# Table of Contents

# Introduction

# Data Structures and Algorithms

## Introduction

Data structures and algorithms looks at how data for computer programs can best be represented and processed. This book is a survey of several standard algorithms and data structures. It will also introduce the methodology used to perform a formal analysis of an algorithm so that the reason behind the different implementations can be better understood.

This book is not an introductory programming book. C/C++ will be used as the language for examples. However, there will not be much of a discussion about C/C++ syntax if at all. If there is a discussion of syntax, it will be in the context of data structures and algorithms. Although the language used in this book for the coding examples is C++, you can just as easily use other languages to implement the algorithms and data structures introduced here. Remember to focus on the algorithms and data structures itself as opposed to the syntax and language details.

The contents of this book is meant as an introduction to data structures and algorithms. There are many books out there that will do a far better job of formal analysis than this one and go more in depth with different implementations and I highly recommend that you look at other books.

Finally, this book serves as the subject notes for the Data Structures and Algorithms course at Seneca College (Toronto, Ontario, Canada). As such, there will be many code samples that are completed in class as part of the course work and not included here at this time. This may change in the future.

# Introduction to Algorithms Analysis

## Introduction to 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. 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 growh rates of certain functions.

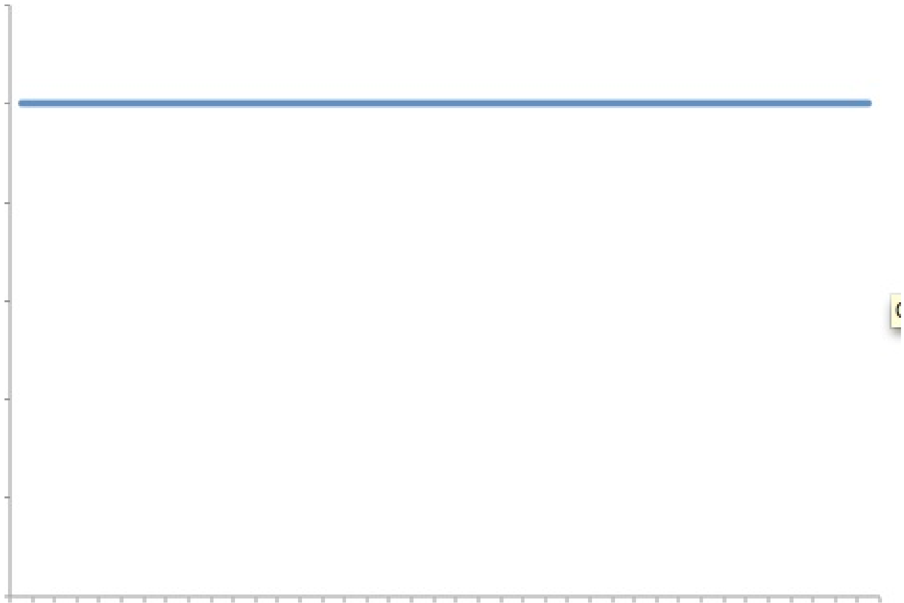
# Growth Rates

## Growth Rates

Algorithms analysis is all about understanding growth rates. That is as the amount of data gets bigger, how much more resource will my algorithm require? Typically, we describe the resource growth rate of a piece of code in terms of a function. To help understand the implications, this section will look at graphs for different growth rates from most efficent to least efficient.

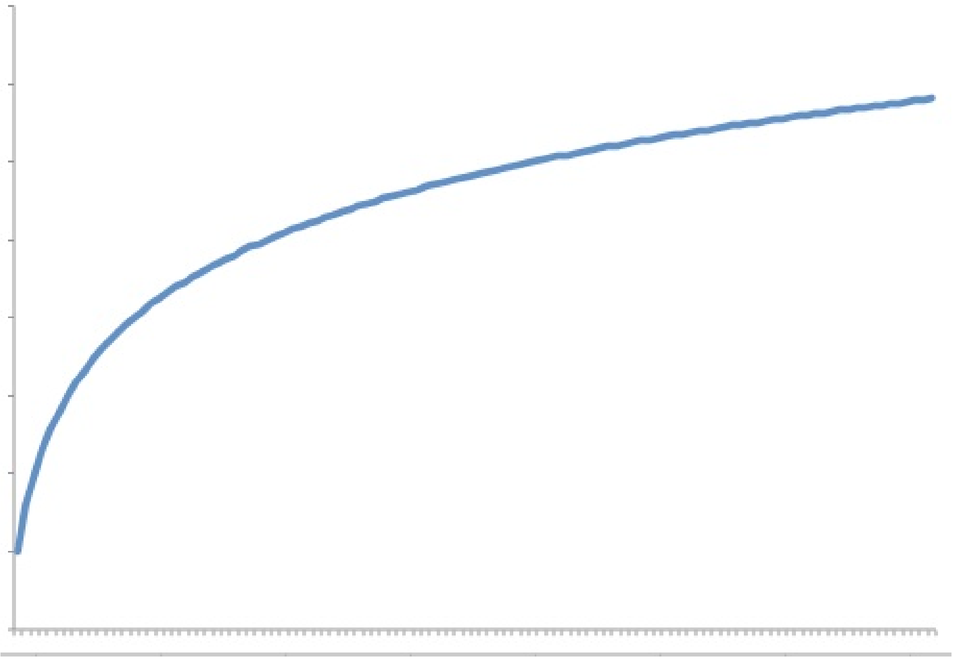
## Constant Growth Rate

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



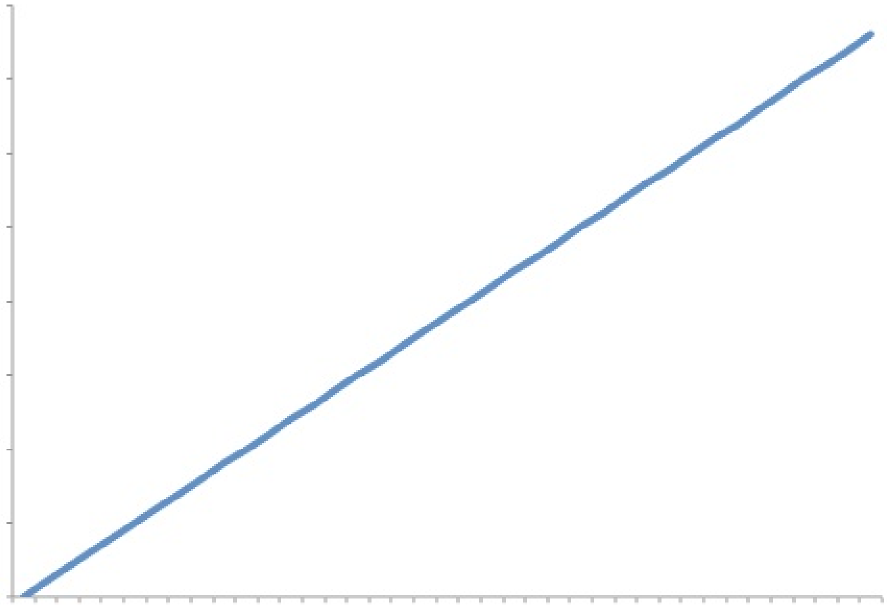
## Logrithmic Growth Rate

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.



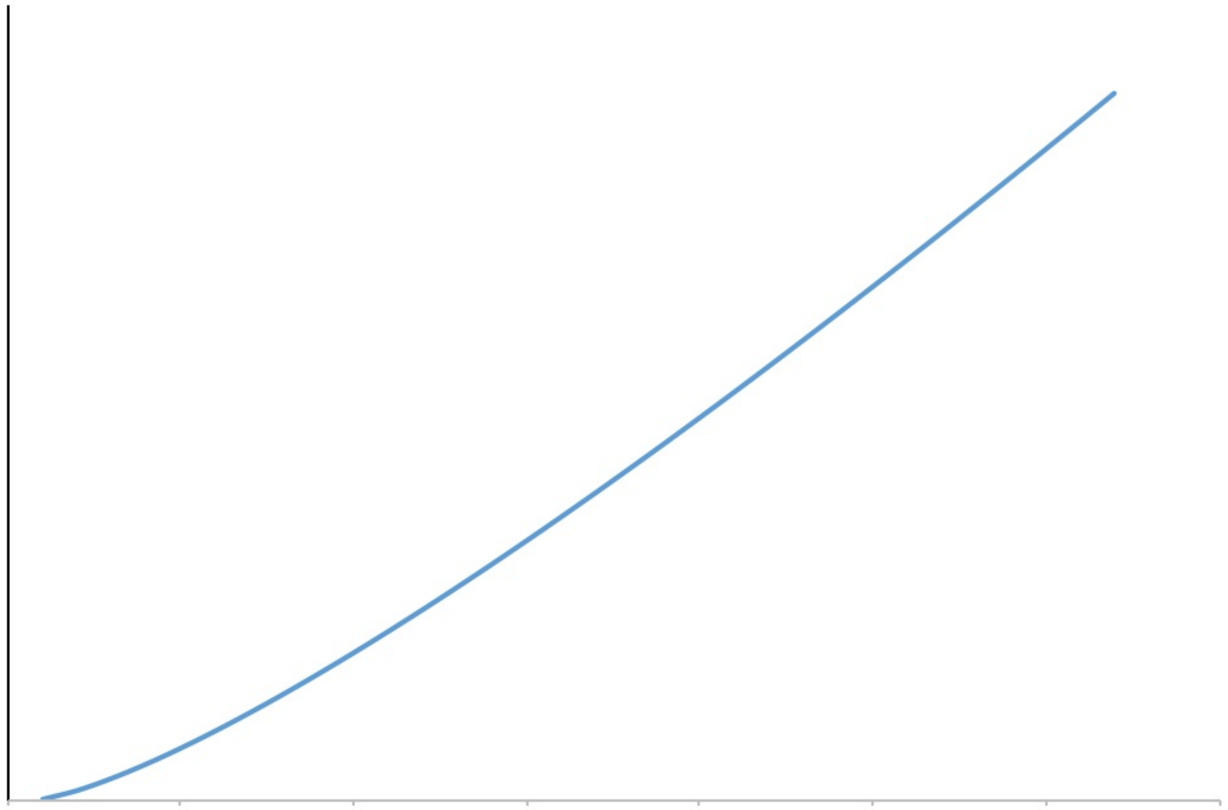
## Linear Growth Rate

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.



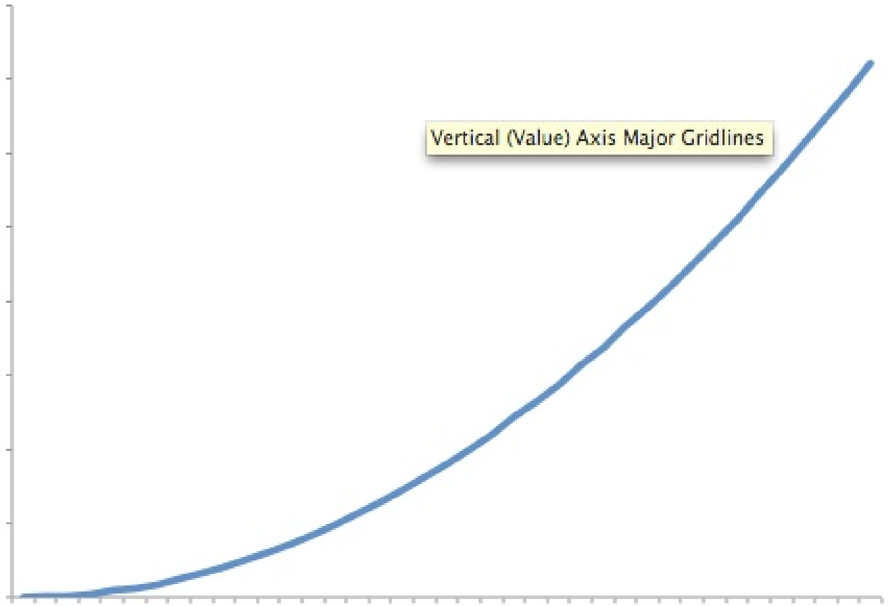
## Log Linear

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



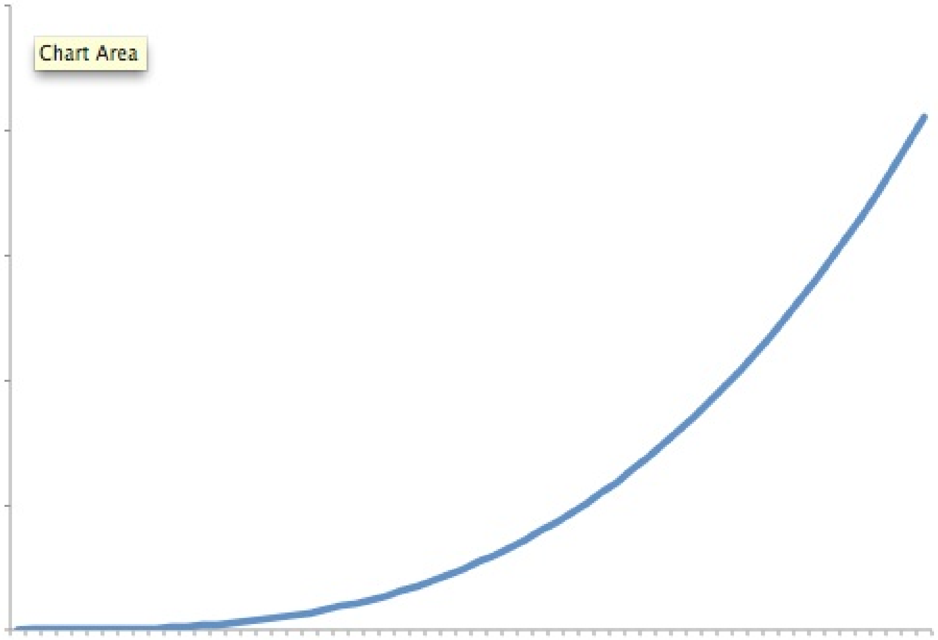
## Quadratic Growth Rate

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



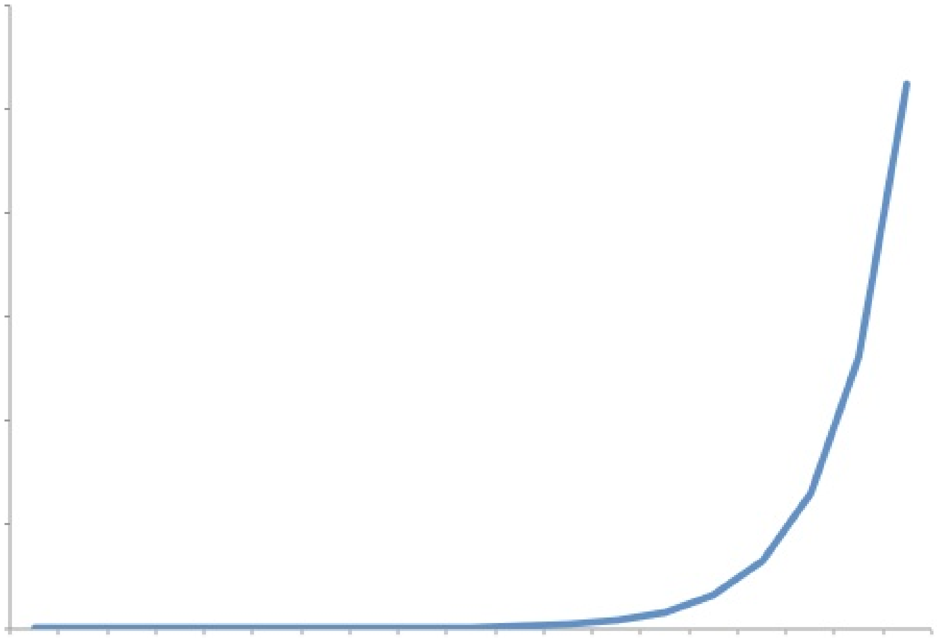
## Cubic Growth Rate

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



## Exponential Growth Rate

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)



# Big-O, Little-o, Theta, Omega

## Big-O, Little-O, Theta, Omega

Big-O, Little-o, Omega, and Theta are formal notational methods for stating the growth of resource needs (efficiency and storage) of an algorithm. There are four basic 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)

### Formally:

"$T(n)$ is $O(f(n))$" iff for some constants $c$ and $n_0$, $T(n) <= cf(n)$ for all $n >= n_0$

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

"$T(n)$ 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 $\Theta(f(n))$

### Informally:

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

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

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

"$T(n)$ is $o(f(n))$" basically means that $f(n)$ is the upper bound for $T(n)$ but that $T(n)$ can never be equal to $f(n)$

#### Another way of saying this:

"$T(n)$ is $O(f(n))$" growth rate of $T(n)$ <= growth rate of $f(n)$

"$T(n)$ is $\Omega(f(n))$" growth rate of $T(n)$ >= growth rate of $f(n)$

"$T(n)$ is $\Theta(f(n))$" growth rate of $T(n)$ == growth rate of $f(n)$

"$T(n)$ is $o(f(n))$" growth rate of $T(n)$ < growth rate of $f(n)$

### An easy way to think about big-O

The math in big-O analysis can often be intimidates students. One of the simplest ways to think about big-O analysis is that it is basically a way to apply a rating system for your algorithms (like movie ratings). It tells you the kind of resource needs you can expect the algorithm to exhibit as your data gets bigger and bigger. From best (least resource requirements ) to worst, the rankings are: $O(1)$,

$O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$. Think about the graphs in the grow rate section. The way each curve looks. That is the most important thing to understand about algorithms analysis

## What all this means

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

"$T(n)$ is $O(f(n))$" if for some constants $c$ and $n_0$, $T(n) <= cf(n)$ for all $n >= n_0$

The way to read the above statement is as follows.

- n is the size of the data set.
- $f(n)$ is a function that is calculated using n as the parameter.
- $O(f(n))$ means that the curve described by $f(n)$ is an upper bound for the resource needs of a function.

This means that if we were to draw a graph of the resource needs of a particular algorithm, it would fall under the curve described by $f(n)$. What's more, it doesn't need to be under the exact curve described by $f(n)$. It could be under a constant *scaled* curve for $f(n)$... so instead of having to be under the $n^2$ curve, it can be under the $10n^2$ curve or 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 constant is. The actual value of the constant does not matter though.

The other portion of the statement $n >= n_0$ means that $T(n) <= cf(n)$ does not need to be true for all values of $n$. It means that as long as you can find a value $n_0$ for which $T(n) <= cf(n)$ is true, and it never becomes untrue for all $n$ larger than $n_0$, then you have met the criteria for the statement $T(n)$ is $O(f(n))$

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.

# Analysis of Linear Search

## How to do an Analysis

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

```
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;
}
```

We will make a few assumptions. arr.size() runs in constant time. That is no matter how big the array is, arr.size() takes the same amount of time to run.

When we run the above algorithm, 2 things can occur. The first is that we will find the key. The second is that we won't. The worst case scenario occurs when key is not in the array. Thus, let us start by performing the analysis base on that worst case.

### 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.

Looking at the code, we see that there are some operations that we have to perform one time no matter what:

~~~
    int rc = -1
    int i = 0
    return rc;
````
So here, we can count these as 3 operations that have to occur at lea

Now we then look at the rest of the code, if our search was to fail 

The following has to be done each time through the loop:

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

```
~~~

6 operations in the loop total

Now... some of you may also think... should the function call arr.size() be an operation itself? or what about the [i] in arr[i]? The truth is, it doesn't actually matter if we count them or not. Let's take a look at why this is the case by finishing this analysis.

Now... using the above we can express $T(n)$ as follows:

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

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 number 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.

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

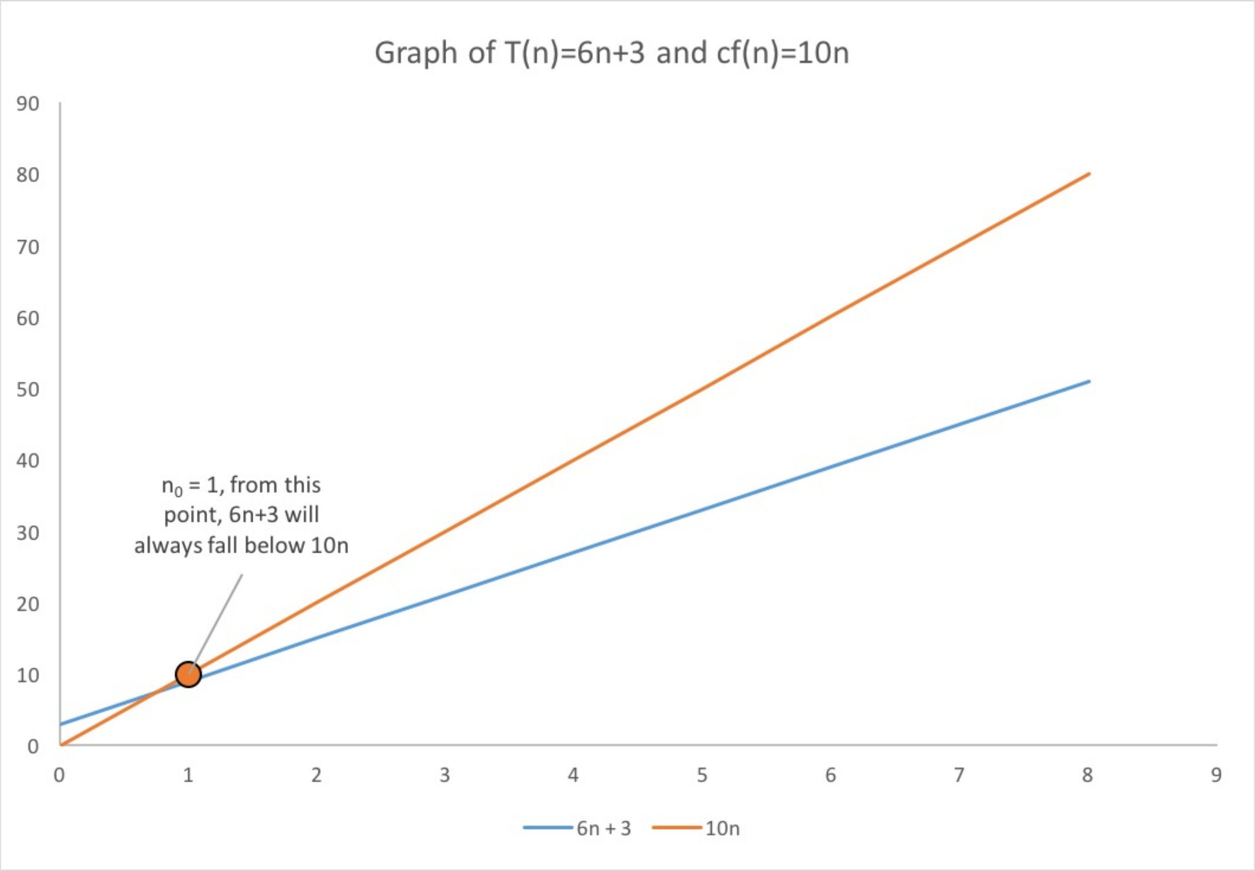
"$T(n)$ is $O(f(n))$" if for some constants $c$ and $n_0$, $T(n) <= cf(n)$ for all $n >= n_0$

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 $T(n) <= cn$ is true for all $n >= n_0$

The important part about this... is to realize that we can pick any constant we want. Therefore, I will pick any value greater than 6 for $c$. 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) $c = 10$ and 1 for $n_0$, therefore $n_0 = 1$

In fact for c, I can pick any number that is bigger than 6 and for $n_0$ any number greater than 6/7

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



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 size() function call or the [i] operator. If we counted both of them,

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

The dominating term would still be 8n.

The proof would go exactly the same way except that we can't use $n_0 = 1$ as the statement $T(n) < cn$ 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 $n > 2$.

## Average case analysis

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.

# Analysis of Binary Search

# Analysis of Binary Search

The following is the code for a binary search. Similar to linear search, we make an assumption that the size() function has a constant run time. For a binary search to work the data must be sorted. In this case we assume that the data is sorted from smallest (at arr[0]) to biggest (at arr[size-1]).

The second part that is important to remember about a binary search is that you need to be able to access any item given its index in constant time. If that is not possible, the analysis will fail.

```
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(arr[mid] > key)
      high=mid-1;
    else if(arr[mid] < key)
      low= mid+1;
    else
      rc=mid;
  }/*while*/
  return rc;
}
```

Like a Linear search, the determining factor on runtime for binary search is also the loop:

```
  while(low<=high && rc==-1){
    mid=(low+high)/2;
    if(arr[mid] > key)
      high=mid-1;
    else if(arr[mid] < key)
      low= mid+1;
    else
      rc=mid;
  }/*while*/
```

The code within this loop is constant, meaning that no matter what size is one iteration of the loop will take the same number of operations (more or less) and thus, the question really becomes how many times will this loop run?

Again, we can either find the key or not find the key.

If the vector does NOT contain the key, how long will it take to run?

At the beginning high - low = n -1. With each iteration we "move" high or low towards each other so that their difference is halved (their new values are near the mid point)

By doing this, it would take the loop at most log n iterations before low>high

Thus for an unsuccessful search, the function's runtime $O(logn)$

For a successful search we might still need to search when high==low and to get to that point would also require iterations. Thus, the worst case runtime for this function when the key is found is also O(log n)

Therefore, the worst case runtime for this function is $O(logn)$

# Recursion

## Recursion

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 runtime stack

# The runtime 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

# How to Write a Recursive Function

# How to Write a Recrusive Function

Recursive functions are sometimes hard to write because we are not use to thinking about problems recursively. However, there are two things you should do:

1. state the base case (aka easy case). what argument value will be the simplest case that you can possibly have for your problem? what should the result be given this simplest case
2. state the recursive case. if you are given something other than the simplest case how can you simplify it to head towards the simplest case?

# Example: the Factorial Function

# Example: the Factorial Function

Write the function:

```
int factorial(int n);
```

This function returns n! (read n factorial) where n is an integer.

```
n!=n* n-1* n-2*....2*1  by definition 0! is 1
```

for example

```
if n==5, then n! would be 5! = 5*4*3*2*1=120
```

To write this we must come up with several things. What is the base case? In other words, for what value of n do I immediately know what the answer would be without doing more than a simple operation or two.

In this case, we know what 0! is. It is 1 by definition 1! is another base case because 1! is simply 1 as well. However, it is not actually necessary to explicitly state the base case for when n is 1 as we can further reduce that to the 0! base case

So the base case occurs when n is 0 or 1. In this case, the function can simply return 1

Now the recursive case. How can we state the solution in terms of itself. First thing you should notice is that:

```
5!=5 * 4 * 3 * 2 * 1 but 4*3*2*1 is really 4!
So:

5! = 5* 4!  but 4! is just 4* 3!  and so on.
```

Thus if I had a function that can give me the factorial of any number I can use it to find the factorial of that number-1 and thus allowing me to calculate the factorial of the original by multiplying that result with number. In other words I can use the int factorial(int) function to solve int factorial(int)

```
int factorial(int n){
    int rc;              //stores result of function
    if(n==1 || n==0){    //check for base case
        rc=1;            //if n is 1 or 0 rc is 1
    }
    else{                //else it is recursive case
        rc=n * factorial(n-1);  //rc is n * (n-1)!
    }
    return rc;
}
```

### Why does this work?

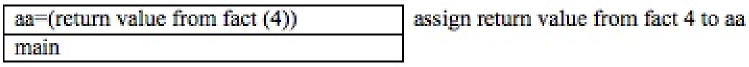
Example: the Factorial Function

To understand why recursion works, we need to look at the behavior of the run time stack as we make the function calls:

Suppose we have the following program:

```
int fact(int n){
    int rc;              //stores result of function
    if(n==1 || n==0){    //check for base case
        rc=1;            //if n is 1 or 0 rc is 1
    }
    else{                //else it is recursive case
        rc=n * fact(n-1);  //rc is n * (n-1)!
    }
    return rc;
}
int main(void){
   int aa = fact(4);
}
```

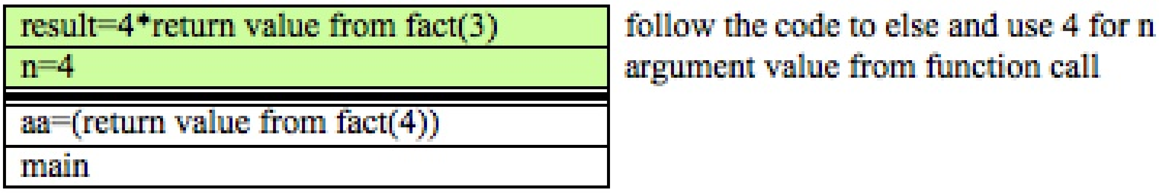
When the program starts this is our run time stack:



aa=(return value from fact (4))
main

assign return value from fact 4 to aa

fact(4) means that we call function fact() with 4 as argument so push that information onto the stack

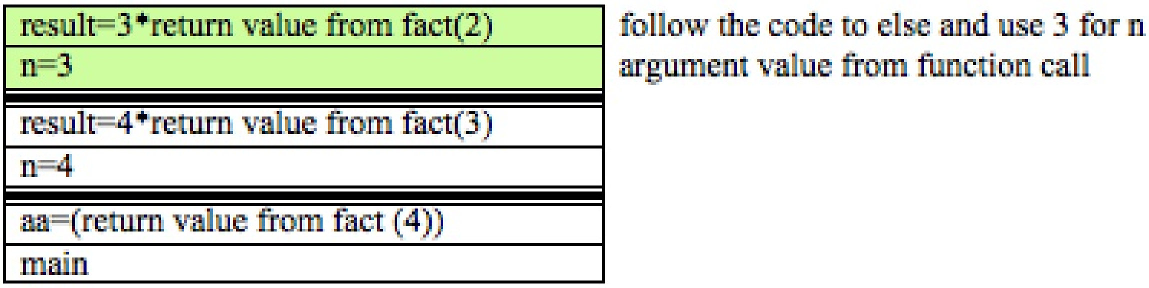


result=4*return value from fact(3)
n=4

aa=(return value from fact(4))
main

follow the code to else and use 4 for n argument value from function call

Since we make a function call to fact(3) we must now push that information onto the stack also. Note, return statement not reached before calling fact(3) so stack isn't popped at this point



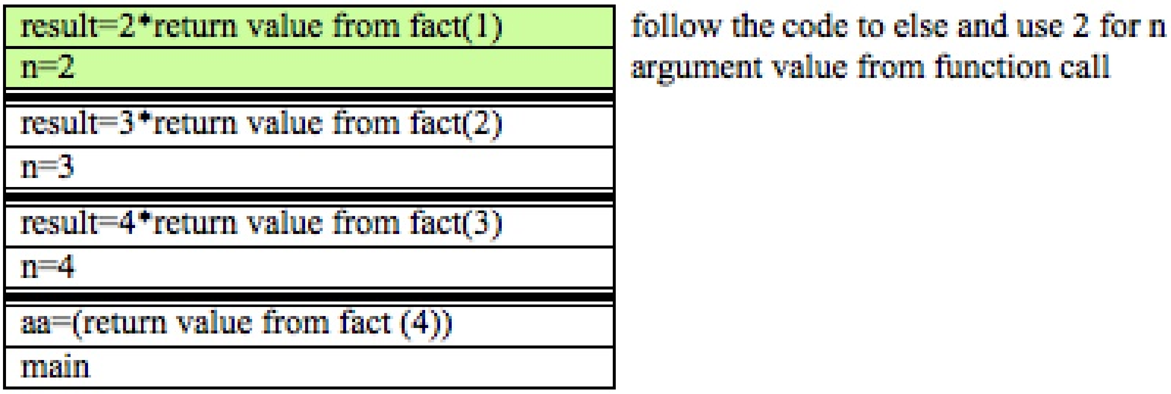
result=3*return value from fact(2)
n=3

result=4*return value from fact(3)
n=4

aa=(return value from fact (4))
main

follow the code to else and use 3 for n argument value from function call

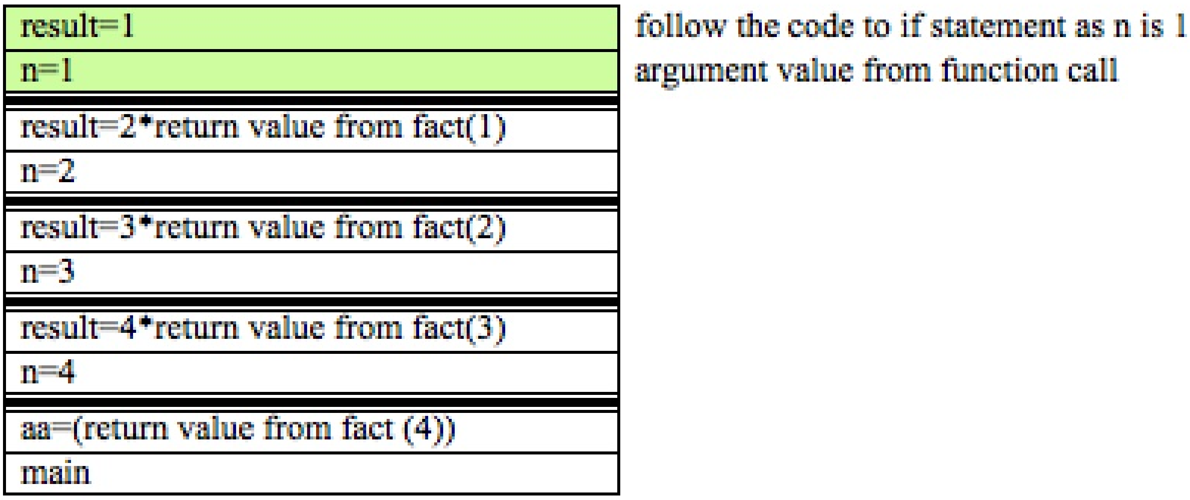
Again, because we are calling the function with fact(2) , we must also push this onto the stack



result=2*return value from fact(1)
n=2
result=3*return value from fact(2)
n=3
result=4*return value from fact(3)
n=4
aa=(return value from fact (4))
main

follow the code to else and use 2 for n
argument value from function call

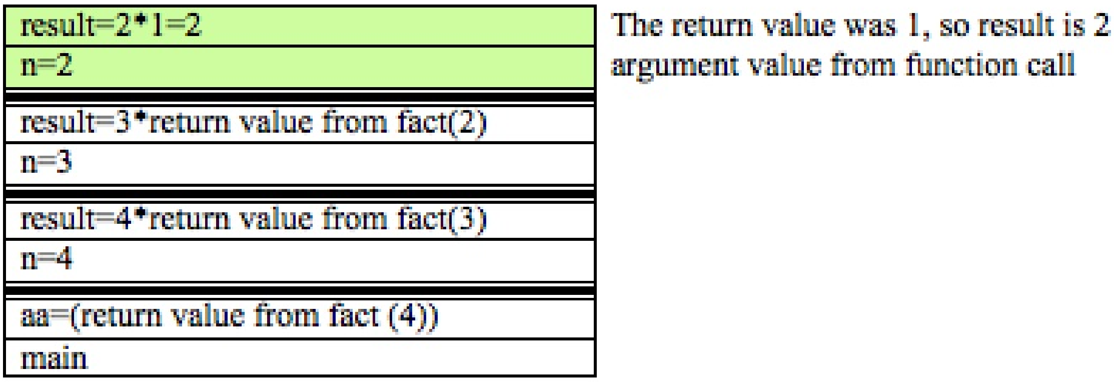
Once again, because we are calling the function with fact(1) we must also push this onto the stack



result=1
n=1
result=2*return value from fact(1)
n=2
result=3*return value from fact(2)
n=3
result=4*return value from fact(3)
n=4
aa=(return value from fact (4))
main

follow the code to if statement as n is 1
argument value from function call

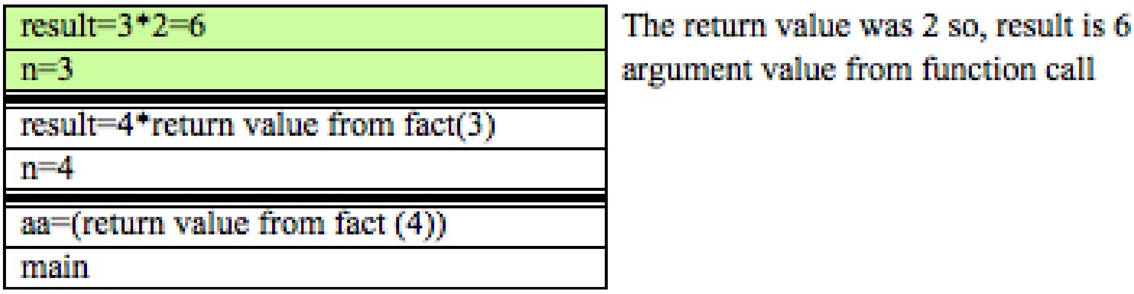
If we follow the code at this point we can see that we are able to reach the return statement without further function calls. Thus, we can now pop the stack.



result=2*1=2
n=2
result=3*return value from fact(2)
n=3
result=4*return value from fact(3)
n=4
aa=(return value from fact (4))
main

The return value was 1, so result is 2
argument value from function call

This will lead to the completion of the topmost function call and again, it can be removed from the stack. This time, the return value is used to solve the problem one more layer above:



result=3*2=6

n=3

result=4*return value from fact(3)

n=4

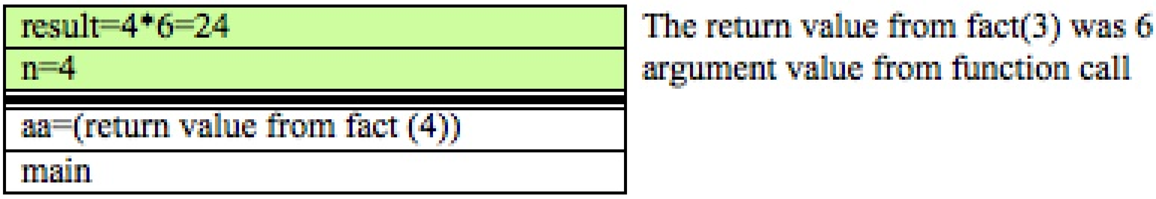
aa=(return value from fact (4))

main

The return value was 2 so, result is 6

argument value from function call

This will lead to the completion of the topmost function call and again, it can be removed from the stack. This time, the return value is used to solve the problem one more layer above:



result=4*6=24

n=4

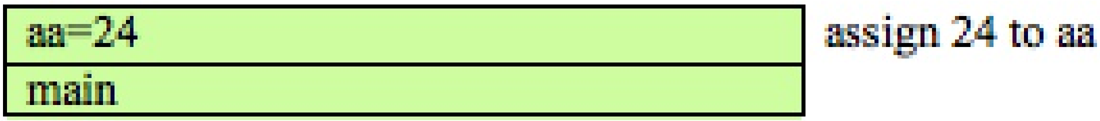
aa=(return value from fact (4))

main

The return value from fact(3) was 6

argument value from function call

Finally, we can go back to main, as we have reached the final result:



aa=24

main

assign 24 to aa

# Drawbacks of Recursion and Caution

# 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. .

# Lists

# Lists

A list is an **ordered 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. 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.

If you used a fixed array you will need to state an initial size. Items are stored in memory consecutively and you can have direct access to any particular item in constant time through the use of its index.

# Implementation

# 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. There are advantages and disadvantages to each implementation.

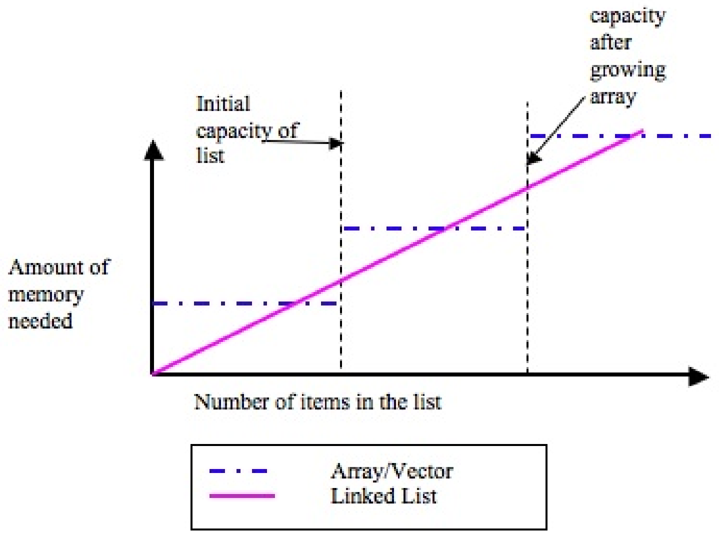
If you used array, items are stored in memory consecutively and you can have direct access to any particular item through the use of its index in constant time. When sorted, the list can be searched using binary search. Making the list grow can be expensive and space is often wasted as large amount of space may be allocated but not used. 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.

A linked list is very easy to grow and shrink. 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. Each piece of data requires the storage of an extra pointer. However, the amount of extra data is related to number of items in the list already. A linked list cannot be searched using binary search as direct access to nodes are not available. However, both insertion and removal of any node in the list (assuming that the position of the insertion/removal is known) is very efficient and runs in constant time.

## 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. As the process of growing an array requires the typically copying of the entire array, this is not something you want to do often. That is, you do not grow the array a few elements at a time but in large chunks instead. The exact number of elements to grow by is implementation specific.

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.


capacity
after
growing
array
Initial
capacity of
list
Amount of
memory
needed
Number of items in the list
Array/Vector
Linked List

# Linked List

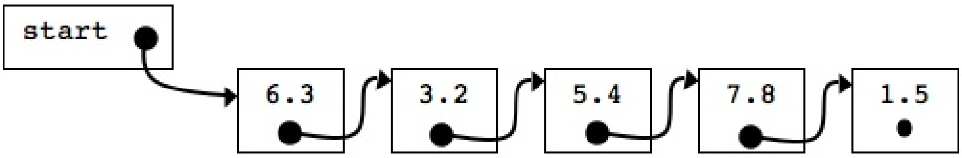
## Linked List

The linked list data structure is made up of multiple nodes. Each node is made up of two portions, a data portion and a pointer portion. The data portion contains one instance of the data to be stored. While the pointer portion indicates the location of another node.

An array of 5 doubles:

| | | | | |
|---|---|---|---|---|
| 6.3 | 3.2 | 5.4 | 7.8 | 1.5 |

A linked list storing the same data as the above array:



Each linked list must have a pointer to some node in the list. Any other node in the list is reached by following the pointers from one node to the next.

Like an array, a linked list stores data of the same data type. The data type of a linked list will determine how each node is declared.

# Nodes

## 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.

The following shows how to declare a simple node for storing doubles.

```
struct Node{
  double data_;    // the data portion of the node
  Node* next_;     // a pointer to the next node
};
```

To create linked lists that hold other data types the data portion of a node would need to be a different data types.

```
class Hamster
  char name_[50];
  int age_;
public:
....
};

//Each node holds one instance of Hamster.
struct Node{
  Hamster data_;
  Node* next_;
};
```

Other data structures such as trees, also store data in nodes. If you wish to create a library of data structures and want to avoid naming conflicts, you can nest the Node declaration within your linked list class. This also allows you to use a struct of a node and not worry about access permissions

# Iterator

## Iterator

An iterator is an object that allows you to traverse a list. You can observe their usage in container classes within the C++ Standard Library such as &ltvector&gt and &ltlist&gt. A Linked list is a container class. Thus, our implementation of a linked list should also include the concept iterators.

Like Nodes, other container objects could also have an iterator. To avoid naming conflict, iterators can be nested within the list class.

There are two types of iterator. An iterator and a const_iterator. An iterator allows changes to the objects being referred to while a const_iterator does not. Thus, we need both.

Once our list is created, we will be able to go through the list using iterator objects.

For the list class, iterators should have the following

- operators to advance to the next piece of data in the linked list. (++) If the list is doubly linked, operators to advance to the previous piece of data (--)
- The dereference operator (*) to access the data stored.
- operators to compare two iterators to see if they are pointing at the same data object (not just the same data value, but the same actual instance of the object)
- assignment operator

When thinking about iterators, the important concept is that an iterator lets us go through our list one data item at a time. It is similar in nature to the loop counter that we use to go through an array. While you may view it as being similar to a node pointer, it is not the same. To the user of the list, there is no such thing as a node. They only have iterators. This is really important to remember.

# Template Singly Linked List

# List Declaration

In this section we will look at how to implement a template of a singly linked list.

```
template <class T>
class SList{
    //nesting to avoid naming conflicts with other classes that have
    //nodes
    struct Node{
    ...
    };
public:
    //nesting iterators to avoid naming conflicts, these are public
    //as we want non-members to be able to create and use them.
    class const_iterator{
    ...
    };
    class iterator: public const_iterator{
    ...
    };
    ...
};
```

# Data Members

This section will go through the type of data each of the classes must store.

## SList

The list itself must store minimally a pointer to the first node. Sometimes it is also a good idea to have a pointer to the last node in the linked list as it will make it more efficient for some functions.

## Node

A node must contain an instance of the unknown data type. As our template is a singly linked list, our nodes must also have a pointer for the next node.

## Iterators

The iterators are the public objects that allows us to access the data within the linked list. The data needed to implement the iterator is Node* so that it is possible to record which node the iterator is referring to. Furthermore, if you wish to have more reliable code, the iterator will also need to store the address of the list the node belong's to. Otherwise, it will be possible use an iterator that refers to a node from a different list.

The const_iterator is the base class object, you will only need to store the pointer in the const_iterator class and not the iterator class.

```
template <class T>
class SList{
    //nesting to avoid naming conflicts with other classes that have
    //nodes
    struct Node{
        T data_;
        Node* next_;
        ...
    };
    Node* first_;  //pointer to first node in linked list
    Node* last_;   //pointer to last node in linked list
public:
    //nesting iterators to avoid naming conflicts, these are public
    //as we want non-members to be able to create and use them.
    class const_iterator{
        Node* curr_;
    ...
    };
    class iterator: public const_iterator{
    ...
    };
    ...
};
```

# Functions/Operators

## SList Constructor

The SList constructor creates an empty linked list. It takes no arguments

## Node Constructor

To simplify the creation of a node, it is best to have a constructor for the node class. The node constructor takes a const reference to an instance of the unknown data type, and a pointer to the next node.

These arguments should be given default value of empty object and null pointer

## Iterator constructors

The public iterator constructors should take no arguments and initialize the iterator to point at nothing.

The iterator's private constructor should take a pointer to a node to set the address of the node the iterator needs to point to.

## Functions for Manipulating the Linked List

```
/*begin() returns iterator to first node with data in linked list*/
iterator begin();
const_iterator begin() const;

/*end() returns iterator to node AFTER the last node in
the linked list*/
iterator end();
const_iterator end() const;

/*insert a node at start of list*/
void push_front(TYPE data);

/*insert a node at the end of the list*/
void push_back(TYPE data);

/*remove node from front of the list*/
void pop_front()

/*remove node from end of the list*/
void pop_end();

/*adds a node before the node referred to by the iterator*/
iterator insert(iterator itr,const T& data);

/*removes the node referred to by the iterator*/
iterator erase(iterator itr);

/*removes the nodes starting with the "from" iterator up to and incl
iterator erase(iterator from, iterator to);
```

## Initialization and Cleanup

The list also needs to be properly initialized. When the list goes out of scope, resources must be freed up. Therefore a public constructor and destructor are also needed.

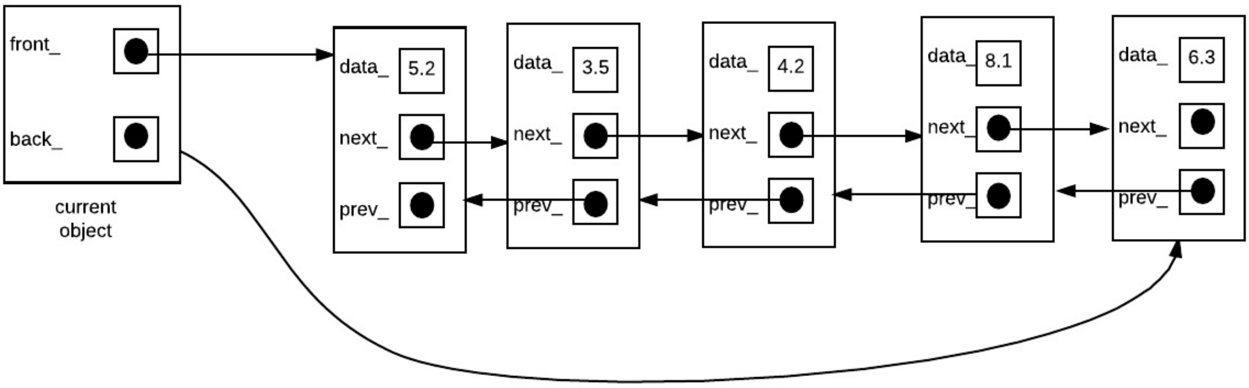
```
SList();
~Slist();
```

# Doubly Linked List

# Doubly Linked List

The previous linked list has a forward link only. Thus, at any point we can find out what the next node is with relative ease but to find the previous node, you would need to start at the beginning and search for a node who's next pointer has the same value as curr_.

One improvement that you could make to your list is to create a doubly linked list. A doubly linked list is a linked list where every node has both a forward and backwards



## The advantage of a doubly linked list

- No need to search entire list to find previous pointer
- Can move/search in both directions on list
- can access entire list from any point

## The disadvantages of a doubly linked list

- more memory is needed to store back pointer
- requires more work to set up back links properly

# Circular Linked List

## Circular Linked List

Another method of implementing a linked list involves using a circular form so that the next_ pointer of the last node points back to the first node.

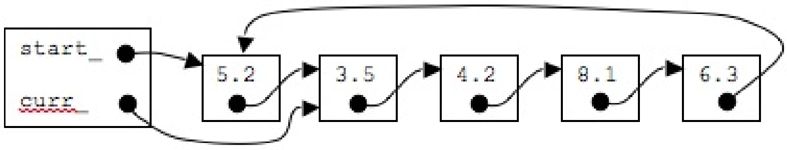
### Advantages of a circular linked list

- Some problems are circular and a circular data structure would be more natural when used to represent it
- The entire list can be traversed starting from any node (traverse means visit every node just once)
- fewer special cases when coding(all nodes have a node before and after it)

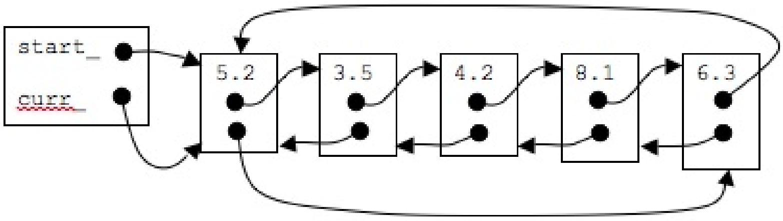
### Disadvantages of a circular linked list

- Depending on implementation, inserting at start of list would require doing a search for the last node which could be expensive.
- Finding end of list and loop control is harder (no NULL's to mark beginning and end)

### Picture of a singly linked circular linked list



### Picture of a doubly linked circular linked list



### Implementational Improvement

With a non-circular linked list, we typically have a pointer to the first item. However, with a circular linked list (especially a singly linked one) this implementation may not be a good idea. The reason for this is that if we point to the start of the list and we want to add/remove an item to the front, we would need to go through the entire list in order to find the last node so that we could keep the linked list hooked up properly.

One thing we could do is add another pointer to the list called last_ which points to the last node in the list. However, this means that our object will have another pointer to worry about setting properly.

Another method of implementation is to forget about the start *pointer entirely and just have a last* pointer. The reason for this is because if we point to just the last node, it is very very easy to find out what the first one was (remember start==last->*next*).

# Stacks

## The Stack

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 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.

# Stack Operations

# Stack Operations

- **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

# Stack Implementations

# Stack Implemenations

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:

Stack Implemented with an array 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:

Stack Implemented with a linked list http://cathyatseneca.github.io/DSAnim/web/llstack.html

# Stack Applications

# Stack Applications

### Bracket checking

One example of application of a stack is in checking that bookend semantics such as brackets are properly matched. That is if you have an expresion containing various brackets, the function would tell you if the brackets are correctly placed and matched.

```
int bracketcheck(char expr[]);
```

returns true if expr is a string where (), {} and [] brackets are properly matched. false if not.

### Postfix expression calculator

The way we write expressions uses infix notation. In other words, all operations look like A operator B (operator is "in" the middle of the expression). In order to change the order of operations, we must use (). Order of operations also matter

Another way to write expressions is to use postfix expression. All operations look like A B operator (the operator is after the operands) The advantage of postfix expressions is that brackets are not needed and order of operators are not needed. Example: infix

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

equivalent postfix:

```
1 2 + 3 4 5 + * -
```

Some calculators actually use postfix notation for entry.

# Queue

## Queue

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 (FIFO structure). A queue is a line up.

Note that when we refer to the front and back of queue, it is not necessarily talking about the front and back of the data structure used to implement the queue. The front should be thought of as where the oldest item in the queue is and the back is where the newest item is. Most important is to maintain the FIFO structure.

# Queue Operations

# Queue Operations

- **enqueue** - adds an item to the end of the queue
- **dequeue** - remove an item from front of the queue
- **initialize** - create an empty queue
- **isEmpty** - tests for whether or not queue is empty
- **isFull** - tests to see if queue is full (not needed if data structure grows automatically)
- **front** - looks at value of the first item but do not remove it

# Queue Implementations

# Queue Implementations

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:

Queue implemented as linked list 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:

Queue Implemented with an array http://cathyatseneca.github.io/DSAnim/web/arrayqueue.html

# Queue Applications

# Queue Applications

Queues are a useful representation of problems for different applications. For example, jobs to a network printer is enqueued so that the earlier a job is submitted the earlier it will be printed. Breadth first searches use queues. Queues also have applications in graph theory. Later when we study trees we will make use of a queue to perform a breadth first traversal of a tree.

# Tables

# 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

# Simple Table

# Simple Table

A simple way to implement a table is by using an array. Each element in the array can store one record. The records in the array are kept sorted according to the key. Note that a table does not have to have an ordering. The sorting of the keys just helps to make certain functionality faster

## 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)

# Hash Table

# Hash Table

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.

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 $\lambda$ measures the fullness of the hash table. It is calculated by the formula:

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

## Collisions

A hash function translates all possible keys into indexes in an array. This typically means that there are many many more keys than there are indexes. Thus, it is always possible that two or more keys will be translated into exactly the same index. When this happens 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.

# Bucketing

# Bucketing

Bucketing makes the hash table a 2D array instead of a single dimensional array. Every entry in the array is big enough to hold N items (N is not amount of data. Just a constant).

Problems:

- Lots of wasted space.
- If N is exceeded, another strategy will need to be used
- Not good for memory based implementations but doable if buckets are disk-based)

For bucketing it is alright to have $\lambda > 1$. However, the higher $\lambda$ is the higher a chance of collision. $\lambda > 1$ guarantees there will be at least 1 collision (pigeon hole principle). That will increase both the run time and the possibility of running out of buckets.

For a hash table of N locations and X buckets at each location:

- Successful Search - O(X) worst case
- Unsuccessful Search - O(X) worst case
- Insertion - O(X) - assuming success, bucketing does not have good way to handle non-successful insertions.
- Deletion - O(X)
- Storage: O(N * X)

# Chaining

## Chaining

At every location (hash index) in your hash table store a linked list of items. This is better than bucketing as you only use as many nodes as necessary. Some space will still be wasted for the pointers but not nearly as much as bucketing. Table will also not overflow (ie no pre-defined number of buckets to exceed). 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 $\lambda$. The average length of each chain is $\lambda$. $\lambda$ 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.

# Linear Probing

# Linear Probing

Both bucketing and chaining essentially makes use of a second dimension to handle collisions. This is not the case for linear probing. Linear Probing uses just a regular one dimensional array.

## 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.

# Quadratic Probing and Double Hashing

# Quadratic Probing and Double Hashing

Quadratic Probing and Double Hashing attempt to find ways to reduce the size of the clusters that are formed by linear probing.

## Quadratic Probing

Quadratic Probing is similar to Linear probing. The difference is that if you were to try to insert into a space that is filled you would first check $1^2 = 1$ element away then $2^2 = 4$ elements away, then $3^2 = 9$ elements away then $4^2 = 16$ elements away and so on.

With linear probing we know that we will always find an open spot if one exists (It might be a long search but we will find it). However, this is not the case with quadratic probing unless you take care in the choosing of the table size. For example consider what would happen in the following situation:

Table size is 16. First 5 pieces of data that all hash to index 2

- First piece goes to index 2.
- Second piece goes to 3 ((2 + 1)%16
- Third piece goes to 6 ((2+4)%16
- Fourth piece goes to 11((2+9)%16
- Fifth piece dosen't get inserted because (2+16)%16==2 which is full so we end up back where we started and we haven't searched all empty spots.

In order to guarantee that your quadratic probes will hit every single available spots eventually, your table size must meet these requirements:

- Be a prime number
- never be more than half full (even by one element)

### Double Hashing

Double Hashing is works on a similar idea to linear and quadratic probing. Use a big table and hash into it. Whenever a collision occurs, choose another spot in table to put the value. The difference here is that instead of choosing next opening, a second hash function is used to determine the location of the next spot. For example, given hash function H1 and H2 and key. do the following:

- Check location hash1(key). If it is empty, put record in it.
- If it is not empty calculate hash2(key).
- check if hash1(key)+hash2(key) is open, if it is, put it in
- repeat with hash1(key)+2*hash2(key), hash1(key)+3*hash2(key) and so on, until an opening is found.

like quadratic probing, you must take care in choosing hash2. hash2 CANNOT ever return 0. hash2 must be done so that all cells will be probed eventually.

Quadratic Probing and Double Hashing

# Sorting

## Sorting

Sorting is a fundamental task that can be a deciding factor in the overall performance of the system. In this section, we will consider several algorithms that can sort data. The performance of these algorithms can vary widely. Some algorithms are complex but work fast. Some are slow but very easy to write.

# Simple Sorts

## Simple Sorts

There are several sorts that are relatively easy to code. These are considered to be "simple sorts" and generally have bad run time O(\(n^2)\). In our implementation, we are assuming that we are sorting in ascending order (small to big). Sorting in descending order would typically involve flipping the check in the comparison of the sort. Also note that while all the examples use integer arrays, the general algorithm applies to arrays(and vectors) of all types.

# Bubble Sort

## Bubble Sort

A bubble sort is one of the simplest sorts to write. The idea behind a bubble sort is to start at the beginning of the array and swap adjacent elements that are not in order. Repeat this n-1 times where n is the size of the array and the array will be sorted.

```
void bubble(int array[],int sz){
    int i,j;
    int tmp;
    for(i=0;i<sz-1;i++){
        for(j=0;j<sz-i-1;j++){
            if(array[j] > array[j+1]){
                //swap arr[j] and arr[j+1]
                tmp=array[j];
                array[j]=array[j+1];
                array[j+1]=tmp;
            }
        }
    }
}
```

Bubble sort animation: http://cathyatseneca.github.io/DSAnim/web/bubble.html

# Insertion Sort

## Insertion Sort

The insertion sort algorithm works by looking at the array as if it is being filled in. The idea is that the array starts off in the first iteration as if it has only one element in it. The numbers that come after are then inserted into the "array" at the correct position. This is continued until all values in the entire array have been properly "inserted"

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

Insertion sort animation: http://cathyatseneca.github.io/DSAnim/web/insertion.html

# Selection Sort

## Selection Sort

The selection sort algorithm works by selecting the smallest value in the unsorted portion of the array then swapping it with the first value of the unsorted portion of the array.

```
void selectionSort(int arr[],int size){
    int minIdx;
    int tmp;
    for(int i=0;i<size;i++){
        minIdx=i;
        for(int j=i;j<size;j++){
            if(arr[j] < arr[minIdx]){
                minIdx=j;
            }
        }
        //swap
        tmp=arr[i];
        arr[i]=arr[minIdx];
        arr[minIdx]=tmp;
    }
}
```

Selection sort animation: http://cathyatseneca.github.io/DSAnim/web/selection.html

# Merge Sort

# 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.

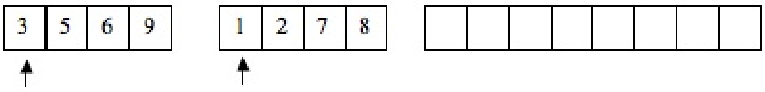
## Merge algorithm:

The algorithm to do so works as follows:

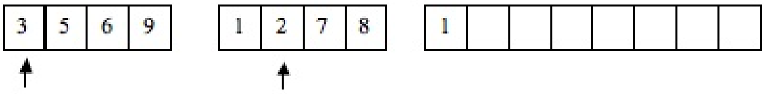
- Have a way to "point" at the first element of each of the two list
- compare the values being pointed at and pick the smaller of the two
- copy the smaller to the merged list, and advance the "pointer" of just that list to the next item.
- Continue until one of the list is completely copied then copy over remainder of the rest of the list

### Example

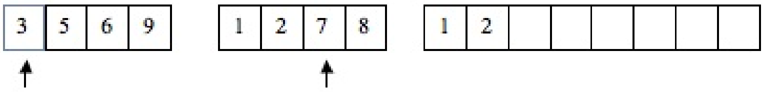
Here we have 2 sorted lists. Note that being already sorted is a must. This algorithm does not work on unsorted lists.



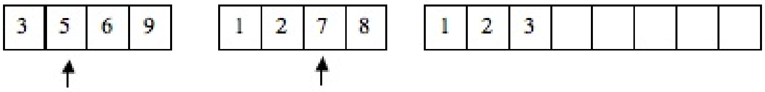
look at first element of each list and find smaller, copy it to the resulting list, then advance pointer.



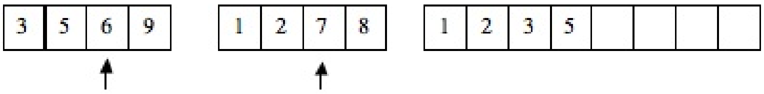
between 2 and 3, 2 is smaller:



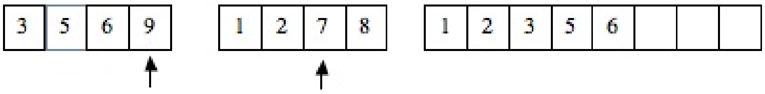
between 3 and 7, 3 is smaller:



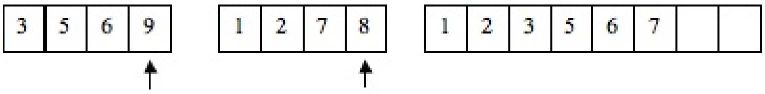
between 5 and 7, 5 is smaller:



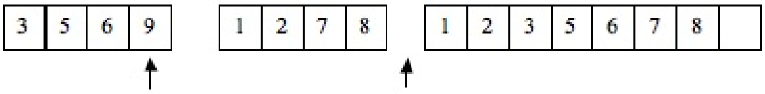
between 6 and 7, 6 is smaller:



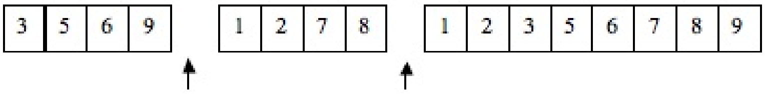
between 7 and 9, 7 is smaller:



between 8 and 9, 8 is smaller:



list 2 is now empty, copy rest of list 1 over



# Merge Sort Implementation

# Merge Sort Implementation

Now, 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.

We also need a second array for storaging the merged list. Our mergesort function should also not look any different than any other sorting function (ie all you need for a sorting function is the array). The sorting algorithm creates a temporary array 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 array that stays around for the entire process.

```
template <class TYPE>
void mergeSort(vector<TYPE>& arr){
  vector<TYPE> tmp(arr.size());
  //call mergeSort with the array, the temporary
  //and the starting and ending indices of the array
  mergeSort(arr,tmp,0,arr.size()-1);
}
```

This algorithm is recursive in nature. The idea is that we have a function that if it works, will sort an array between start and end inclusive. So how it works is we start with the big array and we mergeSort() each of the two halves of it. We can then merge the two lists together. to form a sorted list.

The base case for this algorithm is when we have an array that is one element long (ie start and end are the same). A list that has one element is always sorted. So if we merged two of these lists that are one element big together, we will create a list that is 2 elements and sorted.

```
//this function sorts arr from index start to end
template <class TYPE>
void mergeSort(vector<TYPE>& arr, vector<TYPE>& tmp, int start, int
  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);
  }
}
```

This merge function is the code as stated in the previous description

```
/*This function merges the two halves of the array arr into tmp and
template <class TYPE>
void merge(vector<TYPE>& arr, vector<TYPE>& tmp, int start,
                                        int start2, int end){
  int aptr=start;
  int bptr=start2;
  int i=start;
  while(aptr<start2 && bptr <= end){
    if(arr[aptr]<arr[bptr])
      tmp[i++]=arr[aptr++];
    else
      tmp[i++]=arr[bptr++];
  }
  while(aptr<start2){
    tmp[i++]=arr[aptr++];
  }
  while(bptr<=end){
    tmp[i++]=arr[bptr++];
  }
  for(i=start;i<=end;i++){
    arr[i]=tmp[i];
  }
}
```

Merge sort animation: http://cathyatseneca.github.io/DSAnim/web/merge.html

# Quick Sort

# 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(n2)

QuickSort works like this:

1. Pick a value from the array as the pivot
2. Let i=front, j= back
3. advance i until you find a value arr[i] > pivot
4. move j towards front of array until arr[j] < pivot
5. swap these arr[i] and arr[j].
6. repeat step 3 to 5 until i and j meet
7. The meeting point is used to break the array up into two pieces
8. QuickSort these two new arrays

A simple version of the algorithm can be written as:

```
template <class TYPE>
void QuickSort(vector<TYPE>& arr, int left, int right){
  if(right-left <=3){
    InsertionSort(arr,left,right);
  }
  else{
    int pivotpt=right;
    int i=left;
    int j=right-1;
    TYPE pivot=arr[pivotpt];
    while(i<j){
      while(arr[i]<pivot) i++;
      while(arr[j]>pivot) j--;
      if(i<j)
        swap(arr[i++],arr[j--]);
    }
    swap(arr[i],arr[pivotpt]);
    QuickSort(arr,left,i-1);
    QuickSort(arr,i+1,right);
  }
}

template <class TYPE>
void QuickSort(vector<TYPE>& arr){
  QuickSort(arr,0,arr.size()-1);
}
```

The idea of a quicksort is that with each call, the quicksort algorithm would break the vector up into two parts. The best thing that can happen is that each time we get pieces that are of equal size (ie we get two pieces that is half the size of the original vector). As you will recall from binary search, you can only half something log n times (where n is the size of the vector) before you get a vector of size 0. Thus, for the best possible case, the runtime is O(n log n).

The worst thing that can happen for the quicksort algorithm is to choose a pivot such that we end up getting a pivot point that breaks up the vector into an empty vector and a vector containing everything else every single time. If you do this, you will end up with not reducing the size of the vector by much (just one for the pivot) and thus, your work will be almost as much as it was to begin with. The worst case run time is O(\n^2\)

## Median of Three Partitioning

The average runtime for quicksort is also O(n log n).

The algorithm that is shown above chooses the last element in the vector as the pivot. This was done for simplicity but is actually not very good. If the elements are random, then using the end of the array is not bad. However, if the data is nearly sorted to begin with then picking the end point could very well mean picking the biggest value. This would create the worst case scenario where one vector was empty and the other contained everything else.

Recall that the best possible scenario occurs when we break down the list into two pieces of the same size. To do this, our pivot must be the median value. The median is the number where half of the other numbers are below it and half the other numbers are above it. Thus, if our pivot was the median of the list each and every time, we would end up with the best case scenario. However, finding the median is not an easy problem to solve quickly so instead of trying to find the exact median, one strategy is to choose the median of three values in the vector (use the three values like a sampling of the population. Like polling!) This can be done by looking at the first, last and middle element. From here choose the median value of the three as the pivot.

# Heap Sort

## 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 repeated call delete to remove the smallest value and put it back into the array.

# Binary heap

# Binary heap

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, we would not be able to get a very good run time to our sort.

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 that... and we want to be able to find it and remove it quickly but all other values can be essentially anywhere.

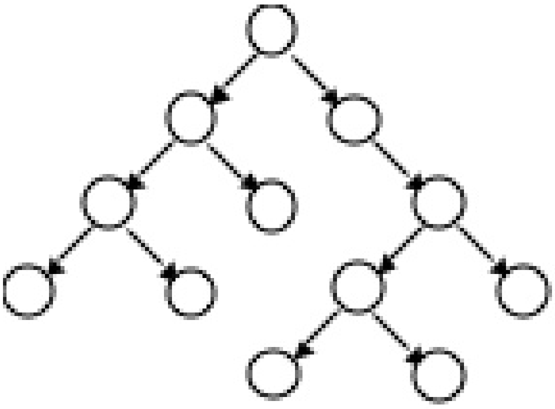
A binary heap is a data structure that can help us do this.

# Binary heap basics

# Binary heap basics

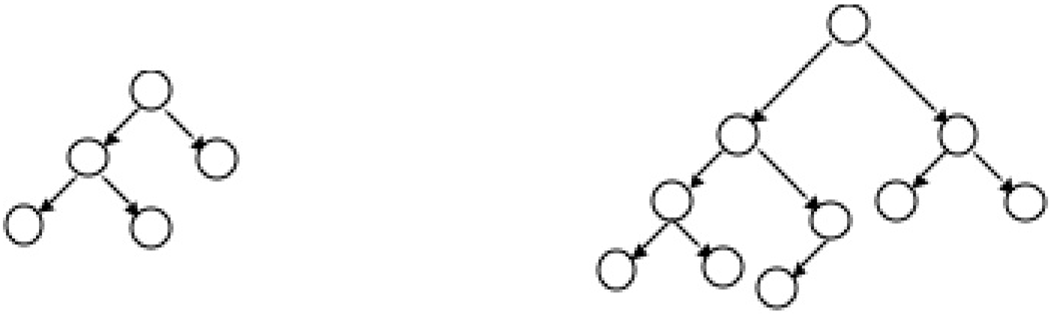
**Binary Heap** - A binary heap is a **complete binary tree** where the **heap order property** is always maintained.

**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.

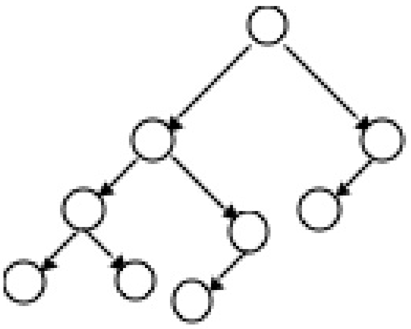


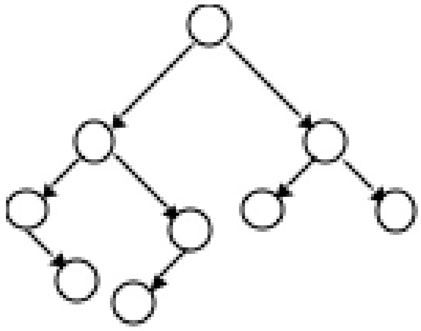
**Complete Binary Tree** - A binary tree where there are no missing nodes in all except at the bottom level. At the bottom level the missing nodes must be to the right of all other nodes.

complete binary trees:

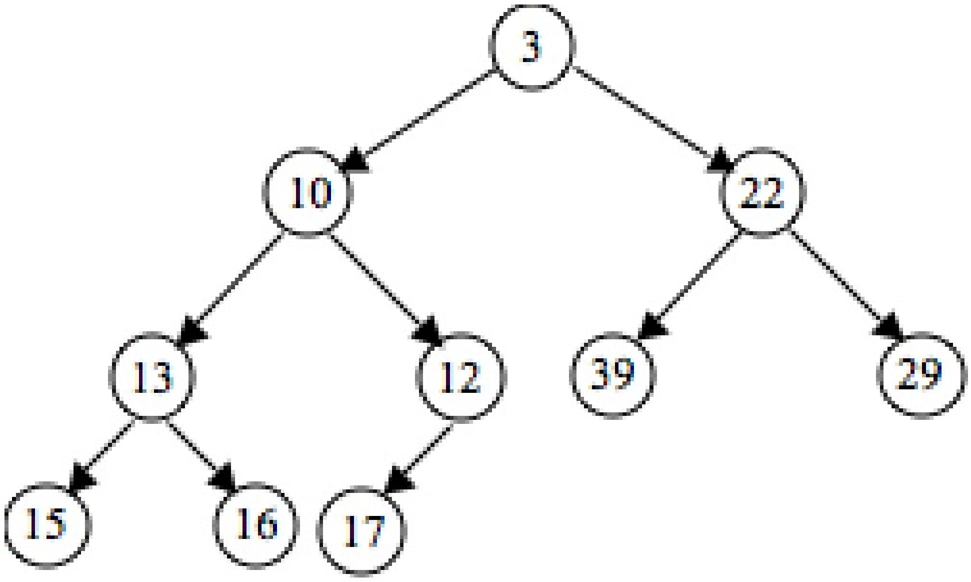


not complete binary trees:





**Heap Order Property**: The priority of the children of each node must be lower than that in the node. In this picture, we define priority to be smaller value having higher priority, thus the smallest value, has the highest priority:



# Insertion into a binary heap

# Insertion into a binary heap

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.

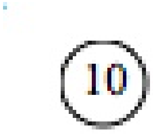
We know that in order to maintain the complete binary tree structure, we must add a node to the first open spot at the bottom level. So we begin by adding a node without data there. After that what we do is we see if inserting our new value will violate the heap property. If it doesn't put it in and we are done. Otherwise, we move the value from the parent of the empty node down , thus creating an empty node where the parent use to be. We again check to see if inserting the value violates the heap order property. If not, we add it in. Otherwise repeat the process until we are able to add the value. This process of moving the empty node towards the root is called ***percolate up***

## Illustrated example:

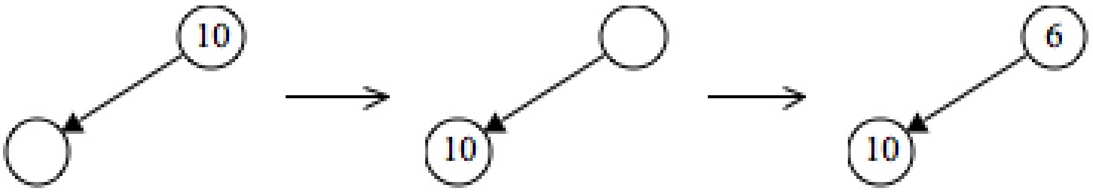
Insert into a heap the following values in order: 10,6,20,5, 16, 17, 13,2

We will use smaller values has higher priority as our priority ordering.

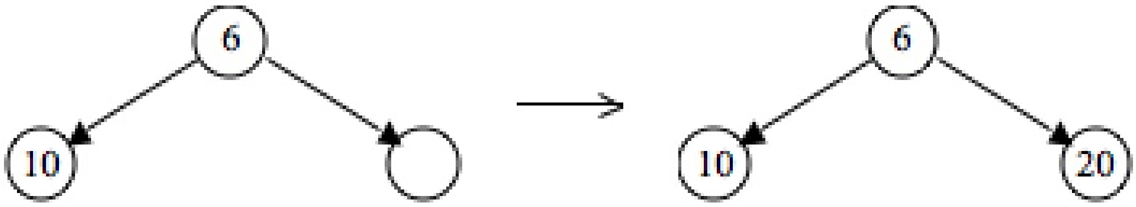
insert 10:



insert 6:

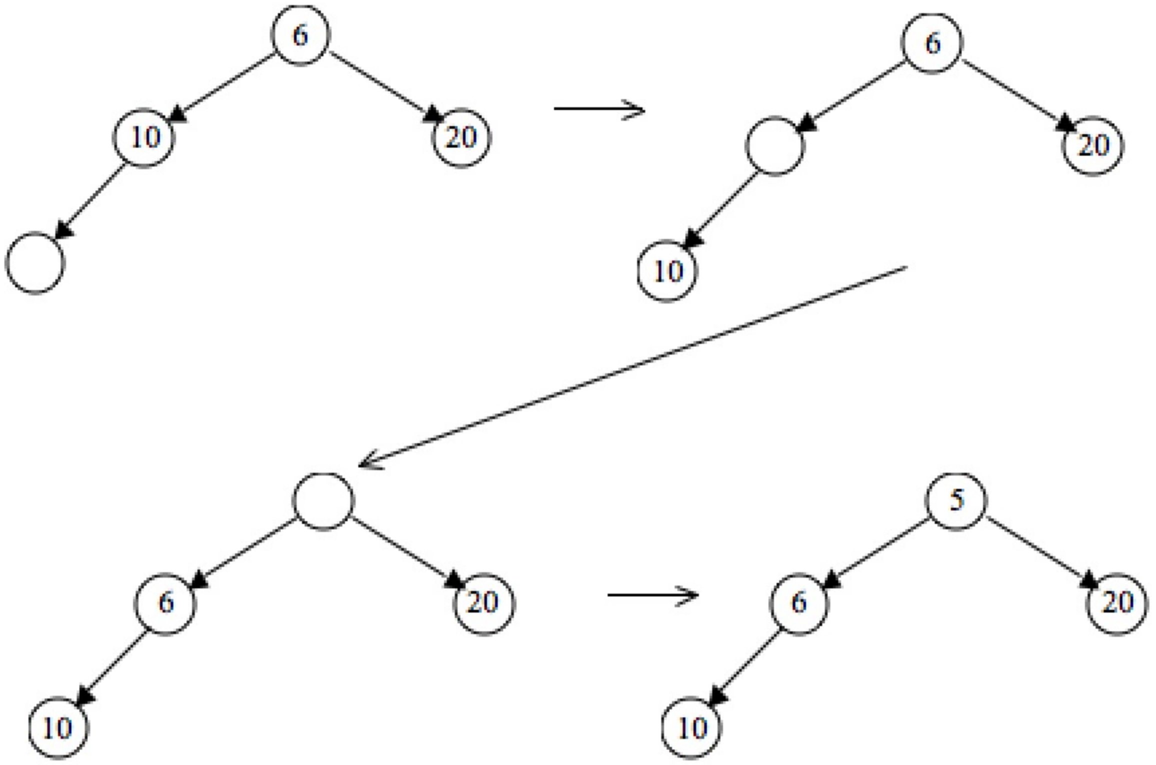


insert 20:

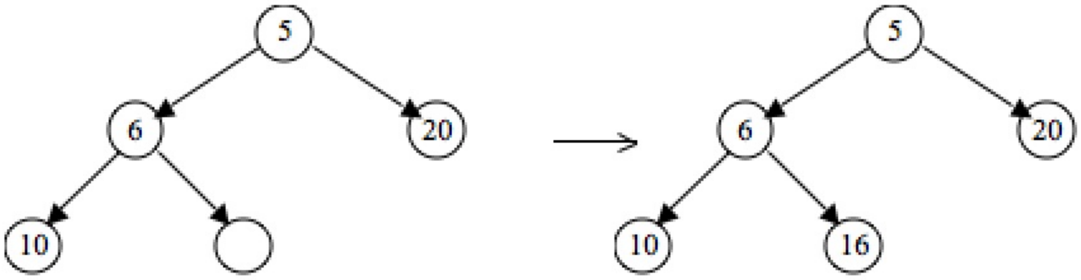


insert 5:

Insertion into a binary heap

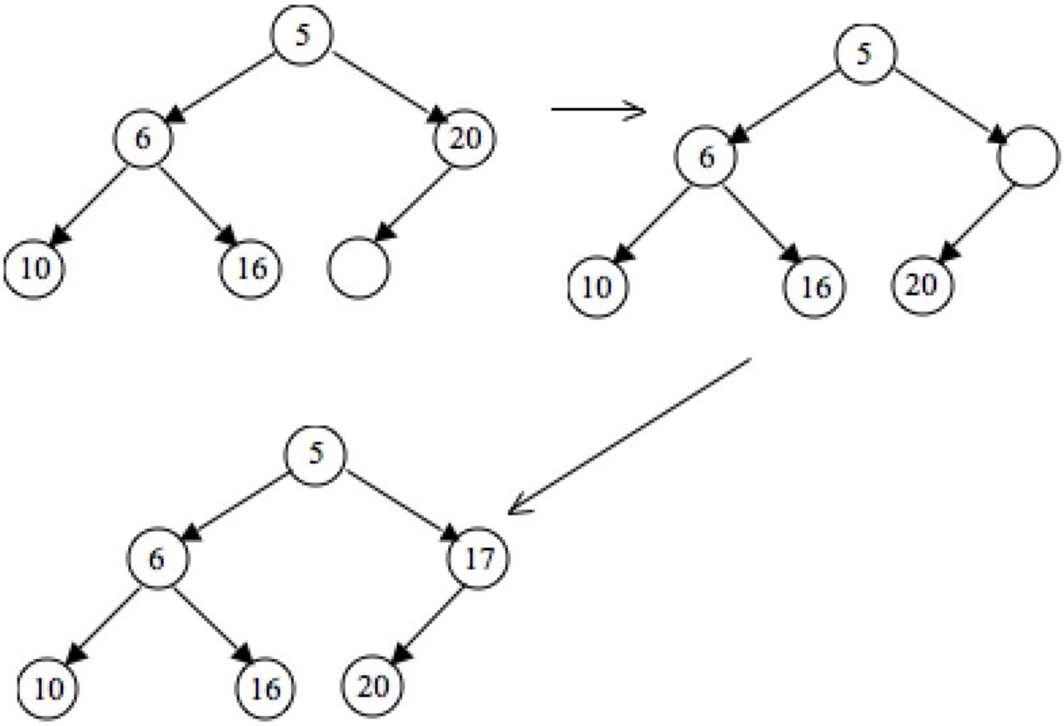


insert 16:

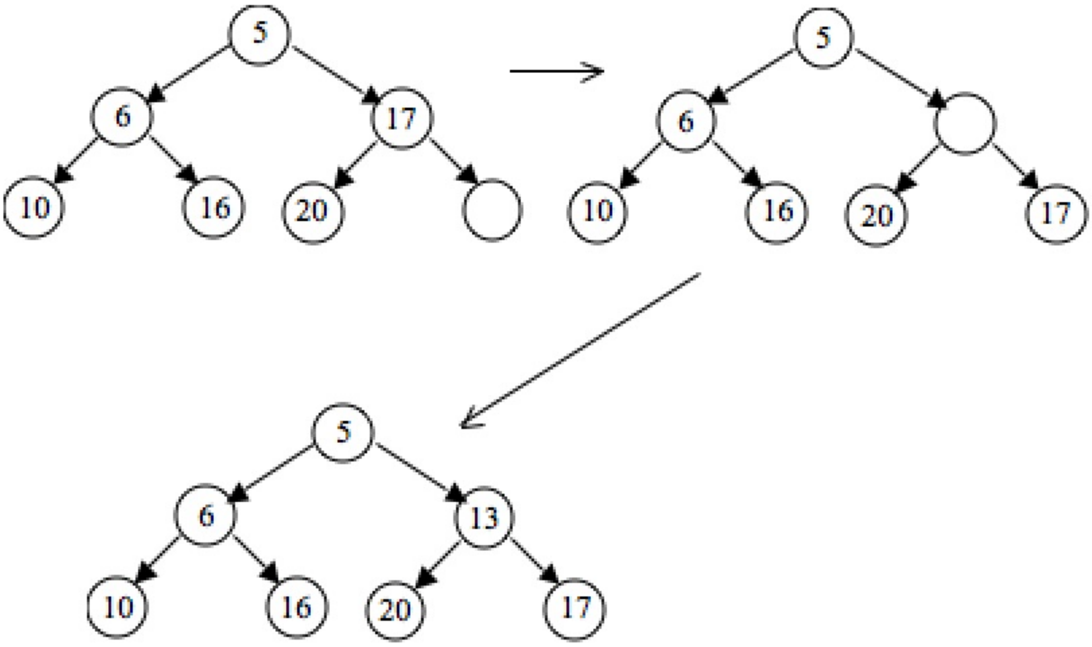


insert 17:

Insertion into a binary heap

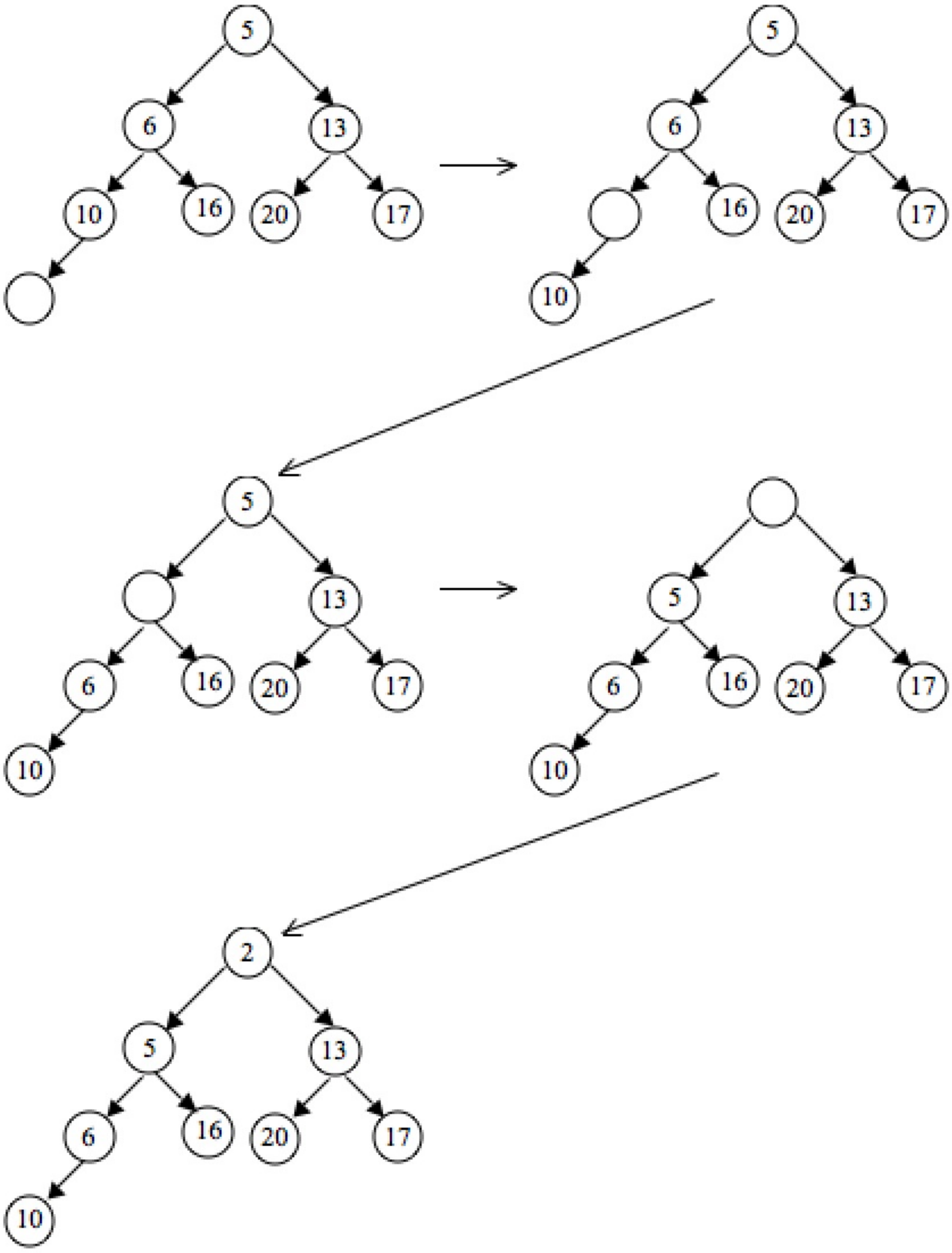


insert 13:



insert 2:

Insertion into a binary heap



# Delete from a binary heap

# Delete from a binary heap

This is the process of removing the highest priority value 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 wo 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.

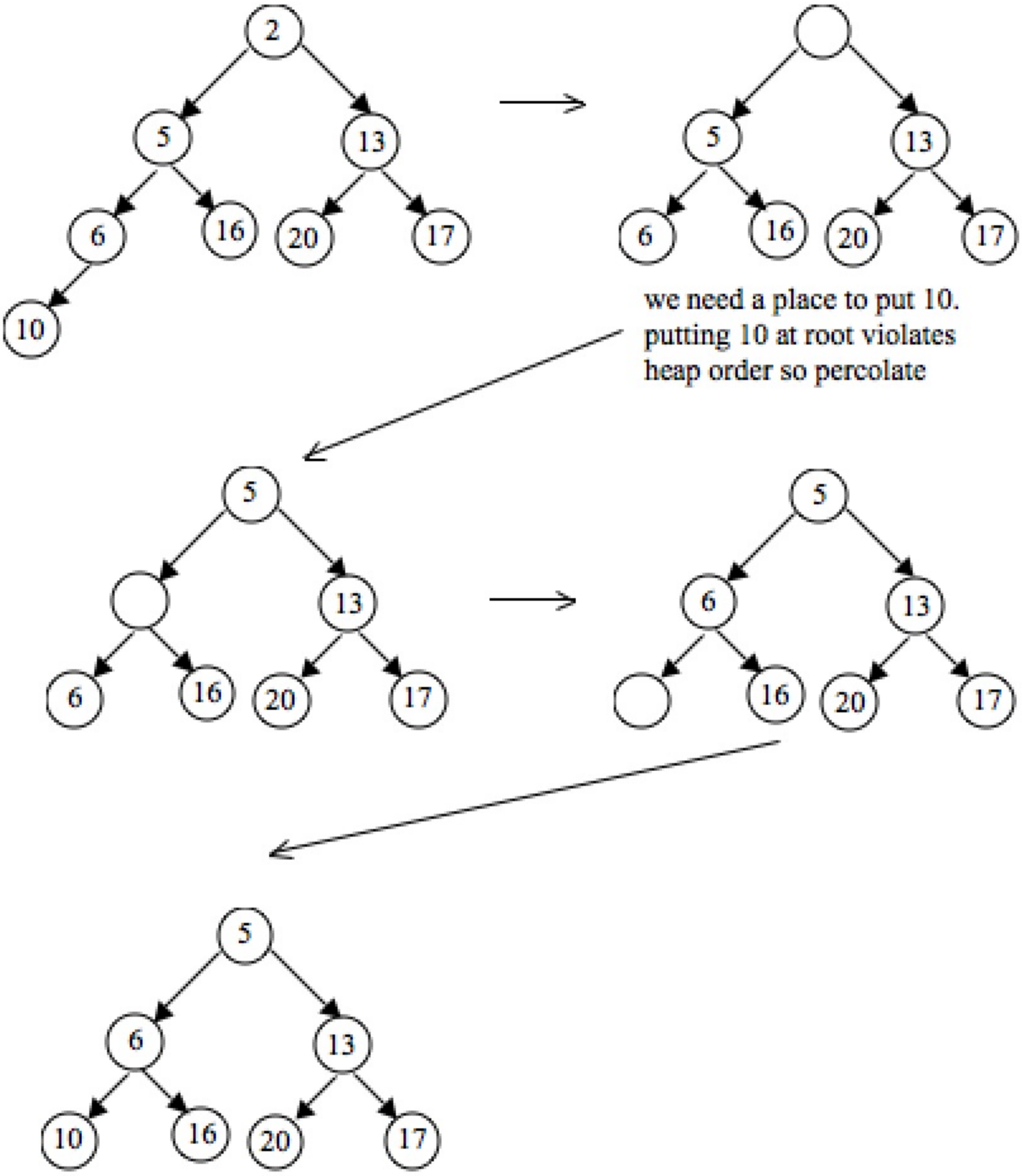
The empty spot that had been created by the removal of the value at root must be filled and the value that had been in the rightmost node must go back into the heap. We can accomplish this by doing the following:

- 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***

## Illustrated example:

# Delete from a binary heap



we need a place to put 10. putting 10 at root violates heap order so percolate

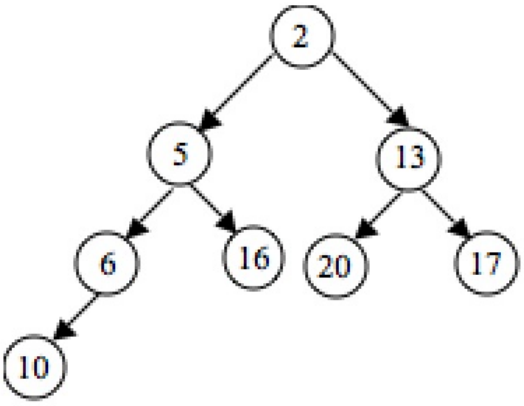
# Implementation

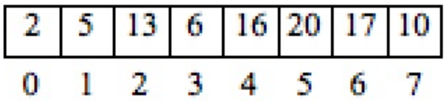
# Implementation of a heap

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)





| 2 | 5 | 13 | 6 | 16 | 20 | 17 | 10 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

consider node with 5 in it. 5 is at element with index 1 in array. According to formulas:

Left child should be in index 2i+1 = 2(1)+1= 3 Right child should be in index 2i+2 = 2(1)+2 = 4 Parent is in index (i-1)/2 = (2-1)/2 = 1/2 = 0

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.

As long as we keep track of number of elements in the array, we do not have to worry about empty nodes.

We also eliminate any need for pointers or the overhead of allocating nodes as we go.

# Sorting

# Sorting and Analysis

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(n 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 and then doing the following with it:

```
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.

What we want to be able to do is to implement the heap sort in place.

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.

For heap sort we start off by using the first element as the root of the heap. Successive elements are then placed into the heap in place. Because new nodes are always bottom level leftmost, the element added will always be "end" of the array. Thus, similar to doing an insertion sort...we insert subsequent values into our heap which is being formed at the front end.

One key thing we have to change though is the priority. We actually need to ensure that items with 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 use give larger numbers a higher priority.

The reason is this... when we start to remove from the heap, the node that is first removed is from the bottom level, right most. this is essentially last node in the heap part of the array. However, the value that is removed comes from the root. Since the position that is freed up is at the end of the array, we want to ensure that the value that is removed first is the lowest priority item instead of highest otherwise our list will be sorted backwards.

Thus, to sort an array in ascending order (small to big), our heap should be built so that larger values have higher priority.

# Introduction to Trees, Binary Search Trees

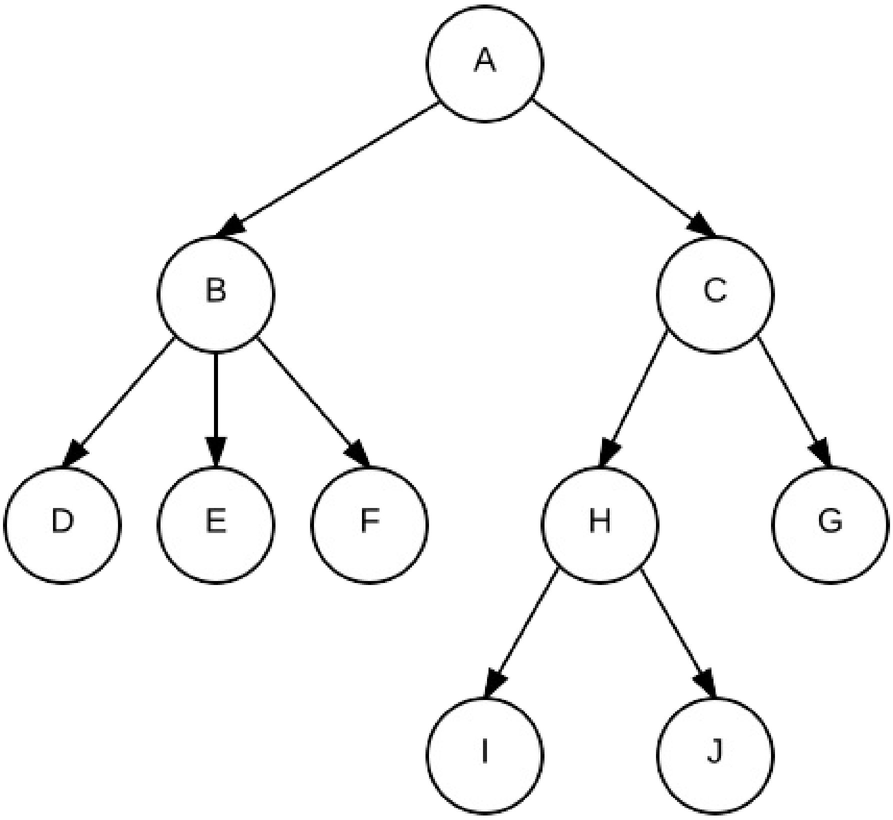
# Introduction to Trees, Binary Search Trees

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

# Definitions

This section will include some basic terminology used when describing trees. To help you understand the terminology, use the following diagram:



:

**Node**: data stored by the tree (All the circles)

**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. Descendents of C are G,H,I and J

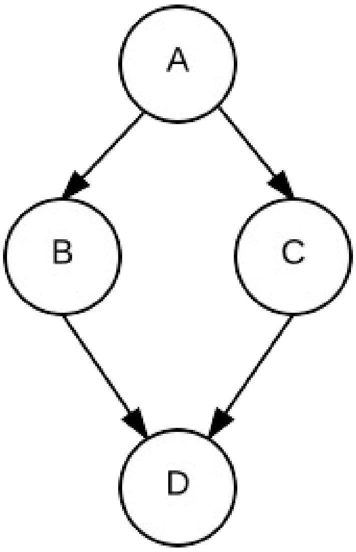
**Depth**: Disthance from root node of tree. Root node is at depth 0. B and C are at depth 1. Nodes at depth 2 are D,E,F,G and H. Nodes at depth 3 are I and J

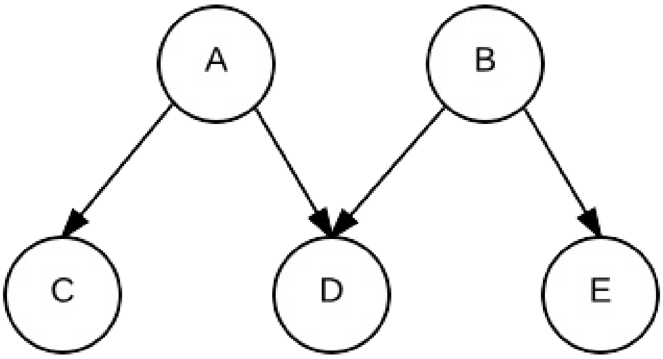
**Height**: Total number of nodes from root to furthest leaf. Our tree has a height of 4.

**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 Implementations

## Tree Implementations

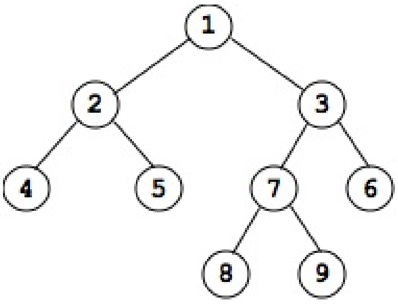
By its nature, trees are typically implemented using a Node/link data structure like that of a linked list. In a linked list, each node has data and it has a next (and sometimes also a previous) pointer. In a tree each node has data and a number of node pointers that point to the node's children.

However, a linked structure is not the only way to implement a tree. We saw this when we used an array to represent a binary heap. In that case, the root was stored at index 0, for any node stored at index i, we calculate the left, right subtree, and parent node's indexes with formulas. However, if you do this, for non-complete trees, you could end up with many unused spaces in an array.

# Binary Trees

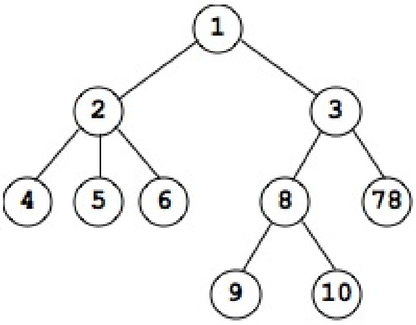
## Binary Trees

A binary tree is a tree where every node has 2 subtrees that are also binary trees. The subtrees may be empty. The following is a binary tree:



Note that the term ***binary tree*** describes only the shape, it does not specify the ordering of values within the tree.

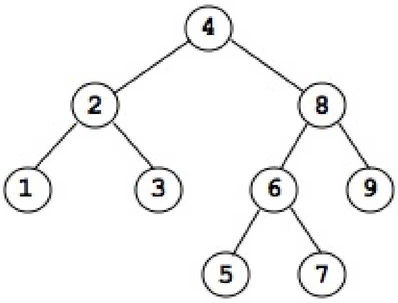
The following isn't (node with 2 has three subtrees):



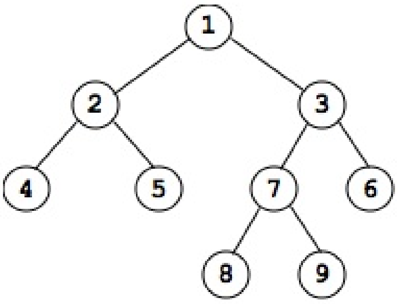
# Binary Search Trees

## Binary Search Trees

Binary search trees (BST) are binary trees where values are placed in a way that supports efficient ( $O(logn)$) 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:



A binary search tree allows us to quickly perform a search on a linking structure (under certain conditions). 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. Note that 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.

```
TNode<TYPE>* BST::Search(TYPE key){
  TNode* retval=NULL;
  TNode* curr=root_;
  while(retval && curr){
    if(curr->data==key)
      retval=curr;
    else if(key < curr->data)
      curr=curr->left_;
    else
```

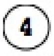
```
      curr=curr->right_;
  }
  return retval;
}
```

# Insertion

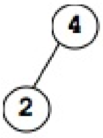
## Insertion

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:

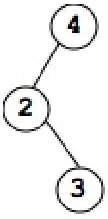
Insert 4:



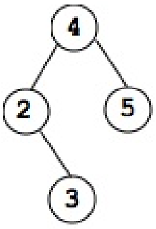
Insert 2: 2 is < 4 so it goes left



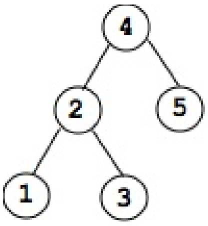
Insert 3: 3 is less than 4 but more than 2 so it goes to left of 4, but right of 2



Insert 5: 5 is > 4 so it goes right of 4

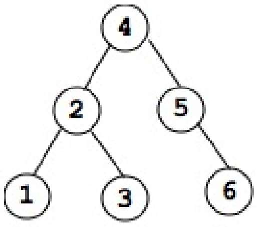


Insert 1: 1 is < 4 so it goes to left of 4. 1 is also < 2 so it goes to left of 2



Insert 6: 6 is > 4 so it goes to right of 4. 6 is also > 5 so it goes to right of 5

Insertion

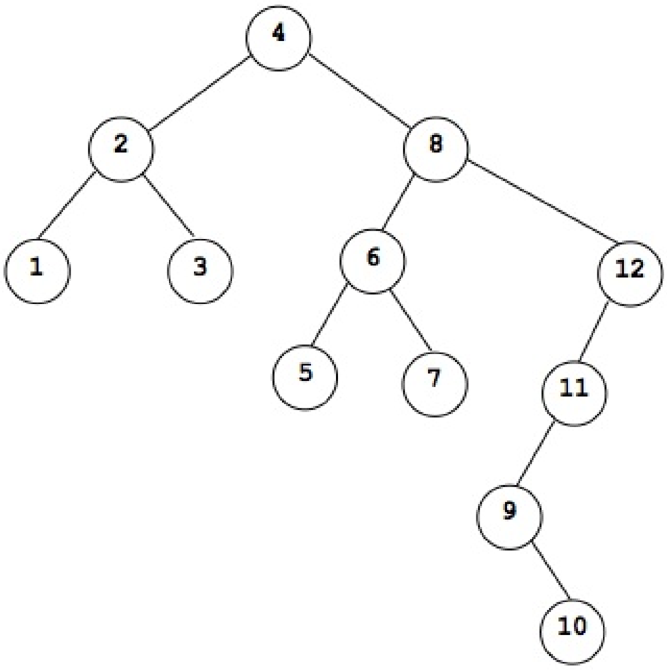


# Removal

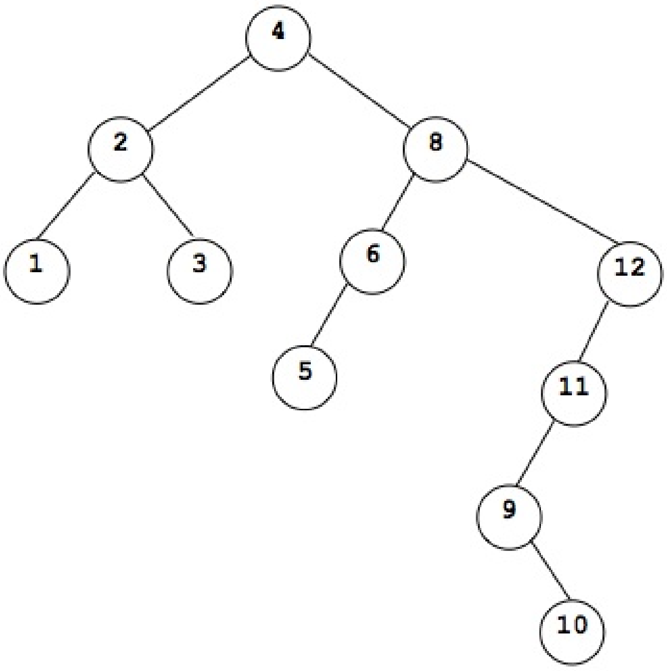
## Removal

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.

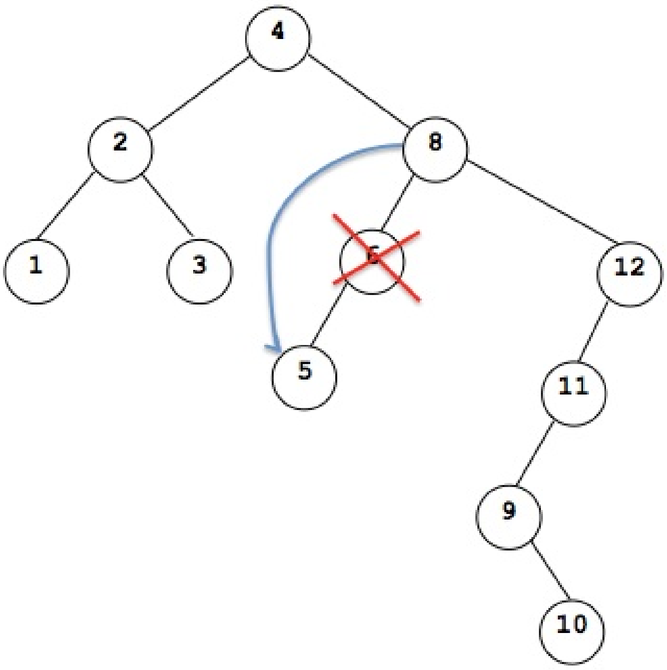
Suppose we start with the following binary search tree:



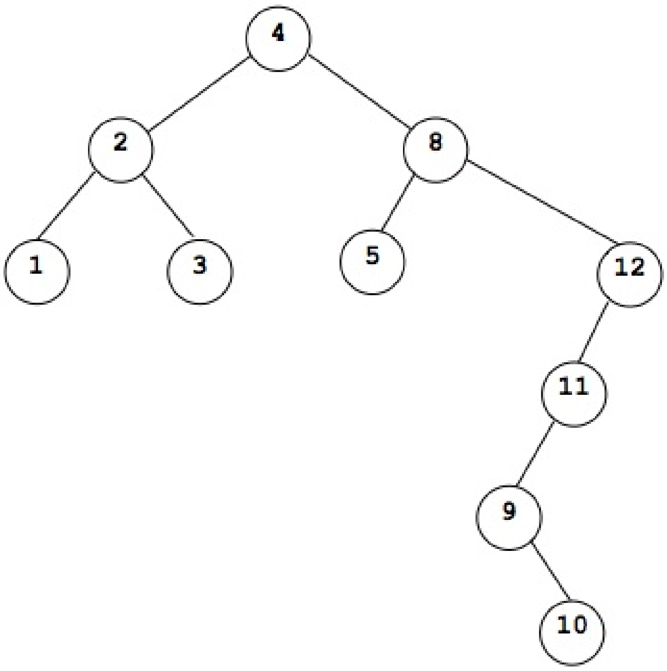
Suppose we were to remove 7 from the tree. Removing this node is relatively simple, we simply have to ensure that the pointer that points to it from node 6 is set to NULL and delete the node:



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.



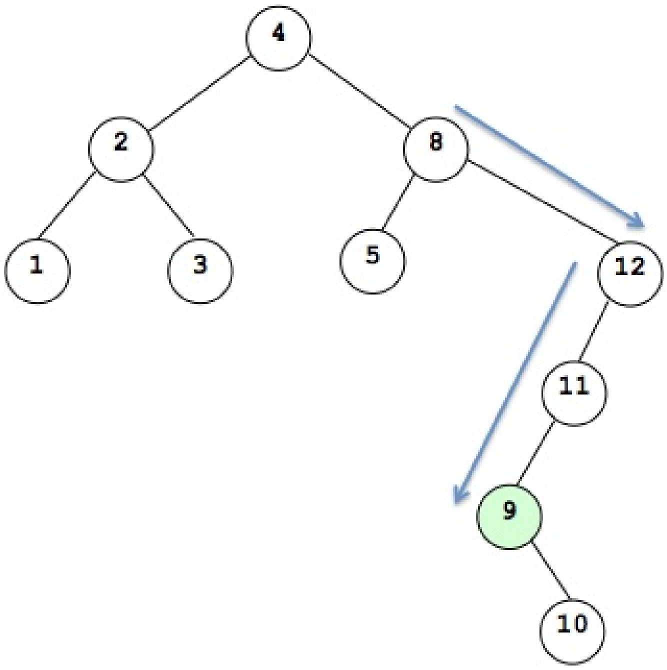
Thus our tree is now:



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.

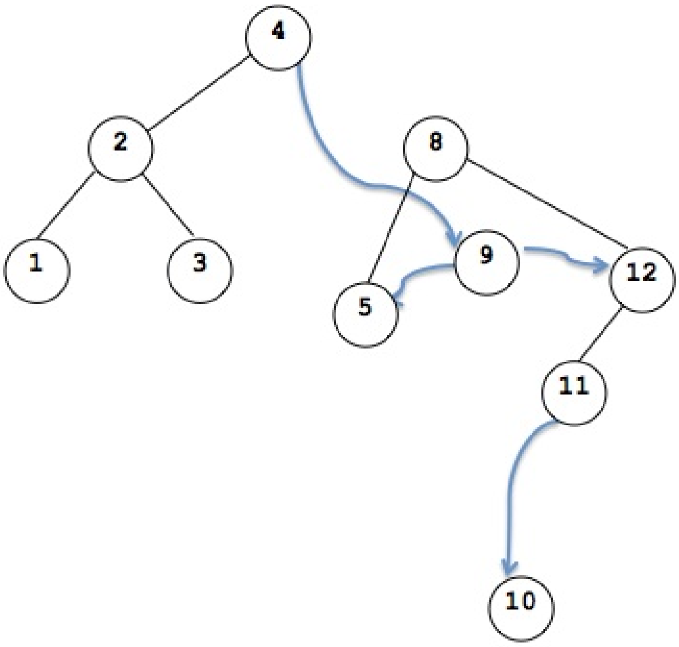
Instead, we should find the inorder successor (next biggest descendent) to 8 and promote it so that it replaces 8.

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:

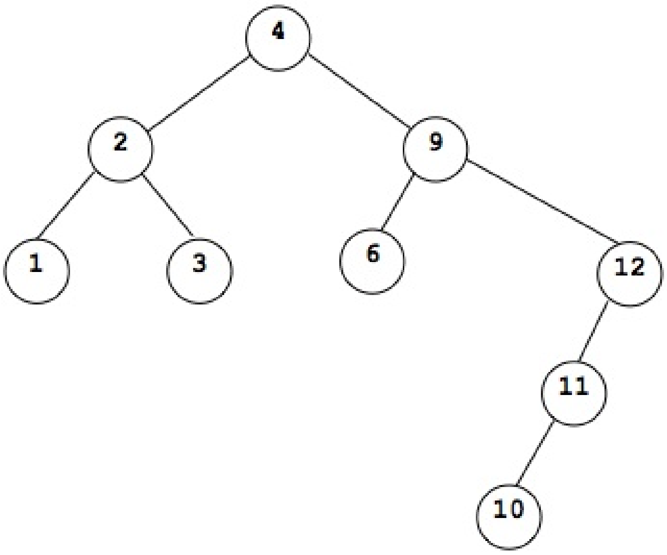


The inorder successor will either be a leaf node or it will have a right child only. It will never have a left child because we found it by going as far left as possible.

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.



In the end we have:



# Traversals

# 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.

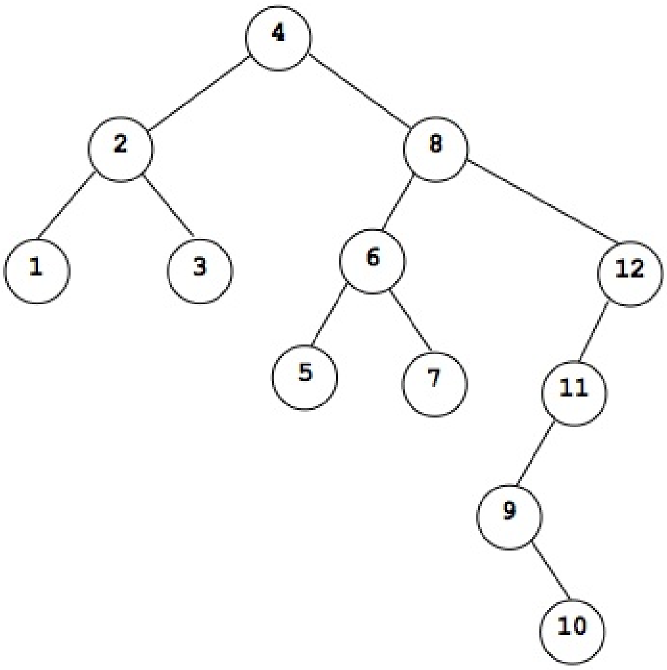
Traversals can be done either depth first (follow a branch as far as it will go before backtracking to take another) or breadfirst, go through all nodes at one level before going to the next.

## 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, 12, 11, 9, 10

### 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

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

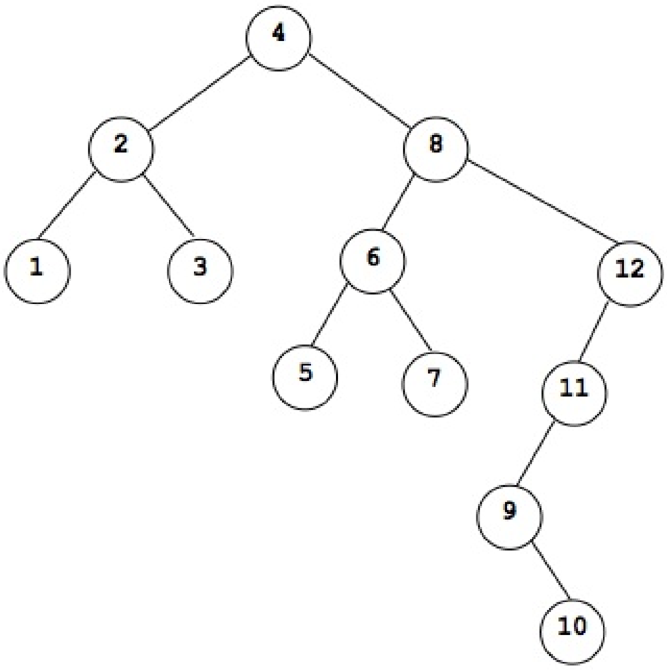
1, 3, 2, 5, 7, 6, 10, 9, 11, 12, 8, 4

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, 12, 5, 7, 11, 9, 10

# AVL Trees

## AVL Trees

A tree is perfectly balanced if it is empty or the number of nodes in each subtree differ by no more than 1. In a perfectly balanced tree, we know that searching either the left or right subtree from any point will take the same amount of time.

The search time in a perfectly balanced tree is O(log n) as the number of nodes left to consider is effectively halfed with each node considered. However, to get a tree to be perfectly balance can require changing every node in the tree. This makes trying to create a perfectly balanced tree impractical.

An AVL tree does not create a perfectly balanced binary search trees. Instead it creates a ***height balanced*** binary search trees. 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

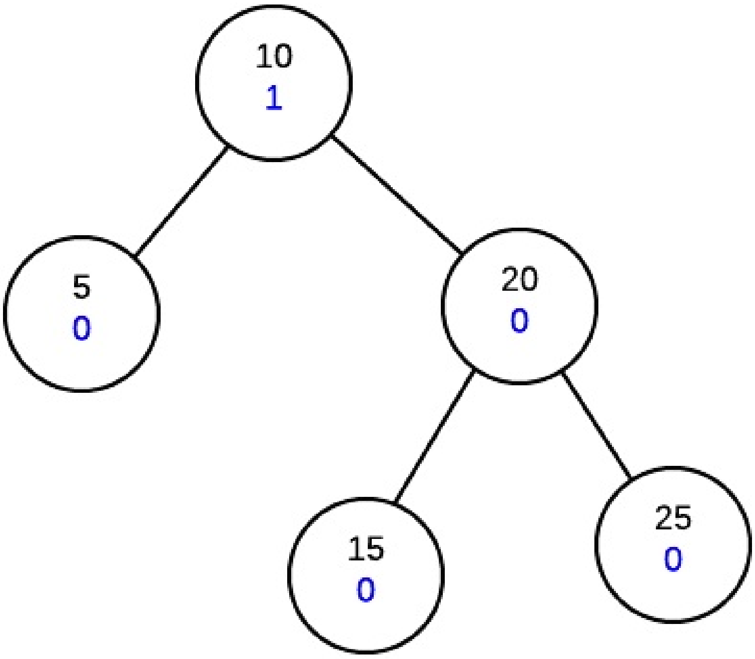
## Height Balance

AVL trees work by ensuring that the tree is height balanced after an operation. If we were to have to calculate the height of a tree from any node, we would have to traverse its two subtrees making this impractical. Instead, we store the height balance information of every subtree in its node. Thus, each node must not only maintain its data and children information, but also a height balance value.

The height balance of a node is calculated as follows:

```
height 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

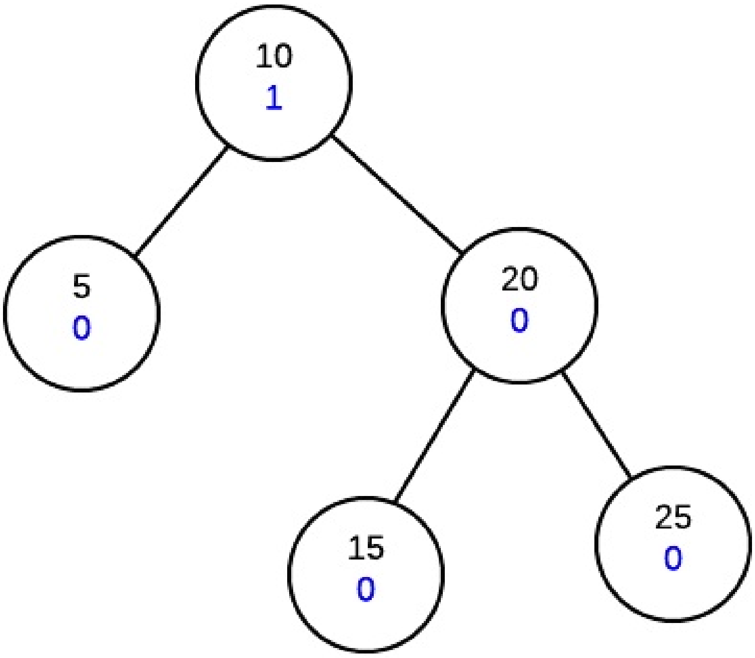
# Insertion

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

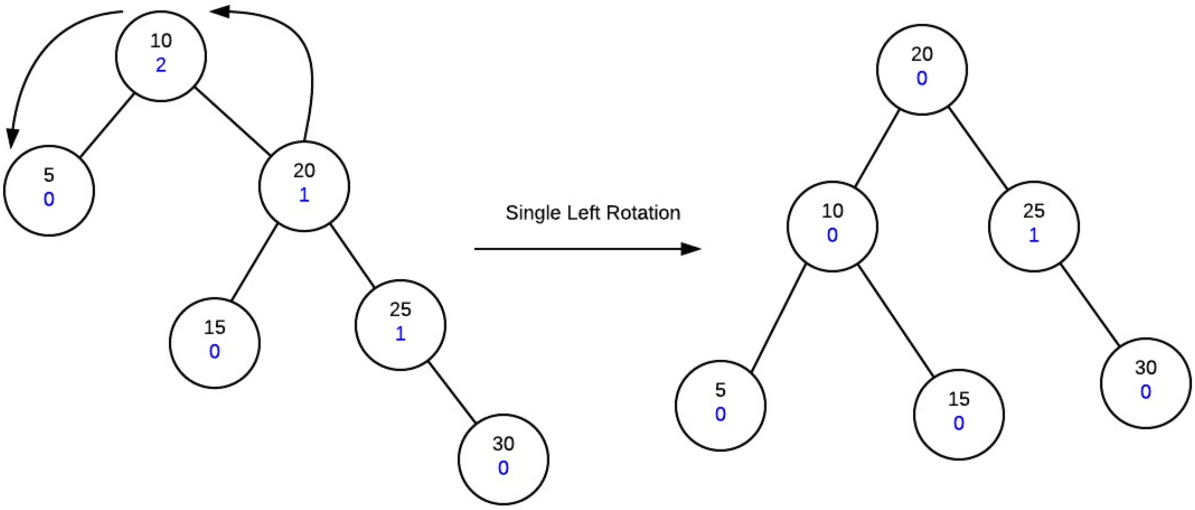
Suppose we have start with the following tree (value on top is the value, value on bottom is the height balance



### Insert 30

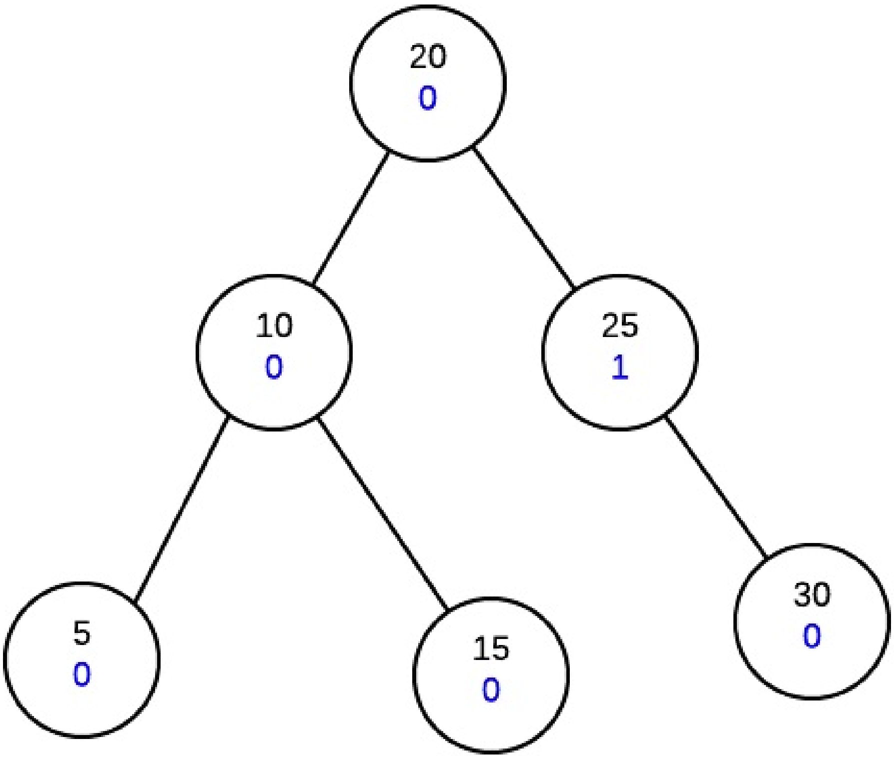
At this point, we have adjusted all the height balances along the insertion path and we note that the root node has a height balance of 2 which means the tree is not height balanced at the root.

In order to fix our tree, we will need to perform a rotation

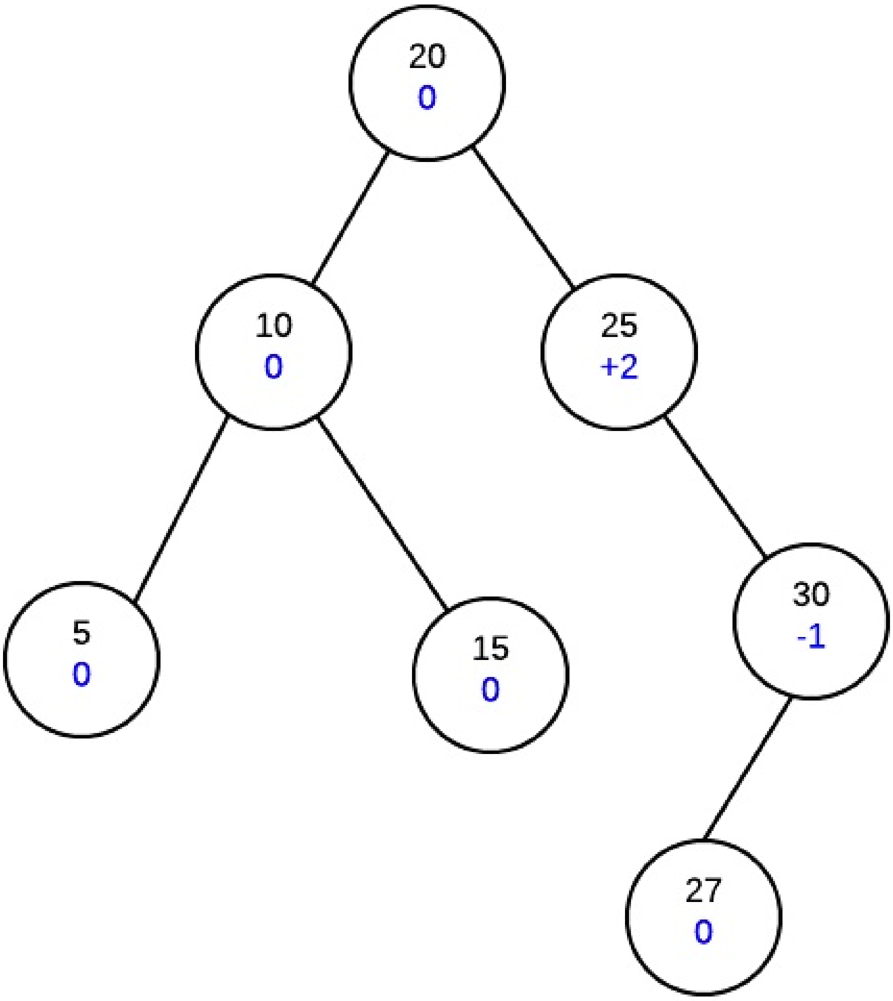


## Insert 27

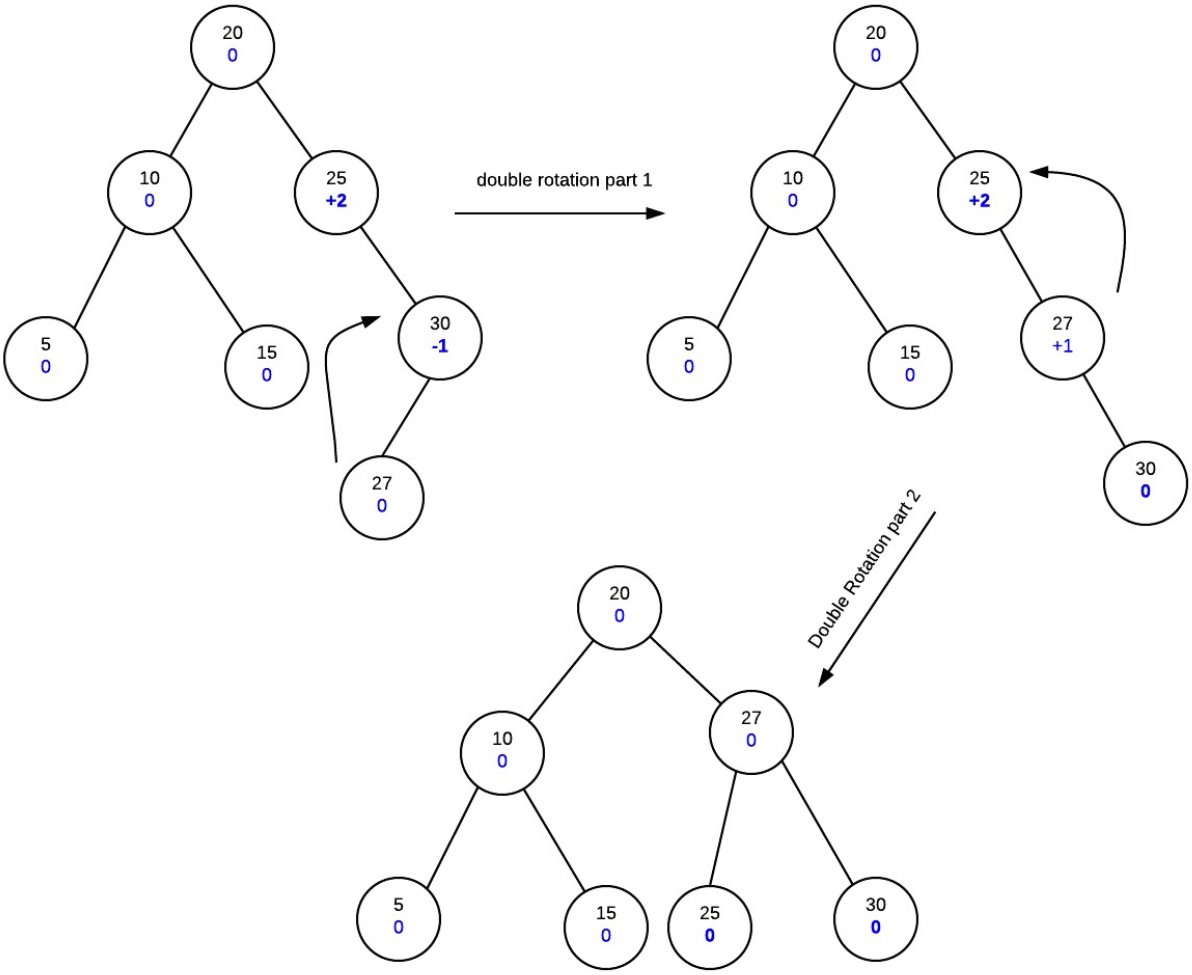
We start with our tree:



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.


20
0
10
0
25
+2
5
0
15
0
30
-1
27
0
double rotation part 1
20
0
10
0
25
+2
5
0
15
0
27
+1
30
0
Double Rotation part 2
20
0
10
0
27
0
5
0
15
0
25
0
30
0

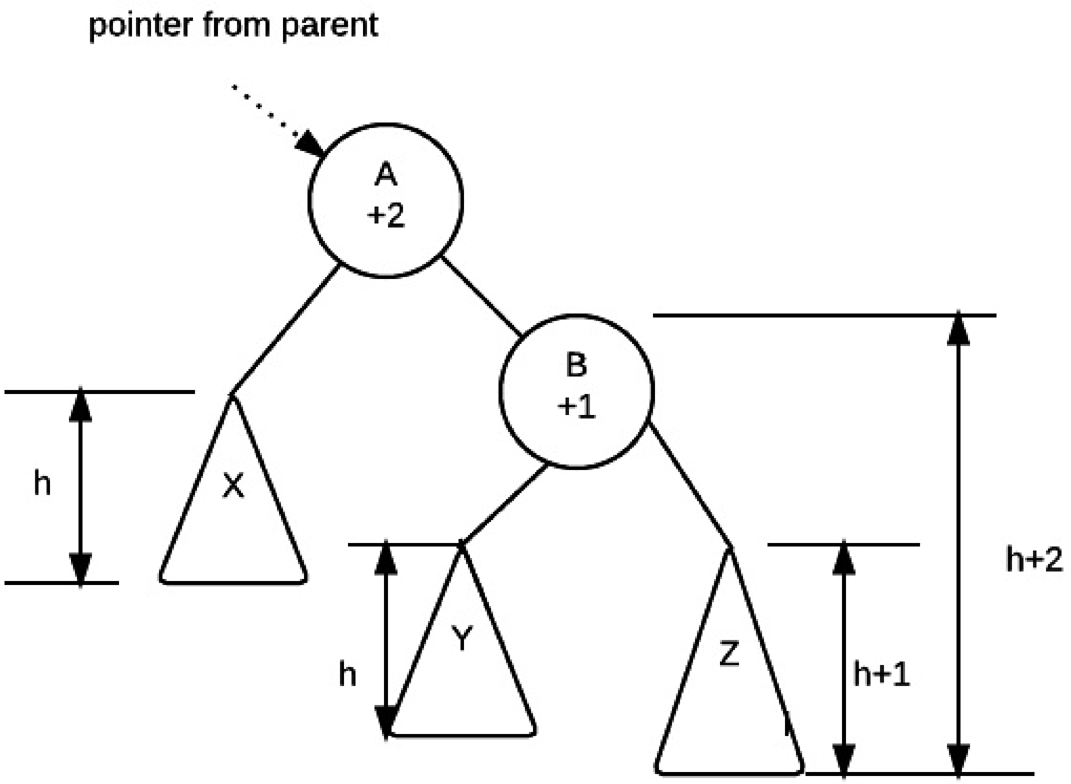
# Why it works

# Why it works

## 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
```

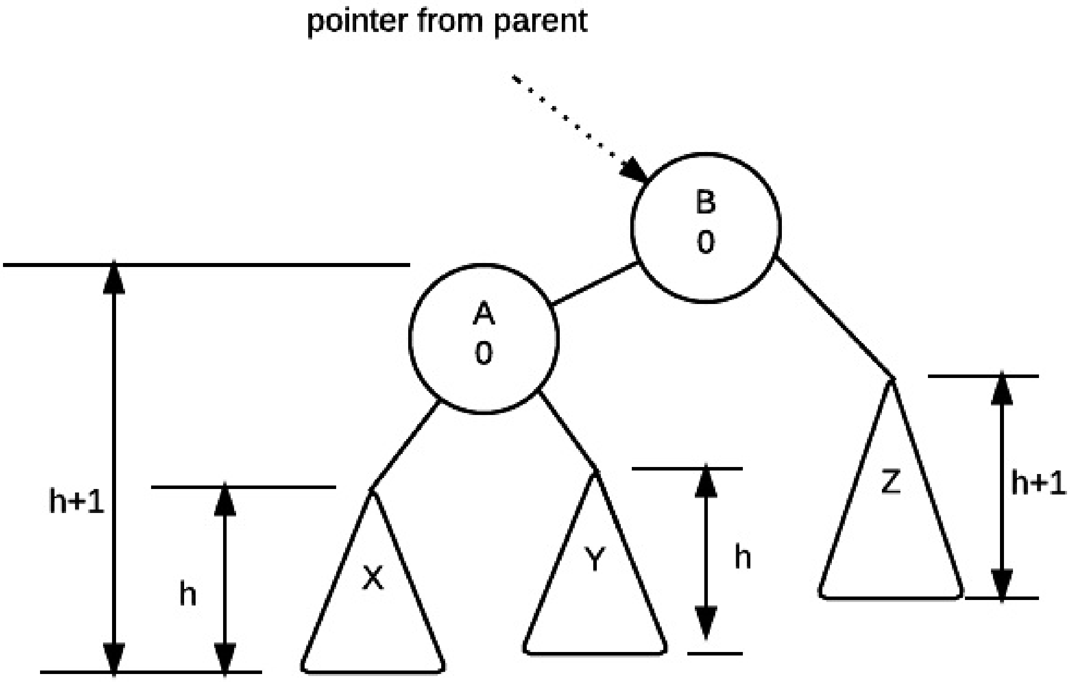
While we don't know how tall X, Y and Z are, we know their relative heights because we know the height balance.

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 X the right child of A.

Given that X, Y and Z are unchanged, the height balance at A and B will become 0.

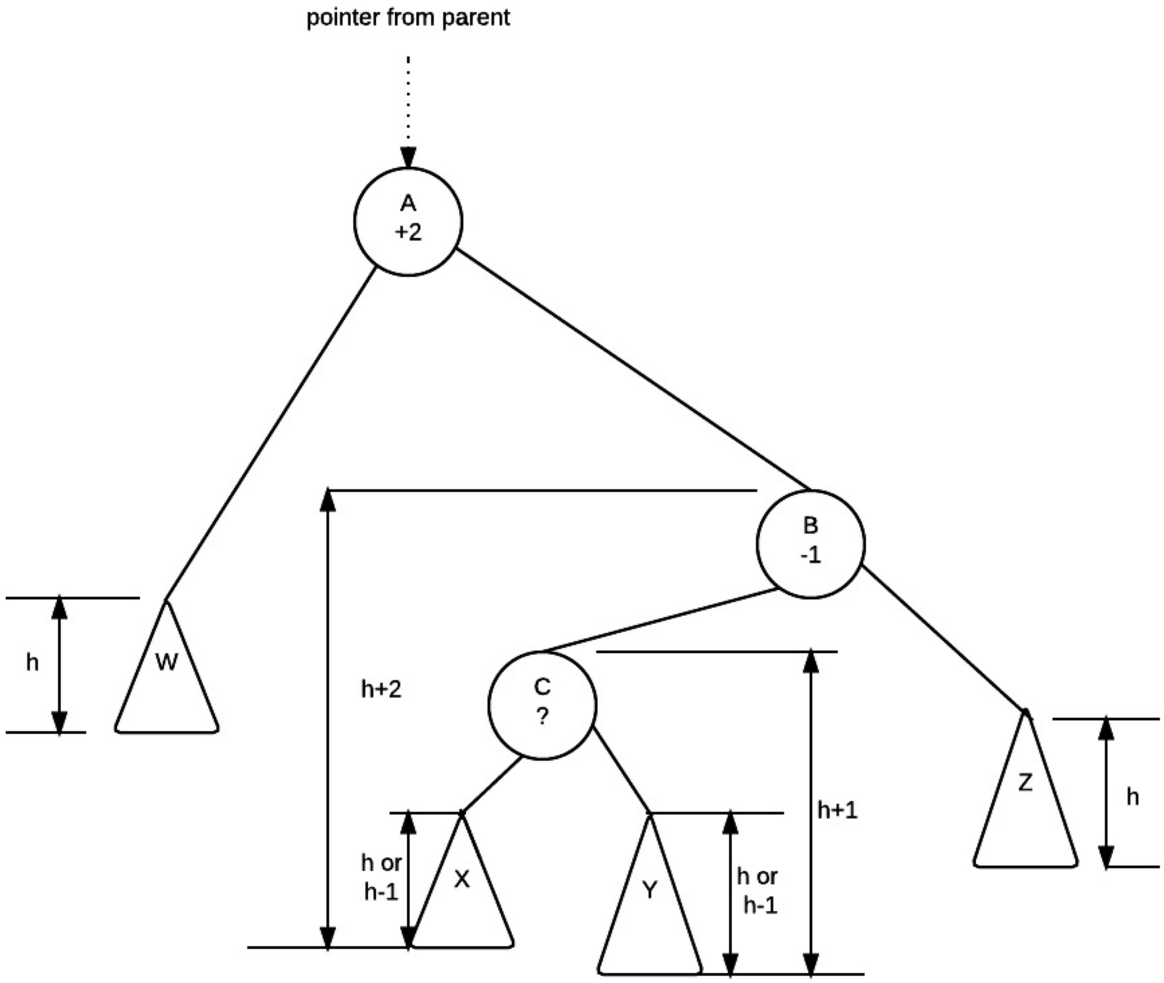


The mirror of the above is true for single left rotation

## Double Rotations

A similar explanation of why double rotations work can be reasoned out by looking at the following tree:

Why it works



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.

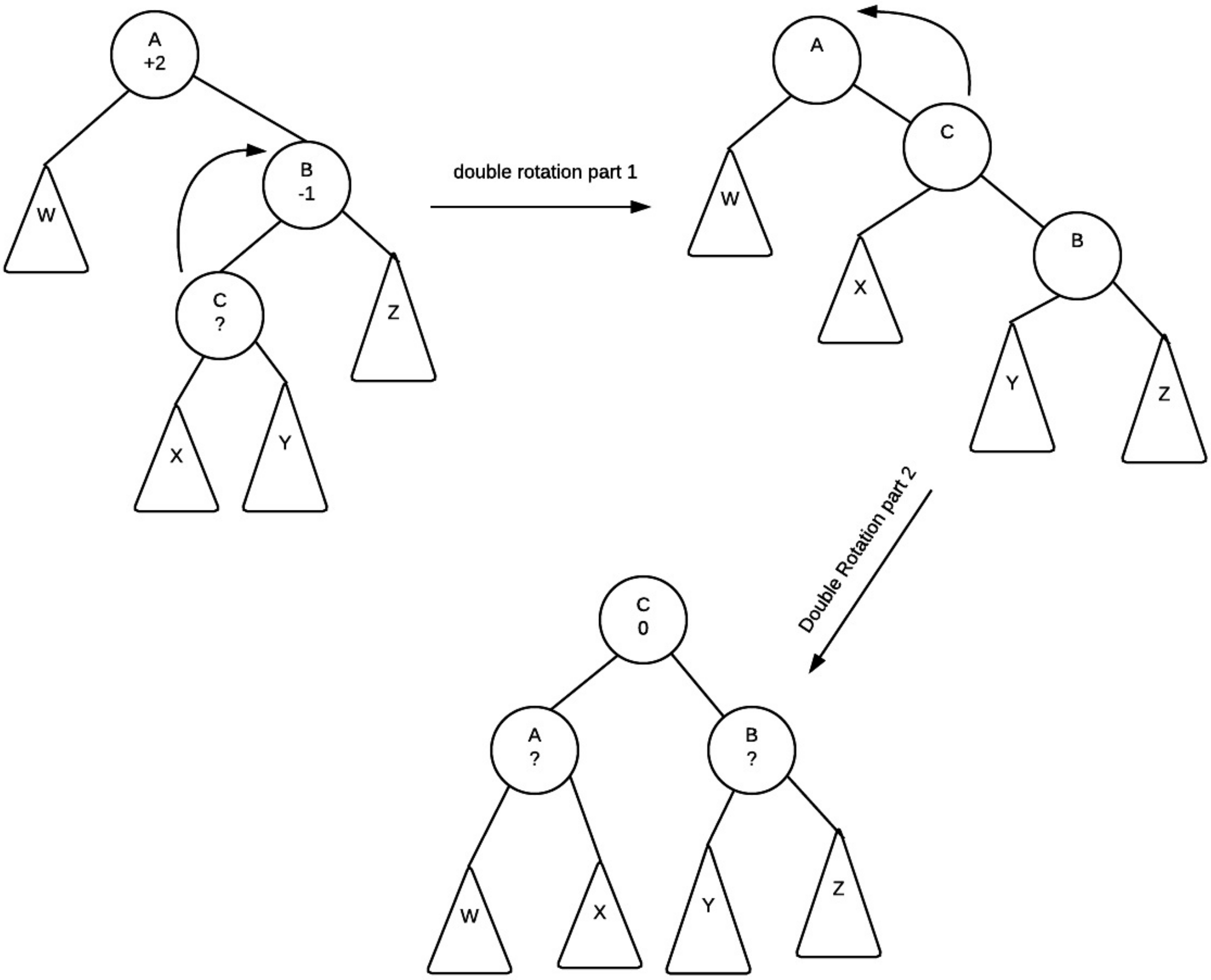
So, if C's height balance is 0, then both x and y will have height of h.

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:

## Why it works



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 |

# Red Black Trees

# 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. null nodes are considered to be black.
4. Every path from root to null pointer must have exactly the same number of balck nodes.

## 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. New nodes are added as red nodes. "Fix" the tree starting at the parent of the newly inserted node so that none of the rules are violated.

### General Insertion Algorithm

To insert into a red-black tree:

1) find the correct empty tree 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:

1. If a node is red at the root, change it to black.
2. If the new node is red, and its parent is black you don't need to do anything.
3. If a new node is red and its parent is red, what you do depends on colour of sibling of the parent
    - if the sibling of the parent is black, a rotation needs to be performed
    - if the sibling of the parent is red, a color swap with grandparent must be performed

# Insertion Example

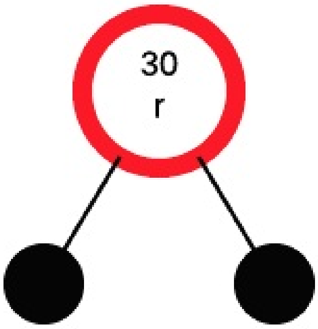
## Insertion Example

Starting with an empty tree let us take a look at how red-black tree insertions work.

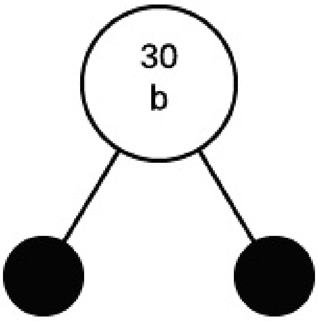
In the pictures below, null nodes (empty subtrees) are denoted as black circles

### Insert 30

All nodes are inserted as red nodes:

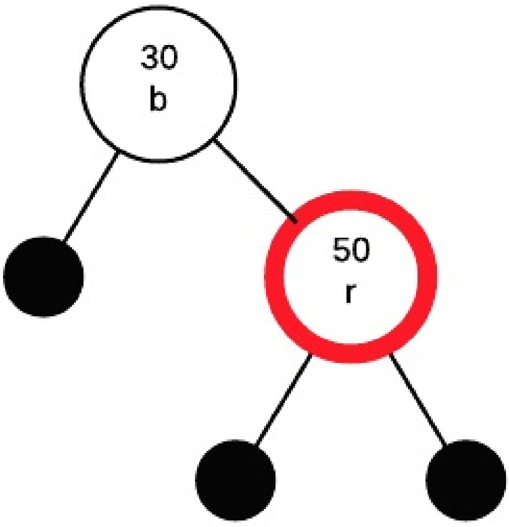


If the root is red, make it black:



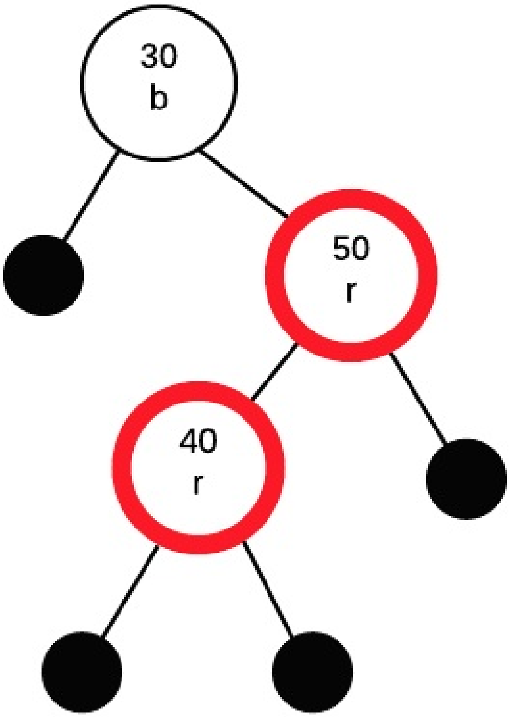
### Insert 50

Insert 50 as a red node, parent is black so we don't have to change anything



## Insert 40

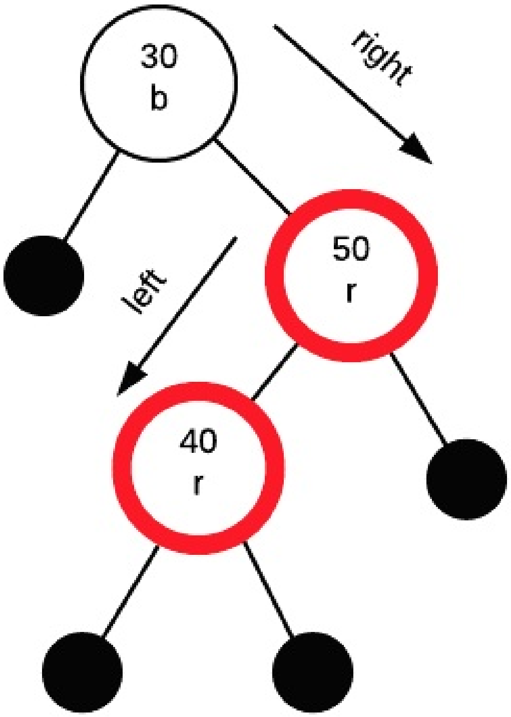
Inserting 40 as a red node.



If parent is red, we must apply fix. Parent's sibling (50's sibling) is black, so we must perform a rotation.

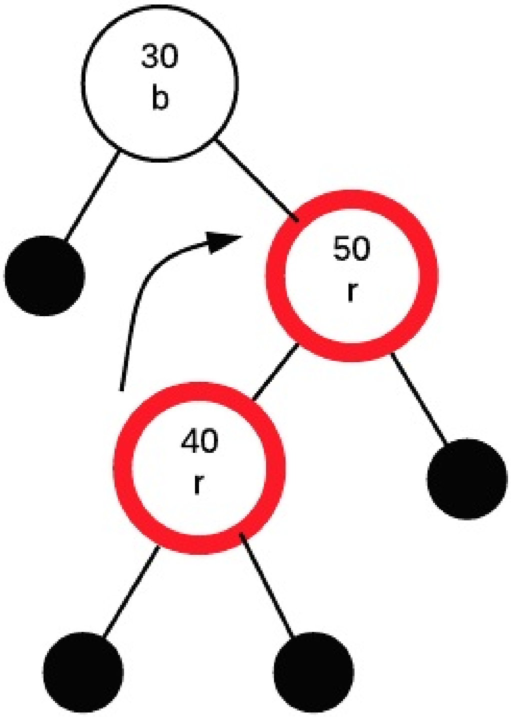
The type of rotation depends on the insertion pathway. If the insertion path from grand parent to parent to node is straight (both left or both right) do a single rotation. If it is angled (left then right or right then left) we need to do a double.

## Insertion Example

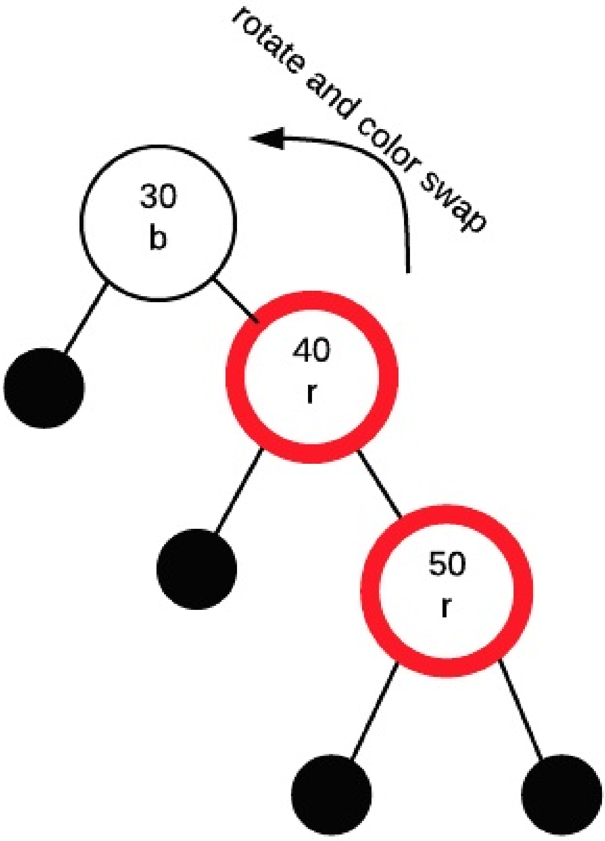


In this case, we need to do a double (aka zig zag) rotation.

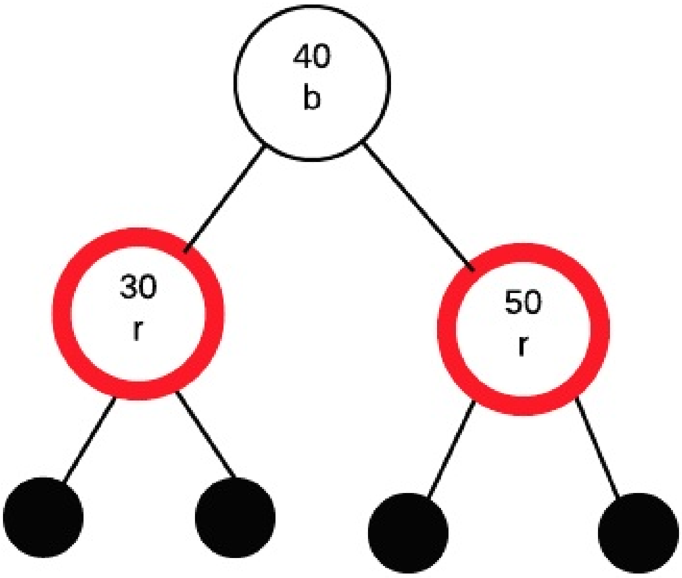
Rotate first 40 and 50



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.

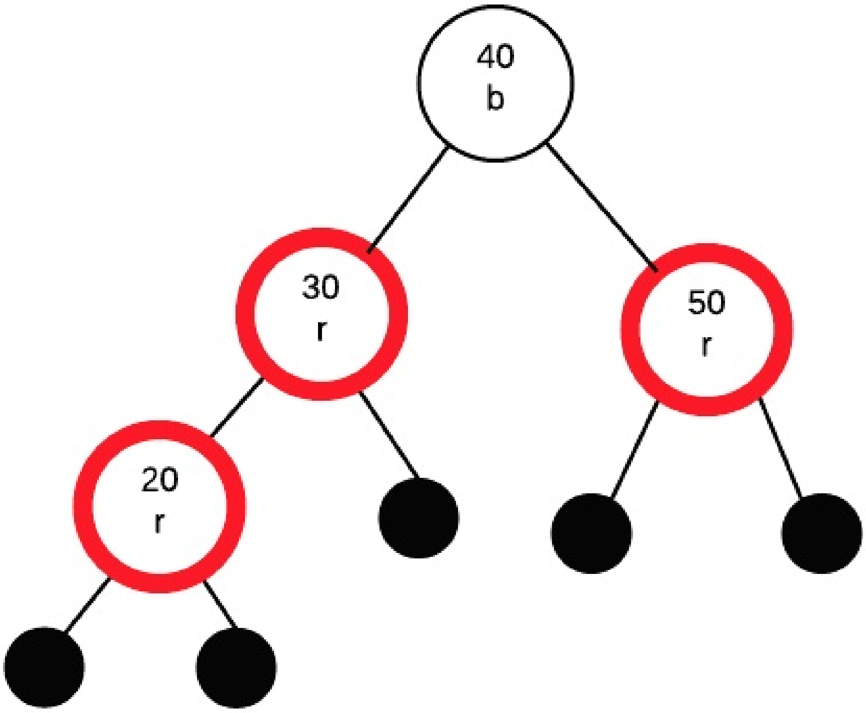


Finally we end up with:

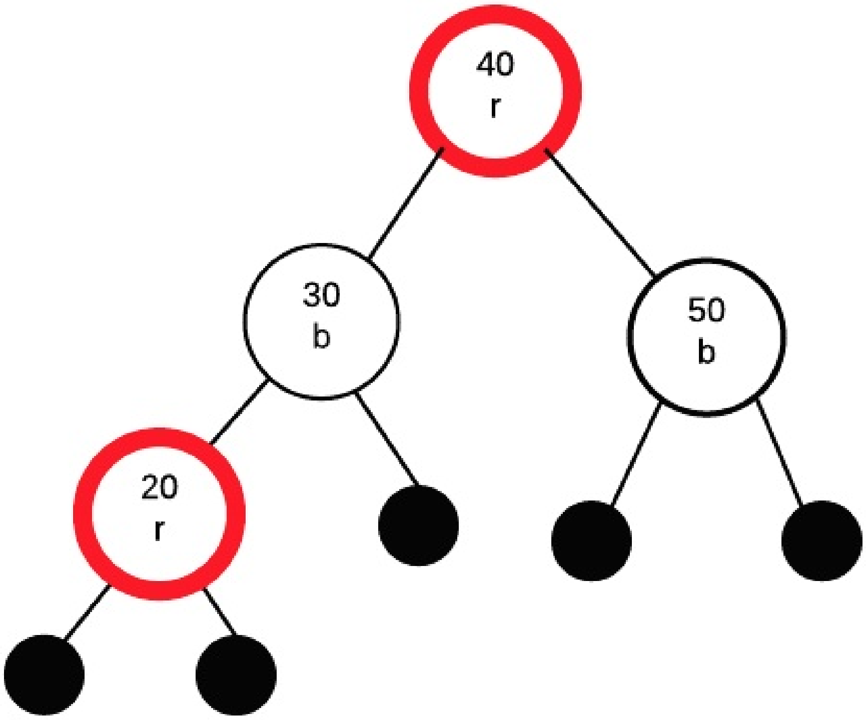


## Insert 20

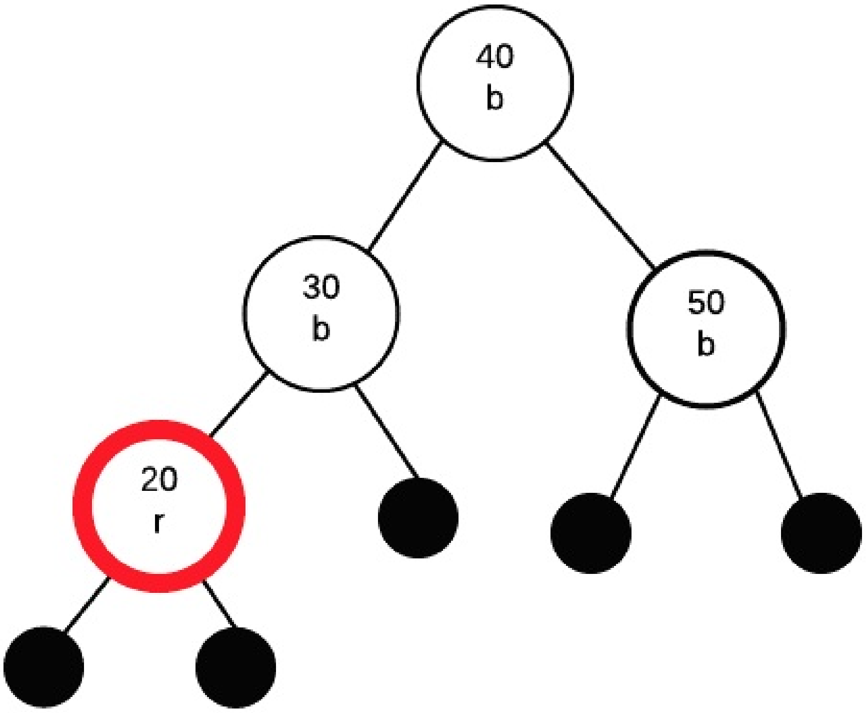
inserting 20 as a red node.



If parent is red we must apply a fix. Parent's sibling is red so we need to do a colour swap between parent+sibling and grandparent.

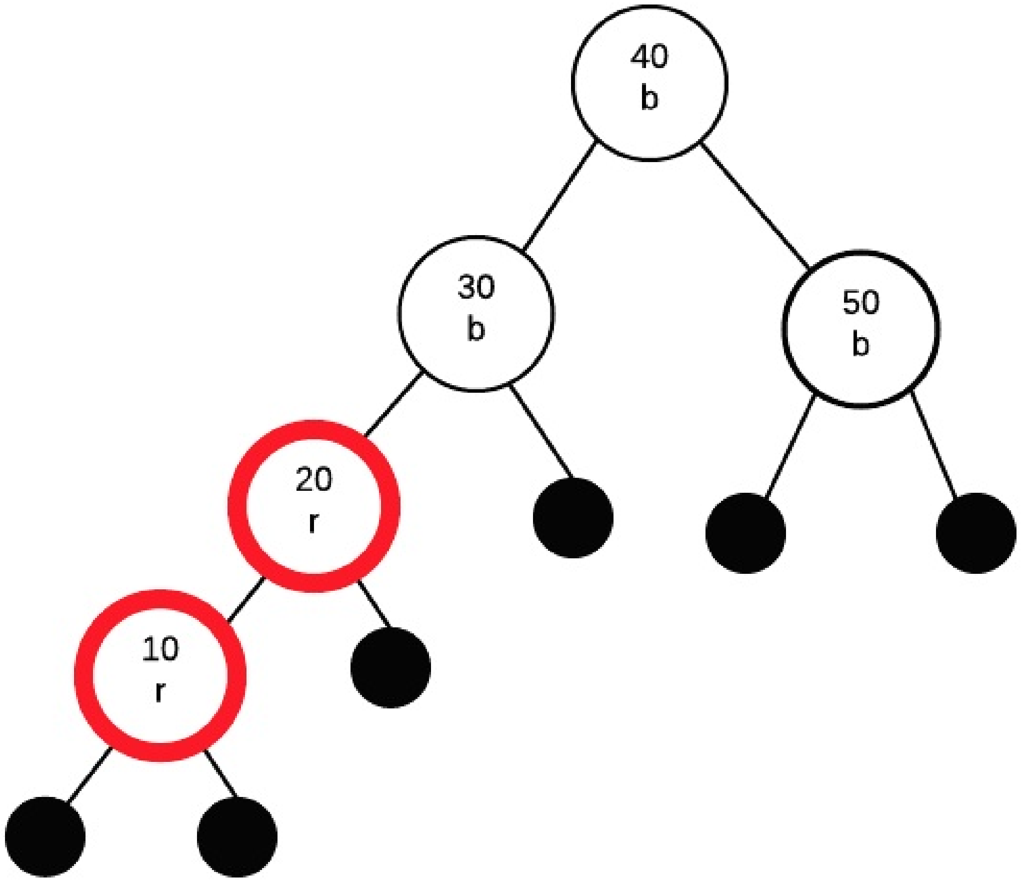


Now as 40 is the root of the whole tree we must change 40's colour to black. As it is the root, we can just change it to black without causing other problems.

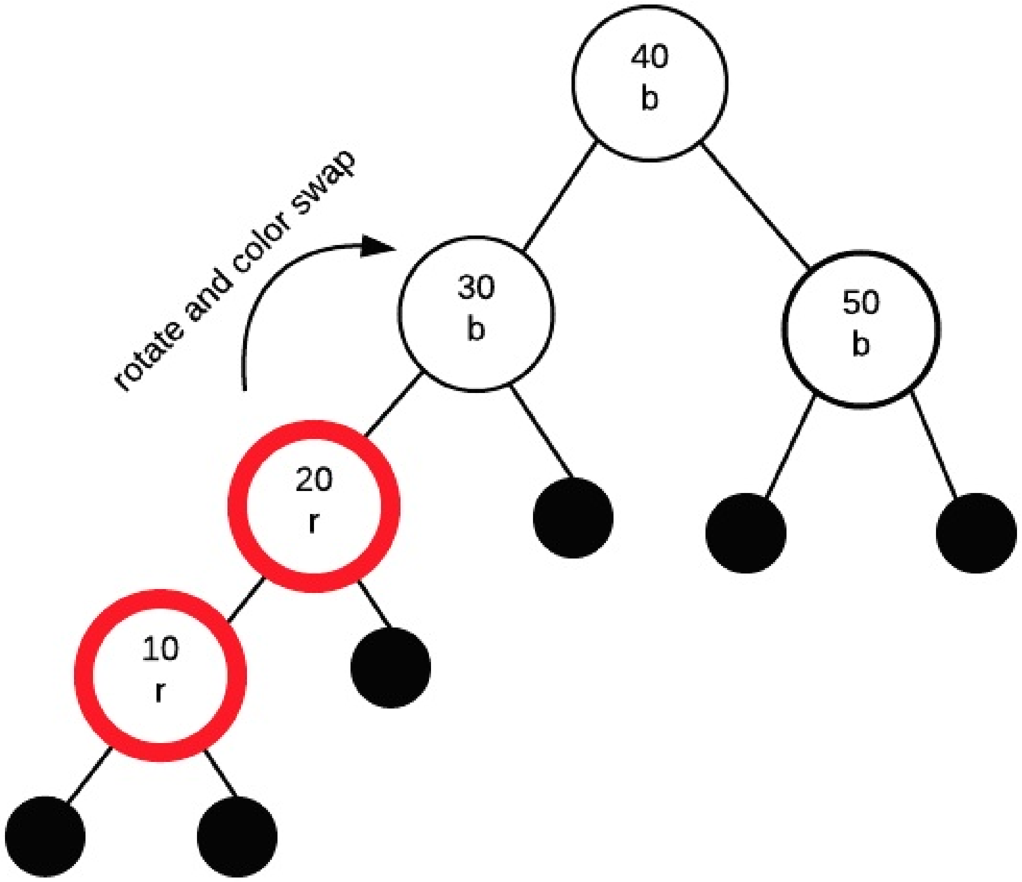


## Insert 10

inserting 10 as a red node.

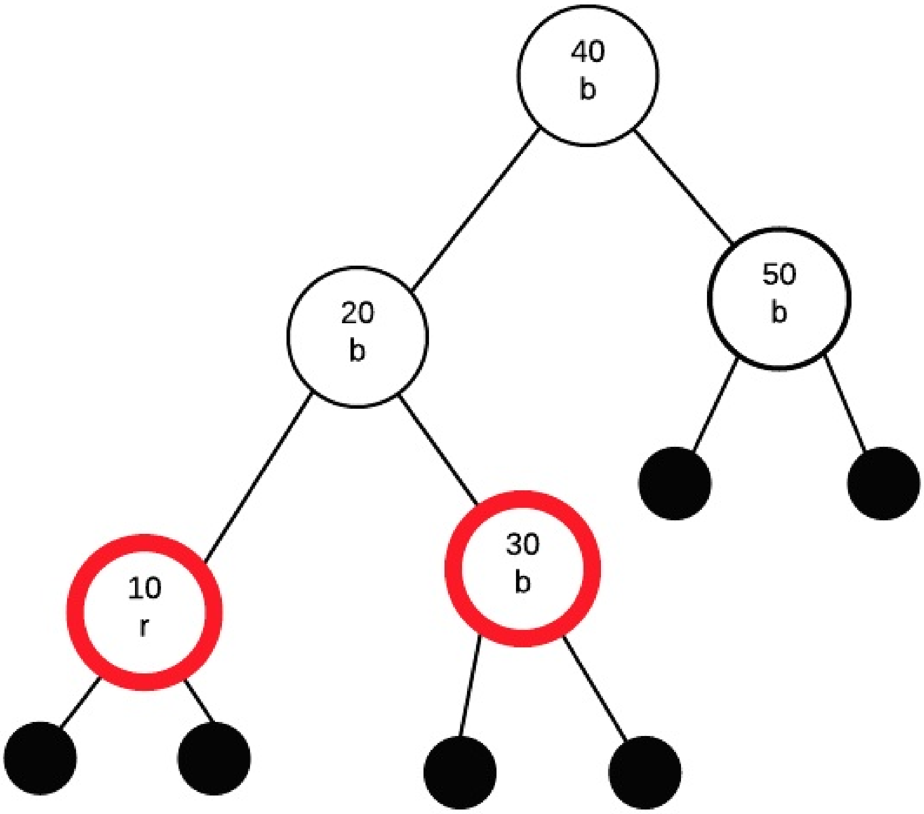


If parent is red, we must apply a fix. Parent's sibling is black, so a rotation is needed. This time, the insertion path from grandparent to parent to node is "left" then "left". Thus, we only need to perform a single rotation and colour swap.



Finally we get:

Insertion Example



# 2-3 Trees

# 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

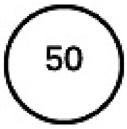
## Insertion

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:

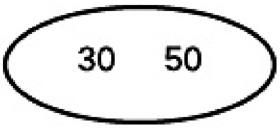
1. If the tree is empty, create a node and put value into the node
2. Otherwise find the leaf node where the value belongs.
3. If the leaf node has only one value, put the new value into the node
4. If the leaf node has more than two values, split the node and promote the median of the three values to parent.
5. If the parent then has three values, continue to split and promote, forming a new root node if necessary

### Example:

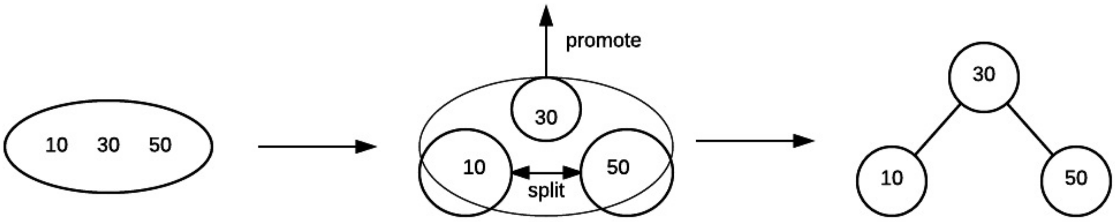
Insert 50

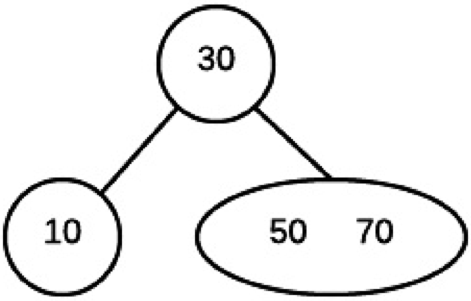


Insert 30



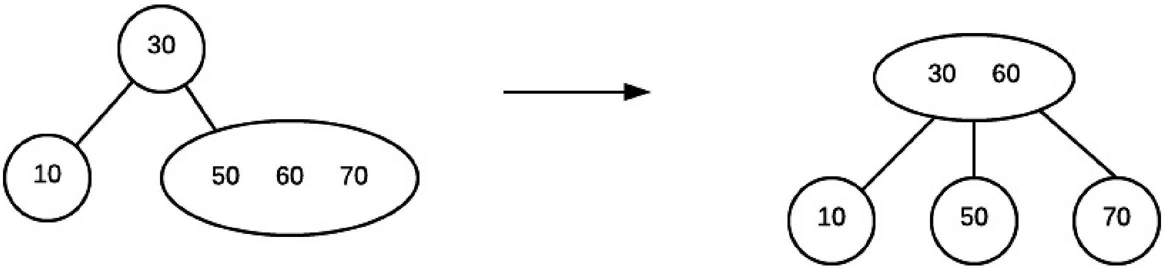
Insert 10





Insert 70

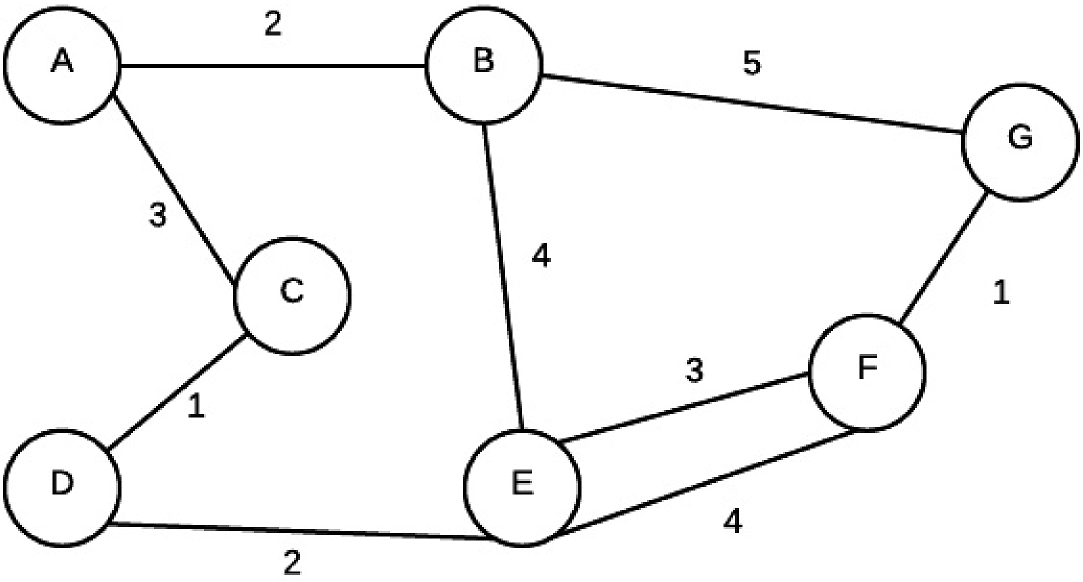
Insert 60



# Graphs

# Graphs

A graph is 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.



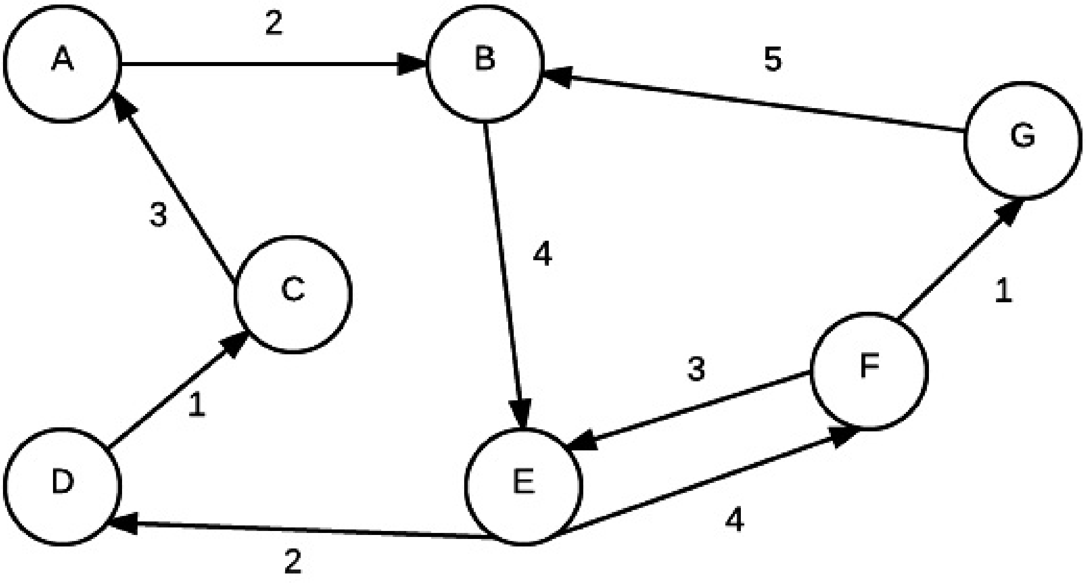
## Definitions

- 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
- pathlength number of 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 connect - 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

# 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

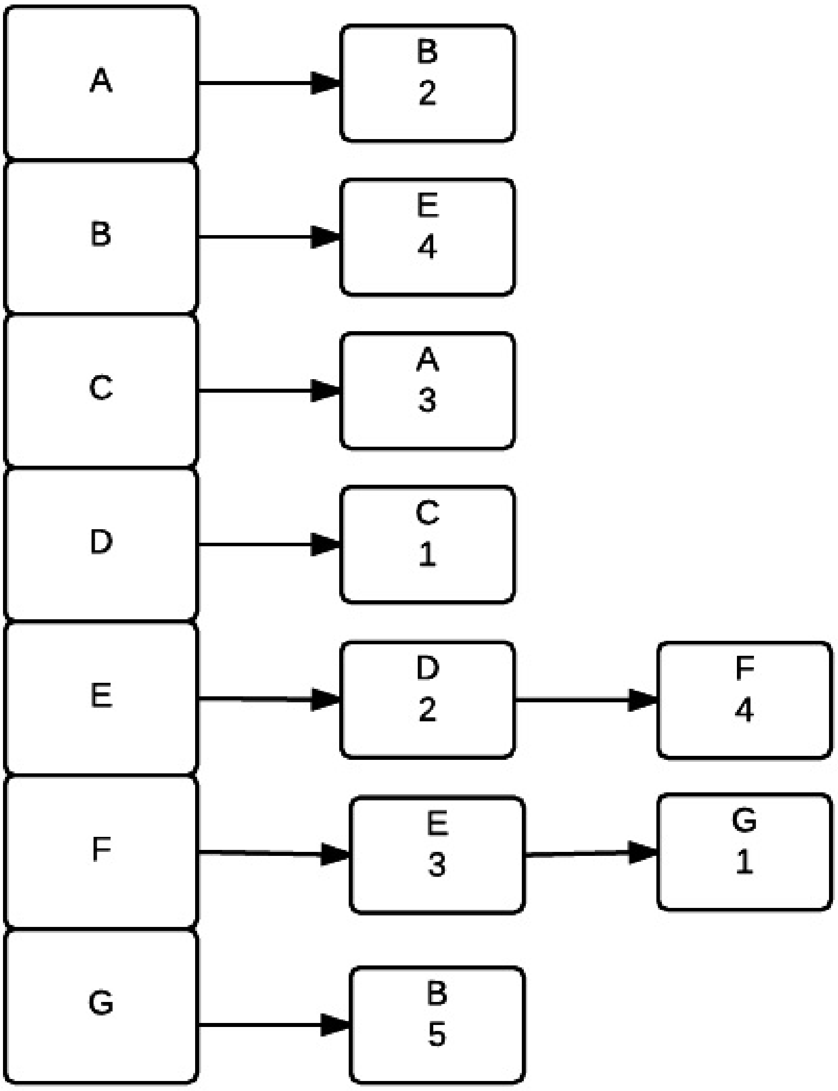
An adjacency matrix is in essence a 2 dimensional array. Each index value represents a node. When given 2 nodes, you can find out whether or not they are connected by simply checking if the value in corresponding array element is 0 or not. For graphs without weights, 1 represents a connection. 0 represents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

When a graph is dense, 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.

| | **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. For each vertex it is connected to, a node is added to it's 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.



# Complexity 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 how computer algorithms are classified according to its complexity. This section of the notes looks at complexity theory.

## 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 it 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 class Problems

Any problem that can be solved in polynomial time is considered to be class P. For example, sorting is a class P problem because we have several algorithms who's solution is in polynomial time.

## 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. Our computers today are deterministic machines. That is, instructions are executed one after the other. You can map out for a given input what the execution pathway of the program will be. 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 problem 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

# Appendix: Mathematics Review

# 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

## Mathematical Notations and Shorthands

### Shorthands

| shorthand | meaning |
|---|---|
| iff | if and only if |
| $\therefore$ | therefore |
| $\approx$ | approximately |
| $ab$ | $a * b$ |
| $a(b)$ | $a * b$ |
| $\lvert a\rvert$ | absolute value of $a$ |
| $\lceil a\rceil$ | ceiling, round $a$ up to next biggest whole number. Example: $\lceil 2.3\rceil = 3$ |
| $\lfloor a\rfloor$ | floor, round $a$ 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

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 $n^2 + n + 2$

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")

Note that when we talk about big-O notation (and related little-o, theta and omega notation) those are not functions (though it kind of looks like it)

### 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 mathetmatical 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 i (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

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**

$\log_b a = \frac{\log_c a}{\log_c b}$, where $c > 0$

$$\log ab = \log a + \log b$$

$$\log(\frac{a}{b}) = \log a - \log b$$

$$\log a^b = b\log a$$

$\log x < x$ 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**

$\sum_{i=0}^{n} c(f(i)) = c\sum_{i=0}^{n} f(i)$, where $c$ is a constant

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$$

$\sum_{i=0}^{n} a^i \leq \frac{1}{a-1}$ if $0 < a < 1$

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

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

$$\sum_{i=1}^{n} f(n) = nf(n)$$

$$\sum_{i=n_0}^{n} f(i) = \sum_{i=1}^{n} f(i) - \sum_{i=1}^{n_0-1} f(i)$$