
**230 HOURS SUMMER TRAINING REPORT**

**On**

**"Data Structures and Algorithms - Self Paced"**

Submitted by

**Kamma Sai Pujitha**

**Registration Number :12209373**

**Program Name :B.Tech.(CSE)**

Under the guidance of

**Mr. Sandeep Jain**

**Assistant Professor**

**School of Computer Science and Engineering**

**Lovely Professional University, Phagwara**

(June-July,2024)

## DECLARATION

**I hereby declare that I have completed my 230-Hours summer training at GeeksForGeeks from 6th June 2024 to 24th July 2024 under the guidance of Mr. SandeepJain. I declare that I have worked with full dedication during these 230 Hours of training and my learning outcomes fulfill the requirements of training for the award of degree of B.Tech.(Computer Science & Engineering) at Lovely Professional University, Phagwara.**

k.Sai Pujitha

Registration No: 12209373

# ACKNOWLEDGEMENT

To acknowledge all the persons who had helped in completing the training and project is not possible for any trainee. However, despite all that, it becomes a foremost responsibility of the trainee and the part of research ethics to acknowledge those who had played a significant role in the completion of the training and project.

So, in the same sequence at very first, we would like to acknowledge Mr. Sandeep Jain, the mentor of our Training. He has been instrumental in guiding and helping us while undergoing the planning phase of our project and helped clear the concepts related to the topics and project. Because the Training and Placement Cell of School of Computer Science & Engineering has allowed me to do the summer training, I would like to thank for the same. I would like to thank Data Structures and Algorithms - Self Paced, GreeksForGeeks also for organizing such a wonderful summer training program.

Later, I would like to confer the flower of acknowledgment to our parents because of whom we got the existence in the world for the inception and the conception of this training and project. Rest all those people who helped us are not only a matter of acknowledgment but also authorized to share our success.

With regards

K.Sai Pujitha

**SUMMER TRAINING CERTIFICATE**


# CERTIFICATE

## OF COURSE COMPLETION

THIS IS TO CERTIFY THAT

**Kamma Sai Pujitha**

has successfully completed a 230-hour course on Data Structures and Algorithms - Self Paced.

Sandeep Jain

**Mr. Sandeep Jain**
Founder & CEO, GeeksforGeeks
https://media.geeksforgeeks.org/courses/certificates/504e0fadd735a4bdd161f6cf0b351fea.pdf

# TABLE OF CONTENTS

# 1. INTRODUCTION

Online Summer Training Course on Data Structures And Algorithms Self-Paced has helped me in Understanding the fundamentals of various data structures and applying for solving real-world problems. It has updated my skills in developing useful projects by applying the concepts of Data Structures and Algorithms. I have undergone experiential learning that has covered the basics of programming, fundamentals of Data Structures and Application of the same in solving some competitive coding problems.

After doing this course I had :

- Acquired basic programming skills using Data Structures.
- Worked with a Variety of Data Structures.
- Solved basic competitive coding problems using Data Structures.
- Applied Data Structures and Algorithmic concepts in developing useful projects.

# 2. Technology Learnt

During the 230 Hours online summer training " Data Structures And Algorithms Self-Paced ", I have learnt basics of Data Structures and applied those concepts in solving some standard competitive coding problems using C++. The brief outlines of the contents covered is as following:

**2.1 Data Structures :** Data structures are a specific way of organizing data in a specialized format on a computer so that the information can be organized, processed, stored, and retrieved quickly and effectively. They are a means of handling information, rendering the data for easy use.

**Classification of Data Structures**

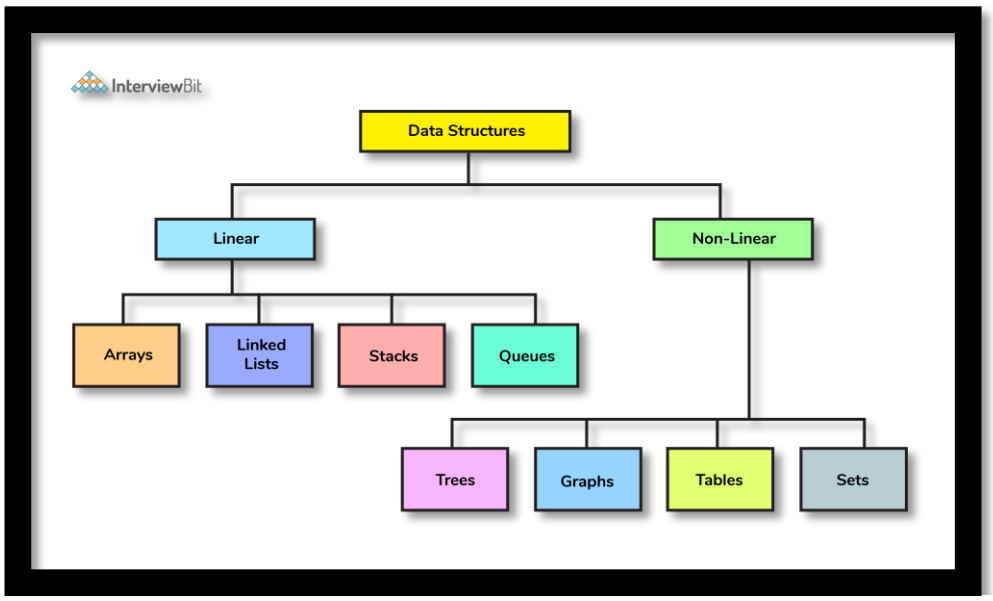



**Figure 2.1: Classification of Data Structures**

## 2.2 Asymptotic Notations and Complexity Analysis :

Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by identifying its behavior as its input size grows.

Asymptotic analysis of an algorithm refers to defining the mathematical foundation or framing of its run-time Performance. Using asymptotic analysis, We can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and maybe for another operation it is computed as g(n^2). This means the first operation's running time will increase linearly with the increase in **n** and the running time of the secondoperationwill increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly Small.

Usually, the time required by an algorithm falls under three types-

- **Best Case** – Minimum time required for Program execution.
- **Average Case** – Average time required for Program execution.

➢ **Worst Case** – Maximum time required for Program execution.

There are mainly three asymptotic notation's :

- Big-O Notation (O-notation)
- Omega Notation (Ω-notation)
- Theta Notation (Θ-notation)

- Big Oh Notation (O-notation)

The Notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the **Worst-case complexity** or the longest amount of time an algorithm can possibly take to complete.

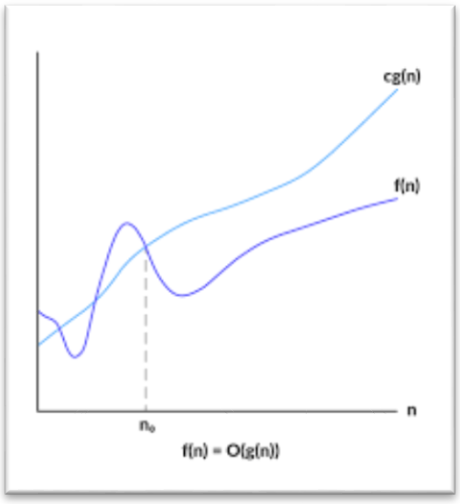



**Figure 2.2.1 : Big Oh Notation Analysis**

- Omega Notation (Ω-notation)

Omega Notation represents the lower bound of the running time of an algorithm.

Thus, it provides the **Best-Case Complexity** of an algorithm.

The execution time serves as a lower bound on the algorithm's time complexity.

It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

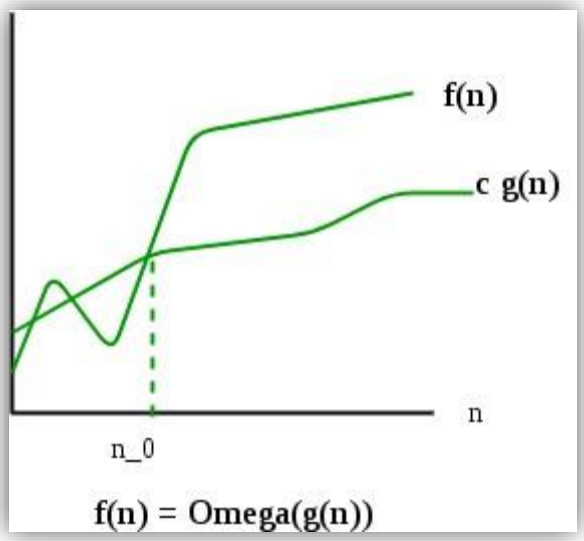



**Figure 2.2.2 : Omega Analysis**

- Theta Notation (Θ-notation)

Theta Notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analysing the **average-case complexity** of an algorithm.

Theta(Average case) You add the running times for each possible input combination and take the average in the average case. The execution time serves as both a lower and upper bound on the algorithm's time complexity. It exists as both, most, and least boundaries for a given input value.

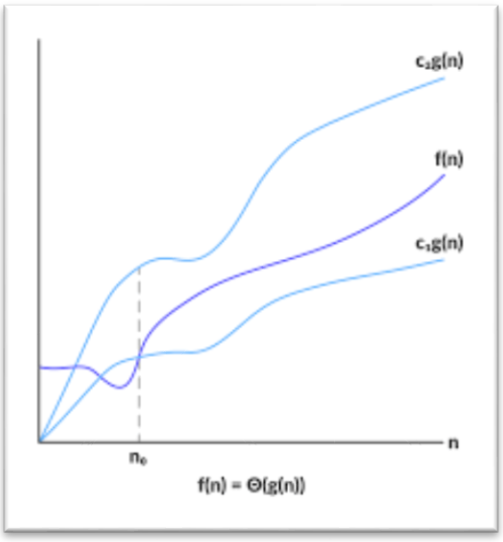
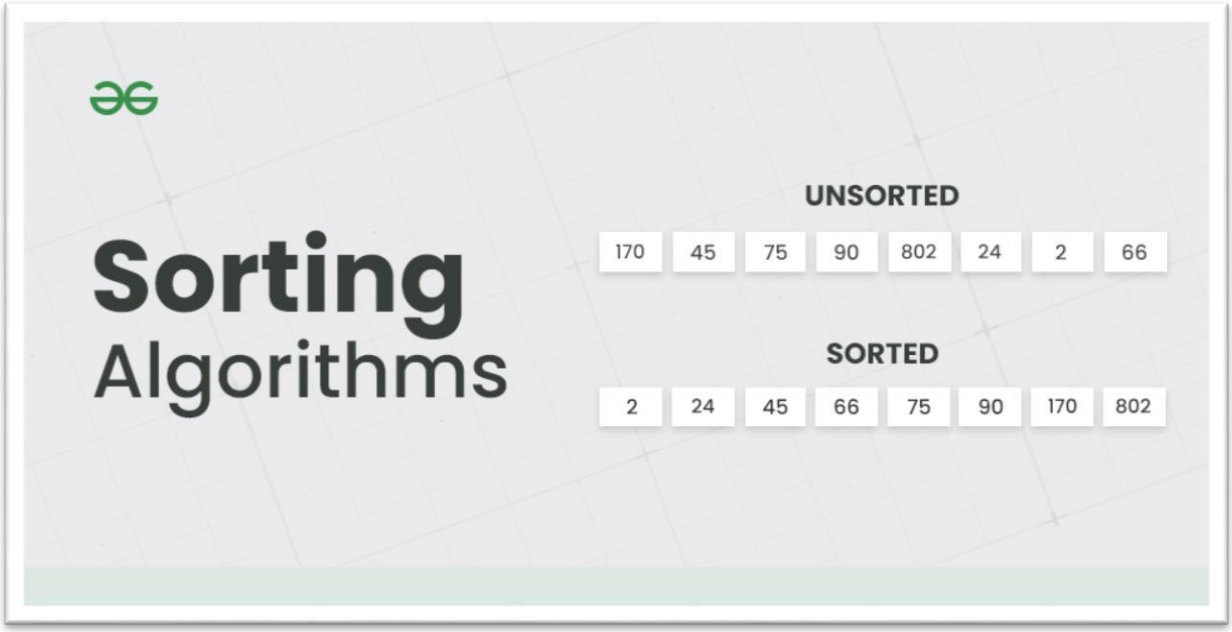



**Figure 2.2.3 : Theta Analysis**

## 2.3 Sorting Algorithms :

**Sorting:** A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

**For Example**, The below list of characters is sorted in increasing order of their ASCII Values. That is, the character with a lesser ASCII value will be placed first than the character with a higher ASCII Valued.




### i. Selection Sort :

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparisonbased algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst-case complexities are of $O(n^2)$, where **n** is the number of items.

**Algorithm:**

Step 1 − Set MIN to location 0.

Step 2 − Search the minimum element in the list.

Step 3 − Swap with value at location MIN Step 4

− Increment MIN to point to next element.

Step 5 − Repeat until list is sorted.

**Pseudocode :**

```
void selectionSort(int[] nums) {
System.out.println("Algorithm chosen : SELECTION SORT\n");
  for(int i =0;i<nums.length-1;i++) {                    int
  min = i;
  for(int j =i+1;j<nums.length;j++) {
        if(nums[j] <nums[min]) {
                    min = j;}
                        }
          swap(nums,i,min);
  System.out.println("PASS "+(i+1)+" : \n\t");
  printArray(nums);
  System.out.println();
                }
  }
```

**ii. Bubble Sort :**

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped

if they are not in order. This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$ where **n** is the number of items.

**Algorithm :**

begin BubbleSort(arr)
for all array elements
if arr[i] > arr[i+1]
swap(arr[i], arr[i+1])
end if end for
return arr
end BubbleSort

**Pseudocode :**

```
void bubbleSort(int[] nums){
  System.out.println("Algorithm chosen : BUBBLE SORT \n");
  for(int i =0;i<nums.length;i++)       {
for(int j =0;j<nums.length-i-1;j++) {
if(nums[j] >nums[j+1])
swap(nums,j,j+1);
}
}
System.out.print("PASSp          \n\t");
rintArray(nums);
System.out.println();
} }
```

```
"+(i
+1)+
" :
```

**iii.Insertion Sort :**

This is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

**Algorithm :**

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted **Pseudocode**

**:**

```
void insertionSort(int[] nums) {
  System.out.println("Algorithm Chosen : INSERTION SORT\n");
for(int i =1;i<nums.length;i++) {
```

```
 int j = i-1;
int val = nums[i];
while(j >= 0 &&nums[j] >val) {
        nums[j+1] = nums[j];
        j--;
                }
        nums[j+1] = val;
System.out.println("PASS "+(i)+" : \n\t");                printArray(nums);
System.out.println(); } }
```

**iv. Quick Sort :**

This is a sorting algorithm based on the **divide and conquer algorithm** that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Working of Quick Sort :

The key process in quicksort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all the greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worstcase complexity are $O(n^2)$, respectively.

**Pivot Algorithm :**

Step 1 − Choose the highest index value has pivot

Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

**Main Algorithm :**

Step 1 − Make the right-most index value pivot

Step 2 − partition the array using pivot value

Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

**Pseudocode :**

```
void quickSort(int[] unmint low,int high) {
if(low < high) {
    int i =low-1,j =low;
 int pivot = nums[high];
for( j= low's<high's++) {
if(nums[j] < pivot) {
i++;
        swap(nums,i,j);
            }}
              int temp = nums[i+1];
              nums[i+1]  = pivot;
              nums[high] = temp;
            System.out.println("PASS "+(quickCount++)+" : \n\t");
    System.out.println("Pivot index : "+pivot+"\n");
```

```
        printArray(nums);
                System.out.println();
                quickSort(nums,low,i);
                quickSort(nums,i+2,high);
    }           }
```

**v. Merge Sort :**

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

**Algorithm :**

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

**Pseudocode :**

```
void mergeSort(int[] nums,int left,int right) {
        if(left < right) {
int mid = (left + right - 1)/2;
        mergeSort(nums,left,mid);
        mergeSort(nums,mid+1,right);
        merge(nums,left,mid,right);
```

```
                    System.out.println("PASS :  "+(mergeCount++)+" :");
                    printArray(nums);
                    System.out.println();
                }
            }
            void merge(int[] nums,int left,int mid,int right) {
                int n1 = mid - left + 1;
        int n2 = right - mid;
 int L[] = new int[n1];
int R[] = new int[n2];
 for (int i = 0; i < n1; ++i) {
            L[i] = nums[left + i];
        }
        for (int j = 0; j < n2; ++j) {
            R[j] = nums[mid + 1 + j];
        }
  int i = 0, j = 0;
int k = left;
 while (i < n1 && j < n2) {
if (L[i] <= R[j]) {
nums[k] = L[i];
              i++;            }
else {
   nums[k] = R[j];
j++;
}            k++;
        }
  while (i < n1) {
nums[k] = L[i];
```

```
            i++;
k++;
        }
 while (j < n2) {
nums[k] = R[j];
            j++;
k++;
        }
            }
```

**vi.Heap Sort :**

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

**Algorithm :**

```
  HeapSort(arr)
  BuildMaxHeap(arr)
  for i = length(arr) to 2
  swap arr[1] with arr[i]
      heap_size[arr] = heap_size[arr] ?
  1
      MaxHeapify(arr,1)
End
```

**Pseudocode :**

```
  void heapSort(int[] nums) {
  int N = nums.length;
        for (int i = N / 2 - 1; i >= 0; i--) {
```

```
            heapify(nums, N, i);
        }
        for (int i = N - 1; i > 0; i--) {
swap(nums,0,i);
            heapify(nums, i, 0);
            System.out.println("PASS "+(heapCount++)+" :");
            printArray(nums);
          System.out.println();
      }
        System.out.println("Sorted Array : \n");
        printArray(nums);
        System.out.println();
}                   void heapify(int[] nums,int N,int i ) {
                      int largest = i;
        int l = 2 * i + 1;
 int r = 2 * i + 2;
if (l < N && nums[l] > nums[largest])
            largest = l;
if (r < N && nums[r] > nums[largest])
            largest = r;
  if (largest != i) {
swap(nums,i,largest);
            heapify(nums, N, largest);
        }
            }
```

**Table 2.1: Comparison of Complexities of different Sorting Algorithms**

| Sorting Algorithm | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Bubble Sort | $n^2$ | $n^2$ | $n^2$ |
| Selection Sort | $n^2$ | $n^2$ | $n^2$ |
| Insertion Sort | n | $n^2$ | $n^2$ |
| Quick Sort | $n \log_2 n$ | $n \log_2 n$ | $n^2$ |
| Merge Sort | $n \log_2 n$ | $n \log_2 n$ | $n \log_2 n$ |
| Heap Sort | n | $n \log_2 n$ | $n \log_2 n$ |

## 2.4 Searching Algorithms :

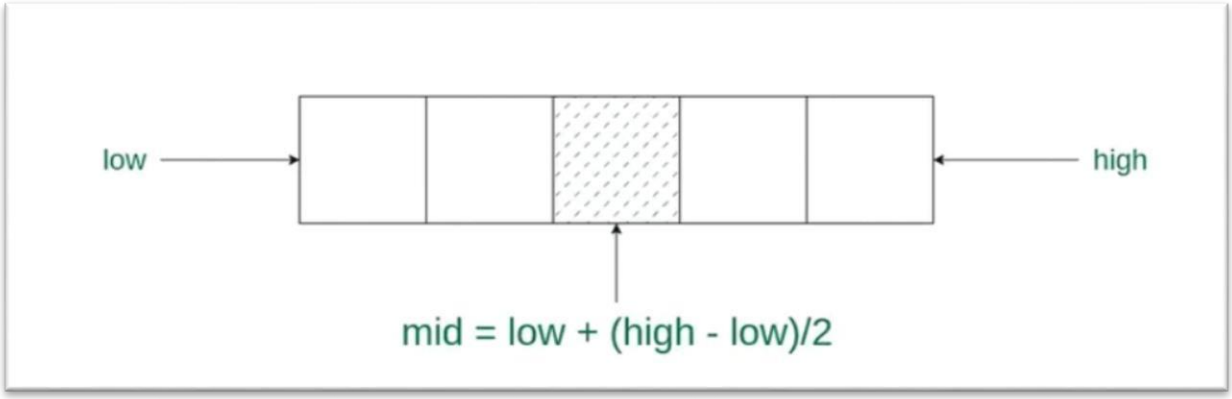
### i. Binary Search :

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Conditions for when to apply Binary Search in a Data Structure:
To apply Binary Search algorithm:

- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

How to Apply Binary Search Algorithm ?




- Divide the search space into two halves by finding the middle index “mid”.

- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
- If the key is smaller than the middle element, then the left side is used for next search.
- If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.

**Algorithm :**

Binary_Search(a, lower_bound6, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search

Step 1: set beg = lower_bound, end = upper_bound, pos = - 1

Step 2: repeat steps 3 and 4 while beg <=end

Step 3: set mid = (beg + end)/2

Step 4: if a[mid] = val

set pos = mid print

pos go to step 6

else if a[mid] > val

set end = mid - 1

else

set beg = mid + 1

[end of if]

[end of loop] Step 5: if pos = -1

print "value is not present in the

array"

[end of if]

Step 6: exit

**Pseudocode :** void searchTarget(int[]

```
 nums,int target) {

                int low = 0,high = nums.length-1;

  int count =1;

 while(low <= high) {

  int mid = (low+high)/2;

  if(nums[mid] == target) {

        System.out.println("Element found at index :\t"+mid);

                return ;

                }

        System.out.println("PASS "+count+++" :\n");

        System.out.println("Mid index          :\t"+mid+"\tMid     Element
 :\t"+nums[mid]+"\n");
if(target >nums[mid]) {

                low = mid+1;

                }

   else {

  high = mid-1;

        }

}

                System.out.println("Target element not found in the given array");

        }
```

**ii. Linear Search :**

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.

How Does Linear Search Algorithm Work?

In Linear Search Algorithm,

- Every element is considered as a potential match for the key and checked for the same.
- If any element is found equal to the key, the search is successful and the index of that element is returned.
- If no element is found equal to the key, the search yields “No match found”.

**Algorithm :**

Linear Search ( Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

**Pseudocode :**

```
procedure linear_search (list, value)
   for each item in the list
      if match item == value
return the item's location
end if
   end for
end procedure
```

**Table 2.2: Comparison of Complexities of different Searching Algorithms**

| Searching Algorithm | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Binary Search | 1 | $\log_2 n$ | $\log_2 n$ |
| Linear Search | 1 | n | n |

## 3. Reason For Choosing This Technology

A lot of beginners and experienced programmers avoid learning Data Structures and Algorithms because it's complicated and they think that there is no use of all the above stuff in real life. So before we discuss the topic we are going to throw a simple problem at you and you need to find the solution for that.

If you need to search your roll number in 20000 pages of PDF document (roll numbers are arranged in increasing order) how would you do that?

If you will try to search it randomly or in a sequential manner it will take too much time. You might get frustrated after some time.

You can try another solution which is given below…

Go to page no. 10000

If your roll no. is not there, but all other roll no. in that page are lesser than your than Go

to page no. 15000

Still if your roll no. is not there. but this time all other roll no. is greater than your.

Go to page no. 12500

Continue the same process and within 30-40 seconds you will find your roll number. **Congratulations… you just have used the Binary Search algorithm unintentionally..**

From the above example, we can straightforward give two reasons to Learn Data Structure and

Algorithms…

- If you want to crack the interviews and get into the product based companies
- If you love to solve real-world complex problems.

From the above example, we can straightforward give two reasons to Learn Data Structure and

Algorithms…

**To Crack the Interviews of the Top Product Based Companies**

Data structures and algorithms play a major role in implementing software and in the hiring process as well. A lot of students and professionals have the question of why these companies' interviews are focused on DSA instead of language/frameworks/tools specific questions? Let us explain why it happens…

When you ask someone to make a decision for something the good one will be able to tell you "I choose to do X because it's better than A, B in these ways. I could have gone with C, but I felt this was a better choice because of this". In our daily life, we always go with that person who can complete the task in a short amount of time with efficiency and using fewer resources. The same things happen with these companies. The problem faced by these companies is much harder and on a much larger scale. Software developers also have to make the right decisions when it comes to solving the problems of these companies.

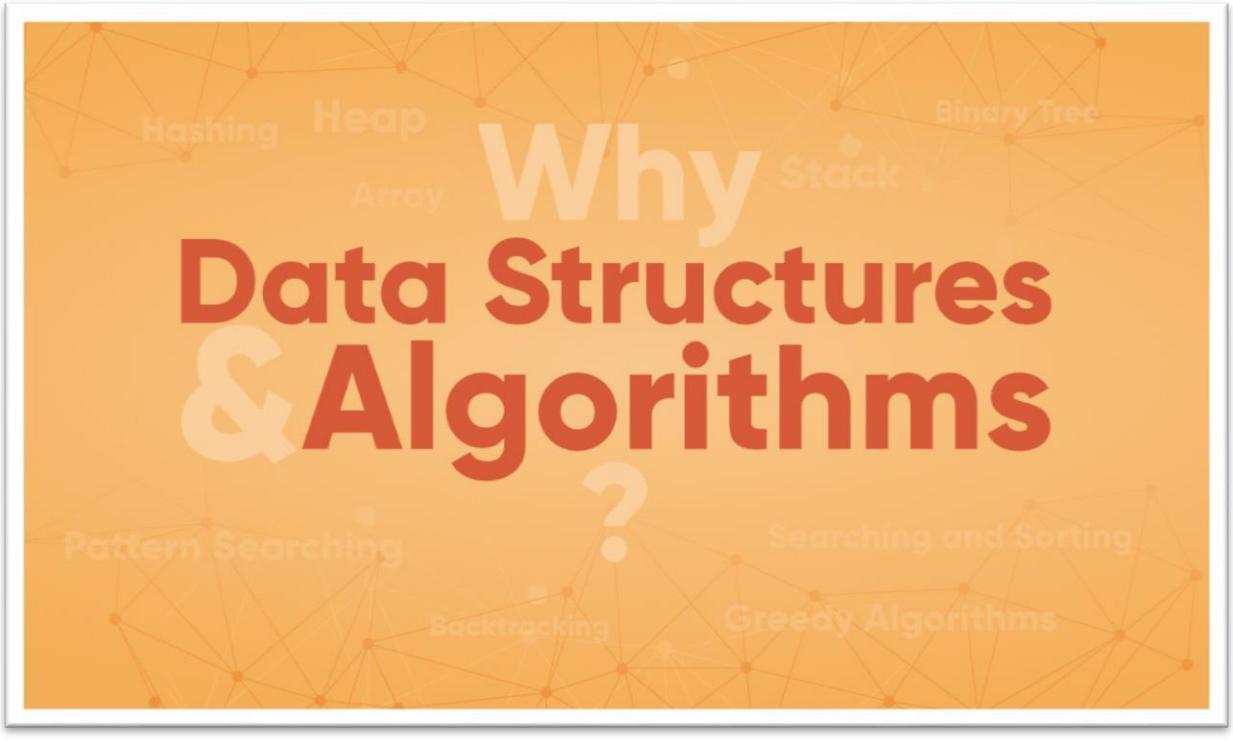


**Figure 3.1 : Why Data Structures ?**

Knowledge of DS and Algo like Hashing, Tree, Graph, and various algorithms goes a long way in solving these problems efficiently and the interviewers are more interested in seeing how candidates use these tools to solve a problem. Just like a car mechanic needs the right tool to fix a car and make it run properly, a programmer needs the right tool (algorithm and data structure) to make the software run properly. So the interviewer wants to find a candidate who can apply the right set of tools to solve the given problem. . If you know the characteristics of one data structure in contrast to another you will be able to make the right decision in choosing the right data structure to solve a problem.

**To Solve Some Real-World Complex Problems**

Let's take the example of a library. If you need to find a book on Set Theory from a library, you will go to the maths section first, then the Set Theory section. If these books are not organized in this manner and just distributed randomly then it will be frustrating to find a specific book. So data structures refer to the way we organize information on our computers. Computer scientists process and look for the best way we can organize the data we have, so it can be better processed based on the input provided.

## 4. Brief on All Data Structures

### 4.1 Array :

**Basic Definition:** An array is a collection of items of same data type stored at contiguous memory locations.It is a static data structure with a fixed size. It combines data of similar types.

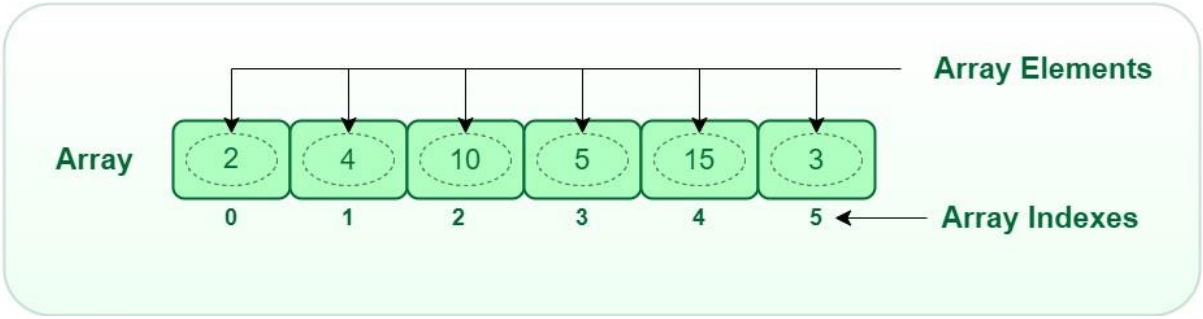



**Figure 4.1 : Representation of an Array**

**Why are Arrays required?**

- Sorting and searching a value in an array is easier.

- Arrays are best to process multiple values quickly and easily.
- **Arrays are good for storing multiple values in a single variable**.

**Properties of Array:**

There are some of the properties of an array that are listed as follows -

- Each element in an array is of the same data type and carries the same size that is 4 bytes.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

**Basic Operations on Array:**

- Traversal - This operation is used to print the elements of the array.
- Insertion - It is used to add an element at a particular index.
- Deletion - It is used to delete an element from a particular index.
- Search - It is used to search an element using the given index or by the value.
- Update - It updates an element at a particular index.

**Applications of Array D.S:**

- Storing and accessing data: Arrays are used to store and retrieve data in a specific order. For example, an array can be used to store the scores of a group of students, or the temperatures recorded by a weather station.
- Sorting: Arrays can be used to sort data in ascending or descending order. Sorting algorithms such as bubble sort, merge sort, and quicksort rely heavily on arrays.
- Searching: Arrays can be searched for specific elements using algorithms such as linear search and binary search.
- Matrices: Arrays are used to represent matrices in mathematical computations such as matrix multiplication, linear algebra, and image processing.
- Stacks and queues: Arrays are used as the underlying data structure for implementing stacks and queues, which are commonly used in algorithms and data structures.

- Graphs: Arrays can be used to represent graphs in computer science. Each element in the array represents a node in the graph, and the relationships between the nodes are represented by the values stored in the array.
- Dynamic programming: Dynamic programming algorithms often use arrays to store intermediate results of subproblems in order to solve a larger problem.

**Advantages of Array D.S:**

- Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

**Disadvantages of Array D.S:**

- Array is homogenous. It means that the elements with similar data type can be stored in it.
- In array, there is static memory allocation that is size of an array cannot be altered.
- There will be wastage of memory if we store less number of elements than the declared size.

## 4.2 Linked List :

**Basic Definition:** A linked list is a collection of "nodes" connected together via links. These nodes consist of the data to be stored and a pointer to the address of the next node within the linked list. In the case of arrays, the size is limited to the definition, but in linked lists, there is no defined size. Any amount of data can be stored in it and can be deleted from it.

There are three types of linked lists −

- Singly Linked List − The nodes only point to the address of the next node in the list.
- Doubly Linked List − The nodes point to the addresses of both previous and next nodes.

- Circular Linked List − The last node in the list will point to the first node in the list. It can either be singly linked or doubly linked.

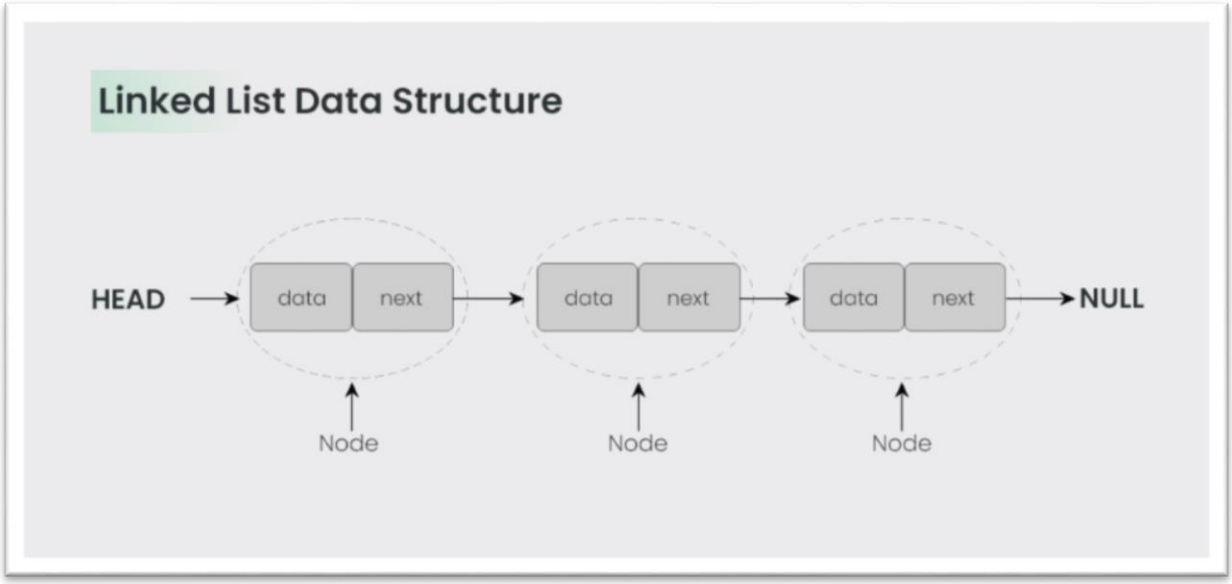



**Figure 4.2.1 : Representation of a Linked List**

**Singly Linked List:**Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.


**Figure 4.2.2 : Representation of a Singly Linked List**

**Doubly Linked List:**Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.

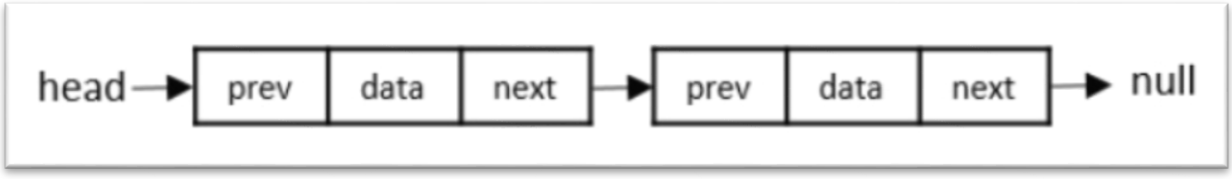



**Figure 4.2.3 : Representation of a Doubly Linked List**

**Circular Linked Lists:**

Circular linked lists can exist in both singly linked list and doubly linked list.

Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.

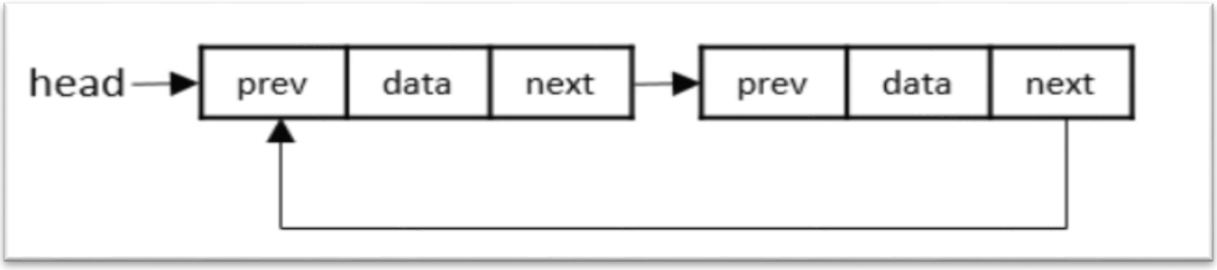



**Figure 4.2.4 : Representation of a Circular Linked List**

**Why use linked list over array?**

**Arrays have some problems:**

- You need to know the size beforehand.
- It's hard to make them bigger once created.
- All elements must be together in memory, making insertions tricky.

**Linked lists solve these issues:**

- They allocate memory dynamically, as needed.
- Size isn't a problem, as they grow as you add items.
- Elements are linked using pointers, so insertions are easier.

**To declare a linked list:**

- Think of it as two parts: the actual data and a pointer.
- Declare a structure with these parts.
- This structure becomes your linked list's "node." ➢ Nodes are linked together to form the list.

**Properties of a Linked List:**

- The linked list starts with a HEAD which denotes the starting point or the memory location of first node.

- Linked list ends with the last node pointing to NULL value.
- Unlike array the elements are not stored in contiguous memory locations, but are stored in random location.
- Random allocation of memory location helps to add any number of elements to the lined list.
- Linked List does not waste memory space.

**Basic Operations on Linked List :**

- Insertion − Adds an element at the beginning of the list.
- Deletion − Deletes an element at the beginning of the list.
- Display − Displays the complete list.
- Search − Searches an element using the given key.
- Delete − Deletes an element using the given key.

**Applications of Linked List :**

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

**Advantages of Linked List:**

- Dynamic data structure - The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.

- Insertion and deletion - Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.
- Memory efficient - The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- Implementation - We can implement both stacks and queues using linked list.

**Disadvantages of Linked List:**

- Memory usage - In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
- Traversal - Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- Reverse traversing - Backtracking or reverse traversing is difficult in a linked list. In a

  doubly-linked list, it is easier but requires more memory to store the back pointer.

## 4.3 Stack :

**Basic Definition:**

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.

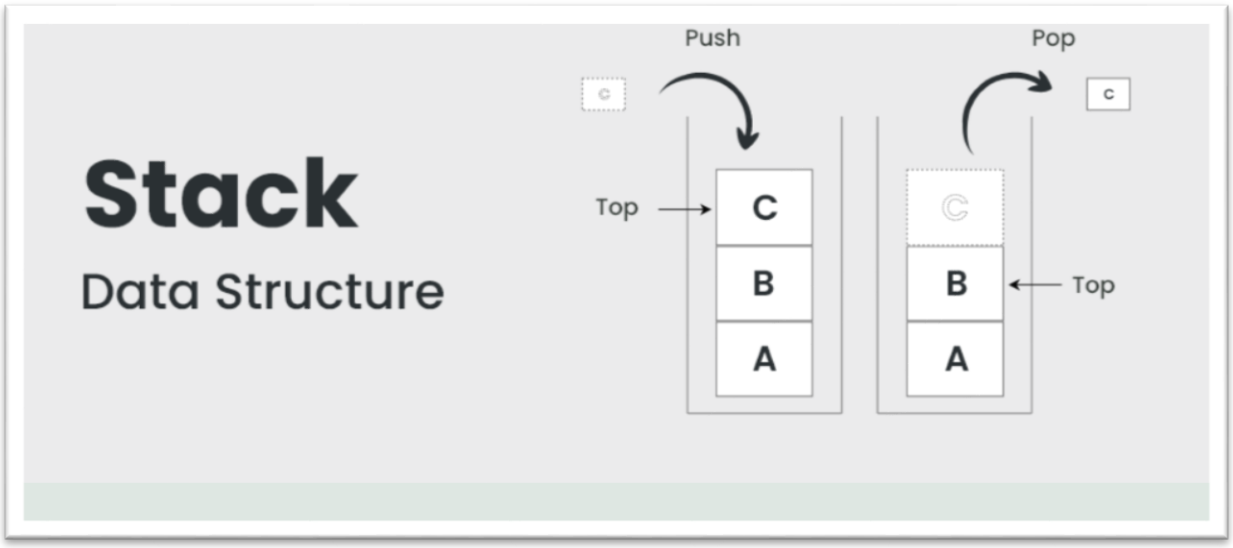



**Figure 4.3 : Representation of Stack**

**Standard Stack Operations:**

The following are some common operations implemented on the stack:

- push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- isEmpty(): It determines whether the stack is empty or not.
- isFull(): It determines whether the stack is full or not.
- peek(): It returns the element at the given position.
- count(): It returns the total number of elements available in a stack.
- change(): It changes the element at the given position.
- display(): It prints all the elements available in the stack.

**Application of the Stack:**

- A Stack can be used for evaluating expressions consisting of operands and operators.
- Stacks can be used for Backtracking, i.e., to check parenthesis matching in an expression.
- It can also be used to convert one form of expression to another form.

- It can be used for systematic Memory Management.

**Advantages of Stack:**

- A Stack helps to manage the data in the 'Last in First out' method.
- When the variable is not used outside the function in any program, the Stack can be used.
- It allows you to control and handle memory allocation and deallocation.
- It helps to automatically clean up the objects.

**Disadvantages of Stack:**

- It is difficult in Stack to create many objects as it increases the risk of the Stack overflow.
- It has very limited memory.
- In Stack, random access is not possible.

## 4.4 Queue:

**Basic Definition:** A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

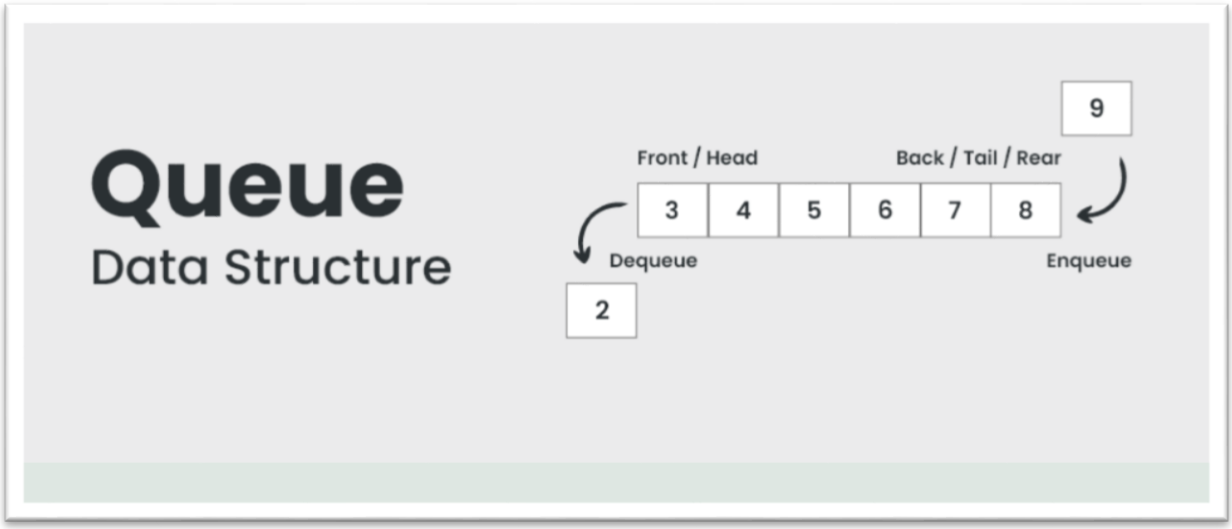



**Figure 4.4.1 : Representation of Queue**

**Basic Operations on Queue:**

- enqueue() Process of adding or storing an element to the end of the queue
- dequeue() Process of removing or accessing an element from the front of the queue

- peek() Used to get the element at the front of the queue without removing it
- initialize() Creates an empty queue
- isfull() Checks if the queue is full
- isempty() Check if the queue is empty

**Types of Queue D.S:**

**Simple Queue:**

It is the most basic queue in which the insertion of an item is done at the front of the queue and deletion takes place at the end of the queue.

- Ordered collection of comparable data kinds.
- Queue structure is FIFO (First in, First Out).
- When a new element is added, all elements added before the new element must be deleted in order to remove the new element.


**Figure 4.4.2 : Representation of Simple Queue**

**Circular Queue:**

A circular queue is an extended version of a linear queue as it follows the First In First Out principle with the exception that it connects the last node of a queue to its first by forming a circular link. Hence, it is also called a Ring Buffer.

- The last node is connected to the first node.
- Also known as a Ring Buffer as the nodes are connected end to end.
- Insertion takes place at the front of the queue and deletion at the end of the queue.
- Circular queue application: Insertion of days in a week.

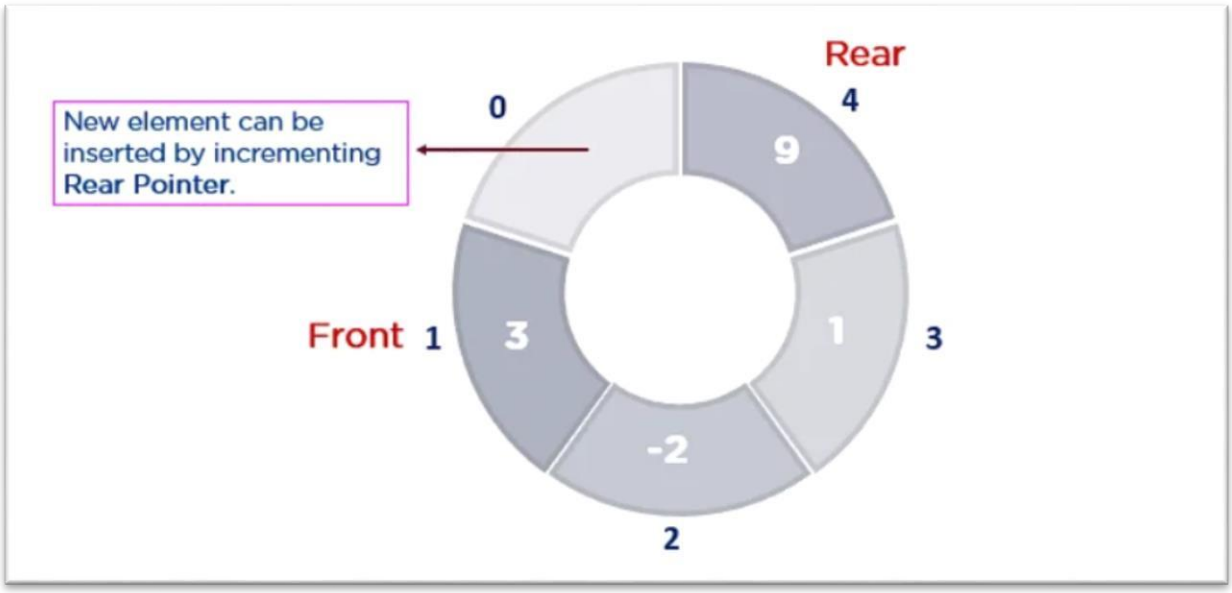



**Figure 4.4.3 : Representation of Circular Queue**

**Priority Queue:**

In a priority queue, the nodes will have some predefined priority in the priority queue. The node with the least priority will be the first to be removed from the queue. Insertion takes place in the order of arrival of the nodes.

Some of the applications of priority queue:

- Dijkstra's shortest path algorithm
- Prim's algorithm
- Data compression techniques like Huffman code

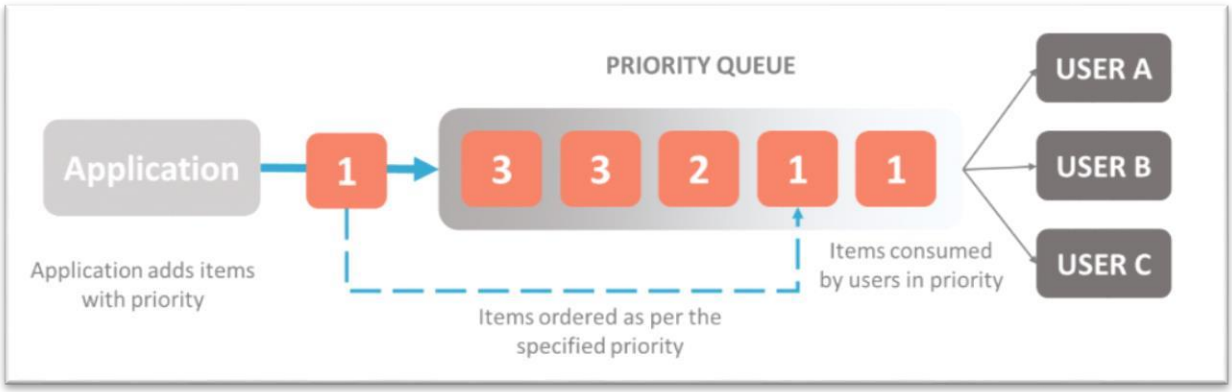



**Figure 4.4.4 : Representation of Priority Queue**

**Deque (Double Ended Queue):**

In a double-ended queue, insertion and deletion can take place at both the front and rear ends of the queue.

**Operations on Deque:**

- push_front() : Inserts the element at the beginning.
- push_back() : Adds element at the end.
- pop_front() : Removes the first element from the deque.
- pop_back() : Removes the last element from the deque.
- front() : Gets the front element from the deque.
- back() : Gets the last element from the deque.
- empty() : Checks whether the deque is empty or not.
- size() : Determines the number of elements in the deque.


**Figure 4.4.5 : Representation of Deque**

## 4.5 Trees :

A tree is a non-linear abstract data type with a hierarchy-based structure. It consists of nodes (where the data is stored) that are connected via links. The tree data structure stems from a single node called a root node and has subtrees connected to the root.

**Important Terms :**

Following are the important terms with respect to tree.

- Path − Path refers to the sequence of nodes along the edges of a tree.
- Root − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- Parent − Any node except the root node has one edge upward to a node called parent.
- Child − The node below a given node connected by its edge downward is called its child node.

- Leaf − The node which does not have any child node is called the leaf node.
- Subtree − Subtree represents the descendants of a node.
- Visiting − Visiting refers to checking the value of a node when control is on the node.
- Traversing − Traversing means passing through nodes in a specific order.
- Levels − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- Keys − Key represents a value of a node based on which a search operation is to be carried out for a node.

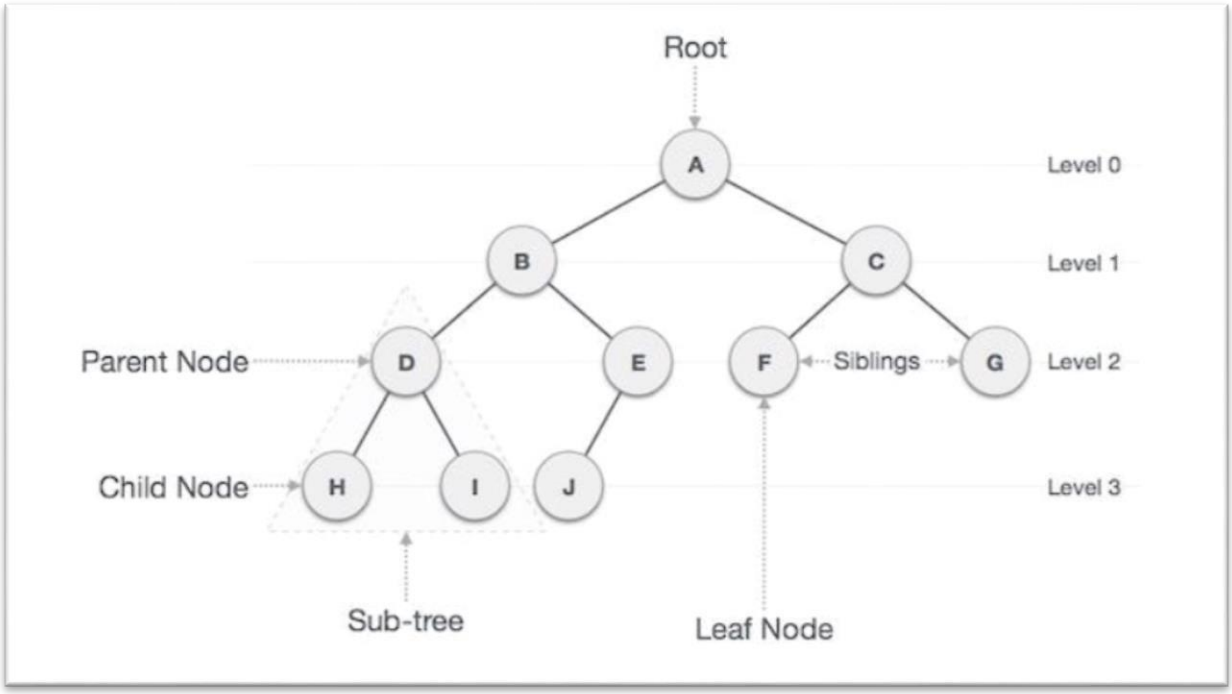



**Figure 4.5.1 : Representation of a Tree**

**Types of Tree DS:**

- **General Trees :**

General trees are unordered tree data structures where the root node has minimum 0 or maximum ‘n’ subtrees.

The General trees have no constraint placed on their hierarchy. The root node thus acts like the superset of all the other subtrees.

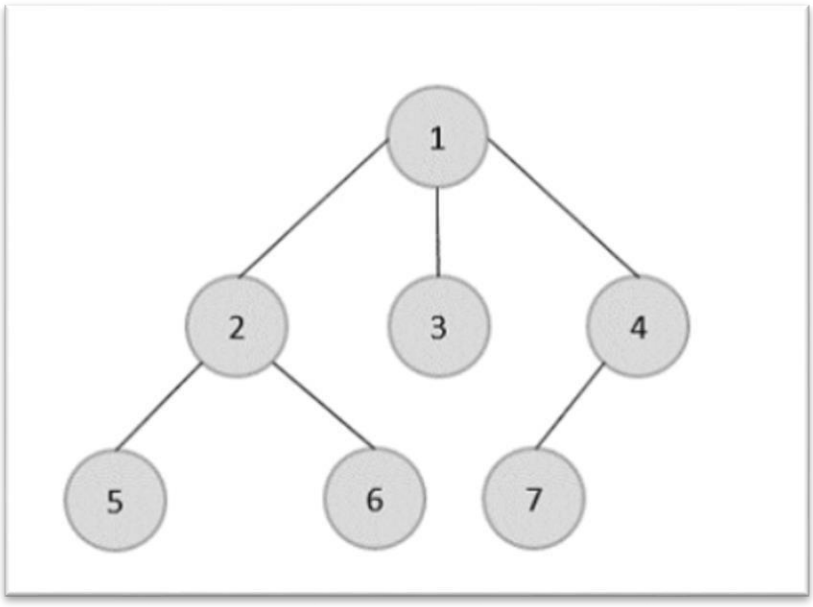



**Figure 4.5.2 : Structure of a General Tree**

➢ **Binary Trees:**

Binary Trees are general trees in which the root node can only hold up to maximum 2 subtrees: left subtree and right subtree. Based on the number of children, binary trees are divided into three types.

Full Binary Tree :

A full binary tree is a binary tree type where every node has either 0 or 2 child nodes.

Complete Binary Tree :

A complete binary tree is a binary tree type where all the leaf nodes must be on the same level.

However, root and internal nodes in a complete binary tree can either have 0, 1 or 2 child nodes. Perfect Binary Tree :

A perfect binary tree is a binary tree type where all the leaf nodes are on the same level and every node except leaf nodes have 2 children.

➢ **Binary Search Trees :**

Binary Search Trees possess all the properties of Binary Trees including some extra properties of their own, based on some constraints, making them more efficient than binary trees.

The data in the Binary Search Trees (BST) is always stored in such a way that the values in the left subtree are always less than the values in the root node and the values in the right subtree are always greater than the values in the root node, i.e. left subtree < root node ≤ right subtree.

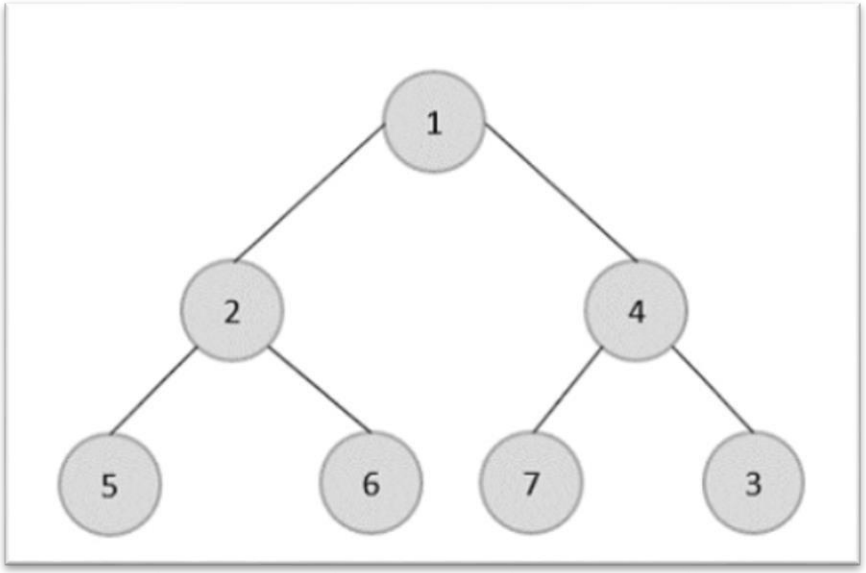



**Figure 4.5.3 : Representation of BT and BST**

- **AVL Tree :**

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the binary tree as well as of the binary search tree. It is a self-balancing binary search tree that was invented by Adelson Velsky Lindas. Here, selfbalancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the **balancing factor**.

We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the **difference between the height of the left subtree and the height of the right subtree**. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.

**Applications of Tree D.S :**

- File System: This allows for efficient navigation and organization of files.
- Data Compression: Huffman coding is a popular technique for data compression that involves constructing a binary tree where the leaves represent characters and their frequency of occurrence. The resulting tree is used to encode the data in a way that minimizes the amount of storage required.
- Compiler Design: In compiler design, a syntax tree is used to represent the structure of a program.
- Database Indexing: B-trees and other tree structures are used in database indexing to efficiently search for and retrieve data.

**Advantages of Tree D.S:**

- Tree offer Efficient Searching Depending on the type of tree, with average search times of O(log n) for balanced trees like AVL.
- Trees provide a hierarchical representation of data, making it easy to organize and navigate large amounts of information.
- The recursive nature of trees makes them easy to traverse and manipulate using recursive algorithms.

**Disadvantages of Tree D.S:**

- Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to inefficient search times.
- Trees demand more memory space requirements than some other data structures like arrays and linked lists, especially if the tree is very large.
- The implementation and manipulation of trees can be complex and require a good understanding of the algorithms.

- **Heap:**

A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

**Operations of Heap Data Structure:**

- Heapify: a process of creating a heap from an array.
- Insertion: process to insert an element in existing heap time complexity O(log N).
- Deletion: deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
- Peek: to check or find the first (or can say the top) element of the heap.

**Types of Heap Data Structure:**

Generally Heaps can be of two types:

- Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
- Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

## 4.6 Graphs :

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices ( V ) and a set of edges ( E ). The graph is denoted by G(E, V).

**Why are Graphs Used ?**

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

**Representations of a Graph :**

Here are the two most common ways to represent a graph :

- Adjacency Matrix
- Adjacency List

Adjacency Matrix:

An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's).

Let's assume there are n vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

If there is an edge from vertex i to j, mark adjMat[i][j] as **1**.
If there is no edge from vertex i to j, mark adjMat[i][j] as **0**.

Adjacency List :

An array of Lists is used to store edges between two vertices. The size of array is equal to the number of vertices (i.e, n). Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex i.

Let's assume there are n vertices in the graph So, create an array of list of size n as adjList[n].

adjList[0] will have all the nodes which are connected (neighbour) to **vertex 0**. adjList[1]

will have all the nodes which are connected (neighbour) to **vertex 1** and so on.

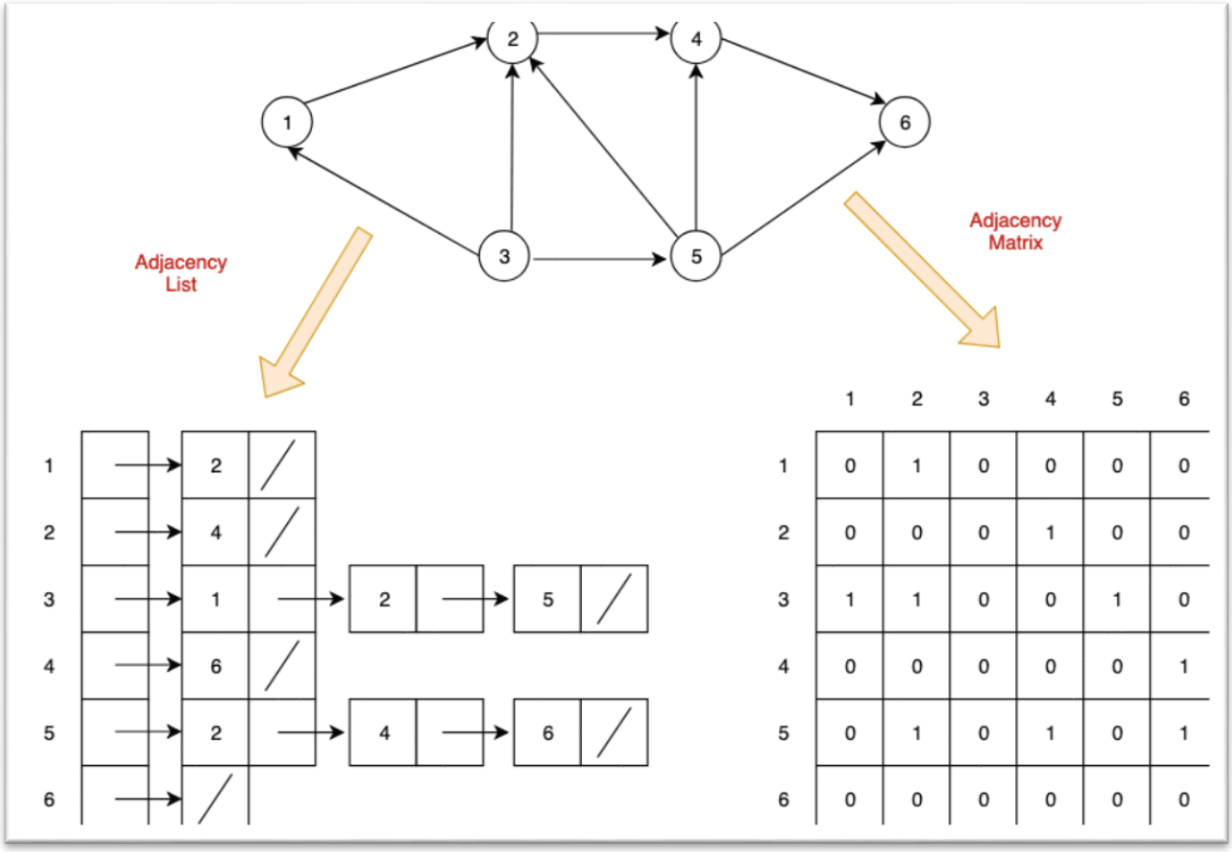



**Figure 4.6.1 : Representation of Adjacency List and Adjacency Matrix**

Graphs are basically represented in Two Ways :

- Directed Graph
- Undirected Graph

**Operations on Graphs :**

There are two types of traversals in Graphs −

- Depth First Search Traversal:

Depth First Search is a traversal algorithm that visits all the vertices of a graph in the decreasing order of its depth. In this algorithm, an arbitrary node is chosen as the starting point and the graph is traversed back and forth by marking unvisited adjacent nodes until all the vertices are marked.

The DFS traversal uses the stack data structure to keep track of the unvisited nodes.

- Breadth First Search Traversal:

Breadth First Search is a traversal algorithm that visits all the vertices of a graph present at one level of the depth before moving to the next level of depth. In this algorithm, an arbitrary node is chosen as the starting point and the graph is traversed by visiting the adjacent vertices on the same depth level and marking them until there is no vertex left.

The DFS traversal uses the queue data structure to keep track of the unvisited nodes.

## 5. Problem Analysis

Sorting any given input in Ascending/Descending Order and displaying all iterations of the chosen algorithm.

### 5.1 Product Definition :

The "Algorithm Simulator" is a software tool designed to provide a visual representation and step-by-step simulation of various sorting algorithms along with additional data structure operations. The main goal of the project is to assist learners and developers in understanding the inner workings of sorting algorithms and related data structures through interactive simulations.

The software allows users to input a list of elements and choose between different sorting algorithms like Selection Sort, Bubble Sort, and Insertion Sort. Additionally, it supports Binary Search and provides demonstrations of Linked List, Stack, and Queue operations. Users can observe the iteration-by-iteration progression of the chosen algorithm, helping them comprehend the sorting process and data manipulation operations.

### 5.2 Feasibility Analysis :

Feasibility analysis was conducted to assess the practicality and viability of the Algorithm Simulator project. This analysis included technical, economic, and operational aspects.

Technical Feasibility: The project's technical feasibility was established by evaluating the availability of required technology, tools, and expertise to develop the software. Since the project involves software development and simulation techniques, it was deemed technically feasible.

Economic Feasibility: The economic feasibility considered the costs associated with software development, such as development tools, resources, and potential revenue from the software's utilization. Given the educational and practical value of the software, it was deemed economically feasible.

Operational Feasibility: Operational feasibility assessed whether the software would align with users' needs and whether the end product would be usable and maintainable. The project's focus on educational purposes and its intuitive user interface contributed to its operational feasibility.

### 5.3 Project Plan :

The project plan outlined the various phases of development, resource allocation, and timelines. It included requirements gathering, system design, implementation, testing, and documentation.

- Requirements Gathering: Understanding the desired functionalities, sorting algorithms, and data structure operations to be included in the software.
- System Design: Creating a comprehensive system design that outlines the software's architecture, user interface, and simulation algorithms.
- Implementation: Developing the software based on the design specifications, incorporating sorting algorithm simulations and data structure operations.
- Testing: Conducting functional and structural testing to ensure the software's correctness, performance, and robustness.
- Documentation: Creating detailed documentation, including user guides and technical manuals, to aid users in understanding and utilizing the software effectively.
- Deployment: Releasing the software for user access and feedback.
- Maintenance: Providing ongoing support and updates to address any issues and improve the software based on user feedback.

The project plan ensured a structured approach to the development process, facilitating efficient and effective completion of the Algorithm Simulator.

## 6. Software Requirement Analysis

### 6.1 Introduction:

The Software Requirement Analysis section outlines the high-level objectives and scope of the Algorithm Simulator project. It provides an overview of the software's purpose and

features, including interactive sorting algorithm simulations and data structure operation demonstrations.

### 6.2 General Description:

The General Description delves deeper into the software's functionalities, detailing the supported sorting algorithms (Selection Sort, Bubble Sort, Insertion Sort), data structure operations (Linked List, Stack, Queue), and their respective objectives. It also highlights the interactive nature of the simulations, allowing users to input their data and witness step-by-step algorithmic progress.

### 6.3 Specific Requirements:

This section outlines the specific technical requirements of the Algorithm Simulator. It includes details about the supported platforms (Windows, macOS, Linux), programming languages (Python, Java), and any additional libraries or frameworks required for the software's development. Additionally, it specifies the hardware and software constraints necessary for optimal performance.

## 7. Design

### 7.1 System Design:

The System Design phase focuses on creating a blueprint of the software's architecture and user interface. It includes diagrams illustrating the flow of the program, the interactions between different modules, and the user's journey through the simulations. This section elaborates on how the sorting algorithm simulations are visualized, how user inputs are processed, and how the data structure operations are showcased.

### 7.2 Design Notations:

Design notations, such as flowcharts and UML diagrams, are used to represent the software's structure visually. Flowcharts depict the step-by-step logic of the sorting algorithms, while UML diagrams illustrate the relationships between classes, methods, and user interactions.

### 7.3 Circuit Diagram:

As the Algorithm Simulator is a software application, it doesn't involve physical circuitry or electrical components. Thus, a circuit diagram may not be relevant for this project.

## 8. Testing

### 8.1 Functional Testing:

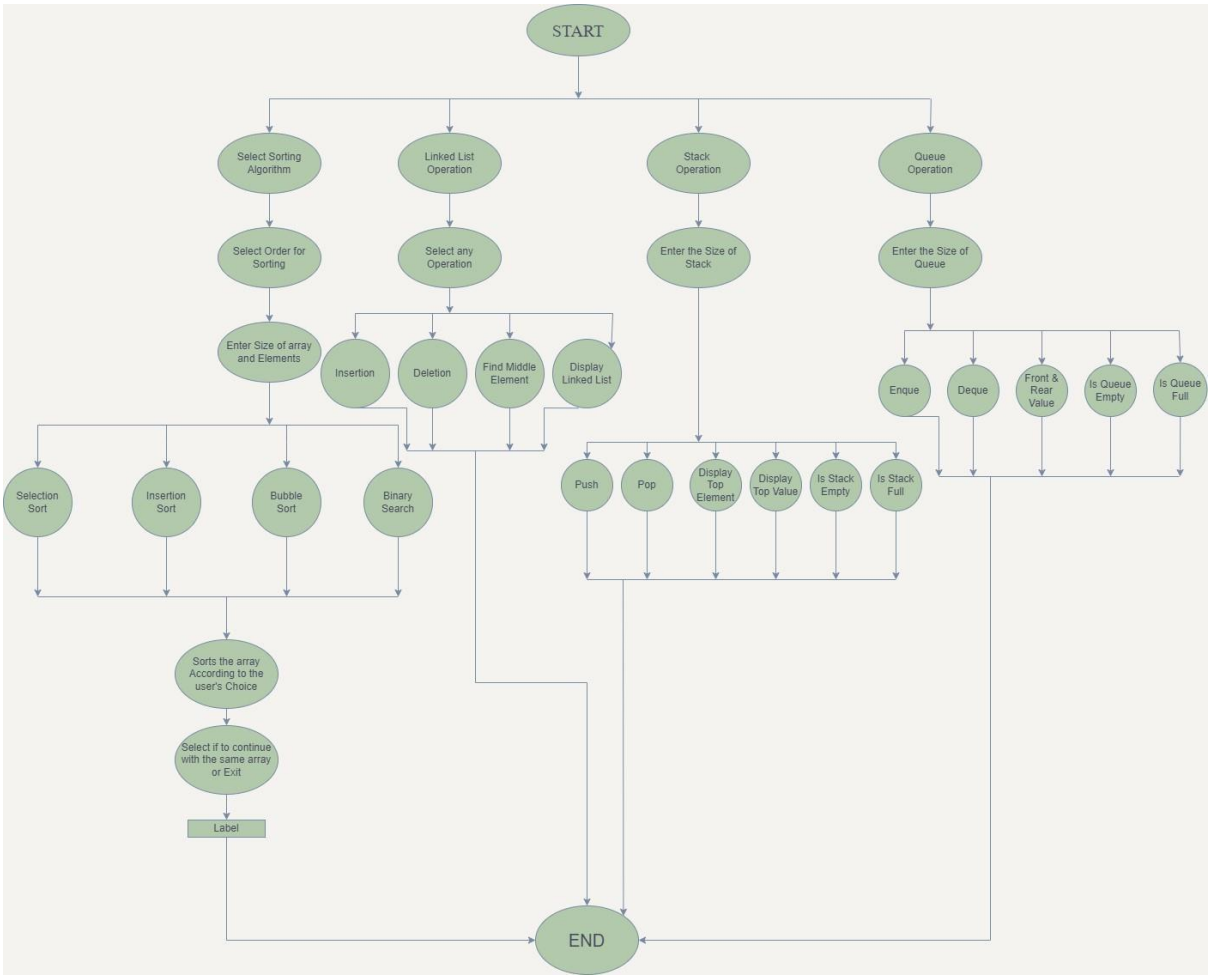
Functional testing involves evaluating the software's individual functions to ensure they perform as intended. This section outlines the testing scenarios for each sorting algorithm and data structure operation, verifying that inputs are processed correctly and outputs match the expected results.

**8.2 Structural Testing:**

Structural testing assesses the interactions between different components and modules within the software. It ensures that the algorithms and operations integrate seamlessly and that data is passed between them accurately.

**8.3 Flow Charts:**

Flowcharts are essential tools for visualizing the logical flow of algorithms and operations. In this section, flowcharts for each sorting algorithm and data structure operation are provided, depicting the step-by-step processes and decision points.




# 9. Implementation

## 9.1 Implementation of Project:

The Implementation phase involves translating the design specifications into actual code. This section briefly discusses the programming languages used (Python, Java), the libraries or frameworks integrated for graphical representation, and the implementation of the sorting algorithms and data structure operations.

## 9.2 Conversion Plan:

As the Algorithm Simulator primarily focuses on software development, a conversion plan related to physical components or processes may not be applicable.

### 9.3 Post-Implementation and Software Maintenance:

Post-implementation activities involve releasing the software, gathering user feedback, and making necessary updates or bug fixes. This section outlines the strategies for addressing user feedback, ensuring the software's stability, and planning for future enhancements.

## 10. Project Legacy

### 10.1 Current Status of Project:

This section provides an overview of the current state of the Algorithm Simulator project, highlighting its completion and readiness for user access.

### 10.2 Remaining Areas of Concern:

Any potential areas of improvement or unresolved issues are discussed in this section. It could include future feature enhancements, performance optimizations, or user experience improvements.

### 10.3 Technical and Managerial Lessons Learned:

Reflecting on the development process, this part discusses the technical skills gained, challenges overcome, and managerial insights acquired during the project's lifecycle.

## 11. User Manual:

A Complete Document of the Software Developed

The User Manual provides comprehensive guidance on how to use the Algorithm Simulator. It includes step-by-step instructions on launching the software, inputting data, selecting algorithms, and interpreting the visual simulations. The manual also explains the purpose of each sorting algorithm and data structure operation, ensuring users understand the educational value of the tool.

By following this User Manual, users can make the most of the software's interactive simulations and gain a deeper understanding of sorting algorithms and data structures. It enhances the software's usability and educational impact.

Please note that the length of each section may vary based on the depth of detail and explanation you wish to provide. You can further elaborate on each aspect to achieve the desired page count while ensuring the content remains informative and engaging.

## 12. Source Code or System

### 12.1 Selection Sort:

```
for(int i=0;i<n-1;i++) cout<<"\n\n\t\t* * * PASS
"<<i+1<<" * * *\t\t\n";
        {
for(int i=0;i<n;i++)
 int min=i;          {
       cout<<"\n\nMinumum small selected element is:\t"<<s_arr[min]<<"\n\n";
cout<<"\t"<<s_arr[i];
for(int j=i+1;j<n;j++)
                  }
                  {
                  cout<<"\n\t- - - - - - - - - - - - - - - - - -\n\n";
                        if(s_arr[j]<s_arr[min])
         }
                  {

min=j;

cout<<"\n\nNext       updated       small   element

is:\t"<<s_arr[min]<<"\n\n";

                        }

       }

if(min!=i)   {

       swap(s_arr[min],s_arr[i]);

                  }
```

### 12.2 Insertion Sort :

```
for(int i=1;i<n;i++)

{
for(int
i=1;i<n;i++) {
```

```
int temp=i_arr[i]; cout<<"\n\nElement selected is "<<temp<<"\n\n"; int temp=i_arr[i]; int j=i-1;
cout<<"\n\nElement selected is "<<temp<<"\n\n";
while(j>=0&&i_arr[j]>temp) int j=i-1;
{
while(j>=0&&i_arr[j]>temp)          i_arr[j+1]=i_arr[j];
{
        j--;
        i_arr[j+1]=i_arr[j];
}
        j--;
i_arr[j+1]=temp; }

i_arr[j+1]=temp;
cout<<"\n\t\t* * * PASS"<<i<<" * **\t\t\n";
for(int i=0;i<n;i++) {
cout<<"\n\t\t* * * PASS"<<i<<" * * *\t\t\n";

for(int
i=0;i<n;i++){cout<<"\t"<<i_arr[i];

}
cout<<"\t"<<i_arr[i];
cout<<"\n\t- - - - - - - - - - - - - - - - - \n\n";
}
cout<<"\n\t- - - - - - - - - - - - - - - - - \n\n";
```

```
    }

    }
```

**12.3 Bubble Sort:**

```
for(int i=0;i<n-1;i++) {

  cout<<"\n\n\t\t*   *   *   PASS   "<<i+1<<"   *   *   *\t\t\n";

  for(int j=0;j<n-1-i;j++)
```

```
                {
            if(b_arr[j]>b_arr[j+1])
            {                                   int
temp=b_arr[j+1];
        b_arr[j+1]=b_arr[j];
                b_arr[j]=temp;
                        }
                    for(int i=0;i<n;i++)
                        {
                            cout<<"\t"<<b_arr[i];
        }
        cout<<"\n\n";
                }
        cout<<"\n\t- - - - - - - - - - - - - - - - - -\n\n";          }
```

**12.4 Binary Search :**

```
int loc=-1;
 int       beg=0,end=n-1;
int mid;

        while(beg<=end)
        {
```

```
        mid=(beg+end)/2;
        cout<<"\n\nMiddle element is "<<bs_arr[mid]<<" found at index "<<mid<<"\n";
if(item==bs_arr[mid])
        {
loc=mid;
break;
        }
        else if(item<bs_arr[mid])
        {               end=mid-1;
cout<<"\n\nupdated end index is: "<<end;
        }
else
        {
           beg=mid+1;    cout<<"\n\nupdated beginning index is:
"<<beg<<"\n";
        }
      }

      if(loc==-1)
      {
         cout<<"\n\t* * * Element "<<item<<" is not found in the array * * *\n\n";
      }
else
      {
        cout<<"\n\t* * * Element "<<item<<" is found at index "<<loc<<" * * *\n\n";
      }
```

## 12.5 Linked List Operation :

```
3   struct node{
4       int data;
5       node* next;
6   };
7   node* head = NULL;
8   node* last = NULL;
9   int choice = 0, ch =0;
10  void insert_at_beg(int data){
11      node *newNode = new node();
12      newNode->data = data;
13      newNode->next = NULL;
14      if(head == NULL){
15          head = newNode;
16          last = newNode;
17      }
18      else{
19          newNode->next = head;
20          head = newNode;
21      }
22  }
23
24  void insert_at_end(int data){
25      node *newNode = new node();
26      newNode->data = data;
27      newNode->next = NULL;
28      if(head == NULL){
29          head = last = newNode;
30      }
31      else{
32          last->next = newNode;
33          last = newNode;
34      }
35  }
36
37  void insert_at_pos(int data,int pos){
38      node *newNode = new node();
39      newNode->data = data;
40      newNode->next = NULL;
41      node *temp = head;
42      while(temp != NULL && pos-- > 1){
43          temp = temp->next;
44      }
45      if(temp == NULL){
```

```
        if(head == NULL){
            head = last = newNode;
        }
        else{
            last->next = newNode;
            last = newNode;
        }
    }
    else{
        newNode->next = temp->next;
        temp->next = newNode;
    }
}

void delete_at_beg(){
    if(head == NULL){
        cout << "Empty List\n";
    }
    else{
        head = head->next;
    }
}
void delete_at_end(){
    node *temp = head;
    while(temp != NULL && temp->next != last){
        temp = temp->next;
    }
    temp->next = NULL;
    last = temp;
}
void delete_at_pos(int pos){
    if(head == NULL){
        cout << "Empty List\n";
        return ;
    }
    node *temp = head,*temp1 = NULL;
    while(temp != NULL && temp->next != NULL && pos-- > 1){
        temp1 = temp;
        temp = temp->next;
    }
    if(temp1 == NULL){
        head = head->next;
    }
    else{
        temp1->next = temp->next;
```

```
91          }
92      }
93      void display_middle(){
94          node *slow = head,*fast = head;
95          while(fast != NULL && fast->next != NULL){
96              slow = slow->next;
97              fast = fast->next->next;
98          }
99          if(slow == NULL){
100             cout << "Empty List\n";
101         }
102         else{
103             cout << "Middle element " << slow->data << "\n";
104         }
105     }
106
107     void display_list(){
108         node *temp = head;
109         cout << "Linked List \n";
110         while(temp != NULL){
111             cout << temp->data << "  ";
112             temp = temp->next;
113         }
114     }
115
```

## 12.6 Stack Operation:

```
155 void stack_operation(){
156     int top = -1;
157     int size;
158     cout << "Enter the size of the stack";
159     cin >> size;
160     int stack[size];
161     cout << "1.PUSH\n2.POP\n3.DISPLAY TOP ELEMENT\n4.DISPLAY TOP VALUE\n5.Is STACK EMPTY\n6.Is STACK FULL\n0.EXIT";
162     int choice;
163     do{
164         cout << "\nSelect any of the choices \n";
165         cin >> choice;
166         switch(choice){
167             case 1 : if(top < size-1){
168                         int data;
169                         cout << "Enter the data \t";
170                         cin >> data;
171                         stack[++top] = data;
172 //                      cout << "top\t"<<top<<"\n";
173                     }
174                     else{
175                         cout << "OVERFLOW !!!\n";
176                     }break;
177             case 2 : if(top != -1){
178                         cout << "Popped element \t" << stack[top--];
179
180                     }
181                     else{
182                         cout << "UNDERFLOW !!!\n";
183                     }break;
```

```
184                 case 3 : if(top == -1){
185                             cout << "Empty Stack";
186                         }
187                         else{
188                             cout << "Top element \t" << stack[top];
189                         }break;
190                 case 4 : cout << "\n Top value  \t" << top;break;
191                 case 5 : if(top == -1)
192                             cout << "\nYES";
193                         else
194                             cout << "\nNO";
195                         break;
196                 case 6 : if(top == size-1)
197                             cout << "\nYES";
198                             else
199                                 cout << "\nNO";
200                             break;
201                 default : cout << "Invalid selection\n\n";break;
202                 case 0 : return;
203             }
204         }while(choice != 0);
205     }
```

## 12.7 Queue Operation :

```
207 void queue_operation(){
208 
209     int front = -1,rear = -1;
210     cout << "Enter the size of the Queue\n";
211     int size;;
212     cin >> size;
213     int queue[size];
214     cout << "1.ENQUEUE\n2.DEQUEUE\n3.FRONT VALUE and REAR VALUE \n4.Is QUEUE EMPTY\n5.Is Queue FULL\n0.exit";
215     do{
216         int ch= 1;
217         cout << "\nSelect any of the choices \n";
218         cin >> ch;
219         switch(ch){
220             case 1 : if(rear== size-1){
221                         cout << "OVERFLOW !!\n";
222                     }
223                     else{
224                         int data;
225                         cout << "Enter data\t";
226                         cin >> data;
227                         if(front == -1){
228                             front = rear = 0;
229                         }
230                         else{
231                             rear++;
232                         }
233                         queue[rear] = data;
234                     }break;
235             case 2 : if(front == -1 || front > rear){
236                         cout << "UNDERFLOW !! \n";
237                     }
238                     else{
239                         cout << "Dequeued element \t" << queue[front++]<<"\n";
240                     }break;
```

```
241                 case 4 : if(front == -1)
242                             cout << "YES";
243                         else
244                         cout << "NO";
245                         break;
246                 case 5 : if(front ==0 && rear == size-1)
247                             cout << "YES";
248                         else
249                             cout << "NO";
250                         break;
251                 case 3:cout << "Front value\t" << front;
252                        cout << "\nRear Value\t"<< rear;break;
253                 case 0:return ;
254                 default : cout << "Invalid Selection\n";
255             }
256         }while(true);
257     }
```

## 13. References


1. https://www.geeksforgeeks.org/

2. https://www.programiz.com/

3. https://www.tutorialspoint.com/

4. https://www.javatpoint.com/

5. https://www.interviewbit.com/

6. https://www.simplilearn.com/