LAB REPORT

DATA STRUCTURE AND IT'S APPLICATIONS

by Lima Islam

5th November,2018

Stack, Queues, Linked list, Array, Binary Tree

Table of Contents

Stack in Programming Terms 4

Stack Specification 4

How stack works 4

Use of stack 5

Queue Specifications 6

How Queue Works 6

Limitation of this implementation 7

Linked List Representation 9

How another node is referenced? 9

Linked List Operations 10

How to traverse a linked list 10

How to add elements to linked list 10

How to delete from a linked list 11

How to declare an array in C? 12

Elements of an Array and How to access them? 12

Few key notes: 12

How to initialize an array in C programming? 12

Multidimensional Array 13

How to initialize a multidimensional array? 13

Initialization of a two dimensional array 14

Initialization of a three dimensional array. 14

Multidimensional Arrays 14

Tree Terminology 16

Terminology 16

1. Root 16

2. Edge 16

3. Parent 17

4. Child 17

5. Siblings 18

6. Leaf 18

7. Internal Nodes 18

8. Degree 19
9. Level 19
10. Height 20
11. Depth 20
12. Path 20
13. Sub Tree 21
Binary Tree Representations 21
1. Array Representation 21
2. Linked List Representation 22
Binary Tree Traversals 22
1. In - Order Traversal 22
2. Pre - Order Traversal 22
3. Post - Order Traversal 22
1. In - Order Traversal (leftChild - root - rightChild) 23
2. Pre - Order Traversal (root - leftChild - rightChild) 23
3. Post - Order Traversal (leftChild - rightChild - root) 23
Priority Queue 23
1. Max Priority Queue 23
2. Min Priority Queue 23
1. Max Priority Queue 23
1. isEmpty() - Check whether queue is Empty. 23
2. insert() - Inserts a new value into the queue. 23
3. findMax() - Find maximum value in the queue. 23
4. remove() - Delete maximum value from the queue. 23
Max Priority Queue Representations 24
1. Using an Unordered Array (Dynamic Array) 24
2. Using an Unordered Array (Dynamic Array) with the index of the maximum value 24
3. Using an Array (Dynamic Array) in Decreasing Order 24
4. Using an Array (Dynamic Array) in Increasing Order 24
5. Using Linked List in Increasing Order 24
6. Using Unordered Linked List with reference to node with the maximum value 24
2. Min Priority Queue Representations 24

1. isEmpty() - Check whether queue is Empty. 24
2. insert() - Inserts a new value into the queue. 24
3. findMin() - Find minimum value in the queue. 24
4. remove() - Delete minimum value from the queue. 24
Heap Data Structure 24
Operations on Max Heap 25
Finding Maximum Value Operation in Max Heap 25
Insertion Operation in Max Heap 25
Deletion Operation in Max Heap 27
..... 28
..... 28

Stack

A stack is a useful data structure in programming. It is just like a pile of plates kept on top of each other.

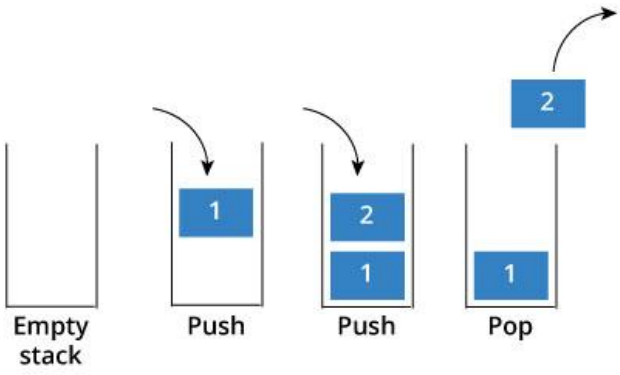
Think about the things you can do with such a pile of plates

- Put a new plate on top
- Remove the top plate

If you want the plate at the bottom, you have to first remove all the plates on top. Such kind of arrangement is called **Last In First Out** - the last item that was placed is the first item to go out.

Stack in Programming Terms

In programming terms, putting an item on top of the stack is called "push" and removing an item is called "pop".



In the above image, although item 2 was kept last, it was removed first - so it follows the Last In First Out(LIFO) principle.

Stack Specification

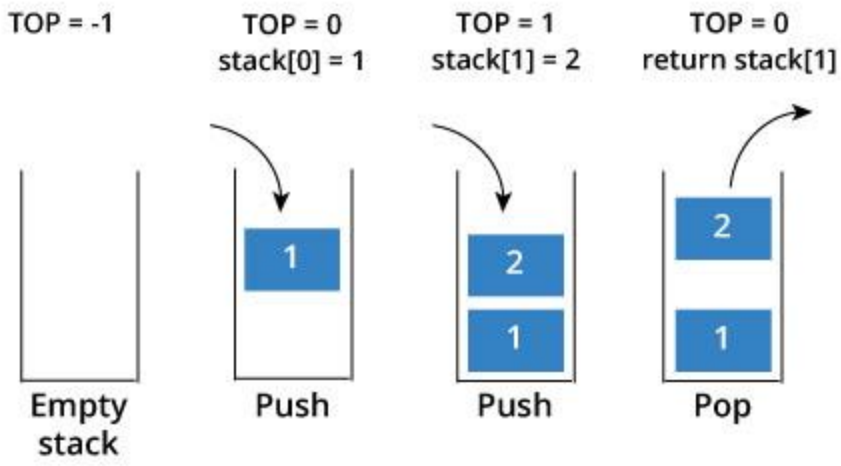
A stack is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- `Push`: Add element to top of stack
- `Pop`: Remove element from top of stack
- `IsEmpty`: Check if stack is empty
- `IsFull`: Check if stack is full
- `Peek`: Get the value of the top element without removing it

How stack works

The operations work as follows:

1. A pointer called *TOP* is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing `TOP == -1`.
3. On pushing an element, we increase the value of *TOP* and place the new element in the position pointed to by *TOP*.
4. On popping an element, we return the element pointed to by *TOP* and reduce its value.
5. Before pushing, we check if stack is already full
6. Before popping, we check if stack is already empty



Use of stack

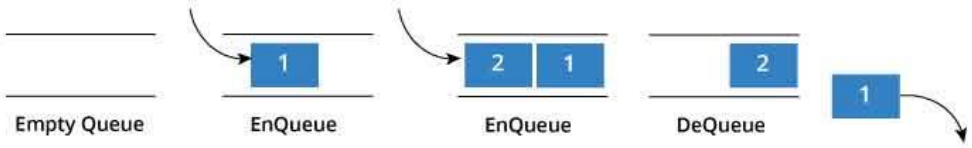
Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

- **To reverse a word** - Put all the letters in a stack and pop them out. Because of LIFO order of stack, you will get the letters in reverse order.
- **In compilers** - Compilers use stack to calculate the value of expressions like 2+4/5*(7-9) by converting the expression to prefix or postfix form.
- **In browsers** - The back button in a browser saves all the urls you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack and the previous url is accessed.

Queue

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the **First In First Out(FIFO)** rule - the item that goes in first is the item that comes out first too.



In the above image, since 1 was kept in the queue before 2, it was the first to be removed from the queue as well. It follows the FIFO rule.

In programming terms, putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".

We can implement queue in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

Queue Specifications

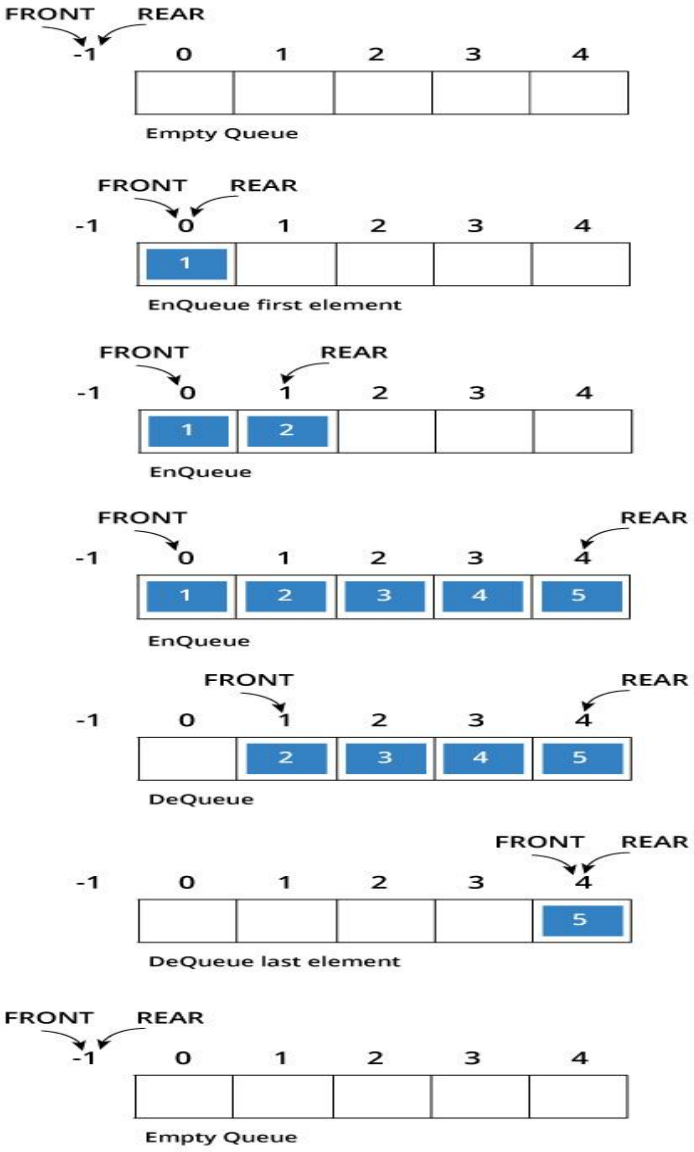
A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- Enqueue: Add element to end of queue
- Dequeue: Remove element from front of queue
- IsEmpty: Check if queue is empty
- IsFull: Check if queue is full
- Peek: Get the value of the front of queue without removing it

How Queue Works

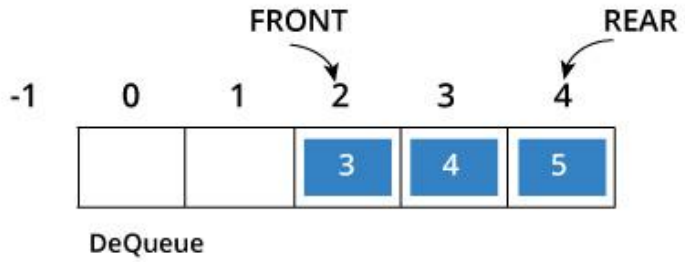
Queue operations work as follows:

1. Two pointers called *FRONT* and *REAR* are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of *FRONT* and *REAR* to -1.
3. On enqueing an element, we increase the value of *REAR* index and place the new element in the position pointed to by *REAR*.
4. On dequeueing an element, we return the value pointed to by *FRONT* and increase the *FRONT* index.
5. Before enqueing, we check if queue is already full.
6. Before dequeuing, we check if queue is already empty.
7. When enqueing the first element, we set the value of *FRONT* to 0.
8. When dequeing the last element, we reset the values of *FRONT* and *REAR* to -1.



Limitation of this implementation

As you can see in the image below, after a bit of enqueueing and dequeueing, the size of the queue has been reduced.



The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

By tweaking the code for queue, we can use the space by implementing a modified queue called

Linked list

A linked list is similar. It is a series of connected "nodes" that contains the "address" of the next node. Each node can store a data point which may be a number, a string or any other type of data.

Linked List Representation

You have to start somewhere, so we give the address of the first node a special name called *HEAD*.

Also, the last node in the linkedlist can be identified because its next portion points to *NULL*.

How another node is referenced?

Some pointer magic is involved. Let's think about what each node contains:

- A data item
- An address of another node

We wrap both the data item and the next node reference in a struct as:

```
struct node
{
  int data;
  struct node *next;
};
```

n just a few steps, we have created a simple linkedlist with three nodes.

The power of linkedlist comes from the ability to break the chain and rejoin it. E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

- Create a new struct node and allocate memory to it.
- Add its data value as 4
- Point its next pointer to the struct node containing 2 as data value
- Change next pointer of "1" to the node we just created.

Doing something similar in an array would have required shifting the positions of all the subsequent elements.

Linked List Operations

Now that you have got an understanding of the basic concepts behind linked list and their types, its time to dive into the common operations that can be performed.

Two important points to remember:

- *head* points to the first node of the linked list
- *next* pointer of last node is *NULL*, so if next of current node is *NULL*, we have reached end of linked list.

In all of the examples, we will assume that the linked list has three nodes `1 --->2 --->3` with node structure as below:

```
struct node
{
  int data;
  struct node *next;
};
```

How to traverse a linked list

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When temp is *NULL*, we know that we have reached the end of linked list so we get out of the while loop

How to add elements to linked list

You can add elements to either beginning, middle or end of linked list.

Add to beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

Add to end

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

Add to middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

How to delete from a linked list

You can delete either from beginning, end or from a particular position.

Delete from beginning

- Point head to the second node

Delete from end

- Traverse to second last element
- Change its next pointer to null

Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

Array

An array is a collection of data that holds fixed number of values of same type. For example: if you want to store marks of 100 students, you can create an array for it.

The size and type of arrays cannot be changed after its declaration.

Arrays are of two types:

1. One-dimensional arrays
2. Multidimensional arrays

How to declare an array in C?

For example,

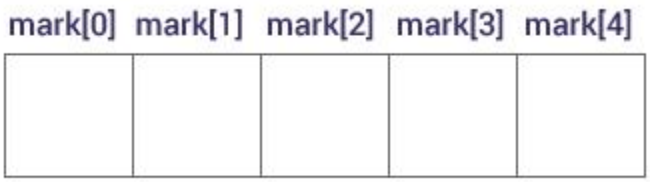
```
float mark[5];
```

Here, we declared an array, *mark*, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

Elements of an Array and How to access them?

You can access elements of an array by indices.

Suppose you declared an array *mark* as above. The first element is *mark[0]*, second element is *mark[1]* and so on.



Few key notes:

- Arrays have 0 as the first index not 1. In this example, *mark[0]*
- If the size of an array is *n*, to access the last element, `(n-1)` index is used. In this example, *mark[4]*
- Suppose the starting address of `mark[0]` is 2120d. Then, the next address, `a[1]`, will be 2124d, address of `a[2]` will be 2128d and so on. It's because the size of a float is 4 bytes.

How to initialize an array in C programming?

It's possible to initialize an array during declaration. For example,

```
int mark[5] = {19, 10, 8, 17, 9};
```

Multidimensional Array

In C programming, you can create an array of arrays known as multidimensional array. For example,

Here, *x* is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as table with 3 row and each row has 4 column.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Similarly, you can declare a three-dimensional (3d) array. For example,

```
float y[2][4][3];
```

Here, The array *y* can hold 24 elements.

You can think this example as: Each 2 elements have 4 elements, which makes 8 elements and each 8 elements can have 3 elements. Hence, the total number of elements is 24.

How to initialize a multidimensional array?

There is more than one way to initialize a multidimensional array.

Initialization of a two dimensional array

```
// Different ways to initialize two dimensional array

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
```

int c[2][3] = {1, 3, 0, -1, 5, 9}; Above code are three different ways to initialize a two dimensional arrays.

Initialization of a three dimensional array.

You can initialize a three dimensional array in a similar way like a two dimensional array. Here's an example,

```
int test[2][3][4] = {
                        { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} },
                        { {13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9} }
                    };
```

Multidimensional Arrays

- Multi-dimensional arrays are declared by providing more than one set of square [ ] brackets after the variable name in the declaration statement.
- One dimensional arrays do not require the dimension to be given if the array is to be completely initialized. By analogy, multi-dimensional arrays do not require **the first** dimension to be given if the array is to be completely initialized. All dimensions after the first must be given in any case.
- For two dimensional arrays, the first dimension is commonly considered to be the number of rows, and the second dimension the number of columns. We will use this convention when discussing two dimensional arrays.
- Two dimensional arrays are considered by C/C++ to be an array of (single dimensional arrays). For example, "int numbers[5][6]" would refer to a single dimensional array of 5 elements, wherein each element is a single dimensional array of 6 integers. By extension, "int numbers[12][5][6]" would refer to an array of twelve elements, each of which is a two dimensional array, and so on.
- Another way of looking at this is that C stores two dimensional arrays by rows, with all elements of a row being stored together as a single unit. Knowing this can sometimes lead to more efficient programs.
- Multidimensional arrays may be completely initialized by listing all data elements within a single pair of curly {} braces, as with single dimensional arrays.

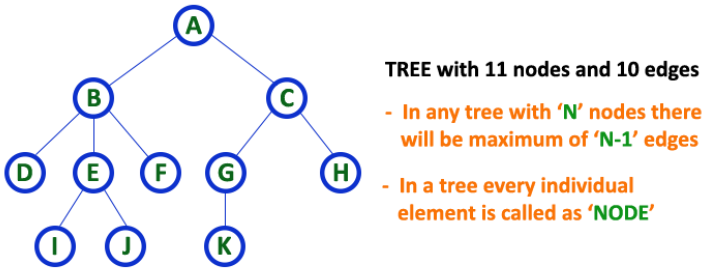
- It is better programming practice to enclose each row within a separate subset of curly {} braces, to make the program more readable. This is required if any row other than the last is to be partially initialized. When subsets of braces are used, the last item within braces is not followed by a comma, but the subsets are themselves separated by commas.
- Multidimensional arrays may be partially initialized by not providing complete initialization data. Individual rows of a multidimensional array may be partially initialized, provided that subset braces are used.
- Individual data items in a multidimensional array are accessed by fully qualifying an array element. Alternatively, a smaller dimensional array may be accessed by partially qualifying the array name. For example, if "data" has been declared as a three dimensional array of floats, then data[1][2][5] would refer to a float, data[1][2] would refer to a one-dimensional array of floats, and data[1] would refer to a two-dimensional array of floats. The reasons for this and the incentive to do this relate to memory-management issues that are beyond the scope of these notes.

Tree

Tree Terminology

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

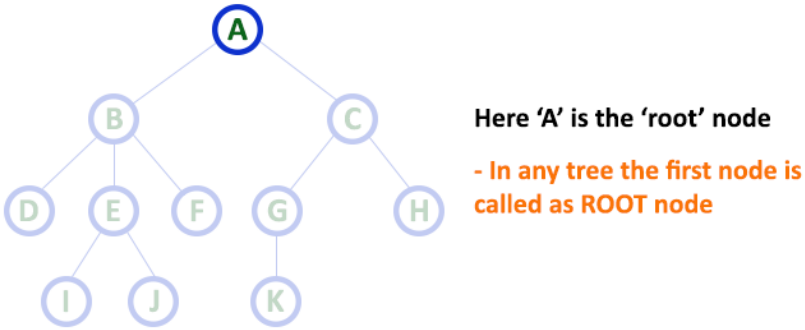
Example:



Terminology

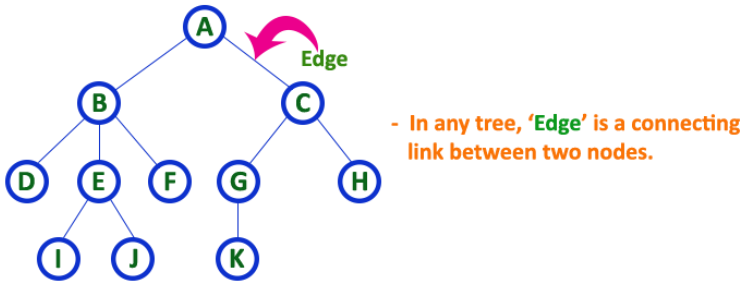
1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.



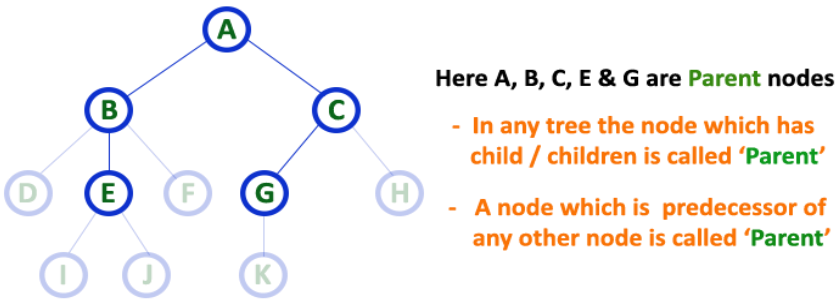
2. Edge

In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



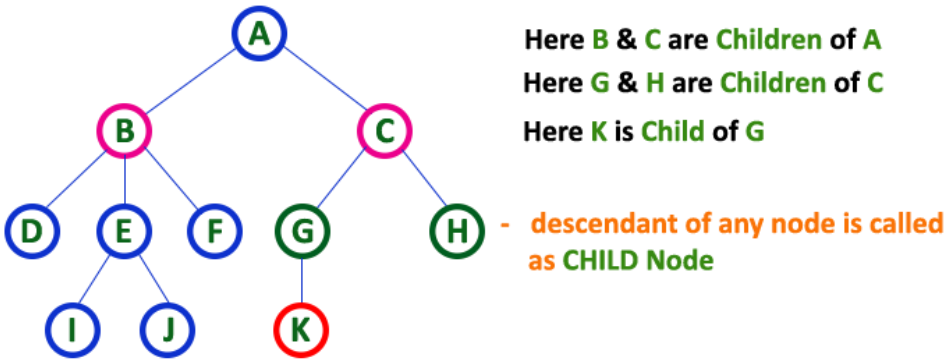
3. Parent

In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".



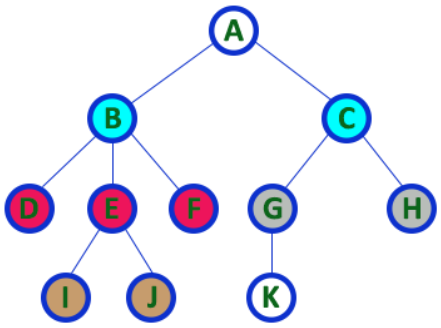
4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



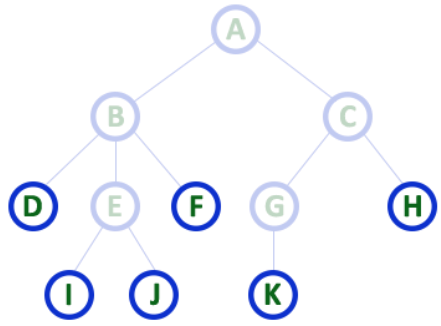
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree,

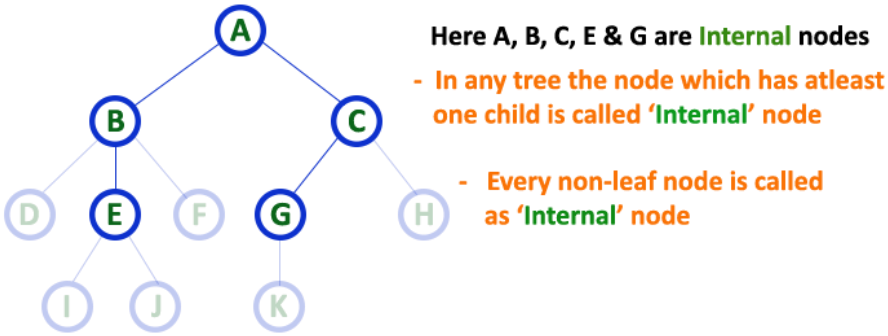


Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

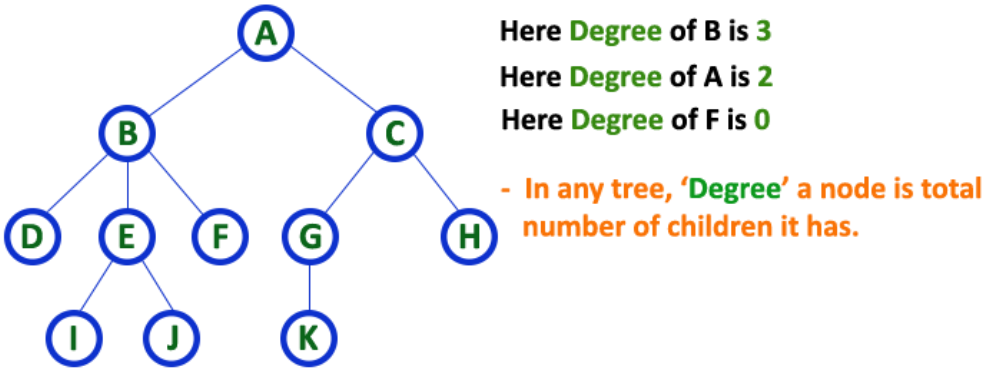
7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.



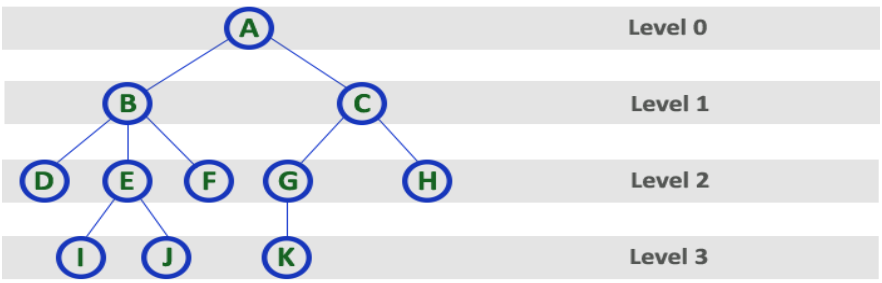
8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



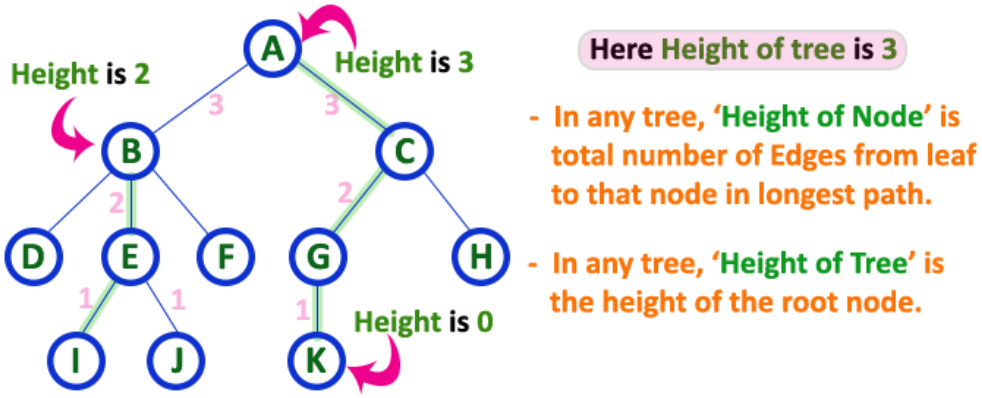
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



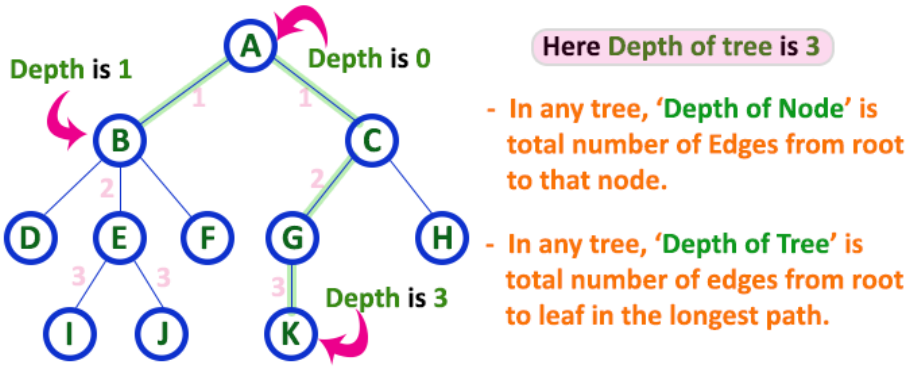
10. Height

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node.



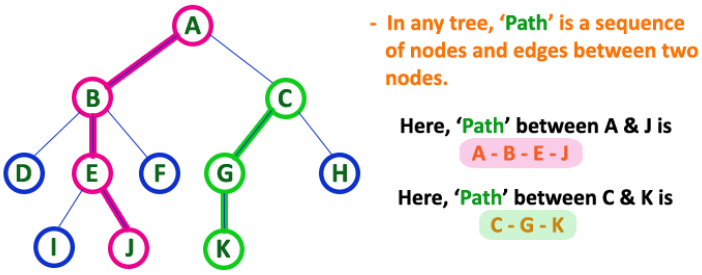
11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**



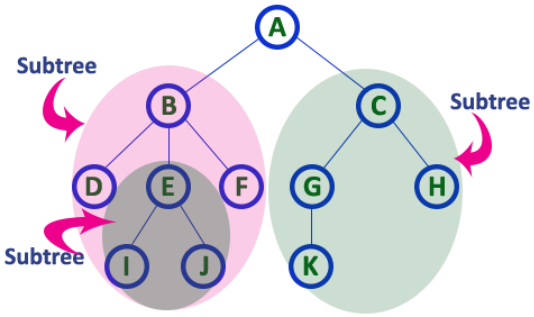
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

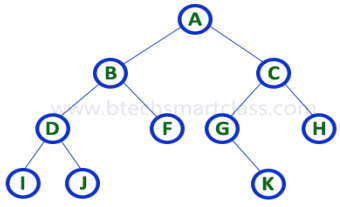


Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation 2. Linked List Representation

Consider the following binary tree...



1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. Consider the above example of binary tree and it is represented as follows...

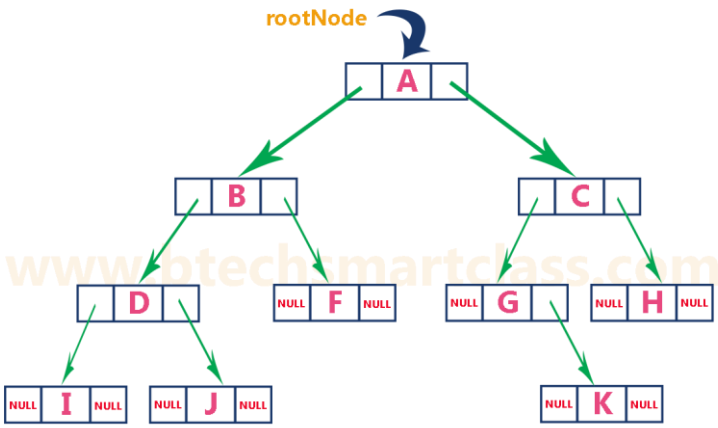
To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n+1 - 1.

2. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...

The above example of binary tree represented using Linked list representation is shown as follows...

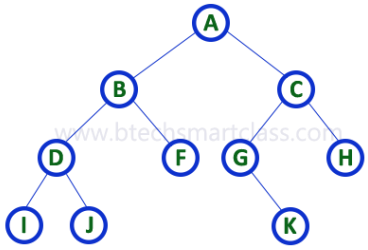


Binary Tree Traversals

There are three types of binary tree traversals.

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

Consider the following binary tree...



1. In - Order Traversal (leftChild - root - rightChild)

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (root - leftChild - rightChild)

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

3. Post - Order Traversal (leftChild - rightChild - root)

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Priority Queue

There are two types of priority queues they are as follows...

1. **Max Priority Queue**
2. **Min Priority Queue**

1. Max Priority Queue

n max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2. The following are the operations performed in a Max priority queue...

1. **isEmpty() - Check whether queue is Empty.**
2. **insert() - Inserts a new value into the queue.**
3. **findMax() - Find maximum value in the queue.**
4. **remove() - Delete maximum value from the queue.**

Max Priority Queue Representations

There are 6 representations of max priority queue.

1. **Using an Unordered Array (Dynamic Array)**
2. **Using an Unordered Array (Dynamic Array) with the index of the maximum value**
3. **Using an Array (Dynamic Array) in Decreasing Order**
4. **Using an Array (Dynamic Array) in Increasing Order**
5. **Using Linked List in Increasing Order**
6. **Using Unordered Linked List with reference to node with the maximum value**

2. Min Priority Queue Representations

Min Priority Queue is similar to max priority queue except removing maximum element first, we remove minimum element first in min priority queue.

The following operations are performed in Min Priority Queue...

1. **isEmpty() - Check whether queue is Empty.**
2. **insert() - Inserts a new value into the queue.**
3. **findMin() - Find minimum value in the queue.**
4. **remove() - Delete minimum value from the queue.**

Min priority queue is also has same representations as Max priority queue with minimum value removal.

Heap Data Structure

There are two types of heap data structures and they are as follows...

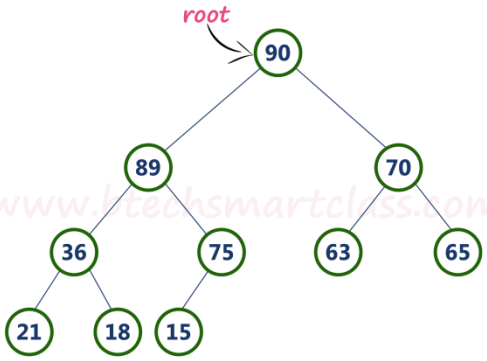
1. **Max Heap**
2. **Min Heap**

Every heap data structure has the following properties...

Property #1 (Ordering): Nodes must be arranged in a order according to values based on Max heap or Min heap.

Property #2 (Structural): All levels in a heap must full, except last level and nodes must be filled from left to right strictly.

Example:



Operations on Max Heap

The following operations are performed on a Max heap data structure...

1. **Finding Maximum**
2. **Insertion**
3. **Deletion**

Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In max heap, the root node has the maximum value than all other nodes in the max heap. So, directly we can display root node value as maximum value in max heap.

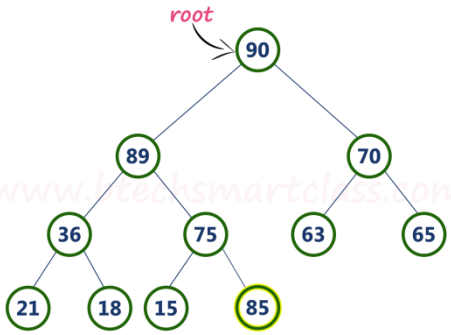
Insertion Operation in Max Heap

Insertion Operation in max heap is performed as follows...

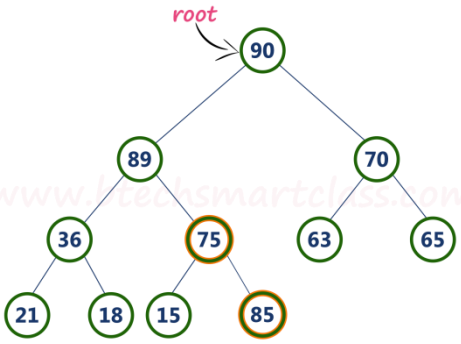
- **Step 1:** Insert the **newNode** as **last leaf** from left to right.
- **Step 2:** Compare **newNode value** with its **Parent node**.
- **Step 3:** If **newNode value is greater** than its parent, then **swap** both of them.
- **Step 4:** Repeat step 2 and step 3 until newNode value is less than its parent nede (or) newNode reached to root.

Example
Consider the above max heap. **Insert a new node with value 85.**

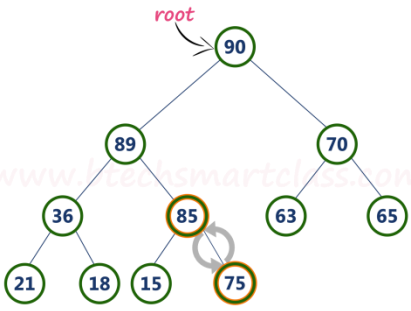
- **Step 1:** Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...



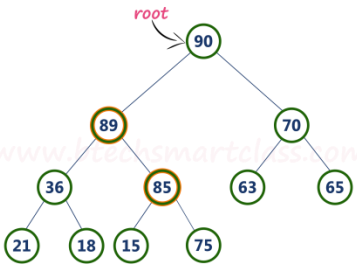
- **Step 2:** Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**



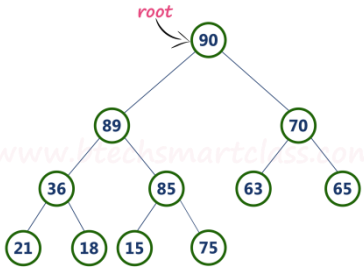
Step 3: Here **newNode value (85) is greater** than its **parent value (75)**, then **swap** both of them. After wsapping, max heap is as follows...



tep 4: Now, again compare newNode value (85) with its parent nede value (89).



Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, max heap after insetion of a new node with value 85 is as follows...



Deletion Operation in Max Heap

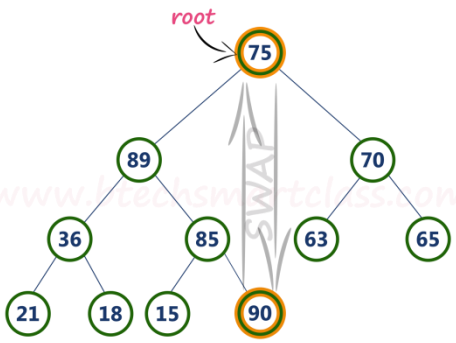
In a max heap, deleting last node is very simple as it is not disturbing max heap properties.

Deleting root node from a max heap is title difficult as it disturbing the max heap properties. We use the following steps to delete root node from a max heap...

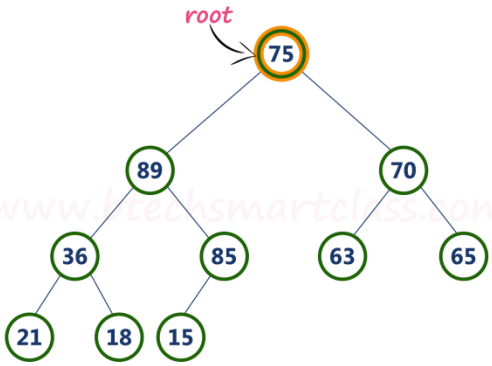
- **Step 1: Swap** the **root** node with **last** node in max heap
- **Step 2: Delete** last node.
- **Step 3:** Now, compare **root value** with its **left child value**.
- **Step 4:** If **root value is smaller** than its left child, then compare **left child** with its **right sibling**. Else goto **Step 6**
- **Step 5:** If **left child value is larger** than its **right sibling**, then **swap root** with **left child**. otherwise **swap root** with its **right child**.
- **Step 6:** If **root value is larger** than its left child, then compare **root value** with its **right child** value.
- **Step 7:** If **root value is smaller** than its **right child**, then **swap root** with **rith child**. otherwise **stop the process**.
- **Step 8:** Repeat the same until root node is fixed at its exact position.

Example
Consider the above max heap. **Delete root node (90) from the max heap.**

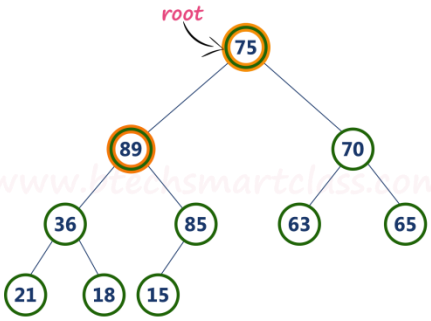
- **Step 1: Swap** the **root node (90)** with **last node 75** in max heap After swapping max heap is as follows...



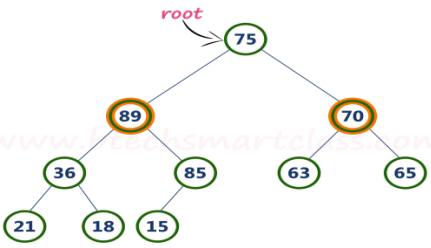
Step 2: Delete last node. Here node with value 90. After deleting node with value 90 from heap, max heap is as follows...



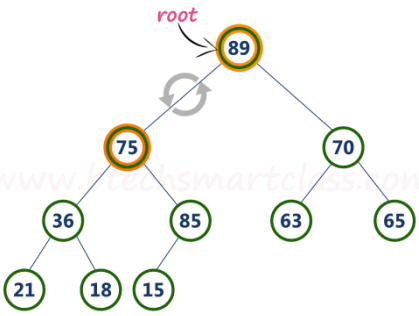
Step 3: Compare **root node (75)** with its **left child (89)**.



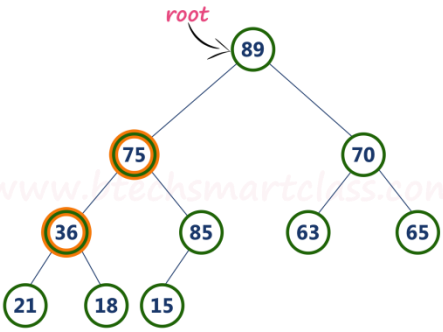
Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).



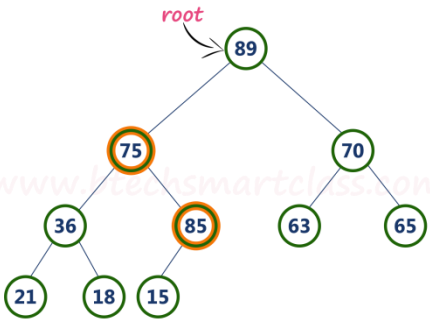
Step 4: Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.



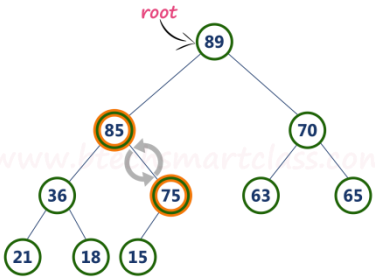
Step 5: Now, again compare **75** with its **left child (36)**.



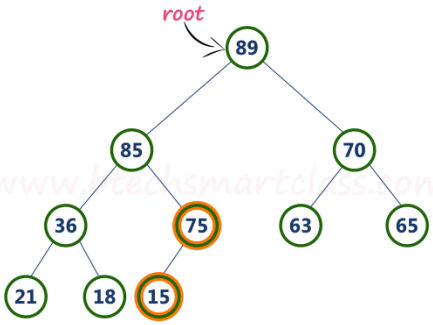
Here, node with value **75** is larger than its left child. So, we compare node with value **75** is compared with its right child **85**.



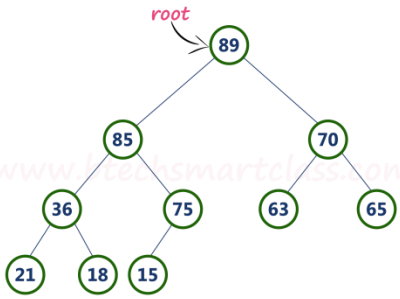
Step 6: Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...



Step 7: Now, compare node with value **75** with its left child (**15**).



Finally, max heap after deleting root node (**90**) is as follows...



References

1.Array *https://www.w3schools.in/data-structures-tutorial/data-structures-arrays/*

2.Stack *https://www.studytonight.com/data-structures/stack-data-structure*

3.Queue https://www.hackerearth.com/practice/data-structures/queues/basics-of-queues/tutorial/

4.Linked List *https://www.geeksforgeeks.org/data-structures/linked-list/*

5.Tree https://www.geeksforgeeks.org/binary-tree-data-structure/

6.Sorting https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm