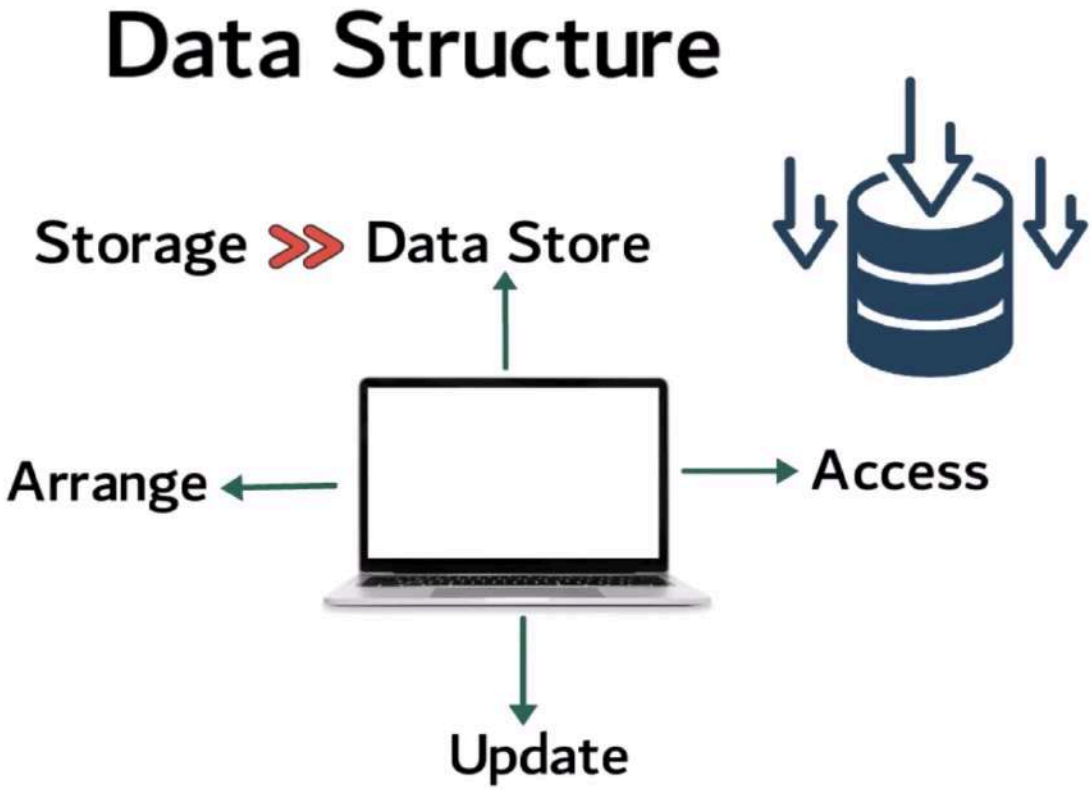
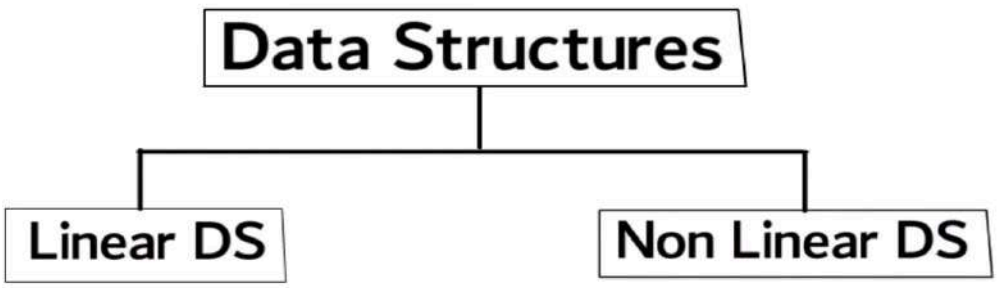
Data Structure and Algorithm

Data structures are different ways of organizing information or data in your computer program. Like folders , they help you store things efficiently. There are many data structures, each suited for different tasks.

Algorithms are step-by-step instructions for solving problems. They're like the librarian's search method for finding a specific book. There are many algorithms too, some good for sorting things, some for searching, and so on.





1.Linear Data Structure

A linear data structure is a type of data structure where elements are arranged sequentially, one after the other. Each element is connected to its previous and next element, making traversal through the elements straightforward. Linear data structures are characterized by their ordered nature and the fact that they store data elements in a linear order.

Here are some common types of linear data structures:

1. Arrays:

- An array is a collection of elements, each identified by an index or a key.
- Arrays are of fixed size, and all elements are of the same data type.
- They allow random access to elements using the index.

2. Linked Lists:

- A linked list is a collection of nodes where each node contains data and a reference (or link) to the next node in the sequence.
- There are different types of linked lists: singly linked lists (each node points to the next node), doubly linked lists (each node points to both the next and the previous nodes), and circular linked lists (the last node points back to the first node).

3. Stacks:

- A stack is a linear data structure that follows the Last In First Out (LIFO) principle.
- Elements can be added (pushed) and removed (popped) only from the top of the stack.

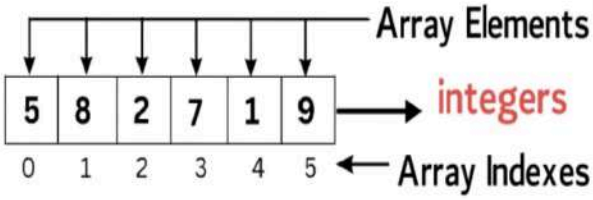
4. Queues:

- A queue is a linear data structure that follows the First In First Out (FIFO) principle.
- Elements are added (enqueued) at the rear and removed (dequeued) from the front.

Linear Data Structures

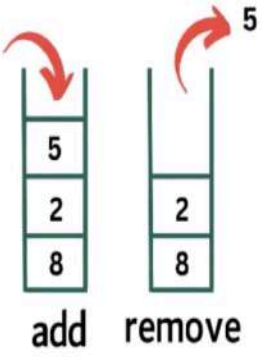
Data Elements >> Sequence >> Store

Array



Stack (LIFO)

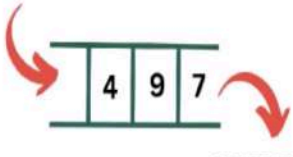
EXAMPLE



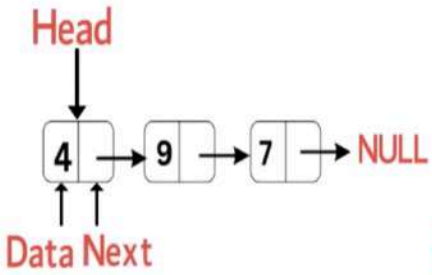
Queue (FIFO)

add

EXAMPLE



Linked List



2. Non-linear data structures

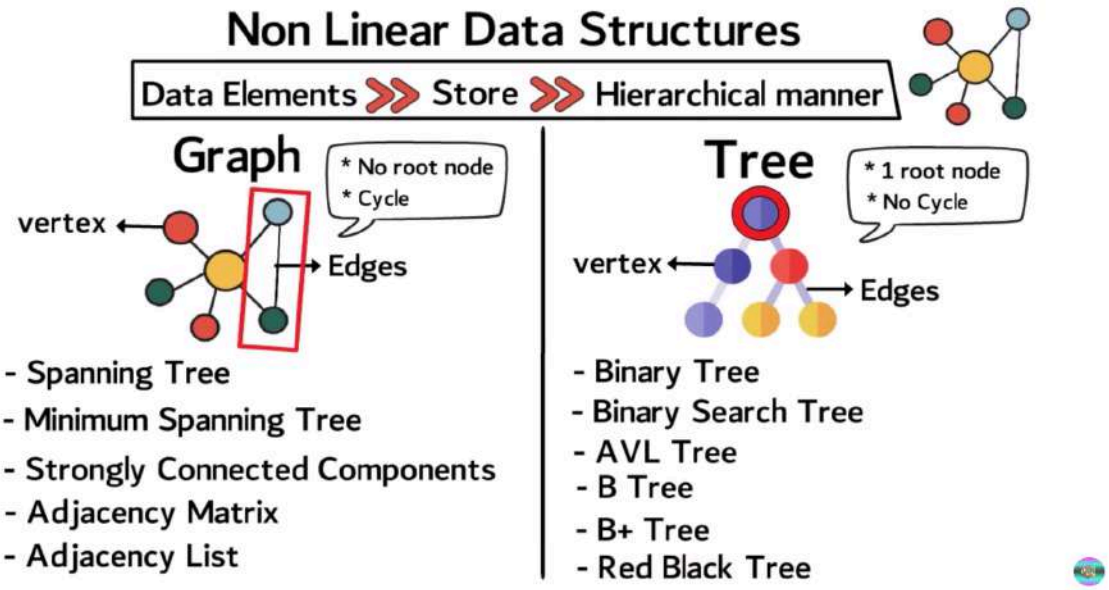
Non-linear data structures are data structures where elements are not arranged in a sequential order. Instead, they are organized in a hierarchical or interconnected manner, allowing for more complex relationships among the data elements. Non-linear data structures are useful for representing data with multiple relationships or for efficiently performing certain types of operations.

1. Trees:

- A tree is a hierarchical data structure consisting of nodes, with a single node designated as the root. Each node has zero or more child nodes.
- Common types of trees include binary trees, binary search trees (BST), AVL trees, and B-trees.
- Trees are used in various applications such as representing hierarchical data, performing quick searches, and sorting data.

2. Graphs:
A graph is a collection of nodes (vertices) and edges connecting pairs of nodes. Graphs can be directed (edges have a direction) or undirected (edges do not have a direction).

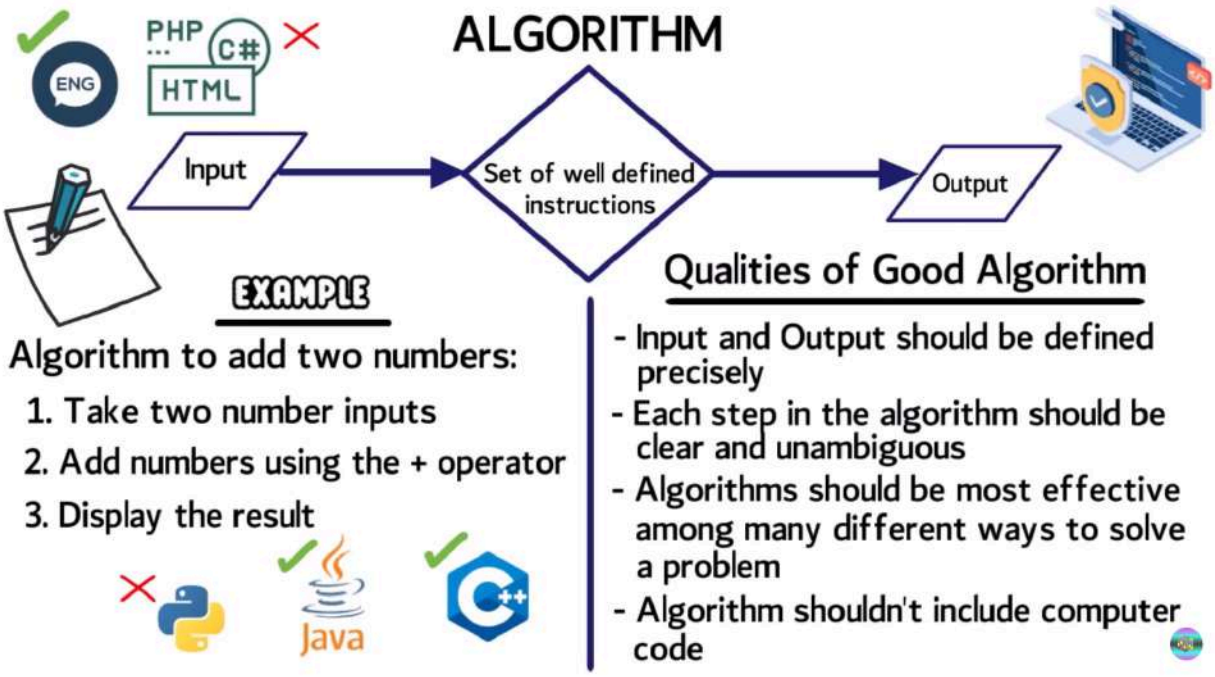
- Graphs are used to represent networks such as social networks, transportation networks, and dependency graphs.
- There are various ways to represent graphs, including adjacency lists and adjacency matrices.



DIFFERENCE

Linear Data Structures	Non linear Data Structures
Data items are arranged in sequential order, one after another.	Data items are arranged in non sequential order (Hierarchical manner).
Data items are present on the single layer.	Data items are present on different layers.
It can be traversed on a single run.	It requires multiple runs for traversing.
Time complexity increases with size of data.	Time complexity remains the same.
Memory utilization is not efficient.	Different structures utilize memory in efficient ways depending on the need.
EXAMPLE	EXAMPLE
Array - Stack - Queue - Linked list	Tree - Graph

ALGORITHM



Algorithm : To add two numbers entered by the user

Step 1 : Start

Step 2 : Declare varibales num 1 , num 2 , sum

Step 3 : Read values of num 1 and num 2

Step 4 : Add num 1 and num 2 and assign the result to sum

Step 5 : Display sum

Step 6 : Stop

Algorithm : Find largest number among three numbers

Step 1 : Start

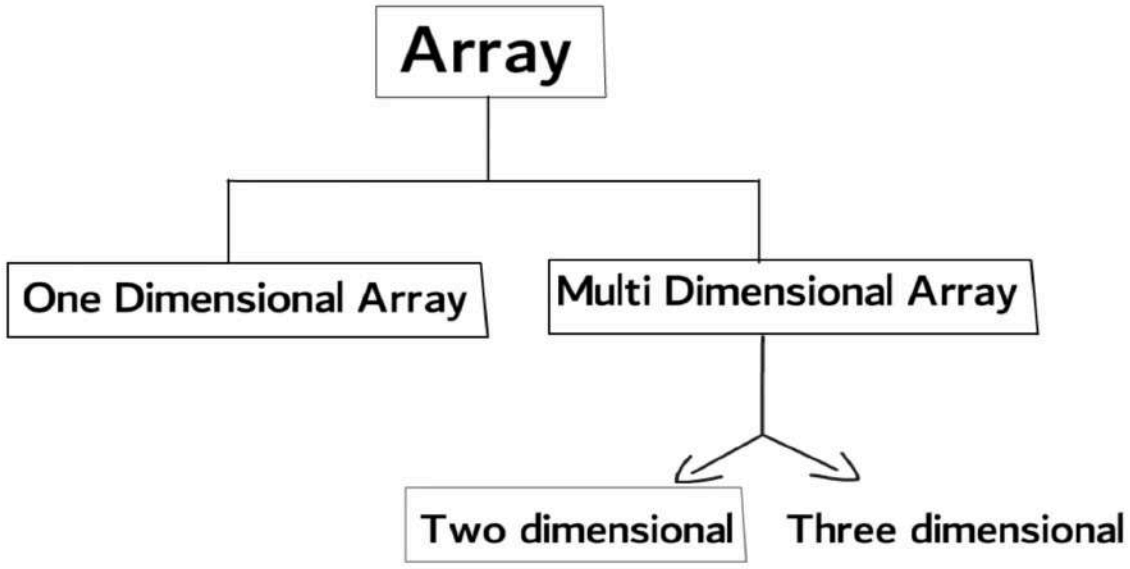
Step 2 : Declare varibales a , b and c

Step 3 : Read variables a , b and c

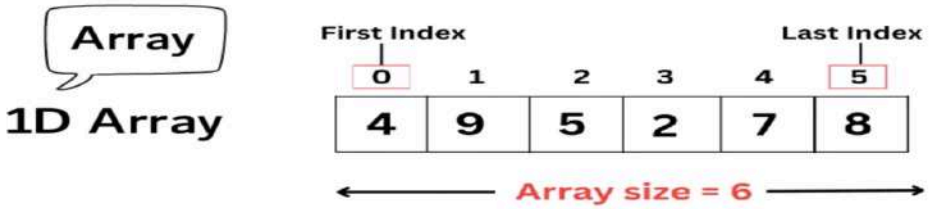
Step 4 : if a > b

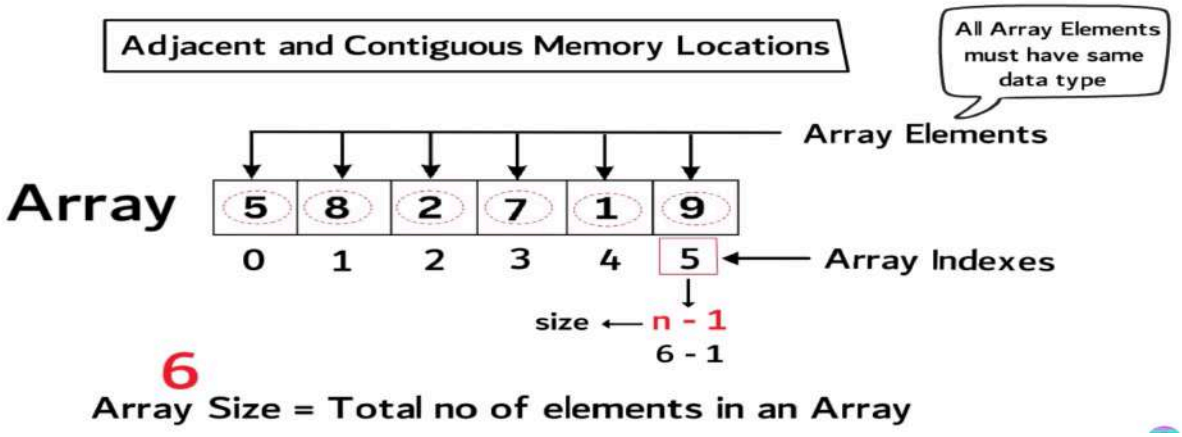
```
        if a > c
          Display a is the largest number
        Else
          Display c is the largest number
    Else
        if b > c
          Display b is the largest number
        Else
          Display c is the largest number
```

Step 5 : Stop

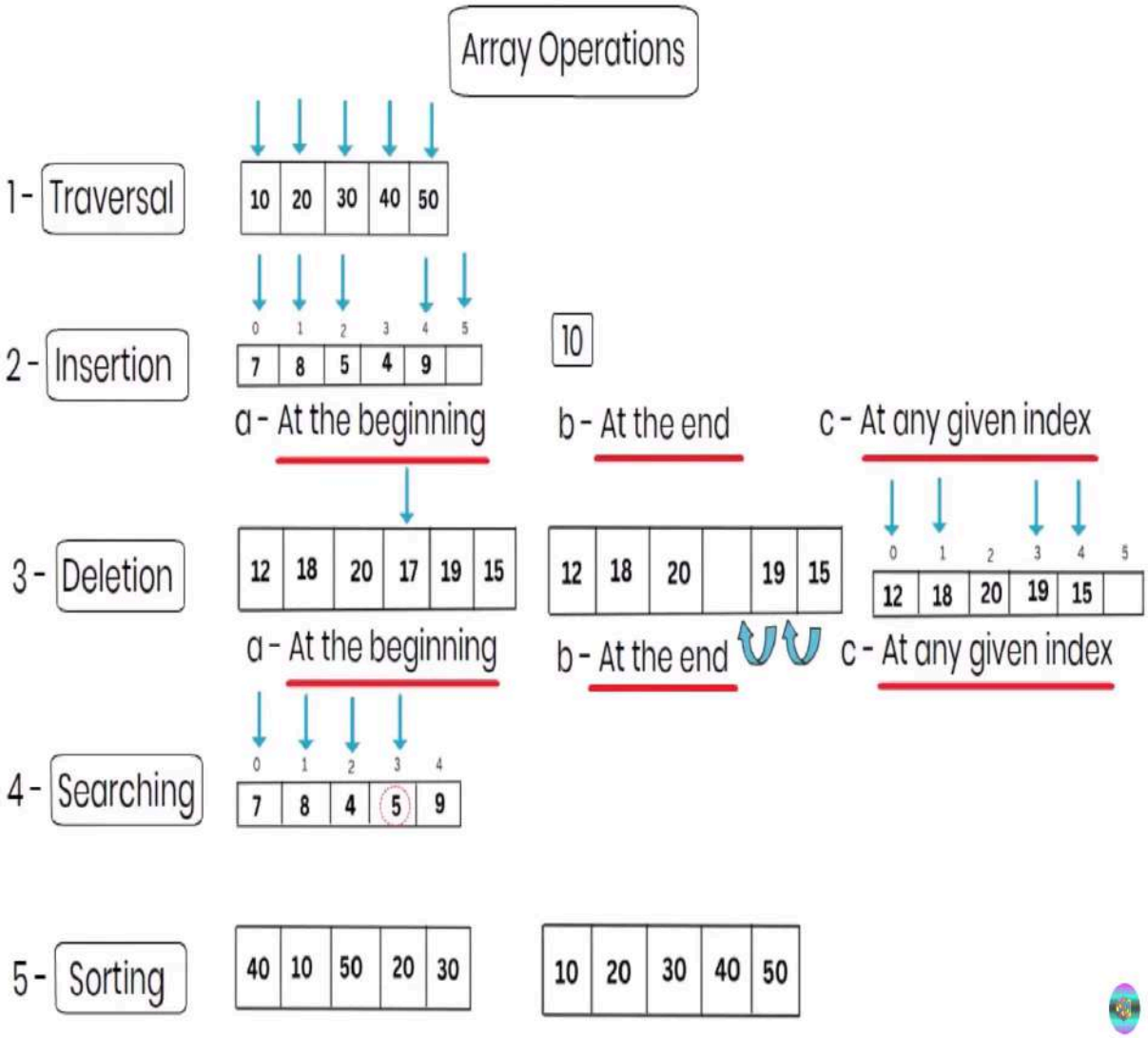


One Dimensional Array

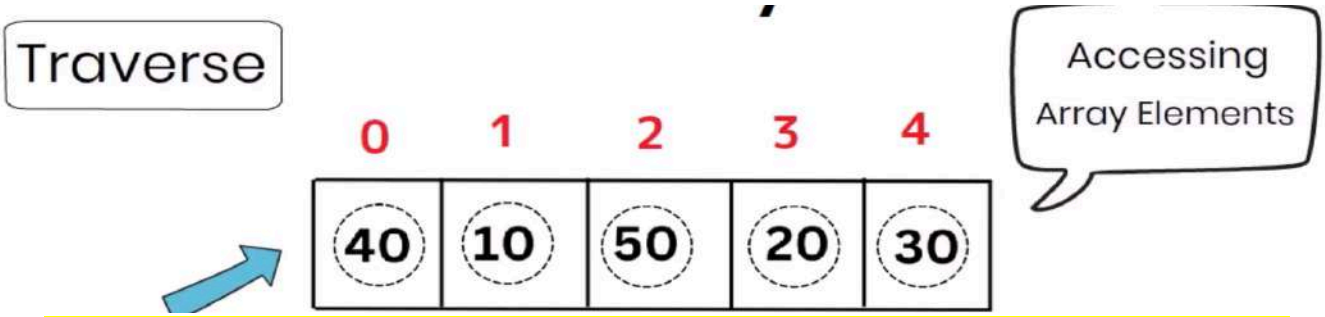




ARRAY OPERATIONS



1: Traversal Operation

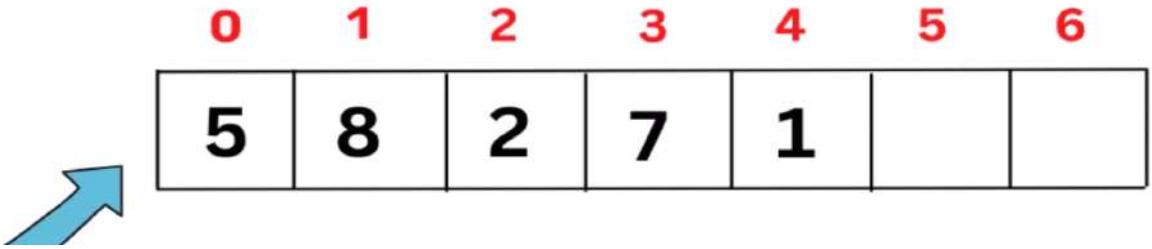


Traversing an array means accessing each element of the array, typically one at a time, in order to process it in some way. This is a fundamental operation that can be performed in several programming languages.

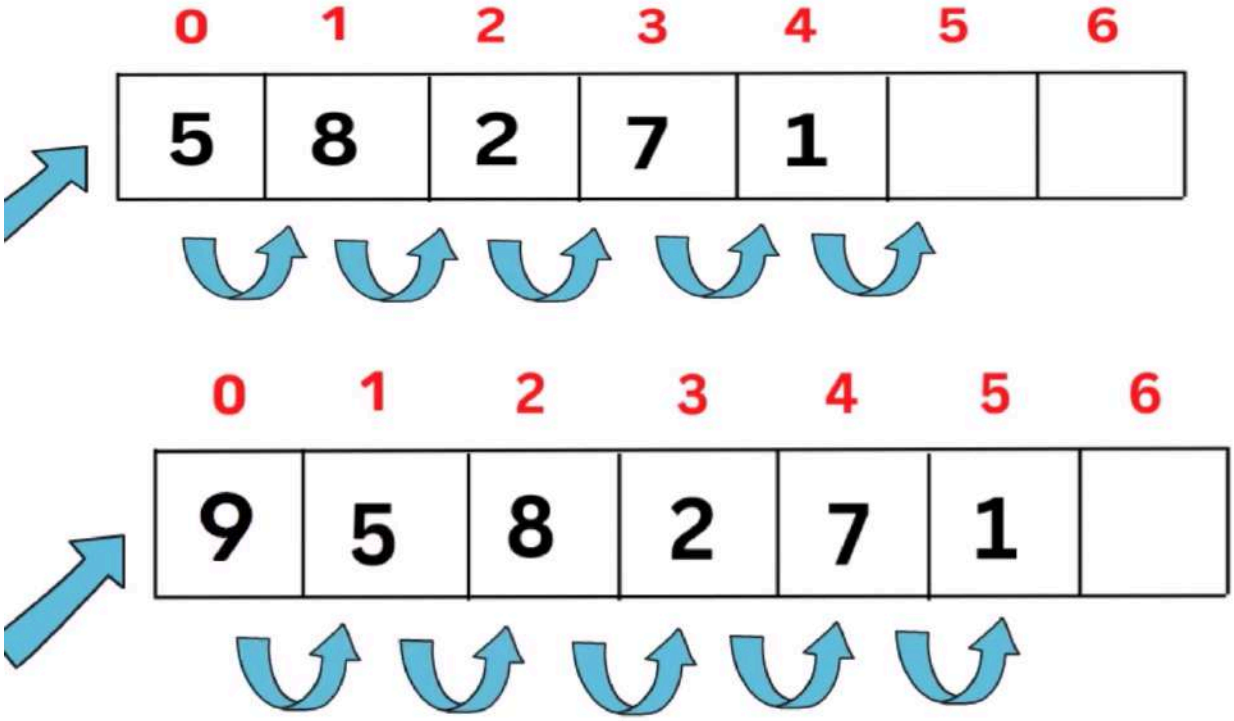
```cpp
#include <iostream>
using namespace std ;

int main() {
    int a[5],n;
    cout <<"ENTER SIZE OF ARRAY : ";
    cin>>n;
    cout<< "ENTER THE ELEMENTS OF ARRAY : "<<endl;
    for(int i =0; i<n ; i++)
      {
    cin>>a[i];
      }

    cout<<"ELEMENTS ARE : ";

    //    Array Traversing

    for(int i =0; i<n ; i++)
      {
    cout<< "\t"  << a[i];
      }
    return 0;
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS
Warning: PowerShell detected that you might be using a screen reac

PS C:\Users\Aditya Yadav> cd "c:\Users\Aditya Yadav\Desktop\CODES\
ENTER SIZE OF ARRAY : 5
ENTER THE ELEMENTS OF ARRAY :
1 2 3 4 5
ELEMENTS ARE :  1       2       3       4       5
```

2.1 : Insertion At Beginning



Inserting an element **at the beginning** of an array involves shifting all existing elements one position to the right and then placing the new element at the first position.

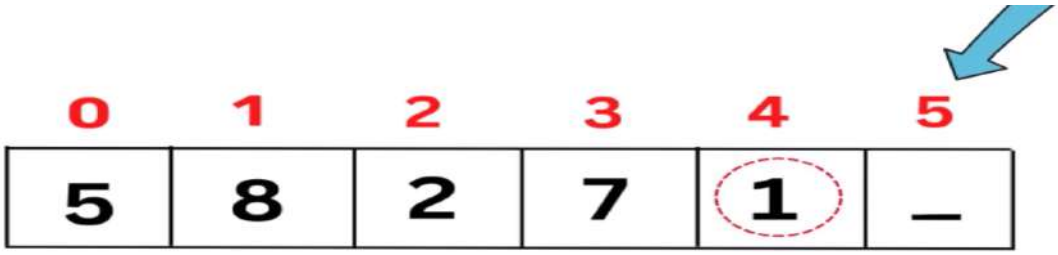


```cpp
#include <iostream>
using namespace std;
int main ()
{
    int array[10], n , x;
    cout<< " ENTER AARAY SIZE :";
    cin>> n;
    cout<< " ENTER AARAY ELEMENTS :";
    for  (int i=0; i < n; i++)
    {
        cin >> array [i];
    }
cout << "ENTER THE ELEMENT AT THE BEGINNING : ";
cin >> x ;
for (int i=n-1 ; i >= 0; i--)
{
    array[i+1] =array[i];
}
array[0]= x;
n++;
cout <<"ARRAY ELEMENTS ARE : "  ;
for (int i=0; i < n; i++)
{
    cout << "\t" << array[i] ;
}
return 0;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
Warning: PowerShell detected that you might be using a screen reader and has disa

PS C:\Users\Aditya Yadav> cd "c:\Users\Aditya Yadav\Desktop\CODES\DSA\" ; if ($?)
 ENTER AARAY SIZE :5
 ENTER AARAY ELEMENTS :1 2 3 4 5
ENTER THE ELEMENT AT THE BEGINNING : 11
ARRAY ELEMENTS ARE :    11      1       2       3       4       5
```

2.2 : Insertion At The End

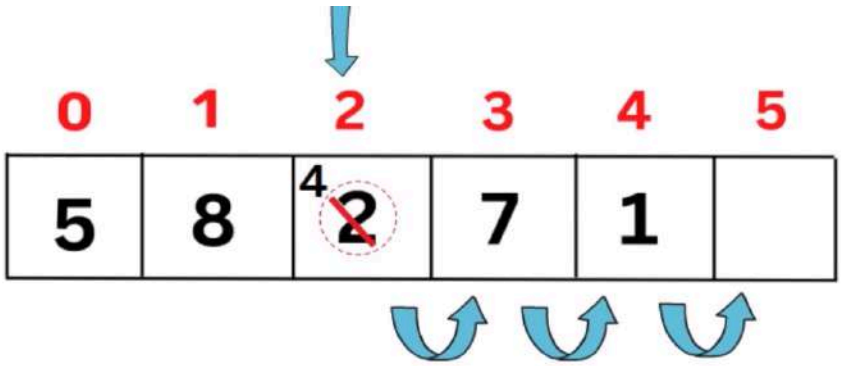


Inserting an element **at the end** of an array typically involves adding the new element after the last existing element.

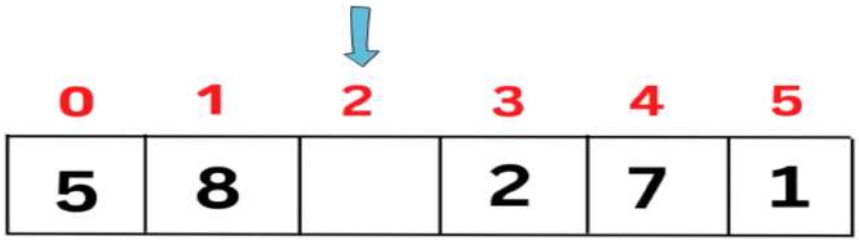
```cpp
#include <iostream>
using namespace std;
int main ()
{
    int a[10] ,n ,x;
    cout <<"ENTER THE SIZE OF ARRAY : ";
    cin >> n ;
    cout <<"ENTER THE ELEMNTS OF THE ARRAY : ";
    for (int i = 0; i <n; i++)
    {
        cin >> a[i] ;
    }

            cout <<"ARRAY IS : ";
            for (int i = 0; i <n; i++)
            {
                cout <<" " << a[i] ;
            }

    cout << "\nENTER THE LAST ELEMENT OF THE ARRAY : ";
    cin >> x ;

    a[n] = x;
    cout << "NEW ARRAY IS : ";
    for (int i=0; i <=n; i++)
    {
        cout <<" " << a[i] ;
    }
return 0;
}
```

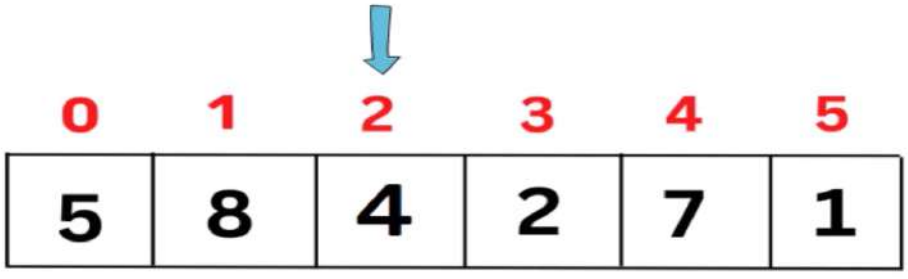
```
ENTER THE SIZE OF ARRAY : 5
ENTER THE ELEMNTS OF THE ARRAY : 1 2 3 4 5
ARRAY IS :  1 2 3 4 5
ENTER THE LAST ELEMENT OF THE ARRAY : 6
NEW ARRAY IS :  1 2 3 4 5 6
```

2.3 : Insertion At Specific Position



Inserting an element **at any specific position** in an array typically requires shifting the existing elements to the right from the insertion point to make space for the new element. Directly inserting an element at any position without this shift would result in overwriting the existing data.

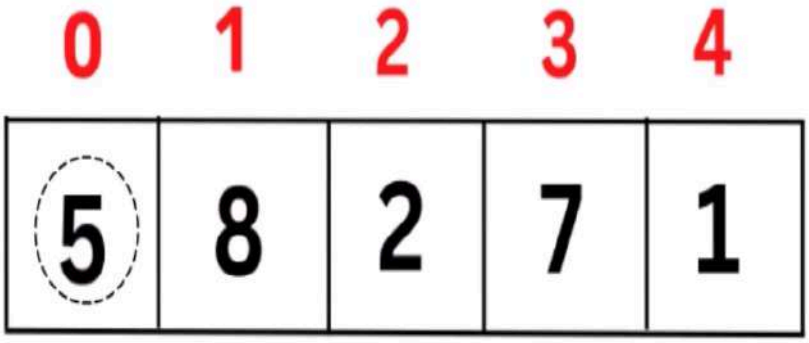




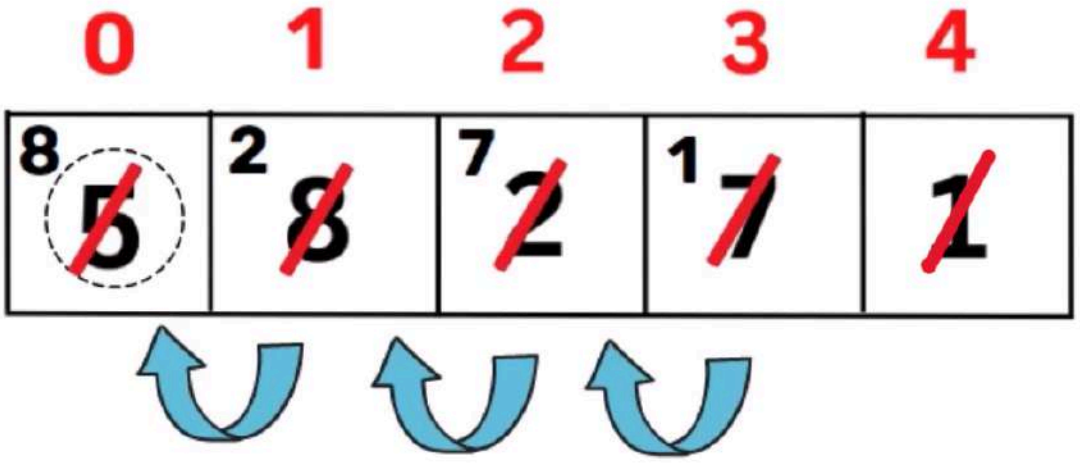
```cpp
#include <iostream>
using namespace std ;
int main ()
{
    int a[10],n,x, pos;
    cout << "Enter the size of the array: ";
    cin >> n;
    cout << "Enter the elements of the array: ";
    for (int i=0; i<n; i++)
    {
        cin >> a[i];
    }
    cout<<" ARRAY ARE : ";
    for (int i = 0 ; i<n ; i++)
    {
        cout << a[i] << " ";
    }
    cout << "\n Enter the insertion location :";
    cin >> pos;
    cout << "Enter the value to insert : ";
    cin >> x;
    for (int i =n-1 ;i>=pos-1 ; i--)
    {
        a[i+1] = a[i];
    }
    a[pos-1] = x;
    n++;
    cout <<" NEW  ARRAY : ";
    for (int i = 0 ; i<n ; i++)
    {
        cout << a[i] << " ";
    }
    return 0;
}
```

```
Enter the size of the array: 5
Enter the elements of the array: 1 2 3 4 5
 ARRAY ARE : 1 2 3 4 5
 Enter the insertion location :4
Enter the value to insert : 44
 NEW  ARRAY : 1 2 3 44 4 5
```

3.1 : Deletion At The Beginning



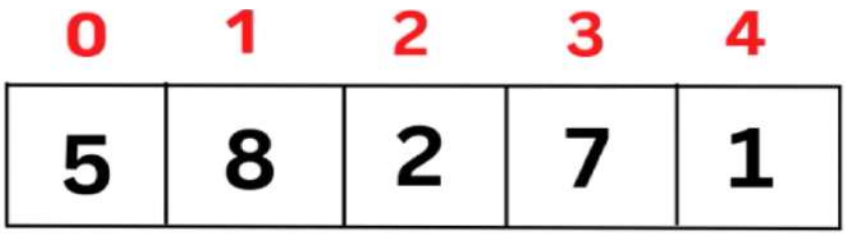
Deleting an element from **the beginning** of an array typically involves shifting all subsequent elements one position to the left to fill the gap. In this process, we do not delete or remove any index elements; instead, we overwrite the next index element with the previous index element.



```cpp
#include <iostream >
using namespace std ;
int main ()
{
    int a [10],n;
    cout << "ENTER THE SIZE OF THE ARRAY : ";
    cin >> n ;
    cout << "ENTER THE ELEMENTS OF THE ARRAY : ";
    for (int i = 0; i <n ; ++i)
    {
        cin >> a[i] ;
    }
    cout << "ELEMENTS OF THE ARRAY IS : ";
    for (int i = 0; i <n ; ++i)
    {
        cout <<" " << a[i] ;
    }
    cout << "\nAFTER DELETATION OF 1st ELEMENT OF THE ARRAY : ";
    for (int i = 0; i <n-1 ; ++i)
    {
        a[i] =a[i+1];
    }
    n-- ;

    for (int i = 0; i <n ; ++i)
    {
        cout<< " " <<  a[i] ;
    }
}

```

```
ENTER THE SIZE OF THE ARRAY : 5
ENTER THE ELEMENTS OF THE ARRAY : 1 2 3 4 5
ELEMENTS OF THE ARRAY IS :  1 2 3 4 5
AFTER DELETATION OF 1st ELEMENT OF THE ARRAY :  2 3 4 5
```

3.2 : Deletion At The End

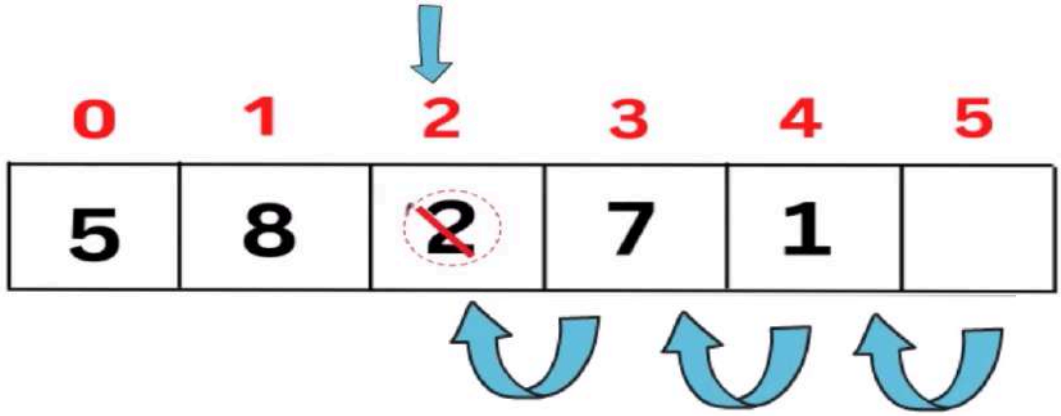


Deleting an element from **the end of an array** involves removing the last element. This operation is typically simpler than deleting from the beginning or from any arbitrary position because no shifting of elements is necessary.
We just reduce the size of Array (n) .

```
#include <iostream>
using namespace std;
int main ()
{
    int a[10],n ;
    cout <<"ENTER THE SIZE OF THE ARRAY : ";
    cin >> n;
    cout << "ENTER THE ELEMENTS OF THE ARRAY : ";
    for(int i=0; i<n; i++)
    {
        cin >> a[i];
    }
    cout << "ELEMENTS OF THE ARRAY IS : ";
    for(int i=0; i<n; i++)
    {
        cout <<" " << a[i];
    }
    cout << "\nAFTER THE DELETEATION :";
    n--;
    for(int i=0; i<n; i++)
    {
        cout <<" " << a[i];
    }

}
```

```
ENTER THE SIZE OF THE ARRAY : 5
ENTER THE ELEMENTS OF THE ARRAY : 1 2 3 4 5
ELEMENTS OF THE ARRAY IS :  1 2 3 4 5
AFTER THE DELETEATION : 1 2 3 4
```

3.3 : Deletion At any Position



Deleting an element from **any specific position** in an array typically involves shifting the elements to the left from the deletion point to fill the gap left by the deleted element.

```cpp
#include <iostream>
using namespace std ;
int main ()
{
    int a [10],n , x;
    cout <<"ENTER THE SIZE OF THE ARRAY : ";
    cin >>  n ;
    cout << "ENTER THE ELEMENTS OF THE ARRAY : ";
    for(int i =0 ; i< n ; i++)
    {
        cin >> a[i] ;
    }
    cout << "ELEMENTS OF THE ARRAY IS : ";
    for(int i =0 ; i< n ; i++)
    {
        cout<<" "<< a[i] ;
    }
    cout << "\nENTER THE POSIITION OF THE ELEMEMNT TO DELETE : ";
    cin >> x;
    for(int i=x ; i<n ; i++)
    {
        a[i-1] = a[i];
    }
    n-- ;
    cout << "NEW ELEMENTS OF THE ARRAY IS : ";
    for(int i =0 ; i< n ; i++)
    {
        cout<<" "<< a[i] ;
    }
}
```

```
ENTER THE SIZE OF THE ARRAY : 5
ENTER THE ELEMENTS OF THE ARRAY : 1 2 3 4 5
ELEMENTS OF THE ARRAY IS :  1 2 3 4 5
ENTER THE POSIITION OF THE ELEMEMNT TO DELETE : 4
NEW ELEMENTS OF THE ARRAY IS :  1 2 3 5
```

4.1: Linear Search Array

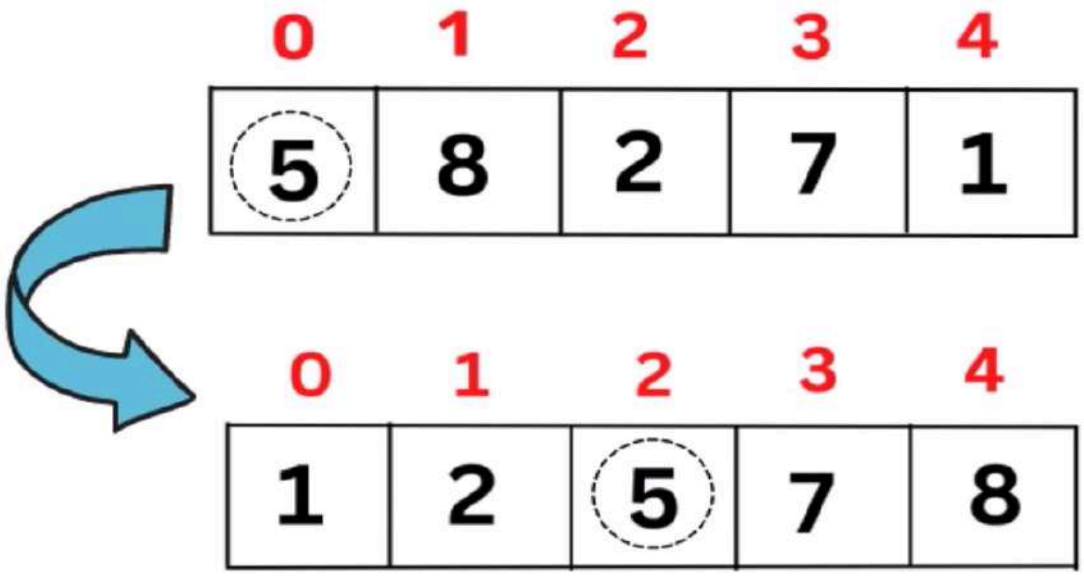
Linear search is a simple searching algorithm that checks each element in an array until the desired element is found or the array is exhausted.
In Linear Search an Array element is searched sequentially.

```cpp
#include<iostream>
using namespace std ;
int main()
{
    int a[10],n,x,i;
    cout <<"ENTER THE SIZE OF THE ARRAY : ";
    cin >> n ;
    cout <<"ENTER THE ELEMENTS OF THE ARRAY : ";
    for(int i=0; i<n; i++)
    {
        cin >> a[i] ;
    }
    cout <<"ELEMENTS OF THE ARRAY IS : ";
    for(int i=0; i<n; i++)
    {
        cout <<" " << a[i] <<" | ";
    }
    cout << "\nENTER A NUMBER TO SEARCH IN ARRAY : ";
    cin >> x;
    for(int i=0; i<n; i++)
    {
        if (a[i] == x)
        {
            cout << "ELEMENT FOUND AT INDEX NUMBER : " <<i;
            break;
        }
      else {
            cout << "ELEMENT NOT FOUND IN ARRAY .... ";
            break;
        }
}
}
```

```
ENTER THE SIZE OF THE ARRAY : 5
ENTER THE ELEMENTS OF THE ARRAY : 1 2 3 4 5
ELEMENTS OF THE ARRAY IS :  1 2 3 4 5
ENTER A NUMBER TO SEARCH IN ARRAY : 3
ELEMENT FOUND AT INDEX NUMBER : 2
```

4.2: Binary Search in Array

In **binary search** an element is searched in a **sorted array**.



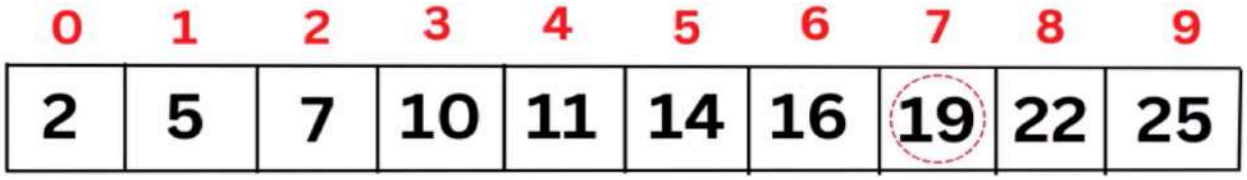
Binary Search Algorithm:

step 1: Sort the array in ascending order.
step 2: Set the low index to the first element of the array and the high index to the last element.
step 3: Set the middle index to the average of low and high indices.
step 4: If the element at the middle index is the target element, return the middle index.
step 5: If the target element is less than the element at the middle index, set the high index to the middle index - 1.
step 6: If the target element is greater than the element at the middle index, set the low index to the middle index + 1.

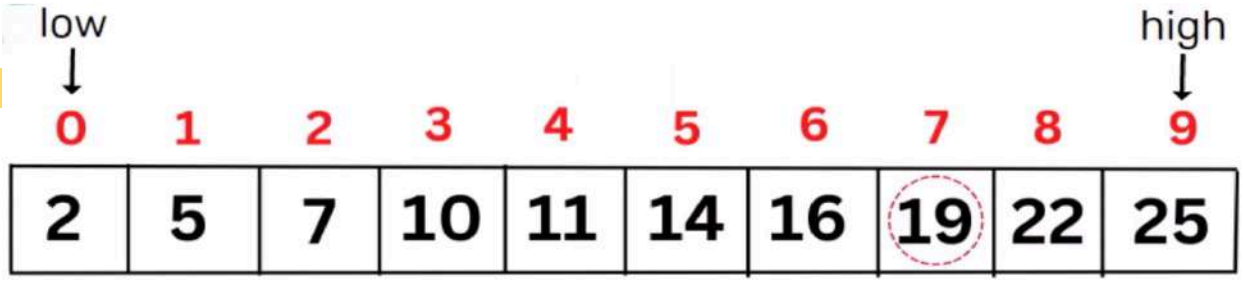
EXAMPLE

Binary Search Working

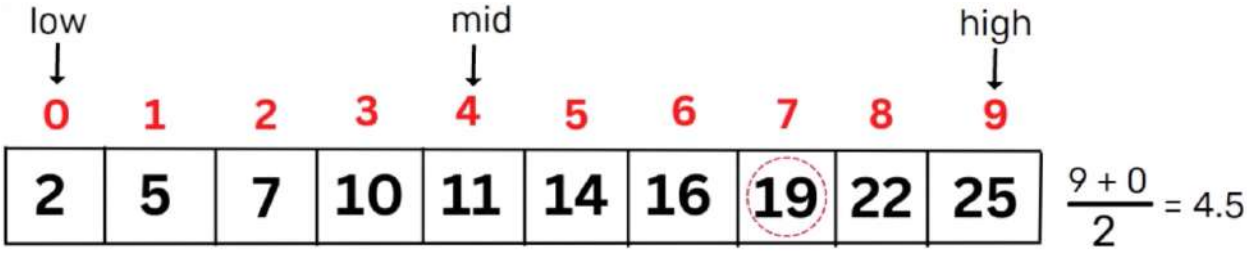
Step 1:



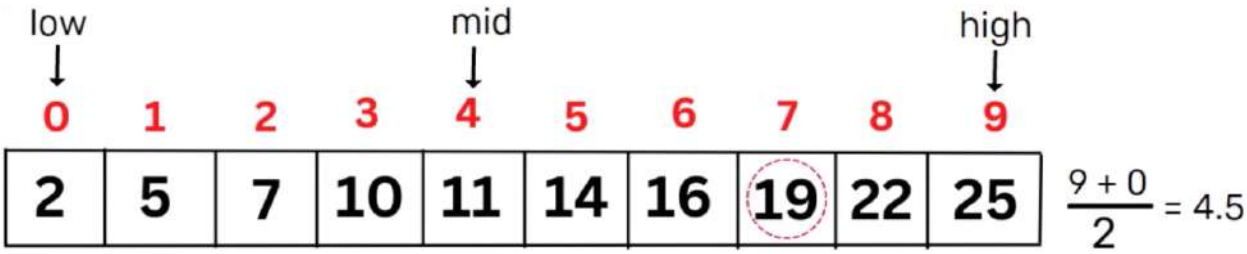
Step 2:



Step 3:

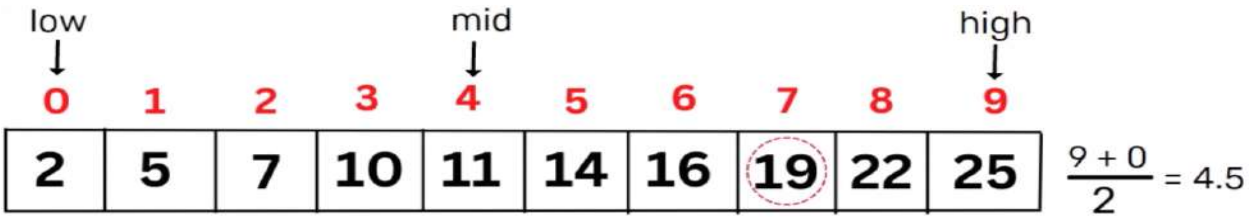


Step 4:



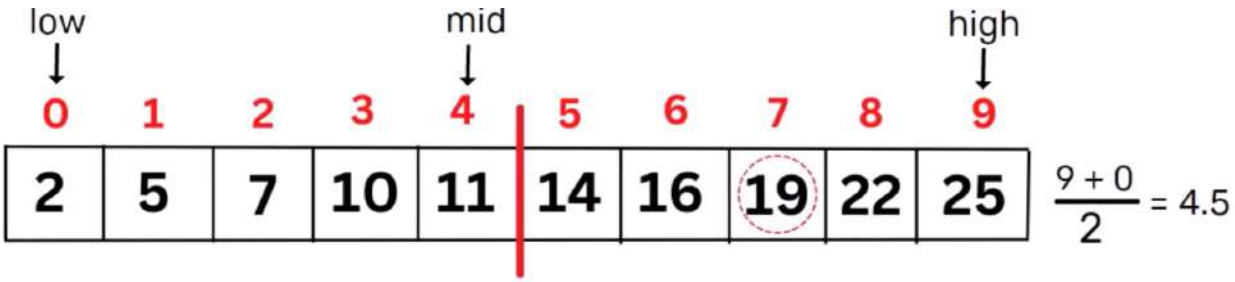
middle index ≠ target element

Step 5:

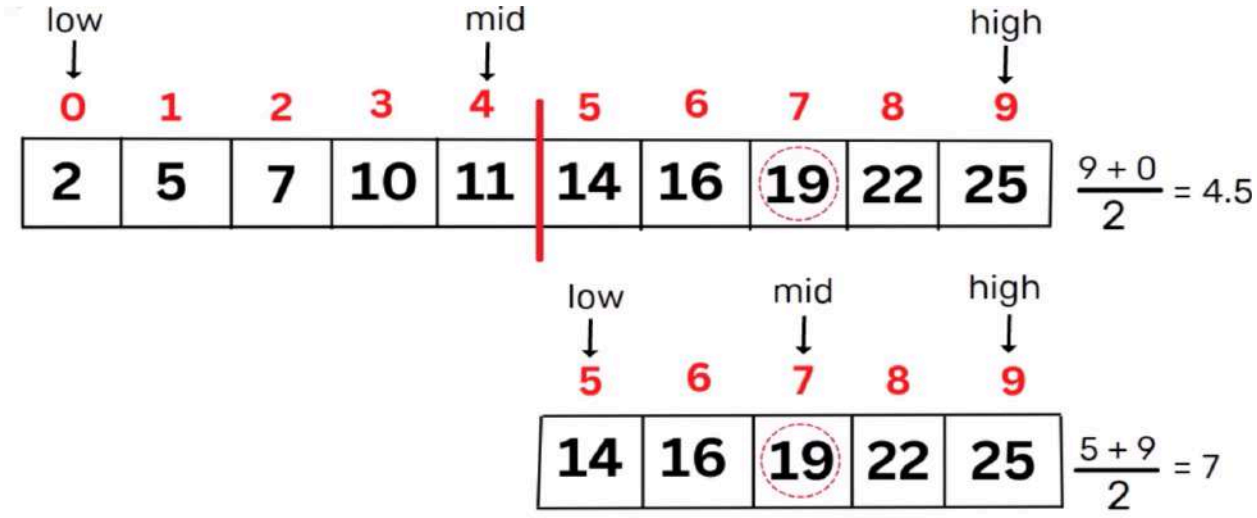


target element is not less than the element at the middle index means Step 5 statement is not equivalent to this situation.

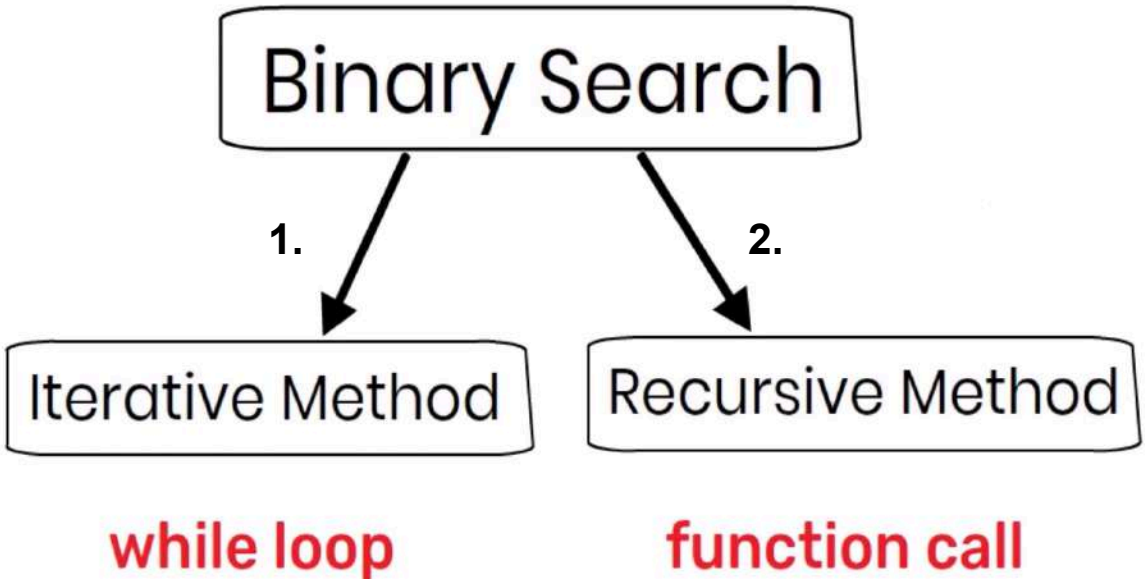
Step 6: If the target element is greater than the element at the middle index, set the low index to the middle index + 1.



So this condition which is followed as statement step 6 so we have to set the low index to the middle index +1 . and again follow all the steps till mid index is equal to the target element.



There are 2 method in Binary Search :



1.Iterative Method (while loop) :-

- A method of finding an element in a sorted array by repeatedly dividing the search interval in half using a loop.
- In this ,a function uses a while loop to repeatedly narrow down the search range by comparing the target with the middle element of the current range.
- If the target value matches the middle element, the search is successful, it returns the middle index.
- If the target is greater than the middle element, it shifts the search range to the right half.
- If the target is smaller than the middle element, it shifts the search range to the left half.
- The process continues until the target is found or the search range is empty.

Iterative binary search is a method to find a specific element in a sorted array by repeatedly dividing the search interval in half. The algorithm uses a loop to narrow down the search range until the target element is found or the range is empty.

9BinarySearchIterativeMethod.cpp > binary(int, int)

```cpp
#include<iostream>
using namespace std;
    int a[10],n,x,i;
int binary(int low, int high)
{
    while(low<=high)
    {
        int mid = (low+high)/2;
        if(a[mid]==x)
        {
            return mid;
        }
        else if(a[mid]>x)
        {
            high = mid-1;
        }
        else {
            low = mid+1;
        }
    }
    return -1;
}
int main()
{
    cout << "Enter size of array : ";
    cin >> n;
    cout << "Enter elements of array in ascending ordeer : ";
    for(int i=0; i<n; i++)
    {
        cin >> a[i];
    }
    cout <<"Enter the elment to search : ";
    cin >>x;
    int result = binary(0,n-1);
    if(result == -1)
    {
    cout << x << " not found in array";
   }
   else{
    cout << x << " found at index " << result;
   }
    return 0;
}
```

```
Enter size of array : 5
Enter elements of array in ascending ordeer : 1 2 3 4 5
Enter the elment to search : 4
4 found at index 3
```

2.Recursive Method (function call):-

- A method of finding an element in a sorted array by repeatedly dividing the search interval in half using recursive function calls.
- In this ,a function uses recursion to achieve the same result.
- It compares the target with the middle element of the current range.
- If the target is found at the middle index, it returns the index.
- If the target is greater than the middle element, it recursively searches the right half.
- If the target is smaller than the middle element, it recursively searches the left half.
- The recursion continues until the target is found or the search range is empty.

Recursive binary search is a method to find a specific element in a sorted array by repeatedly dividing the search interval in half through recursive function calls. The function calls itself with updated bounds (left and right) until the target element is found or the range is empty.

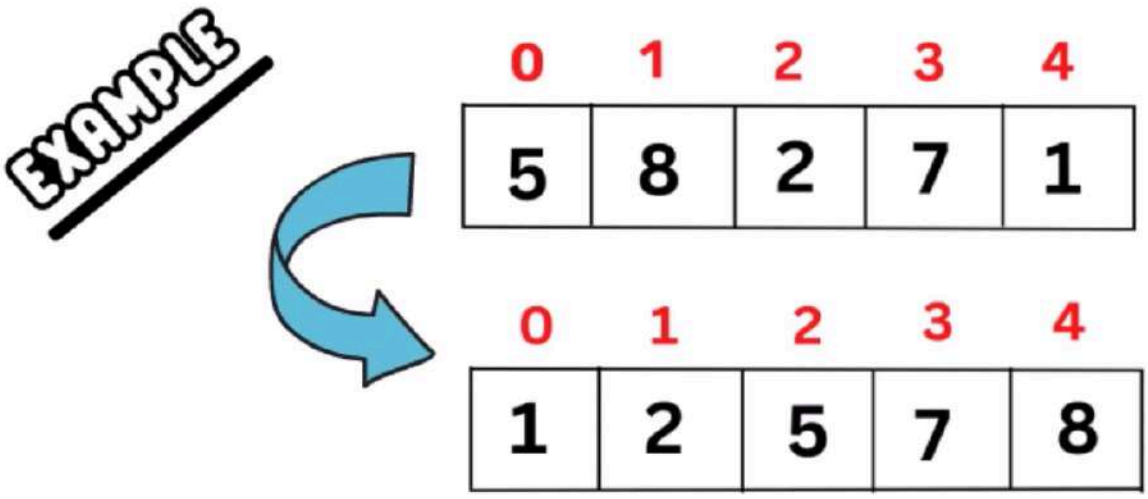
10BinarySearchRecursiveMethod.cpp > main()

```cpp
#include <iostream>
using namespace std;
int a[10],n,x;
int binary(int low , int high){
if (low<=high)
{
    int mid = (low + high)/2;
    if (a[mid] == x)
    return mid;
    else if (a[mid] < x)
    return binary(mid + 1, high);
    else
    return binary(low, mid - 1);
}    else
        return -1;
}
int main()
{
    cout << "Enter the size of array: ";
    cin >> n;
    cout << "Enter elements of array: ";
    for (int i = 0; i < n; i++){
        cin >> a[i];
    }
    cout << "Enter the element to search: ";
    cin >> x;
    int result = binary(0, n - 1);
    if (result==-1){
        cout << x << " is not present in array";
    }
    else{
        cout << x << " is present at index " << result;
    }
    return 0;
}
```

```
Enter the size of array: 5
Enter elements of array: 1 2 3 4 5
Enter the element to search: 6
6 is not present in array
```

5: ARRAY SORTING

Array sorting is the process of arranging the elements of an array in a specific order, typically in ascending or descending order.



```cpp
#include <iostream >
using namespace std;
int main ()
{
    int a[10],n,temp;
    cout << "Enter size of array : ";
    cin >> n ;
    cout << "Enter array elements : ";
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }
    cout << "Entered array elements are : ";
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << " ";
    }
    for (int i = 0; i < n; i++)
    {
        for( int j = i+1; j <n; j++)
        {
            if ( a[i] > a[j])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    cout << "\nSorted array in ascending order is : ";
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << " ";
    }
}
```

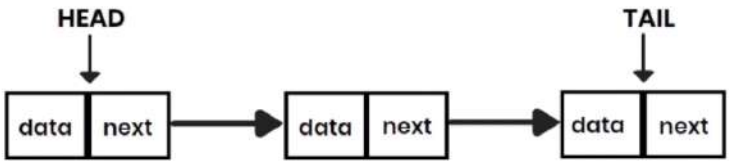
```
Enter size of array : 5
Enter array elements : 2 1 5 4 3
Entered array elements are : 2 1 5 4 3
Sorted array in ascending order is : 1 2 3 4 5
```

Credit goes to

1. Aditya Kumar Yadav **2. [Coding With Clicks]** **3. Chat GPT**

Linked Lists

Linked List is a **linear data structure**, in which elements are not stored at a contiguous(adjacent) location, rather they are **linked using pointers.**

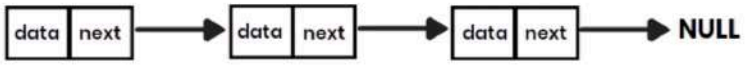


Key components

1. **Node:** Each element in a linked list is represented by a node. A node contains two parts: the data and pointer to the next node.
2. **Head:** The head of the linked list points to the first node. It provides the starting point for accessing the elements in the list.
3. **Tail:** The tail of the linked list points to the last node. It helps in efficient insertion at the end of the list.

Linked List Types

1. **Single Linked List:** In a singly linked list, each node contains a pointer to the next node in the sequence. Traversing a singly linked list is done in a forward direction.



2. **Double Linked List:** In a doubly linked list, each node contains pointers to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it requires additional memory for the backward pointer.

3. **Circular Linked List:** In a circular linked list, the last node points back to the first node. It can be either singly or doubly linked.

Linked List Operations

1. **Insertion:** Add a new node to the linked list. Insertion can be performed at the beginning, end, or any position within the list.
2. **Deletion:** Removing the node from the linked list. Deletion can be performed at the beginning, end, or any position within the list.
3. **Traversing:** Traversing a linked list involves visiting each node of the list from from the beginning to the end. Traversing can be done in both the forward direction or backward direction (double linked list).
4. **Searching:** Searching for a specific value in the linked list. In searching, we match each element of the list with the given element. If the element is found on any of the locations then the location of that element is returned otherwise null is returned.

Why use Linked List over Array?

Array Limitations:

* The size of array must be known in advance before using it in the program.
* The size of an array is fixed once it's created, meaning you cannot directly increase or decrease the size of the array after it has been initialized.
* All the elements in the array need to be contiguously stored in the memory. When you insert a new element into an array at a specific position, all the elements that come after the insertion point need to be shifted to make room for the new element.

Using linked list is useful because,

* It uses dynamic memory allocation, "dynamic" refers to something that is not fixed or static, but rather can change or be adjusted as needed during program execution. Dynamic memory allocation, refers to the process of allocating memory for data structure (linked list) at runtime, rather than at compile time.

  allocate / deallocate

  Memory Allocation: reserving memory for program use

  Memory Deallocation: releasing memory that is no longer needed

Using linked list is useful because,

* Linked lists do not have a fixed size like arrays. There's no need to define the size of the linked list at the time of declaration. The size of a linked list grows and shrinks dynamically based on the number of nodes added or removed from the list.

* The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the pointers needed to be updated.

* As Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.

* Various dynamic data structures like a stack, queue, graph, hash maps, etc can be implemented using a linked list.

Array Data Structure

* An array is a linear data structure in which elements are stored at adjacent contiguous locations, one after another.

* Arrays use static memory allocation, where memory is allocated at compile time and remains fixed throughout the program's execution.

* In arrays, elements are stored in contiguous memory locations, so each element can be accessed directly using its index number.

* Because of this, accessing an element in an array by its index is a constant - time operation, meaning it takes the same amount of time regardless of the size of the array. This time complexity is denoted as O(1).

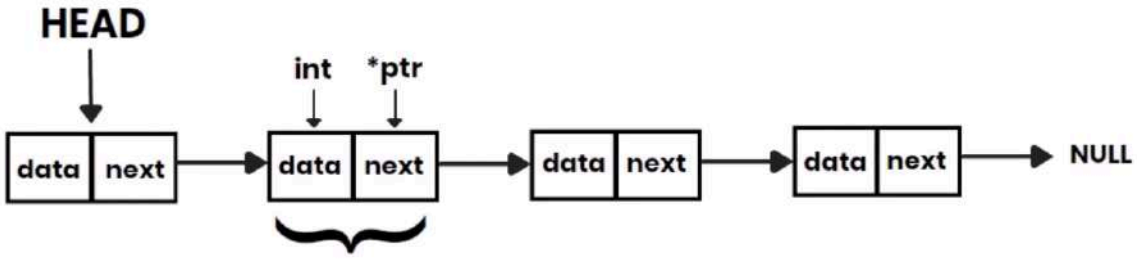
Drawbacks of Linked List

* In linked lists, elements (nodes) are not stored contiguously in memory. To access an element in a linked list, you need to start from the head (or tail) of the list and traverse through the list, until you reach the desired element. The time taken to access an element in a linked list is proportional to the size of the list. In worst case, where you need to traverse the entire list to reach the desired element, the time complexity is O(n), where n is the number of elements in the list.

* Linked lists require additional memory to store pointers for linking nodes, which can increase memory usage compared to arrays.

1. Single Linked List

- A **singly linked list** is a data structure that consists of a sequence of elements, called nodes, where each node contains two parts: data and a pointer to the next node in the sequence.
- In a singly linked list, each node points only to the next node in the list, and the last node points to NULL to indicate the end of the list.



- Each node contains two data types variables : 1st **integer or character or any other variable** and 2nd part of the node contains a **pointer variable** .

1: Traversal Operation

Traversal in a **linked list** means visiting each node and accessing its data. To perform traversal, we use a pointer called **temp** , which initially points to the head node. Then, we use a while loop to go through the list until **temp** becomes NULL (indicating the end of the list).

1. Start with **node *temp = head;**
2. while (**temp != NULL**) :
    - Access the data of the current node.(**cout << temp -> data << " | ";**)
    - Move current to the next node (**temp = temp ->next ;**)

This way, you can visit and process each node in the linked list.

```cpp
#include <iostream>
using namespace std ;
struct Node
{
    int data;
    Node *next;
};
Node *head = NULL;
void insert (int n)
{
    Node *newNode = new Node;
    newNode -> data = n;
    newNode -> next = head;
    head = newNode;
}
void print()
{
    cout << "Data elements in a single linked list : " ;
    Node *temp = head;
    while (temp!= NULL)
    {
        cout << temp -> data << " | ";
        temp = temp -> next;
    }
}
int main ()
{
    insert (1);
    insert (3);
    insert (5);
    insert (7);
    insert (9);
    insert (10);
    print();
}
```

```
PS C:\Users\Aditya Yadav\Desktop\CODES\DSA> cd "c:\Users\Aditya Yadav\
\17SingleLinkedList }
Data elements in a single linked list : 10 | 9 | 7 | 5 | 3 | 1 |
```

2 : Insertion Operation

Insertion in a linked list means adding a **new node** to the list. In the provided code, each new node is inserted at the beginning of the list.

Steps:

1. Create a new node: Dynamically allocate memory for the new node using `**new node**`.
2. Assign data: Set the new node's data (`**newnode->data = n**`).
3. Link to the list: Point the new node's `**next**` to the current **'head' (newnode->next = head;)**
4. Update head: Make the new node the new `**head**` of the list (`**head = newnode**`).
Each new node becomes the first node in the list.

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

struct node
{
    int data;
    node *next;
};
node *head = NULL;

void insert(int n, int pos)
{
    node *newnode = new node;
    newnode->data = n;
    newnode->next = head;

    if (pos == 1)  // Inserting at the head
    {
        head = newnode;
        return;
    }

    node *temp = head;
    for (int i = 1; i < pos - 1 && temp != NULL; i++)
    {
        temp = temp->next;
    }

    if (temp == NULL)
    {
        cout << "Position is out of bounds." << endl;
        delete newnode;  // Deallocate memory if insertion fails
        return;
    }

```

```
        newnode->next = temp->next;
        temp->next = newnode;
    }

    void print()
    {
        cout << "Data elements in the single linked list: ";
        node *temp = head;
        while (temp != NULL)
        {
            cout << temp->data << " | ";
            temp = temp->next;
        }
        cout << endl;
    }

    int main()
    {
        insert(1, 1);   // Insert 1 at position 1
        insert(3, 2);   // Insert 3 at position 2
        insert(5, 3);   // Insert 5 at position 3
        insert(7, 4);   // Insert 7 at position 4
        insert(9, 5);   // Insert 9 at position 5
        insert(10, 6); // Insert 10 at position 6

        print();

        // Inserting at the 4th position
        insert(4, 4);

        print();  // Print again to show the updated list

        return 0;
    }
```

```
PS C:\Users\Aditya Yadav\Desktop\CODES\DSA> cd "c:\Users\Aditya Yadav\Deskto

Data elements in the single linked list: 1 | 3 | 5 | 7 | 9 | 10 |
Data elements in the single linked list: 1 | 3 | 5 | 4 | 7 | 9 | 10 |
```

3 : Deletion Operation

Deletion in a singly linked list involves removing a node (at the beginning, at the end, or from a specific position) while maintaining the list structure.

Steps:

1. **Deleting from the Beginning:**
    - Point the head to the next node.
    - Free the memory of the removed node.
2. **Deleting from the End:**
    - Traverse to the second last node.
    - Set its `next` to `NULL`.
    - Free the last node.

3. **Deleting from a Specific Position:**
    - Traverse to the node just before the target position.
    - Adjust its `next` pointer to skip the target node.
    - Free the memory of the removed node.

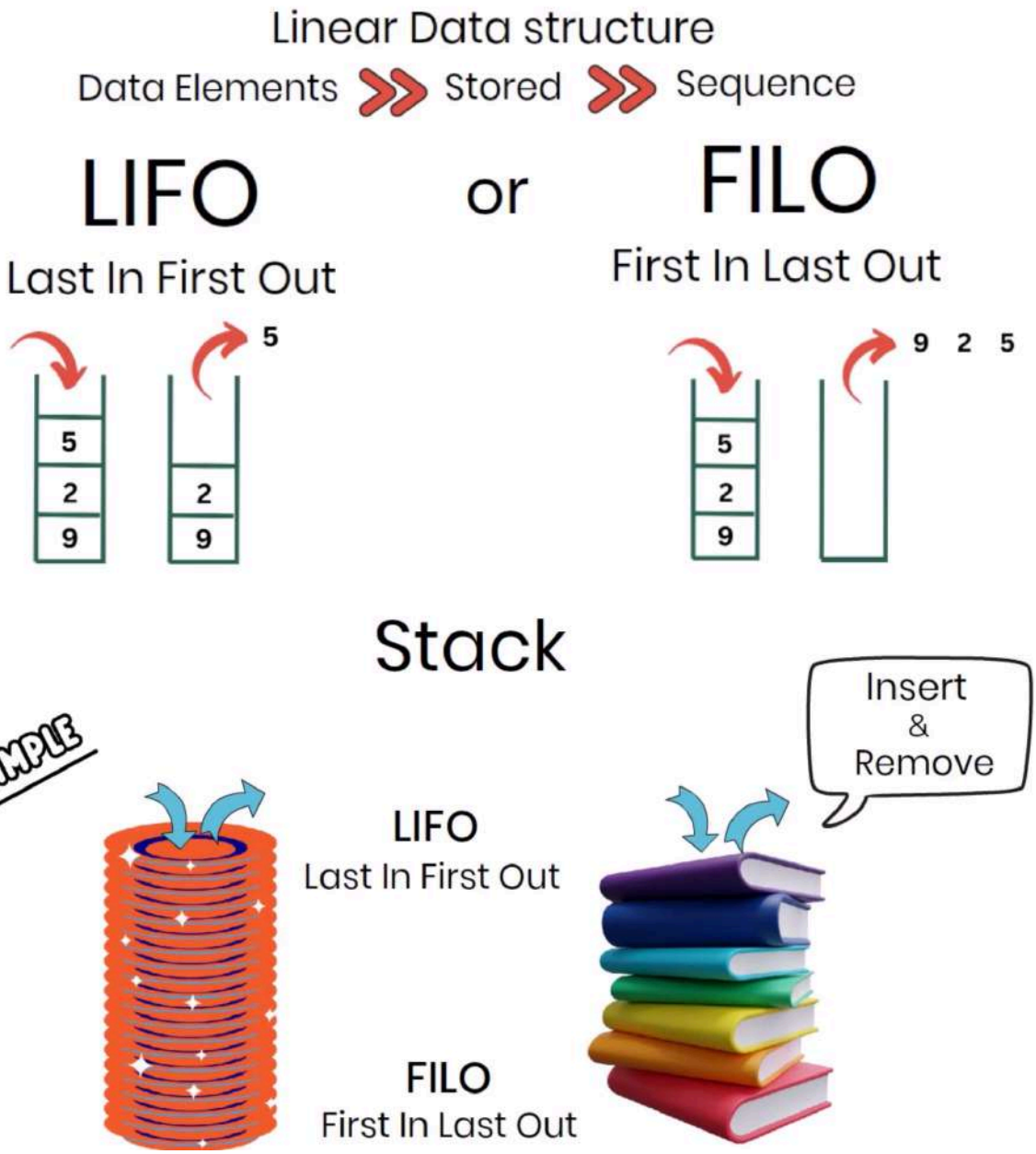
```
Tabnine | Edit | Test | Explain | Document | Ask
void deleteAtPosition(int pos) {
    if (head == NULL) {
        cout << "List is empty." << endl;
        return;
    }

    node *temp = head;

    // If the head needs to be removed
    if (pos == 1) {
        head = temp->next;  // Move head to the next node
        cout << "Deleted element is " << temp->data << " at position number " << pos << endl;
        delete temp;        // Delete the old head
        return;
    }

    // Traverse to the node just before the desired position
    for (int i = 1; i < pos - 1 && temp != NULL; i++) {
        temp = temp->next;
    }

    // If position is out of bounds
    if (temp == NULL || temp->next == NULL) {
        cout << "Position out of bounds" << endl;
    } else {
        node *nodeToDelete = temp->next;
        temp->next = nodeToDelete->next;
        cout << "Deleted element is " << nodeToDelete->data << " at position number " << pos << endl; // Print deleted element
        delete nodeToDelete; // Delete the node
    }
}
```

Credit goes to

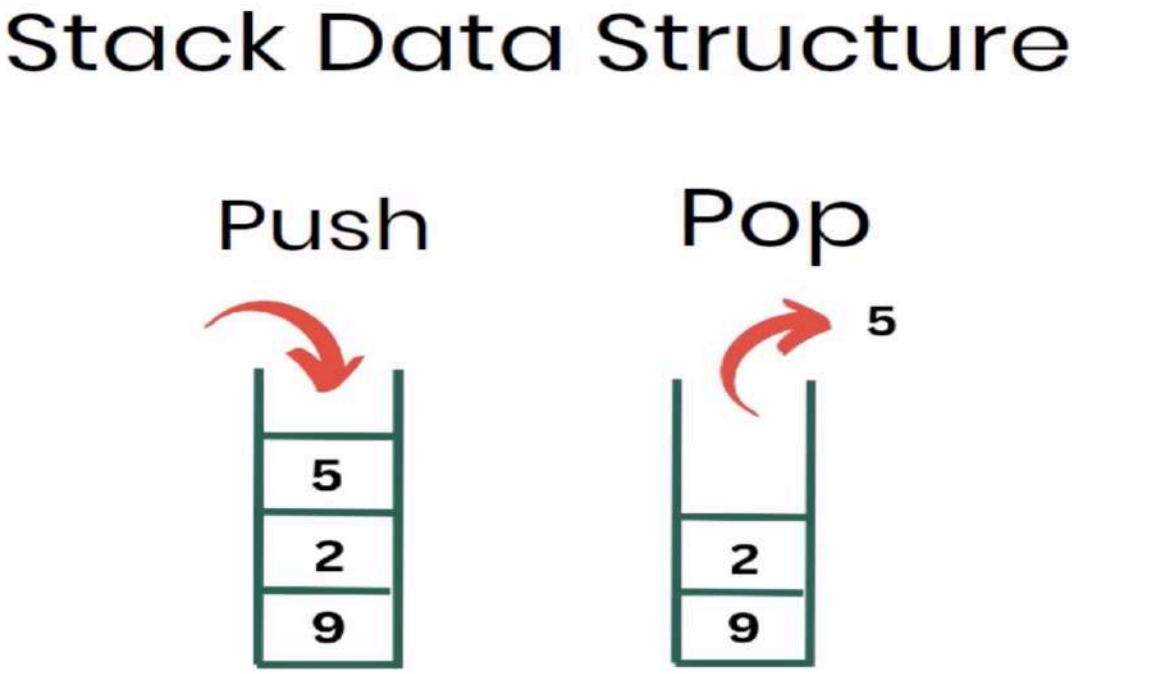
1. Aditya Kumar Yadav **2. [Coding With Clicks]** **3. Chat GPT**

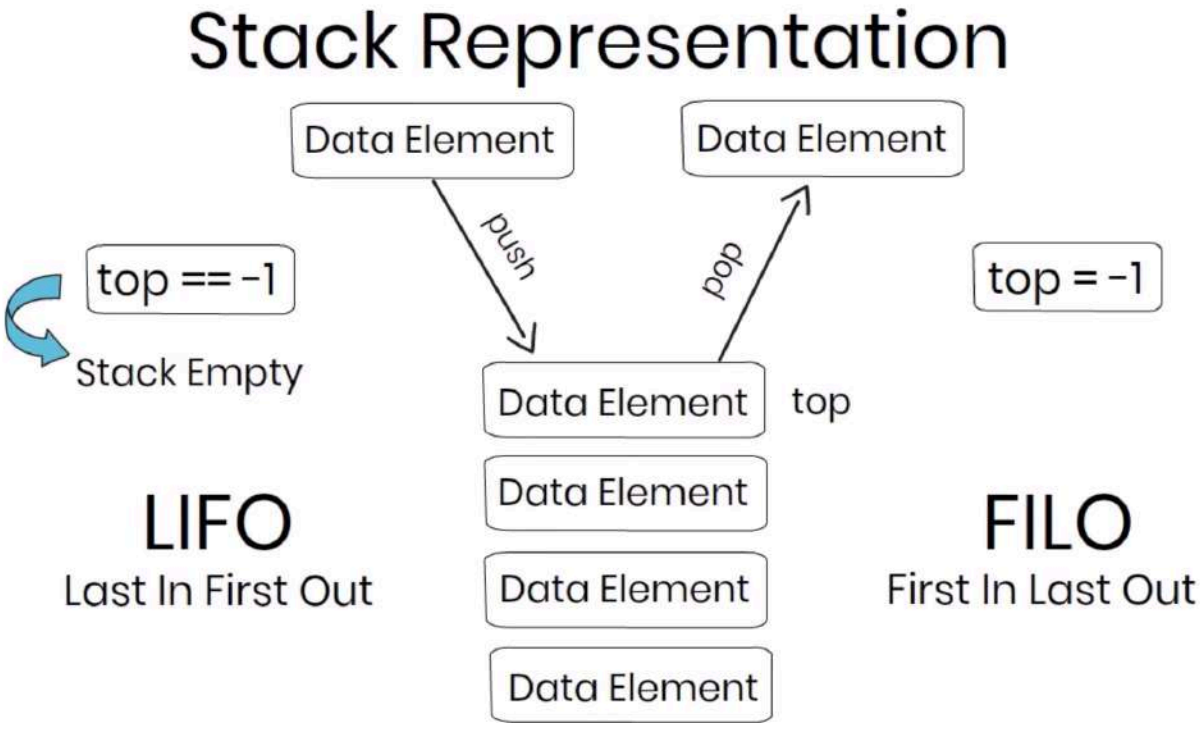
STACK

A **stack** is a linear data structure that follows a strict order for adding and removing elements. It operates on a **LIFO** (Last-In-First-Out) principle . The element you add last (**push**) is the first one you can remove (**pop**). This means you can only access and modify elements from one designated end, called the "**top**" of the stack.



- Adding some elements to the stack is called Push and removing any element is called Pop.





STACK OPERATIONS

1. Push
2. Pop
3. Peek
4. isFull
5. isEmpty

1: Push Operation

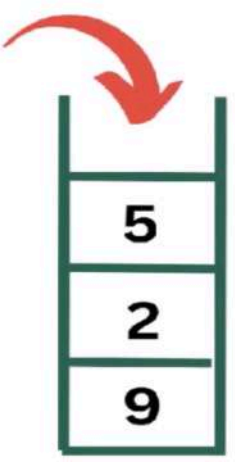
Step 1: First, check whether or not the stack is full ⟶ isFull

Step 2: If stack is full, then exit

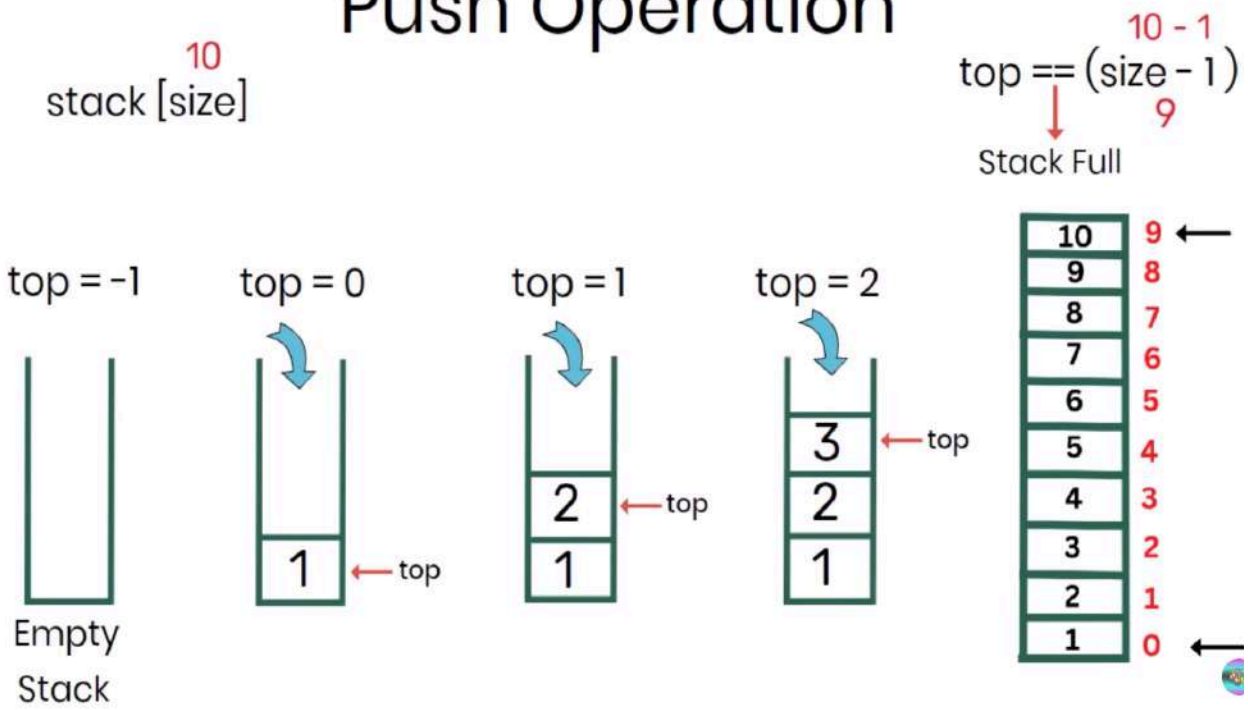
Step 3: If stack is not full, increment the top pointer by one

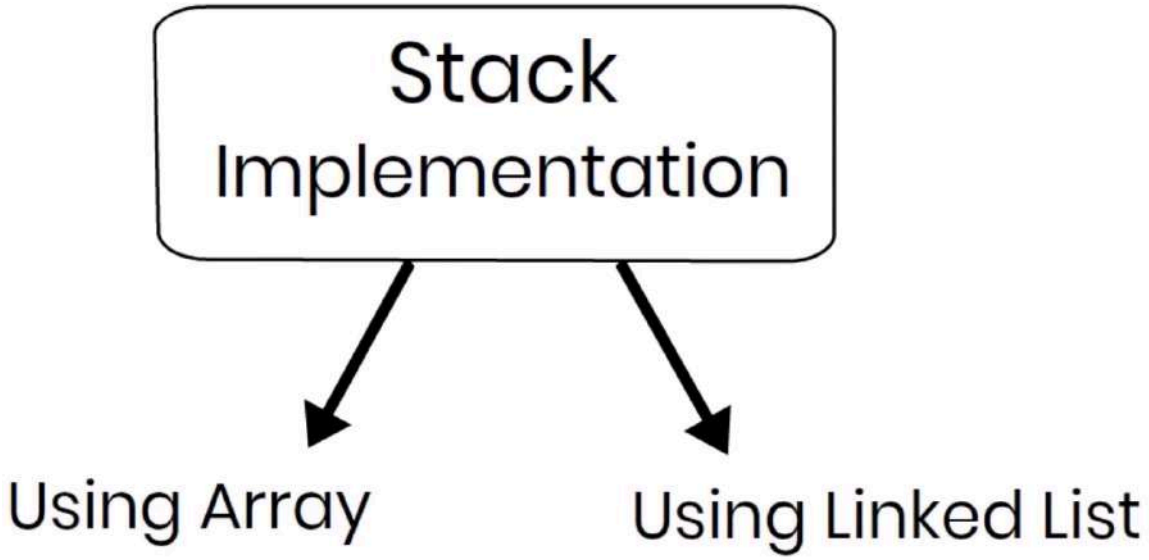
Step 4: Insert the element where the top pointer is pointing

Insert elements in the stack



Push Operation





push.cpp > main()

```cpp
#include <iostream >
using namespace std;
int stack[5], n = 5 , top = -1;
void push (int x)
{
  if (top == n-1)
    cout <<"Stack is full! \n";
 else
    {
        top++;
        stack[top] = x;
    }
}
void display ()
{
    if (top>=0)
    {
        cout <<"Stack elements are: \n";
        for (int i=top; i>=0; i--)
            cout << stack[i] <<endl;
    }
    else
    cout <<"Stack is empty!";
}
int main ()
{
    push(1);
    push(2);
    push(3);
    push(4);
    push(5);
    push(6);
    display();
}
```

```
Stack is full!
Stack elements are:
5
4
3
2
1
```

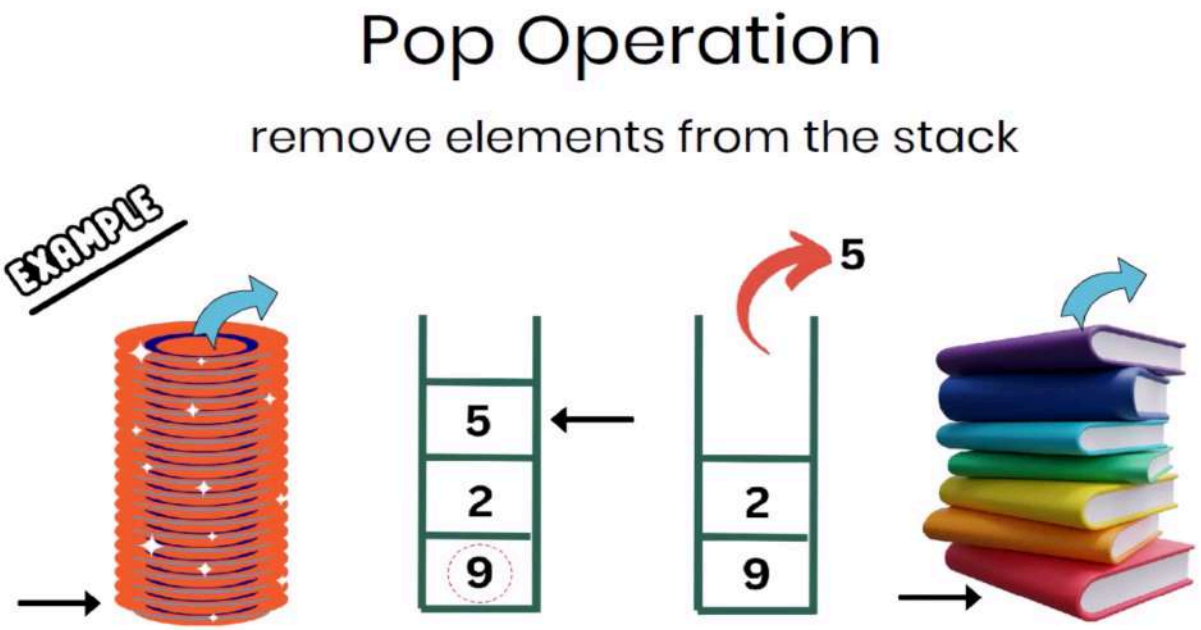
2: Pop Operation

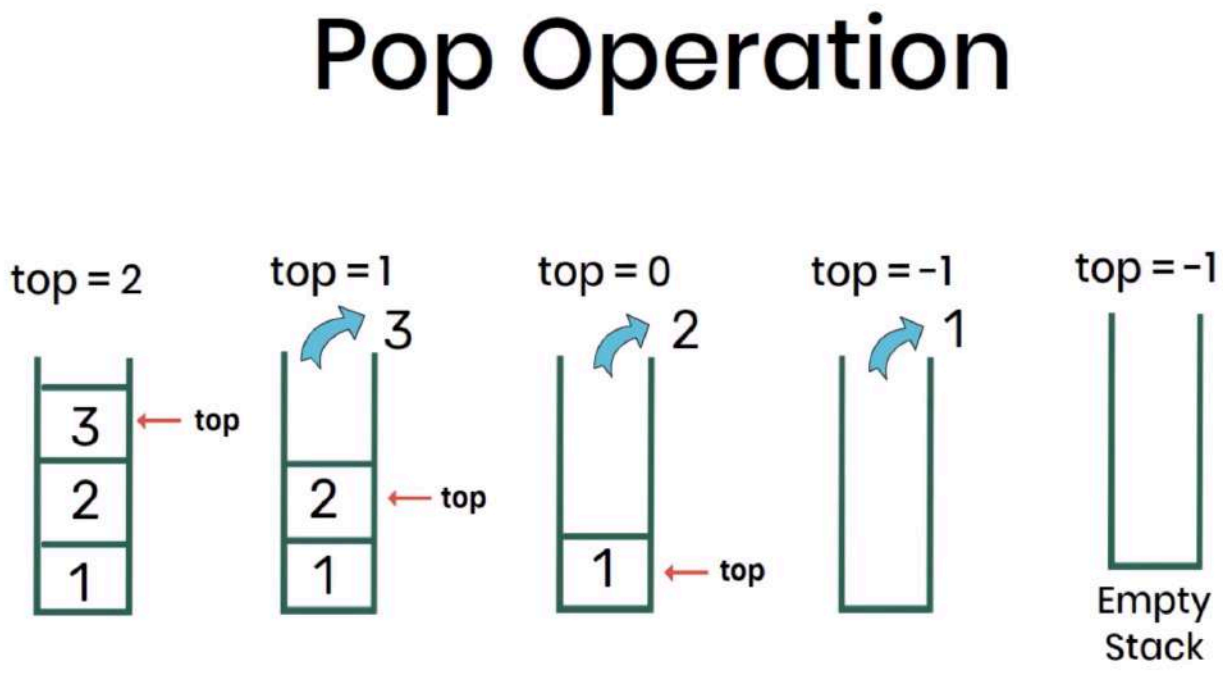
Step 1: First, check whether or not the stack is empty ⟶ isEmpty

Step 2: If stack is empty, then exit

Step 3: If stack is not empty, remove topmost data element

Step 4: Decrement the top pointer by one





14pop.cpp

: > Users > Aditya Yadav > Desktop > CODES > DSA > 14pop.cpp > pop()

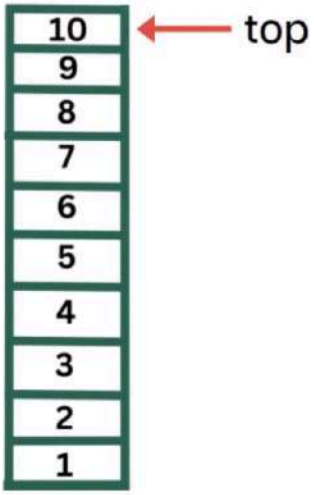
```cpp
#include <iostream>
using namespace std;
int stack[5], n = 5 , top = -1;

//PUSH OPERATION
void push ( int x)
{
    if (top == n-1)
    cout << "Stack is full \n";
    else
    {
        top++;
        stack[top] = x;
    }
}

//POP OPERATION
void pop()
{
    if(top == -1)
    cout << "Stack is empty\n";
    else
    {
        cout << "Popped element is " << stack[top] << endl;
        top--;
    }

}
    void display()
    {
        if(top >= 0)
        {
            cout << "Stack elements are: \n";
            for(int i=top; i>=0; i--)
            cout << stack[i] <<endl;
        }
        else
            cout << "Stack is empty!" << endl;
    }
int main()
{
    push(10);
    push(20);
    push(30);
    display();
    pop();
    pop();
    display();
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
Warning: PowerShell detected that you might be using a screen read

PS C:\Users\Aditya Yadav> cd "c:\Users\Aditya Yadav\Desktop\CODES\
Stack elements are:
30
20
10
Popped element is 30
Popped element is 20
Stack elements are:
10
```

Peek Operation

Get the top element of the stack, without removing it.




```cpp
#include <iostream>
using namespace std;
int stack[5], n=5 , top = -1;
//PUSH OPERATION
void push ( int x)
{
    if (top==n-1)
    cout << "Stack is full \n";
    else
    {
        top++;
        stack[top] = x;
    }
}
//POP OPERATION
void pop()
{
    if(top== -1)
    cout << "Stack is empty\n";
    else
    {
        cout << "Popped element is " << stack[top] << endl;
        top--;
    }
}

//PEEK OPERATION
int peek()
{
    return stack[top];
}
void display()
{
    if (top>=0)
    {
        cout << "Stack elements are: \n";
        for (int i=top; i>=0; i--)
```

```
38          cout << stack[i] <<endl;
39      }
40      else
41      cout << "Stack is empty\n";
42  }
43  int main ()
44  {
45      push(1);
46      push(2);
47      push(3);
48      push(4);
49      push(5);
50      display();
51      pop();
52      pop();
53      pop();
54      display();
55      int y = peek();
56      cout << "Top element of the stack is : " << y << endl;
57  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

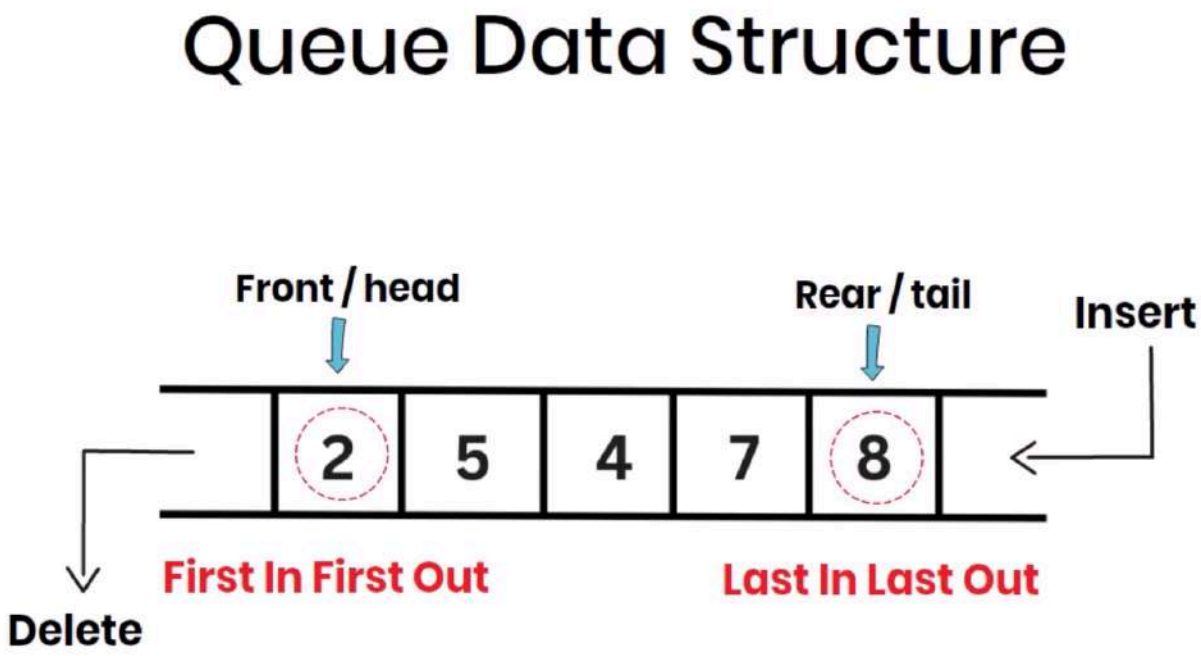
```
Stack elements are:
5
4
3
2
1
Popped element is 5
Popped element is 4
Popped element is 3
Stack elements are:
2
1
Top element of the stack is : 2
```

Credit goes to

1. Aditya Kumar Yadav **2. Coding With Clicks** **3. Chat GPT**

QUEUE

A **Queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle.



A queue typically supports the following operations:

1. **Enqueue**: Adds an element to the end of the queue.
2. **Dequeue**: Removes and returns the element from the front of the queue.
3. **Peek/Front**: Returns the element at the front of the queue without removing it.
4. **IsEmpty**: Checks if the queue is empty.
5. **IsFull** (for bounded queues): Checks if the queue is full.
6. **Size**: Returns the number of elements in the queue.

1: Enqueue Operation

Enqueue is an operation in a queue data structure that adds an element to the end (rear) of the queue.

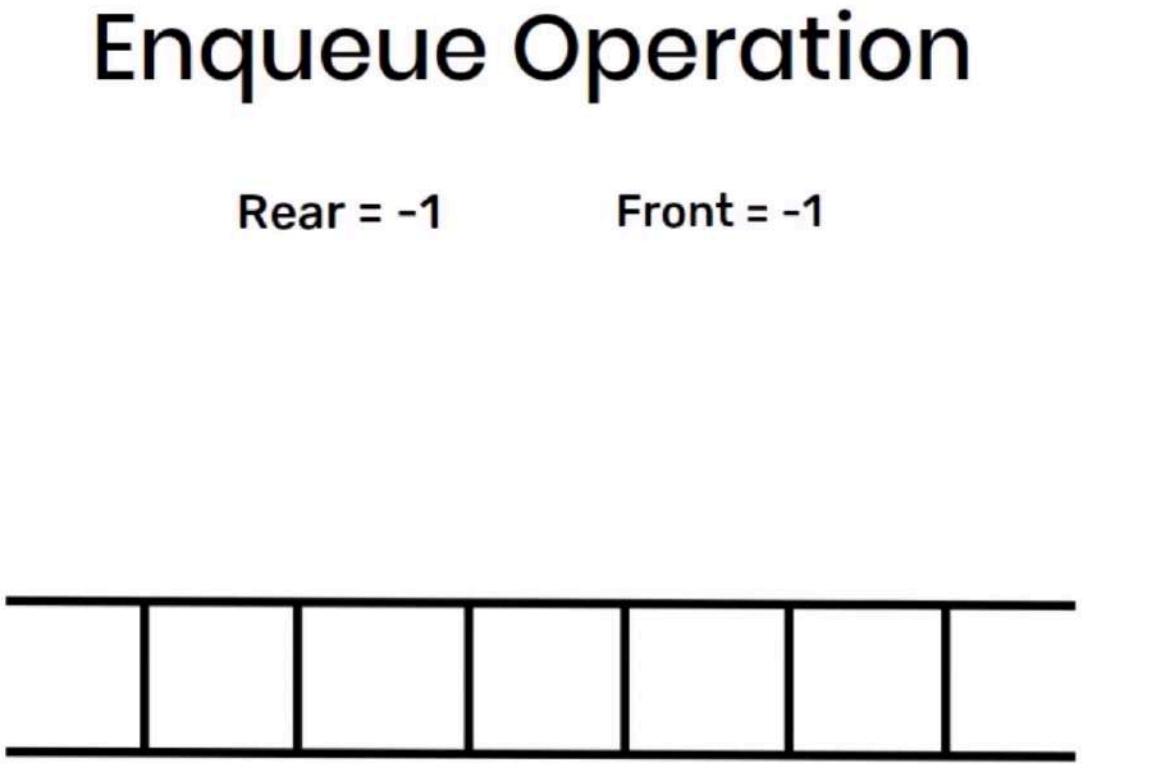
Step1: Check if the Queue is full ⟶ isFull

Step2: If Queue is full, then exit

Step3: If not, increment the rear pointer by one

Step4: Insert the element where the rear pointer is pointing

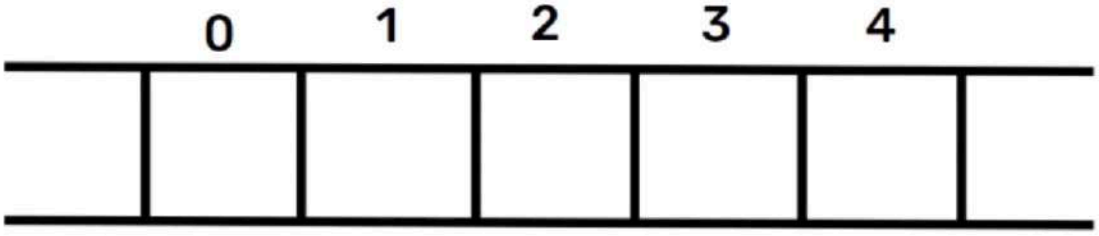
Step5: Success



Rear = -1 Front = -1

-1 5 - 1

rear == size - 1

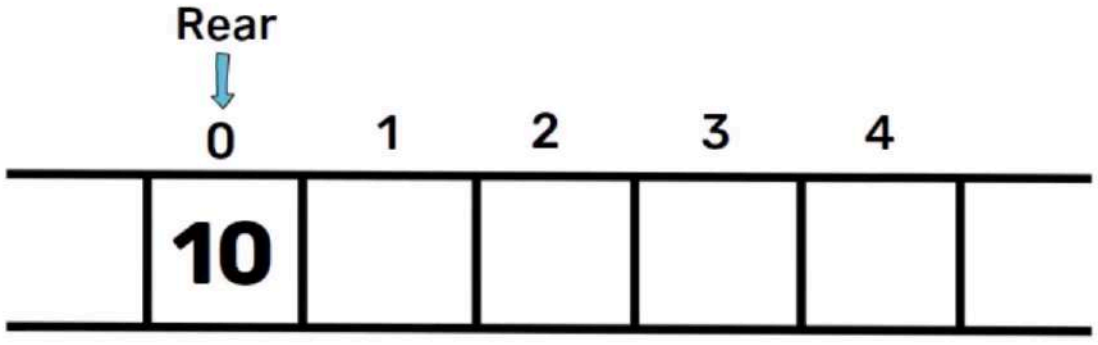


size = 5

Rear = -1 Front = -1

-1 5 - 1

rear == size - 1

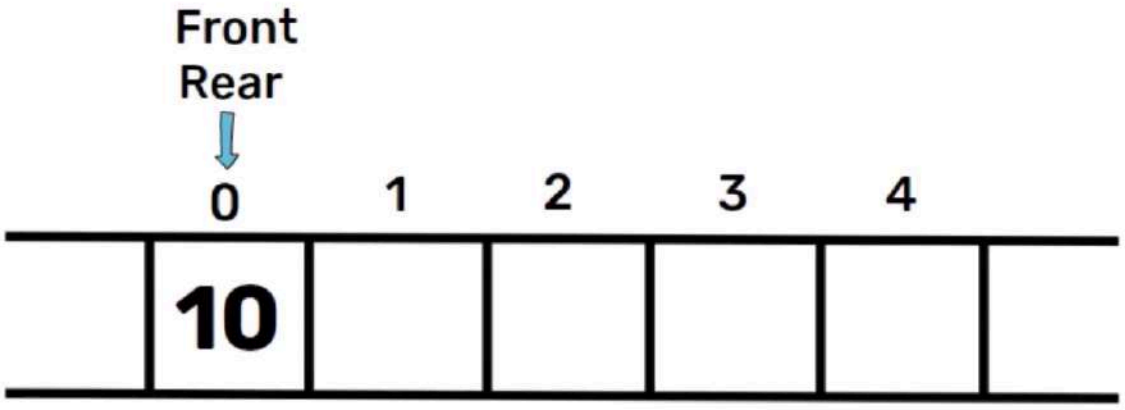


size = 5

Rear = -1 Front = -1

-1 5 - 1

rear == size - 1

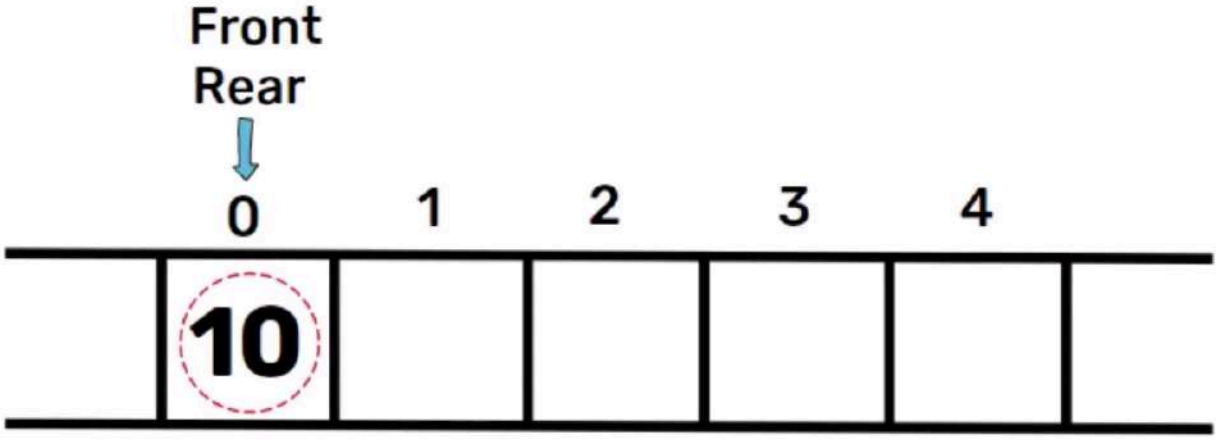


size = 5

Rear = -1 Front = -1

0 5 - 1

rear == size - 1

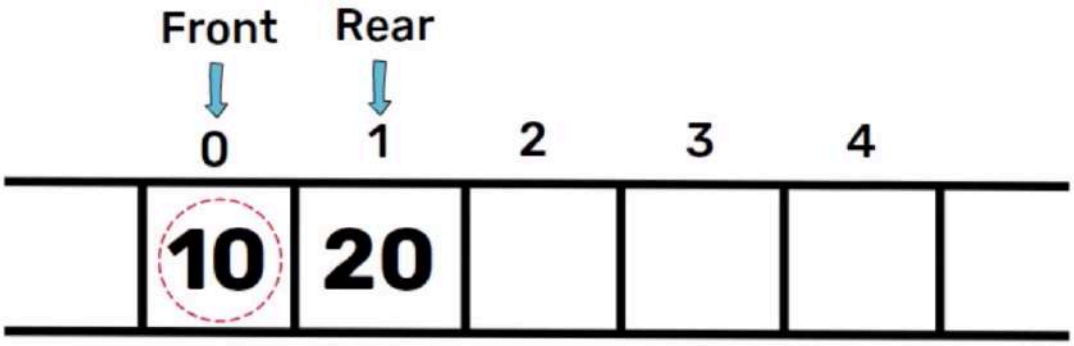


size = 5

Rear = -1 Front = -1

0 5 - 1

rear == size - 1

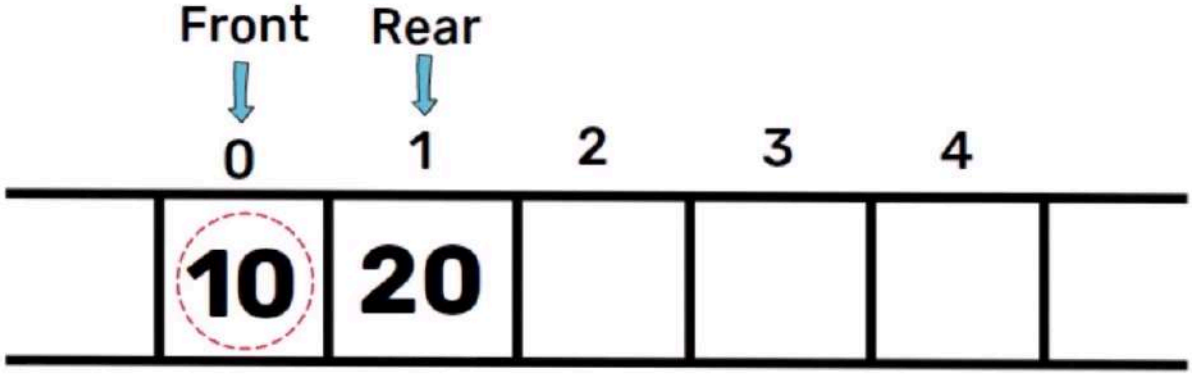


size = 5

Rear = -1 Front = -1

1 5 - 1

rear == size - 1

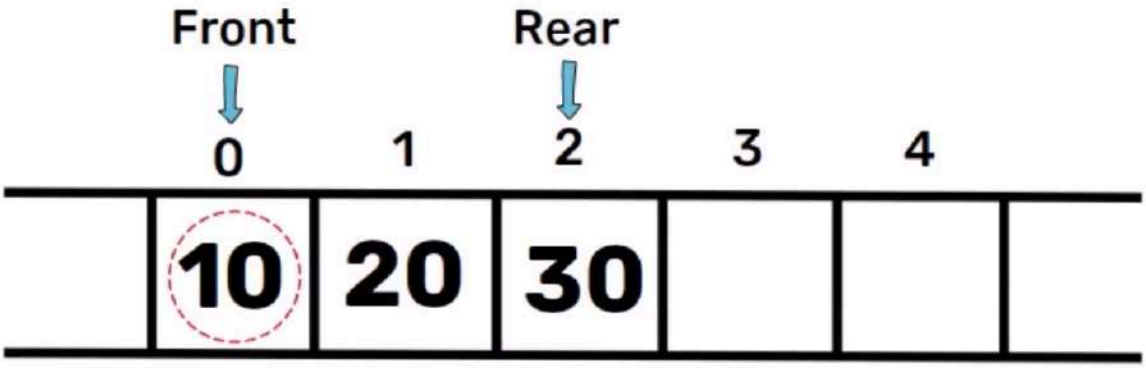


size = 5

Rear = -1 Front = -1

1 5 - 1

rear == size - 1

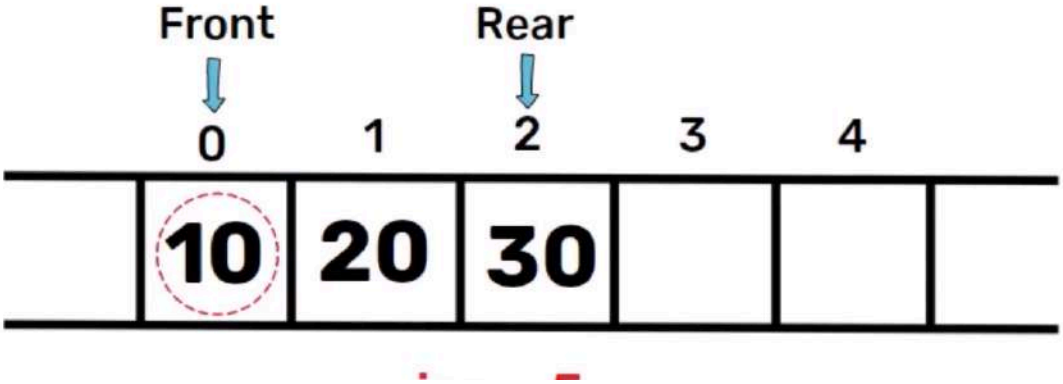


size = 5

Rear = -1 Front = -1

2 5 - 1

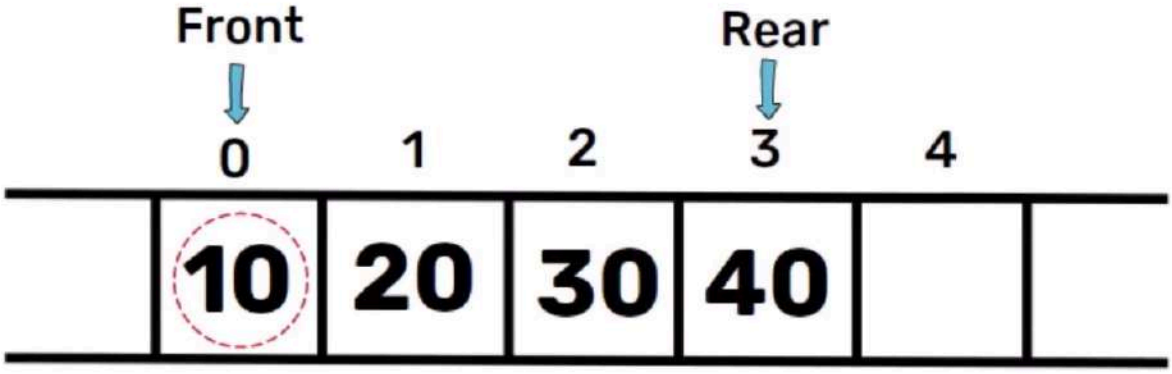
rear == size - 1



Rear = -1 Front = -1

2 5 - 1

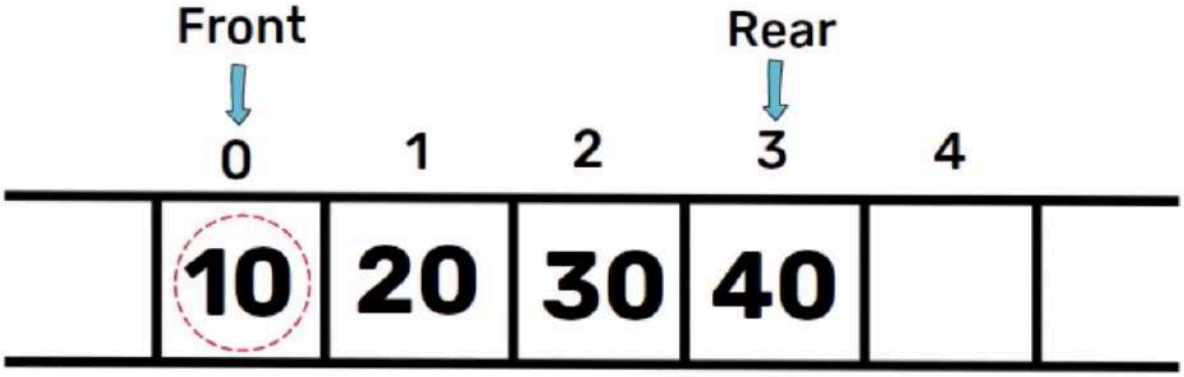
rear == size - 1



Rear = -1 Front = -1

3 5 - 1

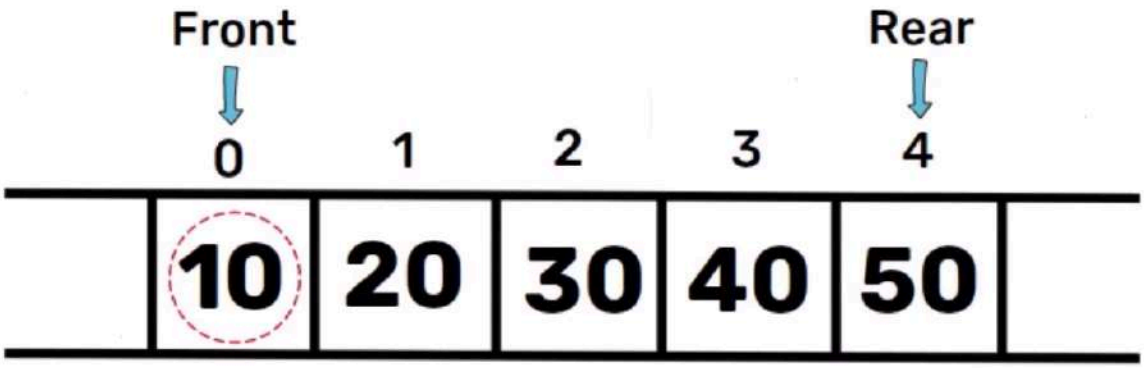
rear == size - 1



Rear = -1 Front = -1

3 5 - 1

rear == size - 1

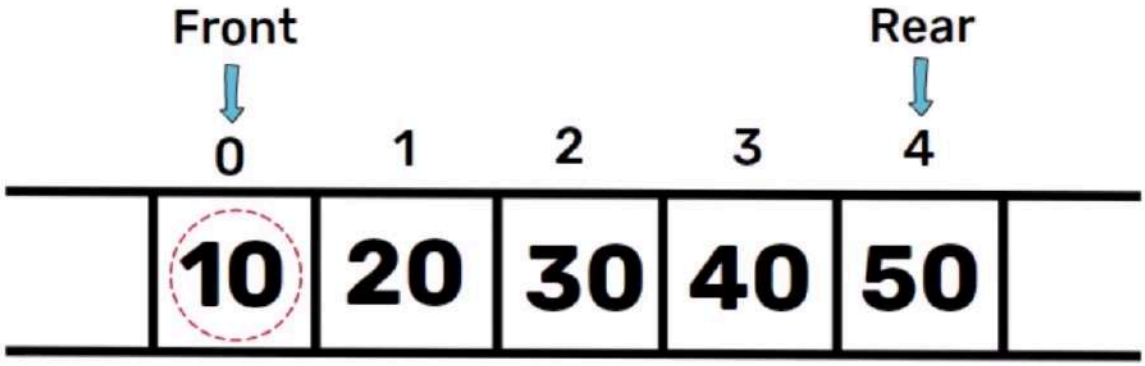


Enqueue Operation

Rear = -1 Front = -1

4 5 - 1

rear == size - 1



size = 5

2: Dequeue Operation

The **dequeue operation** is the process of removing an element from a data structure called a **queue**. A queue follows the **FIFO (First In, First Out)** principle, meaning the first element added is the first one to be removed.

Step1: Check if the Queue is empty ⟶ isEmpty

Step2: If Queue is empty, then exit

Step3: If not, access the element where the front pointer is pointing

Step4: Increment the front pointer by one

Step5: Success

3: Peek Operation

Access the data element where the front is pointing without removing from the queue.

Step1: Check if the queue is empty.

Step2: If the queue is empty, return "Queue is Empty."

Step3: If the queue is not empty, access the data where the front pointer is pointing.

Step4: Return data element.

Code of all operation in queue :

Enqueue , Dequeue , Peek and Display

16-Queue.cpp > main()

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

int queue[10], n, front = -1, rear = -1;

void enqueue(int x) {
    if (rear == n - 1) {
        cout << "Queue is Full !!!\n";
    } else {
        if (front == -1) front = 0; // Initialize front on first enqueue
        rear++;
        queue[rear] = x;
    }
}

void dequeue() {
    if (front == -1 || front > rear) {
        cout << "Queue is Empty !!!\n";
    } else {
        cout << "Dequeued element is: " << queue[front] << endl;
        front++;
    }
}

void peek() {
    if (front == -1 || front > rear) {
        cout << "Queue is Empty !!!\n";
    } else {
        cout << "Front element is: " << queue[front] << endl;
    }
}

void display() {
    if (front == -1 || front > rear) {
        cout << "Queue is Empty !!!\n";
    } else {
        cout << "Queue elements are: \n";
        for (int i = front; i <= rear; i++) {
            cout << queue[i] << " | ";
        }
        cout << endl;
    }
}

int main() {
    int choice, y;
    cout << "Enter Size of Queue: ";
    cin >> n;

    do {
        cout << "Choose an operation: 1. ENQUEUE 2. DEQUEUE 3. PEEK 4. DISPLAY 5. EXIT : - ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter element to ENQUEUE: ";
                cin >> y;
                enqueue(y);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                peek();
                break;
            case 4:
                display();
                break;
            case 5:
```

```
            case 5:
                cout << "Exiting...\n";
                break;
            default:
                cout << "Invalid choice...\nRe-enter the choice:\n";
        }
    } while (choice != 5);

    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
Enter Size of Queue: 4
Choose an operation: 1. ENQUEUE 2. DEQUEUE 3. PEEK 4. DISPLAY 5. EXIT : - 1
Enter element to ENQUEUE: 11
Choose an operation: 1. ENQUEUE 2. DEQUEUE 3. PEEK 4. DISPLAY 5. EXIT : - 1
Enter element to ENQUEUE: 22
Choose an operation: 1. ENQUEUE 2. DEQUEUE 3. PEEK 4. DISPLAY 5. EXIT : - 1
Enter element to ENQUEUE: 33
Choose an operation: 1. ENQUEUE 2. DEQUEUE 3. PEEK 4. DISPLAY 5. EXIT : - 1
Enter element to ENQUEUE: 44
Choose an operation: 1. ENQUEUE 2. DEQUEUE 3. PEEK 4. DISPLAY 5. EXIT : - 2
Dequeued element is: 11
Choose an operation: 1. ENQUEUE 2. DEQUEUE 3. PEEK 4. DISPLAY 5. EXIT : - 4
Queue elements are:
22 | 33 | 44 |
Choose an operation: 1. ENQUEUE 2. DEQUEUE 3. PEEK 4. DISPLAY 5. EXIT : - 3
Front element is: 22
Choose an operation: 1. ENQUEUE 2. DEQUEUE 3. PEEK 4. DISPLAY 5. EXIT : -
```

Credit goes to

1. Aditya Kumar Yadav **2. [Coding With Clicks]** **3. Chat GPT**

Tree & Graph

Trees: A hierarchical structure where data is connected like branches of a tree. Each node can have child nodes.
Binary Trees: A type of tree where each node has at most two children (left and right).
Binary Search Trees (BST): A binary tree where the left child has smaller values and the right child has larger values than the parent.
Recursive Traversal: A way to visit tree nodes using a function that calls itself (e.g., in-order, pre-order, post-order).
Non-Recursive Traversal: Visiting tree nodes using loops and stacks instead of recursion.
Searching in BST: Locating a value by repeatedly moving left or right based on comparisons.
Insertion in BST: Adding a value while maintaining the tree's sorted order.
Deletion in BST: Removing a value and rearranging the tree to keep its structure.
Graphs: A collection of nodes (vertices) connected by edges, used to represent relationships or networks.
Graph Terminology: Includes terms like vertex (node), edge (connection), and degree (number of edges a node has).
Depth First Search (DFS): A graph traversal method that explores as far as possible in one path before backtracking.
Breadth First Search (BFS): A graph traversal method that explores all neighboring nodes level by level.

TREE

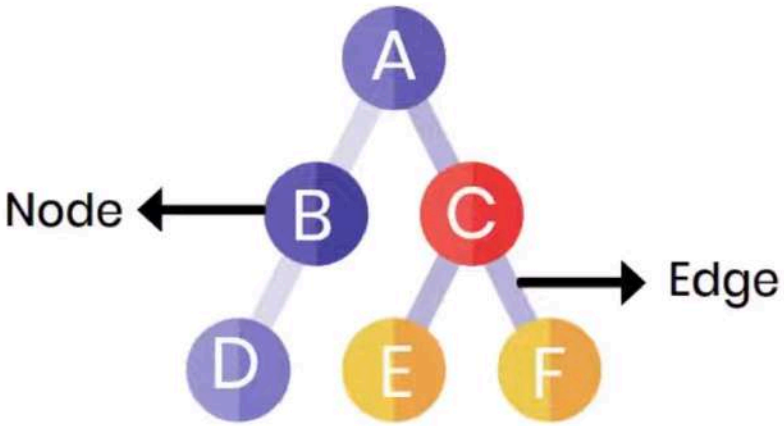
A **tree** is a **hierarchical data structure** consisting of nodes connected by edges. It begins with a **root node** and branches out to **child nodes,** with each node having only one parent (except the root). Trees allow efficient data storage, organization, and retrieval, making them ideal for hierarchical data. Common types include

- Binary Trees
- Binary Search Trees
- AVL Trees
- B-Trees

Tree

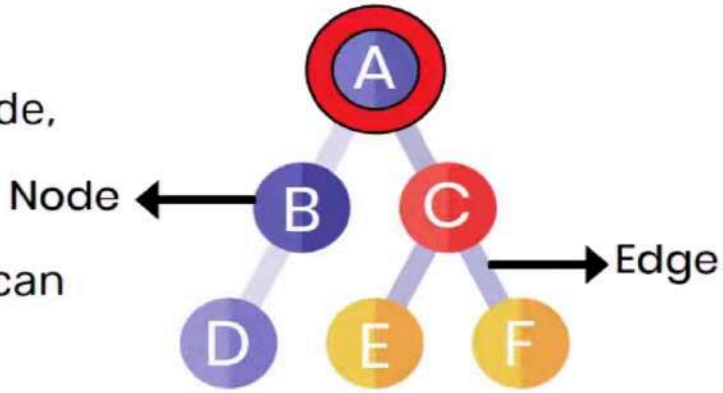
Non Linear Data structure

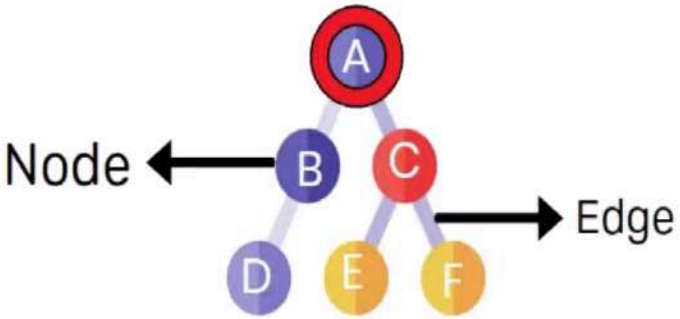
Data Elements >> Stored >> Hieracrchical manner



Properties

1. The tree has one node called root node, it does not have any parent.
2. Each node has one parent only, but can have multiple children.
3. Each node is connected to its children via edge.

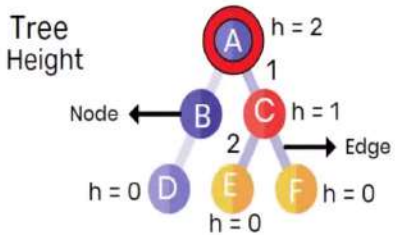




	D E F
Leaf Node	which does not have any child node
Edge	is a connection between two nodes.
	B & C E & F
Siblings	Nodes with the same parent
	A - C - E
Path / Traversing	number of successive edges from source node to destination node

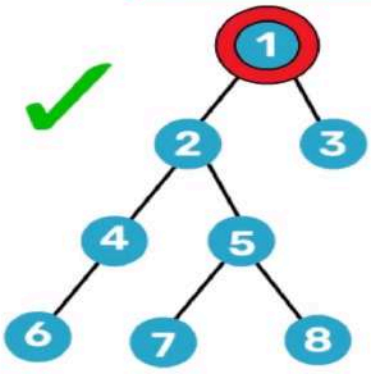
	A
Root Node	First node of the tree, it does not have any parent.
	A B C
Parent Node	is an immediate predecessor of a node
	B C D E F
Child Node	is an immediate successor of a node
Height of a Node	represents the number of edges on the longest path between that node and a leaf
Level / Depth of a Node	number of edges from the root node to the node.
Degree of a Node	represents the number of children of a node
Subtree	descendents of a node represent subtree.

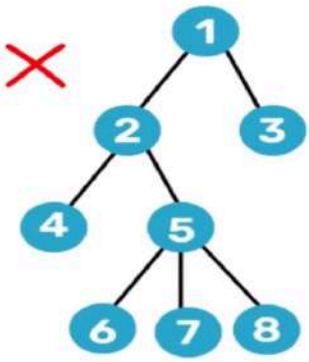
Height of a Node



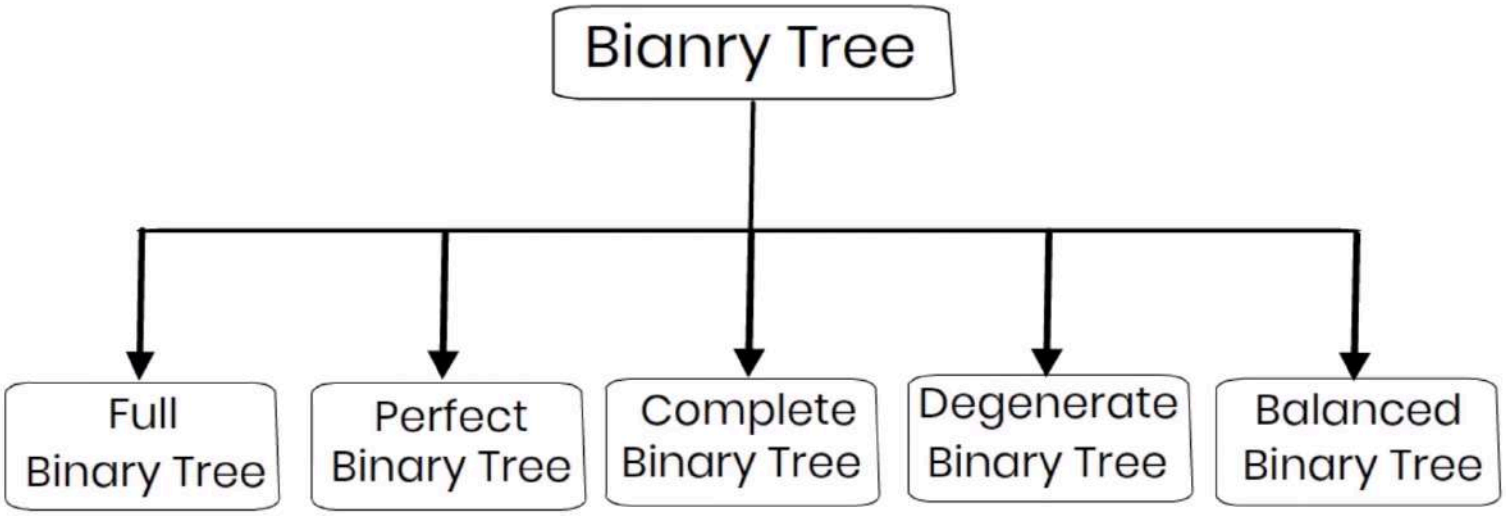
1.Binary Tree

In a binary tree, every node can have at most 2 children, left and right.



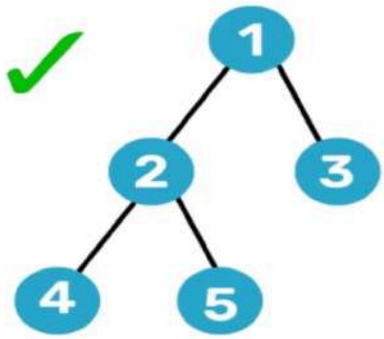


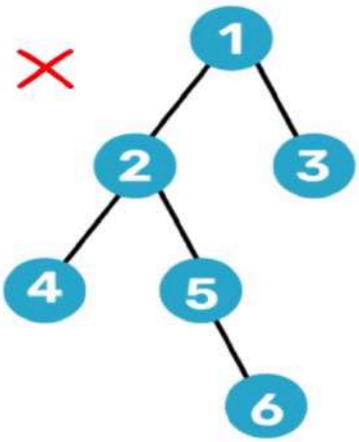
* 0 child node
* 1 child node
* 2 child nodes



1. Full Binary Tree

In Full Binary Tree, every node has either 0 or 2 Children.

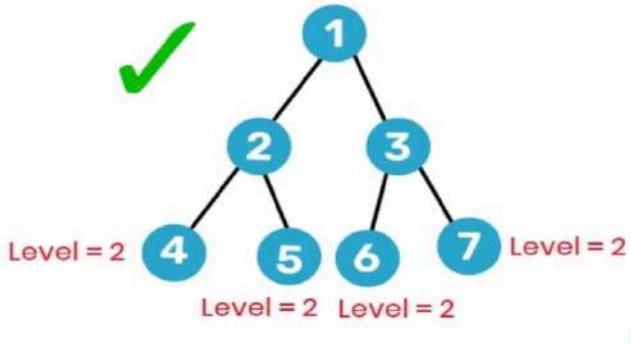


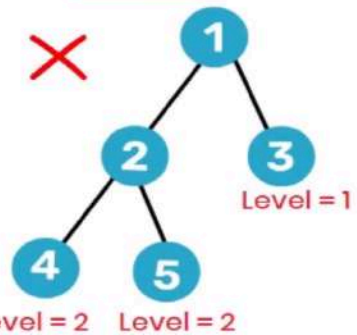


* 0 child node
* 2 child node

2. Perfect Binary Tree

In perfect binary tree, in which every internal node (Parent node) has exactly two child nodes, and all the leaf nodes (External node) are at the same level. (depth)

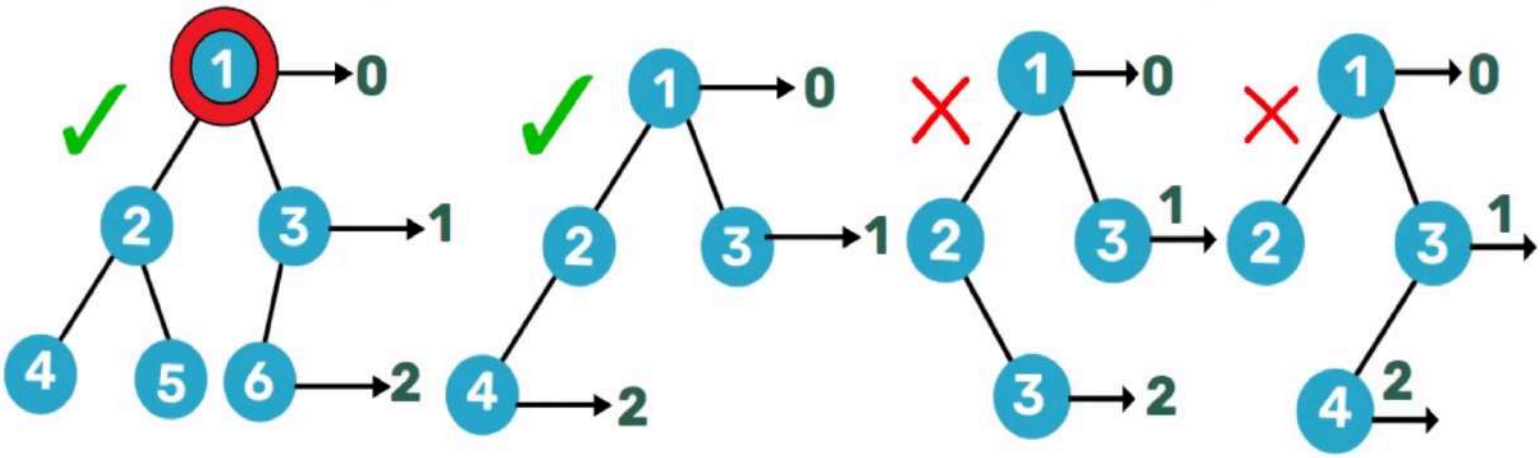




* 2 child nodes
* All leaf nodes (level same)

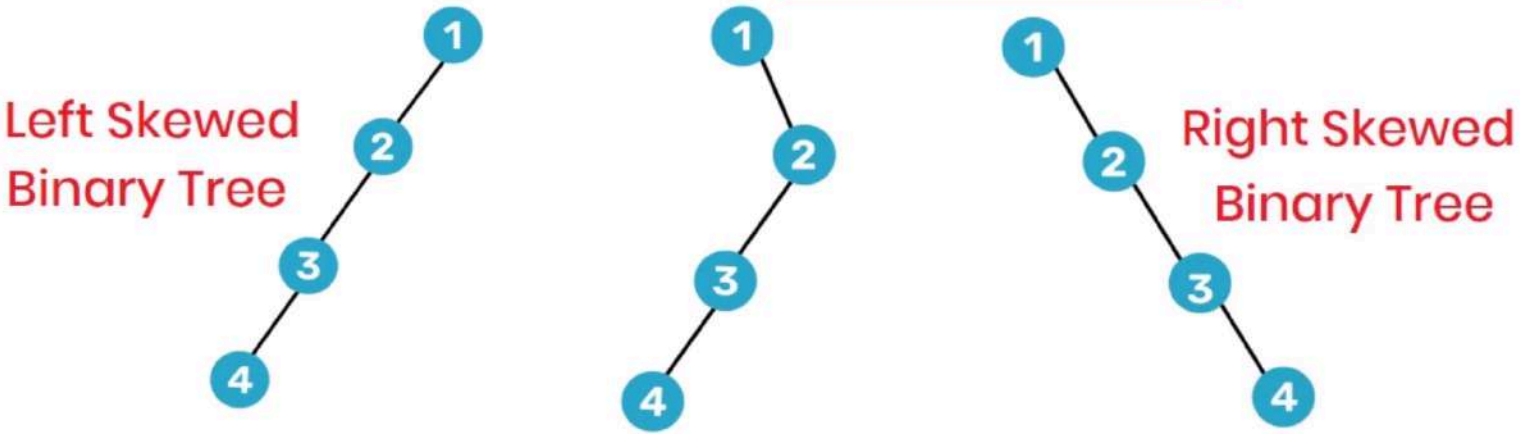
3. Complete Binary Tree

In complete binary tree, all the levels are completely filled with nodes except the last level, in which nodes should be added from from the left side.



4. Degenerate / Pathological Binary Tree

In degenerate binary tree, every internal node has only a single child, either left or right.

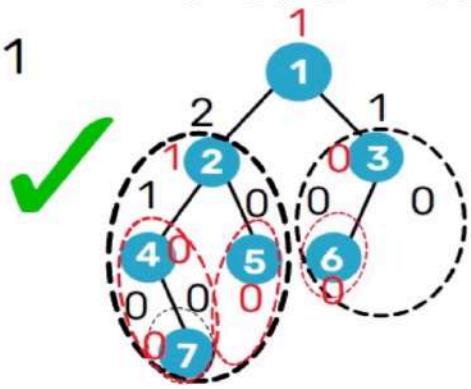


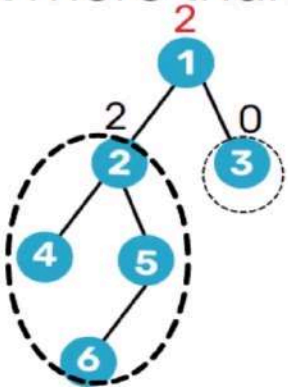
5. Balanced Binary Tree

In balanced binary tree, the absolute difference between the height of the left subtree and right subtree for each node is not more than 1

|-4| = 4

|-1| = 1



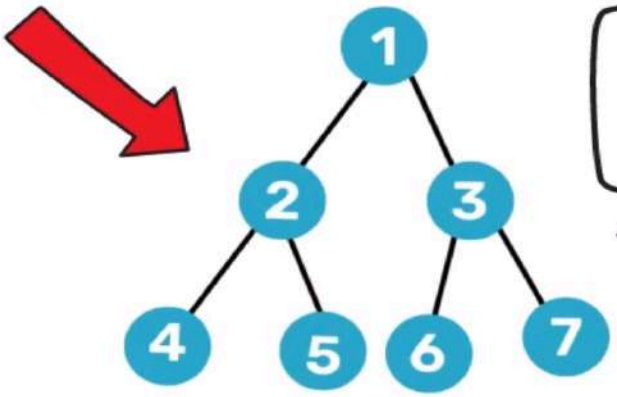


2-3 = -1

0 or 1

Tree Traversal

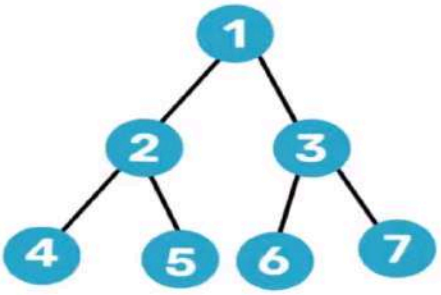
Tree Traversal means traversing or visiting each node of a tree.



Multiple ways to traverse tree

Tree Traversal Techniques

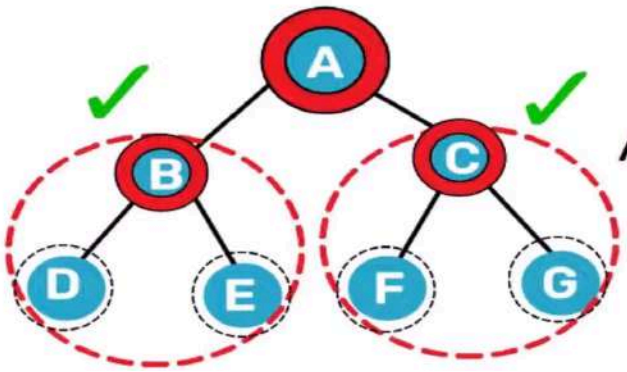
1. Preorder Traversal
2. Inorder Traversal
3. Postorder Traversal



1. Preorder Traversal

root left right

First root node is visited after that left subtree is traversed and finally right subtree is traversed



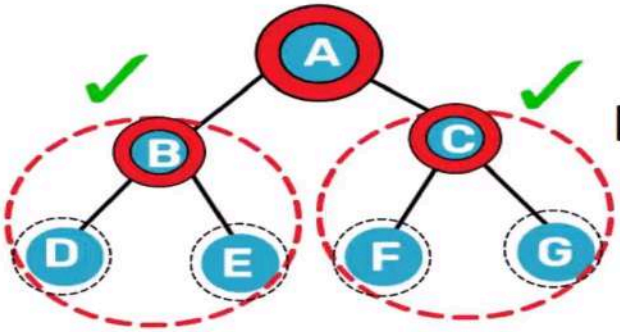
Output

A → B → D → E → C → F → G

2. Postorder Traversal

left right root

First left subtree is traversed after that righ subtree is traversed and finally root node is traversed



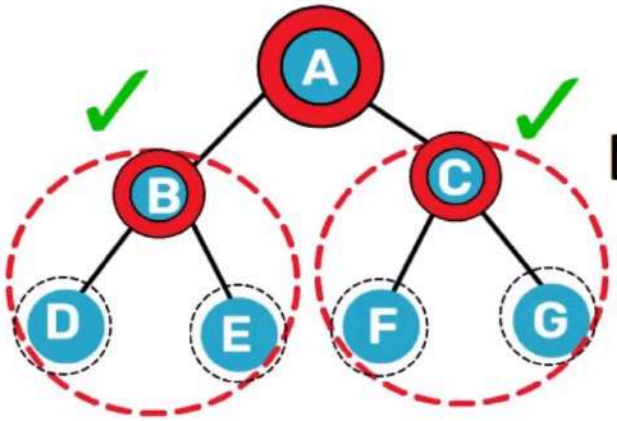
Output

D → E → B → F → G → C → A

3. Inorder Traversal

left root right

First left subtree is traversed after that root node is traversed and finally right subtree is traversed



Output

D → B → E → A → F → C → G

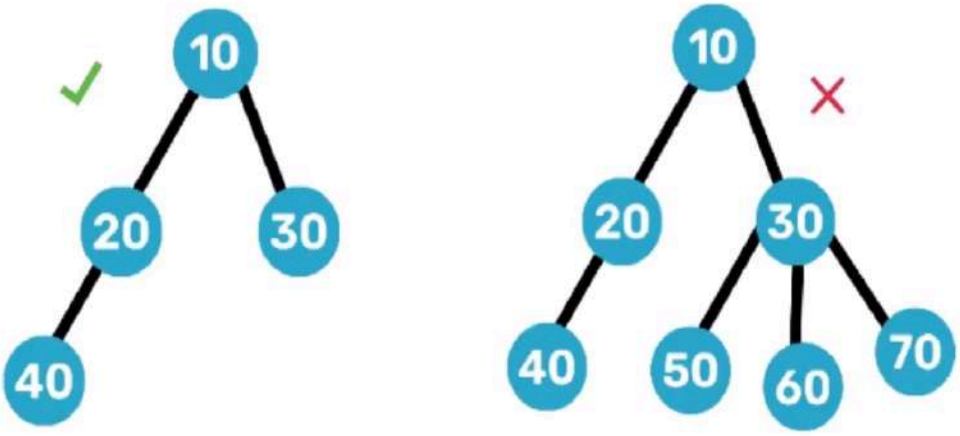
Heap Data Structure

What is Heap?

A heap is a complete binary tree, and the binary tree is a tree in which each node can have at most two children, referred to as the left child and the right child.

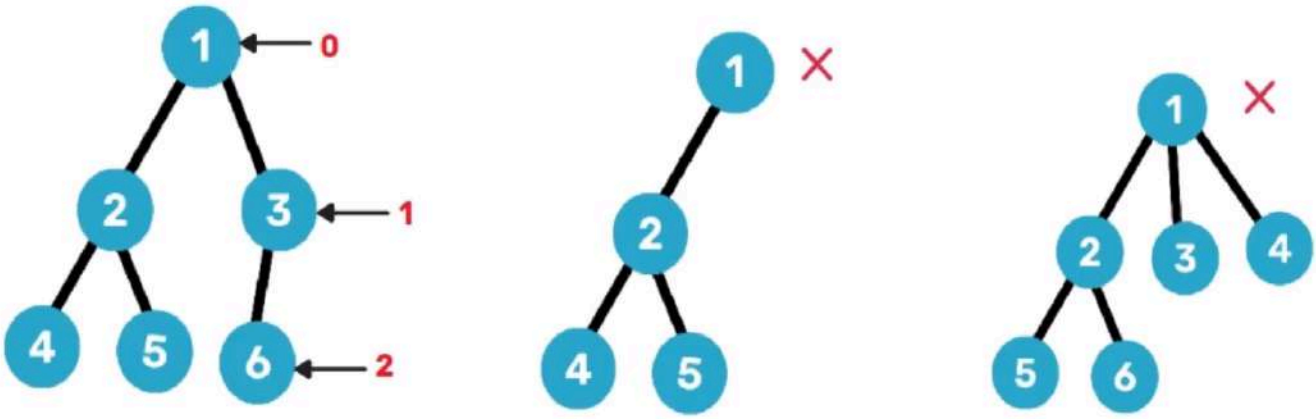
0 / 1 / 2

Binary Tree



Complete Binary Tree

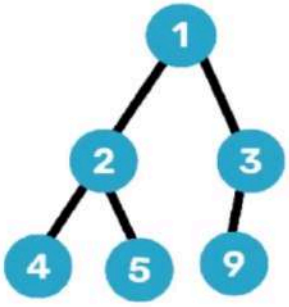
A complete binary tree is a binary tree in which all the levels are completely filled, except possibly the last level, which is filled from left to right.



Types of Heap

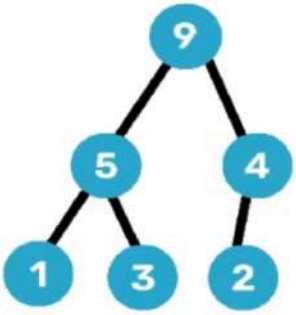
Min Heap

* The value of each parent node is less than or equal to the values of its children nodes.
* The root node contains the smallest value in the heap.



Max Heap

* The value of each parent node is greater than or equal to the values of its children nodes.
* The root node contains the largest value in the heap.



HEAP OPERATIONS

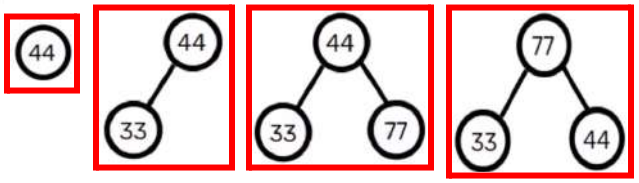
1: Heapify:(Heap Build)

This operation converts an array of elements into a heap.

Max Heap

0	1	2	3	4	5	6
44	33	77	11	55	88	66

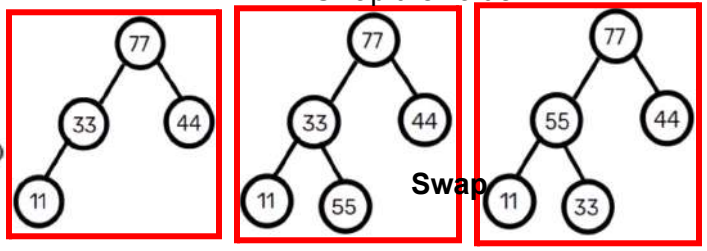
* First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
* Secondly, the value of the parent node should be greater than or equal to that of its child.

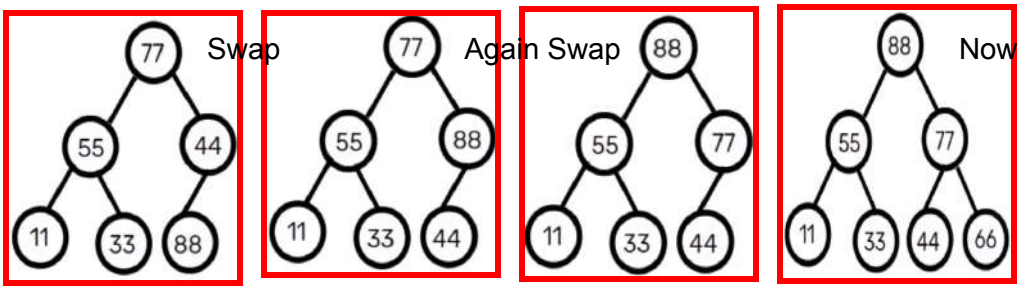


Swap the value

Min Heap

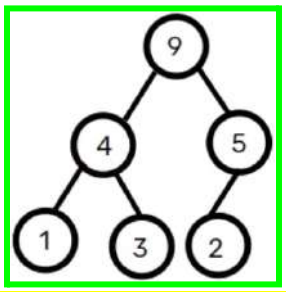
* The value of each parent node should be less than or equal to that of its child.

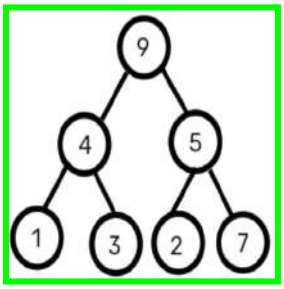




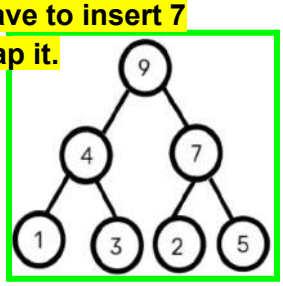
2: Insertion:(Heapify up)

* To insert a new element into the heap, it is placed at the end of the tree to maintain the shape property of a complete binary tree.
* After insertion, the heap might violate the heap property (either max-heap or min-heap).
* To restore the heap property, the newly inserted element is swapped with its parent node.



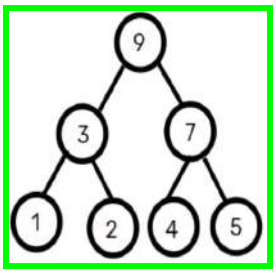


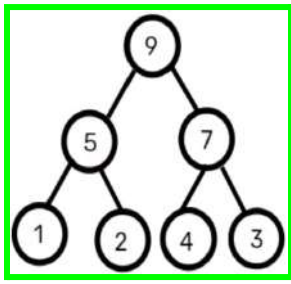
In this heap we have to insert 7
Then add and swap it.



3: Deletion:

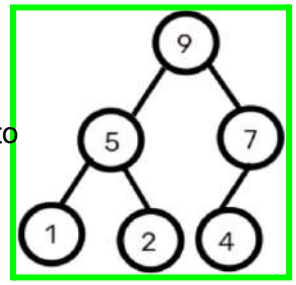
* Select the element to be deleted, swap it with the last element, and remove the last element.
* After deletion, the heap may violate the heap property.
* To restore the heap property, heapify the tree.



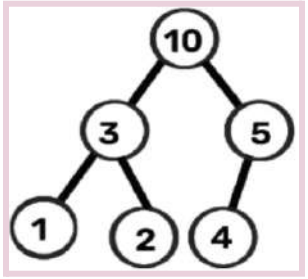


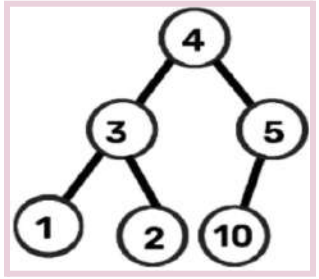
Swap 3 to 5

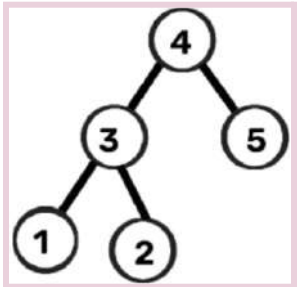
In this tree no need to heapify because it is already in heap Structure

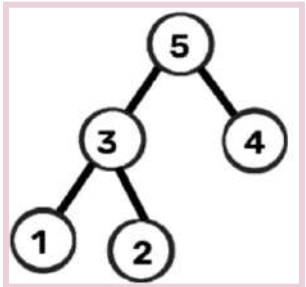


Another example: 10 waana remove





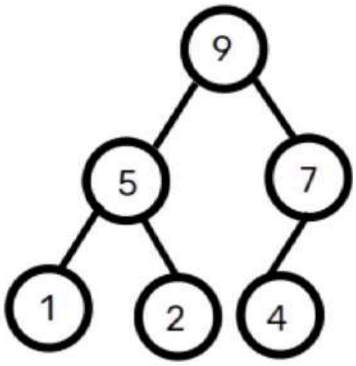




4: Peak:

* Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

* This operation returns the value of the root node.



5: Extract:

* Extract operation returns the maximum element from Max Heap or minimum element from Min Heap after removing it from the heap.

* After extraction, the heap is adjusted to maintain the heap property.

2.Binary Search Tree

A **Binary Search Tree (BST)** is a type of binary tree where:

- The **left child** of a node contains values **less than or equal to** the node's value.
- The **right child** of a node contains values **greater than** the node's value.

Uses of BST

- Efficiently perform operations like **searching**, **inserting**, and **deleting** data.
- Inorder traversal of BST gives the data in **sorted order**.

Rule 1

The left subtree of a node contains only nodes with keys lesser than the node's key.

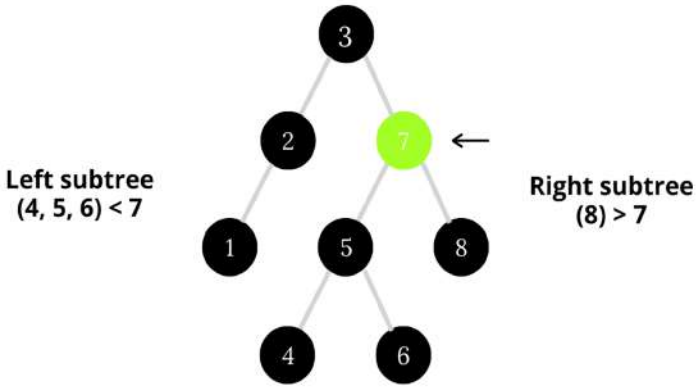
Rule 2

The right subtree of a node contains only nodes with keys greater than the node's key.

Rule 3

The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.

example:



BST OPERATIONS

1: Traversal Operation

Inorder Traversal (Left, Root, Right)
Steps:

1. Traverse the left subtree in inorder.
2. Visit the root node.
3. Traverse the right subtree in inorder.

Output: For a BST, this traversal gives nodes in **sorted order**.

Build a BST

Step 1: Start with an Empty Tree

- Initialize the tree as `NULL` (or empty).

Step 2: Insert the Root Node

- Take the first value from the input set.
- Create the root node with this value.

Step 3: Insert Remaining Values

For each subsequent value in the input set:

1. **Start at the root**.
2. Compare the value to be inserted with the current node:
    - If the value is **less than or equal**, move to the **left child**.
    - If the value is **greater**, move to the **right child**.
3. Repeat this comparison until you find a `NULL` position in the tree.
4. **Create a new node** at this position and insert the value.

Step 4: Repeat Until All Values Are Inserted

- Continue this process for every value in the input set.
- Ensure the BST property (left ≤ root ≤ right) is maintained at every step.

BST.cpp > inorder(node *)

```cpp
#include<iostream>
using namespace std;
struct node{
    int data;
    node *left,*right;
    //constructor
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};
node* insertBST(node *root,int val){
    if(root == NULL){
        return new node(val);
    }
    if(val < root->data){
        root->left = insertBST(root->left, val);
    }else {
        //val > root->data
        root->right = insertBST(root->right, val);
    }
    return root;
}

void inorder(node *root){
    if(root == NULL){
        return;
    }
    inorder(root -> left);
    cout<< root->data<<" | ";
    inorder(root -> right);
}

int main(){
     node *root =NULL;
     root = insertBST(root,5);
     insertBST(root,1);
     insertBST(root,3);
     insertBST(root,4);
     insertBST(root,2);
     insertBST(root,7);
     //print INORDER
     inorder(root);
     cout<<endl;
     return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
Warning: PowerShell detected that you might be using a screen reader and has disab
 PSReadLine'.

PS C:\Users\Aditya Yadav\Desktop\CODES\DSA> cd "c:\Users\Aditya Yadav\Desktop\CODE
1 | 2 | 3 | 4 | 5 | 7 |
```

2: Insertion Operation

Adding a new value to a binary search tree while maintaining its order (smaller values go to the left and larger values to the right).

1. Start at the root.
2. Compare the value with the current node:
    - If smaller, go left.
    - If larger, go right.
3. Repeat until you find an empty spot, and insert the value there.

BST-User.cpp > ...

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

struct node {
    int data;
    node *left, *right;

    // Constructor to create a new node
    node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};

// Function to insert a value into the BST
Tabnine | Edit | Test | Explain | Document | Ask
node *insert(node *root, int val) {
    if (root == NULL) {
        return new node(val);
    }
    if (val < root->data) {
        root->left = insert(root->left, val);
    } else {
        root->right = insert(root->right, val);
    }
    return root;
}

// Function for in-order traversal
Tabnine | Edit | Test | Explain | Document | Ask
void inorder(node *root) {
    if (root == NULL) {
        return;
    }
```

```
    inorder(root->left);
    cout << root->data << " | ";
    inorder(root->right);
}

Tabnine | Edit | Test | Explain | Document | Ask
int main() {
    node *root = NULL; // Start with an empty tree

    int n;
    cout << "Enter the number of nodes you want to insert: ";
    cin >> n; // Number of values user wants to insert

    cout << "Enter the values: " << endl;
    for (int i = 0; i < n; i++) {
        int val;
        cin >> val; // Take the value from the user
        root = insert(root, val); // Insert it into the BST
    }

    cout << "In-order traversal of the BST: ";
    inorder(root); // Print the BST in sorted order
    cout << endl;

    return 0;
}
```

```
Enter the number of nodes you want to insert: 5
Enter the values:
8 4 9 7 6
In-order traversal of the BST: 4 | 6 | 7 | 8 | 9 |
```

3: Deletion Operation

Deleting a node in a binary search tree involves finding and removing a node while maintaining the BST properties (smaller values on the left and larger values on the right).

Cases of Deletion in BST:

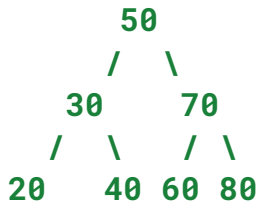
1. **Node with One or No Child**:
   - If the node has **no child**: Simply remove it (set the parent's link to null).
   - If the node has **one child**: Replace the node with its child (either left or right).
2. **Node with Two Children**:
   - Replace the node with its **in-order successor** (smallest value in the right subtree) or **in-order predecessor** (largest value in the left subtree).
   - Then delete the successor or predecessor node (this will fall under case 1).

Steps for Deletion:

1. **Find the Node**:
   Start at the root and locate the node to delete by comparing values:
   - Go left if the value is smaller.
   - Go right if the value is larger.
2. **Handle Each Case**:
   - **Case 1: No Child**
     - Remove the node directly by setting its parent's link to `null`.
   - **Case 2: One Child**
     - Replace the node with its only child by linking the parent directly to the child.
   - **Case 3: Two Children**
     - Find the **in-order successor** (leftmost node in the right subtree) or **in-order predecessor** (rightmost node in the left subtree).
     - Replace the value of the node to be deleted with the successor's or predecessor's value.
     - Delete the successor or predecessor node (which will now be in case 1 or 2).

Examples

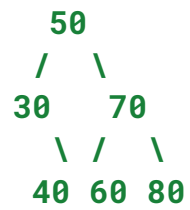
For a tree with:



1. **Deleting `20` (No Child)**:

Remove the node directly.

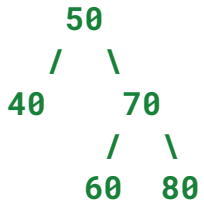
Result:



2. **Deleting `30` (One Child)**:

Replace `30` with `40`.

Result:

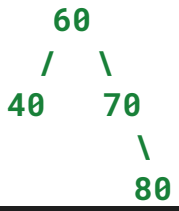


3. **Deleting `50` (Two Children)**:

Replace `50` with its in-order successor `60`.

Delete `60`.

Result:




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

struct node {
    int data;
    node *left, *right;
    node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};

Tabnine | Edit | Test | Explain | Document | Ask
node *inorderSucc(node *root) {
    node *curr = root;
    while (curr && curr->left != NULL) {
        curr = curr->left;
    }
    return curr;
}

Tabnine | Edit | Test | Explain | Document | Ask
node *del(node *root, int key) {
    if (root == NULL) {
        return NULL;
    }

    if (key < root->data) {
        root->left = del(root->left, key);
    } else if (key > root->data) {
        root->right = del(root->right, key);
    } else {
        if (root->left == NULL) {
            node *temp = root->right;
```

```
            delete root;
            return temp;
        } else if (root->right == NULL) {
            node *temp = root->left;
            delete root;
            return temp;
        }

        node *temp = inorderSucc(root->right);
        root->data = temp->data;
        root->right = del(root->right, temp->data);
    }
    return root;
}

Tabnine | Edit | Test | Explain | Document | Ask
void inorder(node *root) {
    if (root == NULL) {
        return;
    }
    inorder(root->left);
    cout << root->data << " | ";
    inorder(root->right);
}

Tabnine | Edit | Test | Explain | Document | Ask
int main() {
    node *root = new node(50);
    root->left = new node(30);
    root->right = new node(70);
    root->left->left = new node(20);
    root->left->right = new node(40);
    root->right->left = new node(60);
    root->right->right = new node(80);

    cout << "Inorder Traversal (Before Deletion): ";
    inorder(root);

    cout << "\nDeleting Node 70...\n";
    root = del(root, 70);

    cout << "Inorder Traversal (After Deletion): ";
    inorder(root);

    return 0;
}
```

```
Inorder Traversal (Before Deletion): 20 | 30 | 40 | 50 | 60 | 70 | 80 |
Deleting Node 70...
Inorder Traversal (After Deletion): 20 | 30 | 40 | 50 | 60 | 80 |
PS C:\Users\Aditya Yadav\Desktop\CODES\DSA>
```

4: Searching Operation

Searching in a **Binary Search Tree (BST)** is a process to find whether a particular key (value) exists in the tree. BST properties help in efficient searching by deciding whether to go left or right at each step.

Steps for Searching in BST:

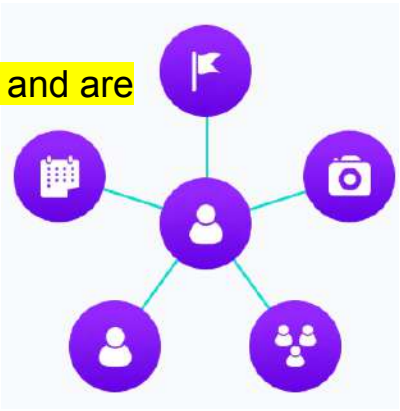
1. **Start at the Root**:
    - Begin the search from the root node of the BST.
2. **Compare the Key**:
    - If the key matches the root node's value, the search is successful, and the key is found.
3. **Decide the Direction**:
    - If the key is **smaller** than the current node's value:
        - Move to the **left child**.
    - If the key is **greater** than the current node's value:
        - Move to the **right child**.
4. **Repeat**:
    - Continue the process recursively or iteratively for the left or right subtree until:
        - The key is found (success).
        - You reach a `NULL` node (key does not exist).

```cpp
#include<iostream>
using namespace std;
struct node{
    int data;
    node *left,*right;
    node(int val){
        data=val;
        left=NULL;
        right=NULL;
    }
};
Tabnine | Edit | Test | Explain | Document | Ask
void inorder(node*root){
    if(root==NULL){
        return;
    }
    inorder(root->left);
    cout<<root->data<<" | ";
    inorder(root->right);
}
Tabnine | Edit | Test | Explain | Document | Ask
node *search(node *root,int key){
    if(root==NULL){
        return NULL;
    }
    if(root->data==key){
        return root;
    }
    if(root->data>key){
        return search(root->left,key);
    }
    else{
        return search(root->right,key);
    }
}
int main(){
    node *root=new node(50);
    root->left=new node(30);
    root->left->left=new node(20);
    root->left->right=new node(40);
    root->right=new node(70);
    root->right->left=new node(60);
    root->right->right=new node(80);
    inorder(root);
    int key ;
    cout<<"\nEnter key to search : ";
    cin>>key;
    if(search(root,key)==NULL){
        cout<<"Key doesn't exist.....";
    }else{
        cout<<"Key exist....";
    }
    return 0;
}
```

```
20 | 30 | 40 | 50 | 60 | 70 | 80 |
Enter key to search : 60
Key exist....
PS C:\Users\Aditya Yadav\Desktop\CODES\DSA>
```

GRAPH

A graph data structure is a collection of nodes that have data and are connected to other nodes.



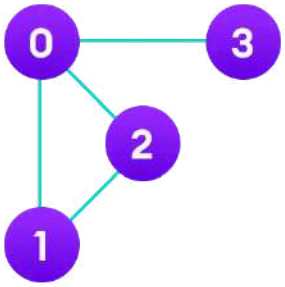
Graph Representation: -

Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

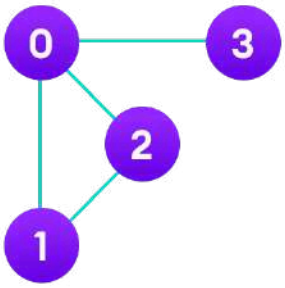
Example

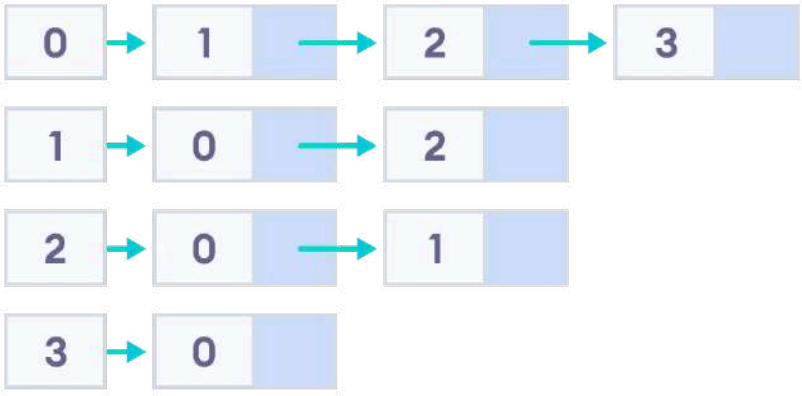


	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

2. Adjacency List

An adjacency list represents a graph as an array of linked lists.
The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.





Types Of Graphs in Data Structure and Algorithms

1. Null Graph

A graph is known as a null graph if there are no edges in the graph.

2. Trivial Graph

Graph having only a single vertex, it is also the smallest graph possible.

Null Graph

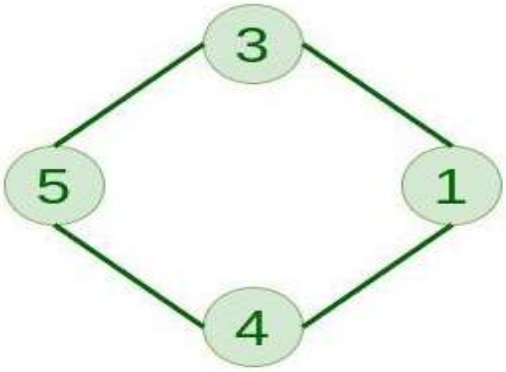
Trivial Graph

3. Undirected Graph

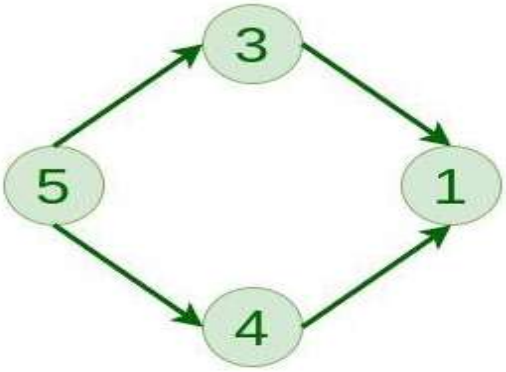
A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

4. Directed Graph

A graph in which the edge has direction. That is the nodes are ordered pairs in the definition of every edge.



Undirected Graph



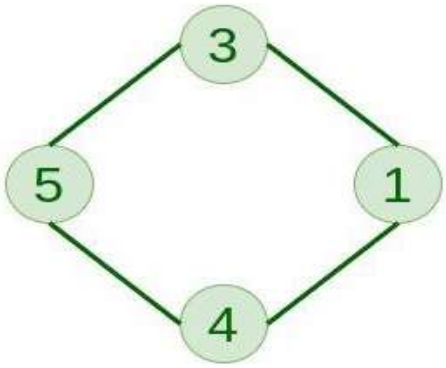
Directed Graph

5. Connected Graph

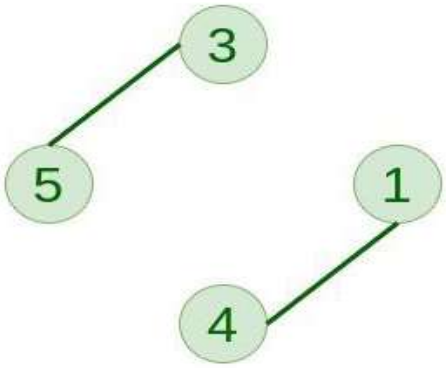
The graph in which from one node we can visit any other node in the graph is known as a connected graph.

6. Disconnected Graph

The graph in which at least one node is not reachable from a node is known as a disconnected graph.



Connected Graph



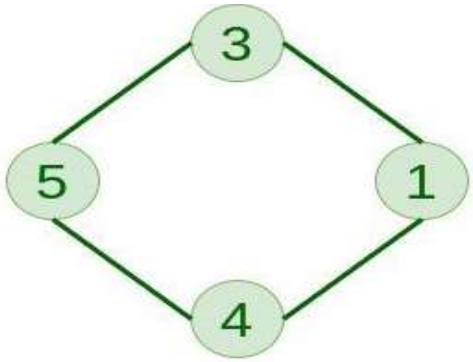
Disconnected Graph

7. Regular Graph

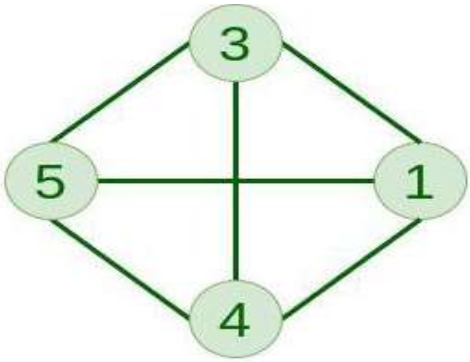
The graph in which the degree of every vertex is equal to K is called K regular graph.

8. Complete Graph

The graph in which from each node there is an edge to each other node.



2-Regular



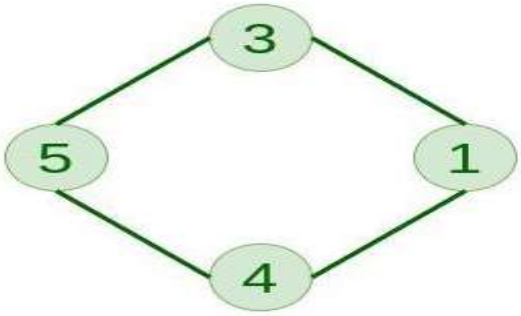
Complete Graph

9. Cycle Graph

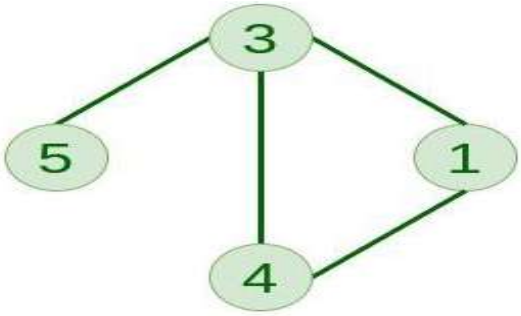
The graph in which the graph is a cycle in itself, the minimum value of degree of each vertex is 2.

10. Cyclic Graph

A graph containing at least one cycle is known as a Cyclic graph.



Cycle Graph



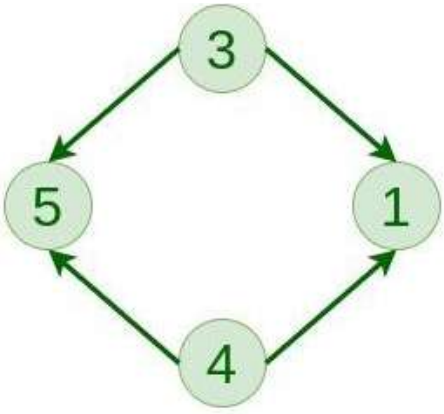
Cyclic Graph

11. Directed Acyclic Graph

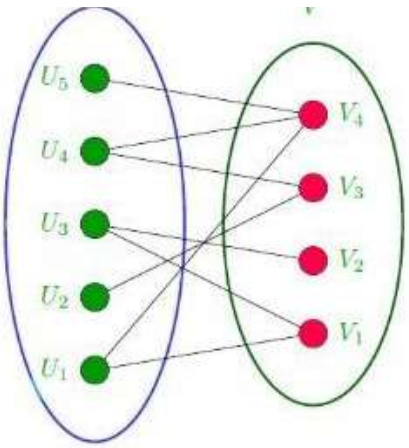
A Directed Graph that does not contain any cycle.

12. Bipartite Graph

A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.



Directed Acyclic Graph



Bipartite Graph

13. Weighted Graph

- A graph in which the edges are already specified with suitable weight is known as a weighted graph.
- Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

1: Traversal in Graph:

Traversal in a **graph** refers to the process of visiting all the vertices (nodes) of the graph in a systematic way. The goal is to ensure that each vertex is visited exactly once (if required) while following specific rules of traversal.

There are two main types of graph traversal techniques: **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**.

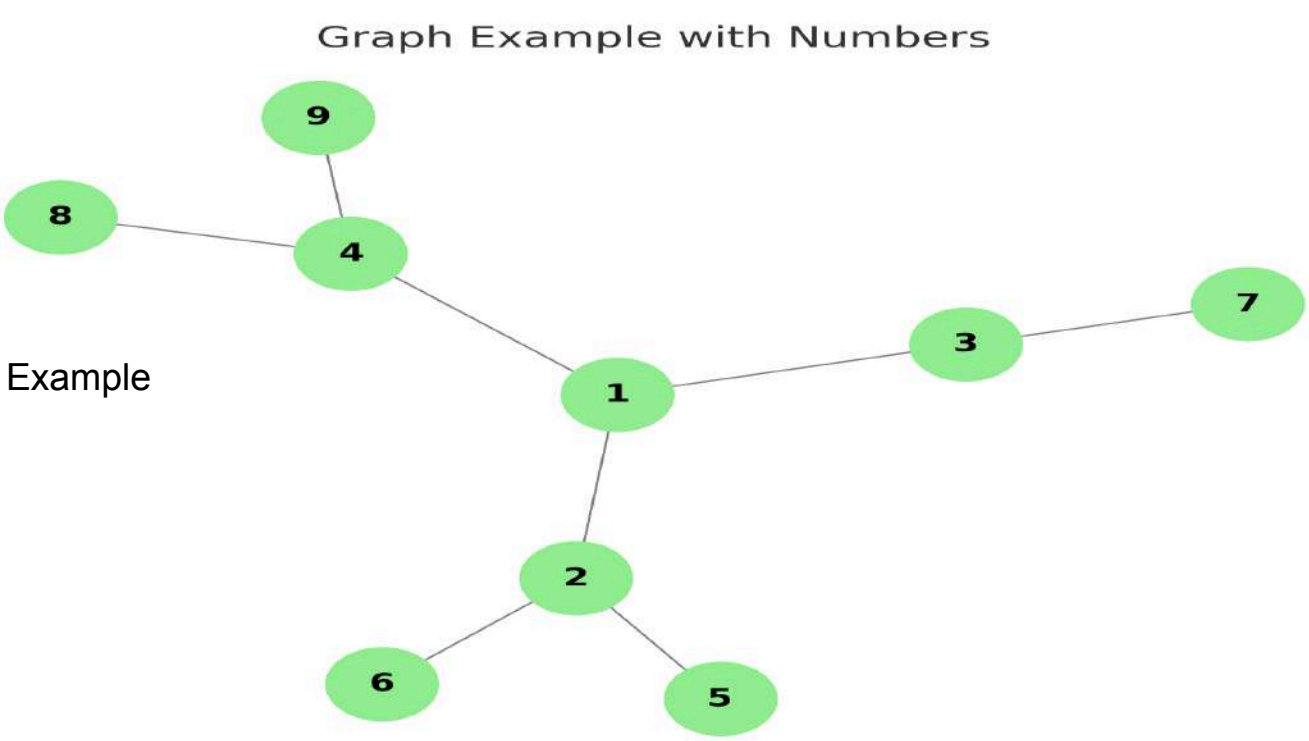
Depth-First Search (DFS) :

DFS is a graph traversal method where you start from a node and go as deep as possible along one path. If you process the lowest value first, you visit the neighbors in **ascending order** (smallest to largest) before backtracking.

Breadth-First Search (BFS) :

BFS is a graph traversal method where you visit all nodes at the current level before moving to the next level. If you process the lowest value first, you visit neighbors in **ascending order** (smallest to largest) at each level.

Example



DFS(Lowest Value First)

- Traversal: 1 → 2 → 5 → 6 → 3 → 7 → 4 → 8 → 9

Starts from **1**, goes to **2** (lowest neighbor), explores its neighbors (**5, 6**) before backtracking to **3** and **4**.

BFS(Lowest Value First)

- Traversal: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

Starts from **1**, explores all its neighbors (**2, 3, 4**) in ascending order, then moves to the next level (**5, 6, 7, 8, 9**) in ascending order.

Credit goes to

1. Aditya kumar yadav **2. Coding With Clicks** **3. Chat GPT**