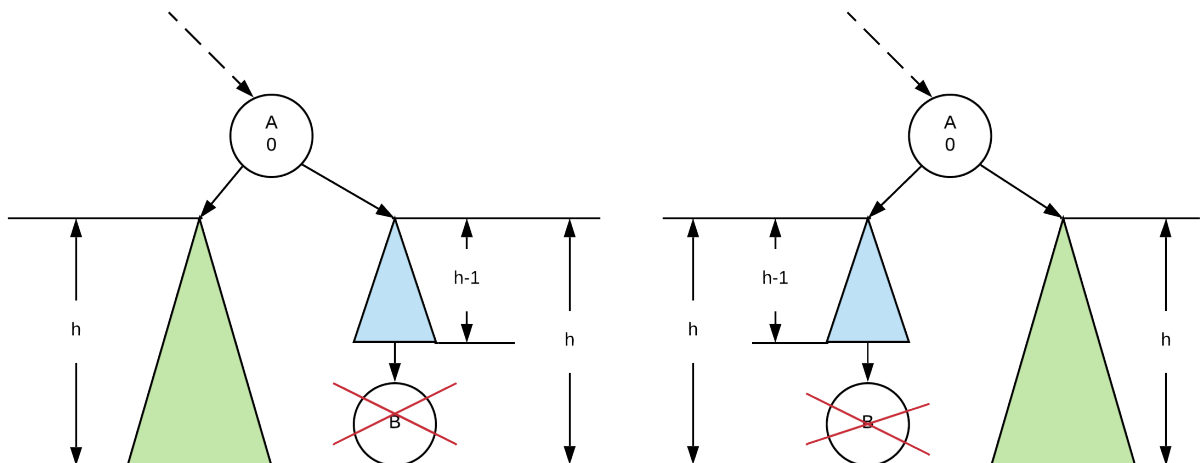
Balance of A was initially +/-1

Two things can occur. In the first case our node comes from the taller of the subtrees tree and it causes the tree to shorten.
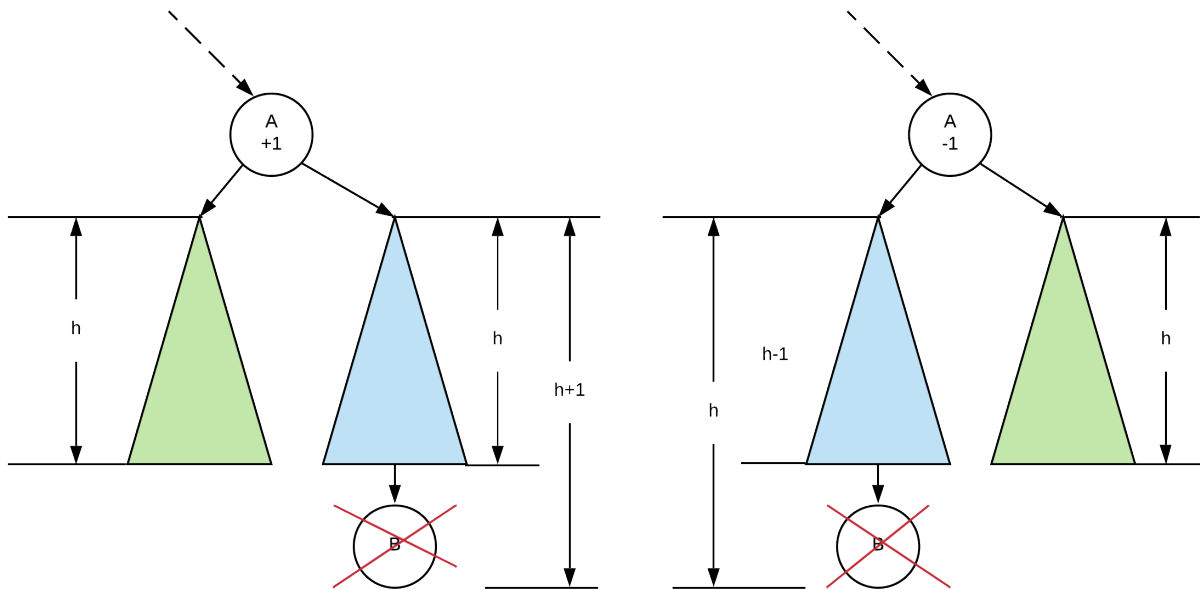
If that is the case then we do not have to do any rotations at A because A's balance will become 0. However, the subtree with root A did get shorter, so we must continue going up the tree as other nodes further up may require a rotation

The second case is when we take out a node from the shorter subtree:
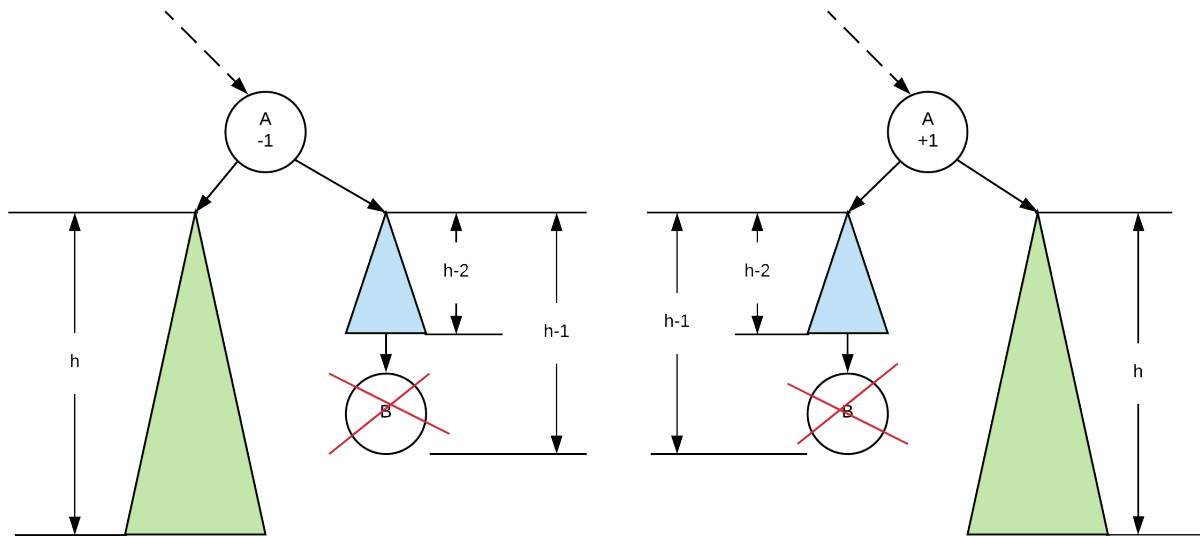
If we did this then our balance will go to +/- 2. This would require rebalancing. Like an insertion, rebalancing is simply a matter of applying a single or double rotation. We simply need to follow the same rules and perform the rotation

After a rebalancing a node the subtree might have become shorter than it was because of the rotation, and thus we may need to go further up the tree to fix it.

Red Black Trees

Red-Black Trees are binary search trees that are named after the way the nodes are coloured.

Each node in a red-black tree is coloured either red or black. The height of a red black tree is at most 2 * log(n+1).

A red black tree must maintain the following colouring rules:

1. every node must have a colour either red or black.
2. The root node must be black
3. If a node is red, its children must be black (note that this also implies that red nodes cannot have red parents)
4. Every path from root to a **null node** must have exactly the same number of black nodes.

> **⚠ INFO**
>
> **null nodes** are also sometimes called **null leafs**. these are not really nodes.. they are the "nodes" that null pointers point to... so we when we think about counting black nodes we think about how many black nodes there are to every nullptr in the tree

Insertion

We will begin our look at Red-Black trees with the bottom up insertion algorithm. This insertion algorithm is similar to that of the insertion algorithm we looked at for AVL trees/Binary search trees.

Insert the new node according to regular binary search tree insertion rules. Of the 4 colouring rules, the one rule we don't want to break is rule number 4. Everything we do is to avoid breaking rule 4 (every path from root to every null leaf has same number of black nodes). Thus, New nodes are added as red nodes. We then "fix" the tree if any of the rules are broken.

Note that because we inserted a red node to a proper red-black tree, the only 2 rules that might be broken are:

1. rule 2: root must be black
2. rule 3: red node must have black children

Rule 1 is pretty much not broken as we coloured it red. We also won't break rule 4 because we added a red node so the number of black nodes has not increased.

General Insertion Algorithm

To insert into a red-black tree:

1. find the correct empty tree (like bst) and insert new node as a red node.
2. working way up the tree back to parent fix the tree so that the red-black tree rules are maintained.

Fixing nodes:

- If root becomes red, change it to black.
  - This won't break any rules because you are just adding 1 black node to every branch of the tree, the number of black nodes increase by 1 everywhere. This can only happen as the root as it is the only node that is part of every path from root to nullleaf
- If there are two red nodes in a row:

- ◦ Identify the following nodes:
  - ▪ upper red node as the Parent (**P**)
  - ▪ the lower red node as the Child (**C**)
  - ▪ parent of parent is Grandparent (**G**)
  - ▪ sibling of Parent as Parent's sibling (**PS**)
- ◦ if the **PS** is black
  - ▪ perform a rotation (look at G->P->C, if they form a straight line do a zig-zig(single) rotation, if there is a bend, do a zig-zag (double rotation)
  - ▪ after rotation exchange G's colour with the node that took over G's spot. In otherwords
    - ▪ make which ever node (depends if it was zigzig or zigzag rotation... it will either be **P** or **C**) that took over G's node black
    - ▪ make **G** red
- ◦ if the **PS** is red
  - ▪ exchange the colour of the grand parent with its two children. In otherwords
    - ▪ **G** becomes red
    - ▪ **P** and **PS** becomes black

Example

Starting with an empty tree let us take a look at how red-black tree insertions work.

In the pictures below, null nodes (empty subtrees) are denoted as black circles

Null Nodes

Insert 30

All nodes are inserted as red nodes:
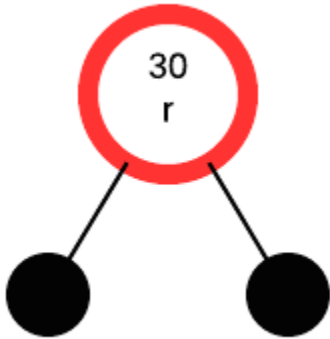
If the root is red, make it black:
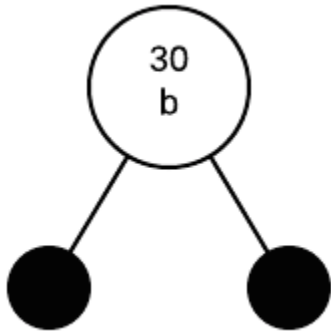
Insert 50

Insert 50 as a red node, parent is black so we don't have to change anything
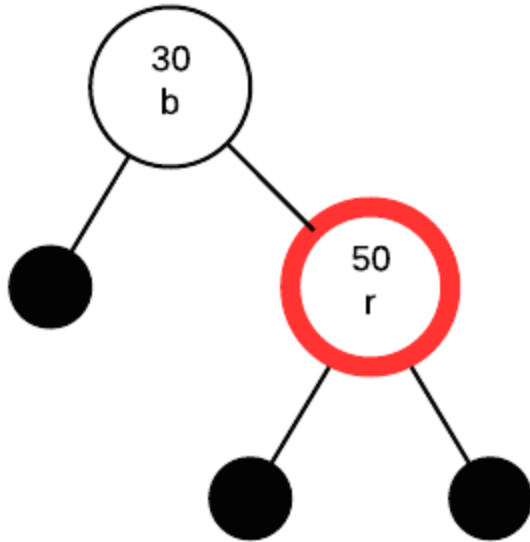
Insert 40
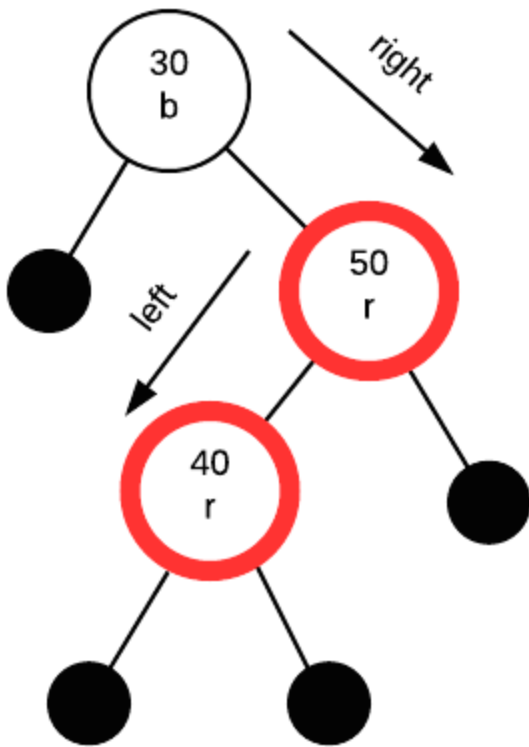
Inserting 40 as a red node.
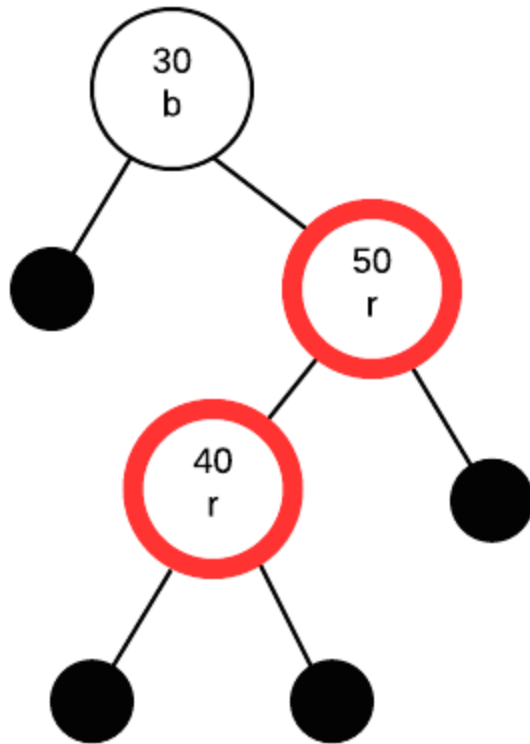
Two red nodes in a row. Identify G, P, PS and C

- **P** - parent (upper red) - 50
- **C** - child (lower red) - 40
- **G** - grandparent 30
- **PS** - parent's sibling - null node to left of 30

What we do depends on colour of **PS.** In this case PS is black, so we will be fixing this with a rotation

The type of rotation depends on the configuration of G, P and C. If the path is from G to C is straight (both left or both right) do a zigzig (single) rotation. If it is angled (left then right or right then left) we need to do a zigzag (double) rotation.

In this case, we need to do a zig zag (double) rotation.

Rotate first 40 and 50
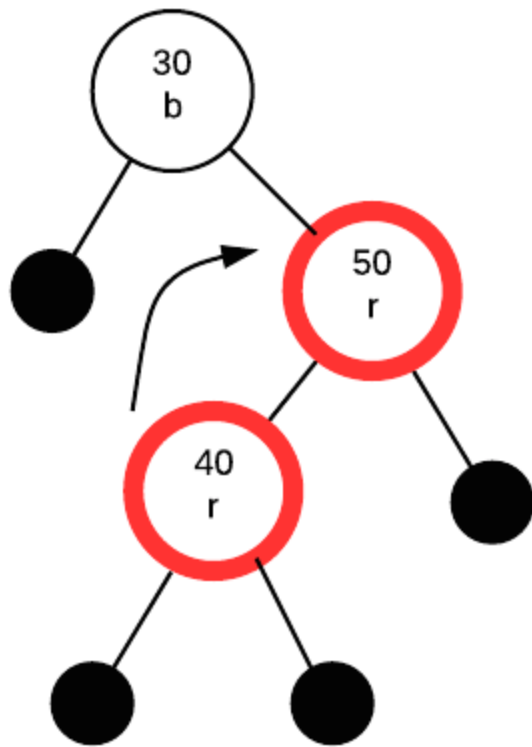
then rotate again with 30 and 40, this time doing a colour swap. A zigzag rotation is just an extra step that is needed to make the insertion path go in the same direction.
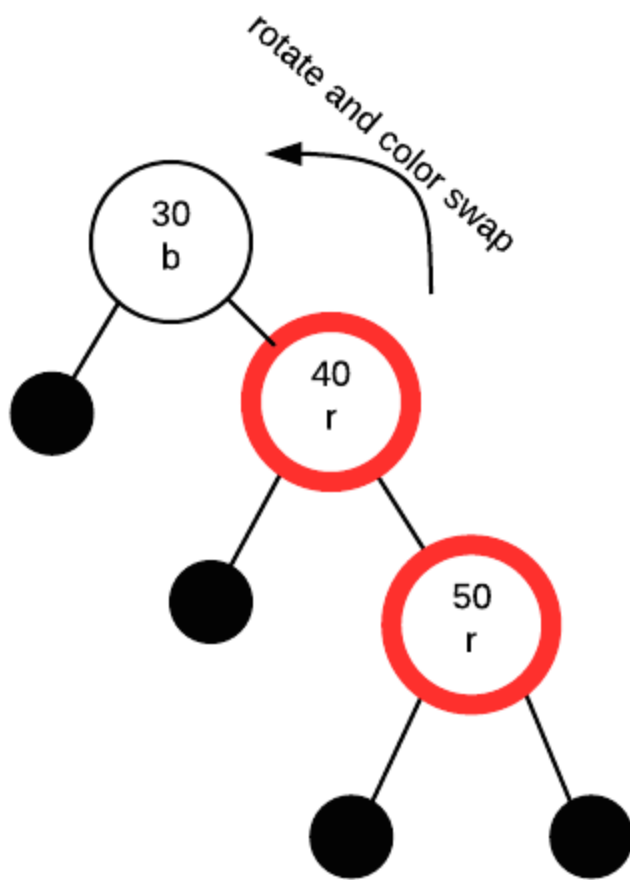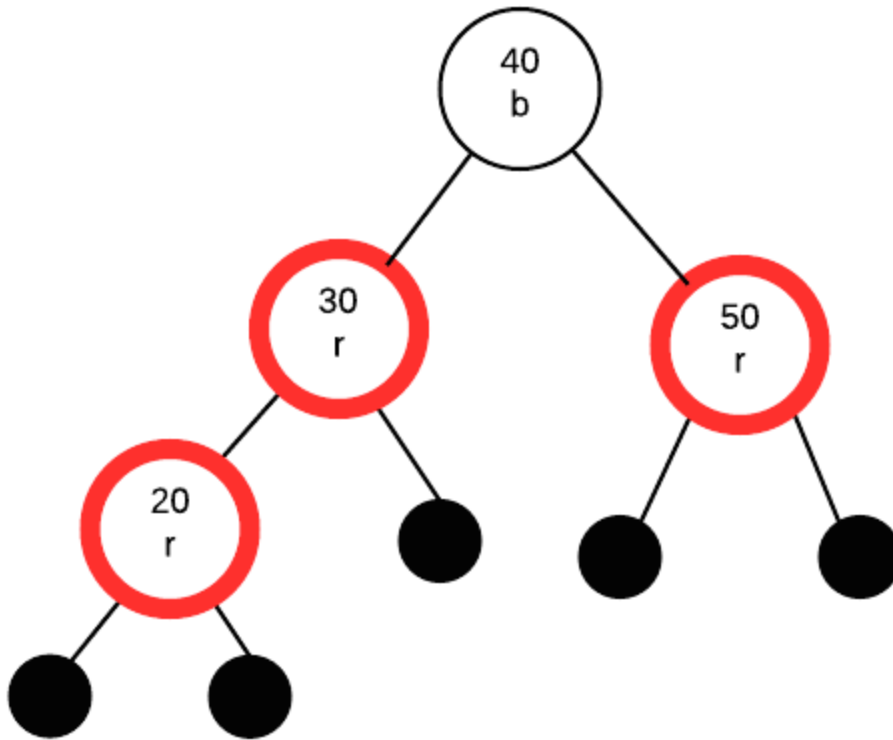
After rotations are complete, we exchange the colour between the node that took over G's spot (40 in this case) and G. Thus, 40 becomes black and 30 becomes red.
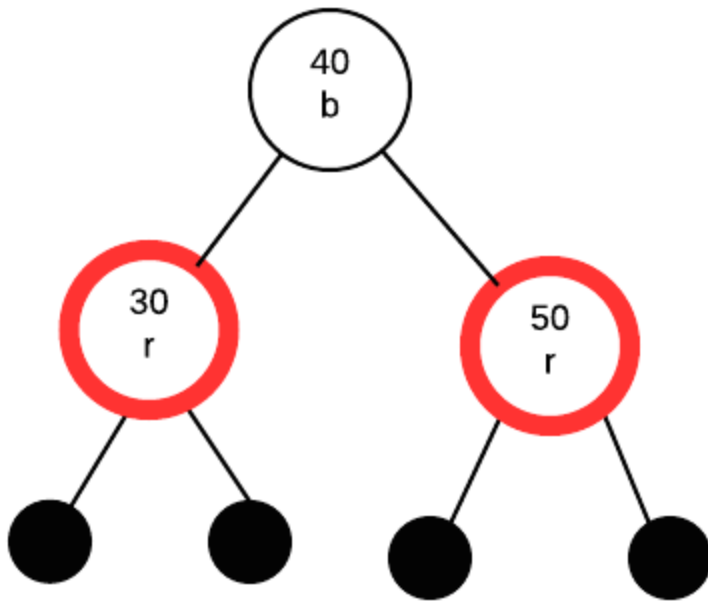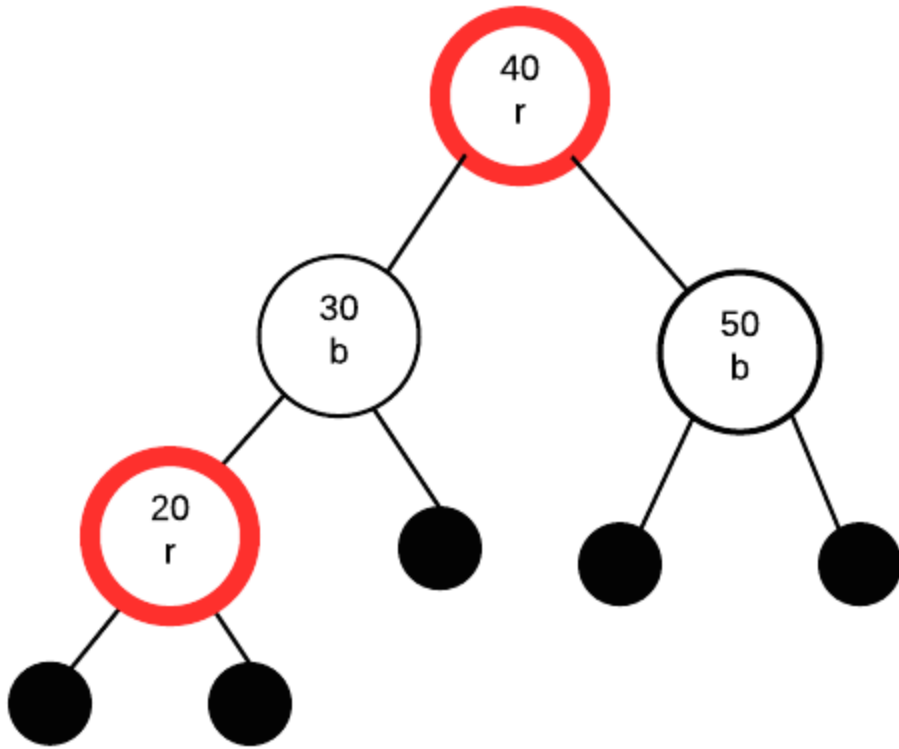
Insert 20
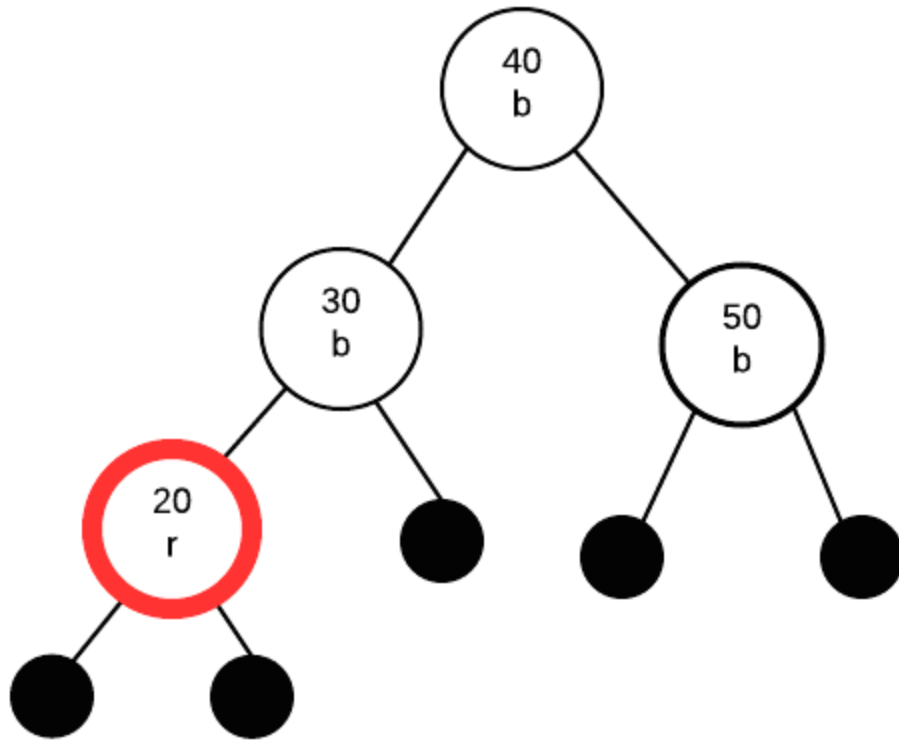
inserting 20 as a red node.

Two red nodes in a row. Identify G, P, PS and C

- **P** - parent (upper red) - 30
- **C** - child (lower red) - 20
- **G** - grandparent - 40
- **PS** - parent's sibling - 50

What we do depends on colour of **PS.** In this case PS is red. Thus we exchange colours between G and its two children:
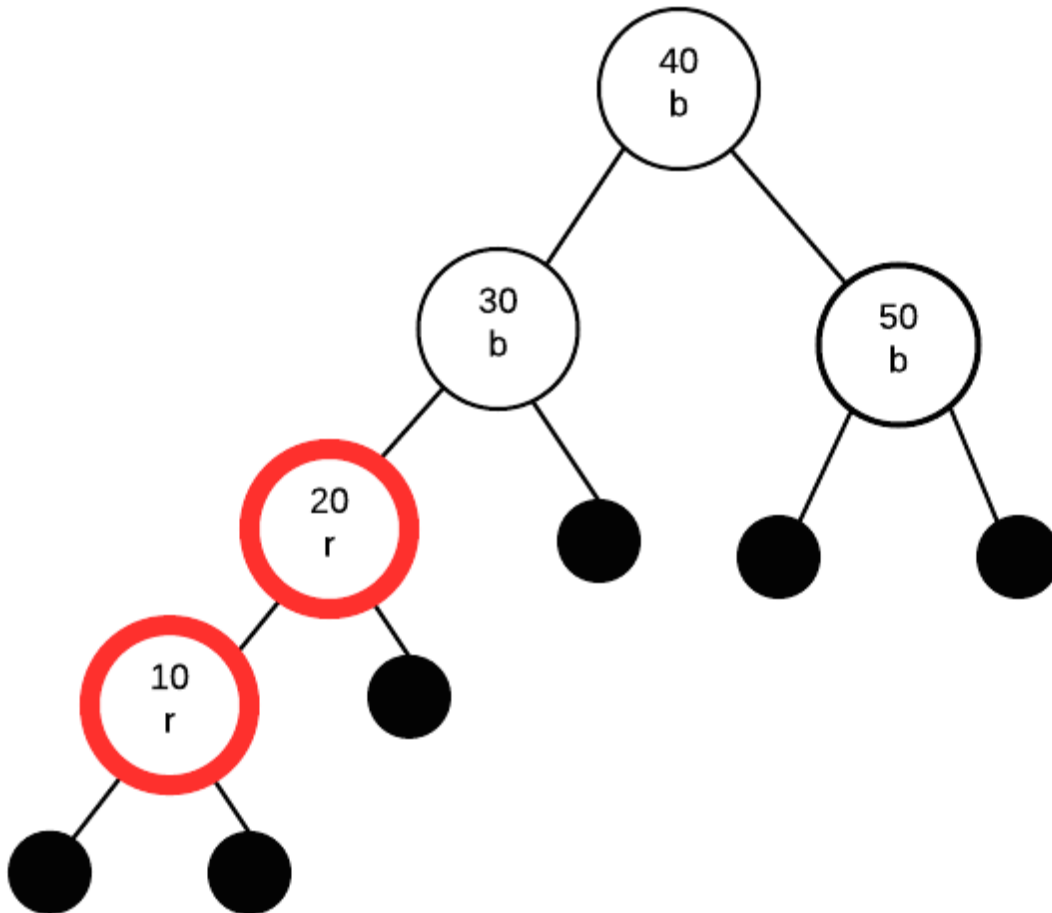
Doing so breaks rule 2: roots must be black. Thus, we need to fix that. As it is the root, we can just change it to black without causing other problems.

Insert 10

inserting 10 as a red node.

Two red nodes in a row. Identify G, P, PS and C

- **P** - parent (upper red) -20
- **C** - child (lower red) - 10
- **G** - grandparent 30
- **PS** - parent's sibling - null node to right of 30

What we do depends on colour of **PS.** In this case, the parent's sibling is black (null nodes are black). Thus, we will fix this with a rotation. Rotations are always done with G as the root of the rotation (the A in the rotation diagram)

This time, the path from G to P to C is "left" then "left". Thus, we only need to perform a single rotation, followed by swapping the colours of G and the node

that took G's spot.
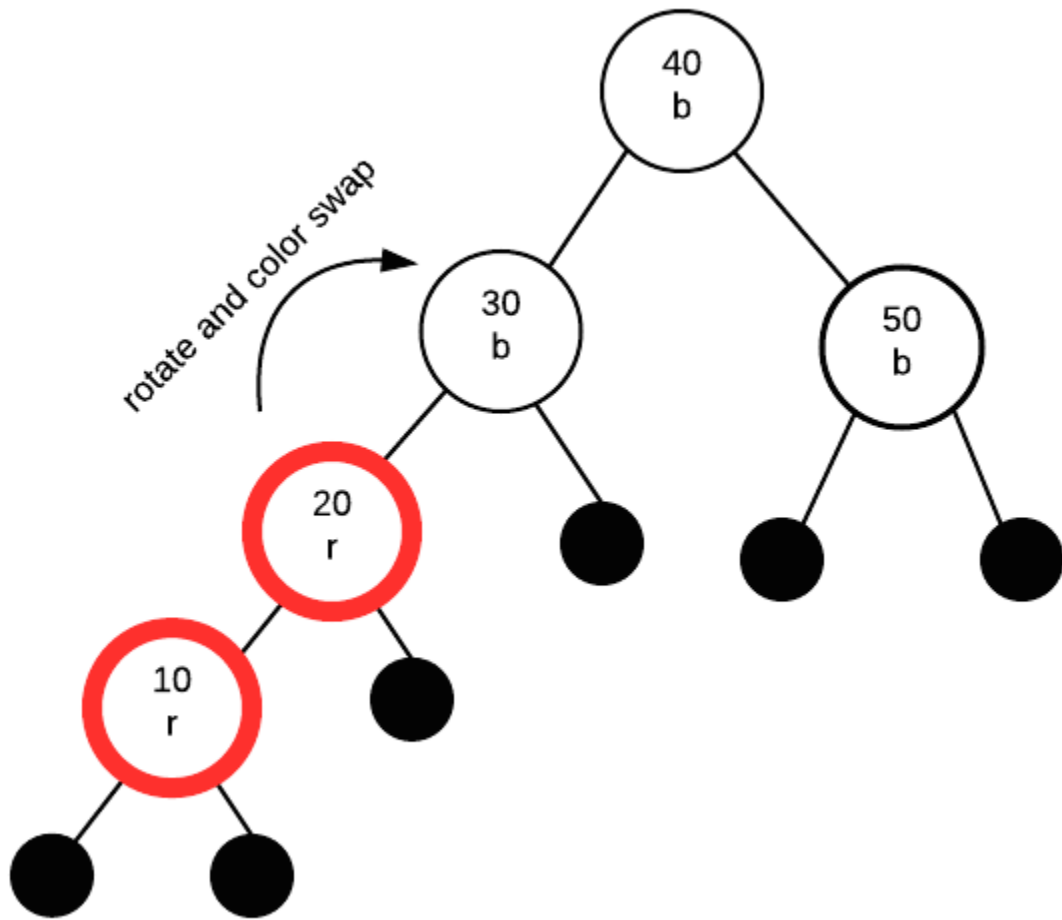
Finally we get:
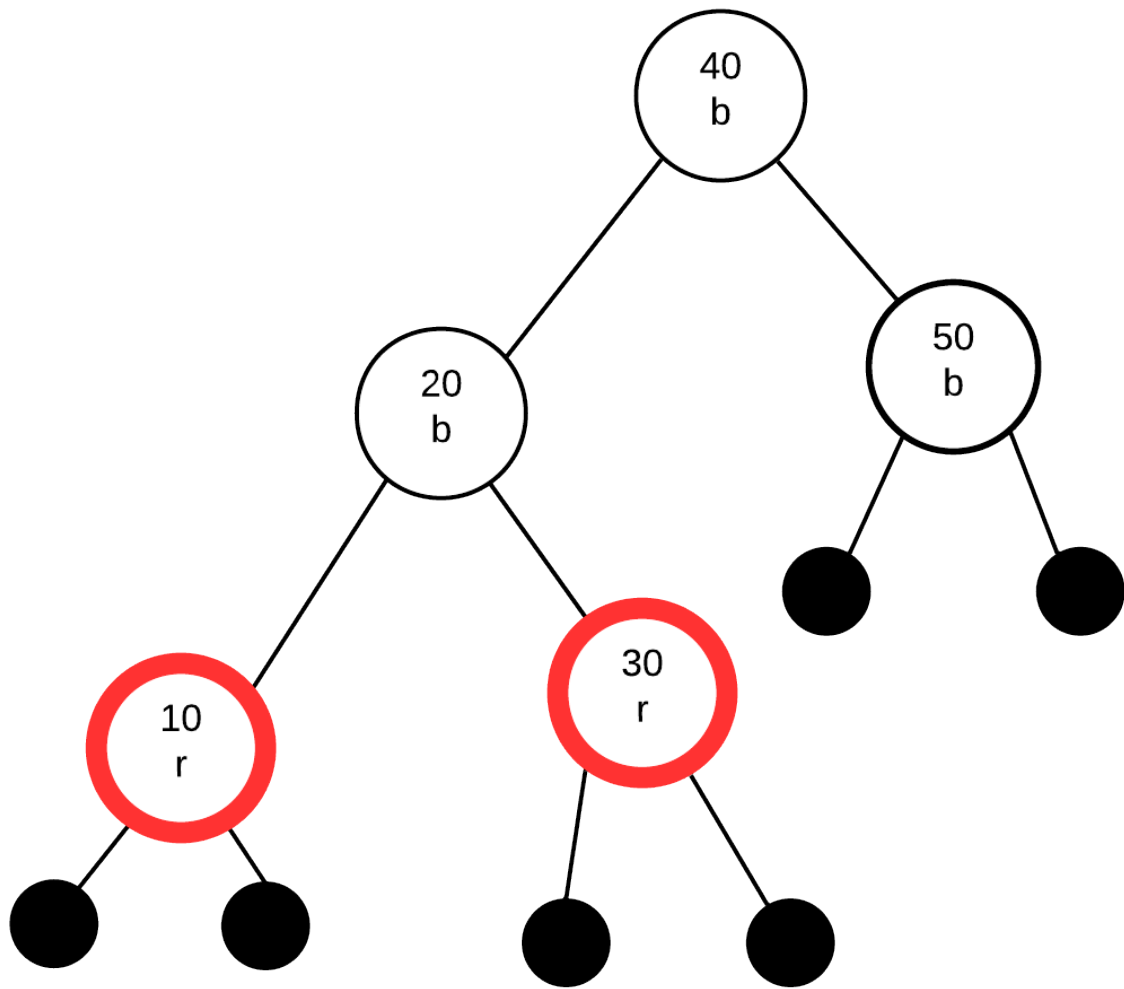
2-3 Trees

A 2-3 Tree is a specific form of a B tree. A 2-3 tree is a search tree. However, it is very different from a binary search tree.

Here are the properties of a 2-3 tree:

1. each node has either one value or two value
2. a node with one value is either a leaf node or has exactly two children (non-null). Values in left subtree < value in node < values in right subtree
3. a node with two values is either a leaf node or has exactly three children (non-null). Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
4. all leaf nodes are at the same level of the tree

Insertion

The insertion algorithm into a two-three tree is quite different from the insertion algorithm into a binary search tree. In a two-three tree, the algorithm will be as follows:

1. If the tree is empty, create a node and put value into the node
2. Otherwise find the leaf node where the value belongs.
3. If the leaf node has only one value, put the new value into the node
4. If the leaf node has more than two values, split the node and promote the median of the three values to parent.
5. If the parent then has three values, continue to split and promote, forming a new root node if necessary
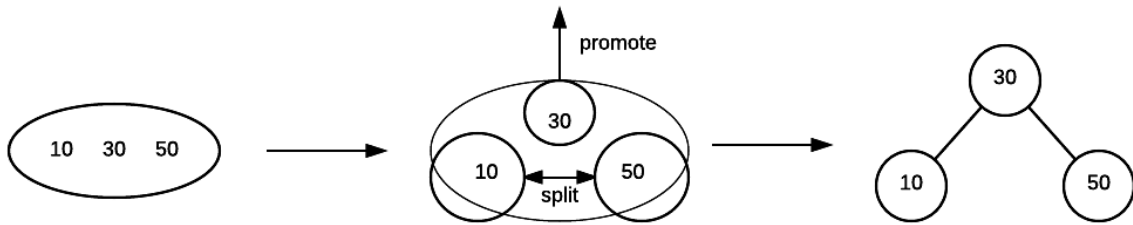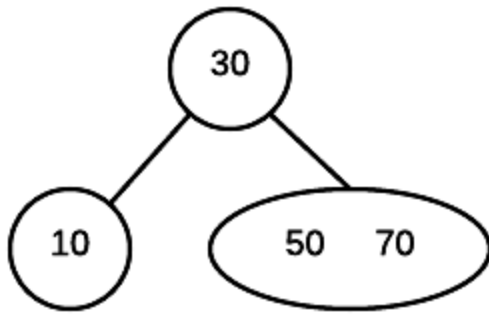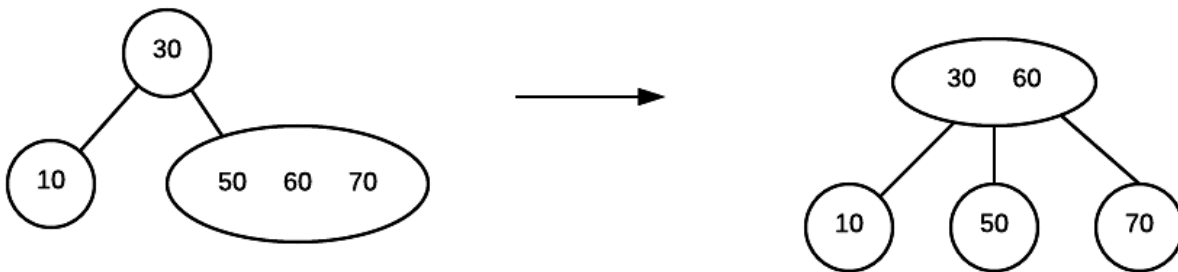
Example:

Insert 50
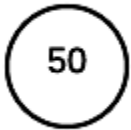
Insert 30
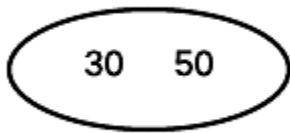
Insert 10

Insert 70

Insert 60

Complexity Theory

Introduction to Computational Theory

Complexity Theory

Over the course of this semester, we have considered many different problems, data structures and algorithms. Aside from knowing what good solutions are to common problems, it is also useful to understand the theoretical aspects of computation. This section of the notes deal with computational theory. Computational theory is actually divided into several branches. This section of the notes will focus on the branch called complexity theory which essentially classifies the difficulty of problems based on the complexity of their solution. The difficulty is based on resource requirements (such as time or space requirements).

There are many complexity classes but for our discussion we are going to be focusing on just a few of these. In particular we are going to look at the computation classes for **decision problems**. Decision problems are problems where for any given input, you will end up with a "yes" or "no" answer. These are the simplest answers.

Undecidable Problems

Some problems like the halting problems are undecidable. The halting problem is described simply as this... is it possible to write a program that will determine if any program has an infinite loop.

The answer to this question is as follows... suppose that we can write such a program. The program InfiniteCheck will do the following. It will accept as input a program. If the program it accepts gets stuck in an infinite loop it will print "program stuck" and terminate. If the program does terminate, the InfiniteCheck program will go into an infinite loop.

Now, what if we give the InfiniteCheck the program InfiniteCheck as the input for itself.

If this is the case, then if InfiniteCheck has an infinite loop, it will terminate.

If infiniteCheck terminates, it will be stuck in an infinite loop because it terminated.

Both these statements are contradictory. and thus, such a program cannot exist.

P vs NP

P class Problems

P class problems are decision problems that can be solved in polynomial time. Note that linear is polymial time, but so is quadratic... polynomial is essentially n^c where *c* is a constant. For example, matrix multiplication is a polynomial class problem even though the solution is n^2

NP class Problems

When we talk about "hard" problems then, we aren't talking about the impossible ones like the halting problem. We are instead talking about problems where its possible to find a solution, just that no good solution is

currently known.

The NP, in NP class stands for non-deterministic polynomial time. A non-deterministic machine is a machine that has a choice of what action to take after each instruction and furthermore, should one of the actions lead to a solution, it will always choose that action.

A problem is in the NP class if we can **verify** that a given positive solution to our problem is correct in polynomial time. In other words, you don't have to find the solution in polynomial time, just verify that a solution is correct in polynomial time.

Note that all problems of class P are also class NP.

NP-Complete Problems

A subset of the NP class problems is the NP-complete problems. NP-complete problems are problems where any problem in NP can be polynomially reduced to it. That is, a problem is considered to be NP-complete if it is able to provide a mapping from any NP class problem to it and back.

NP-Hard

A problem is NP-Hard if any NP problem can be mapped to it and back in polynomial time. However, the problem does not need to be NP... that is a solution does not need to be verified in polynomial time.

P vs NP

One of the Millennium Prize Problems is the problem of P vs NP. It essentially asks whether every problem that can be verified in P time is also solvable in P

time. In essence are all NP class problem (NP and NP complete) solvable in P time or not. This problem is easiest to visualize with the following diagram from wikipedia:
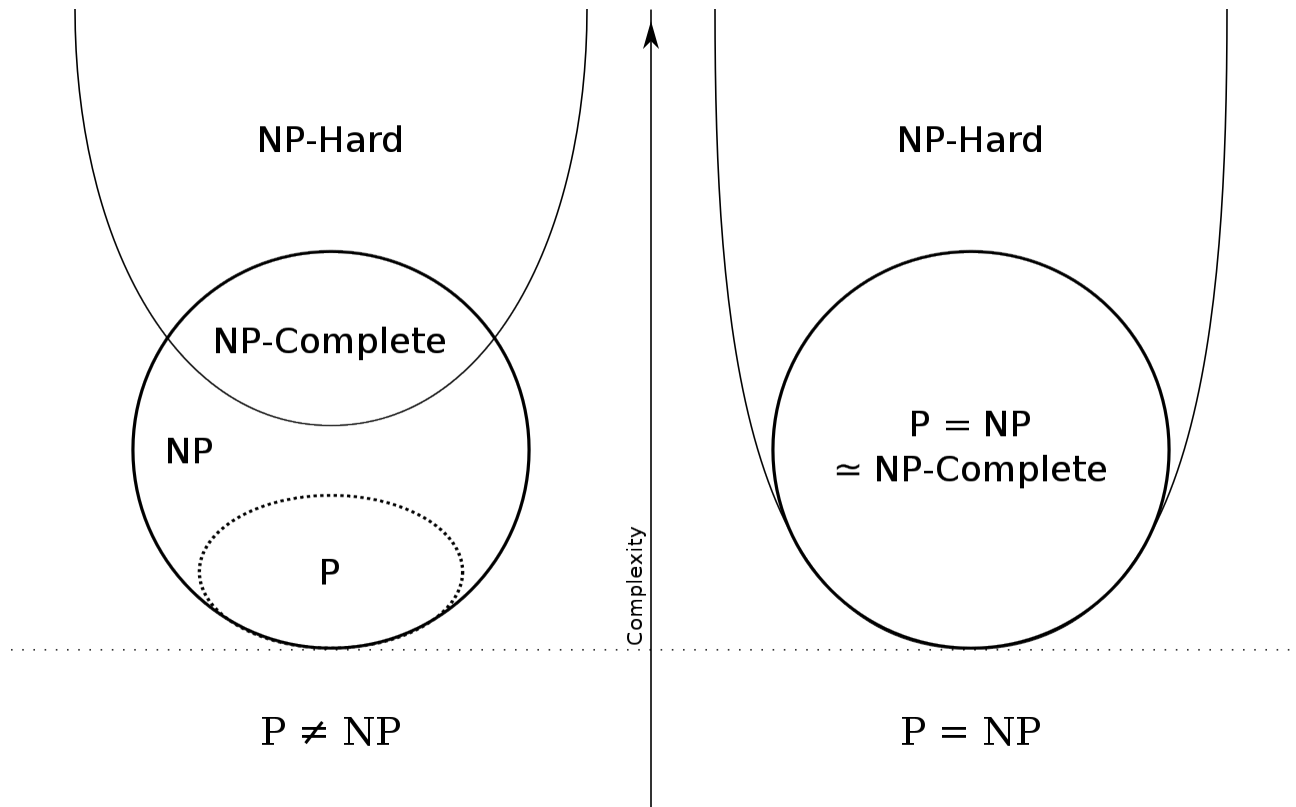
The problem essentially is asking whether the diagram on the left is true or the diagram on the right is true.

The first diagram basically means that there will some problems that are verifiable in P time but cannot be solved in P time. The second diagram means that any problem that can be verified in P time can also be solved in P time.

Appendix: Mathematics Review

A certain familiarity with certain mathematical concepts will help you when trying to analyze algorithms. This section is meant as a review for some commonly used mathematical concepts, notation, and methodology. Where possible analogies between mathematical and programming concepts are drawn

Mathematical Notations and Shorthands

Shorthands

shorthand	meaning
iff	if and only if
$\therefore$	therefore
$\approx$	approximately
ab	a * b
$a(b)$	a * b

shorthand	meaning
$\|a\|$	absolute value of a
$\lceil a \rceil$	ceiling, round aa up to next biggest whole number. Example: $\lceil 2.3 \rceil = 3$
$\lfloor a \rfloor$	floor, round aa down to the next smallest whole number. Example: $\lfloor 2.9 \rfloor = 2$

Variables

In math, like programming, we use variables. Variables can take on some numeric value and we use it as a short hand in a mathematical expression. Before using a variable, you should define what it means (like declaring a variable in a program)

For example:

"Let **n** represent the size of an array"

This means that the variable n is a shorthand for the size of an array in later statements.

Functions

Similar to functions in programming, mathematics have a notation for functions. Mathematically speaking, a function has a single result for a given set of arguments. When writing out mathematical proof, we need to use the language of math which has its own syntax

As a function works with some argument, we first define what the arguments mean then what the function represents.

For example:

Let n represent the size of the array - (n is the name of the argument).

Let $T(n)$ represent the number of operations needed to sort the array - T is the name of the function, and it accepts a single variable n

We pronounce $T(n)$ as "T at n". Later we will assoicate $T(n)$ with a mathematical expression that we can use to make some calculation. The expression will be a mathematical statement that can be used to calculate the number of operations needed to sort the array. If we supply the number 5, then $T(5)$ would be the number of operations needed to sort an array of size 5

Summary

$T(n)$ - read it as **T at n**, we call the function T .

$T(n) = n^2 + n + 2$ means that $T(n)$ is the same as the mathematical expression $n^2 + n + 2$. Think of $T(n)$ as being like the function prototype, and $n^2 + n + 2$ as being like the function definition

n can take on any value (unless there are stated limitations) and result of a function given a specific value is calculated simply by replacing n with the value

$T(5) = 5^2 + 5 + 2 = 32$ (we pronounce T(5) as "T at 5")

{% hint style="danger" %} When we talk about big-O notation (and related little-o, theta and omega notation) those are NOT functions. For example O(n) is NOT a function named O that takes a variable n. It's meaning is different. {% endhint %}

Sigma Notation

Sigma notation is a shorthand for showing a sum. It is similar in nature to a for loop in programming.

General summation notation.

$$\sum_{i=1}^{n} t_i = t_1 + t_2 + ... + t_n$$

The above notation means that there are n terms and the summation notation adds each of them together.

Typically the terms t_i, is some sort of mathematical expression in terms of i (think of it as a calculation you make with the loop counter). The i is replaced with every value from the initial value of 1 (at the bottom of the $\sum$), going up by 1, to n (the value at the top of the $\sum$)

Example:

$$\sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5$$

Mathematical Definitions and Identities

Mathematical identities are expressions that are equivalent to each other. Thus, if you have a particular term, you can replace it with its mathematical identity.

Exponents

Definition

x^n means $(x)(x)(x)...(x)$ (n x's multiplied together)

Identities

$x^a x^b = x^{a+b}$

$\frac{x^a}{x^b} = x^{a-b}$

$(x^a)^b = x^{ab}$

$x^a + x^a = 2x^a \neq x^{2a}$

$2^a + 2^a = 2(2^a) = 2^{a+1}$

Logarithms

::: tip

In computer text books, unless otherwise stated log means log_2 as opposed to log_{10} like math text books

:::

Definition

$b^n = a$ iff $log_b a = n$ In otherwords $log_b a$ is the exponent you need to raise b by in order to get a

Identities

$$\log_b a = \frac{\log_c a}{\log_c b} \text{ , where } c > 0$$

$$\log ab = \log a + \log b$$

$$\log\left(\frac{a}{b}\right) = \log a - \log b$$

$$\log a^b = b\log a$$

$$\log x < x \text{ for all } x > 0$$

$$\log 1 = 0$$

$$\log 2 = 1$$

Series

A series is the sum of a sequence of values. We usually express this using sigma notation (see above).

Identities

$$\sum_{i=0}^{n} c(f(i)) = c\sum_{i=0}^{n} f(i) \text{ , where } c \text{ is a constant}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$$

$$\sum_{i=0}^{n} a^i \leq \frac{1}{a-1}$$

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

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

$$\sum_{i=1}^{n} f(n) = nf(n)$$

$$\sum_{i=n_0}^{n} f(i) = \sum_{i=1}^{n} f(i) - \sum_{i=1}^{n_0-1} f(i)$$