Foundational Data Structures: Theory

By: Anthony Nguyen

December 7, 2013

Table of Contents

Introduction

This review is meant to be a cumulative review of ELEC 278 at Queen's University. Some diagrams have been taken off of Ahmed's lecture slides. This review package does not contain much code. It contains the detailed steps of how operations are done and is more theory based. It is good for the understanding of data structures and algorithms as well as for practice to know the steps and code these yourself. Ultimately what's important is that you understand these data structures for implementation in several languages, as these data structures are **widely implemented in industry and are fair game in interviews**.

I understand that a lot of the final exam and midterms are coding related, and so I will be making another package containing some code and explanation of the code.

1. Pointers and Structures

1.1 Pointers

A pointer is a variable that contains an address which points to a location in the memory. Pointers are used because often times it makes the program more efficient. This is due to the fact that pointers point to a memory location in the heap, which is dynamically allocated at runtime. This is opposed to normal variables where memory is permanently allocated, at compile time, in the stack.

A pointer variable is declared by putting the asterisk in front of the variable name (i.e. int *p;). Normally, pointers are used to point to the address of some other variable (i.e. p = &variable). The value stored at the address of the pointer can be accessed or manipulated by putting the asterisk in front (i.e. *p = 2 would store the value '2' wherever p points to).

In this course, the pointer is used to access memory in the heap that is dynamically allocated specifically for the pointer variable. The command *malloc* does this. To dynamically allocate memory to the pointer variable, you would set the pointer without the asterisk equal to the *malloc* command (i.e. p = (int *) malloc(sizeof(int))).

1.2 Structures

A structure is a collection of a bunch of variables grouped under one name. It's good because you can organize data conveniently. This is important when you are declaring a complex data structure like a linked list or even a single link.

Structures are declared like so:

```
1. typedef struct structure{ //Creates a new data structure type
2. a bunch of variables..
3. }
```

Items in the structure can be accessed one of two ways. If the item in the structure is a regular variable, then you access the variable in the structure by using structure.variable. If the item in the structure is a pointer, you would access the pointer by using structure->pointer.

2. Big Oh

2.1 Introduction

In the real world, when your programs are being benchmarked against other programs for efficiency, you want your program to run not only correctly but as quickly as possible. This is why we need to analyse the runtime and space of the algorithm to make sure it is as low as possible.

We don't just put two computers side by side to measure their runtime because that doesn't tell the whole story. It also takes a lot of work to debug/compile the program and then compare runtime. Using a few calculations, we can decide which algorithm is the better one.

A mathematical tool used to measure runtime and space used of an algorithm is **asymptotic notation**, or the **big Oh**. It gives a very basic but surprisingly accurate runtime for a specific algorithm in its **worst case scenario** (i.e. needing to traverse an entire list to find something).

2.2 Assumptions Made when Finding Theoretical Runtime

When we measure the runtime of an algorithm, we assume that the algorithm is working correctly and runs **sequentially** (i.e. in the task main the program runs one line at a time). We also assume that there is no compiler optimization (same runtime on all compilers).

Now these things cannot be ignored in actual runtime, and actual runtime depends on a number of things including hardware, compiler, and programming language.

2.3 Big Oh Definition

2.3.1 Relating Two Functions

Big oh is meant to be a simple representation of runtime or space based on the amount of data inputted. The function for big Oh is:

$$f(n) = O(n) = O(g(n))$$

f(n) is the actual function that determines runtime

O basically states that it is in big Oh notation and is simplified

n is the amount of data inputted in the algorithm

g(n) is the simplified function for runtime without a constant in front

Where does this hold true? How much data do you need for this to hold true? Well, the big Oh function holds true when n is big enough (we'll call the 'big enough' or 'threshold' n value n_o) and there is a constant **c** (c is greater than 0) so that

$$f(n) <= cg(n)$$

Whenever $n \geq n_o$

From this we can see the big Oh function multiplied by a constant is basically **an upper bound on the runtime actual function**.

2.3.2 Proving the Function Falls Under Big Oh

Example: Prove that $f(n) = 8n + 128 = O(n^2)$

1. Classify your f function and your g function. The g function is the one inside the big Oh. g(n) = n^2 and f(n) = 8n + 128.
2. To prove that the function falls under big oh, we need to find both some **constant** and its **threshold n value**. The constant can be chosen arbitrarily (or may be given in the question). We will pick c = 1.
3. Assemble the function $f(n) <= cg(n)$
$$8n + 128 <= n^2$$
4. Solve for the factors of n. These factors will be your n_o values. If there are any positive values for n_o then the condition is satisfied.
$$0 <= n^2 + 8n + 128$$
$$0 <= (n - 16)(n + 8)$$

There is a value n_0 that is positive for the constant c=1 which is also positive. Therefore,

$$f(n) = 8n + 128 = O(n^2)$$

2.3.3 Finding the Big Oh Function

Consider an arbitrary polynomial function, say f(n) = $6n^2$ + 3n + 2. At f(1), the function equals 11. The 2 in the function has a relatively large effect on the answer.

If we increase the value of n to something large, such as 100**, the value 2 has a very small impact on the final answer**. Increasing it even further makes **the 3n term have a small impact.** In fact, for very large numbers, the 6 coefficient in front of the n^2 has a low impact as well. This simplification is essentially what the big Oh notation is.

Big Oh simplification for polynomials is when you eliminate all of the lower growth terms (i.e. 3n, 2 in our example) first, and then eliminate the coefficient in front of the largest growth term, to create g(n), which is the simplified function. For the example above, g(n) = n^2, so that means:

$$f(n) = O\big(g(n)\big) = O(n^2)$$

2.4 Types of Big Oh

There are different functions that relate the data inputted to the runtime of the program. Some algorithms run efficiently, so the runtime of the program does not increase very much in comparison to the increase of the data input. Other less efficient algorithms will make runtime increase exponentially as more data is added. Below is a small list of big Oh functions and the explanation of the runtime.

2.4.1 O(1)

O(1) runtime basically says that no matter how big the array, data structure, etc., the algorithm will perform the operation at the same runtime. A good example of this is prepending a linked list. It doesn't matter if there are 2 or 200 links in the linked list, the prepend operation starts at the head of the list so there is no traversal involved. Similarly, if there is a tail for the list, the append operation also has O(1) runtime.

2.4.2 O(n)

O(n) runtime basically says that the runtime of the operation will be directly proportional to the size of the array, data structure, input, etc. A good example of this is traversing through a linked list. The time it takes to traverse the linked list is directly proportional to the amount of links in the linked list. A linked

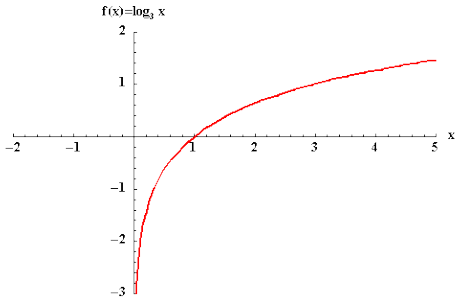
list with 2 links will be much quicker to traverse than a linked list of 200 links. It also applies to a **for loop**. The difference between the first two conditions of the for loop decide how quickly the operation will be done (i.e. for(i=0, i=n, i++) the runtime will depend directly on n so the big Oh function will be O(n)).

2.4.3 $O(n^k)$

$O(n^k)$ runtime sucks. This is because the runtime of the operation will be proportional to the size of the array to the power of k. In the notes, it'll show up as $O(n^2)$ or $O(n^3)$ or something like that, basically saying the same thing. An example of an algorithm that has $O(n^k)$ runtime is a **nested for loop**. It runs the for loop n times, but within the for loop, there is another for loop that also runs n times. This means the runtime would be proportional to n*n which is n^2. The number of nested for loops determines the **k** value.

2.4.4 O(log(n))

O(log(n)) runtime basically says that if the data inputted increases by a certain amount, the runtime will increase but a whole lot less than that of the data input increase, like the log function. Refer to this graph, where y is runtime and x is data input.



A "Refresher" on Binary Search

An example of O(log(n)) runtime is the **binary search**. If you don't remember from 142 ("remember" is a weird word to use because they never really taught it to us well….), binary search is this:

1. With your sorted, linear array of data, select your upper bound to be the last element in the array and your lower bound to be the first element in the array.
   Example: 1 2 3 4 5 6 7 8 9 10, we are searching for the value 8. Use 1 as the lower bound and 10 as the upper bound.
2. Choose the value directly in the middle of the lower bound and the upper bound. Compare it to the value you are looking for. If it is the value you're looking for, then great, but if not, we move on.
   5 isn't 8 so we haven't found our value
3. If the value you are looking for is greater than the middle bound, set the lower bound to the middle bound. If the value you are looking for is lower than the middle bound, set the higher bound to the middle bound.
   8 is greater than 5 (thanks Sherlock) so we set the new lower bound to be equal to 5.
4. Repeat steps 2-3 until you have found your value.

The binary search algorithm statistics for runtime are impressive. An array of several hundred thousand values only takes seconds to search in its worst case scenario. This is O(log n) runtime.

2.4.5 O(nlog(n))

N log n is basically a combination of n runtime and log n runtime. This is worse than n and log n runtime on their own. An example of this is running a for loop for a binary search.

2.5 Tight Vs Loose Big Oh

Remember that first example I used for proving the function falls under big oh? Notice how the highest exponent of the actual function is 1, while big oh is $O(n^2)$. Since the condition still holds true, the big Oh holds true, but only as a **loose big Oh** because something is able to slip through, i.e. a bigger exponent value.

Tight big Oh is when the highest exponent of the original function is the same as the exponent of what's inside the big Oh function. Refer back to the "finding the Big Oh function" section for an example of big Oh.

2.6 Big Theta and Omega

2.6.1 Big Omega

Big omega is similar to big Oh, but instead of acting as an upper bound, it acts as a lower bound. The mathematical definition is that:

$$f(n) >= cg(n)$$

Whenever $n \geq n_o$

2.6.2 Big Theta

Big theta basically means that the function relating the runtime (g(n)) is an upper and lower bound on the runtime. This means that the relating function g(n) is both **big Omega and big Theta** of the runtime. The condition is that there is exists two constants c1 and c2 in which:

$$c1g(n) \leq f(n) \leq c1g(n)$$

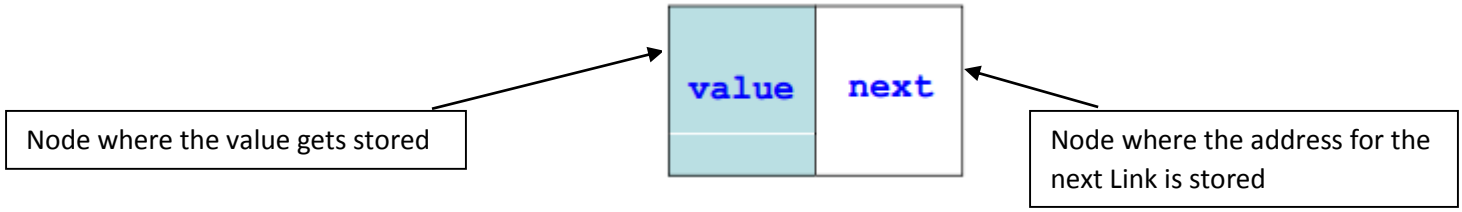
3. Linked Lists

3.1 Introduction

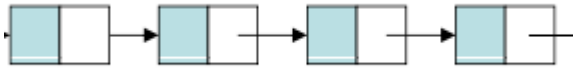
A **linked list** is a complex data structure which puts a bunch of **links** together in a sequence. A **link** is a structure (declared in the program or the header files) that contains at least two nodes:

-Value, which is where the actual data is stored in the link

-Some pointer that points to the next or previous link in the linked list. Some variations use a next pointer, some use a previous pointer, and some use both



A single link inside a Linked List



An entire linked list

Every linked list structure will contain a pointer link that acts as either the beginning or the end of the linked list. For example, the standard linked list structure (variation 1) contains a pointer link called the **head** which points to the beginning of the linked list. In another variation, there could be a pointer link called the **tail** which points to the end. The purpose of these pointers is to show the program where the linked list starts or ends. If the program didn't have one, it wouldn't know where to begin.

Now, when declaring a linked list, there needs to be an existing structure for the **entire linked list** as well as the **link itself**. The entire linked list structure (in lecture note it's called llist) would the head/tail pointer and an integer for the size of the linked list. The link structure would contain the nodes.

Real Life Comparison (Variation 1)

Suppose you have a five piece puzzle, but someone scattered the pieces of the puzzle around your house. At the location of each piece, there is the puzzle piece (obviously) and a note that tells you where the next piece of the puzzle is. The person also left a note on your front door that tells you where the first piece is, so you know where to start.

The note on the door is the **head pointer**, because it shows you where the first piece is.

The piece of the puzzle and the note directing you to the next piece together make up a **Link**.

The puzzle piece itself is the **Value** node**.**

The note that comes with the puzzle piece which directs you to the next puzzle piece is the **Next** node.

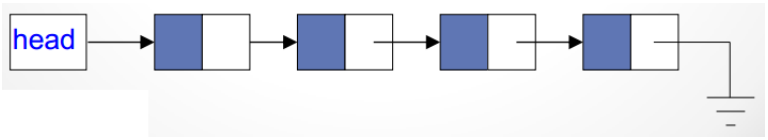
All five pieces, together, make up the **Linked List**.

3.2 Linked List Variations

There are four variations of linked lists. The variations differ based on what kind of pointer node they have as well as the head/tail pointer.

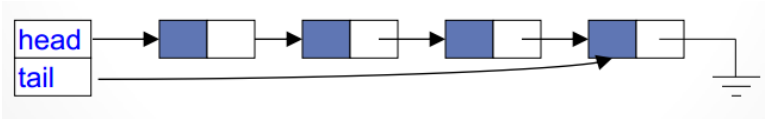
Variation 1

The variation 1 linked list is the standard linked list. The **link** structure contains a **value node** and a **next pointer node**. The actual linked list structure contains the **size** and a **head pointer link**.



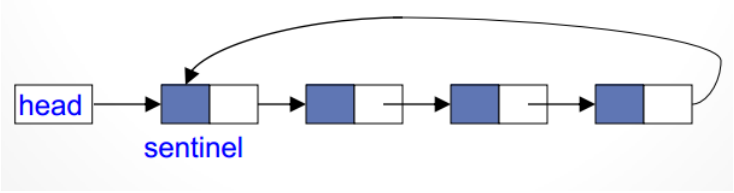
Variation 2

The variation 2 linked list is similar to variation 1 in the sense that the **link** structure is exactly the same. What makes it different is that the linked list structure contains a **tail pointer link** in addition to the **head pointer link**. The addition of the tail pointer makes attaching items to the end of the linked list much easier.



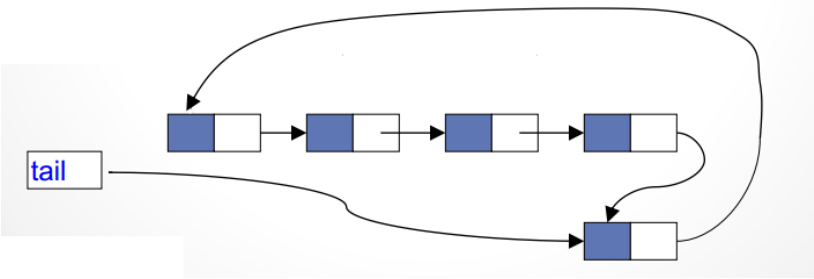
Variation 3

The variation 3 linked list contains a **sentinel link**, which acts as the beginning of the node. The sentinel link contains no value and acts as the first link in the linked list. It points to the actual first link in the linked list. The last link in the linked list points back to the sentinel link, making the linked list cyclical. The sentinel link is pointed to by the **head pointer link**, and accessed that way as well. The link and linked list structure is exactly the same as that of variation 1.



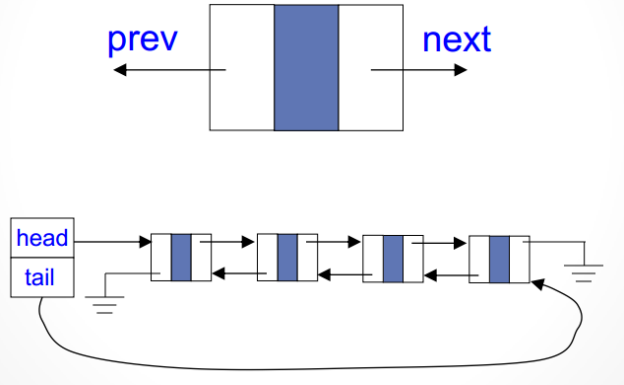
Variation 4

The variation 4 linked list contains only a tail pointer link that points to the last link in the linked list. This linked list is cyclical, as in the tail link points to the first link in the linked list. This actually makes it easy to identify the tail and the head.



Doubly Linked List

The doubly linked list is a linked list but is not one of the four variations. Each link in the doubly linked list contains a value node, a **next** node, and a **previous** node.



3.3 Linked List Operations

There are a number of operations you can do to a linked list. You can append, prepend, remove front, and remove last. There are also algorithms to find the first and last link of a linked list. These operations vary for each variation. I'm not going to code these, and I'm just going to write pseudo.

3.3.1 Find First Link

Variations 1 and 2

- The function would check to see if the linked list is empty
- If the linked list is not empty, it will return the link at which the head pointer is pointing to

Variation 3

- The function would check to see if the sentinel link is pointing to anything
    - If it isn't, nothing is returned
    - If it is, then it returns the link that the '**next' pointer in the sentinel link is pointing to**

Variation 4

- The function would check to see if the linked list is empty
- If it is not empty, it will return the link that the '**next' pointer in the tail link** is pointing to

3.3.2 Find Last Link

Variation 1

- Check to see if the linked list is empty
- If the linked list is not empty, it will check to see if the link points to null (it starts at head the first tim)
    - If the link does point to null, return the link
    - If it does not, it recursively runs the find last link function again passing off the next link in the linked list

Variations 2 and 4

- The function will check to see if the linked list is empty
- If it is not empty, it will return that link that the **tail pointer link** points to

Variation 3

- The function will check to see if the sentinel link points anywhere
- If it does, then it will check the link that is passed to see if it points back to the sentinel link
    - If it does, the function will return the link
    - If it does not, the function will recursively run, but passing off the next link

3.3.3 Remove from Front

Variation 1 and 2

- The function will check to see if the linked list is empty
- If it is not empty, it will get the first link in the linked list
- It will make the head pointer link **point to wherever the first link points with its next pointer**
- It then decreases the size of the linked list by 1

Variation 3

- The function will check to see if the sentinel link points anywhere
- If it does point somewhere, it will make the sentinel link's *next* pointer point to wherever the **next link's *next pointer* points** (yeah I realize it's not a great sentence so here's a diagram)

Variation 4

- The function will check to see if the linked list is empty
- If it isn't empty:
    - It will get the last link in the linked list
    - It will then get the link that the last link is pointing to, we'll call this the first link
    - It will make the last link point to wherever the **next pointer of the first link points**

3.3.4 Remove Last Link

Variation 1

- Check to see if the linked list is empty
- Check to see if there is only one link. If there is only one, then remove the lone link by making head point to null, and end the function.
- Find the link previous to the last link (i.e. modify the find last link code to check if link->next->next points to null)
- Make the link previous to the last link point to null, effectively removing the link
- Decrease the size of the linked list by 1

Variation 2

- Check to see if the linked list is empty
- Check to see if there is only one link. If there is only one, then remove the lone link by making head point to null.
- Find the link previous to the last link (check to see if the link points to the tail's link)

- Make that link point to null, and make the tail pointer point to that link
- Decrease the size of the linked list by 1

Variation 3

- The function will check to see if the sentinel link points anywhere
- The function will then check to see if there is only one real link in the linked list. If there is, then it will make the sentinel link point to null.
- If there is more than one link, it will then find the link previous to the last link (check to see if link->next->next = llist->head)
- It will make this previous link point to the sentinel
- It will then decrease the size of the linked list by 1

Variation 4

- The function will check to see if the linked list is empty
- It will also check to see if there is only one link in the linked list. If there is, then it will make the tail pointer link point to null
- If there is only one link, it will traverse the list to find the link previous to the tail (check if the link->next points to tail)
- Once it finds this previous link, it will make the previous link point to wherever the tail link's next pointer points.
- It will then make the tail pointer point to the previous link
- It will then decrease the size of the linked list by 1

3.3.5 Prepend

Prepending is adding a link to the beginning of the linked list.

Variation 1

- The function checks to see if the linked list is empty
  - If it is empty, then it makes the head pointer point to the new link
- If the linked list isn't empty, it does the following steps:
  - It makes the new link point to the address stored in the head pointer
  - It then makes the head pointer point to the new link
- The size of the linked list is then increased by 1

Variation 2

- The function checks to see if the linked list is empty
  - If it is empty, it makes the head and tail pointer point to the new link
- If it isn't empty, it will run the following steps:
  - It makes the new link point to the address stored in the head pointer
  - It then makes the head pointer point to the new link
- The size of the linked list is increased by 1

Variation 3

- The function checks to see if the sentinel link is pointing to anything
  - If it isn't, it makes the sentinel link point to the new link
- If the sentinel link points somewhere:

    - It makes the new link point to wherever the sentinel link points
    - It then makes the sentinel link point to the new link
- It then increases the size of the linked list by 1

Variation 4

- The function checks to see if the linked list is empty
    - If it is, it makes the tail pointer point to the new link
- If the linked list isn't empty:
    - It makes the new link point to wherever the last link's next pointer points
    - It makes the last link's next pointer point to the new link
    - It makes the tail pointer point to the new link
- It then increases the size of the linked list by 1

3.3.6 Append

Variation 1

- The function checks to see if the linked list is empty
    - If it is empty, then it makes the head pointer point to the new link
- If the linked list isn't empty, it does the following steps:
    - It finds the last link with the find last link function **or** finds the last link by traversing the link in a loop until link->next points to null
    - It makes the last link's next pointer point to the new link
- The size of the linked list is then increased by 1

Variation 2

- The function checks to see if the linked list is empty
    - If it is empty, it makes the head and tail pointer point to the new link
- If it isn't empty, it will run the following steps:
    - It makes the last link (tail) next pointer point to the new link
    - It makes the tail pointer point to the new link
- The size of the linked list is increased by 1

Variation 3

- The function checks to see if the sentinel link is pointing to anything
    - If it isn't, it makes the sentinel link point to the new link
- If the sentinel link points somewhere:
    - It finds the last link with the find last link function, **or** it traverses the link in a loop until the link->next points to the sentinel link
    - It makes the last link point to the new link
    - It makes the new link point to the sentinel link
- It then increases the size of the linked list by 1

Variation 4

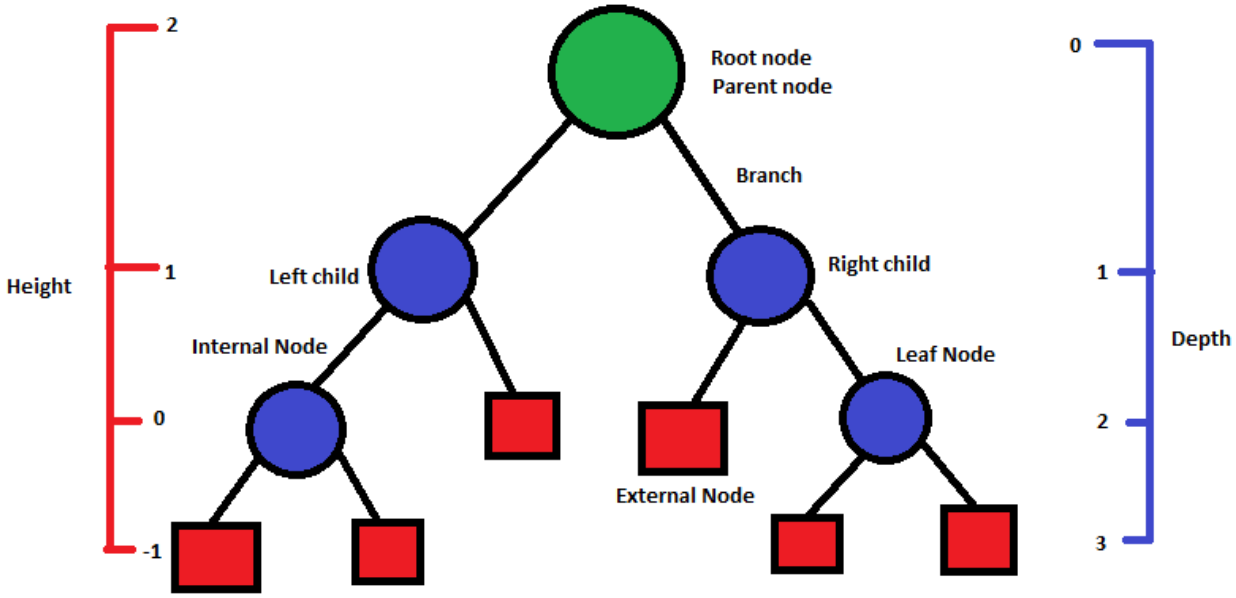
- The function checks to see if the linked list is empty
    - If it is, it makes the tail pointer point to the new link
- If the linked list isn't empty:
    - It makes the new link point to wherever the last link's next pointer points

    - It makes the last link's next pointer point to the new link
    - It makes the tail pointer point to the new link
- It then increases the size of the linked list by 1

4. Trees

Linked lists suck because you can't find things quickly. A tree is basically a data structure with hierarchies that make it easy to traverse and easy to look for values. A sorted tree can be searched in O(log n) time rather than a linked list which on worst case scenario takes O(n) time to search. One thing to consider is that you can't have two of the same value in a tree.

A tree looks like this:



4.1 Terminology

Node: Something that contains a value and pointers for possible child nodes

Child node: Any node that has a node above it that links to it.

Internal node: Any node that contains a value

Parent node: Any node that contains child nodes

External node: A node with no value, used as a sentinel for internal nodes without child internal nodes

Leaf node: An internal node that has two external nodes as children

Depth: Distance from the root to the specific node

Height: Distance from the leaf node to the specific node

External path length: Sum of the depths of all of the external nodes

Internal path length: Sum of the depths of all the internal nodes

N-ary tree: A tree where the parent node can have **n** number of children. (i.e. **Bi**nary tree means that parent nodes can have a maximum of two children).

Root: The top node of the tree

The trees that we mostly deal with are **binary trees**. Binary trees have nodes with two child nodes: a left child node and a right child node.

4.2 Types of Tree Traversals (Binary Trees)

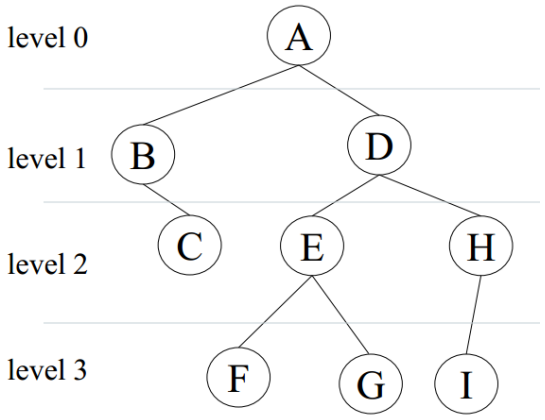
Traversal Steps: The code will check the node's value, left child, and right child, in the specified traversal order. When it checks the node's value, it looks for the value pointer within the node.

Pre order traversal: Traversal that involves checking the value of the node, then recursively going to the left child, and then recursively going to the right child. It checks the value of the node at the beginning hence the word **pre**.

Post order traversal: Traversal that involves recursively going to the left child, recursively going to the right child, and then checking the value of the node. It checks the value of the node at the end hence the word **post**.

In order traversal: Traversal that involves recursively going to the left child, checking the value of the node, and then recursively going to the right child. It checks the value of the node between the left child traversal and right child traversal hence the word **in**.

Breadth order traversal: This one is different. It traverses the nodes by descending height (levels) from left to right.



A,B,D,C,E,H,F,G,I

4.3 AVL Trees

Binary Search Trees

Before we talk about AVL trees, we must know what a **binary search tree** is. If you want your operation to be efficient, you can't just throw these nodes wherever you want. It has to be sorted so that a search algorithm can run properly. A binary search tree is a binary tree, so every parent node has two children. All values in the left subtree of the parent node are less than the value of the parent node, and all values in the right subtree of the parent node are greater than the value of the parent node. A binary tree has 2^{h+1}-1 nodes, where *h* is the depth of the tree. It has 2^{h} leaf nodes.

What is an AVL Tree

An AVL tree is a **type of binary search tree** with one distinct quality: the heights of the two children subtrees of any parent node cannot differ by more than one. The difference in height of the two subtrees is known as the **balancing factor**. The balancing factor is taken by subtracting the height of the right subtree from the height of the left subtree. It is **not absolute value** because the negative decides which rotation we're going to use (described later).

4.4 Binary Tree Structures

The nodeType Structure

The nodeType structure is the structure for each node inside the tree. The data structure "nodeType" is defined inside either a header file or within the actual *.c file. The node file consists of:

nodeType *_leftChild: This is the node pointer to the left child node.

nodeType *_rightChild: This is the node pointer to the right child node

char *_val: This is the value stored inside the node. In the assignment the value being stored inside the structure is a character, so we use char to declare the value node. This node can be any variable type.

The contents of the node can be accessed like any structure, using **nodeName->_____**.

The treeType Data Structure

The data structure "treeType" is also defined inside the header file and is used to declare an entire tree. The only member of the treeType data structure is **nodeType *root**, which contains the root node of the tree.

4.5 Binary Search Tree Operations

Again, I am going to write these operations in pseudo because the coding review will be a separate one.

4.5.1 Finding the right position for insertion

- Start your search at the root node (the function takes in the value being inserted as well as a node in the tree. The first time the function runs, the root is passed off)
- Compare the value being inserted to the value inside the node
    - If the value being inserted is the same as the node being compared it will return that node
    - If the value being inserted is not the same, it will check to see if the inserted value is greater than or less than the node's value. If the inserted value is less, it will check to see if the left child is null. If the inserted value is greater, it will check to see if the right child is null.
        - If the child is null, then we have found the correct node to attach our new inserted value
        - If the child is not null, then the function will run recursively, passing off the compared child as the root and the value being inserted.

4.5.2 Insertion

Now that we have a function for finding the right place to insert, we can go ahead and write our insertion function.

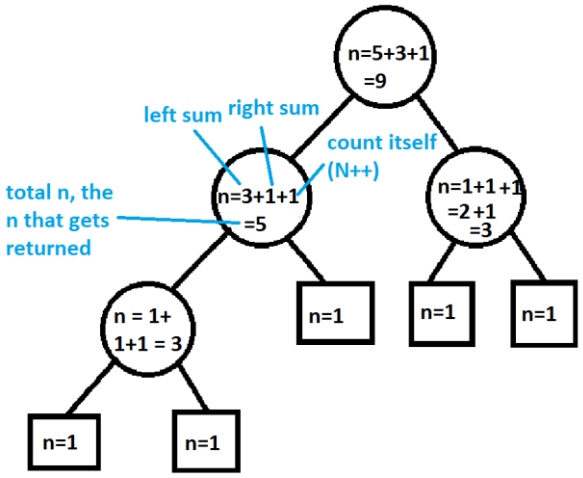
- The function accepts only a value for insertion, as well as the tree, so we have to create a new node to accommodate this value.
- Dynamically allocate memory for the node as well as the value being stored inside the node, and put the insertion value into the node's value part
- Set the left and right children of the new node to point to NULL
- Check to see if the tree is empty
    - If the tree is empty, make the root node of the tree the new node
    - If the tree is not empty, find the correct place to insert the new node using the previously defined function.
        - Compare the new node to the node returned by the function. If the two nodes' values are the same, then it will not insert. If the new node is greater, attach the new node to the right child of the returned node. If the new node is less, attach the new node to the left child of the returned node.
- Depending on how it's set up, the function could return a 1 or a 0 indicating a successful or unsuccessful insertion.

4.5.3 Calculating the Number of Nodes

This calculates the number of internal **and** external nodes. I'm just going to declare this all in one function. As inputs, the function takes in a node. It is an integer function as it returns an integer. On first calling, it takes in the root node of the tree. We are assuming a non-empty tree.

- Declare a counter integer
- If the node passed is null, it returns 1. This is to account for external nodes.
- If the node passed is not null, it does a post order traversal, which is basically:
    - It runs the function recursively on the left child of the tree. The result is given to the counter variable.
    - It runs the function recursively on the right child of the tree. The result is **added** to the counter variable.
    - It then adds 1 to the counter variable, to account for the node being passed.
- Finally, it returns the counter variable.

Here is a diagram showing how it works, and showing the counter variable at each level.



Calculating External Nodes Only

If you want to calculate external nodes, you would do the exact same thing as number of nodes, only you wouldn't add 1 to the counter variable after calling the function recursively. This is because we **do not want to take into account the node being passed**.

Calculating Internal Nodes Only

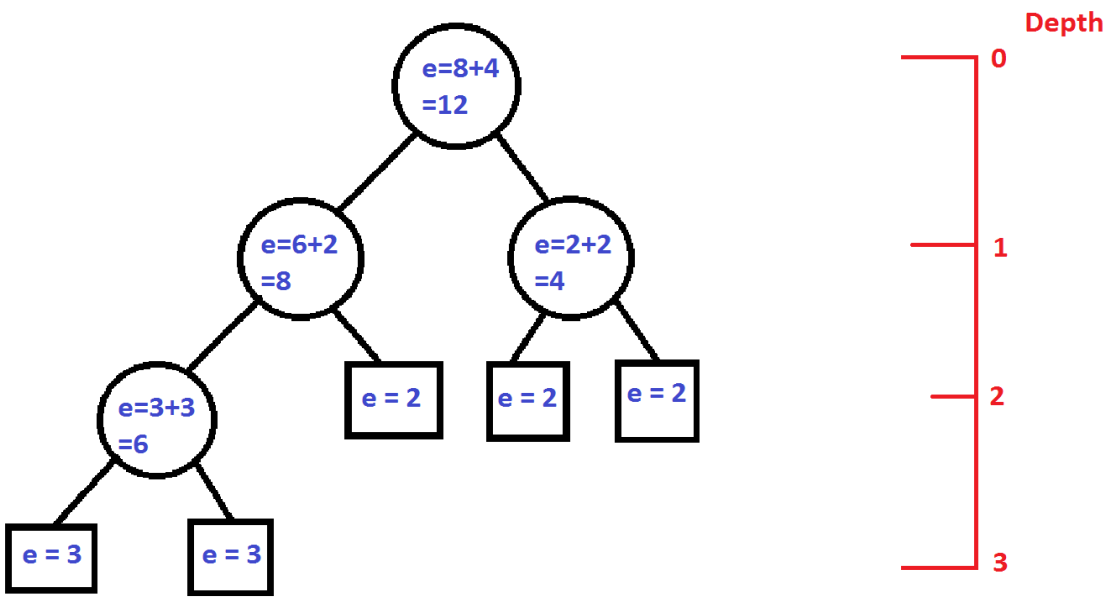
If you want to calculate the number of internal nodes, you would do the same thing as calculating the number of nodes, only if the node that's passed is null, a 0 would be returned (i.e. it doesn't count the external node).

4.5.4 Calculating the External Path Length

The external path length, again, is the sum of the depths of the external nodes. It is an integer function that takes in some node (initially the root) and the initial depth as input variables.

- Declare a variable to keep track of the sum of depths
- If the node passed is equal to NULL, then it will set the variable equal to the depth that is passed
- If the node passed not equal to null
    - It will run the function recursively, passing off the left child as the node and the current depth + 1 to account for the moving down in depth of 1. The returned value of the recursively run function is set equal to the variable declared earlier
    - It will do the same thing except this time it passes off the right child, and **adds the returned value to the variable**
- It then returns the variable

Here is a diagram showing how it works. The counter variable is e.



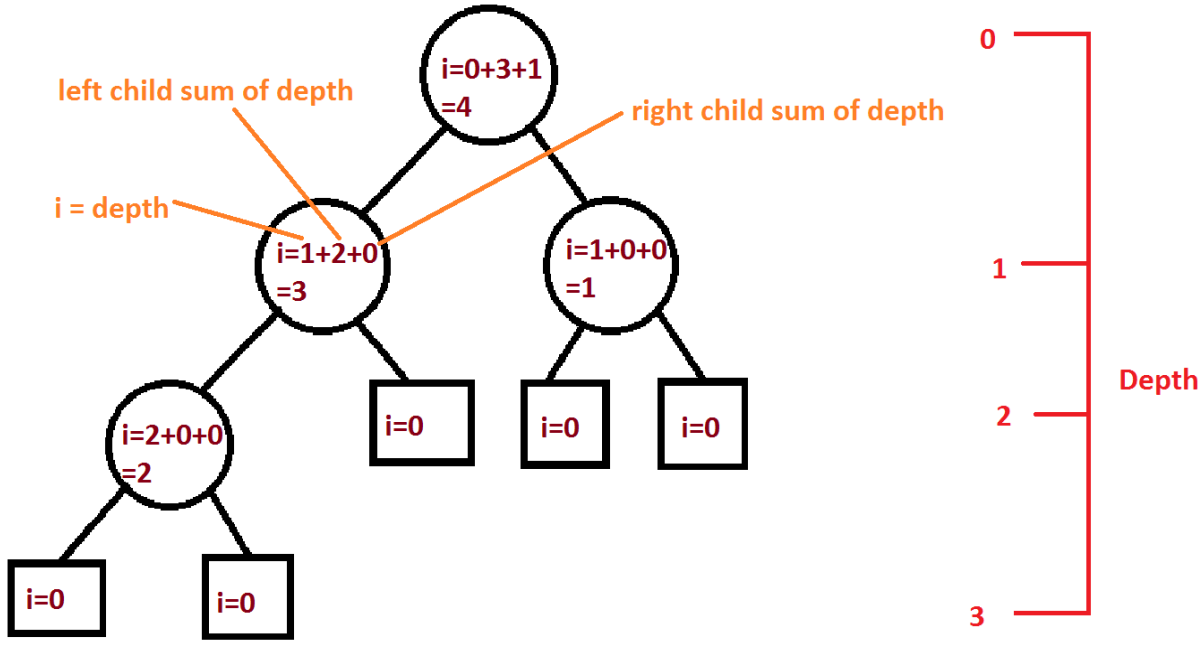
4.5.5 Calculating the Internal Path Length

The internal path length of a tree is the sum of the depths of the internal nodes of the tree.

The function declaration and initialization is very similar to that of the external path length function.

- Declare a variable to keep track of the sum of depths at the function
- Check to see if the node passed is null (indicating an external node). If it is not, it continues.
- It does a pre order traversal. First, it sets the variable equal to the current depth.
- It then calls the function recursively passing off the left child and the depth + 1 (just like the external path length function). The result returned is **added to the variable**.
- It then does the exact same thing except it passes off the right child instead of the left
- It then returns the variable keeping track of the sum of depths

Here is a diagram showing how it works, using i as the counter.



4.5.6 Freeing the Tree

To free the tree, we write two functions. The first function is used to free the entire tree and is used to call the helper function that frees the subtrees and nodes in the tree. It accepts the tree as input.

Free Tree Function

- It first checks to see if a tree exists. If it does, then it proceeds.
- It calls the helper function to free all of the subtrees and nodes. It sends off the root of the tree to this helper function.
- It then frees the actual tree.

Free Subtree Function

- It checks to see if the node is null. Obviously we don't want it to run the following steps if there is no node there.

- If it isn't null, it will run the function recursively on the left child and then the right child of the node. This is because in order for a node to be freed, all of the children of the node need to be freed first.
- Finally, it frees the node itself.

Again, this is a post-order traversal.

4.5.7 Finding the Height of the Tree

This function basically calculates the height of all the paths and returns the longest path. It takes in some node as input (at first it is the root node).

- Check to see if the node is null. If it is null, it will return 0.
- If not, it then runs the function recursively on the left and right children of the tree **simultaneously**, takes the greater result of the two, and add 1 to the result. All of this is done inside of the return statement. This gives you the height of the tree.

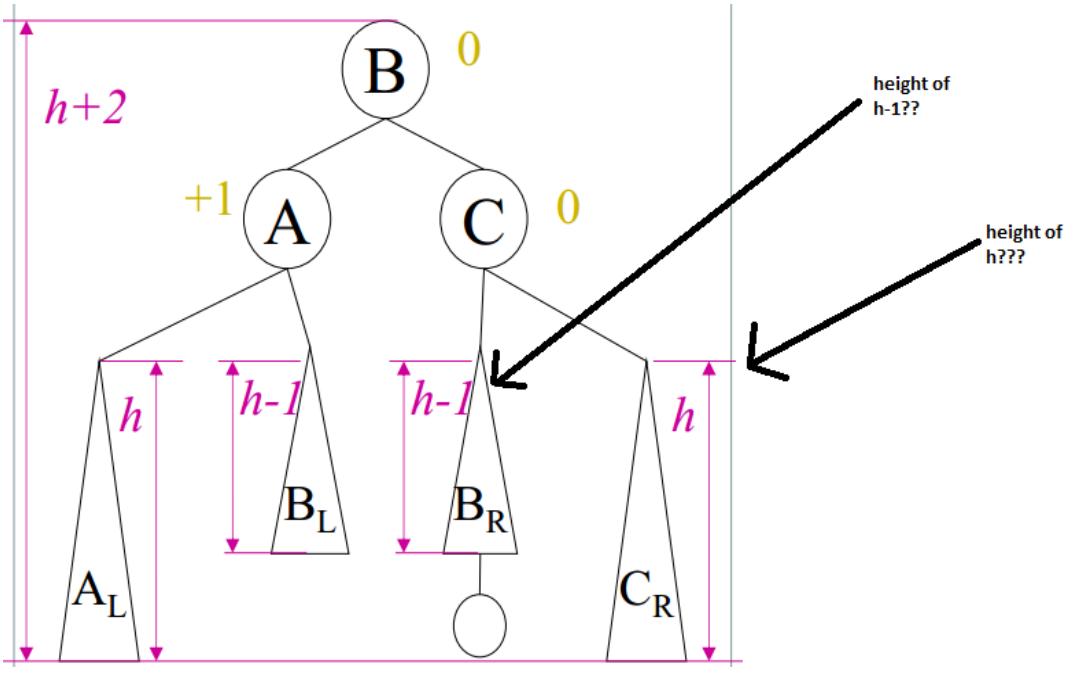
4.6 AVL Tree Balancing

4.6.1 Finding the Balance Factor

Before we even balance the tree, we need to find the balance factor to know which balancing method to use. There are four types of balancing: LL, LR, RL, and RR. The type of balancing done is decided by the balance factor. It takes in some node as input, and on first initialization that is the root node of the tree.

- It checks to see if the node passed is null. If it is, it returns 0.
- If it is not null, it runs the height calculation function on the left and right child of the node. The function returns the result of the left height subtract the right height.

Side note: Subtree notation



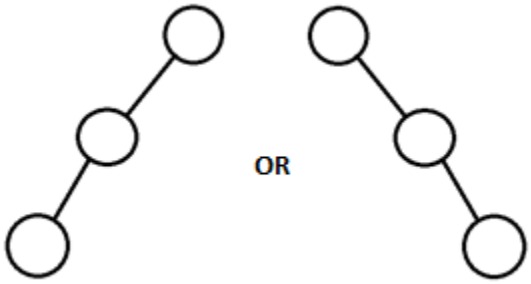
The triangles in the diagram are used to represent an entire subtree. Subtrees have varying heights which is why these triangles vary in size, and why some can have height **h** and some can have a height of **h-1**.

4.6.2 Balancing Function

When a new node is inserted, it may change the balance factor to something greater than 1 or something less than -1. If this is the case, we need to balance our tree. Now, since our nodes are inserted one at a time, the tree balancing algorithm will make sure that the balance factor is never more than 2 or less than -2 before balancing.

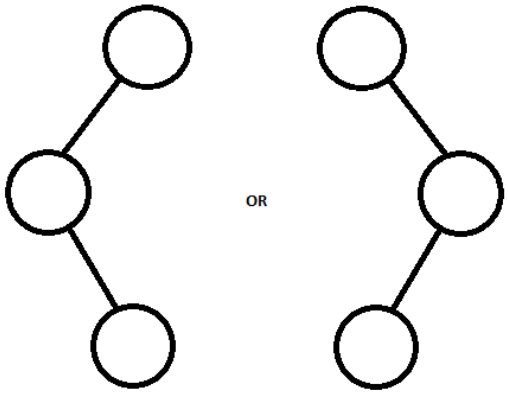
There are four types of balancing, but this can be further simplified to two types of balancing with different directions.

Simple balancing rotations are the left-left and right-right rotations. Only one round of balancing needs to happen when you do these types of rotations. You use this type of balancing when the path of the three nodes in the subtree looks something like this:



In CS, they call this type of path a **zig-zig pattern**, as in the path goes the same way both times.

Double balancing rotations are the left-right and right-left rotations. These are done when the path isn't straight and one rotation is not enough to balance the tree. You use double balancing when your three node path looks something like this:

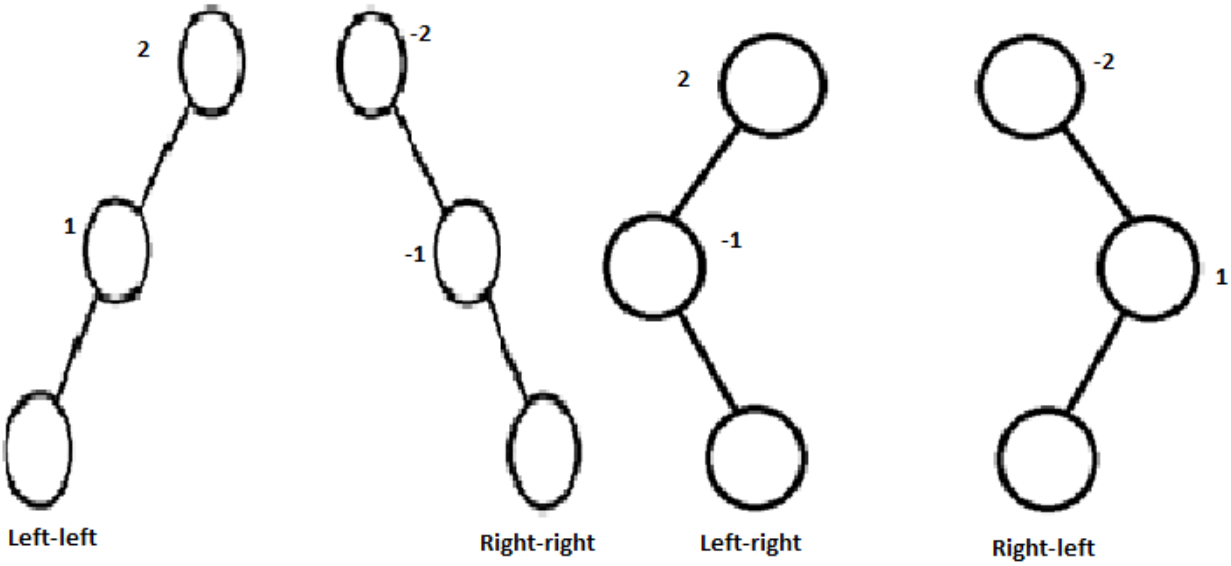


In CS, they call this type of path a **zig-zag pattern**, as the path goes one way and then the other.

In the code, what happens is the function will find the balance factor at the top node. If the balance factor is >1, then it knows the imbalance is in the left subtree, and it will again find the balance factor of

the left child of that top node. If the balance factor at that node is 1, then it will know to do a left-left rotation, because there is a node to the left. If the balance factor at that node is -1, then it will know to do a left-right rotation, as the balance factor indicates that there is a node at the right but not at the left.

If the balance factor at the root node is <-1, then it knows the imbalance is in the right subtree, and it will again find the balance factor of the right child node of the root node. If the balance factor of the right child node is -1, then it will know to do a right-right rotation because the imbalance is at the right path both times. If the balance factor at that node is 1, then it will to a right-left rotation.



If for whatever reason the balance factor at one of the subtrees is >2 or <-2, it will recursively run itself on either the left or right child depending on the balance factor of the node, so it balances the subtree.

Pseudo

The function takes in some node as input. It first takes the root node of the tree as input.

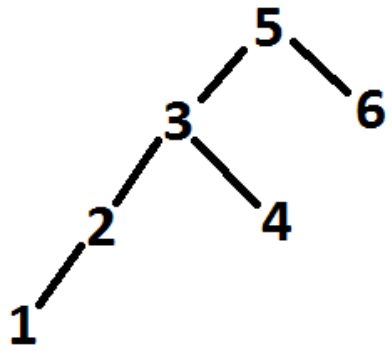
- It declares a temporary node to hold the current root node of the tree, and checks to see if the node passed is null. If it isn't, it continues.
- It then gets the balance factor at the node that was passed (declare a variable to hold this).
- It checks to see if the balance factor is greater than 1 or less than -1.
    - If it is greater than 1, it will find the balance factor at the left child (in a new variable).
        - If the left child balance factor is 1, it will run a left-left rotation.
        - If the left child balance factor is -1, it will run a left-right rotation.
        - If the left child balance factor is greater than 1 or less than -1, it will recursively run the AVL balance function at the left child.
    - If it is less than 1, it will find the balance factor at the right child (in a new variable).
        - If the right child balance factor is -1, it will run a right-right rotation.
        - If the right child balance factor is 1, it will run a right-left rotation.

        - If the left child balance factor is greater than 1 or less than -1, it will recursively run the AVL balance function at the right child.
    - If the balance factor at the node is not greater than 1 or less than -1, it will still recursively run the function on the left child and right child.
    - It then returns the temporary node

4.6.3 Left-left rotation

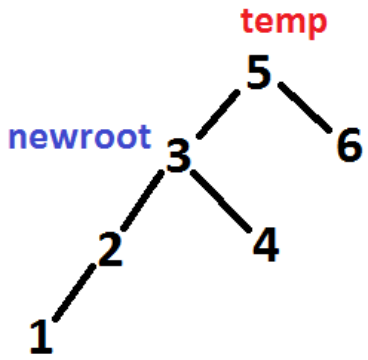
The left-left rotation is done when there is an imbalance in the left child and in the left subtree of the left child. The balance factor would be 2 at the root and 1 at the left subtree.

Consider this tree here:

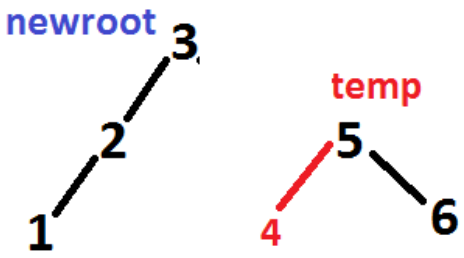


The balance factor of the root is 2, and the balance factor of the left child is 1, so we do a left-left rotation.

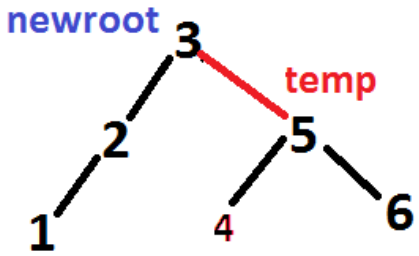
- We first do a check to make sure the old root is not null.
- Once we know the root is not null, we can run our steps. First we are going to give names to our key nodes that we are going to use for the rotation. Let's call the original root **temp**, the new root **newroot**.
- We set the old root equal to the temp variable.
- We want to make the **new root the left child of the old root**.



- We then want to make the **left child pointer of temp point to the right child of the new root.**



- Finally, we want the **right child pointer of the new root to point to the temp.**

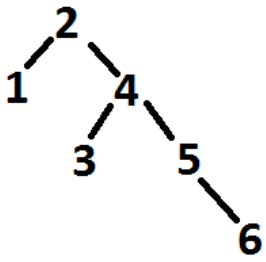


And there it is, your tree is balanced. Finally, we return the new root. If the operation fails we return null.

4.6.4 Right-right rotation

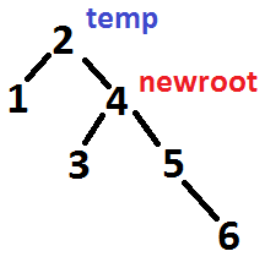
RR rotation is very similar to LL rotation. It is basically a mirror of the LL rotation. The right-right rotation is done when there is an imbalance in the right child and in the right subtree of the right child. The balance factor would be -2 at the root and -1 at the right subtree.

We'll be using this tree here as a guide:

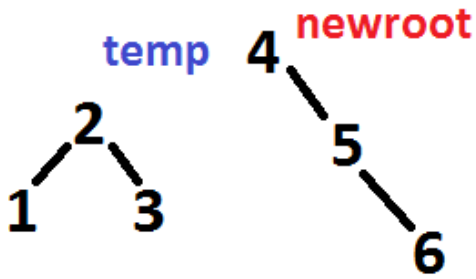


The beginning of the function is the same as that of the left left rotation function. The balancing steps are as follows:

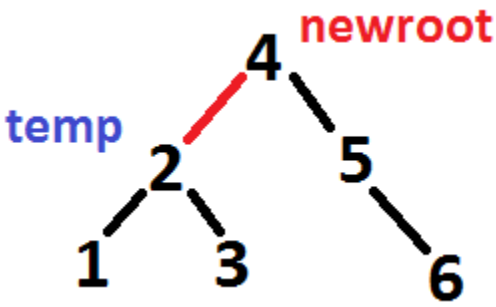
We first want to make the **new root the left child of the old root**.



We then want to make the **right child pointer of temp point to the left child of the new root.**



Finally, we want the **right child pointer of the new root to point to the temp.**



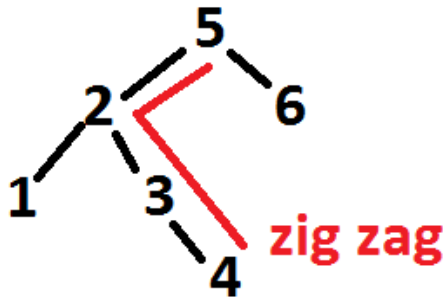
The rest of the steps are the same as left left rotation.

4.6.5 LR Rotation

The left-right rotation is done when there is an imbalance in the left child and in the right subtree of the left child. The balance factor would be 2 at the root and -1 at the left subtree.

This is where the naming comes from. Left right means that there is an imbalance on the left of the root and on the right of the left subtree. It is NOT the order of calling of the functions.

We will use this tree here as an example.

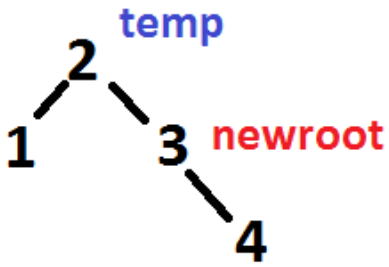


Notice how the pattern is zig zag.

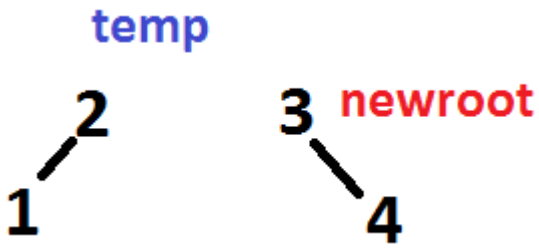
This function is called like the other rotation functions and accepts the old root node of the tree or subtree as input.

Next, we run the **right rotation** on the left subtree of the root. We return the new root for the temp variable.

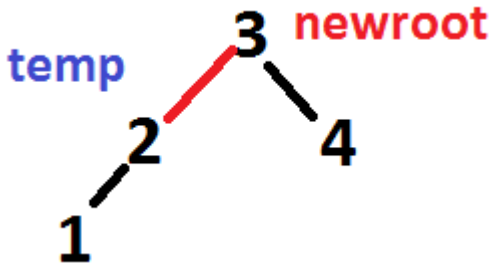
Within the RR rotation function:



Step 1:

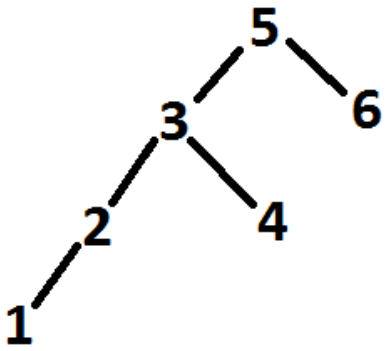


Step 2:



Step 3:

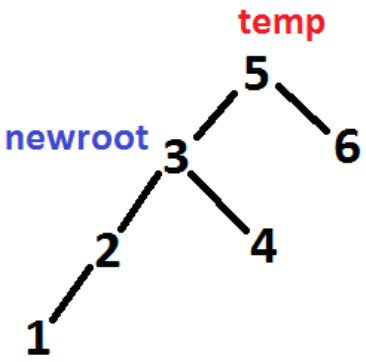
REMEMBER THAT TEMP HERE IS NOT THE TEMP IN THE RL FUNCTION!!! IT IS THE ONE INSIDE THE RR!!



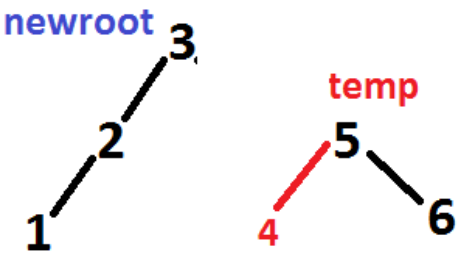
We end up with this:

We then do a LL rotation at the root level. We send the old root as the input variable.

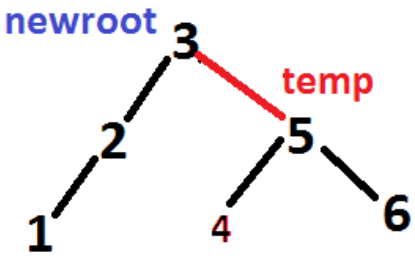
Within the function:



Step 1:



Step 2:



Step 3:

We then return our temp as the new root.

4.6.6 RL Rotation

The right-left rotation is done when there is an imbalance in the right child and in the left subtree of the right child. The balance factor would be -2 at the root and 1 at the right subtree.

RL pretty much a bass akwards version of LR. You run recursively the LL rotation on the right subtree, and then you run the RR rotation at the root level. Code is almost exactly the same except for those changes.

5. Heaps

Heaps are another type of tree that are used to sort, find, and withdraw more efficiently. It sorts the items in the tree by their values in ascending or descending order of the tree. A heap is actually more efficient than an AVL search tree because it doesn't require 'balancing', it just requires easy reordering. It can also be implemented as an array.

Key Terms

Min heap: The values of the children of any parent node are always larger than the parent

Max heap: The values of the children of any parent node are always smaller than the parent

Null path: In any tree, the null path of a node is shortest path from the node to an external node in the tree

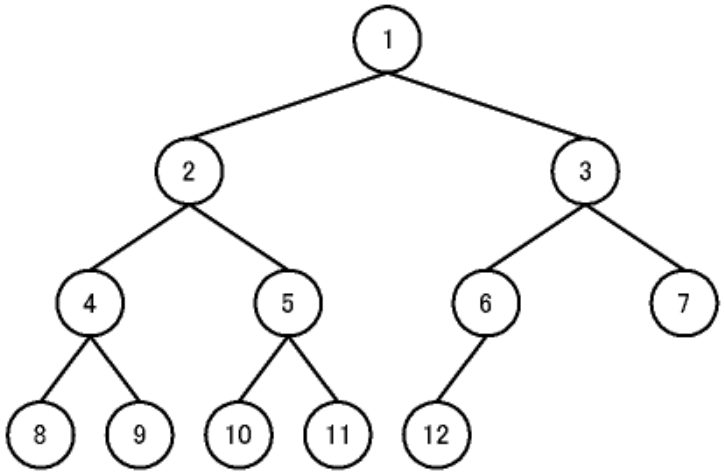
Null path length: The null path length is the length of the null path

5.1 Complete Binary Heap

There are two main types of heap trees. The first is a binary heap and the second is a leftist heap. The prof likes min heaps so I'll use that from now on.

5.1.1 Complete Binary Heap Definition

A binary heap is a binary tree that satisfies the heap condition. Here's an example of a binary heap:



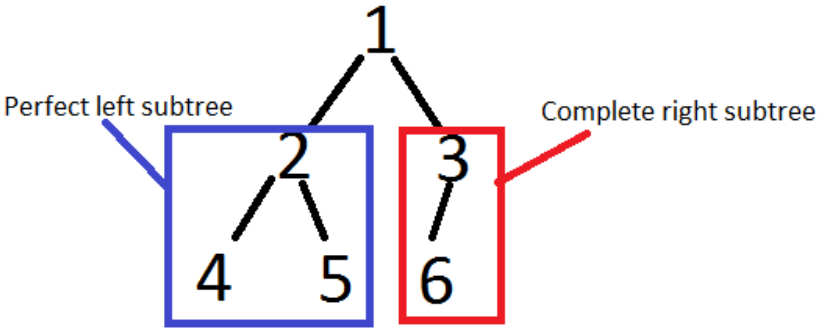
In this example, they actually put all the values in order. It's important to note that in a binary heap, the left child of the parent node doesn't have to be less than the right child (this is **not a binary search tree**).

What makes a binary tree a **complete binary tree** (which, as a heap, would make the tree a complete binary heap) is that it is always filled out **left to right**, and each "level" of the tree needs to be filled before moving onto the next one (breadth order). What makes a tree a **perfect binary tree** is that all of the leaf nodes of the subtree are all of the same depth.

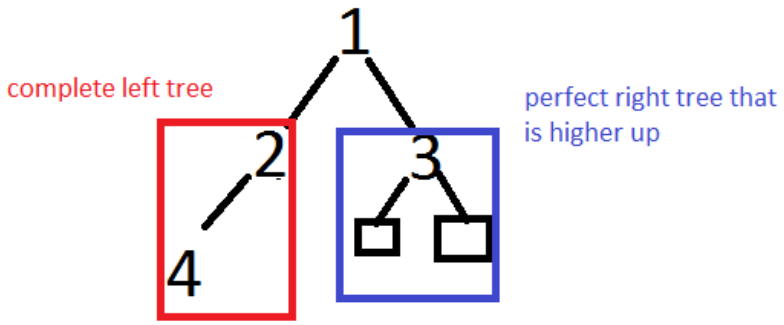
If you want to see fancy schmancy math terms then here they are:

1. If h = 0, the T_L and T_R = NULL
   This basically says that if the height is 0, the tree has either one node or zero nodes

2. For h > 0, there are two possibilities
    a. The left subtree T_L is a perfect binary tree of height of *h-1* and and right subtree T_R is a complete binary tree of height *h-1*. This basically means that the entire row could be filled, or the right subtree is almost filled.



    b. The left subtree T_L is a complete binary tree of height *h-1* and the right subtree T_R is a perfect binary tree of height *h-2*. This means that the left subtree can be either completely filled out or partially filled, and the right subtree has not been filled at all.



Binary Heap as an Array

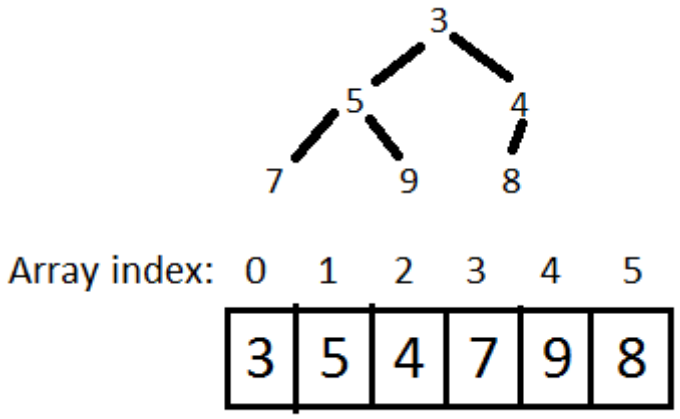
With a binary heap, it is easy to map parents to their children in an array. The relationship between the indexes of parent nodes and child nodes are given by the following formulae, with n as the index in the array:

Parent to child: $Left\ child = 2n + 1, Right\ child = 2n + 2$

Child to parent: $Parent\ node = \frac{n-1}{2}\ rounded\ down$

This is assuming the array starts at index 0.

Here's an example of how to fill out an array with a tree:



Use this to check the formulas that were given before. For example, if we want to find the parent of 8, we would check the index, which is 5, and then use the formula 5-1/2 to get that the parent of the node is at index 2, which holds the value 4. Therefore, the parent of 8 is 4.

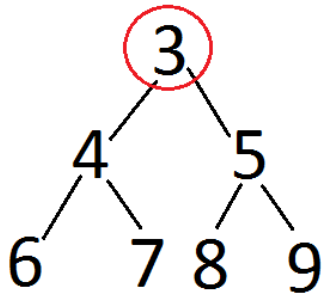
5.1.2 Binary Heap Operations

The two main operations for a binary heap are insert and withdraw. Both are pretty easy operations. I won't supply code because I don't think he'll ask for that on the quiz.

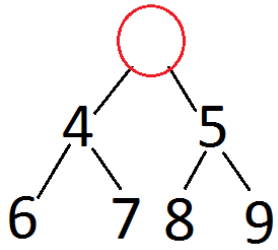
Withdraw

To withdraw from a binary heap:

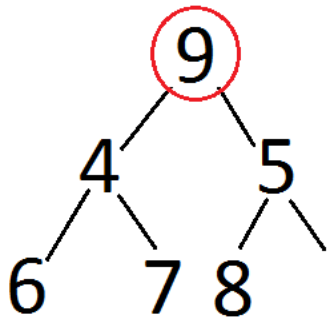
1. First you pick the value you want to withdraw and make sure it is in the tree. In the example we'll withdraw 3, which is the root.



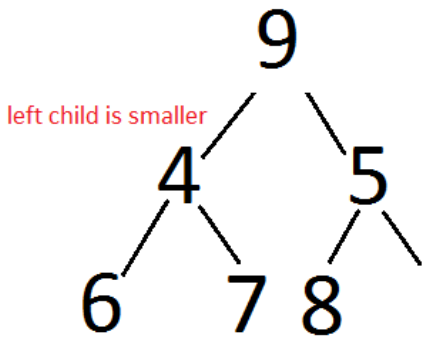
2. Next, we take out the value.



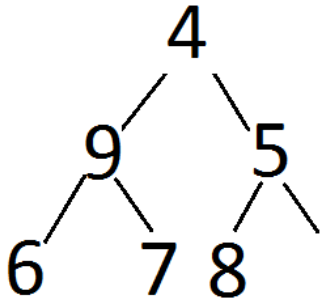
3. Then, we put the **right most value on the bottom row in place of the node we removed.** This makes sure we automatically maintain the **complete binary heap structure**.



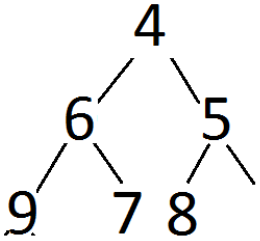
4. We then have to do our re-arranging. Steps 4 and 5 are done in a separate rearranging function, and it is always necessary to run this. We compare the value of the left child and the right child of the new node to see which one is smaller.



5. Now that we have the smaller child, we compare this value to the value of the parent. If it is smaller, then we function that swaps these two nodes, and then will re-run steps 4-5 again recursively on the same node. If the node is not smaller, then no swap is necessary, and the function will not do anything (return NULL). This will be our **terminating condition**.



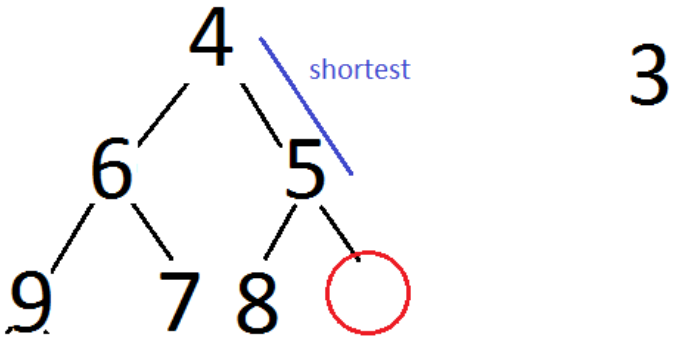
After running these steps recursively, the end result will look like this:



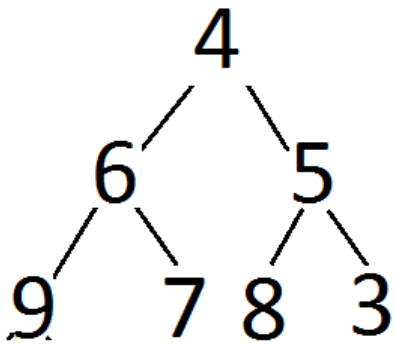
Insert

To insert a value into a binary heap:

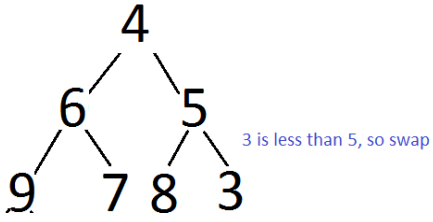
1. Find the perfect location to insert into the binary heap. You would do this by looking for the right most location on the last row. This would be done by picking the shortest path to an internal node (if there's a tie then pick the left most one).

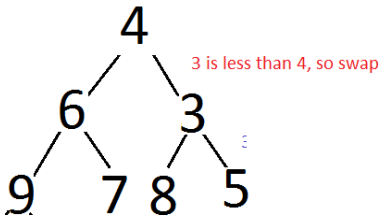


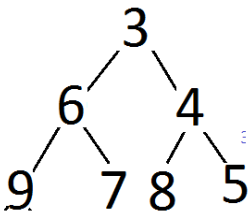
2. You then insert your new value into that location.



3. We then compare the new value to the parent of the new value. If **the new value is less than that of the parent value**, we swap the two values. We would then run this step again recursively. If the new **value is greater than that of the parent value**, then the heap conditions are satisfied and we're good to go (terminating condition, nothing happens).





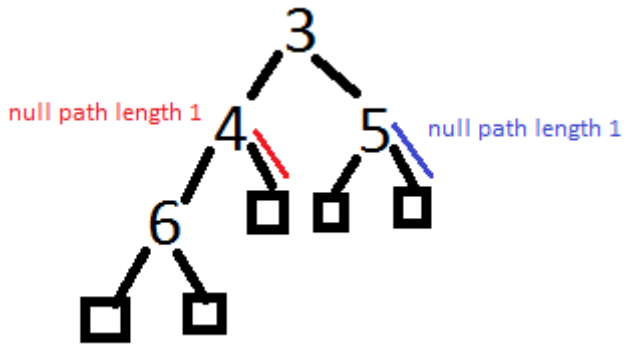


In the best case, both of these operations take O(1) runtime, and in the worst case, it takes O(log n).

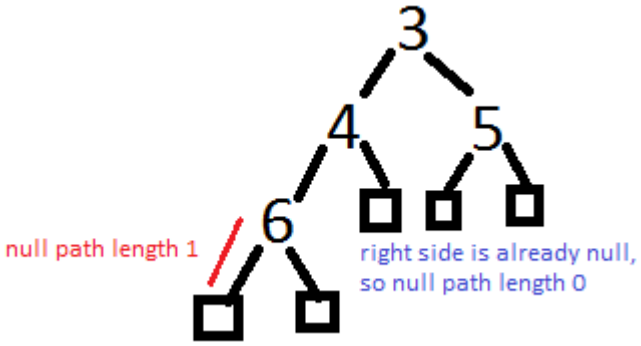
5.1 Leftist Heap

A leftist heap is a heap with the form of a leftist tree.

A leftist tree is basically a tree where the **left subtree of a node always has a null path length greater or equal to that of the right subtree**. Take this example:



To check, we look at the null path length of the left subtree and the null path length of the right. On the left, the null path length is 1, and on the right, it is 1. At the root level, the condition is satisfied. Now we check again at root 4, because it has a subtree attached to that as well.



At the node 4, the condition for a leftist tree is still satisfied, so this is in fact a leftist tree.

Calculating the Null Path Length

This function takes in some node as the input. It is literally the exact same as the calculating height function (4.5.7), except instead of returning the max of the two heights, it returns the min, or the lower value of the two.

6. Hashing

The past few concepts have been data structures where items are put in pretty random locations, and without actually looking at the entire data structure, finding items in the data structure is pretty tedious. Wouldn't it be nice if you were just able to pin point your item straight away (or almost straight away) (or in technical terms O(1))? This is where hashing comes in.

A hash function is a function that takes the value of an item and then spits out an array index to store the item. For consistency, let's call this item a **key**. The array that holds all of these keys is called a **hash table**. As long as the array is big enough, functions like insert, find, and remove are really quick. Why?

1. For **insert**, all you have to do is run the hash function, get the index, and put the key in that index.
2. For **find**, all you have to do is run the hash function, get the index, and find out if it's there or not
3. For **remove**, you run the hash function, get the index, if it's there then take out the key

But then why doesn't everyone just use a hash function? There are some issues.

1. First off, because the size of an array is finite, only so much data can be stored in a single hash table.
2. Second, each spot in the array, or **bucket**, can only hold one key. This means that if a hash function spits out the same array index for two different items, we have to work around that. When there is a conflict for an array spot, we get what is called a **collision**.
3. Third, since array indexes are all integers, if you want to write a hash function for something like a string you have to figure out a way to smartly change your string into an integer.

Some hash functions are better than others. Good hash functions don't cause lots of collisions, use the space efficiently, and spread out the values so you don't get what's called **clustering**.

Hash Function Method: Division Method

If the values of the keys are all integers (or some of them at least), then an appropriate hash function to use would be the **division** hash function. The function looks like this:

$$h(x) = x \; Mod \; n$$

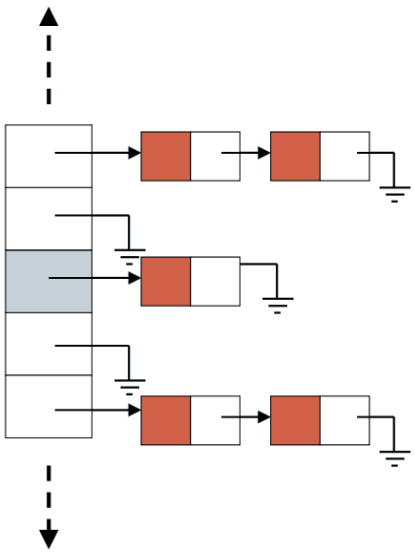
Where h(x) is the array index, x is the numeric value of the key, and n is some number that is pre-defined. The value for n is determined strategically by the programmer. Pick an *n* that will minimize the number of collisions.

6.1 Getting Around Collisions (Without Open Addressing)

Collisions suck but there are ways to get around it. This is completely situational.

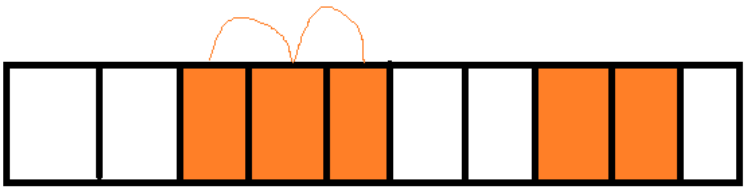
6.1.1 Separate Chaining

For this method, when an index in the array is filled and the hash function tells you to put it there, you simply prepend it onto the array index to create a chain on that index. This **eliminates all collisions**, and is therefore the safest method, but means that you can't necessarily find the array index immediately. It also makes removal more difficult. The hash table looks like this:



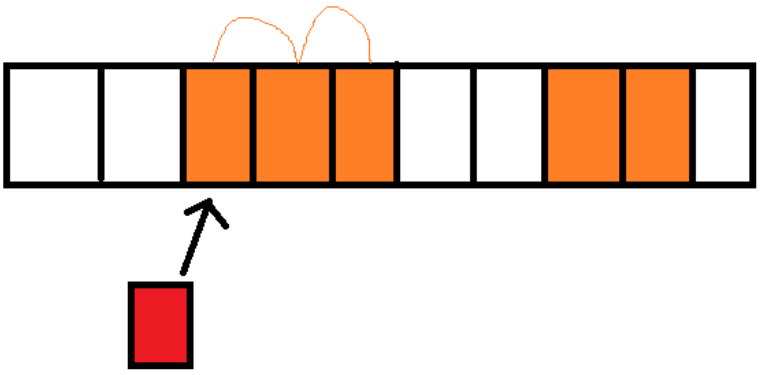
6.1.2 Regular Chained Scatter Table

With a regular chained scatter table, when a collision occurs, the key will be inserted at some other index in the array. Each index in the array will contain a value node and a pointer node much like a linked list. This new location will be pointed to by the pointer from the old array location. Subsequent keys in an array are called **a chain**.

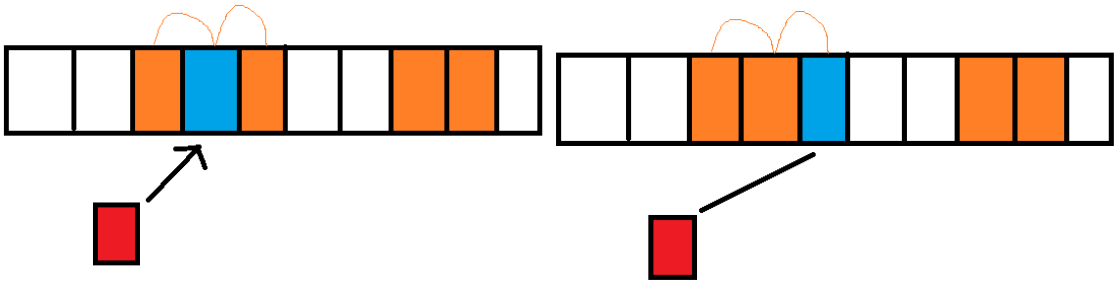


linked together in a chain

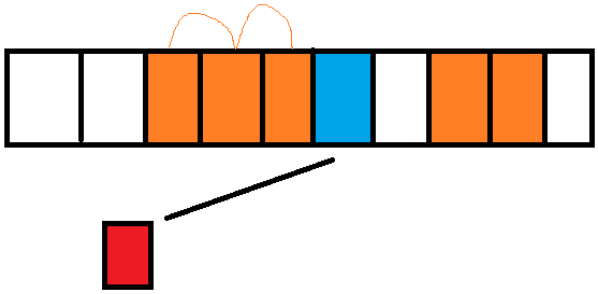
How do we go about finding this new location in the scatter table? Consider this example here.



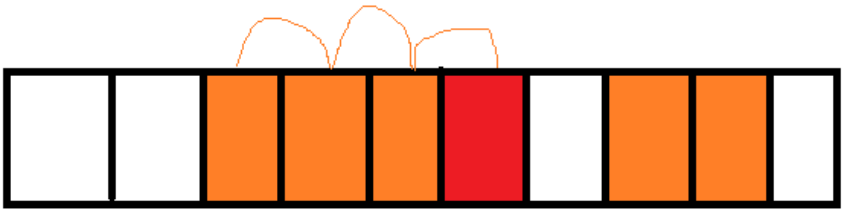
The hash function points to this pre-occupied space, and the red key wants to enter the hash table. What we do first is follow the pointers until we get to the end of the chain.



Once we have the end of the chain, we then do a linear search through the array to find an empty array index.



Once we have an empty area, we place the key into that location, and make the end of the chain point to that location.



For finding, you only have to traverse the chain to see if it's there.

6.2 Getting around Collisions (with open addressing)

Open addressing is a method that does not use pointers but rather some algorithm that will bring you a different array index. It's by far the least expensive but causes more collisions than the linked list ones.

6.2.1 Linear Probing

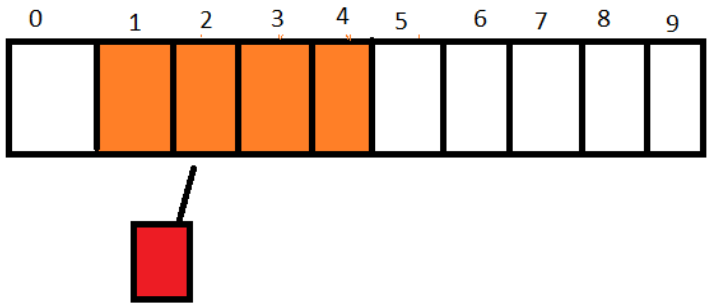
Linear probing involves running your hash function again if there is a collision and adding one to the result. This is basically a linear search for an open array index location. The function looks like this:

$$h(x,i) = (h(x) + i)\ mod\ M$$

This is where h(x,i) is the array index location, *i* is the number of times you've run the probing (minus one), h(x) is the original hash function, and *M* is the size of the array. The mod M is important because it makes sure that you don't go over the size of the array in your search.

Now, suppose we have this scenario here, where the hash function spits out a value of 2 for some key that you want to insert.

$$h(x, 0) = (2 + 0)\ mod\ 10 = 2$$

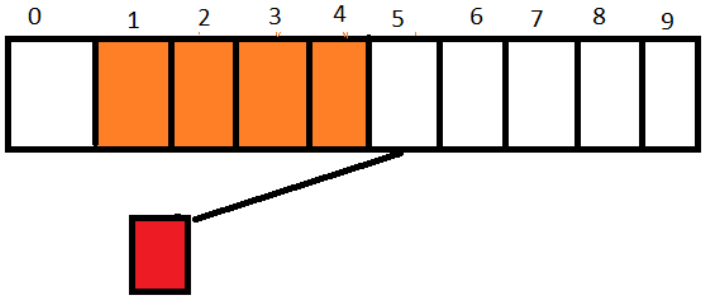


Well, there's a collision. What you do is run the hash function again and add one. You'll get three.

$$h(x, 1) = (2 + 1)\ mod\ 10 = 3$$

That one's filled too. Well writing out all the steps is tedious so I'm just gonna go ahead and skip to *i=3*.

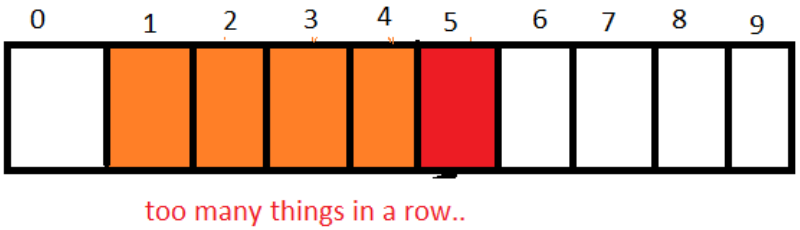
$$h(x, 3) = (2 + 3)\ mod\ 10 = 5$$



No way, and empty index! Fill it in and you're good to go.

6.2.2 Quadratic Probing

Linear probing actually sucks because it takes really long if there's a chain, and if you have values that will give similar array indexes you'll get what's called a **cluster**. A cluster is when there are too many filled array indexes in succession. Hash functions are supposed to spread out keys in the array.



To fix this problem (well sort of), we use quadratic probing. Quadratic probing is very similar to linear probing except we change up the formula a little, so that we square the *i* value.

$$h(x, i) = (h(x) + i^2)\ mod\ M$$

So now, collisions won't be dealt with by putting the item next to the array index, it'll be put at some value farther away. I'm not going to visually show the example to this, you can figure it out. It's very similar to linear except you square your *i*.

What is wrong with quadratic probing? The problem is that it is not hard to repeat array indexes with the quadratic probing. Eventually, when the array gets more than half full, there is a good chance that the probing will go into an infinite loop and you will never find that coveted empty array indexThis is even worse if the value for M is not a prime number.

6.2.3 Double Hashing

In many cases, double hashing works the best. Double hashing takes one hash function, runs it, and if there's a collision, it will run another hash function multiplied by *i*.

$$h(x, i) = (h1(x) + (i * h2(x)) \; mod \; M$$

7. Sorting Algorithms

Sorting is very useful. When an array is sorted, it makes finding an item in the array much easier and much faster (forget hashing for now). There are a bunch of sorting methods and some are better than others. Choosing the right one for your situation is crucial to writing the most efficient program.

We will assume that a sorted list always has values in increasing order.

There are five main sorting algorithms, some that have variations. We will be going through all of them. These are:

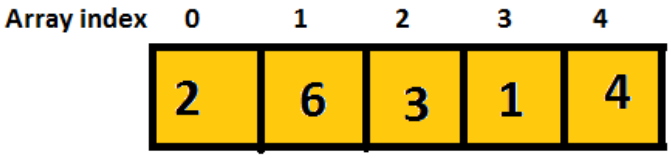
- Insertion sort
- Exchange sort
- Selection sort
- Merge sort
- Distribution sort

7.1 Insertion Sort

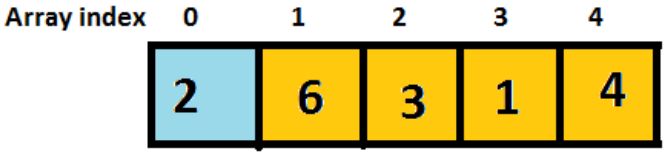
7.1.1 Straight Insertion Sort

Straight insertion sort involves re-inserting the elements from left to right. At each index in the array, we compare the index's value to the value to the left. If the value to the left is greater, then the values swap, and then the process repeats until the value to the left is not greater (the sorting loop).

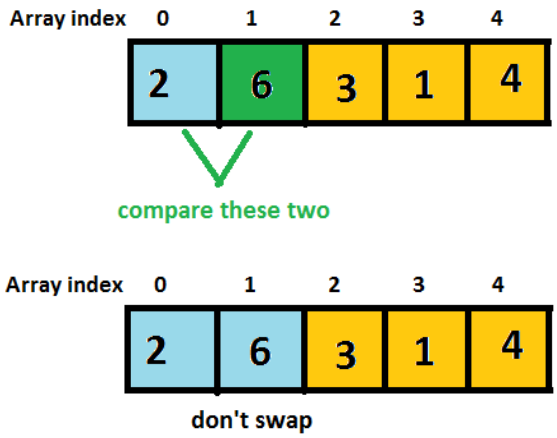
Let's see this in steps. This diagram will use light blue for the already sorted, green for the value we're "inserting", and orange for the non-sorted section. At first, the entire array is non-sorted.



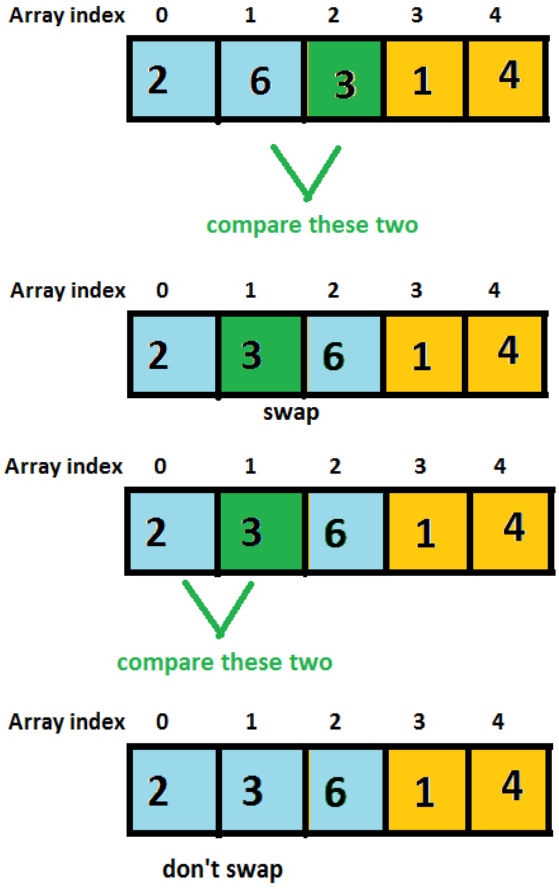
1. We "insert" the first array element. There is no value to the left to compare to, so there's no need to run the loop here. Moving right along.



2. We then "insert" array index 1. After inserting array index 1, we compare it with the value of index 0, the one to the left. If it is greater, we swap, if it isn't, we don't, and the loop breaks.



We then repeat step 2 on the next array index. We keep doing this until we get to the right of the array.

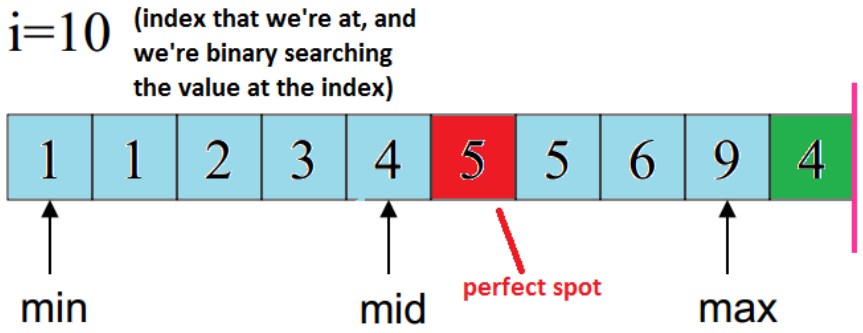


By the end of this, the array will be sorted.

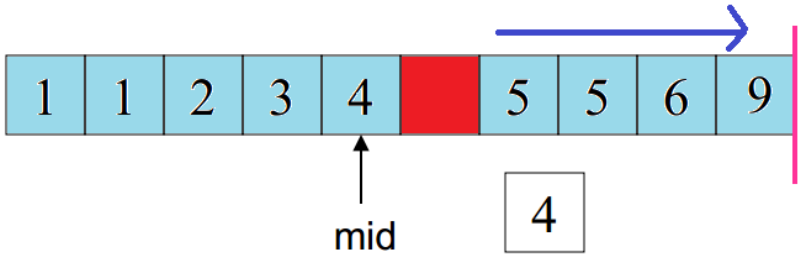
7.1.2 Binary Insertion Sort

Binary insertion sort is another insertion algorithm. The difference between this algorithm and the previous one is that for this one, we look for the perfect place to "insert" the element. Here are the steps:

1. Run a for loop to go through every index in the array. The next steps will be run on the specific index in the array.
2. Use binary search (http://www.youtube.com/watch?v=wNVCJj642n4) to find the perfect place to insert the array element. The (slightly modified) binary search function returns some array index as the perfect location.
    a. If the binary search finds the value, it will return the existing value's array index plus one.
    b. If the binary search does not find the value, the binary search's 'middle' will hold some value that would be to the left or right of the 'inserted' value. If the value in the 'middle' index is greater than that of the 'inserted' value, the function will return the middle. If it is less, then the function returns the middle plus 1.



3. Shift all values in the array, from the 'perfect location', up until the end of the partial list, up one. This is done in a for loop, starting at the top element (i-1), and ending at the perfect spot.



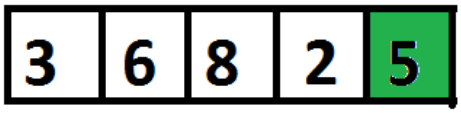
4. Place the value we want to insert into the perfect location.

7.2 Exchange Sort

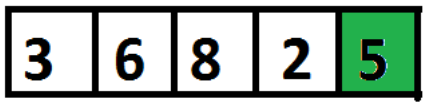
7.2.1 Bubble Sort

Bubble sort is where, on each iteration of the function, the lowest value on each iteration bubbles to the top. Here are the steps:

1. We first want to look for a value for the first array index (we'll call it i=0 for the first iteration). We start at the end of the array (the process of looking for this value is done in a loop that is in descending order).

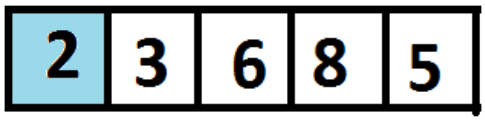


2. Compare the current array index to the one to the left (index – 1).
    a. If the value to the left is greater, swap the current and left array index values.
    b. If the value to the left is less, don't do anything.



3. Change the current array index to the one to the left, and then repeat step 2. Keep going until we reach the first array index.

After running steps 2 and 3 repeatedly until we reach the first array index…



4. Now, check to see if there was any swapping done in the pass. If there wasn't any swapping done, that means the **list is already sorted**, and you can just exit sort. If not, then continue on.
5. Repeat steps 1-4, starting once again at the end of the array. This time, the 'first array index' will be the previous 'first array index' plus one.

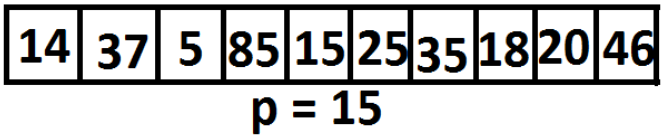
7.2.2 Quicksort

Quicksort is a little more complicated than bubble sort but is very fast for many applications. There are two parts to quicksort: the partitioning and the quicksorting.

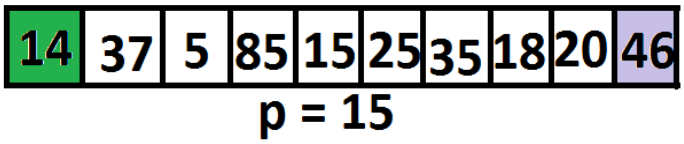
How to Partition an Array

Partitioning an array means having assigning an array pivot, having all elements to the left of the pivot be less than the pivot, and having all elements to the right of the pivot be greater than the pivot. Here are the steps to doing this:

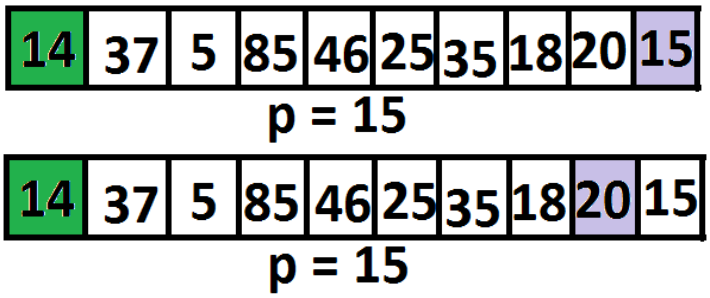
1. Pick a pivot. This is done arbitrarily, and there are different ways to pick one based on the set of data, but with our method we're just going to take the middle for simplicity.



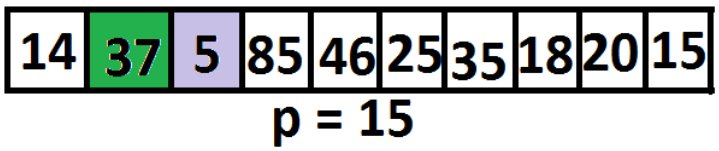
2. Now, assign a **left and right selector**. The left selector should start at the left most side of the array, and the right selector should start at the right. In the diagram, the left selector is green, and the right selector is purple.



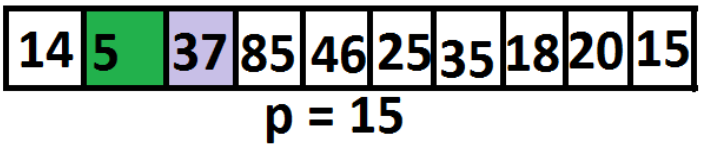
3. Swap the locations of the pivot and whatever is in the right selector. Move the right selector over one.



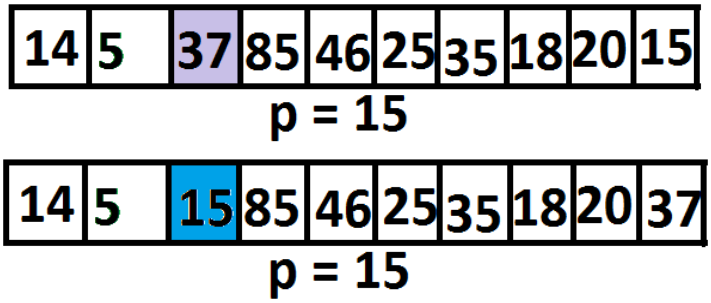
4. Check the value at each selector. Move the left selector to the right **until you reach a value that is greater than or equal to the pivot**. Move the right selector to the left **until you reach a value that is less than or equal to the pivot**.



5. Swap the values of the left selector and the right selector, if at the same time, the left selector has a value greater than that of the pivot, and the right selector has a value less than that of the pivot.



6. Eventually, the selectors will be at the same index in the array. At this point, swap the locations of the current selector's value and the pivot, but **only if the value of the pivot less than that of the selector** (if the value of the pivot is greater, that means that the pivot is already in the correct spot being at the very end of the array).



Now, all the values to the left of the pivot are less than the pivot, and all the values to the right of the pivot are greater than the pivot.

The Entire QuickSort method

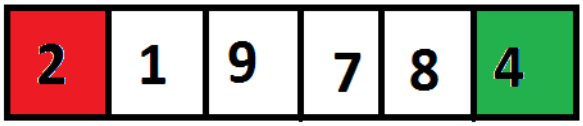
1. If the array has 1 element or less, there is no need to run the quicksort method, and it skips everything.
2. It then partitions the array. At the end of the partitioning, we will be left with a pivot array element that is in the right location.
3. We treat the group of elements to the left and right of the pivot as smaller arrays. We recursively call the quicksort method on the array to the left of the pivot, and the on the array to the right of the pivot.
    a. Note that with a sub array that is small, the quicksort might not be the fastest, so we would call some other sorting method to sort that array. This would be checked in some if condition.

7.3 Selection Sort

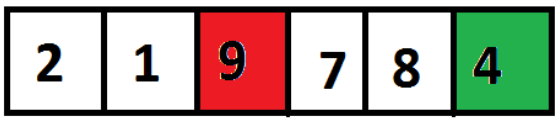
7.3.1 Straight Selection Sort

Straight selection sort is easy. It is similar to insertion sort where you re-insert all of the values, but this time you pick which values you want to insert. We insert from right to left based on the max value to the left of the index. Here are the steps:

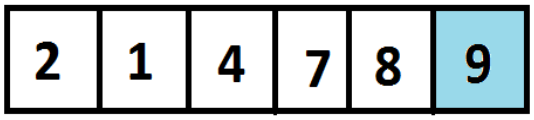
1. Start the insertion at the right side of the array, with the initial max being on the left side of the array. We're going to use green as the array index we're inserting into (insertion index), red for the current maximum value in the array, and light blue for the already sorted section.



2. It will then traverse the entire array to the left of the insertion index, to find the actual max.



3. Once we find the max, we swap the value of the max and the value inside the insertion index **only if the max is greater than the value already in the insertion index**.



Repeat steps 1-3, but with the insertion index being to the left of the previous one.

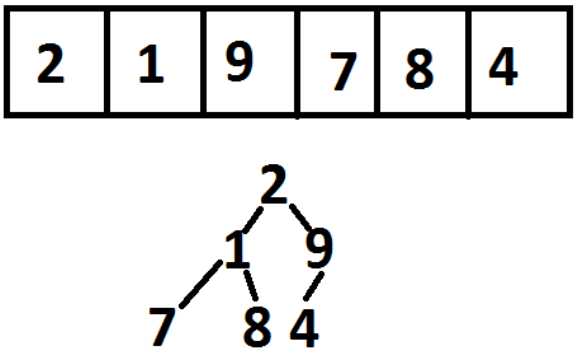
7.3.2 Heap Sort

Remember the heap? Yeah well I hope you haven't forgotten it because we actually implement a heap on an unsorted array to help sort our array. We are assuming an array index of 1, the root will be at array index 1, the left child will be at (parent index * 2) and the right child will be at (parent index * 2 + 1).

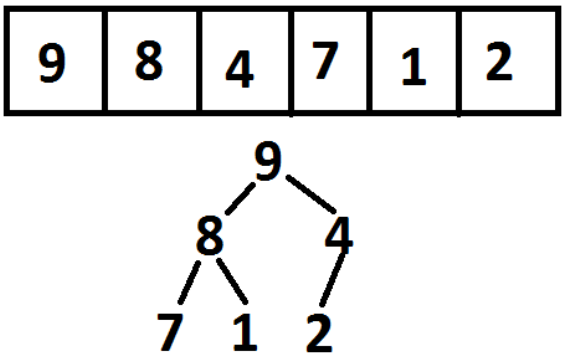
Part 1: Making our unsorted array into a heap

We want our array to be sorted into a **max heap** (refresher: this means that parents are always greater than their children).

1. At first, our tree looks something like this:



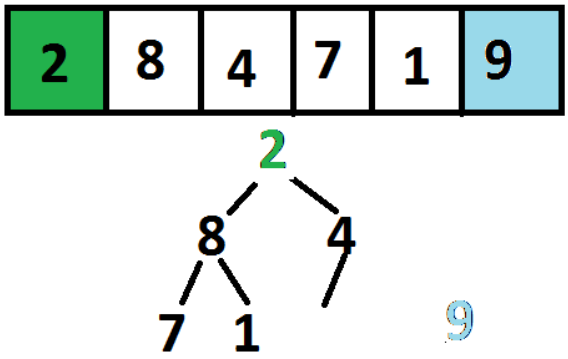
2. Next, we percolate. Percolating is when you take the parent node, compare it to the children nodes, and swap with the larger child (if and only if the parent node is less than the child). You then repeat the process on the parent's new location.
   It is best to start percolating at index (n/2) where n is the size of the array (round down if there is a decimal result). You then work your way back to the root (i.e. percolate at the previous index until you get to the root), and eventually the entire tree will satisfy the heap condition.



Part 2: Sorting

Now, we begin to remove items from the tree and place them in the array from left to right.

1. Swap the root node with the last node that is still in the tree. Remove the old root node from the tree (this is done by cutting down the number of elements in the tree with a loop).



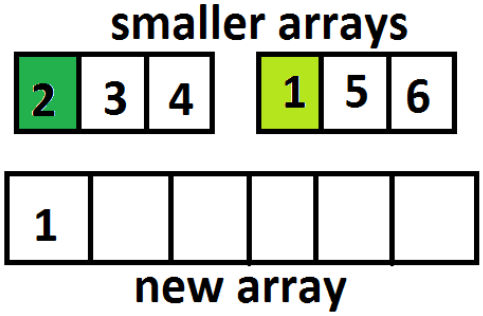
2. Percolate the new root node to its correct position.
3. Repeat steps 1-2. The elements 'removed' from the tree are **no longer part of the tree**.

7.4 Merge Sort

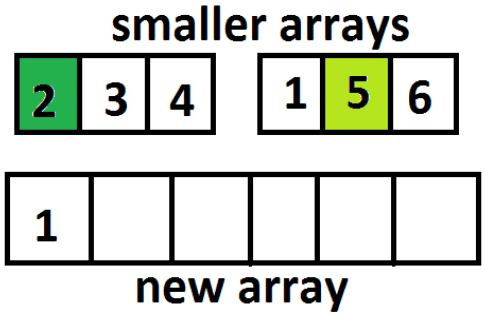
7.4.1 How to Merge Two Arrays

When you merge two arrays, basically you take two already sorted arrays and insert them into a new array. Here's how you do it:

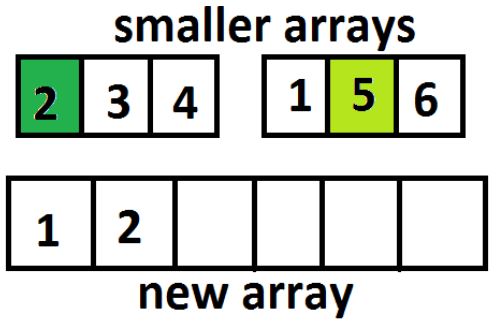
1. Start at the left most element in each array. Compare the values. Insert the smaller value into the new array, at the left most unfilled spot.



2. Move the index one over, for the array that just inserted into the new array.



3. Compare values again, insert again, and then repeat step 2. Keep going until the new array is filled.

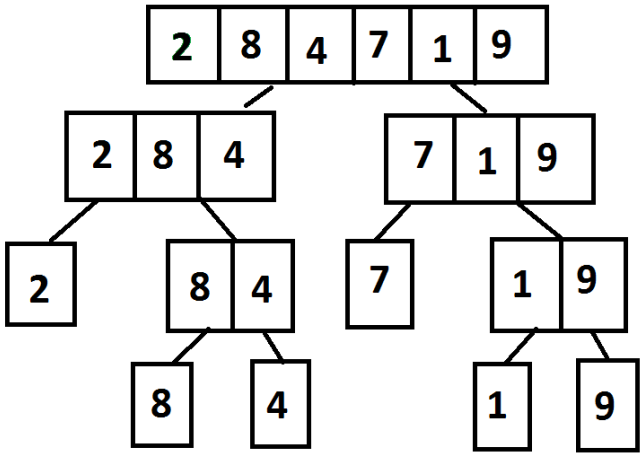


And so on…

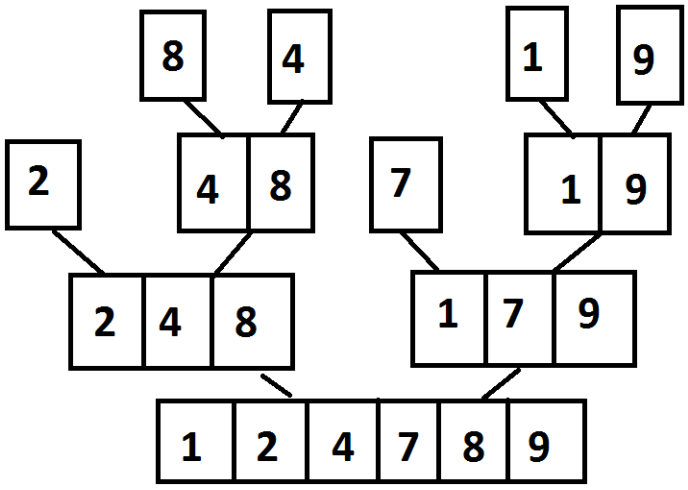
7.4.2 How to Merge Sort

Merge sort is when you split your array into small arrays of size 1, and then merge all of those arrays together. The steps look like this.

1. Recursively split up your array into smaller arrays of size 1. Do this by splitting the array in half recursively until you can't split up the array anymore**. Remember the path taken when doing this (especially if you're doing this by hand)**. Note that when you split an odd numbered array in half, you always round down, so the array to the right always has one more element.



2. Now, merge arrays that came from the same 'parent array', using the merging steps that were described earlier.



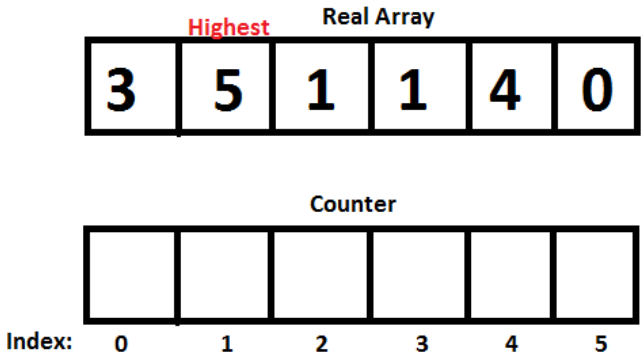
7.5 Distribution Sort

7.5.1 Bucket Sort

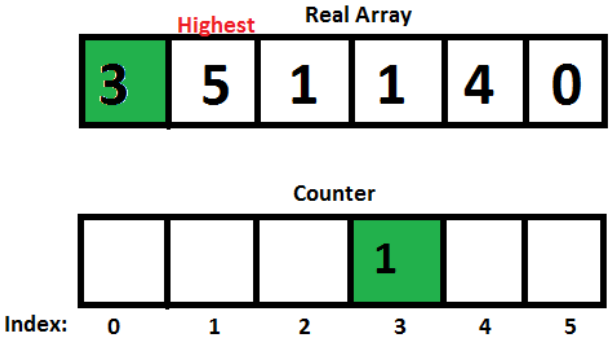
Bucket sort is great for when you know, exactly, the bounds of the values in your array. It should be done when the bounds are small, otherwise the operation takes up a lot of space. The idea is that we count the number of times each value appears in the array, and then insert based on that.

Here are the steps:

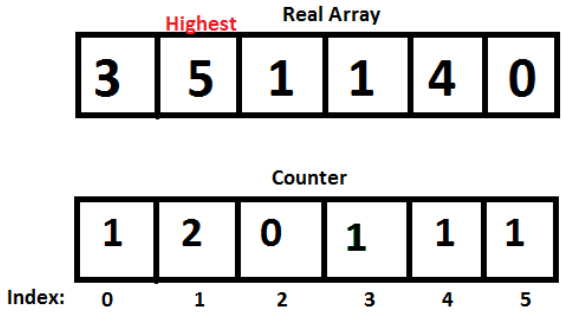
1. Create a **counter** array to count the number of times the index value appears. The end of the counter array should be the highest value in the array.



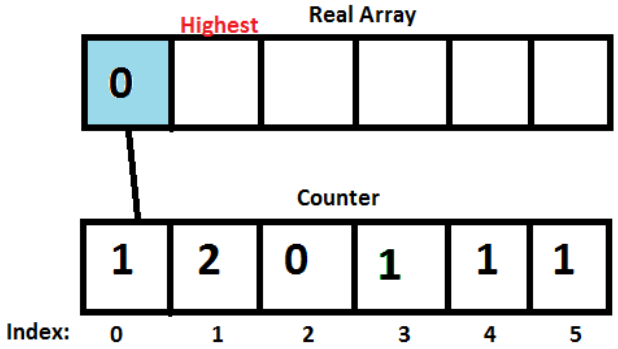
2. Traverse the array. At each element of the real array, add one the index of the counter array equal to the value. In our example, the first element has a value of 3, **so we add one to the counter of index 3.**



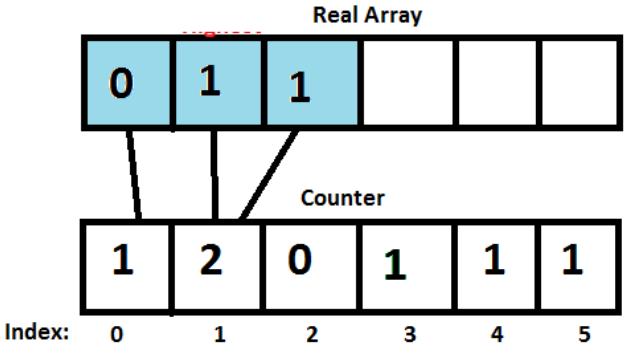
Several steps later…



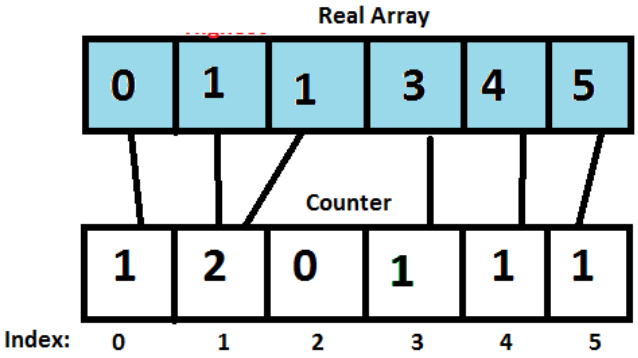
3. Now, re-fill the real array. This is done by inserting the index the number of times it occurred in the original array. In our example, in the counter array, the index of 0 has a value of 1, meaning it occurred once. We insert that in first.



The index of 1 has a value of 2, meaning it occurred twice. We insert 1 into our real array twice.



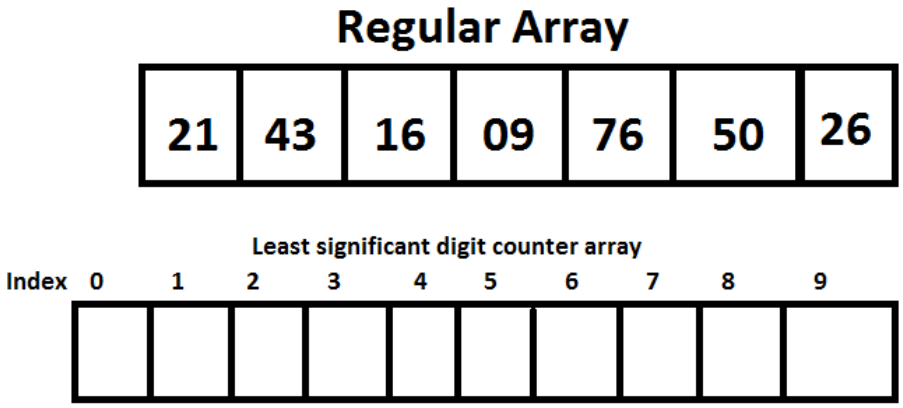
Keep going until we get….



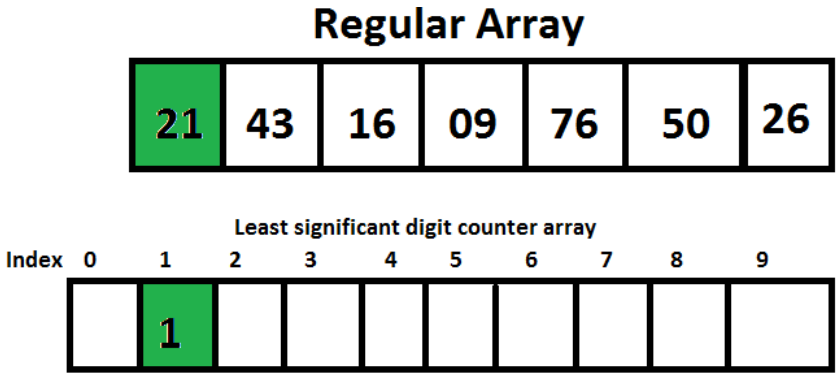
7.5.2 Radix Sort

Radix sort has a similar idea to the bucket sort in the sense that it uses a counter for the least significant digit. Then, it does an offset and it gets pretty complicated but I'm gonna do my best to explain it. This method is better because you only need to know your bounds in base 10. It does, however, take up more space. Here are the steps:

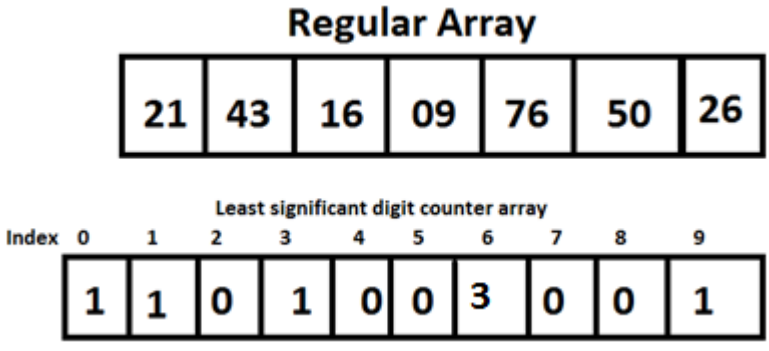
1. Create a counter index of 10 elements (assuming we start at index 0).



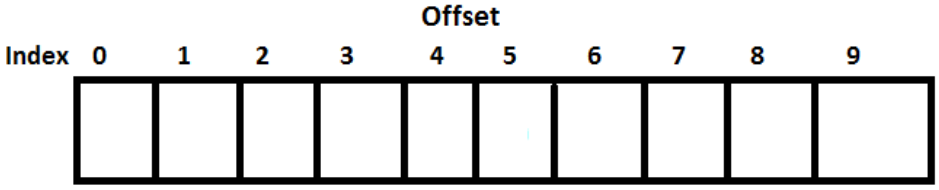
2. Fill in the least significant digit counter array like we did before, except count one at the counter index equal to the value of the LSD (we're treating it as if there's a decimal at the end so 0 counts as a significant digit). In our example, the first array element has a LSD of 1, so we add one at the counter index 1.



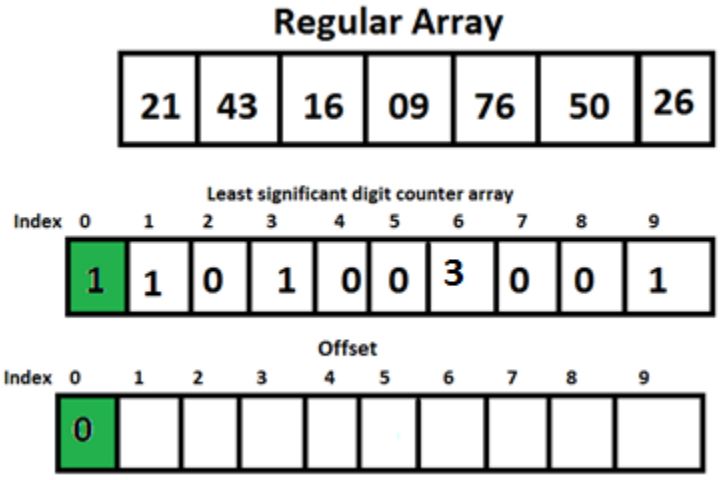
A few steps later…



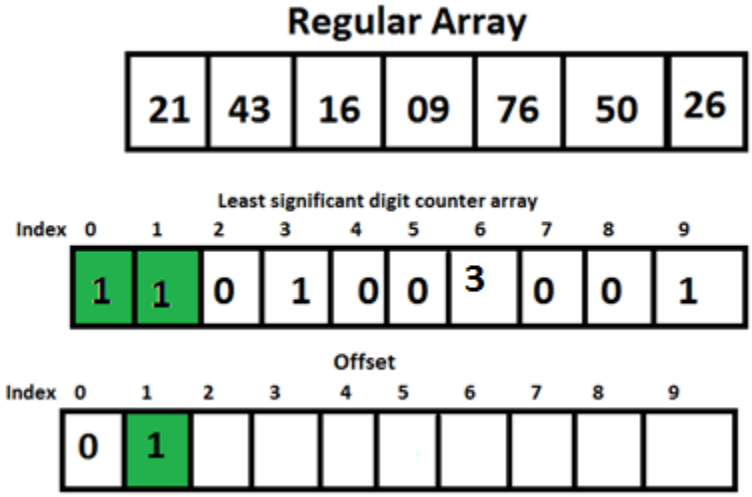
3. This part is a little tricky. Now, we create an array of 10 elements called the **offset array**.



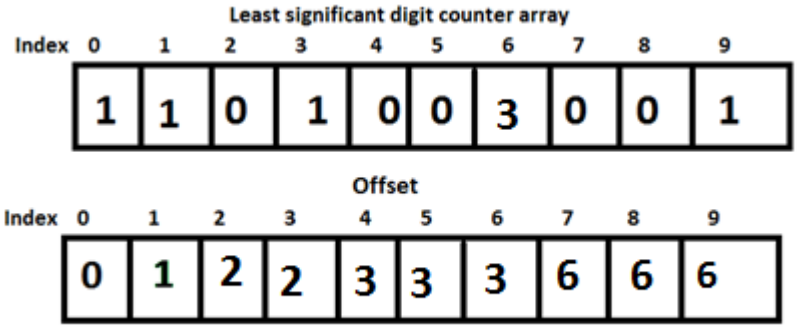
At each index of the offset, you add the first occurrence of the least significant digit in a sorted list. That's how it works technically, but here's the way I like to do it. Add all of the counter values to the left of the corresponding offset index.



The sum of the values to the left of counter index 0 is 0, so we place zero there.

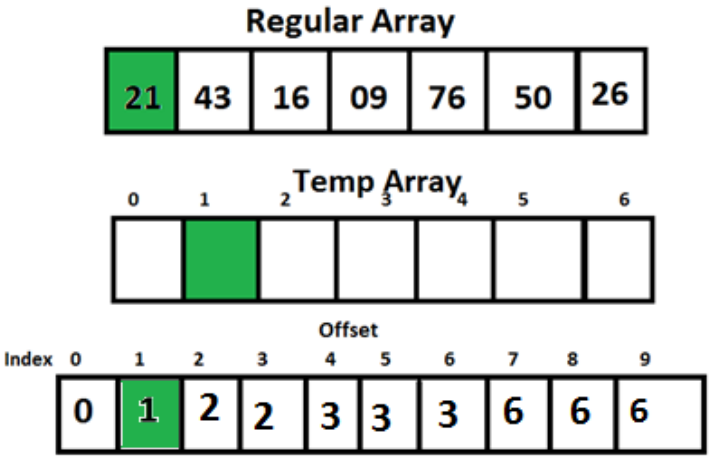


At offset index 1, the sum of the values to the left of index 1 is 1, so we put a 1 there.
Keep going until eventually, we get this:

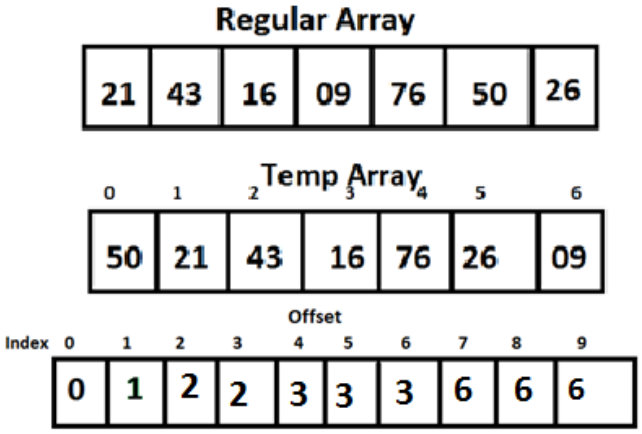


4. Using the offset, we can now insert elements into another temporary array. We do this by traversing the original array and inspecting the LSD. Then, we go to the index equal to the LSD in

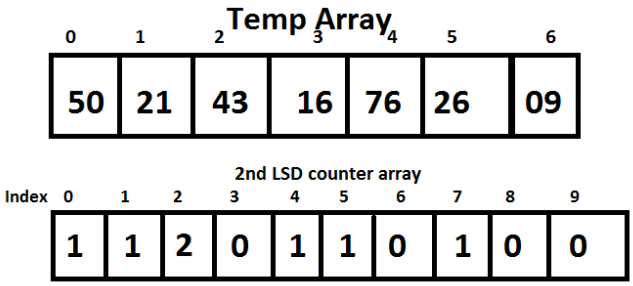
the offset array. We then place the element at the index equal to the value stored inside the offset array. For example,



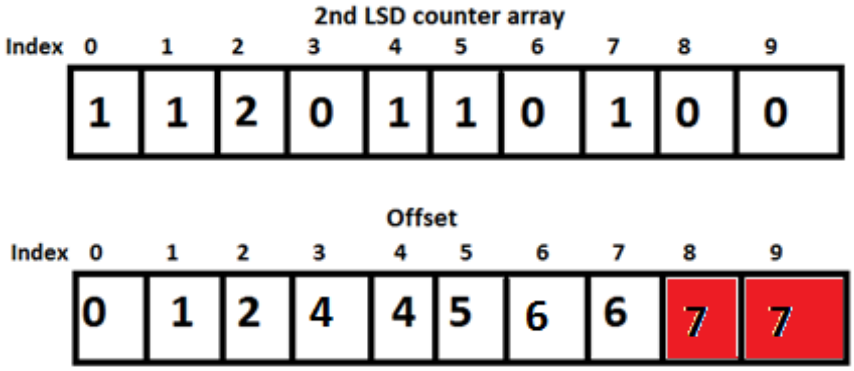
Our first element has a LSD of 1. It goes to the offset array at index 1, and sees that the value here is 1. Therefore, the element is inserted at index 1 of the temp array.
In the event of a collision, we move over one array index in the temp array.
The final result looks like this:



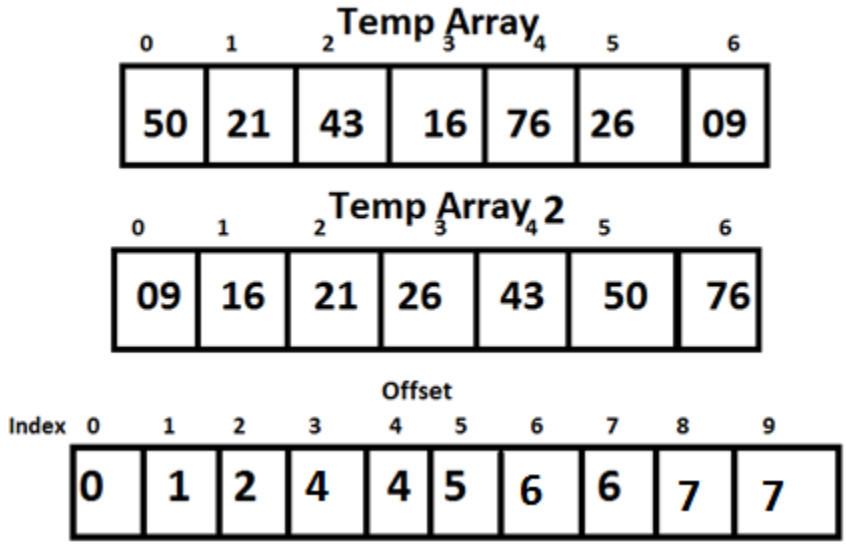
5. Now the elements are in order for LSD. We now replace the regular array with the temp array. Now, we need to repeat steps 1-4 for the *next least significant digit*. We keep going until we reach our most significant digit for the set of data, and we stop. When we finish, we will have a sorted list.
    a. Step 1 and 2



    b. Step 3



c. Step 4

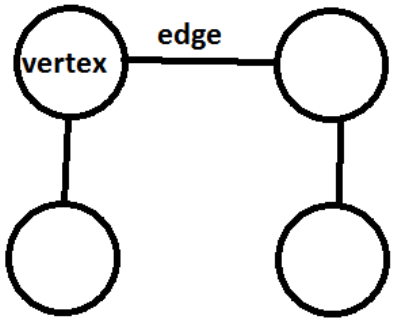


Temp array 2 is now our fully sorted array.

8. Graphs

Graph theory is the study and modelling of several different objects or tasks that have pairwise relations. Graph theory isn't used only in computer science. It can be used for project management, operations management, circuit analysis, and much more. There are different variations of graphs and I will go through each of them.

The two fundamental parts of a graph are the **edges** and the **vertices**. The vertices are the **nodes** in the graph, and the edges are what **connect the vertices**. Here is a diagram to show this:



8.1 Types of Graphs

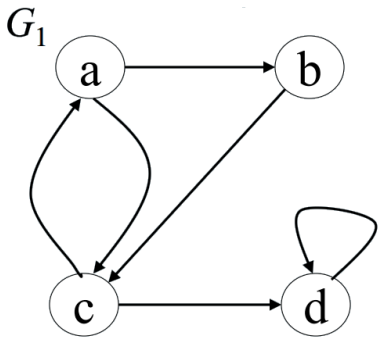
Vertices and edges are represented mathematically like a domain. The vertices are represented with variable **v** and the edges are represented with variable **e**. Vertices are represented the same way in directed and non-directed graphs. For example, if a graph had vertices named A, B, C, and D, the mathematical representation would be:

$$v = \{A, B, C, D\}$$

The edges of a graph differ for directed and non-directed graphs.

8.1.1 Directed Graphs

In a directed graph, the edges between the vertices have only one direction. This basically means that the vertices point to each other, as opposed to being connected. If you want two vertices to be 'connected', you would have two edges between them, and have them point to each other. Here's a diagram representation.



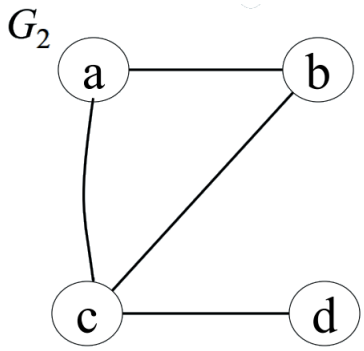
Edges are represented mathematically as ordered pairs, and are declared like this:

$$e = \{(a, b), (a, c), (b, c), (c, a), (c, d), (d, d)\}$$

Each vertex on the graph has an **in-degree** and an **out-degree**. An **in-degree** of a vertex is the number of edges pointing to the particular vertex (also called an **incident**). An **out-degree** of a vertex is the number of edges that the vertex is has pointing outward (also called an **emanation)**.

8.1.2 Undirected Graphs

Undirected is when two vertices are simply connected. Edges are declared simply as pairs of vertices that are connected.

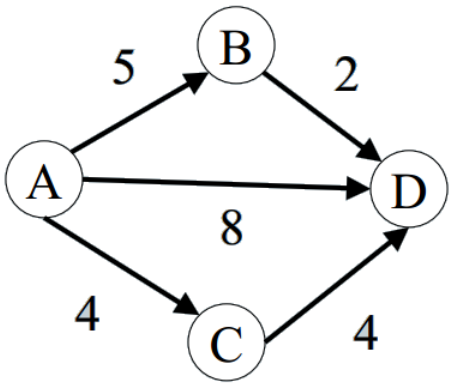


$$e = \{(a, b), (a, c), (b, c), (c, d)\}$$

8.1.3 Weighted Graphs

A weighted graph is a graph that gives weights to either the vertices or the edges. Weights are used to show that certain tasks may take up more time or resources than other tasks. If you were using this for supply chain management, you could think of weights as time taken for each task in manufacturing. If you were using this for finding the distance of your flight, weights could represent the distance to different airports.

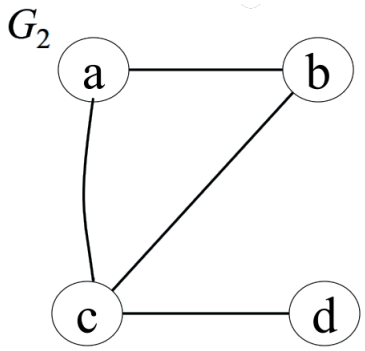
An **edge weighted graph** gives weights to the edges of the graph. Vertices would represent the start and end of the task.



A **vertex weighted graph** gives weights to vertices. Vertices would represent tasks, and edges would be the transition between tasks.

8.2 Matrix Representation of a Graph

Graphs can be represented in matrix form. It would be represented in a **v*v** matrix where v is the number of vertices in the graph. Each vertex is placed on the x and y axis of the matrix. Consider this graph:



You would construct the matrix putting the vertices on the x and y axes of the graph.

	A	B	C	d
A				
B				
C				
D				

Now you fill out your matrix. You first fill out the row for the 'A' vertex. You go across the matrix and see if there is an edge between the 'A' vertex and the other vertex. If there is, you fill in that cell of the matrix with the weight of the edge connecting the vertices (all edges have weight 1 for non-weighted graphs).

	A	B	C	d
A	**1**	**1**	**1**	**0**
B				
C				
D				

In an undirected graph, a vertex is assumed to be connected to itself, so we put a 1 there. The graph clearly shows edges between A and B, and A and C. This is why we put 1's for (A,A), (A,B), and (A,C).

Filled out, the matrix looks like this:

	A	B	C	d
A	1	1	1	0
B	1	1	1	0
C	1	1	1	1
D	9	9	1	1

8.3 Activity-Node and Event-Node Graph

Activity Node

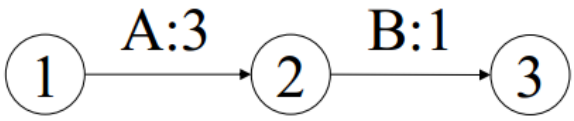
An activity node graph is basically a vertex weighted graph, except it's called an activity node graph. Vertices are called nodes. Each node contains an activity which has a weight based on time or resources taken for that activity. The edges are transitions between activities and do not have weights.

Event Node

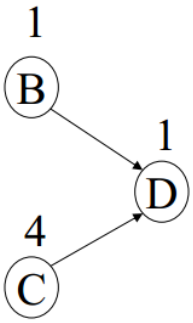
An event node graph is basically an edge weighted graph called an event node graph. It has nodes that represent the beginning or end of one or more tasks. The edges are the activities, and the edges are given weights based on resources or time taken by the activity.

Activity node graphs and event node graphs are duals of each other, and the graphs can be converted from one to the other. For example, this activity node graph:

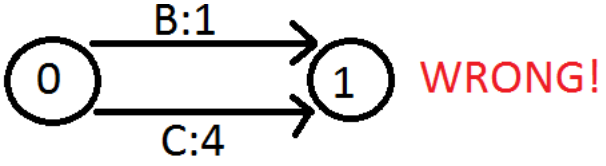
Can be converted to this event node graph:



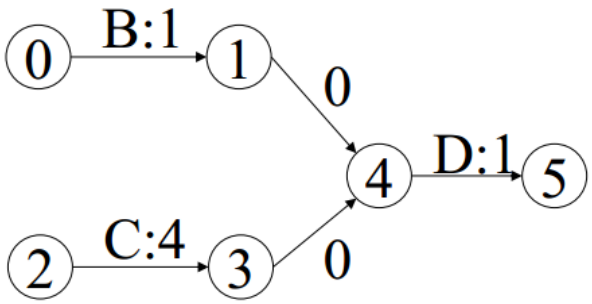
Note that if there's ever a case where an activity has two or more incidents, it is necessary to create dummy nodes for synchronization. In an event node graph, we don't want two nodes connected by two edges. Consider this case here:



B and C start at the same time. If we were to represent it without using dummy nodes, we would end up with this:



Instead, we make dummy nodes for the start and end times of C, and then have the end times of B and C point to the start of D using edges with a weight of 0.

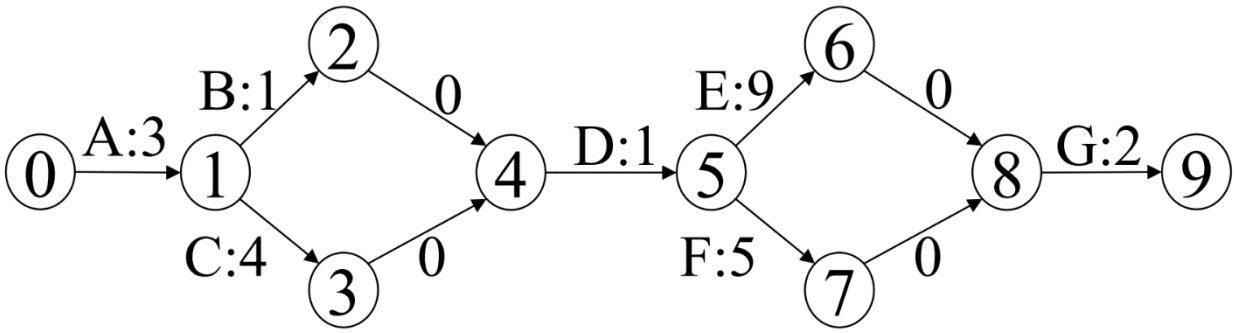


8.4 Critical Path Analysis

Critical path analysis is used to identify a path of tasks that **must start at a certain time** in order for the entire project to finish. Critical path analysis is used on **event node graphs**. In order to find the critical path, we must first find the early event times for every task, the late event times for every task, and then find the 'slack time'.

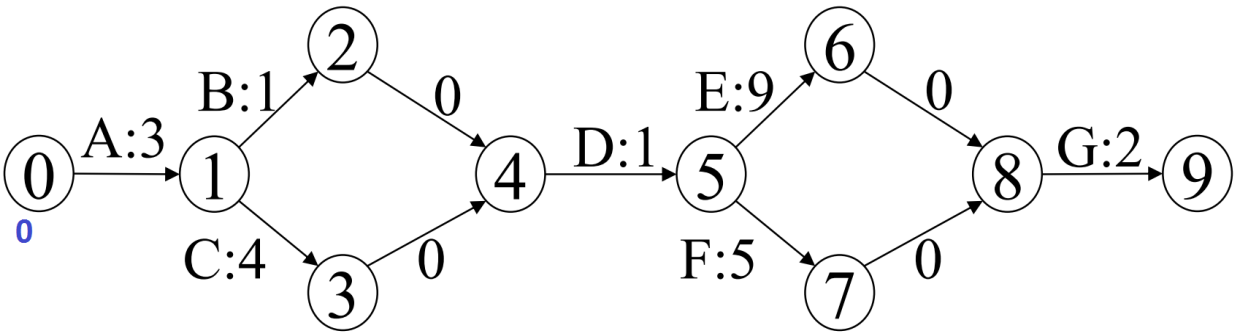
8.4.1 Earliest Event Time

Earliest event time is calculated at each start/end node. I can't explain this without an example so here is the example from lecture (and from the textbook).

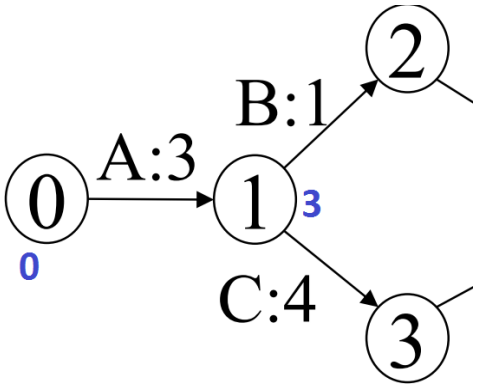


What you do is you add the weight edge before the node and the earliest value in the vertex that emanates that weight. This is done for cases when there is only one incident on the node.

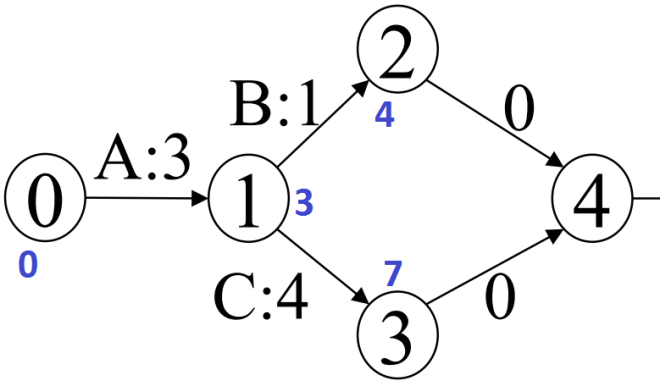
At node 0, there are no edges coming in, and it is the start, so the earliest event time will be 0 (note that the earliest event time is in blue in the diagram).



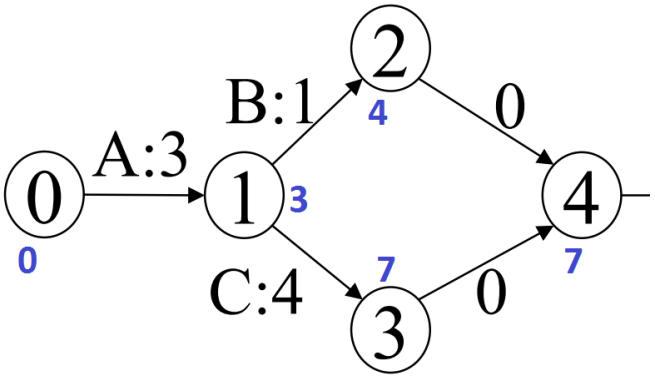
At node 1, the edge that incidents 1 has a weight of 3. The node that emanates that edge has an earliest event time of 0. We add 0 and 3 and that is our earliest event time at 1.



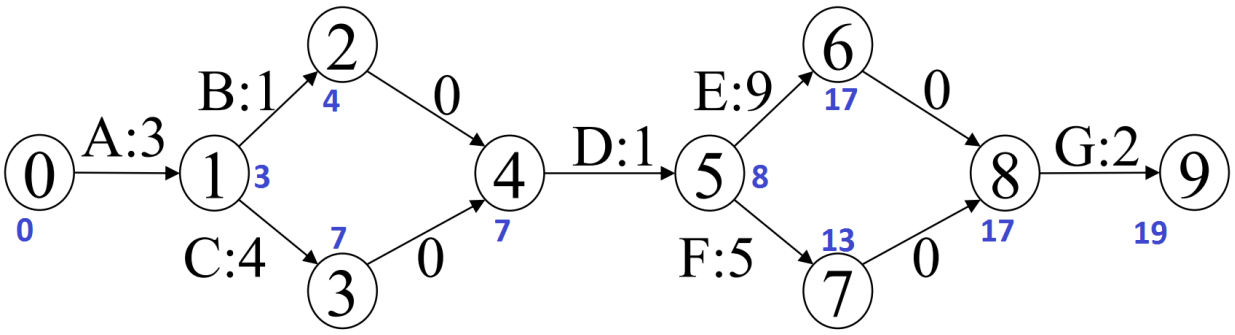
We do the exact same thing with 2 and 3 because there is only one incoming edge.



Node 4 is important. We have two incoming edges for node 4, **and both need to complete BEFORE the entire graph can move forward**. This means that we actually take the **LARGEST** incoming event time/weight sum. In order to reach node 4, activities at nodes 2 and 3 both need to be completed. Since the larger sum is 7, we use that.



Eventually, our entire graph looks like this.

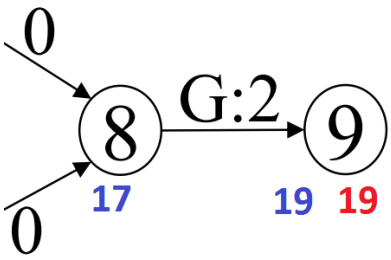


8.4.2 Latest Event Time

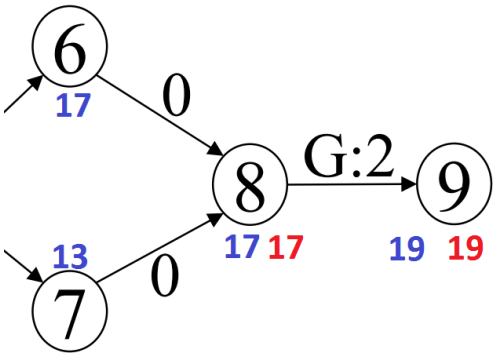
Latest event time is the latest time a task can start without delaying the project. You work backwards from the last node.

For nodes with only one edge emanating from it, you take the value of the succeeding node from that edge, and subtract the weight.

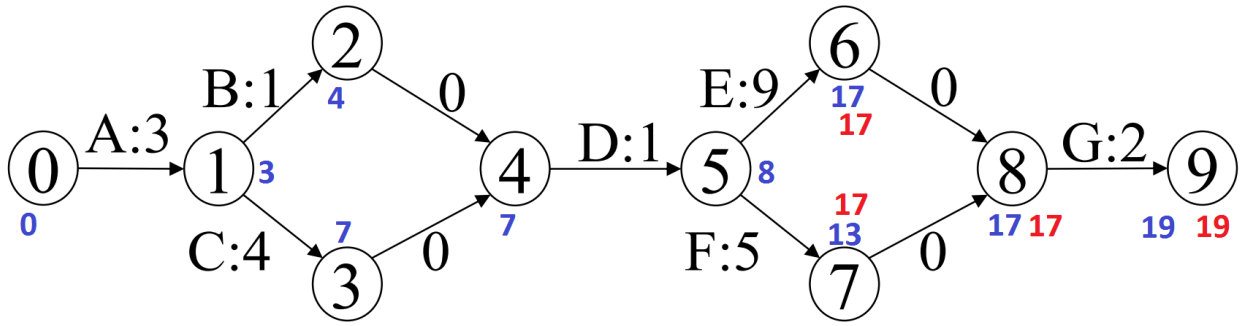
Again I'm going to use the example to illustrate this. The last node of the graph from before, has no edges emanating, so we use the value of 19 (the end node is a special).



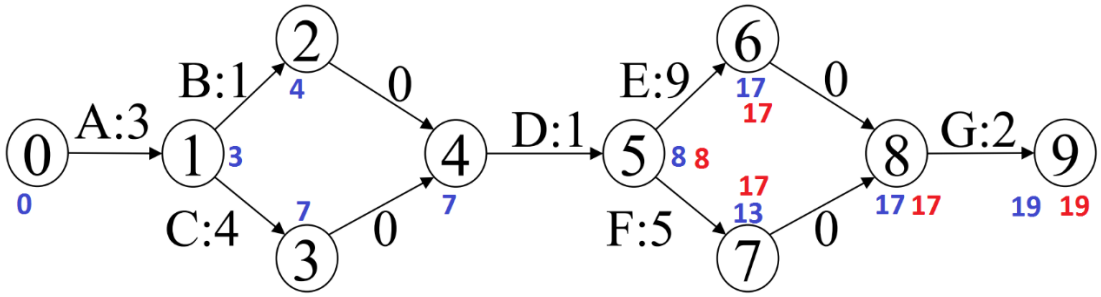
At node 8, there is one edge emanating with weight 2. The latest event time at node 9, which incidents the weight, is 19. Therefore, the latest event time at node 8 is 17.



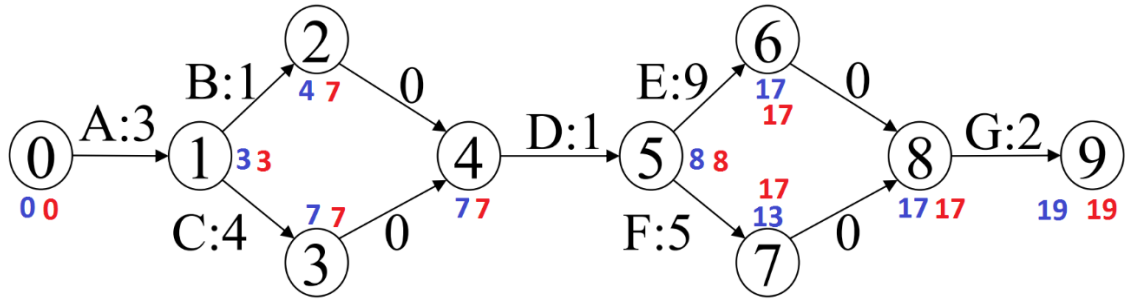
You do the same thing for nodes 6 and 7.



Now, at node 5, there are two emanating edges. You basically do the calculation for both edges and take the **SMALLER** value from the calculation. If we use event E, the latest event time at node 6 is 17, so the result is 8. If we use event F, the latest event time at node 7 is 17, so the resulting event time would be 12. 8 is smaller than 12 so we use 8.

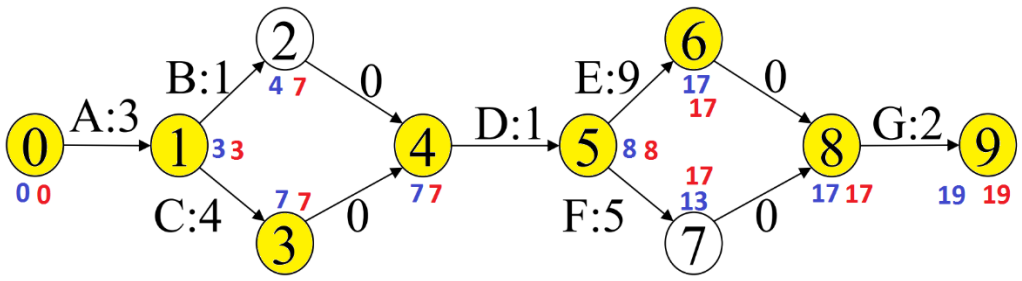


We repeat the process until we get this:



8.4.3 What is on the critical path?

Slack is the difference between the late and early event. Any event with 0 slack is on the critical path. In project management, this means that these tasks must be started at an exact time, otherwise the project will be late.

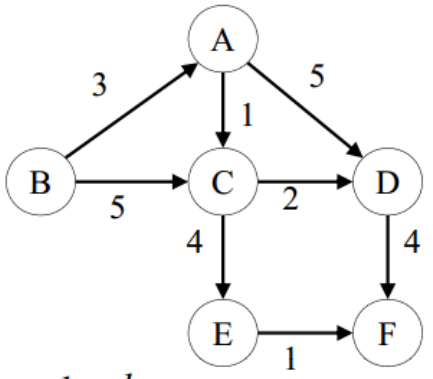


8.5 Dijsktra's Algorithm

Dijsktra's algorithm is a method of calculating the shortest path length on an edge weighted graph. The process goes like this:

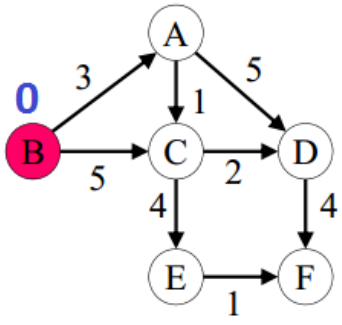
- Put values of infinity as weights for all of your nodes **except the starting node**, which has a value of 0. It is also important to keep track of which nodes have been visited (this is what Kv means in the slides). You start out with something like this:

$$v_s = \text{B}, v_g = F$$

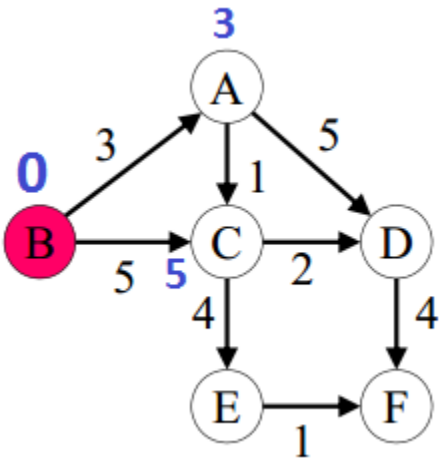


	k_v	d_v
A	0	∞
B	0	0
C	0	∞
D	0	∞
E	0	∞
F	0	∞

We want to start at B and end at F. D_V is the weight at the particular node on iteration. B starts off with a weight of 0. Now, on first iteration, we choose the unvisited node with the shortest distance as a start. This is B. I'm actually going to put the weights on the graph to make it easier to visualize.

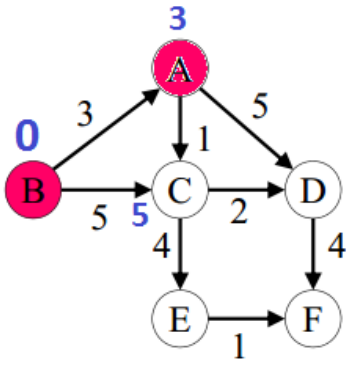


Now we find the distance between the current node and the neighbouring nodes (current weight + edge weight). The distances are then used as weights for the node, but ONLY IF the weight at the node is higher than the distance just calculated. In our case, the distance from B to A is 3, which is lower than infinity, so we replace the weight at A with 3 (assume any node without a weight has a weight of infinity).

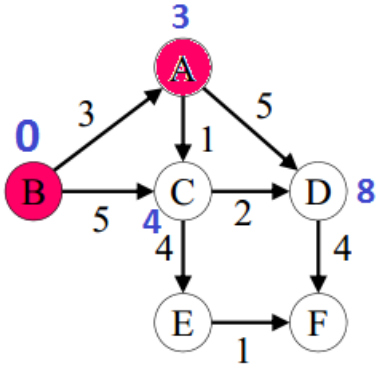


As you can see above, we did the same thing with C.
Note: In the chart in the lecture slides, they use the notation "weight / node", this basically means weight and node pointing to it which gives this weight.

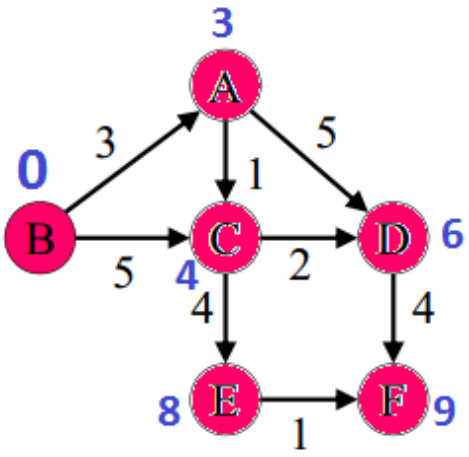
Since B has no more neighbours, we are done with our first iteration. We then choose another node to repeat these steps. The way we pick this node is we select an unselected node with the lowest weight. In our case, that is node A, with a weight of 3.



We now find the distances between A and nodes that emanate from A's edges. These would be C and D. The distance from A to C is 3 (current weight) plus 1 which would be 4. C currently has a weight of 5, but 4 is less than 5, so you replace the value at C with 4 (this is where the notation comes in handy, you would know where the weight of 4 comes from). You would do the same thing with D.



You repeat the process until you reach your final result, which looks like this.



The chart would look like this, with weights shown for ever iteration.

	k_v	d_v	1	2	3	4	5	6
A	1	∞	3/B	3/B	3/B	3/B	3/B	3/B
B	1	0/B	0/B	0/B	0/B	0/B	0/B	0/B
C	1	∞	5/B	4/A	4/A	4/A	4/A	4/A
D	1	∞	∞	8/A	6/C	6/C	6/C	6/C
E	1	∞	∞	∞	8/C	8/C	8/C	8/C
F	1	∞	∞	∞	∞	10/D	9/E	9/E

If you were to find the shortest path to F from the chart, you would start at the last row last column. In our example, there is 9/E there. This means that shortest path length is 9, and the vertex that provides this path length is E. You then go to whatever value is at E in the last column. The shortest path length to E is 8, which comes from C. You do the same for C, which then comes from A, which then comes from B. The shortest path is B, A, C, E, F, and has a path length of 9.

This algorithm assumes all weights are positive.

8.6 Other Graph Traversal Methods

8.6.1 Depth First Search

This involves a stack implementation of a graph. The steps are as follows:

- Start at any node you want. Push that node onto the stack.
- Pop that node from the stack, and keep track of the fact that it has already been pushed into the stack. Compare the value you're searching for with the value of the node just pushed out.
    - If it is the correct value, then we're done.
    - If it isn't, we push neighbouring nodes that have NOT been selected onto the stack. Repeat the process with the new node. Keep going until there are no more nodes.

8.6.2 Breadth First Search

This involves a queue implementation of a graph. The steps are as follows:

- Start at any node you want. Queue that node into the queue.
- Dequeue that node from the queue, and keep track of the fact that it has already been in the queue. Compare the value you're searching for with the value of the node just dequeued.
    - If it is the correct value, then we're done.
    - If it isn't, we queue neighbouring nodes that have NOT been selected into the queue. Repeat the process with the new nodes, and keep going until there are no more nodes.

8.6.3 Best First Search

This involves sorting distances between nodes to make the breadth or depth first search more efficient.

- Start at any node you want. Put that node into an empty sorted list.
- Withdraw the minimum distance node in that sorted list, and keep track of the fact that it was there. Compare the value that was removed to the value we are looking for.
    - If it is the correct value, then we're done.
    - If it isn't, we add neighbouring nodes that have not been in the list into the sorted list. We sort these nodes based on the shortest estimated distance to the value we want. In an unsorted graph, this is extremely difficult to estimate, but in an implementation like a heap, it is very easy to estimate.