Data Structures and Algorithms
Mid-Term Report

Dhruvraj Merchant

June, 2024

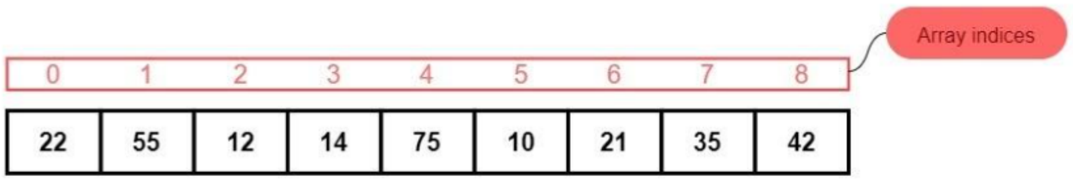
Contents

1 What is Data Stucture and Algorithm?

Data Structures and Algorithms (DSA) are all about how we organize and store data, as well as designing step-by-step procedures (algorithms) to solve problems using these data structures. Basically, DSA is what helps us manage information efficiently and find effective solutions to different computational challenges.

2 Arrays:

An array is a data structure that sequentially stores an element of the same data type. In C/C++ or any other programming language, an array is a collection of similar data items. The data items are always stored in an array at contiguous memory locations. The element of the array can be accessed randomly using the indices of an array.
All arrays consist of contiguous memory locations. The lowest address corresponds to the first element of the array and the highest address to the last element of the array.



- Size Of Array - 9
- First element of array - 22
- Last element of array - 42

2.1 Types of array declaration in C++

- Declaration of an array by specifying the size
- Declaration of an array by initialising elements
- Declaration of an array by specifying the size and initialising elements

2.2 Advantages of an array

- An array can store various data items of the same data type using a single name.
- Traversal of the array becomes easy using a single loop.
- You can randomly access an element in an array by using the index number.
- Sorting of the element is very easy in an array.
- Array allocates memory in the contiguous memory location, so there is no wastage of memory in the array.

2.3 Disadvantages of an array

- The number of elements you want to store in an array should be known in advance.
- Insertion and deletion in the array are very difficult because the elements in the array are stored in consecutive order.
- An array is a static data structure so that you can not increase the size once the array is declared.
- Allocating more memory than the requirement leads to the wastage of memory, and less allocation of memory also leads to a problem

3 Sorting Algorithms:

There are different sorting algorithms; according to the size of the array, we should wisely choose the algorithm. Some of the important sorting algorithms are as follows.

- Selection Sort
- Bubble Sort
- Quick Sort
- Insertion Sort
- Merge Sort

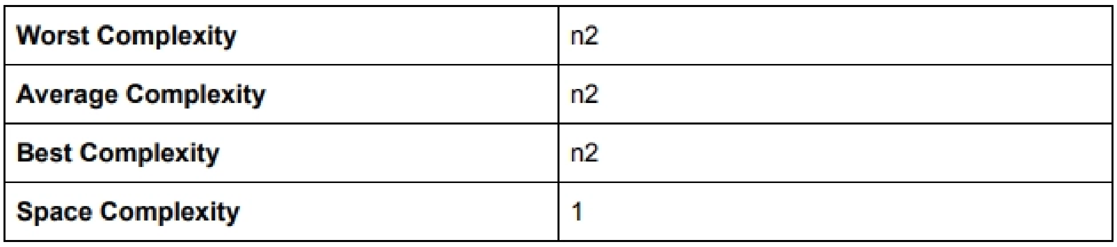
3.1 Selection Sort:

Selection sort repeatedly finds the minimum element from an unsorted array and puts it at the beginning of the array. It is an in-place comparison-based sorting algorithm.
Two arrays are maintained in case of selection sort:

- The unsorted array
- Sorted array

Initially, the sorted array is an empty array and an unsorted array has all the elements. It generally follows the approach of selecting the smallest element from an unsorted array, and that smallest element is placed at the leftmost, which becomes the part of the sorted array finally.
Steps of Selection Sort:

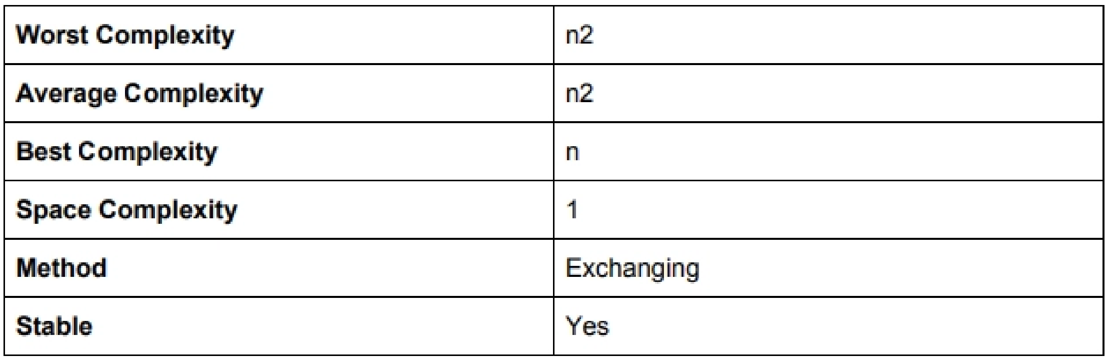
- The first step is to iterate the complete array
- When we reach the end we will get to know the sorted element in the list
- Iterate that sorted element with the leftmost element in the unsorted list
- Now that leftmost element will be part of the sorted array and will not be included in the unsorted array in the next iteration
- Steps will be repeated until all the elements are not sorted



Worst Complexity	n2
Average Complexity	n2
Best Complexity	n2
Space Complexity	1

3.2 Bubble Sort:

In bubble sort, if the adjacent elements are in the wrong order, they are swapped continuously until the correct order is achieved. The Disadvantage of using bubble sort is that it is quite slow
For Example: Consider an unordered list [4, 6, 2, 1].
Pass 1
 $4 < 6$: no change [4, 6, 2, 1]
 Now move next $6 > 2$: swap the elements [4, 2, 6, 1]
 Now $6 > 1$: swap the elements [4, 2, 1, 6]
Pass 2
 4> 2 : swap the elements [2, 4, 1, 6]
 4>1 : swap the elements [2, 1, 4, 6]
 $4 < 6$: no change is needed [2, 1, 4, 6]
Pass 3
 $2 > 1$: swap the elements [1, 2, 4, 6]
 $2 < 4$: no change is needed [1, 2, 4, 6]
 $4 < 6$: no change is needed [1, 2, 4, 6]
Pass 4
 $1 < 2$: no change is needed [1, 2, 4, 6]
 $2 < 4$: no change is needed [1, 2, 4, 6]
 $4 < 6$: no change is needed [1, 2, 4, 6]



Worst Complexity	n2
Average Complexity	n2
Best Complexity	n
Space Complexity	1
Method	Exchanging
Stable	Yes

3.3 Insertion Sort:

It is simple and easy to implement, but it does not have an outstanding performance though. It works on the principle of a sorted item with one item at a time. Basically in each iteration of this sorting, an item is taken from the array, and it is inserted at its correct position by comparing the element from its neighbor. The process is repeated until there is no more unsorted item on the list.
For Example: Consider a list of items as 7, 4, 5, 2

Step 1: There is no element on the left side of 7 so leave the element as it is.
Step 2: Now, 7>4, so swap it. So the new list would be 4, 7, 5, 2.
Step 3: Now 7>5, so swap it as well. So the new list would be 4, 5, 7, 2.
Step 4: As 7>2, so swap it. The new list would be 2, 4, 5, and 7.

Worst Complexity	n2
Average Complexity	n2
Best Complexity	n
Space Complexity	1
Method	Insertion
Stable	Yes

3.4 Quick Sort:

It is a commonly used sorting algorithm. It follows the approach of divide and conquers and follows the following approach.

Takes two empty arrays in which,
a) First array stores the elements that are smaller than the pivot element.
b) Second array stores the elements that are larger than the pivot element.

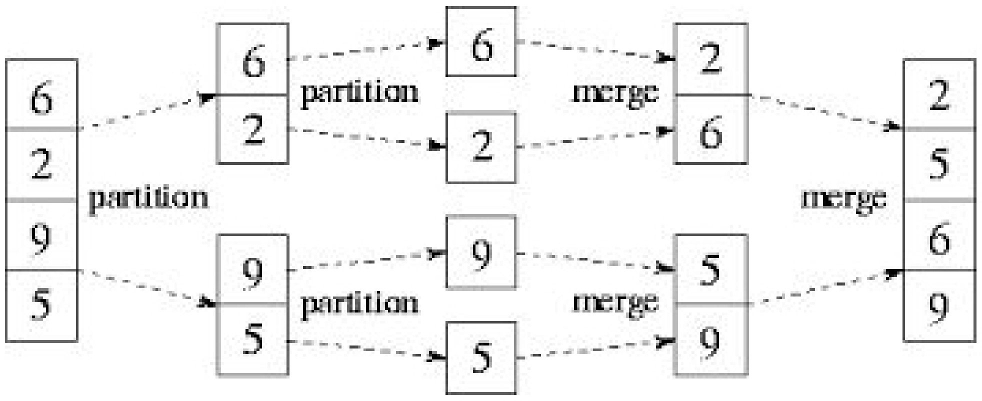
Partitioning the array and swapping them in place.
Steps of Quick Sort:
a)The basis of comparison would be an element that is a "pivot" element in this case.
b)Take two pointers, start one pointer from the left and the other pointer from the right.
c)When we have less value than the pivot element in the left pointer of the array, move it to the right by 1.
d)When we have a larger value than the pivot element in the right pointer of the array, move it to left by 1.
e)When we have a left pointer less than a right pointer, swap the values at these locations in the array.
f)Move the left pointer to the right pointer by 1 and the right to the left by 1.
g)If the left and right pointer does not meet, repeat the steps from 1.

Worst Complexity	n2
Average Complexity	nlogn
Best Complexity	nlogn
Space Complexity	nlogn
Method	Partitioning
Stable	No

3.5 Merge Sort:

It is a sorting algorithm that follows the divide-and-conquer methodology. It recursively breaks down a problem into two or more sub-problems. This recursion is continued until a solution is not found that can be solved easily. Now, these sub-problems are combined together to form the array.

Steps of Merge Sorting:
If the array contains only one element return from the array.
Now divide the complete array into two equal halves, divide until it can not be divided further.
Merge the smaller list into a new list in sorted order



4 Searching Algorithms:

There are mainly two searching algorithm:
1)Linear Search
2)Binary Search

4.1 Linear Search:

In this type of search we traverse the whole array and matches every element of the array with the number we want to search. Below is the code for it.

```
int LinearSearch(int arr[],int element,int size)
{
    for(int i=0;i<size;i++)
    {
        if(arr[i]==element){
            return i;
            break;
        }
    }
}
```

4.2 Binary Search:

The primary condition of this algorithm is, it requires a sorted array. In this algorithm, we basically find the middle element of the array and compare it with the number we want to search. If the middle element is smaller we check in the right side of the array else we search in the left side.

```
int BinarySearch(int arr[],int element,int size)
{
    int low=0,high=size;
    if(arr[low]==element){
        return low;
    }
    else if(arr[high]==element)
    {
        return high;
    }
    else{
        while(arr[low]!=element){
        int mid=(low+high)/2;
        if(arr[mid]==element){
            return mid;
        }
        else if(arr[mid]>element)
        {
            high=mid;
        }
        else{
            low=mid;
        }
        return -1;}

    }

}
```

5 Strings:

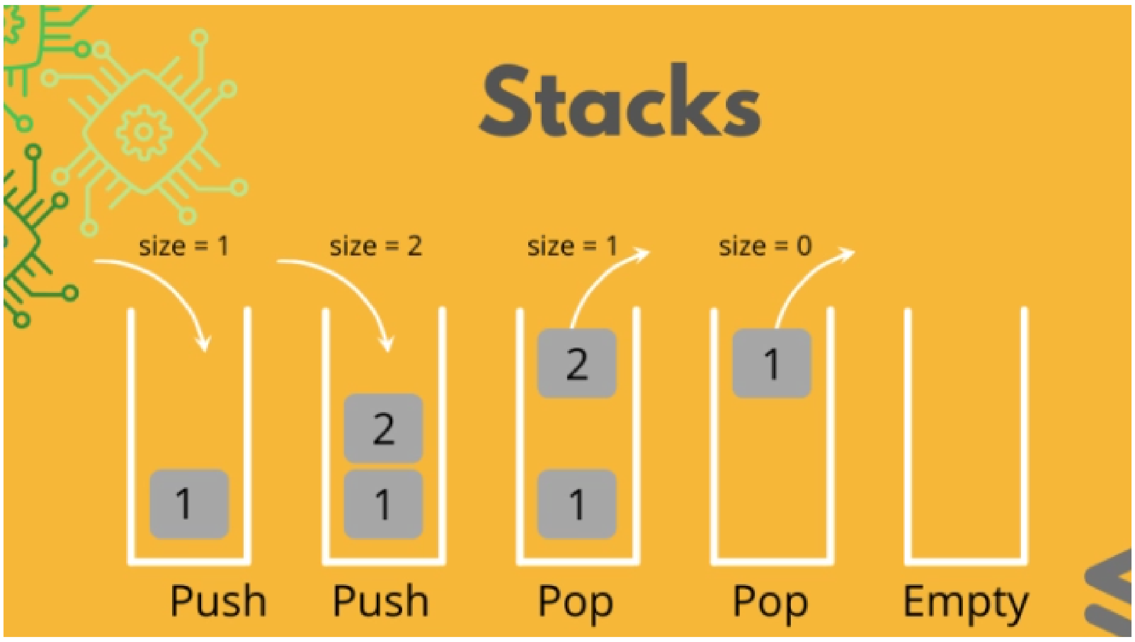
String is basically a char array with some more functionality like we can change the size of the string whenever we want unlike the char arrays which helps in efficient space utilisation.

Some of the functions related to string.
a)length(): used to return the value of the length of the string defined.
b)Push_back(): It pushes a specific character in a string to the end.
c)getline(): We can take input of the strings using the function. It takes in and stores all the input streams until a new line comes in the way.

6 Stacks:

Stacks can be visualized as containers in which we can store elements and remove them as well but only in the LIFO(Last in first out) order. Various functions of stacks are implemented in an array below.

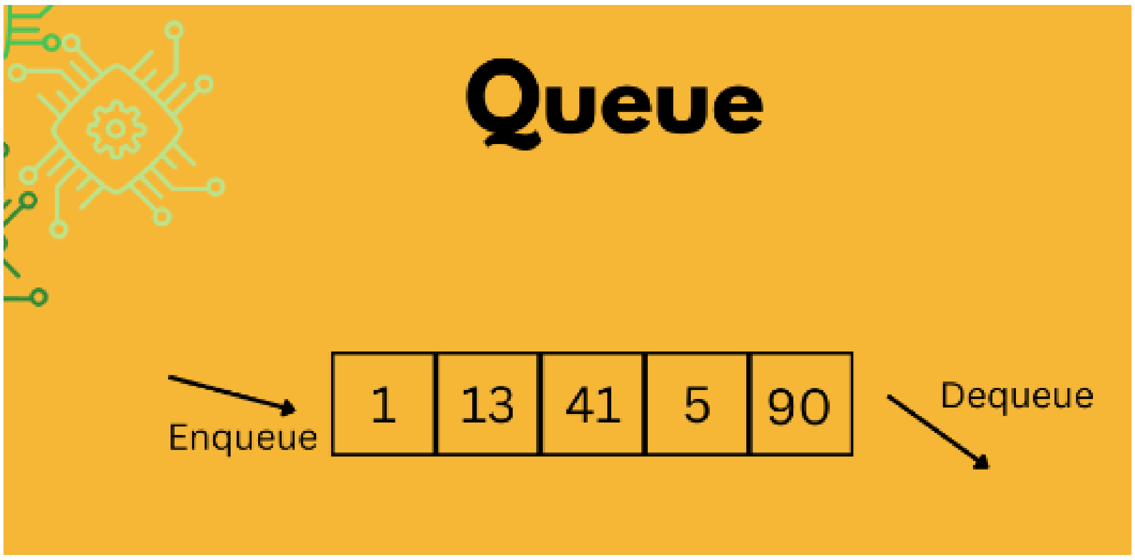
```
    class Stack{
    public:
    int *arr;
    int size;
    int top=-1;
    //contructor
    Stack(int size)
    {
        this->size=size;
        arr=new int[this->size];
    }
    void push(int data)
    {
        if(top!=size)
        {arr[++top]=data;}
        else cout<<"Stack Overflow"<<endl;
    }
    void pop()
    {
        if(top==-1) cout<<"Stack Underflow"<<endl;
        else
        {
            top--;
        }
    }
    void peek()
    {
        cout<<arr[top]<<endl;
    }
    bool empty()
    {
    if(top==-1) return true;
    else return false;
    }
};
```



7 Queue:

A Queue is a linear data structure, which is simply a collection of entries that are tracked in order, such that the addition of entries happens at one end of the queue, while the removal of entries takes place from the other end. It follows FIFO(First In First Out) rule. Below is the implementation of queue using arrays.

```
    class queue1
{
    int *arr;
    int front;
    int rear;
    int size;
    public:
    queue1(int size)
    {
        this->size=size;
        front=rear=0;
        arr=new int[this->size];
    }

    void push(int data)
    {
        if(rear==size) cout<<"Queue Full"<<endl;
        else
        {
            arr[rear++]=data;
        }
    }
    int pop()
    {
        if(front==rear) return -1;
        else
        {
            int ans=arr[front];
            front++;
            if(front==rear)
            {
                front=rear=0;
            }
            return ans;
        }
    }
    bool isEmpty()
    {
        if(rear==front) return true;
        else return false;
    }

};
```



There are other types of queues also
a)Circulary linked Queue:
b)Doubly ended Queue:

7.1 Circularly Linked Queue:

In this type of queue the last element of the queue is linked with the first one.
Here is the code snippet for it.

```
class queue1
{
    int *arr;
    int size;
    int front;
    int rear;
    public:
    queue1(int size)
    {
        this->size=size;
        front=rear=-1;
        arr=new int[this->size];
    }
    void push(int data)
    {
        if(front==0 && rear==size-1 && (front!=-1&&rear==front-1)) //queue full
        { cout<<"Queue Full"<<endl;}
        else if(front==-1) //first elmt to push
        {
            front=rear=0;
        }
        else if(rear==size-1&&front!=0) //rear circular continue me aage elmts dalo
        {
            rear=0;
        }
        else //normal case
        {
            rear++;
        }
        arr[rear]=data;
    }
    int pop()
    {
        if(front==-1) return -1;
        int ans=arr[front];
        if(front==rear) //single element
        {
            front=rear=-1;
        }
        else if(front==size-1)
        {
            front=0;
        }
        else //normal case
        {
            front++;
        }
        return ans;
    }

    bool isFull()
    {
        if((front==0&&rear==size-1)||(rear==front-1))
        cout<<"IS full"<<endl;
    }

};
```

7.2 Doubly ended Queue:

This type of queue is just like the normal queue but with some additional features like you can pop the back element and you can add an element to the front of the queue. Here is the code snippet for it.

```
    class Deque
{
public:
    // Initialize your data structure.
    int *arr;
    int front;
    int rear;
    int size;
    Deque(int n)
    {
        // Write your code here.
        this->size=n;
        front=rear=-1;
        arr=new int[n];

    }

    // Pushes 'X' in the front of the deque. Returns true if it gets pushed into
    the deque, and false otherwise.
    bool pushFront(int x)
    {
        // Write your code here.
        if(isFull()) return false;

        if(front==-1) front=rear=0;
        else if(front==0) front=size-1;
        else front--;
        arr[front]=x;
        return true;
    }

    // Pushes 'X' in the back of the deque. Returns true if it gets pushed into the
     deque, and false otherwise.
    bool pushRear(int x)
    {
        // Write your code here.
        if(isFull()) return false;

        if(rear==-1) front=rear=0;
        else if(rear==size-1) rear=0;
        else rear++;
        arr[rear]=x;
        return true;
    }

    // Pops an element from the front of the deque. Returns -1 if the deque is
    empty, otherwise returns the popped element.
    int popFront()
    {
        // Write your code here.
        if(isEmpty()) return -1;

        int ans=arr[front];
        if(front==rear) front=rear=-1;
        else if(front==size-1) front=0;
        else front++;
        return ans;
    }

    // Pops an element from the back of the deque. Returns -1 if the deque is empty
    , otherwise returns the popped element.
    int popRear()
    {
        // Write your code here.
        if(isEmpty()) return -1;

        int ans=arr[rear];
        if(front==rear) front=rear=-1;
        else if(rear==0) rear=size-1;
        else rear--;
        return ans;
    }

    // Returns the first element of the deque. If the deque is empty, it returns
    -1.
    int getFront()
    {
        // Write your code here
        if(front!=-1)
        return arr[front];
        else return -1;
    }

    // Returns the last element of the deque. If the deque is empty, it returns -1.
    int getRear()
    {
        // Write your code here.
        if(rear!=-1)
        return arr[rear];
        else return -1;
    }

    // Returns true if the deque is empty. Otherwise returns false.
    bool isEmpty()
    {
        // Write your code here.
        if(front==-1) return true;
        else return false;
    }

    // Returns true if the deque is full. Otherwise returns false.
    bool isFull()
    {
        // Write your code here.
        if(front==0 && rear==size-1 || (front!=-1 && rear==front-1))
        return true;
        else return false;
    }
};
```

8 Linked List:

There are majorly three type of linked list.

- Singly Linked List
- Circular Linked List
- Doubly Linked List

8.1 Singly Linked List:

A Singly Linked List contains a node that has both the data part and pointer to the next node. The last node of the Singly Linked List has a pointer to null to represent the end of the Linked List. Traversal to previous nodes is not possible in singly Linked List i.e We can not traverse in a backward direction.

```
    class Node
{
public:
    int data;
    Node *nxt;

    Node()
    {

    }
    Node(int data)
    {
        this->data=data;
    }
};
void traverse(Node* &p)
{
    Node* ptr=p;
    while (ptr != NULL)
    {
        // cout<<"Hi"<<endl;
        cout << "Element--->" << ptr->data << endl;
        ptr = ptr->nxt;
    }
    cout<<"-------------------------------"<<endl;

}




void insertAtTail(Node* &p,int data)
{

    Node *ptr=new Node();
    ptr->data=data;
    ptr->nxt=NULL;
    Node *tail=p;
    while(tail->nxt!=NULL) tail=tail->nxt;
    // cout<<tail->data<<endl;
    tail->nxt=ptr;

}
void insertAtMiddle(Node *&p,int data,Node*q) //since reference & is used no
    requiremen to return Node*
{
    Node *temp=p;
    Node* new1=new Node;
    new1->data=data;
    new1->nxt=q->nxt;
    q->nxt=new1;

}

void deleteNode(Node* &p,Node *q) //delete by address
{
    Node* temp=p;
    while(temp->nxt!=q) temp=temp->nxt;
    temp->nxt=q->nxt;
    delete q;
}


void deleteNode1(Node* &p,int data) //delete by value
{
    Node* temp=p;

    while(temp->nxt->data!=data) temp=temp->nxt;
    Node *q=temp->nxt;
    temp->nxt=temp->nxt->nxt;
    delete q;
}
void deleteFirstNode(Node* &p)
{
    Node *tmp=p;
    p=tmp->nxt;
    delete tmp;

}
void deleteLastNode(Node* &p)
{
    Node* tmp=p;
    while(tmp->nxt->nxt!=NULL) tmp=tmp->nxt;
    Node *q=tmp->nxt;
    tmp->nxt=NULL;
    delete q;
}
//how to copy a linked list part-1;
Node* llcopy(Node* head)

{
    if(head==NULL) return head;
    Node* tmp=head;
    Node* curr=new Node(tmp->data);
    Node* prev=curr;
    Node* ans=curr;
    tmp=tmp->nxt;
    delete curr;
    while(tmp!=NULL)
    {
        curr= new Node(tmp->data);
        prev->nxt=curr;
        prev=curr;
        tmp=tmp->nxt;

    }
    prev->nxt=NULL; //crucial
    return ans;
}

//how to copy a linked list part-2 using recursion
Node* llcopy1(Node *head)
{
    if(head==NULL) return head;
    Node *curr=head;
    curr->nxt=llcopy1(head->nxt);
    return curr;
}
void insertAtHead(Node* &p, int data)
{
    Node *temp = new Node();
    temp->data = data;
    temp->nxt = p;
    p = temp;

```

```

}
Node* reverse(Node* head)
{
    Node* curr=head;
    Node* prev=NULL;
     while(curr!=NULL )
        {
            Node* f=curr->nxt;
            curr->nxt=prev;
            prev=curr;
            curr=f;
        }
        return prev;
}
```

8.2 Circular Linked List:

Circular Linked List is similar to singly Linked List but the last node of Circular Linked List has a pointer to node which points to the first node (head node) of Linked List.

```
    class Node
{
    public:
    int data;
    Node *nxt;

    Node( int data)
    {
        this->data=data;
    }
};
void traverse(Node* &head)
{
    Node* tmp=head;
    while(tmp->nxt!=head)
    {
        cout<<tmp->data<<endl;
        tmp=tmp->nxt;
    }
    cout<<tmp->data<<endl;
    cout<<"--------------------------------------------------------"<<endl;

}
void insertFirst(Node* &head,int data)
{
    Node *new1=new Node(data);
    Node* tmp=head;
    while(tmp->nxt!=head)
    {
        tmp=tmp->nxt;
    }
    new1->nxt=head;
    tmp->nxt=new1;
    head=new1;
}

void insertAtLast(Node* &head,int data)
{
    Node *new1=new Node(data);
    Node* tmp=head;
    while(tmp->nxt!=head)
    {
        tmp=tmp->nxt;
    }
    new1->nxt=head;
    tmp->nxt=new1;
}
void insertAtmiddle(Node* &head,int data,int position)
{
     Node *new1=new Node(data);
    Node* tmp=head;
    for(int i=0;i<position-1;i++)
    tmp=tmp->nxt;
    new1->nxt=tmp->nxt;
    tmp->nxt=new1;

}
void deleteAtfirst(Node* &head)
{
    Node* tmp=head;
    Node* p=head;
    while(tmp->nxt!=head)
    {
        tmp=tmp->nxt;
    }
    tmp->nxt=p->nxt;
    head=head->nxt;
    delete (p);
}

void deleteAtlast(Node* &head)
{
    Node* tmp=head->nxt;
    // Node* p=head;
    Node* sLast=head;
    while(tmp->nxt!=head)
    {
        tmp=tmp->nxt;
        sLast=sLast->nxt;
    }
   sLast->nxt=head;
   delete tmp;
}

void deleteInMiddle(Node *&head,int position)
{
    Node *tmp=head;
    Node*p=head->nxt;
    for(int i=0;i<position-1;i++)
    {
        tmp=tmp->nxt;
        p=p->nxt;
    }
    tmp->nxt=p->nxt;
    delete p;


}
```

8.3 Doubly Linked List:

Doubly Linked List contains a node that has three entries: (1) data part, (2) pointer to the next node, and (3) pointer to the previous node. We can traverse in both forward and backward directions in doubly Linked Lists.

```

class Node
{
    public:
```

```
    int data;
    Node* prev;
    Node* nxt;

    //constructor
    Node(int data)
    {
        this->data=data;
    }
};
void traverse(Node* &head)

{   cout<<"normal"<<endl;
    Node* tmp=head;
    while(tmp!=NULL)
    {
        cout<<tmp->data<<endl;
        tmp=tmp->nxt;
    }
    cout<<"--------------------------------"<<endl;
}

void insertAtHead(int data,Node* &head)
{
    Node* tmp=head;
    Node *new1=new Node(data);
    new1->nxt=tmp;
    tmp->prev=new1;
    new1->prev=NULL;
    head=new1;
}
void insertAttail(int data,Node* &head)
{
    Node* tmp=head;
    while(tmp->nxt!=NULL)
    tmp=tmp->nxt;
    Node* new1=new Node(data);
    tmp->nxt=new1;
    new1->nxt=NULL;
    new1->prev=tmp;

}
void insertAtMiddle(int data,Node* head,int position)
{
    Node* tmp=head;
    for(int i=0;i<position-1;i++) tmp=tmp->nxt;
    Node* new1=new Node(data);
    new1->nxt=tmp->nxt;
    new1->prev=tmp;
    tmp->nxt->prev=new1;
    tmp->nxt=new1;
}
void deleteFirst(Node* &head)
{
    Node* tmp=head;
    head=tmp->nxt;
    head->prev=NULL;
    delete tmp;

}
void deleteAtLast(Node *&head)
{
    Node *tmp=head;
    while(tmp->nxt!=NULL) tmp=tmp->nxt;
    Node* p=tmp->prev;
    p->nxt=NULL;
    delete tmp;

}
void deletemiddle(Node *&head,int position)
{
    Node* tmp=head;
    for(int i=0;i<position;i++) tmp=tmp->nxt;
    Node *p=tmp->prev;
    Node *n=tmp->nxt;
    n->prev=p;
    p->nxt=n;
    // cout<<tmp->data<<"hi"<<endl;
    delete tmp;

}
void traverse_reverse(Node* &head)
{
    cout<<"reverse"<<endl;
    Node* tmp=head;
    while(tmp->nxt!=NULL)
    {
        tmp=tmp->nxt;
    }
    while(tmp!=NULL)
    {
        cout<<tmp->data<<endl;
        tmp=tmp->prev;
    }
    cout<<"----------------------------------------"<<endl;
}
```

9 Recursion:

Recursion is a programming technique that enables a function to call itself to solve a problem. This can be a very effective strategy since it enables us to break down big problems into smaller, more manageable sub-problems, allowing us to find solutions to those difficulties.

Recursion Example

Let's understand the Fibonacci sequence, in which each number is the sum of the two previous numbers. For the n-1th and n-2th numbers in the sequence, the recursive function that calculates the nth number in the series would make two calls to itself. The function can return 1 in the base case of n=1 and 0 in base case of n=0.

```
    int fib(int n)
    {
    if(n==0) return 0;
    if(n==1)return 1;
    return fib(n-1)+fib(n-2)
    }
```

Types of Recursion

- Tail Recursion: When the recursive call is at the end of the function.
- Not-tailed Recursion: When the recursive call is not the end of the function.

10 STL Libraries in C++:

10.1 Iterators:

```
#include <iostream>
#include<vector>
using namespace std;

int main() {

    vector <string> languages = {"Python", "C++", "Java"};

    // create an iterator to a string vector
    vector<string>::iterator itr;

    // iterate over all elements
    for (itr = languages.begin(); itr != languages.end(); itr++) {
        cout << *itr << ", ";
    }

    return 0;
}
```

10.2 Map:

```
    std::map<std::string,int> mymap = {
                { "alpha", 0 },
                { "beta", 0 },
                { "gamma", 0 } };//Initliasting map
     for (auto& x: mymap) {
    std::cout << x.first << ": " << x.second << '\n';
  } //traversing through the map
```

10.3 Vector:

```
  std::vector<int> first;
  std::vector<int> second;
  std::vector<int> third;

  first.assign (7,100);             // 7 ints with a value of 100

  std::vector<int>::iterator it;
  it=first.begin()+1;

  second.assign (it,first.end()-1); // the 5 central values of first

  int myints[] = {1776,7,4};
  third.assign (myints,myints+3);   // assigning from array.

  std::cout << "Size of first: " << int (first.size()) << '\n';
  std::cout << "Size of second: " << int (second.size()) << '\n';
  std::cout << "Size of third: " << int (third.size()) << '\n';
  return 0;
```

10.4 List:

```
    #include <list>
    std::list<string> name; // Declaration of a list
    name.push_back("ram:);  // Insert an element at the end
    name.push_front("shyam"); // Insert an element at the beginning
    name.pop_back(); // Remove the last element
```

10.5 Set:

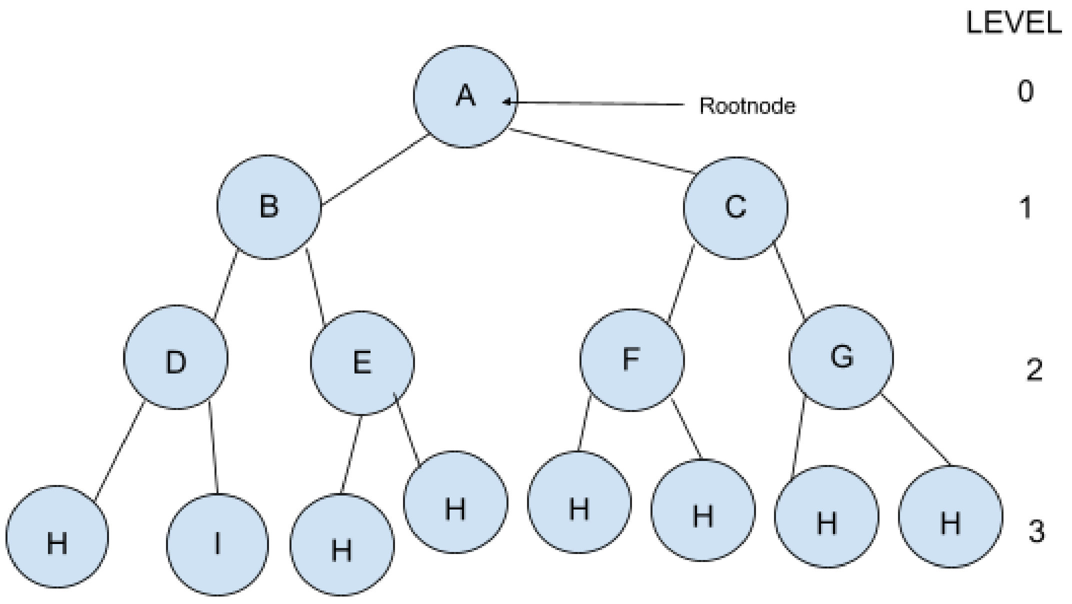
This are just like sets in maths.

```
int myints[] = {75,23,65,42,13,13};
std::set<int> myset (myints,myints+6);

std::cout << "myset contains:";
for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
  std::cout << ' ' << *it;

std::cout << '\n';
```

11 Binary Tree:

Binary trees are a particular type of tree where each internal node has at most two children.
A Binary Tree can also be defined as a tree where each vertex has a degree of at most 2.
The root node divides the tree into two subtrees:-

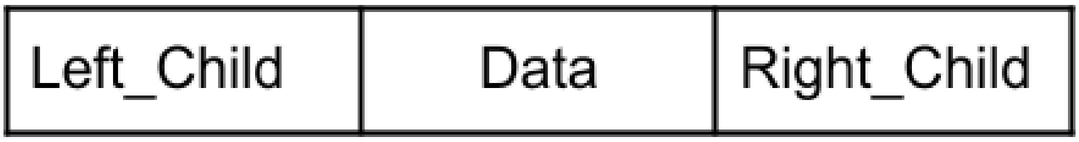
- Left sub-tree
- Right sub-tree



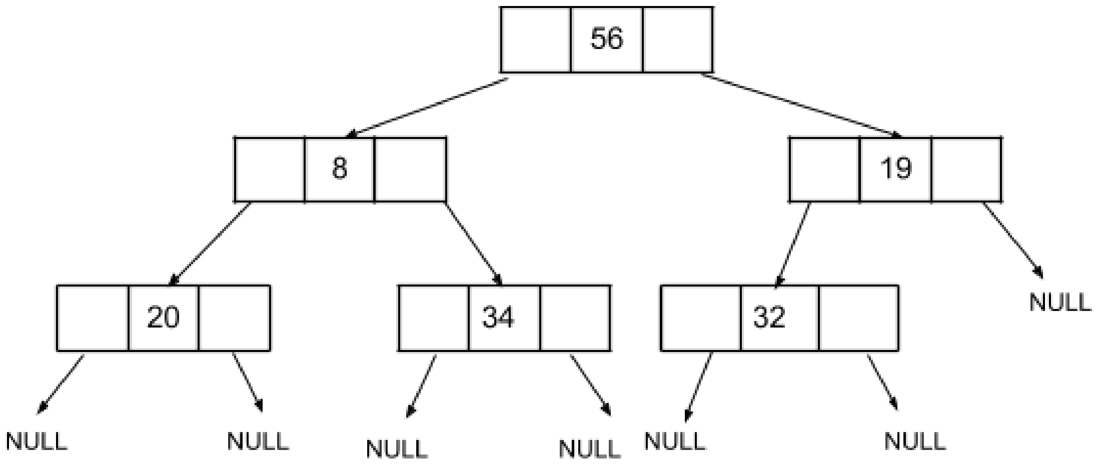
11.1 Linked representation (Explicit Representation)

This linked representation is the most efficient representation of the binary tree. For the linked representation of a binary tree, we use the concept of a Linked List.
Below is the code snippet of the linked list used to represent Binary Tree.

```
  class Node
  {
  public:
      int data;
      Node* left;
      Node* right;
  }
```



So the representation of Binary Tree using linked list look like.



PoA for upcoming Weeks.

Week 5:

Binary Search Trees (BST): Learn about BST properties, insertion, deletion, and traversal (in-order, pre-order, post-order).
Hashmaps: Understand hash functions, collision resolution (chaining, open addressing), and efficient key-value storage.
Heaps: Explore min-heaps and max-heaps, heapify operations, and their applications (e.g., priority queues).

Week 6:

Graphs: Study graph representations (adjacency matrix, adjacency list), traversal (DFS, BFS), and common algorithms (Dijkstra's, Kruskal's, etc.).
Backtracking: Dive into solving problems using backtracking techniques (e.g., N-Queens, Sudoku).

Week 7:

Dynamic Programming (DP): Master the concept of optimal substructure and overlapping subproblems. Solve problems using memoization and tabulation.
Greedy Algorithms: Understand greedy choice property and apply it to problems like fractional knapsack, Huffman coding, and activity selection.

Week 8:

Practice and Review: Solve miscellaneous problems across various topics (e.g., string manipulation, sorting, etc.).