.

Unit Code	**ICS 2105**
Unit Name	Data structures and algorithms
Prerequisite	Introduction to Computer Programming
Cohort	MCS
Lecturer	Wairagu G.R
Contact	Wairagu.rg@gmail.com

Purpose

To enable the students understand the concepts and application of data structures and algorithms.

Course Objectives:

a) Develop sound techniques on designing, developing, and documenting well-structured programs using proper software engineering principles.
b) Understand the purpose and mathematical background of algorithm analysis and be able to apply this to determine the run time and memory usage of algorithms
c) Describe and implement common data structures--lists, stacks, queues, graphs, and trees--for solving complex programming problems.
d) Explain the different sorting and searching techniques

Course Description

Introduction to Data Structures and Algorithms: Definitions and Uses of Data Structures and Algorithms, Role of Data Structures and Algorithms Programming, Choice of Data Structures and Algorithms. Elementary Data Structures: List, Queue, Stack, Tree, Arrays; Types of List: Linear-Linked List, Doubly Linked List, Circular Linked List, Circular Doubly Linked List; Types Of Queue: Circular Queue; Types of Trees: AVL Tree, Red Black Trees, B-Trees; Graphs; Array Based and Pointer-Based Implementation Of Graphs, Hashing, Heap, Linear, Binary Search Algorithms; Sorting Algorithms; Depth-First, Breadth, Hill-Climbing, Least-Cost Search Algorithms Using Either A Structured Programming Language or an OOP Language.

Teaching Methodologies

Lectures, Practical sessions and Tutorials.

Instructional Materials/Equipment

LCD Projector, Whiteboard, Textbooks, Computers and Internet.

Course Outline

WEEK	COURSE CONTENT	REMARKS
Week 1-2	a) Introduction. Course overview. Introduction to data structures and algorithms b) Fundamentals of C++ programming. c) Lab – C++ Syntax	
Week 3	Introduction a) Basic Definitions b) Structured Data Types c) Arrays –Implement arrays Lab – C++ data types, operators.	
Week 4	Lists • Lists as an Abstract Data Type • Implementation Lab - Selection control structures in c++	
Wek 5	Stacks • Stack as an Abstract Data Type • An Array Implementation of Stacks Application of Stacks Lab – loops in c++ Queues • Queue as an Abstract Data Type • An Array Implementation of Queues Applications of Queues	
Week 6	Trees • Binary Trees • Binary Search Trees • Tree Traversal	

	• Lab –implementing arrays in C++	
Week 7	CAT 1 Revision of CAT 1	
Week 8-9	• Heaps Graphs • Definition • Representation • Traversing • Minimum spanning tree • Topological sort • Shortest Path Lab – implementing linked lists in C++	•
Week 10-11	Sorting Algorithms Selection Sort • Selection Sort • Bubble Sort • Insertion Sort Lab – implementing stacks in C++ Implementing BST in C++ •	
Week 12	Sorting Algorithms • Quick Sort • Merge Sort • Heap Sort Lab – implementing Queues in C++ Lab- Implement Heap •	
Week 13	Searching algorithms • Sequential Search • Binary Search	

	Lab –Implementation of bubble sort	
Week 14	CAT2 a) Evaluation Revision	
Week 15 and 16	End of Semester Exams	

Course Assessment Mode:

Laboratory Practicals	10%
Continuous Assessment Tests	20%
Total Continuous Assessment	**30%**
End of Semester Examination	**70%**

Practicals/Laboratory sessions

a) Lab 1 – C++ Syntax
b) Lab 2 – Data types, Operator
c) Lab 3- Selection control structures in c++
d) Lab 4– loops in c++
e) Lab 5 –implementing arrays in C++
f) Lab 6– implementing linked lists in C++
g) Lab 7– implementing stacks in C++
h) Lab 8 – implementing Queues in C++
i) Lab 9 –Implementation of bubble sort
j) Lab 10– Implement quick sort
k) Lab 11 – Implement sequeatial search

Core Reading Materials:

Course Text books

1. Narasimha K. (2011). *Data Structures and Algorithms Made Easy: Data Structure and Algorithmic Puzzles*, (1st Ed.). CreateSpace Independent Publishing Platform. ISBN-13: 978-1456549886

2. Weiss M.A. (2006). *Data Structures and Algorithm Analysis in Java*, (2nd Ed.). Addison Wesley. ISBN-13: 978-0321370136

3. Drozdek A. (2012). *Data Structures and Algorithms in C++* (4th Ed.). Cengage Learning. ISBN-13: 978-1133608424

Course Journals

1. *Journal of Computer and System Sciences.* ScienceDirect. ISSN: 0022-0000
2. *International Journal of Advanced Computer Science and Technology (IJACST).* IJACST. ISSN: 2249-3123
3. *Advances in Computational Sciences and Technology (ACST).* ACST. ISSN: 0974-4738

Reference Materials:

Reference Textbooks

1. Ford W.H. (2001). *Data Structures with C++*, (2nd Ed.). Prentice Hall. ISBN-13: 978-0130858504
2. Standish T.A. (1998). *Data Structures in Java.* Addison-Wesley. ISBN: 978-0201305647
3. Yedidyah AU, L. (2011). *Data Structures using C and C++*, (2nd Ed.). ISBN-13: 978-8120311770

Reference Journals

1. *International Journal of Computational Science and Engineering*, IJCSE. ISSN: 2249-4251
2. *International Journal of Information Science and Education (IJISE*). IJISE. ISSN: 2231-1262
3. *Global Journal of Computational Intelligence Research (GJCIR).* GJCIR. ISSN 2249-0000

Approved for use: Sign: (CoD)

_____Date_____

INTRODUCTION

Computers are being used in virtually all aspects of our lives:

- Scientific Calculations
- Information Management System for an enterprise
- Real-time control system for an assembly line

In performing these operations, to use the computer effectively one needs to:

- Obtain the needed data and their relations
- Store the data into the computer according to their relations
- Design algorithms to solve the problem

The basic functions of a computer are:

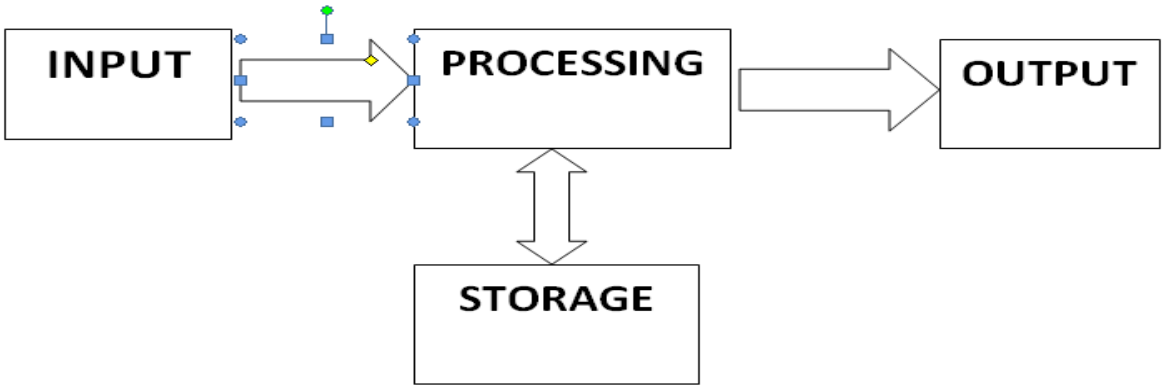
Input: Receiving data and instructions.

Processing: Performing operations on data according to the instructions.

Storage: Saving data and instructions.

Output: Delivering Information or results

Control: Directing and coordinating the other functions of the computer system



- This unit focuses on storage – data structures

- The tasks of programming and data processing require efficient algorithms for accessing the data both in main memory and on secondary storage devices.
- This efficiency is directly linked to the structure of the data being processed.

DATA STRUCTURES

A **data structure** is a term used to denote the organization, management, and storage of data in a computer.

Examples of Data Structures:

- Arrays: A collection of elements identified by an index or key, stored in contiguous memory locations.
- Linked Lists: A sequence of elements, where each element points to the next, allowing for efficient insertion and deletion.
- Stacks: A collection of elements that follows the Last In, First Out (LIFO) principle.
- Queues: A collection of elements that follows the First In, First Out (FIFO) principle.
- Trees: A hierarchical structure with nodes representing elements and edges representing relationships, commonly used in databases and file systems.
- Graphs: A set of nodes connected by edges, used to represent networks, such as social networks or transportation systems.

Data structures are fundamental to computer science and software engineering. They provide the foundation for designing efficient algorithms and are essential for solving complex computational problems. Choosing the appropriate data structure can significantly impact the performance and scalability of an application.

Common operations performed on data structures include:

- Insertion: Adding a new element to the data structure.
- Deletion: Removing an existing element from the data structure.
- Search: Finding a specific element within the data structure.
- Traversal: Visiting all elements in the data structure in a specific order.
- Update: Modifying an existing element's value or properties.
- Access: Retrieving the value of an element at a specific position or index.
- Sort: Arranging the elements in a specific order, such as ascending or descending.
- Merge: Combining elements from two or more data structures into one.
- Split: Dividing a data structure into multiple parts based on certain criteria.
- Peek: Viewing the value of an element without removing it (common in stack and queue operations).An

APPROPRIATE DATA STRUCTURE

When selecting a data structure for a particular problem, it's important to consider various factors to ensure that the chosen data structure meets the requirements of the task efficiently and effectively. Here are some key factors to consider:

- Access Patterns: Consider how you'll be accessing and manipulating the data. Will you be performing frequent insertions, deletions, searches, or updates? Different data structures excel at different types of operations. For example, arrays are good for random access, while linked lists are better for frequent insertions and deletions.
- Time Complexity: Analyze the time complexity of common operations (insertion, deletion, search, etc.) for each potential data structure. Choose a data structure that provides efficient operations for the tasks you need to perform frequently.
- Space Complexity: Consider the memory requirements of the data structure. Some data structures use more memory than others to provide certain advantages. It's important to strike a balance between memory usage and performance.
- Data Size: The size of the data you need to store can influence your choice of data structure. For large datasets, data structures with efficient memory usage and access times are crucial.
- Ordering Requirements: If the data needs to be maintained in a specific order (sorted, chronological, etc.), choose a data structure that supports that order efficiently. For example, binary search trees are good for maintaining sorted data.
- Ease of Implementation: Some data structures are more complex to implement than others. Choose a data structure that aligns with your familiarity with the programming language and your project's timeline.
- Support for Operations: Different data structures are designed for specific operations. For instance, if you need to find the minimum or maximum value quickly, a heap might be suitable. If you need to perform graph traversals, a graph data structure would be necessary.
- Adaptability: Consider whether the data structure needs to adapt to changing data or requirements. Some data structures, like dynamic arrays, can grow or shrink as needed.
- Existing Libraries: If your programming language offers built-in data structures (e.g., Python lists, C++ vectors), consider using these libraries as they are well-optimized and tested.

Overall, the choice of data structure should be based on a careful analysis of your problem's requirements and the trade-offs between different data structures. It's important to understand the strengths and weaknesses of each data structure to make an informed decision that leads to efficient and effective solutions.

Question

Discuss the best data structure in Facebook, Twitter

ALGORITHMS

- This is a term used to refer to the sequence of steps followed in performing a task
- An Computer Algorithm consist of a finite sequence of well-defined, computer-implementable instructions, designed to perform a specific task such performing a computation, sorting data in a file, searching
- Algorithms are a fundamental concept in computer science and play a crucial role in various applications, from solving mathematical problems to optimizing data processing and making decisions in artificial intelligence.
- Computer instructions are derived from algorithms

Characteristics of a Good Algorithm

A good algorithm exhibits several key characteristics that contribute to its efficiency, reliability, and effectiveness in solving problems. Here are some important characteristics of a good algorithm:

- Correctness: A good algorithm should produce the correct and expected output for all valid inputs. It should accurately solve the problem it's designed for without errors.
- Efficiency: Efficiency refers to how well an algorithm performs in terms of time and space usage. A good algorithm should be designed to run in a reasonable amount of time and use a reasonable amount of memory. This involves considering both time complexity (how the algorithm's runtime scales with input size) and space complexity (how much memory the algorithm uses).
- Input: An algorithm should clearly define its input requirements and constraints. It should handle valid inputs gracefully and possibly include error handling for invalid inputs.
- Output: The output of the algorithm should be well-defined and appropriate for the problem it aims to solve. It should also be presented in a clear and understandable format.

- Readability: A good algorithm is readable and easy to understand. Well-named variables, clear comments, and logical organization of steps contribute to the readability of the code.
- Maintainability: An algorithm should be easy to modify and maintain. Changes or updates to the algorithm should not introduce unexpected bugs or side effects.
- Modularity: Breaking down an algorithm into smaller, modular components promotes code reusability and easier debugging. Each module should have a clear purpose and be responsible for a specific part of the algorithm.
- Adaptability: Algorithms should be designed to handle changing requirements and data. They should be flexible enough to accommodate new inputs, constraints, or scenarios without requiring significant restructuring.
- Simplicity: Simple algorithms are often easier to implement, understand, and maintain. Complex algorithms may be necessary for intricate problems, but they should be well-documented and well-explained.
- Scalability: A good algorithm should be able to handle increasing input sizes without a significant decrease in performance. This is particularly important for algorithms used in real-world applications.
- Robustness: An algorithm should handle unexpected situations gracefully and provide appropriate responses. This includes error handling and resilience against variations in input data.
- Testability: It should be possible to test an algorithm's correctness and performance through various test cases. This helps ensure that the algorithm works as intended.
- Documentation: Providing clear and comprehensive documentation, including explanations of the algorithm's purpose, input-output behavior, and any underlying assumptions, is essential for others (and your future self) to understand and use the algorithm effectively.

In summary, a good algorithm balances correctness, efficiency, readability, and adaptability. It is designed to provide reliable and efficient solutions to specific problems while being understandable and maintainable by developers.

Here are some examples of commonly used algorithms:

- Sorting Algorithms:
    - Bubble Sort: Repeatedly compares and swaps adjacent elements in an array.
    - Merge Sort: Divides an array into halves, recursively sorts each half, and merges them.
    - Quick Sort: Divides the array around a pivot, recursively sorts the partitions.

    - Insertion Sort: Builds the sorted array one element at a time by inserting each new element into its correct position.
- Searching Algorithms:
    - Binary Search: Finds an element in a sorted array by repeatedly dividing the search interval in half.
    - Linear Search: Sequentially checks each element in a list until the desired element is found.
- Graph Algorithms:
    - Breadth-First Search (BFS): Explores all nodes at the present depth level before moving on to nodes at the next depth level.
    - Depth-First Search (DFS): Explores as far as possible along each branch before backtracking.
- Optimization Algorithms: Used to find the best solution from a set of possible solutions, given specific constraints and objectives.
- Machine and Deep Learning Algorithms: These algorithms are used to train models and make predictions or decisions based on data patterns.

CLASSIFICATION OF DATA STRUCTURES

Data structures can be broadly classified into two categories:

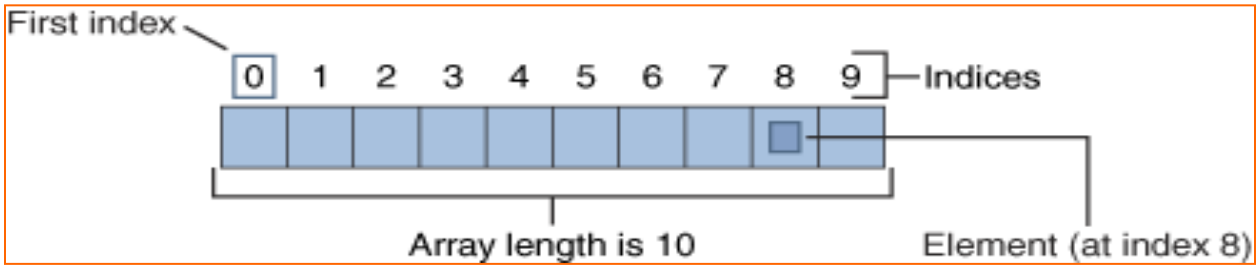
- **Primitive Data Structures:**
  These are the basic building blocks for more complex data structures. They are directly supported by the programming language.
    - Integers: Whole numbers.
    - Floats: Numbers with decimal points.
    - Characters: Single alphabets or symbols.
    - Booleans: True or False values.
- **Structured Data Structures:**
  Structured data structures, often referred to as composite or non-primitive data structures, are those that are made up of multiple elements, which can be of the same or different data types. These structures organize and manage data in a specific format that allows for efficient access, modification, and storage. They can be further classified into:
  Linear Data Structures:
    - Arrays: A collection of elements stored in contiguous memory locations, accessible by index.
    - Linked Lists: A sequence of nodes where each node contains a data element and a reference to the next node.
    - Stacks: A LIFO (Last In, First Out) structure where elements are added and removed from the top.
    - Queues: A FIFO (First In, First Out) structure where elements are added at the rear and removed from the front.

Non-Linear Data Structures:

- Trees: A hierarchical structure with a root node and child nodes, where each node can have zero or more children.
- Binary Trees: A tree structure where each node has at most two children.
- Binary Search Trees (BSTs): A binary tree where the left child contains values less than the parent node, and the right child contains values greater.
- Heaps: A special type of binary tree used for priority queues, where the parent node is always greater (max heap) or smaller (min heap) than its children.
- Graphs: A collection of nodes (vertices) connected by edges, which can represent relationships between elements.
  - Directed Graphs: Graphs where edges have a direction.
  - Undirected Graphs: Graphs where edges do not have a direction.

ARRAY

An **array** is a data structure that consists of a collection of elements, typically of the same data type, stored in contiguous memory locations. Each element in the array can be accessed directly using an index, which represents the element's position within the array.



- An array type is appropriate for representing an abstract data type when the following conditions are satisfied:
  1) *The data objects in the abstract data type are composed of homogeneous objects*
  2) *The solution requires the representation of a fixed, predetermined number of objects*

Key Characteristics of Arrays:

- **Fixed Size**: The size of an array is defined at the time of its creation and cannot be changed during runtime.
- **Homogeneous Elements**: All elements in an array are of the same data type (e.g., all integers, all characters).
- **Indexed Access**: Elements in an array are accessed using a zero-based index, allowing for direct access to any element.
- **Contiguous**: An array occupies continuous memory space

Declaring an array

Declaring an array involves specifying the type of the elements and the size of the array. Here are the different ways to declare an array, along with examples:

```
type arrayName[size];

int numbers[5];    // An array of 5 integers
double prices[10]; // An array of 10 doubles
char name[20];     // An array of 20 characters
```

Data input, manipulation and output in an array

Array initialization

Initializing an array in C++ involves assigning initial values to the elements of the array at the time of its declaration.

syntax

```
int arr[5] = {x,y,k,m,z}; // Array of 5 integers

#include <iostream>

using namespace std;

 int main()

 {

// Initialize an array with 10 specific values

 int arr[10] = {100, 20, 130, 40, 150, 260, 70, 800, 390, 100};

// Display the array elements
```

```
cout << "Array elements are: ";

for (int i = 0; i < 10; i++)

 { cout << arr[i] << " "; }

 return 0;

 }
```

This program prompts a user to enter TEN numbers into an array, display them and the largest among them.

```
#include<iostream>

using namespace std ;

int main()

{

int Numbers[10], i, big;

cout<<"We are illustrating arrays"<<"\n";

cout<<"*** Assign values ***";

        for (int i=0;i<10;i++)

        {     cout<< "Enter Number in Cell :- "<<i<<"\n";

                cin>>Numbers[i];

    }

cout<<"Elements Entered\n";

        for ( i=0;i<10;i++)

        {  cout<<Numbers[i]<<",";

        }

//Get the biggest element
```

```
big=Numbers[0];

for (i=0;i<10;i++)

{

        if(Numbers[i]>big)

        big=Numbers[i];

}

cout<< "The Bigest number is  :"<< big;

}
```

Write a C++ program which prompts a user to enter eight(8) values in an array, display them, find and display their sum and average

```
#include <iostream>

using namespace std;

int main() {

    const int SIZE = 8;

    int arr[SIZE];

    int sum = 0;

    float average;

    // Prompt the user to enter 8 values

    cout << "Enter 8 values: " << endl;

    for (int i = 0; i < SIZE; i++) {

        cout << "Value " << i + 1 << ": ";

        cin >> arr[i];

        sum += arr[i];  // Add each value to the sum
```

```
    }

    // Calculate the average

    average = sum / SIZE;

    // Display the entered values

    cout << "\nThe entered values are: ";

    for (int i = 0; i < SIZE; i++) {

        cout << arr[i] << " ";

    }

    // Display the sum and average

    cout << "\nSum of the values: " << sum;

    cout << "\nAverage of the values: " << average << endl;

    return 0;

}
```

Explanation:

Array Declaration: The program declares an integer array of size 8 to store the values entered by the user.

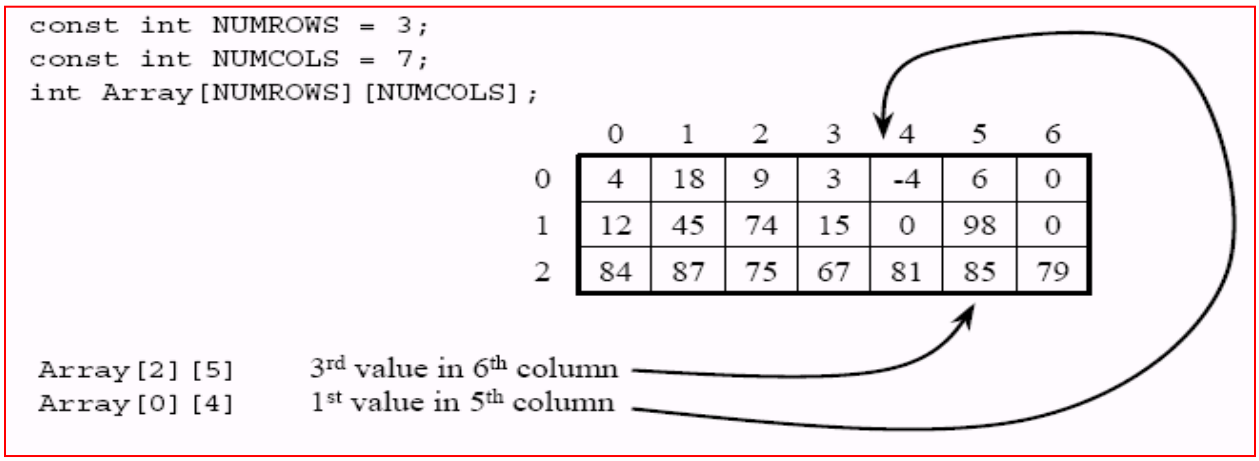
Input: The program prompts the user to enter 8 values.

Sum Calculation: As the values are entered, they are added to the sum.

Average Calculation: After summing the values, the average is calculated by dividing the sum by the number of elements (8).

Output: The program displays the entered values, the sum, and the average.

Multidimensional arrays



Multidimensional arrays can be described as arrays of arrays. For example, a two-dimensional array consists of a certain number of rows and columns:

A one-dimensional array is usually processed via a 'for' loop. Similarly, a two-dimensional array may be processed with a nested for loop:

Write a program which prompts the user to enter values into a 2X3 matrix and then display the values

```
#include <iostream>

using namespace std;

int main() {

    const int ROWS = 2;

    const int COLS = 3;

    int matrix[ROWS][COLS];

    // Prompt the user to enter values for the matrix

    cout << "Enter values for a 2x3 matrix:" << endl;

    for (int i = 0; i < ROWS; i++) {

        for (int j = 0; j < COLS; j++) {

            cout << "Enter value for matrix[" << i << "][" << j << "]: ";
```

```
            cin >> matrix[i][j];
        }
    }
    // Display the matrix values
    cout << "\nThe 2x3 matrix is:" << endl;
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;  // Move to the next row
    }
    return 0;
}
```

Explanation:

Matrix Declaration: A 2D array matrix of size 2x3 is declared to store the values.

Input: The program prompts the user to input values for each element of the matrix using two nested loops (for rows and columns).

Output: After the input, the matrix is displayed in a 2x3 format by using another set of nested loops.

Assignment

Write a program which prompts the user to enter values into Matrix A, Matrix B add them and store the results in Matrix C. The program should display the three matrices

Advantages of Arrays:

- **Direct Access**: Elements can be accessed directly using their index, allowing for fast retrieval.

- **Ease of Use**: Arrays are simple to declare, initialize, and manipulate.
- **Memory Efficiency**: Arrays store data in contiguous memory locations, which can be more memory-efficient compared to other data structures like linked lists.

Disadvantages of Arrays:

- **Fixed Size**: The size of an array is fixed upon creation and cannot be resized, which can lead to wasted memory or insufficient space.
- **Inefficient Insertion/Deletion**: Inserting or deleting elements (except at the end) requires shifting other elements, making these operations inefficient.
- **Homogeneity**: Arrays can only store elements of the same data type, limiting flexibility.

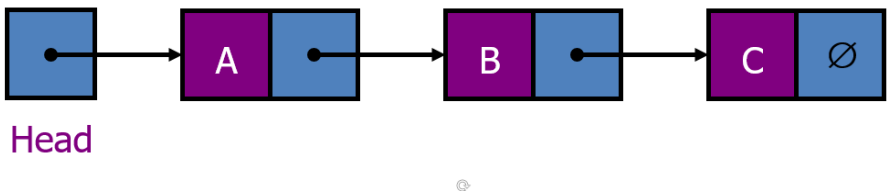
Applications of Arrays:

- **Storing Data**: Arrays are used to store lists of data, such as a list of numbers, characters, or other data types.
- **Implementing Other Data Structures**: Arrays are often used to implement other data structures like stacks, queues, and matrices.
- **Algorithm Implementation**: Many algorithms, such as sorting and searching algorithms, rely on arrays for their implementation.

Arrays are one of the most fundamental data structures and are widely used in programming due to their simplicity and efficiency in accessing data.

LINKED LIST

- A linked list is a data structure specially designed to overcome the limitations of a linked list
- The linked list is a very flexible dynamic data structure: items may be added to it or deleted from it at will.
- An array (linear list) allocates memory for all its elements lumped together as one block of memory.
- In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node".



- A linked list is a series of connected nodes
- Each node contains at least
    - A piece of data (any type)
    - Pointer to the next node in the list
- Head: pointer to the first node
- The last node points to NULL

Linked Lists: the List Node

The list node is a simple self-referential structure that stores an item of data, and a reference to the next item.

```
class Node {
public:
    int data;
    Node* next;
      // Default constructor
    Node()
    {
        data = 0;
        next = NULL;
    }
      // Parameterised Constructor
    Node(int data)
    {
        this->data = data;
        this->next = NULL;
    }
```

The data variable is where the information to be stored resides. It may be of any primitive or reference type appropriate for the data.

The next variable is the self-referential link to the next data item.

The constructor initialises the node object by storing the data that was given as an argument, and sets the next reference to `null`.

Linked Lists: the Header Class

The header class is the public interface for the linked list. It is where the functionality is stored (as methods), and contains a link to the first item of the list (the 'head'.)

```
class Linkedlist {
    Node* head;
 public:
    // Default constructor
    Linkedlist() { head = NULL; }

    // Function to insert a
    // node at the end of the
    // linked list.
    void insertNode(int);
      // Function to print the
    // linked list.
    void printList();
      // Function to delete the
    // node at given position
    void deleteNode(int);
};
```

The head variable is a reference to the first item in the list.

The constructor initialises the list by setting the head to `null` (an empty list.)

The methods provide a way to use the list. They each access the structure through the head reference.

Linked Lists: List Traversal (1)

It is sometimes necessary to traverse the entire length of the list to perform some function (for example, to count the number of items, or display summary information.)

Step 1: Step through the list from the header node forward.

Step 2: Perform the desired operation at that node.

Step 3: Move onto the next node, until the end of the list is reached.

List traversal forms the basis of many of the list manipulation operations such as add, retrieve and delete.

Linked Lists: List Traversal (2)

The code below will traverse the entire list, and print out the data contained in each node.

```
void Linkedlist::printList()
{
    Node* temp = head;

    // Check for empty list.
    if (head == NULL) {
        cout << "List empty" << endl;
        return;
    }
      // Traverse the list.
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
```

Step 1: Maintain a variable to store the current position in the list.

Step 2: Continue stepping through the list, until the end of the list (a `null` reference) is reached.

Step 3: At each step, print out the data present in the current node.

Because of the way a linked list is defined, we can only access data in one direction, and sequentially (one item after another.)

Linked Lists: Adding Data

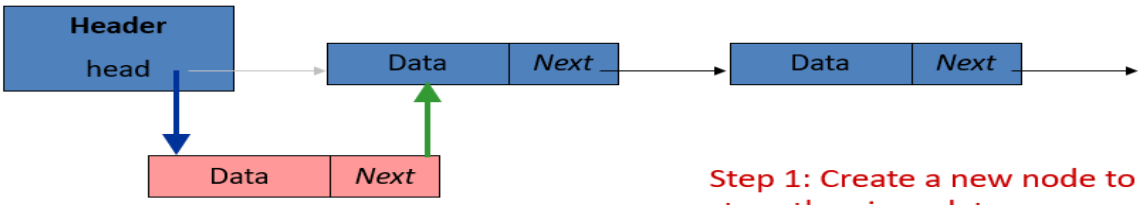
Data is added to a linked list by wrapping the data to add into a node, and then placing that node at the appropriate place in the data structure.

Depending on the circumstances and purpose of the list, there are a number of places where data may be added:

- **At the start (head) of a list**
- **In the middle of the list**
- **At the end (tail) of the list**
- **At the appropriate place to preserve sort order**

Linked Lists: Adding Data to the Head (1)

Adding data to the head of a list is the easiest and quickest way in which it can be done.



The order in which the link manipulations are done are very important; they must always be done from right to left, otherwise data nodes will be lost.

Step 1: Create a new node to store the given data.

Step 2: Set the new node's next reference to the first node.

Step 3: Reset the head reference to point to the newly created node.

Linked Lists: Adding Data to the Head (2)

The algorithm is simple to translate into source code, as each step corresponds with just one simple instruction.

```
Function to insert a new node.
void Linkedlist::insertNode(int data)
{
    // Create the new Node.
    Node* newNode = new Node(data);

    // Assign to head
    if (head == NULL) {
        head = newNode;
        return;
    }
```

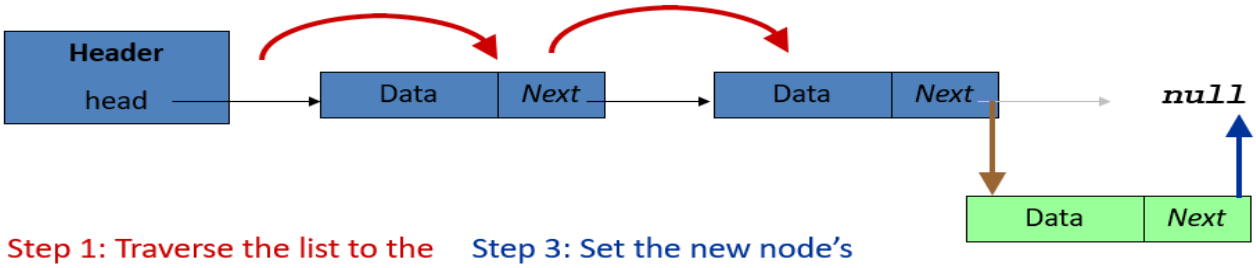
Step 1: Create a new node to store the given data.

Step 2: Set the new node's next reference to the first node.

Step 3: Reset the head reference to point to the newly created node.

Linked Lists: Adding Data to the Middle or Tail (1)

Adding data to the middle or tail of the list is essentially the same process. The diagram below illustrates adding to the end (tail.)



Step 1: Traverse the list to the desired insertion point (in this case, the last item of the list.)

Step 2: Create a new node to store the given data.

Step 3: Set the new node's next reference to that of the insertion point node.

Step 4: Reset the insertion point's next reference to point to the newly created node.

Linked Lists: Adding Data to the Middle or Tail (2)

```
public void addToTail(int data)
{
    ListNode insert = head;

    while (insert.next != null)
        insert = insert.next;

    ListNode newNode = new ListNode(data);

    newNode.next = insert.next;

    insert.next = newNode;
}
```

Step 1: Traverse the list to the desired insertion point (in this case, the last item of the list.)

Step 2: Create a new node to store the given data.

Step 3: Set the new node's next reference to that of the insertion point node.

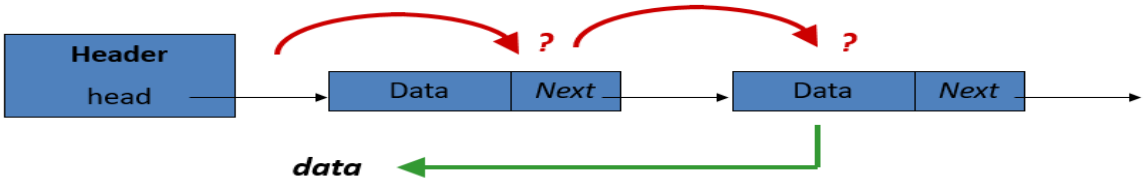
Step 4: Reset the insertion point's next reference to point to the newly created node.

Tip: the traversal in step 1 could have been avoided by maintaining a tail pointer in the header class (as well as a head.)

Linked Lists: Retrieving Data (1)

Data retrieval consists of traversing the data structure until a matching node is found. The data portion of the node is then returned (if the data is not found, some form of failure signal should be returned instead.)

The whole data node should not be returned, as it is part of the list's internal structure.



Step 1: Traverse the list.

Step 2: While traversing, compare the search 'key' value with the data in each node.

If the data keys match, return the data portion of the node.

Step 3: If the list is exhausted without the data being found, return a search failure signal.

Linked Lists: Retrieving Data (2)

```
public int retrieve(int key)
{
    ListNode current = head;

    while (current != null)
    {
        if (current.data == key)
            return data;

        current = current.next;
    }

    return -1;
}
```

Step 1: Traverse the list.

Step 2: While traversing, compare the search 'key' value with the data in each node.

If the data keys match, return the data portion of the node.

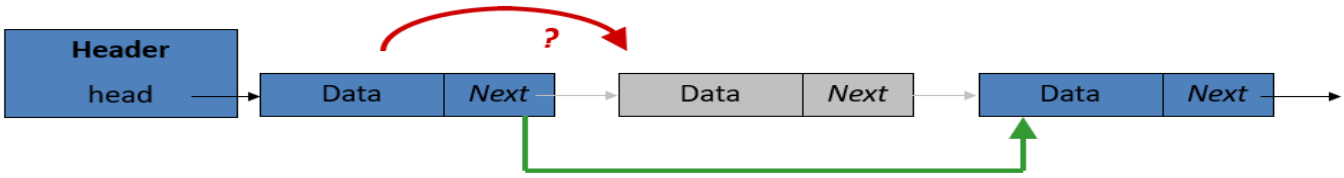
Step 3: If the list is exhausted without the data being found, return a search failure signal.

Note: this program assumes that only positive integers are being stored. This way, the calling program can easily assume that a non-positive answer (e.g. -1) is the signal for a failed retrieval attempt.

Linked Lists: Deleting Data (1)

Deletion is very similar to retrieval. As before, the list is traversed to find data matching a given 'key' value. However, instead of returning the data, the node is to be deleted.

The node can be deleted by having the next references 'jump over' the node to delete. To do this, the node before the one to delete must be known, and as such, the traversal needs to keep track of two references.



Step 1: Traverse the list, maintaining both current and previous references.

Step 2: If the search key matches the current data node, 'jump over' the current node, and return success.

Step 3: If the list is exhausted without the data being found, return a search failure signal.

Linked Lists: Deleting Data (2)

```
public boolean delete (int key)
{
    ListNode current = head;
    ListNode previous = null;

    while (current != null)
    {
        if (current.data == key)
        {
            previous.next = current.next;
            return true;
        }

        previous = current;
        current = current.next;
    }

    return false;
}
```

Step 1: Traverse the list, maintaining both current and previous references.

Step 2: If the search key matches the current data node, 'jump over' the current node, and return success.

Step 3: If the list is exhausted without the data being found, return a search failure signal.

Note: a **boolean** variable is used in this code to return success (**true**) and failure (**false**).

C++ program that allows the user to insert values anywhere in a singly linked list, delete a value, and display the list:

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

// Node class to represent each element in the list
class Node {
public:
    int data;       // Data of the node
    Node* next;     // Pointer to the next node
    // Constructor
    Node(int value) : data(value), next(NULL) {}  // Using NULL instead of nullptr
};
// LinkedList class to manage the nodes
class LinkedList {
private:
    Node* head;  // Pointer to the head of the list
public:
    // Constructor
    LinkedList() : head(NULL) {}  // Using NULL instead of nullptr

    // Destructor
    ~LinkedList();
    // Insert a node at a specific position
    void insertAtPosition(int value, int position);
    // Delete a node by value
    bool deleteNode(int value);
```

```
    // Display the linked list
    void displayList() const;
};
// Destructor to delete the list and free memory
LinkedList::~LinkedList() {
    Node* current = head;
    Node* nextNode;
    // Delete all nodes
    while (current != NULL) {  // Using NULL instead of nullptr
        nextNode = current->next;
        delete current;
        current = nextNode;
    }
    head = NULL;  // Set head to NULL after deletion
}

// Insert a node at a specific position
void LinkedList::insertAtPosition(int value, int position) {
    Node* newNode = new Node(value);

    // If inserting at the head (position 1)
    if (position == 1) {
        newNode->next = head;
        head = newNode;
        return;
    }
```

```
    // Traverse to the position before where the new node should be inserted
    Node* current = head;
    for (int i = 1; i < position - 1 && current != NULL; i++) {
        current = current->next;
    }

    // If the position is valid
    if (current != NULL) {
        newNode->next = current->next;
        current->next = newNode;
    } else {
        cout << "Invalid position!" << endl;
        delete newNode;
    }
}
// Delete a node by its value
bool LinkedList::deleteNode(int value) {
    // If the list is empty
    if (head == NULL) {
        return false;
    }
    // If the node to delete is the head
    if (head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return true;
```

```
    }

    // Traverse to find the node to delete
    Node* current = head;
    while (current->next != NULL && current->next->data != value) {
        current = current->next;
    }
    // If the node was found, delete it
    if (current->next != NULL) {
        Node* temp = current->next;
        current->next = current->next->next;
        delete temp;
        return true;
    }
    return false;  // Node not found
}
// Display the linked list
void LinkedList::displayList() const {
    Node* current = head;
    if (current == NULL) {
        cout << "The list is empty." << endl;
        return;
    }
    // Traverse and print the list
    while (current != NULL) {
        cout << current->data << " -> ";
        current = current->next;
```

```
    }
    cout << "NULL" << endl;
}

// Main function
int main() {
    LinkedList list;
    int choice, value, position;
    do {
        cout << "\nMenu:\n";
        cout << "1. Insert a value at a position\n";
        cout << "2. Delete a value\n";
        cout << "3. Display the list\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
        case 1:
            cout << "Enter value to insert: ";
            cin >> value;
            cout << "Enter position to insert at (1-based index): ";
            cin >> position;
            list.insertAtPosition(value, position);
            break;

        case 2:
            cout << "Enter value to delete: ";
```

```
            cin >> value;
            if (list.deleteNode(value)) {
                cout << "Value " << value << " deleted successfully." << endl;
            } else {
                cout << "Value " << value << " not found in the list." << endl;
            }
            break;
        case 3:
            cout << "Current list: ";
            list.displayList();
            break;
        case 4:
            cout << "Exiting..." << endl;
            break;
        default:
            cout << "Invalid choice! Please try again." << endl;
        }

    } while (choice != 4);

    return 0;
}
```

Explanation:

Insert at Position:

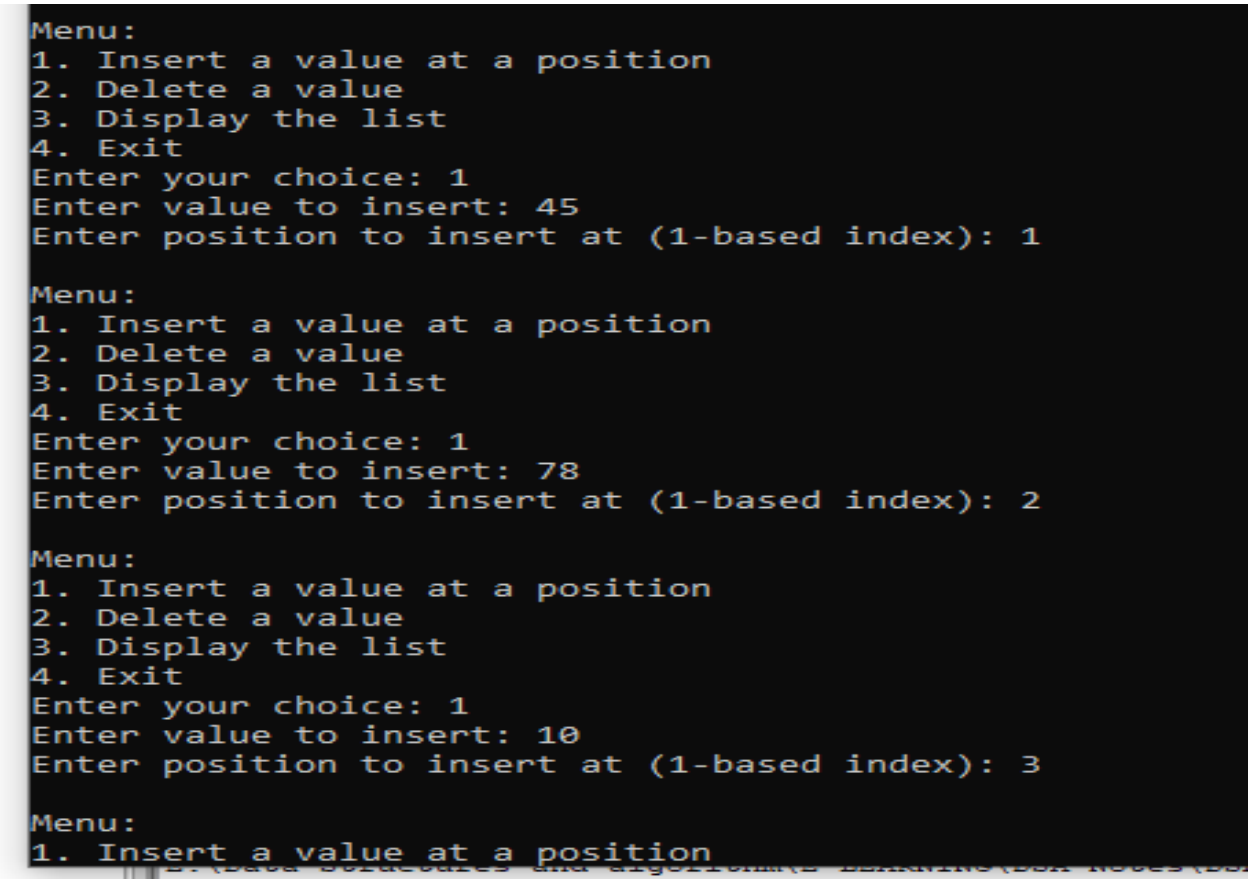
The user is prompted to enter a value and a position where they want to insert the node. The function checks if the position is valid and inserts the node accordingly.

Delete a Node:

The user is prompted to enter a value to delete. The function searches for the node with that value and deletes it if found.

Display List:

The current list is displayed in a formatted manner showing the sequence of nodes.



Assignment

Discuss the various types of linked lists, illustrating them using a diagram. Give advantages and disadvantages of each of them

o Singly Linked List: Each node points to the next node, and the last node points to NULL.

o Doubly Linked List: Each node has two pointers, one pointing to the next node and one to the previous node.

o Circular Linked List: The last node points to the first node, making a circle.

Advantages of Linked Lists:

1. **Dynamic Size**:
    - Linked lists can grow and shrink dynamically at runtime, unlike arrays, which have a fixed size. This allows efficient memory usage as the list size can adapt to the number of elements needed.
2. **Efficient Insertion/Deletion**:
    - Insertion and deletion of nodes can be done efficiently (especially at the beginning or middle) without having to shift elements, unlike arrays. Operations are typically O(1) for insertion/deletion at the head, as only pointers need to be adjusted.
3. **No Wasted Memory**:
    - Memory allocation in linked lists is flexible and happens when nodes are needed, so there is no risk of allocating extra unused space (unlike arrays, where unused space might be allocated).
4. **Easy Implementation of Data Structures**:
    - Linked lists form the basis for other data structures like stacks, queues, and graphs. Their dynamic nature makes it easy to implement these more complex data structures.
5. **Efficient Memory Utilization**:
    - When the number of elements is unknown or changes frequently, linked lists make better use of memory compared to arrays, where memory is pre-allocated, potentially leading to unused space.
6. **No Contiguous Memory Requirement**:
    - Linked lists do not require elements to be stored in contiguous memory locations, which is particularly useful when memory is fragmented and a large block of memory is unavailable.

Disadvantages of Linked Lists:

1. **Increased Memory Overhead**:
   - Each node in a linked list requires additional memory to store a pointer/reference to the next (or previous) node. For large lists, this overhead can be significant, especially in doubly linked lists.
2. **Sequential Access**:
   - Linked lists do not support direct access to elements by index like arrays (which provide constant time access O(1) for reading an element). To access an element in a linked list, you must traverse the list from the head to the desired node, leading to a time complexity of O(n) for search operations.
3. **More Complex Operations**:
   - Linked lists require more complex manipulation for operations such as insertion, deletion, and traversal. Careful pointer management is necessary to avoid memory leaks or accidentally corrupting the structure of the list.
4. **Extra Memory for Pointer Management**:
   - Since each node in a linked list requires extra memory for pointers (next and possibly previous in the case of a doubly linked list), linked lists are less space-efficient compared to arrays, especially for smaller data types.
5. **More Prone to Memory Leaks**:
   - Incorrect handling of pointers in linked lists can lead to memory leaks, especially if nodes are not properly deleted or pointers are incorrectly managed.

Summary:

Advantages	Disadvantages
Dynamic size that can grow or shrink during runtime	Increased memory overhead due to pointers
Efficient insertion and deletion without shifting elements	Sequential access makes searching and accessing elements slower
No wasted memory; no pre-allocation required	More complex operations, prone to pointer management errors
No contiguous memory required, useful in fragmented memory	Poor cache locality, leading to slower access times
Flexible implementation of stacks, queues, and graphs	More memory required for storing pointers, especially in large lists

Linked lists are versatile data structures with unique advantages in dynamic scenarios but come with overhead in memory and operational complexity compared to simpler structures like arrays. They are particularly useful when the size of the dataset is unpredictable or when frequent insertions and deletions are required.

Summary of Key Applications:

Application	Linked List Type	Use Case
Dynamic memory allocation	Singly Linked List	Managing heap memory in operating systems
Undo/redo operations	Doubly Linked List	Text editors and drawing applications
Playlist management	Doubly/Circular Linked List	Music and video playlists, easy forward and backward navigation
Browser history management	Doubly Linked List	Storing and navigating web page history
Sparse matrix representation	Singly Linked List	Efficient memory usage for large matrices with mostly zero values
Round-robin scheduling	Circular Linked List	Process scheduling in operating systems
Graph representation	Singly Linked List	Storing neighbors of each vertex in graph algorithms
Polynomial arithmetic	Singly Linked List	Representing terms of polynomials for algebraic operations
Real-time scheduling	Singly/Circular Linked List	Task management in real-time operating systems
Hash table collision handling	Singly Linked List	Resolving collisions using chaining in hash table implementations

Conclusion:

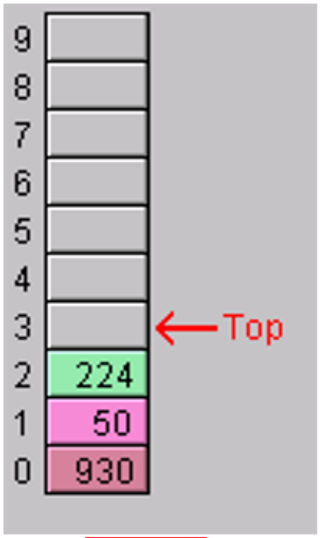
Linked lists provide flexibility, efficient memory usage, and dynamic data storage, making them useful in many practical applications across software systems, including operating systems, real-time systems, multimedia applications, and graph-based algorithms. Their ability to handle dynamic data with frequent insertions and deletions makes them ideal for scenarios where data grows or changes unpredictably.

STACK

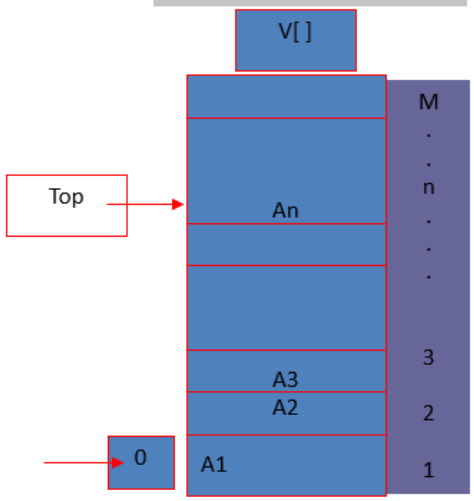
- What is a Stack?
- Stack implementation using array.
- Stack implementation using linked list.
- Applications of Stacks.

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the last element added to the stack is the first one to be removed. Think of it like a stack of CDs: you add a CD to the top and remove them from the top as well.

Stacks: Definition:



- Is a Linear list in which all insertions and deletions are made at one end called the Top. E.g. in Linear List <a_1 a_2 ...a_n > deletion and insertion can only be on element an (Top element).
- It is a LIFO (Last In First Out) list
- Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack. just like a pile of objects.



- The storage is downwards-up so that all operations are on Top element.

- A pointer is restricted to the Top element.

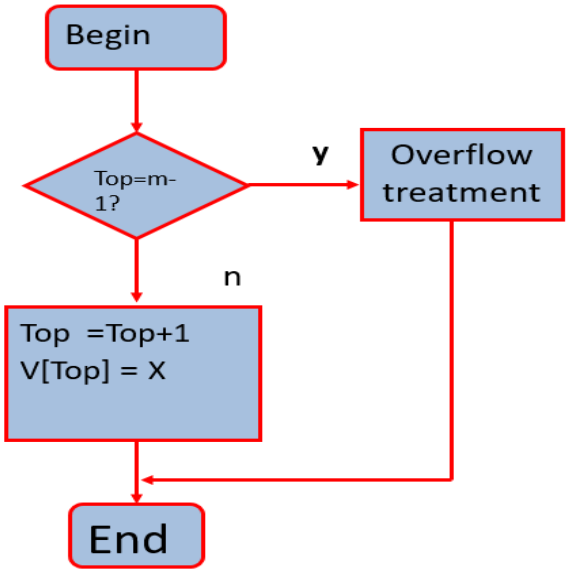
- If there is no element in stack (n=0.) The top will be -1
  Note:
- Index used from 0 i.e. V[0].....V[n-1].
- [Top = -1 means empty Stack]
- [Top = M-1 means full Stack] for an M-capacity storage Stack.

Stack Operations

- makeNull(s) – make s be an empty stack
- top(s) – return the element at the top of the stack
- pop(s) – return and delete the element at the top of the stack. The stack size reduces by 1
- push(x,s) – insert element x at the top of the stack. The size of the stack increases by one.
- empty(s) – returns true if the stack has no elements.

Insertion in a stack

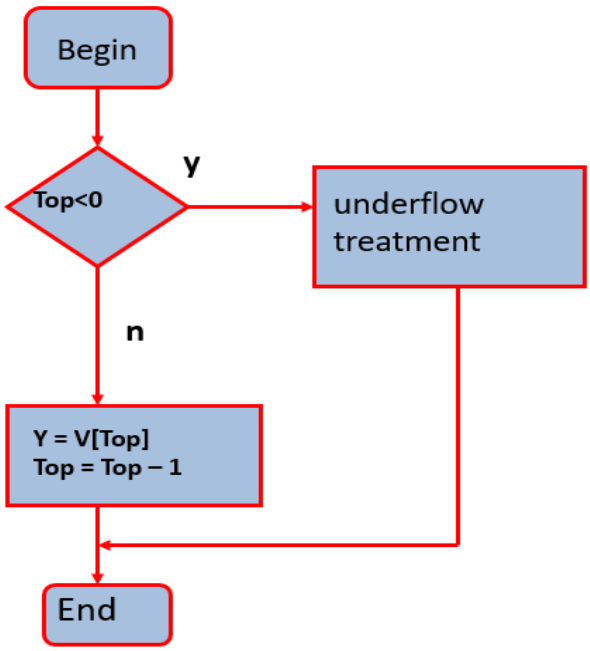
- For example inserting new element say X (Fig)
- A test is made of whether Stack is full: If full then abort the procedure or else move pointer Top to position Top+1 then insert element X, i.e. V[Top] = X
- X becomes the new top element:



26

Stack:Deleting an element (Top element)

- Declare temporary variable Y is to store the deleted element:
- If Stack is empty i.e. Top = -1 then there occurs what is known as an underflow.
- Otherwise delete the top element that is, V[top] , and adjust pointer Top to (Top-1).
- The deleted element Y may be printed out



27

C++ program to an array implement Push() and Push() operations in a stack and display values

```
#include <iostream>

using namespace std;


#define MAX 5 // Maximum size of the stack


class Stack {

private:

    int arr[MAX];  // Array to hold the stack elements

    int top;       // Index of the top element in the stack
```

```
public:
    // Constructor to initialize the stack
    Stack() {
        top = -1;  // Stack is initially empty
    }
    // Push operation to add an element to the stack
    void push(int value) {
        if (top >= MAX - 1) {
            cout << "Stack Overflow! Cannot push " << value << endl;
        } else {
            top++;
            arr[top] = value;
            cout << value << " pushed into stack." << endl;
        }
    }
    // Pop operation to remove the top element from the stack
    void pop() {
        if (top < 0) {
            cout << "Stack Underflow! No element to pop." << endl;
        } else {
            cout << arr[top] << " popped from stack." << endl;
            top--;
        }
    }
    // Display operation to show all the elements of the stack
    void display() {
        if (top < 0) {
```

```
            cout << "Stack is empty!" << endl;
        } else {
            cout << "Stack elements: ";
            for (int i = 0; i <= top; i++) {
                cout << arr[i] << " ";
            }
            cout << endl;
        }
    }
};
int main() {
    Stack stack;
    int choice, value;
    do {
        cout << "\n1. Push\n2. Pop\n3. Display\n4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
        case 1:
            cout << "Enter value to push: ";
            cin >> value;
            stack.push(value);
            break;
        case 2:
            stack.pop();
            break;
        case 3:
```

```
            stack.display();
            break;
        case 4:
            cout << "Exiting..." << endl;
            break;
        default:
            cout << "Invalid choice! Please enter again." << endl;
        }
    } while (choice != 4);
    return 0;
}
```

Explanation of the Program:

Stack Class:

The arr[MAX] array is used to store the stack elements.

The top variable keeps track of the index of the top element in the stack.

Operations:

Push: Adds an element to the top of the stack if it's not full. If the stack is full, it shows a "Stack Overflow" message.

Pop: Removes the top element from the stack if it's not empty. If the stack is empty, it shows a "Stack Underflow" message.

Display: Prints all elements in the stack from bottom to top.

Menu:

The program provides a menu to the user for pushing, popping, displaying the stack, or exiting the program.

```
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 67
67 pushed into stack.

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 32
32 pushed into stack.

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements: 67 32

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
32 popped from stack.

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements: 67

1. Push
2. Pop
3. Display
4. Exit
Enter your choice:
```

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size.

Stack Applications

A stack is a versatile data structure widely used in various applications due to its Last In, First Out (LIFO) behavior. Here are some key applications of a stack:

1. Function Call Management (Call Stack)

Application: When functions are called in a program, the system uses a stack to manage the function calls. Each time a function is invoked, its data (return address, local variables, etc.) is pushed onto the stack. When the function returns, its data is popped from the stack.

Example: In recursive function calls, each recursive call is pushed onto the call stack, and the execution of the function resumes when the last call is popped from the stack.

2. Expression Evaluation and Conversion

Application: Stacks are used to evaluate expressions (infix, prefix, and postfix) and convert between different types of expressions.

Example:

Postfix (Reverse Polish Notation) Evaluation: Stacks are used to evaluate expressions like 5 6 + 7 *, where operands are pushed to the stack, and operators pop and operate on the operands.

Infix to Postfix Conversion: Stacks are used to convert expressions like (A + B) * C (infix) to A B + C * (postfix).

3. Backtracking Algorithms

Application: Stacks are commonly used in backtracking algorithms, where the system needs to backtrack to the previous state to find a solution to a problem.

4. Undo/Redo Features

Application: Stacks are used in software applications like text editors or graphic design tools to implement undo and redo features.

Example:

Undo: Each action (like typing a word) is pushed onto the stack. When the user presses undo, the last action is popped and undone.

Redo: The popped action is stored in another stack, and can be redone if needed by popping it back.

5. Browser History

Application: Web browsers use stacks to manage the history of visited websites.

Example: When a user navigates to a new webpage, the current page is pushed onto the stack. Pressing the "back" button pops the current page and loads the previous one. The "forward" button restores the last popped page.

6. Depth-First Search (DFS)

Application: Stacks are essential in graph traversal algorithms like Depth-First Search (DFS), where nodes are explored as deeply as possible before backtracking.

Example: DFS uses a stack (either explicitly or implicitly via recursion) to keep track of which nodes to visit next. It explores one branch fully before backtracking to explore other branches.

7. Memory Management

Application: The stack is used by operating systems to manage the memory for variables and function calls in the execution of programs.

Example: Local variables and return addresses are stored on the stack in functions. Once the function completes execution, its stack frame is popped, freeing the memory.

9. String Reversal

Application: Stacks can reverse the order of characters in a string.

Example: Push all characters of the string onto a stack, then pop them off one by one, which results in the reversed string.

Advantages of Stack

- Easy to Implement
- LIFO (Last In, First Out) Order:
  Stacks operate on the principle of Last In, First Out, which is ideal for certain types of tasks, such as undo operations, function call management, and backtracking algorithms.
- Efficient Memory Use:
  In stack-based memory management (like in function call management), memory allocation and deallocation are fast and efficient, as elements are only added or removed from the top of the stack. This reduces the need for complex memory management techniques.
- Simple and Fast Operations:

Operations like push (inserting an element), pop (removing an element), and peek (viewing the top element) are simple and can be performed in constant time, i.e., O(1).

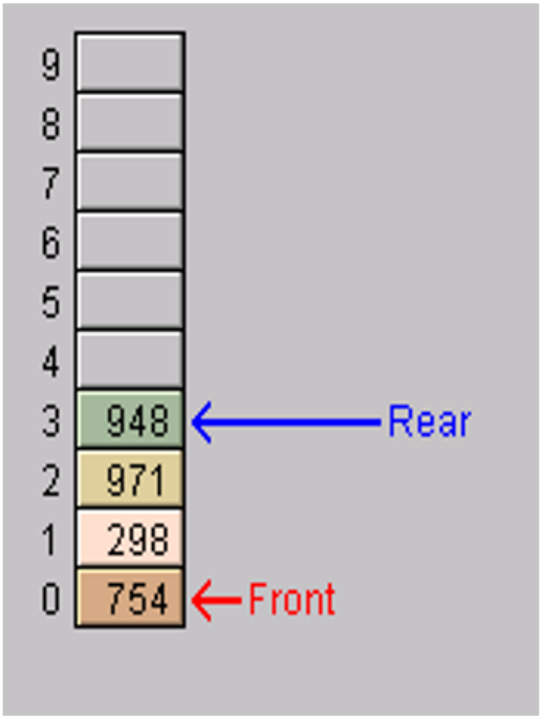
Disadvantages of Stack:

- Limited Access to Elements:
  A stack only allows access to the top element. To access any other element, all elements above it must be removed. This makes it inefficient when random access to elements is needed.
- Size Limitation:
  In the case of fixed-size stacks (such as when using an array), the stack can be prone to overflow (if too many elements are pushed) or underflow (if trying to pop from an empty stack). Dynamic stacks (using linked lists) can overcome this but at the cost of additional memory management.
- Overflow and Underflow Issues:
  Stack overflow occurs if too many elements are pushed into the stack (especially with a fixed-size array). Stack underflow occurs if an attempt is made to pop an element from an empty stack.
- Not Suitable for All Problems:
  Stacks are efficient for problems that require LIFO operations, but they are not well-suited for problems requiring more flexible data access, such as searching or inserting elements in the middle.
- Inefficient in Multi-Level Access:
  To access elements deeper in the stack, you need to pop all elements above it, which is inefficient for problems requiring multi-level access.

QUEUE

A queue is a linear data structure that operates on the FIFO (First In, First Out) principle, meaning that the first element added to the queue will be the first one to be removed. It is analogous to a real-life queue, such as a line of people waiting for a service: the first person in line is served first, and new people join the line at the end.

- Is a linear list in which all insertions and deletions are restricted:
- Uses FIFO algorithm
- All insertions into the queue take place at one end called the rear while all deletions take place at the other end called the front



Queue operations

- makeNull(q) – makes a queue empty and returns an empty queue
- peek(q) – returns the first element on a queue
- enqueue(x,q) – inserts element q at the end of the queue
- dequeue(q)– deletes the first element element of the queue
- Empty(q) – returns true iff the queue is empty.

Program to Implement Enqueue, Dequeue and Display operations in a queue

```
#include <iostream>
using namespace std;
#define MAX 5 // Maximum size of the queue
class Queue {
    int front, rear;
    int queue[MAX]; // Array to store elements
public:
    Queue() {
```

```
    front = -1;
    rear = -1;
}
// Enqueue operation to add an element to the queue
void Enqueue(int value) {
    if (rear == MAX - 1) {
        cout << "Queue is full! Overflow condition.\n";
    } else {
        if (front == -1) front = 0; // Initialize front to 0 on first enqueue
        rear++;
        queue[rear] = value;
        cout << "Enqueued: " << value << endl;
    }
}
// Dequeue operation to remove an element from the queue
void Dequeue() {
    if (front == -1 || front > rear) {
        cout << "Queue is empty! Underflow condition.\n";
    } else {
        cout << "Dequeued: " << queue[front] << endl;
        front++;
        if (front > rear) { // Reset the queue when all elements are dequeued
            front = rear = -1;
        }
    }
}
// Display operation to show the queue elements
```

```
    void Display() {
        if (front == -1) {
            cout << "Queue is empty!\n";
        } else {
            cout << "Queue elements: ";
            for (int i = front; i <= rear; i++) {
                cout << queue[i] << " ";
            }
            cout << endl;
        }
    }
};
int main() {
    Queue q;
    int choice, value;
    do {
        cout << "\nQueue Operations:\n";
        cout << "1. Enqueue\n";
        cout << "2. Dequeue\n";
        cout << "3. Display\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
        case 1:
            cout << "Enter value to enqueue: ";
            cin >> value;
```

```
            q.Enqueue(value);
            break;
        case 2:
            q.Dequeue();
            break;
        case 3:
            q.Display();
            break;
        case 4:
            cout << "Exiting...\n";
            break;
        default:
            cout << "Invalid choice! Please try again.\n";
        }
    } while (choice != 4);
    return 0;
}
```

Explanation of Operations:

Enqueue Operation:

Adds a value to the rear of the queue if the queue is not full. If it's full, an overflow message is displayed.

Dequeue Operation:

Removes the value from the front of the queue if the queue is not empty. If it's empty, an underflow message is displayed.

Display Operation:

Shows all the elements in the queue from front to rear.

Types of Queues:

- Simple Queue:
  Also known as a linear queue, this is the most basic type of queue. It allows operations at the front and rear but suffers from a problem where the front moves forward, and the unused space before it cannot be reused without resetting the queue.
- Circular Queue:
  In a circular queue, the positions in the array are reused. When the rear reaches the last position, it wraps around to the front if there is space, solving the issue of wasted space in a simple queue.
- Priority Queue:
  A priority queue allows elements to be dequeued based on their priority, not their position. Elements with a higher priority are dequeued before elements with a lower priority, regardless of their order of arrival.
- Double-Ended Queue (Deque):
  A deque (double-ended queue) allows insertion and deletion of elements from both ends (front and rear). It can act as both a stack and a queue, depending on how elements are enqueued and dequeued.

Applications of Queue:

- CPU Scheduling:
  In multi-tasking operating systems, processes are scheduled in the CPU using queues. A ready queue stores processes waiting for CPU time, while a job queue holds all processes to be executed.
- Breadth-First Search (BFS) in Graphs:
  A queue is used to traverse a graph using BFS. In this algorithm, nodes are visited layer by layer, and the queue helps keep track of nodes to be explored next.
- Handling Requests in a Web Server:
  A web server uses a queue to manage incoming requests. As requests arrive, they are enqueued, and the server processes them one by one, in the order they arrived.
- Printer Queue:
  A printer processes print jobs in the order they are received. The print jobs are queued and printed in a FIFO manner.
- Simulating Real-Life Queues:
  Queues are used to model scenarios like customer service counters, ticketing systems, or checkout lines, where customers are served in the order they arrive.
- Asynchronous Data Transfer (Buffering):

Queues are used for buffering in data streams. For example, in communication protocols, data is sent from one process to another using queues to maintain order in asynchronous data transfer.

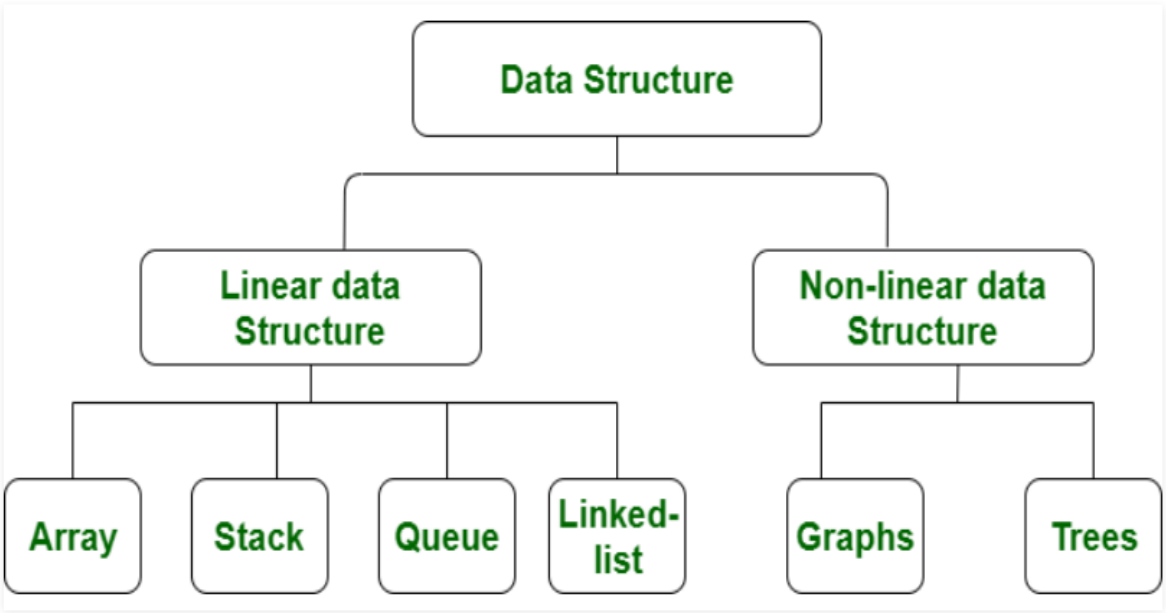
- Load Balancing:
  In distributed systems or cloud environments, queues are used to manage the distribution of tasks to multiple servers to ensure load balancing.

Advantages of Queue:

- FIFO Order:
  Ensures that tasks are processed in the order they arrive, which is suitable for applications like task scheduling and buffering.
- Efficient for Scheduling:
  Queues are ideal for managing tasks that need to be processed sequentially, such as CPU scheduling and process management.
- Simple to Implement:
  Queue operations like enqueue and dequeue are simple and efficient, requiring only constant time, i.e., O(1) for each operation.
- Memory Reuse in Circular Queues:
  In circular queues, memory is reused, which makes it more space-efficient compared to a simple queue.

Disadvantages of Queue:

- Fixed Size (Array-based Queue):
  A queue implemented using an array has a fixed size, which can lead to overflow if the queue is full, or underutilization if the queue is mostly empty.
- Inefficient Memory Use (Simple Queue):
  In a simple queue, once elements are dequeued, the space they occupied cannot be reused, leading to wasted memory.
- Limited Access to Elements:
  Unlike other data structures (like arrays or linked lists), you cannot access arbitrary elements in a queue without dequeuing all preceding elements.
- Not Suitable for Dynamic Data:
  For applications where elements need to be inserted or removed from arbitrary positions, a queue is not suitable because it only allows insertions at the rear and deletions from the front.



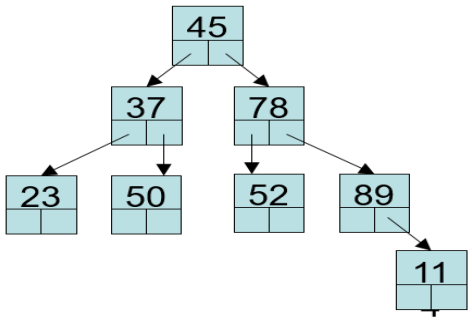
Sr. No.	Key	Linear Data Structures	Non-linear Data Structures
1	Data Element Arrangement	In linear data structure, data elements are sequentially connected and each element is traversable through a single run.	In non-linear data structure, data elements are hierarchically connected and are present at various levels.
2	Levels	In linear data structure, all data elements are present at a single level.	In non-linear data structure, data elements are present at multiple levels.
3	Implementation complexity	Linear data structures are easier to implement.	Non-linear data structures are difficult to understand and implement as compared to linear data structures.

4	Traversal	Linear data structures can be traversed completely in a single run.	Non-linear data structures are not easy to traverse and needs multiple runs to be traversed completely.
5	Memory utilization	Linear data structures are not very memory friendly and are not utilizing memory efficiently.	Non-linear data structures uses memory very efficiently.
6	Time Complexity	Time complexity of linear data structure often increases with increase in size.	Time complexity of non-linear data structure often remain with increase in size.
7	Examples	Array, List, Queue, Stack.	Graph, Map, Tree.

BINARY TREE

A binary tree is a non-linear hierarchical data structure in which each node has at most two children, referred to as the left child and right child. This structure is widely used due to its simplicity and efficiency in handling hierarchical data.

.The tree structure is naturally suited to many applications (e.g. databases, file systems, web sites, etc.), and can often allow algorithms to be significantly more efficient.

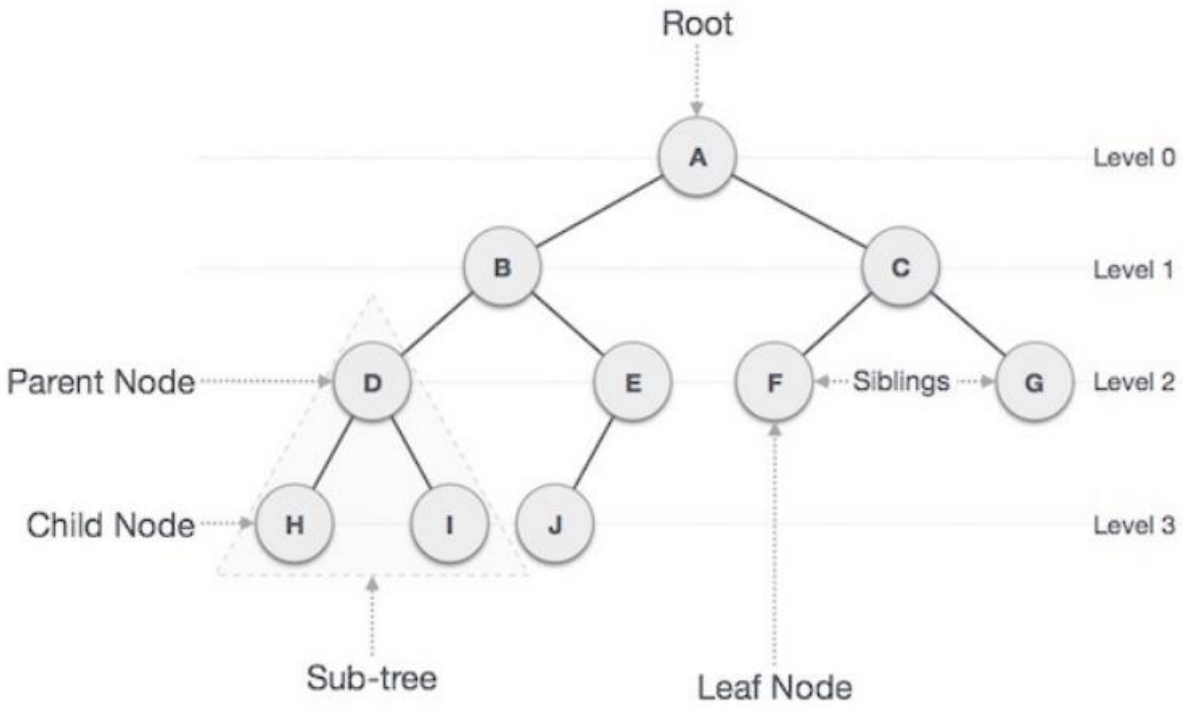


A tree (in data structure terms) is defined as a structure in which a set of *nodes* are connected together by their *edges*, in a parent-child relationship.

A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child

Key Terminologies:



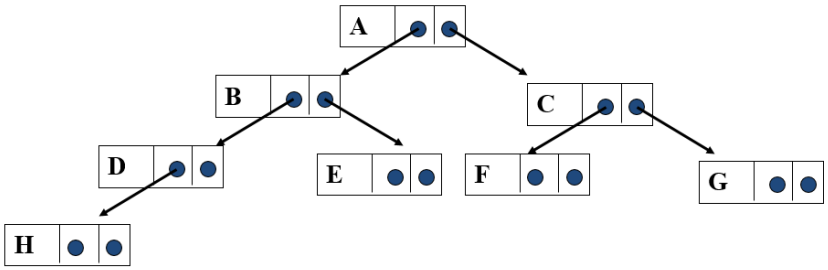
- Node:
  The fundamental unit of a binary tree, consisting of a data value and two pointers (or references) to its left and right children.
- Root:
  The topmost node of the binary tree. The entire tree is accessed from the root.

- Parent/Children/Siblings
  The unique predecessor of a node is called its parent,
  A node's successors are its children. A root node has no parent.
  Children of the same parent are called siblings
- Leaf Node:
  A node with no children. These are the terminal nodes of the binary tree.

- Branches/path
  The lines connecting the nodes are called branches. A sequence of branches from one node to another node lower down the tree is called a path
  The last node along any given path is known as a leaf. A leaf has no children.

- Degree of a node: number of brances/subtrees of a node(0,1,2)

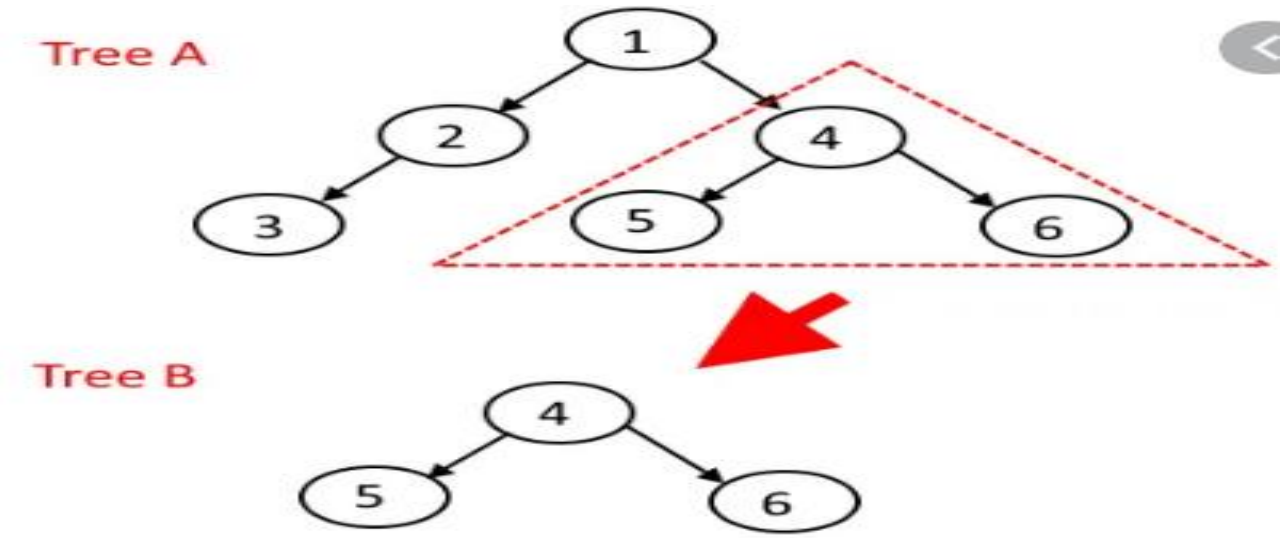
  Nodes of degree 0 are leaf nodes



  Degree of C=2, D=1 G=0

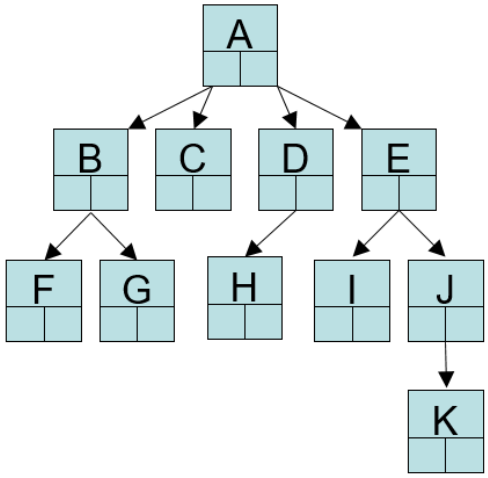
- Size of a tree: the number of nodes in the whole tree
  Size=7

- Height of a Node:
  The number of edges from the node to the deepest leaf. The height of the tree is the height of the root node.
  B=2 C=1 G=0
- Depth of a Node:
  The number of edges from the root to the node.
  H=3 B=1
- Level:
  The level of a node is determined by how far it is from the root. The root is at level 0, its children are at level 1, and so on.
- Subtree:
  A subtree refers to any node along with its descendants. Every node in a binary tree can be the root of its own subtree.



Tree B is a subtree of Tree A

- the **depth** of a node is the length of the path from the root to the node. (The root has a depth of 0.)
- the **height** of a node is the length of the path from the node to the deepest leaf.
- the **size** of a node is the number of nodes in the subtree rooted at that node (including itself.)

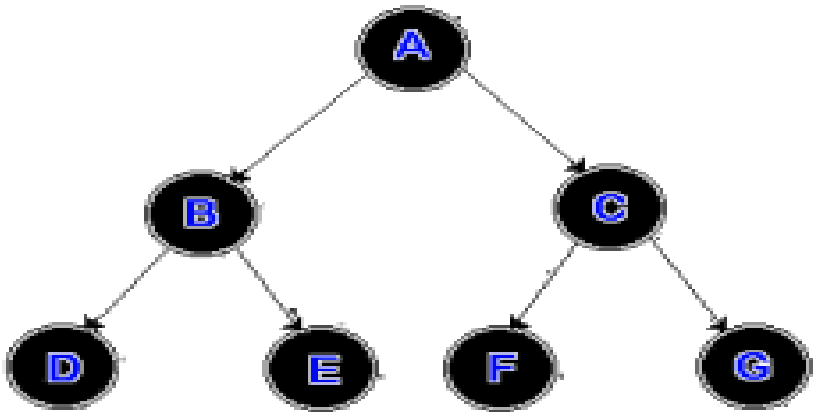


	A	B	C	D	E	F	G	H	I	J	K
Depth	0	1	1	1	1	2	2	2	2	2	3
Height	3	1	0	1	2	0	0	0	0	1	0
Size	11	3	1	2	4	1	1	1	1	2	1

Types of Binary Trees:

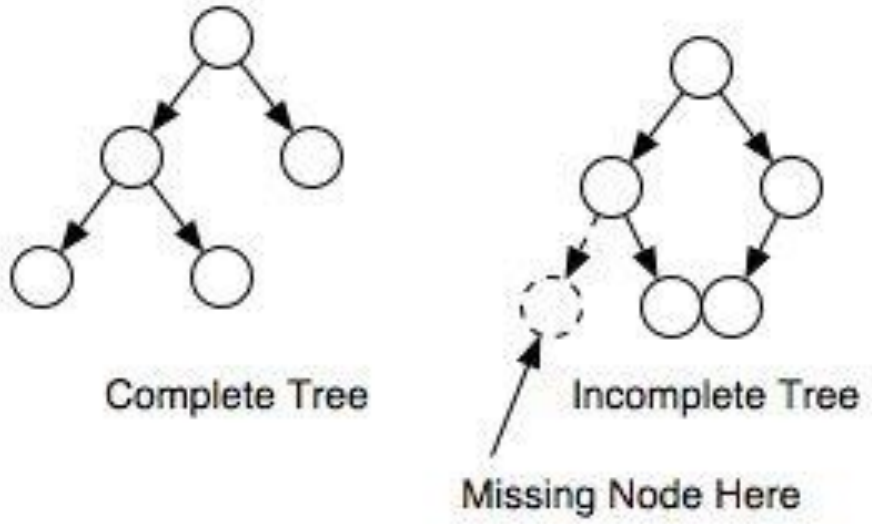
Full Binary Tree:

Every node in the tree has either 0 or 2 children. No node in a full binary tree has only one child.



Complete Binary Tree:

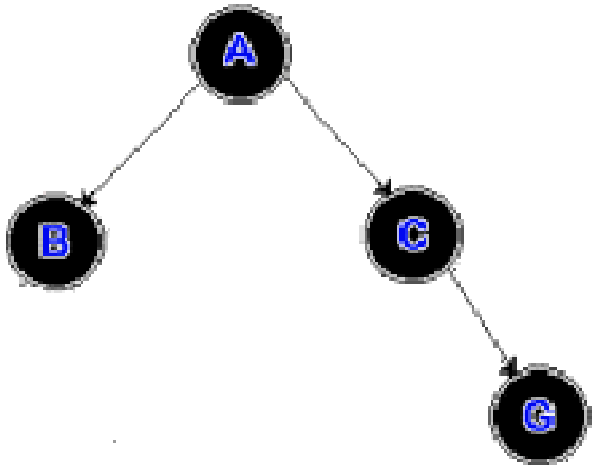
All levels of the tree are completely filled, except possibly the last level, where nodes are filled from left to right.



Full binary tree are complete

Balanced Binary Tree: AVL Tree

In a balanced binary tree, the height of the left and right subtrees of every node differs by at most 1. Examples include AVL trees and Red-Black trees.



Complete Trees are balanced

Binary Search Tree (BST):

A binary tree where for every node:

- The left subtree contains values less than the node's value.
- The right subtree contains values greater than the node's value.

Operations on a Binary Tree:

Insertion:

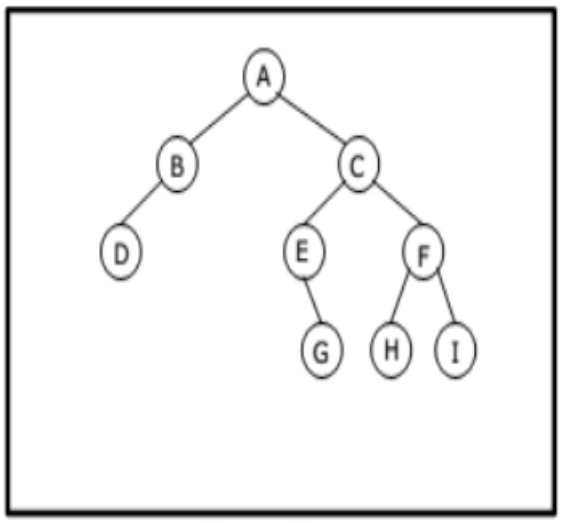
Insertion in a binary tree can happen at any available position, but in the case of specific trees like Binary Search Trees, it follows a sorted order (left child is smaller, right child is larger).

Traversal:

- In-order Traversal (Left, Root, Right):
  Traverse the left subtree, visit the root node, then traverse the right subtree. In a BST, in-order traversal produces elements in sorted order.

Example: Left -> Root -> Right

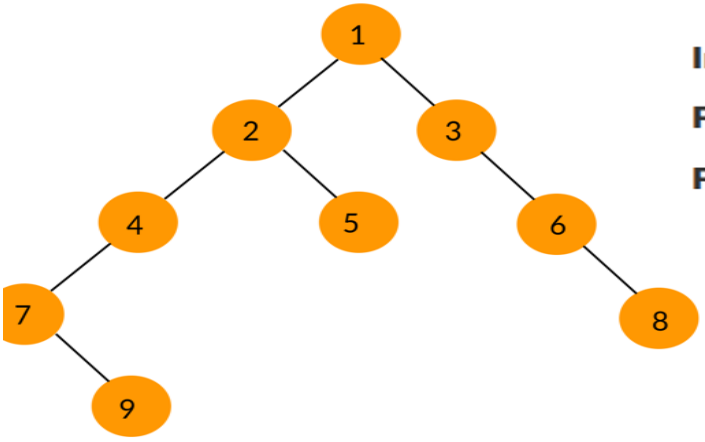
- Pre-order Traversal (Root, Left, Right):
  Visit the root node, then traverse the left subtree, followed by the right subtree.
  Example: Root -> Left -> Right
- Post-order Traversal (Left, Right, Root):
  Traverse the left subtree, traverse the right subtree, then visit the root node.
  Example: Left -> Right -> Root
- Level-order Traversal:
  Visit nodes level by level, from left to right.



Binary Tree

- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing



Inorder Traversal: 7 9 4 2 5 1 3 6 8

Preorder Traversal: 1 2 4 7 9 5 3 6 8

Postorder Traversal: 9 7 4 5 2 8 6 3 1

Deletion:

Involves finding the node to be deleted, removing it, and adjusting the tree to preserve its structure. In a BST, special care is needed to ensure the properties of the BST are maintained.

Search:

Searching in a general binary tree is done through traversal, but in a Binary Search Tree, it is more efficient as it can skip subtrees that don't contain the target value.

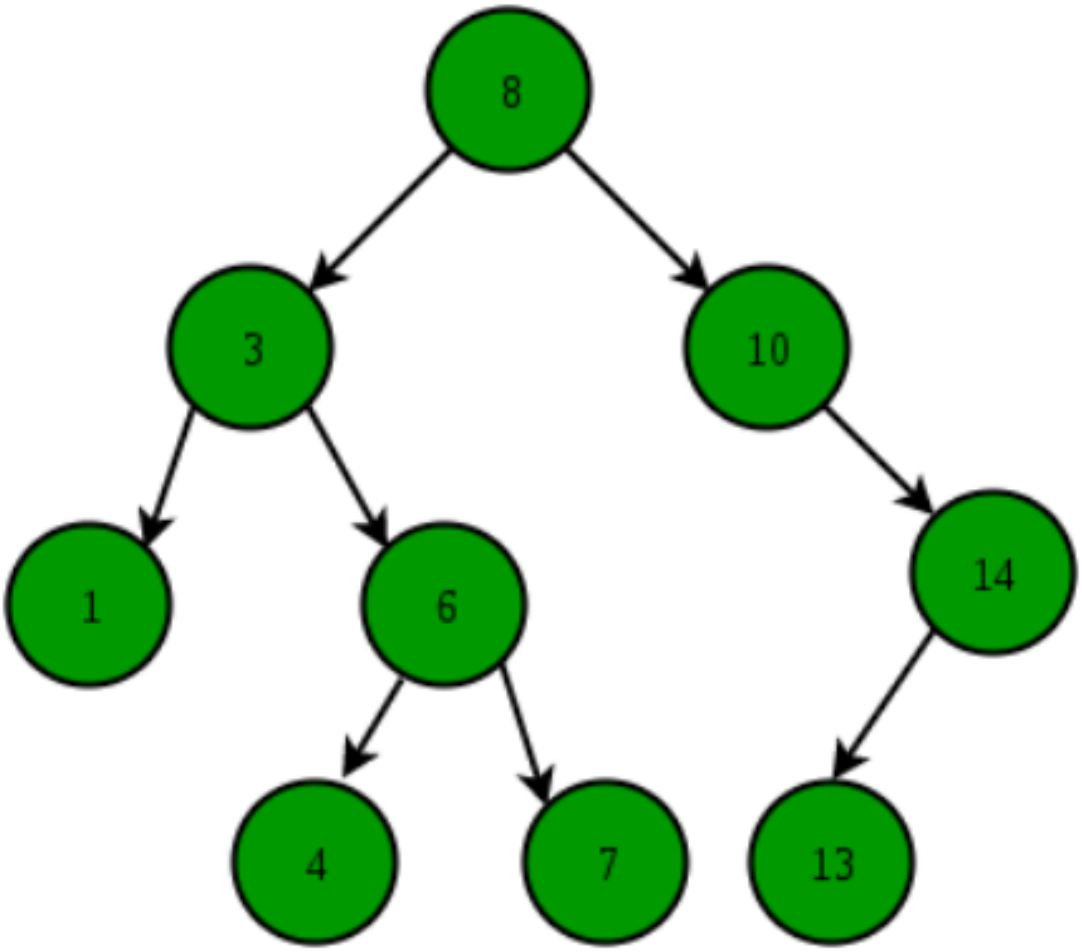
BINARY SEARCH TREE (BST)

A Binary Search Tree (BST) is a special type of binary tree where each node has the following properties:

- Left Subtree Property:
  The left subtree of a node contains only nodes with values less than the node's value.
- Right Subtree Property:
  The right subtree of a node contains only nodes with values greater than the node's value.
- No Duplicate Nodes:
  A binary search tree does not contain duplicate nodes; each value in the tree must be unique.

These properties allow BSTs to efficiently perform operations like search, insertion, and deletion with a time complexity of O(log n) in the average case, making them useful for tasks where sorted data is needed.

Sample BST



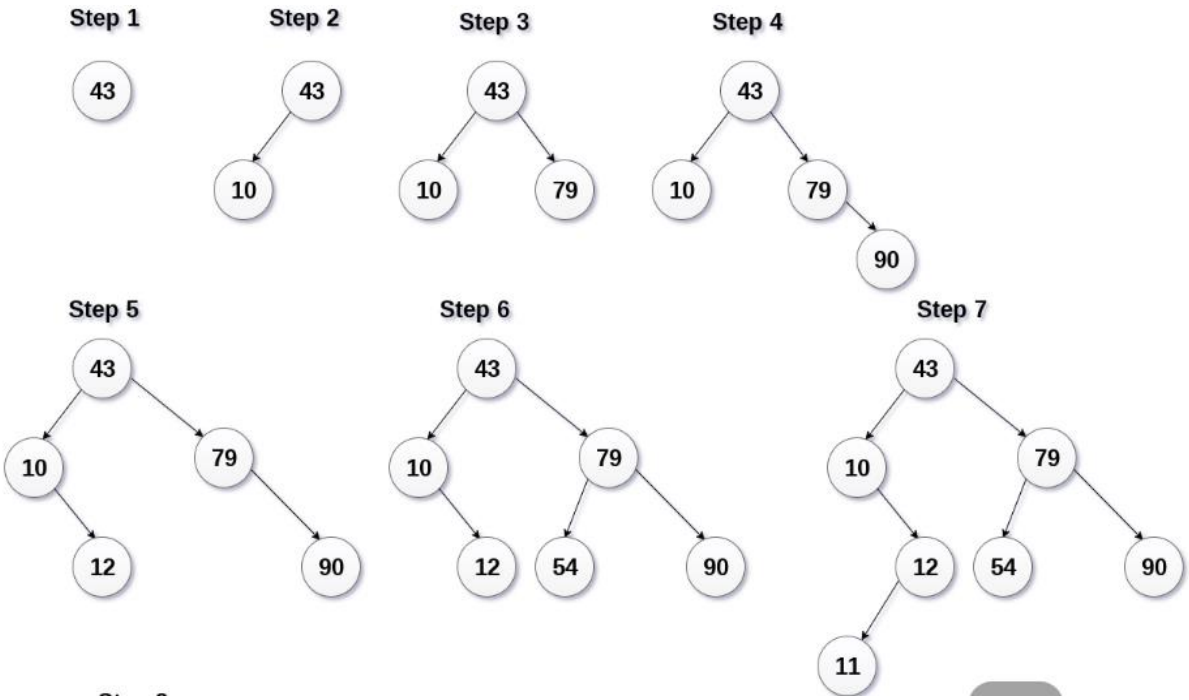
Create binary search tree using the following data elements.

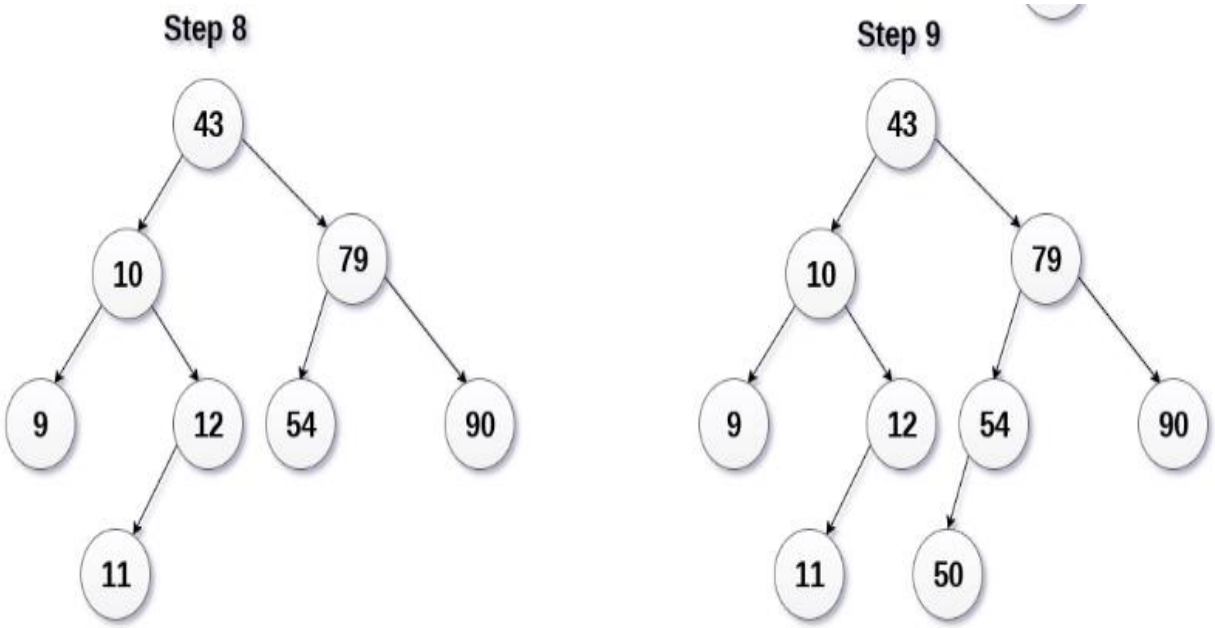
43, 10, 79, 90, 12, 54, 11, 9, 50

a) Insert 43 into the tree as the root of the tree.

b) Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.

c) Otherwise, insert it as the root of the right of the right sub-tree.


Step 1
43
Step 2
43
10
Step 3
43
10
79
Step 4
43
10
79
90
Step 5
43
10
79
12
90
Step 6
43
10
79
12
54
90
Step 7
43
10
79
12
54
90
11


Step 8
43
10
79
9
12
54
90
11
Step 9
43
10
79
9
12
54
90
11
50

Applications of Binary Tree:

- Binary Search Tree (BST):
  Efficient for searching, insertion, and deletion operations in sorted data. Used in databases and file systems.
- Heaps:
  A type of binary tree used for priority queues. A max-heap ensures that the parent node is greater than its children, while a min-heap ensures that the parent node is smaller than its children.
- Binary Expression Trees:
  Used to represent expressions where the internal nodes represent operators and leaf nodes represent operands. These are used in compilers for parsing expressions.
- Decision Trees:
  Used in machine learning for decision-making processes. Each internal node represents a decision, and the leaf nodes represent outcomes.
- Huffman Tree:
  A binary tree used in data compression algorithms. It represents prefix-free codes and ensures efficient encoding and decoding of data.

Advantages of Binary Tree:

- Efficient Searching:
  In a well-balanced Binary Search Tree, searching takes O(log n) time.
- Hierarchical Structure:
  Binary trees allow for a natural hierarchical structure, making them suitable for representing organizational data or file systems.
- Memory Efficient:
  Binary trees use pointers or references, which allow dynamic memory allocation and efficient memory use compared to arrays.
- Versatile:
  Binary trees are flexible and can be modified to create more specialized trees like AVL trees, Red-Black trees, heaps, etc.

Disadvantages of Binary Tree:

- Degenerate Trees:
  In the worst case (like a skewed tree), a binary tree can degenerate into a linked list, leading to inefficient operations with time complexity O(n).
- Balancing:
  Maintaining a balanced binary tree (like AVL trees) requires additional overhead for balancing after each insertion or deletion.
- Complex Implementation:

While binary trees offer efficient searching and sorting, their implementation is more complex than linear data structures like arrays or linked lists.

Write a C++ program to implement a BST. The program should prompt users to enter values and them organize and display them in form of a BST. The program should also enable users to traverse the BST in any order.

```
# include <iostream>
# include <cstdlib>
using namespace std;
//Node Declaration
struct node
{
int info;
struct node *left;
struct node *right;
}*root;
// Class Declaration
class BST
{
public:
void find(int, node **, node **);
void insert(node *tree, node *newnode);
void del(int);
void case_a(node *,node *);
void case_b(node *,node *);
void case_c(node *,node *);
void preorder(node *);
void inorder(node *);
void postorder(node *);
void display(node *, int);
```

```
BST()
        {
root = NULL;
        }
};
//Main Contains Menu
int main()
{
int choice, num;
    BST bst;
node *temp;
while (1)
    {
cout<<"-----------------"<<endl;
cout<<"Operations on BST"<<endl;
cout<<"-----------------"<<endl;
cout<<"1.Insert Element "<<endl;
cout<<"2.Delete Element "<<endl;
cout<<"3.Inorder Traversal"<<endl;
cout<<"4.Preorder Traversal"<<endl;
cout<<"5.Postorder Traversal"<<endl;
cout<<"6.Display"<<endl;
cout<<"7.Quit"<<endl;
cout<<"Enter your choice : ";

cin>>choice;
switch(choice)
        {
case 1:
temp = new node;
cout<<"Enter the number to be inserted : ";
```

```
        cin>>temp->info;
bst.insert(root, temp); break;
case 2:
if (root == NULL)
        {
cout<<"Tree is empty, nothing to delete"<<endl;
continue;
        }
cout<<"Enter the number to be deleted : ";
cin>>num;
bst.del(num);
break;
case 3:
cout<<"Inorder Traversal of BST:"<<endl;
bst.inorder(root);
cout<<endl;
break;
        case 4:
cout<<"Preorder Traversal of BST:"<<endl;
bst.preorder(root);
cout<<endl;
break;
case 5:
cout<<"Postorder Traversal of BST:"<<endl;
bst.postorder(root);
cout<<endl;
break;
case 6:
cout<<"Display BST:"<<endl;
bst.display(root,1);
cout<<endl;
```

```
break;
case 7:
exit(1);
default:
cout<<"Wrong choice"<<endl;
        }
    }
}
//Find Element in the Tree
void BST::find(int item, node **par, node **loc)
{
node *ptr, *ptrsave;
if (root == NULL)
    {
        *loc = NULL;
        *par = NULL;
return;
    }
if (item == root->info)
    {
        *loc = root;
        *par = NULL;
return;
    }
if (item < root->info)
ptr = root->left;
else
ptr = root->right;
ptrsave = root;
while (ptr != NULL)
    {
```

```
if (item == ptr->info)
      {
         *loc = ptr;
         *par = ptrsave;
return;
      }
ptrsave = ptr;
if (item <ptr->info)
ptr = ptr->left;
          else
          ptr = ptr->right;
   }
   *loc = NULL;
   *par = ptrsave;
}
// Inserting Element into the Tree
 void BST::insert(node *tree, node *newnode)
{
if (root == NULL)
   {
root = new node;
root->info = newnode->info;
root->left = NULL;
root->right = NULL;
cout<<"Root Node is Added"<<endl;
return;
   }
if (tree->info == newnode->info)
   {
cout<<"Element already in the tree"<<endl;
return;
```

```
   }
if (tree->info >newnode->info)
   {
if (tree->left != NULL)
      {
insert(tree->left, newnode);
          }
          else
          {
tree->left = newnode;
          (tree->left)->left = NULL;
          (tree->left)->right = NULL;
cout<<"Node Added To Left"<<endl;
return;
      }
   }
else
   {
if (tree->right != NULL)
      {
insert(tree->right, newnode);
      }
else
      {
tree->right = newnode;
          (tree->right)->left = NULL;
          (tree->right)->right = NULL;
cout<<"Node Added To Right"<<endl;
return;
      }
   }
```

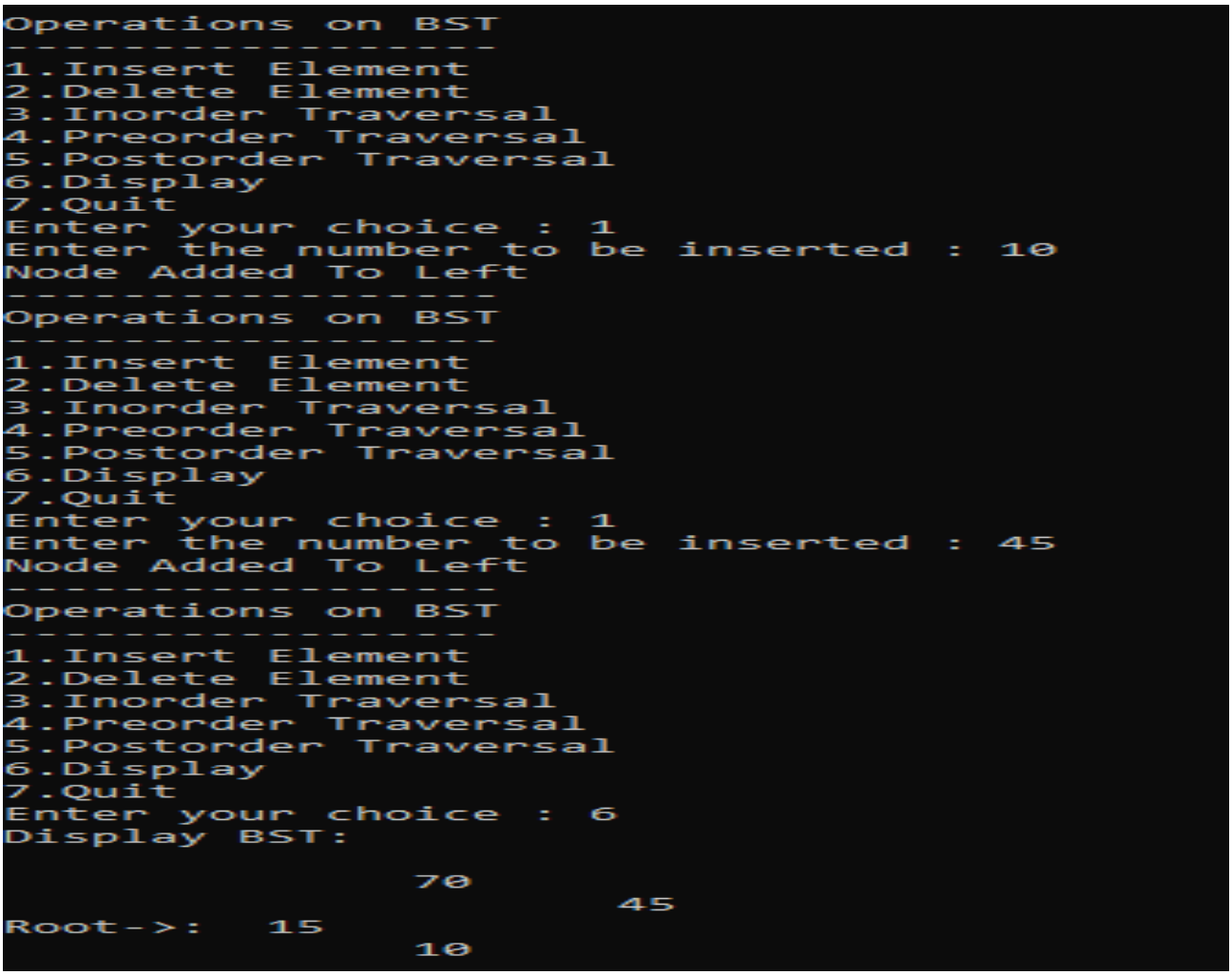
```
}
void BST::del(int item)
{
node *parent, *location;
if (root == NULL)
   {
cout<<"Tree empty"<<endl;
return;
   }
find(item, &parent, &location);
if (location == NULL)
   {
cout<<"Item not present in tree"<<endl;
return;
   }
if (location->left == NULL && location->right == NULL)
case_a(parent, location);
if (location->left != NULL && location->right == NULL)
case_b(parent, location);
if (location->left == NULL && location->right != NULL)
case_b(parent, location);
if (location->left != NULL && location->right != NULL)
case_c(parent, location);
free(location);
}
void BST::case_a(node *par, node *loc )
{
if (par == NULL)
   {
root = NULL;
   }
```

```
else
   {
if (loc == par->left)
par->left = NULL;
else
par->right = NULL;
   }
}
void BST::case_b(node *par, node *loc)
{
node *child;
if (loc->left != NULL)
child = loc->left;
else
child = loc->right;
if (par == NULL)
   {
root = child;
   }
else
   {
if (loc == par->left)
par->left = child;
else
par->right = child;
   }
}
void BST::case_c(node *par, node *loc)
{
node *ptr, *ptrsave, *suc, *parsuc;
ptrsave = loc;
```

```
ptr = loc->right;
while (ptr->left != NULL)
   {
ptrsave = ptr;
ptr = ptr->left;
   }
suc = ptr;
parsuc = ptrsave;
if (suc->left == NULL &&suc->right == NULL)
case_a(parsuc, suc);
else
case_b(parsuc, suc);
if (par == NULL)
   {
root = suc;
   }
else
   {
if (loc == par->left)
par->left = suc;
else
par->right = suc;
   }
suc->left = loc->left;
suc->right = loc->right;
}
 // Pre Order Traversal
 void BST::preorder(node *ptr)
{
if (root == NULL)
   {
```

```
cout<<"Tree is empty"<<endl;
return;
   }
if (ptr != NULL)
   {
cout<<ptr->info<<"  ";
preorder(ptr->left);
preorder(ptr->right);
   }
}
 //In Order Traversal
 void BST::inorder(node *ptr)
{
if (root == NULL)
   {
cout<<"Tree is empty"<<endl;
return;
   }
if (ptr != NULL)
   {
inorder(ptr->left);
cout<<ptr->info<<"  ";
inorder(ptr->right);
   }
}
 // Postorder Traversal
 void BST::postorder(node *ptr)
{
if (root == NULL)
   {
cout<<"Tree is empty"<<endl;
```

```
return;
   }
if (ptr != NULL)
   {
postorder(ptr->left);
postorder(ptr->right);
cout<<ptr->info<<"  ";
   }
}
 // Display Tree Structure
 void BST::display(node *ptr, int level)
{
int i;
if (ptr != NULL)
   {
display(ptr->right, level+1);
cout<<endl;
if (ptr == root)
cout<<"Root->:  ";
else
       {
for (i = 0;i <level;i++)
cout<<"       ";
           }
cout<<ptr->info;
display(ptr->left, level+1);
   }
}
```



HEAP DATA STRUCTURE

A heap is a special type of binary tree that satisfies two important properties:

- Complete Binary Tree:
  A heap is always a complete binary tree, meaning all levels of the tree are fully filled except possibly for the last level, which is filled from left to right.
- Heap Property:
  There are two types of heaps, each with its own heap property:
  - Max-Heap: The value of each parent node is greater than or equal to the values of its children. This means the largest value is always at the root.
  - Min-Heap: The value of each parent node is less than or equal to the values of its children. This means the smallest value is always at the root.

Types of Heaps

Max-Heap:

In a max-heap, for any given node i, the value of i is greater than or equal to its children.

Example of a max-heap:

```
      50
    /    \
  30      20
 / \    / \
15  10 5   7
```

Min-Heap:

In a min-heap, for any given node i, the value of i is less than or equal to its children.

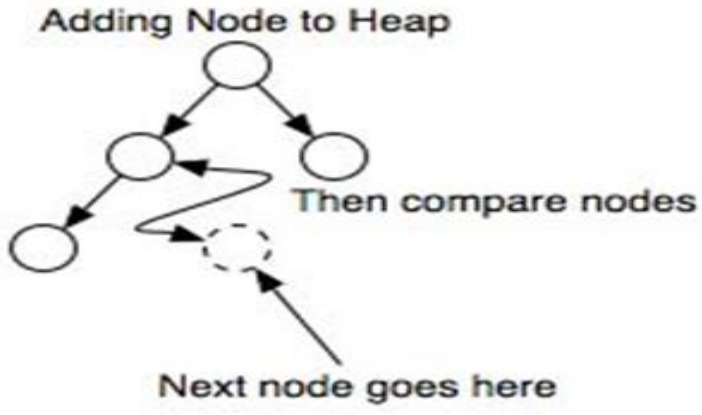
Example of a min-heap:

```
      10
    /    \
  15      30
 / \    / \
50  20 40  60
```

Heap Operations

Insertion:

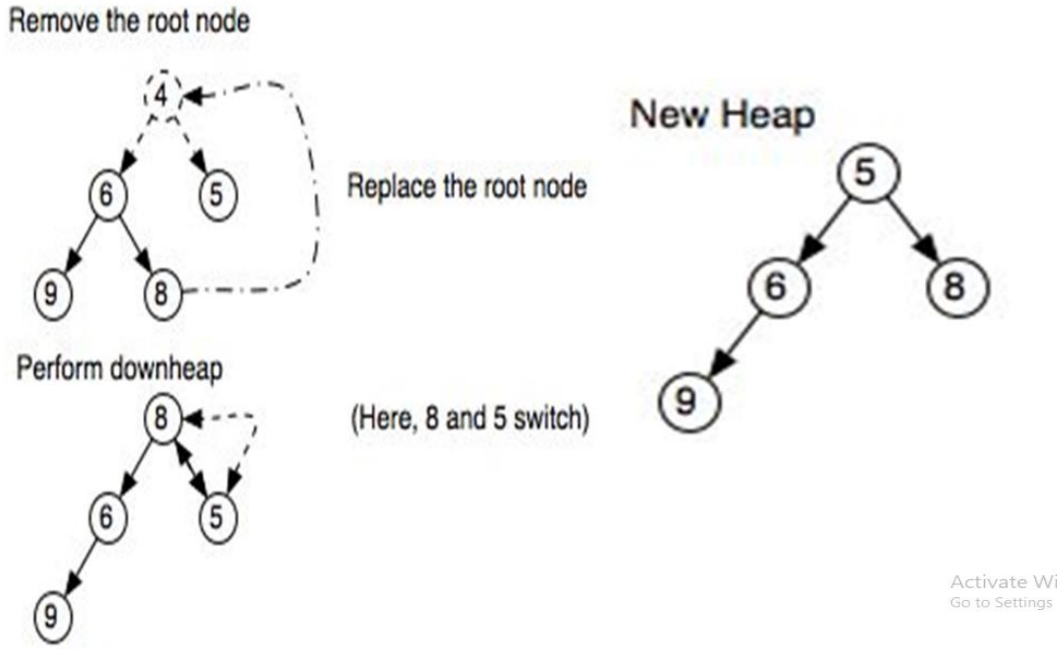
- Insert a new element at the next available position (to maintain completeness)



- Then "heapify" the tree by comparing the newly added element with its parent, swapping if necessary (upheap).
- Time Complexity: O(log n).

Deletion (Extract Max/Min):

- The root element (either max or min) is removed. The last element is moved to the root and then "heapify" (downheap) to maintain the heap property.



- Time Complexity: O(log n).

Heapify:

- A process that ensures the heap property is maintained after insertion or deletion. It can be performed either from top-down (for deletion) or bottom-up (for insertion).

Building a Heap:

To convert an array of elements into a heap, the heapify process is applied starting from the lowest level.

Time Complexity: O(n).

```
Step 1 - Create a new node at the end of heap.
Step 2 - Assign new value to the node.
Step 3 - Compare the value of this child node with its parent.
Step 4 - If value of parent is less than child, then swap them.
Step 5 - Repeat step 3 & 4 until Heap property holds.
```

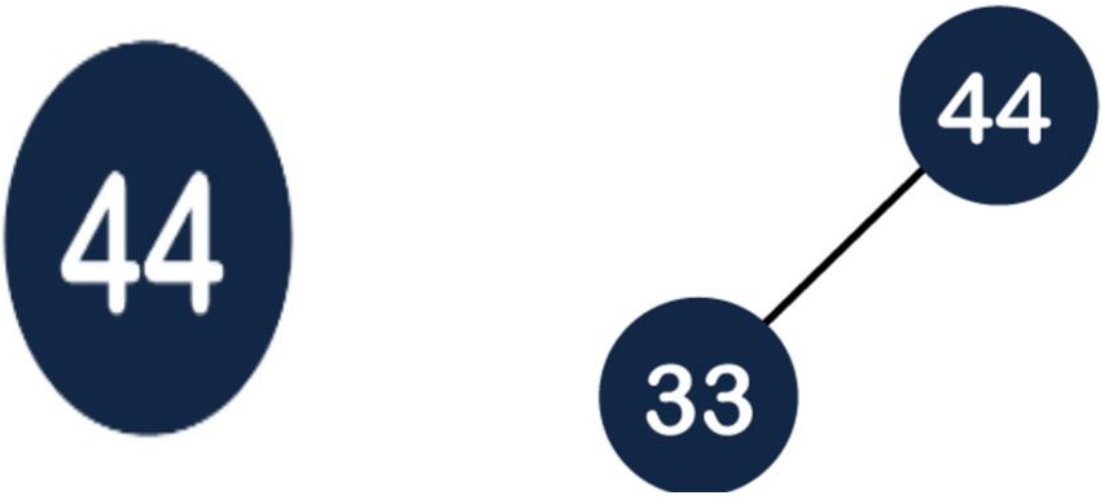
Insertion in the Heap tree

44, 33, 77, 11, 55, 88, 66

Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

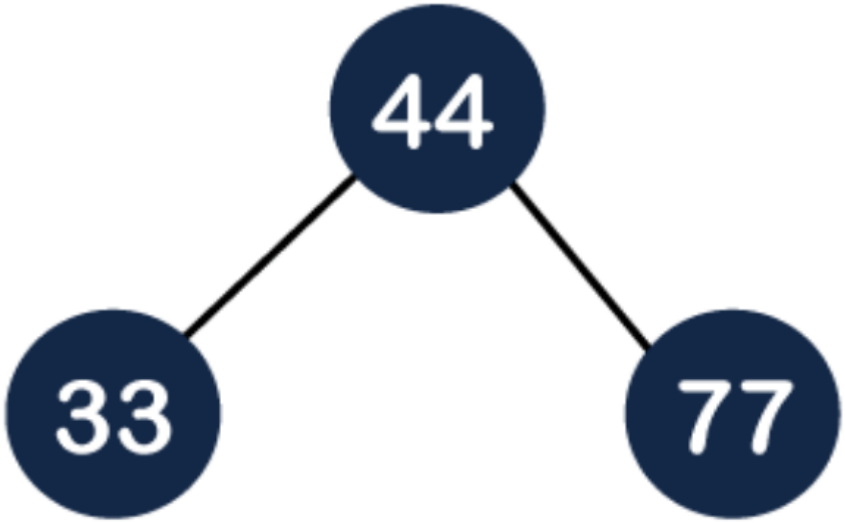
- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
- Secondly, the value of the parent node should be greater than the either of its child.

Step 1: First we add the 44 element in the tree as shown below:

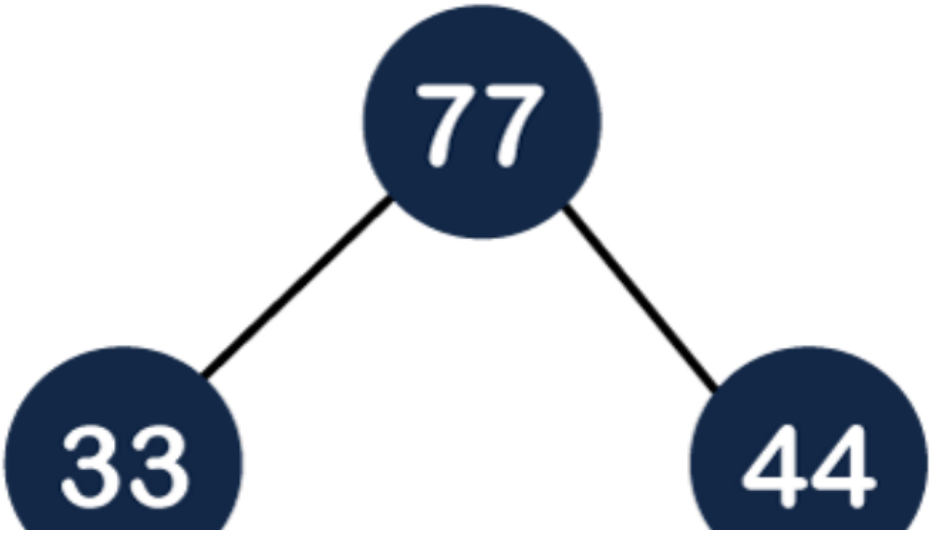


Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:

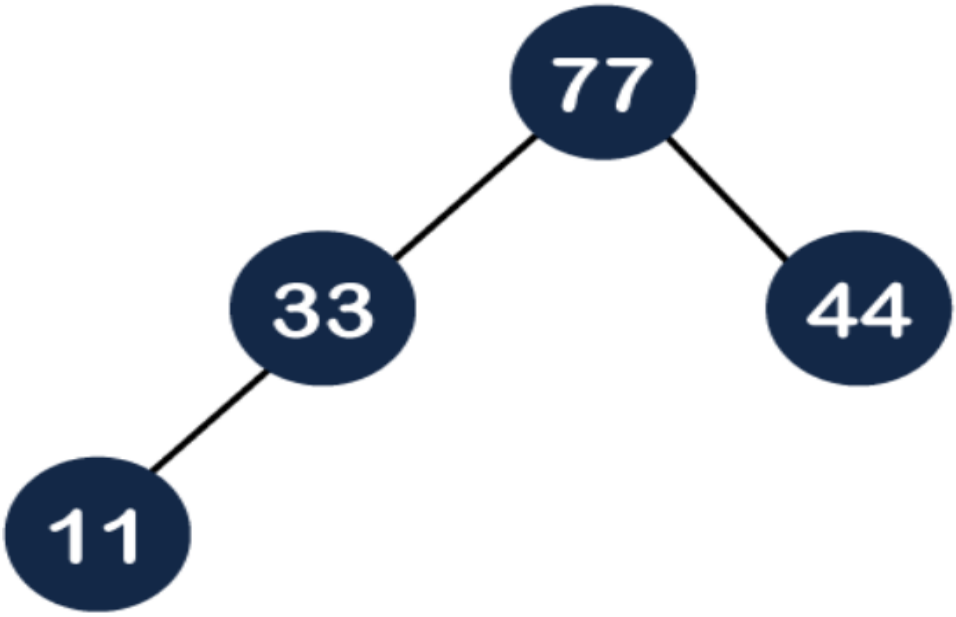
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



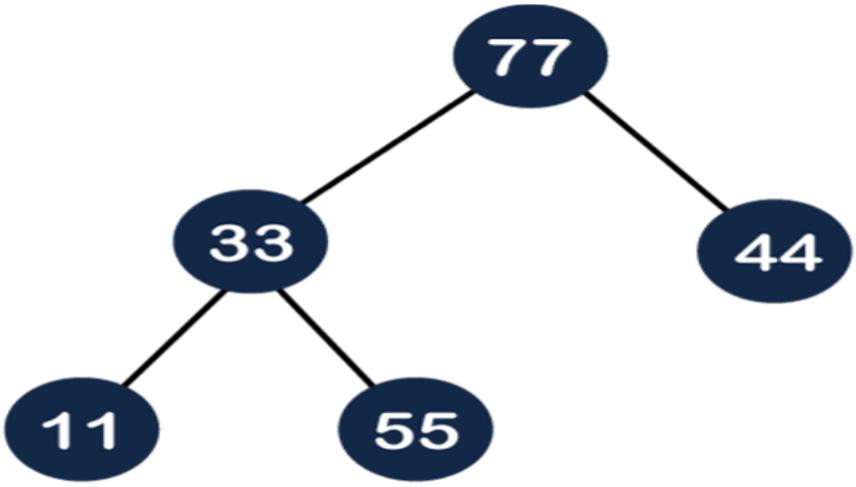
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



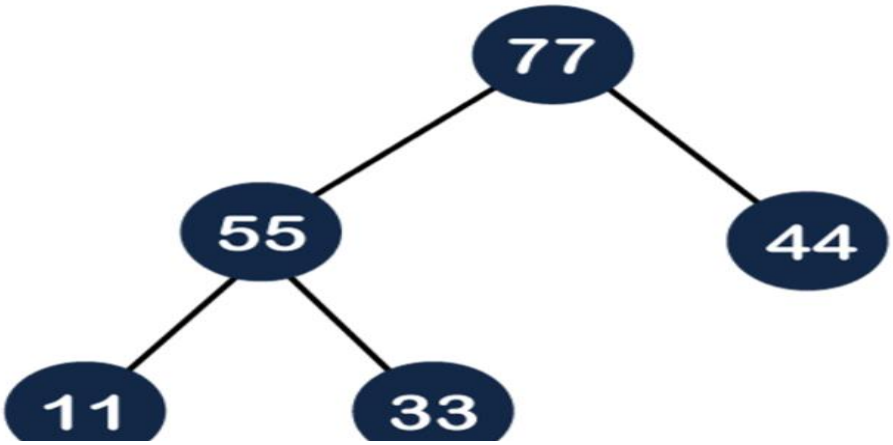
4: The next element is 11. The node 11 is added to the left of 33 as shown below:



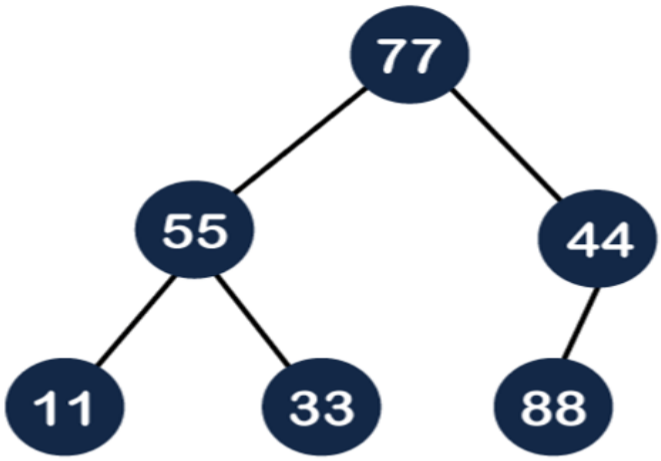
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:



The above figure does not satisfy the property of the max heap because 33 < 55, so we will swap these two values as shown below:



Step 6: The next element is 88. The left sub tree is completed so we will add 88 to the left of 44 as shown below:



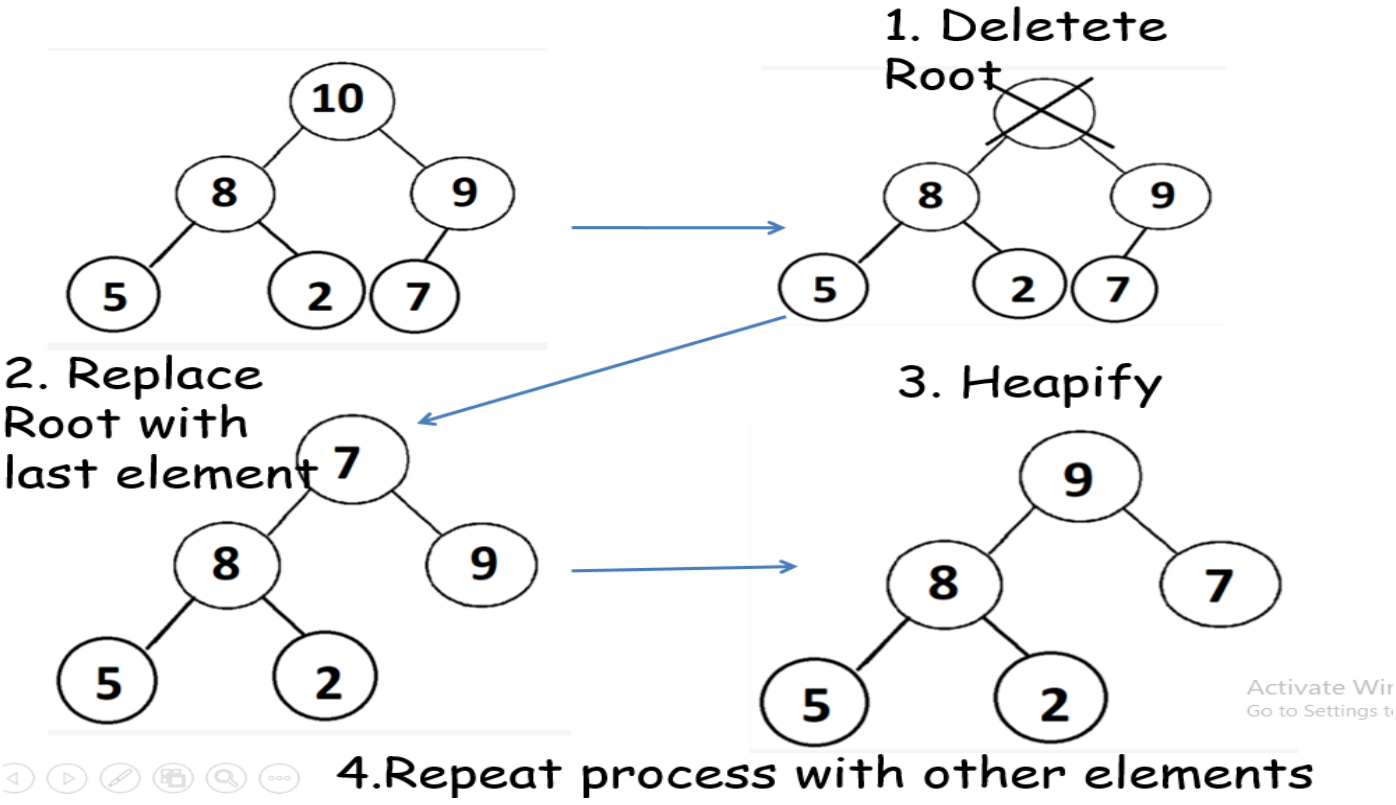
Since 44<88, swap these two values as shown below:
Again, it is violating the max heap property because 88>77 so we will swap these two values below:

Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below

Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

```
Step 1 - Remove root node.
Step 2 - Move the last element of last level to root.
Step 3 - Compare the value of this child node with its parent.
Step 4 - If value of parent is less than child, then swap them.
Step 5 - Repeat step 3 & 4 until Heap property holds.
```



Applications of Heap

- Priority Queue:
  Heaps are commonly used to implement priority queues, where the highest (max-heap) or lowest (min-heap) priority element is dequeued first.
- Heap Sort:
  Heaps can be used to implement an efficient sorting algorithm known as Heap Sort, which has a time complexity of O(n log n).
- Graph Algorithms:
  Heaps are used in several graph algorithms, such as Dijkstra's shortest path algorithm and Prim's Minimum Spanning Tree (MST) algorithm, where elements need to be retrieved based on priority.
- Scheduling Algorithms:
  Heaps are used in operating system schedulers to manage tasks based on their priority.

Advantages of Heap

- Efficient for implementing priority queues.
- Insertion and deletion operations are logarithmic in time complexity, making heaps faster for dynamic datasets than other data structures like arrays.

Disadvantages of Heap

- The heap data structure does not support efficient searching (searching is O(n)).
- It can become inefficient for some types of data where ordered sequences are required.

GRAPH DATA STRUCTURE

A graph is a non-linear data structure made up of a set of vertices (or nodes) and edges. Graphs are used to model relationships between objects and are widely used in many real-world applications, such as social networks, web page ranking, and routing algorithms.

Components of a Graph

- **Vertices (Nodes):**
  Vertices represent the objects or entities in the graph.
  Example: In a social network, each person can be represented as a vertex.
- **Edges:**
  Edges are the connections between vertices.
  Example: In a social network, an edge between two vertices represents a friendship or connection between two people.

- **Degree:**
  The degree of a vertex refers to the number of edges connected to it.
  In-degree: Number of edges directed toward the vertex (in a directed graph).
  Out-degree: Number of edges directed outward from the vertex (in a directed graph).

Consider the graph G, below:

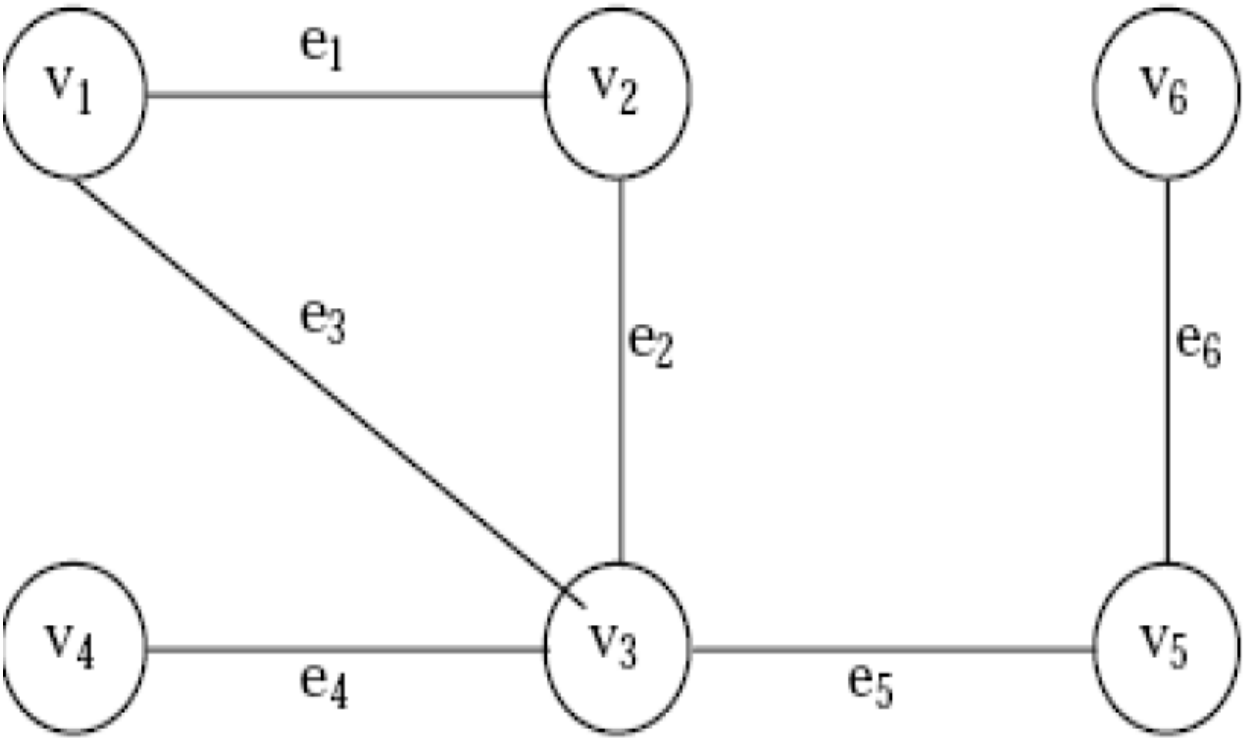
Then the vertex V and edge E can be represented as:

V = {v1, v2, v3, v4, v5, v6}

E = {e1, e2, e3, e4, e5, e6} or

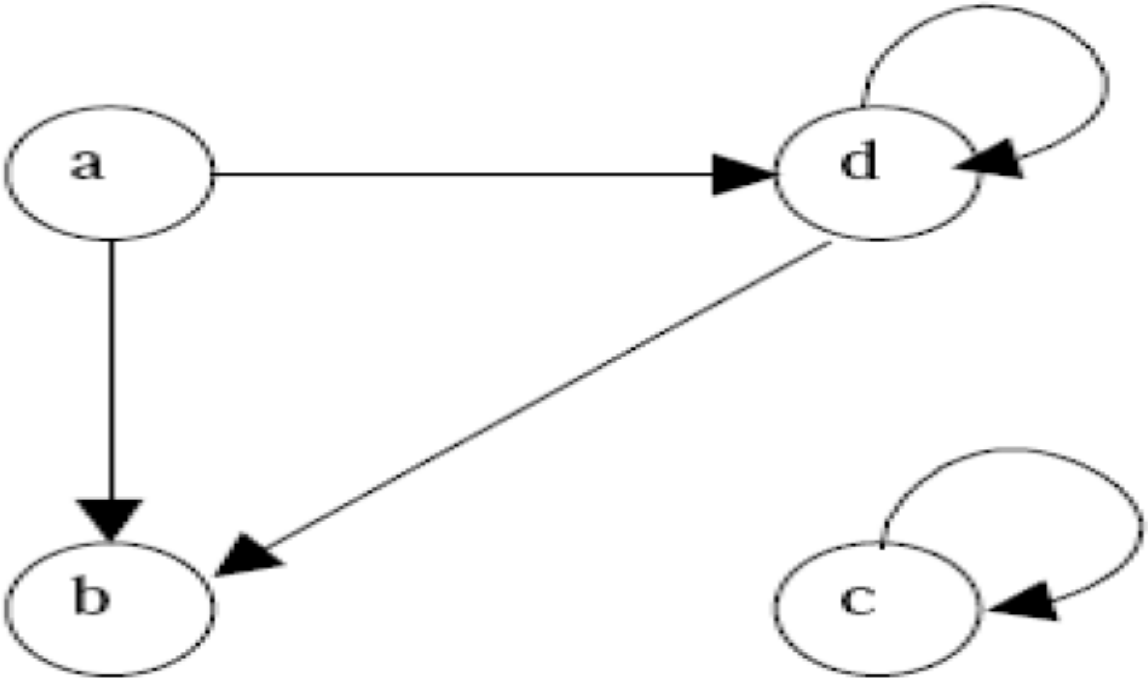
E = {(v1, v2) (v2, v3) (v1, v3) (v3, v4), (v3, v5) (v5, v6)}.

There are six edges and vertex in the graph



Types of Graphs

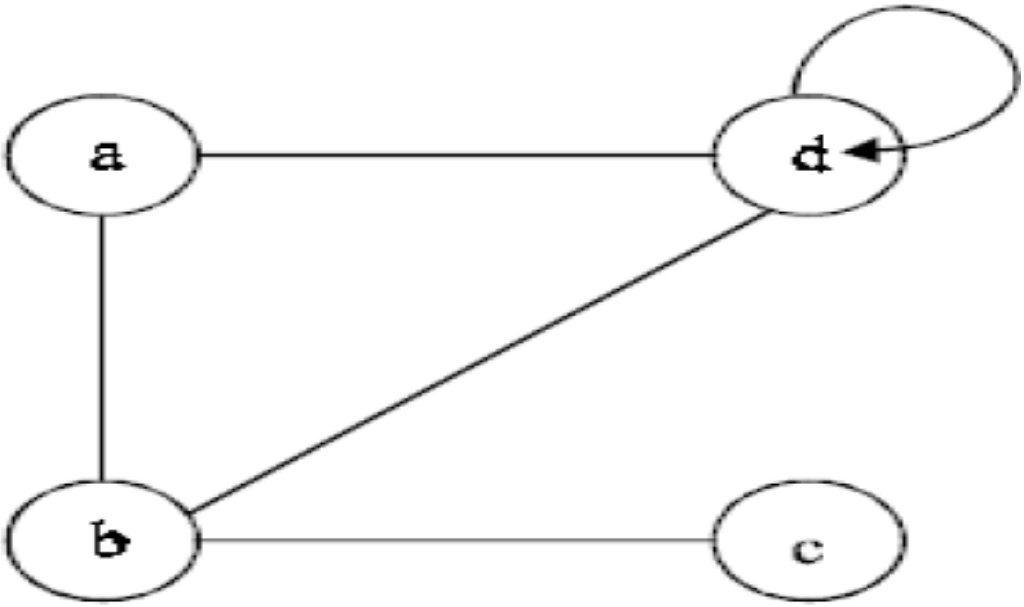
- Directed Graph (Digraph):
  A directed graph is a graph in which the edges have a direction. An edge (u, v) points from vertex u to vertex v.



The vertex a is called the initial vertex and the vertex b is called the terminal vertex of the edge (a, b).
Two vertices are said to be adjacent if they are joined by an edge.
A vertex is said to be an isolated vertex if there is no edge incident with it. eg vertex C is an isolated vertex.

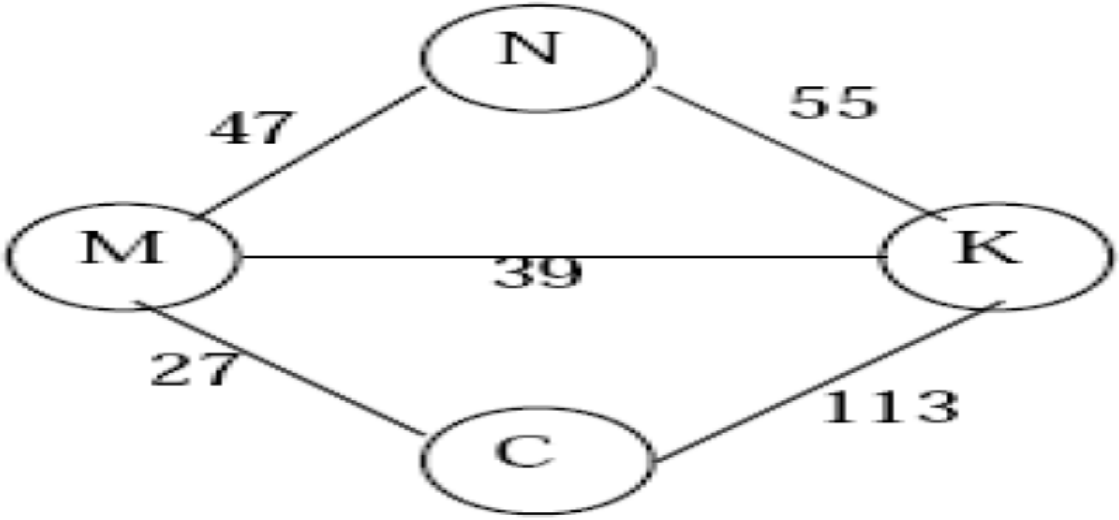
If an edge that is incident from and into the same vertex, say (d, d) of (c, c) , it is called a loop.
Example: Twitter's follow relationship can be modeled as a directed graph, where an edge from u to v indicates that user u follows user v.

- Undirected Graph:
  In an undirected graph, edges have no direction, meaning if there is an edge between u and v, it can be traversed in both directions.



Example: Facebook's friend relationship can be modeled as an undirected graph, where an edge between u and v indicates mutual friendship.

- Weighted Graph:
  A graph where each edge has a numerical weight (or cost) associated with it. The weight of an edge could represent distance, cost, or time.

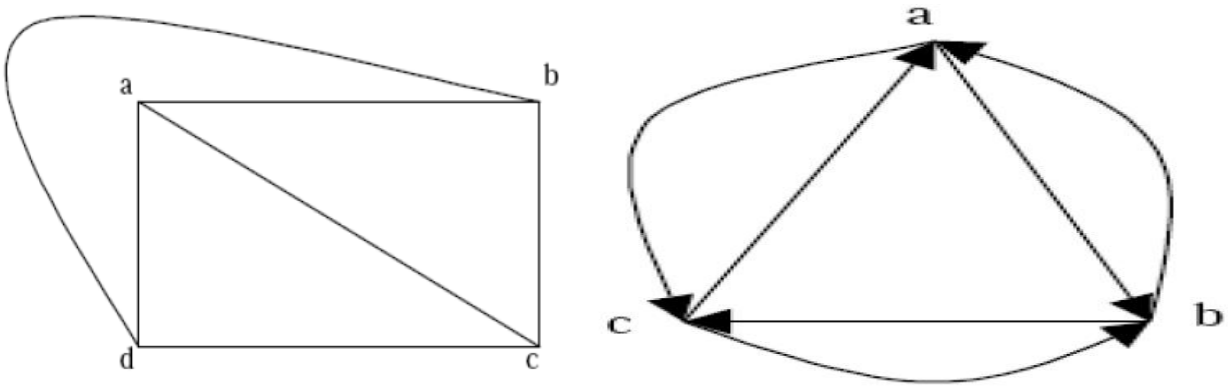


Example: A road network where the vertices represent cities and the edges represent roads with weights denoting the distance between cities.

- Complete (connected) Graph

A graph G is said to complete (or fully connected or strongly connected) if there is a path from every vertex to every other vertex.
In a disconnected graph, at least one pair of vertices cannot be reached from the other.

If a and b are two vertices in the directed graph, then it is a complete graph if there is a path from a to b as well as a path from b to a. A complete graph with n vertices will have n (n – 1)/2 edges.



- **Cyclic and Acyclic Graph:**
  Cyclic Graph: A graph that contains at least one cycle, i.e., a path where the starting and ending vertex are the same.
- **Acyclic Graph:**
  A graph that does not contain any cycles. A DAG (Directed Acyclic Graph) is a directed graph with no cycles. It is widely used in scheduling problems.

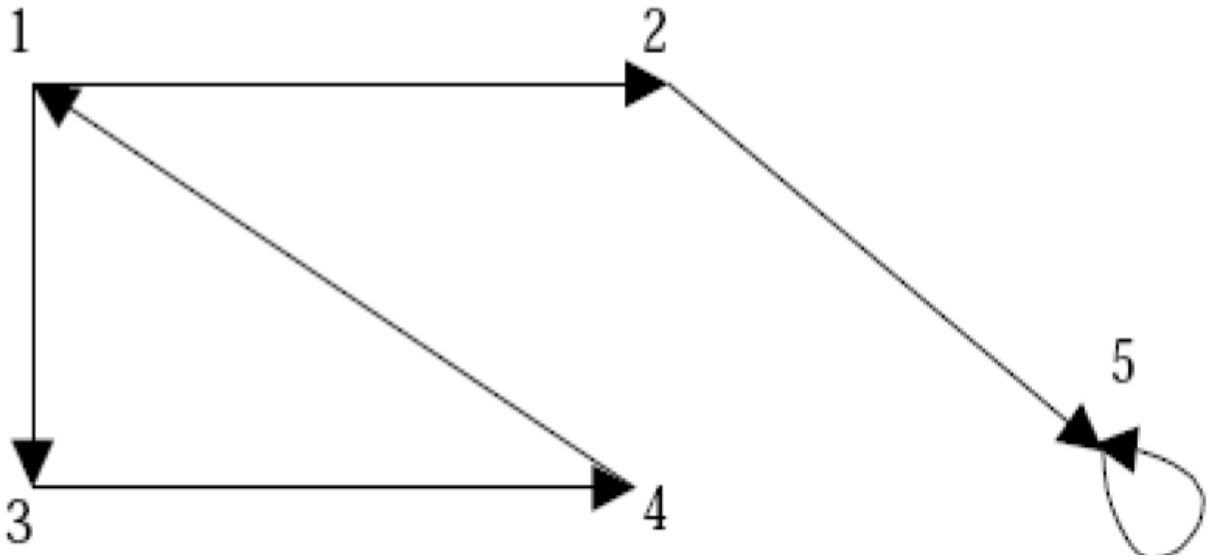
Representation of Graphs

- **Adjacency Matrix:**

A 2D array where the element at row i and column j represents the presence (and possibly the weight) of an edge between vertices i and j.

In an unweighted graph, the matrix element is 1 if there is an edge and 0 if there is no edge.

For a weighted graph, the matrix element contains the weight of the edge.



ADJACENCY MATRIX REPRESENTATION

The adjacency matrix A of a directed graph G = (V, E) can be represented with the following conditions

Aij = 1 {if there is an edge from Vi to Vj or if the edge (i, j) is member of E.}

Aij = 0 {if there is no edge from Vi to Vj}

i \ j	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	1
3	0	0	0	1	0
4	1	0	0	0	0
5	0	0	0	0	1

The adjacency matrix A of an undirected graph G = (V, E) can be represented with the following conditions

Aij = 1 {if there is an edge from Vi to Vj or if the edge (i, j) is member of E}

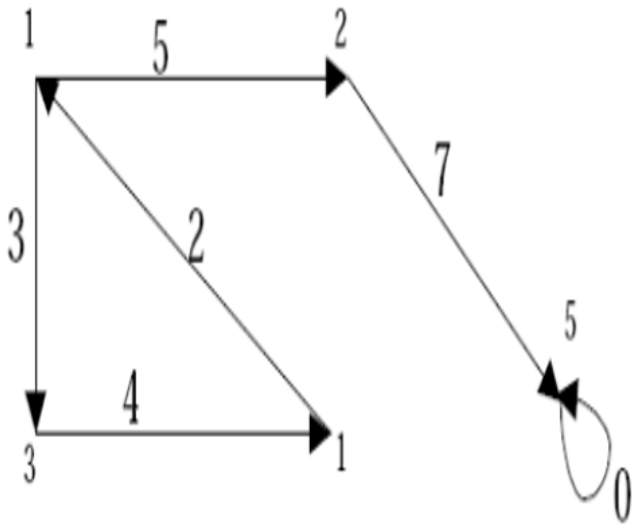
Aij = 0 {if there is no edge from Vi to Vj or the edge i, j, is not a member of E}

To represent a weighted graph using adjacency matrix, weight of the edge (i, j) is simply stored as the entry in i th row and j th column of the adjacency matrix.

The adjacency matrix A for a directed weighted graph G = (V, E, We) can be represented as

Aij = Wij { if there is an edge from Vi to Vj then represent its weight Wij.}

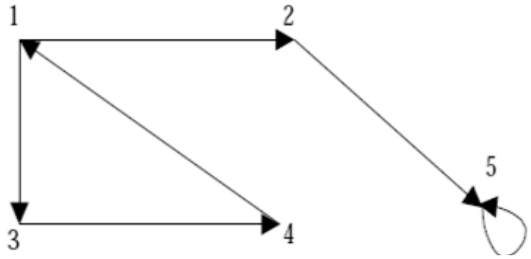
Aij = – 1 { if there is no edge from Vi to Vj}

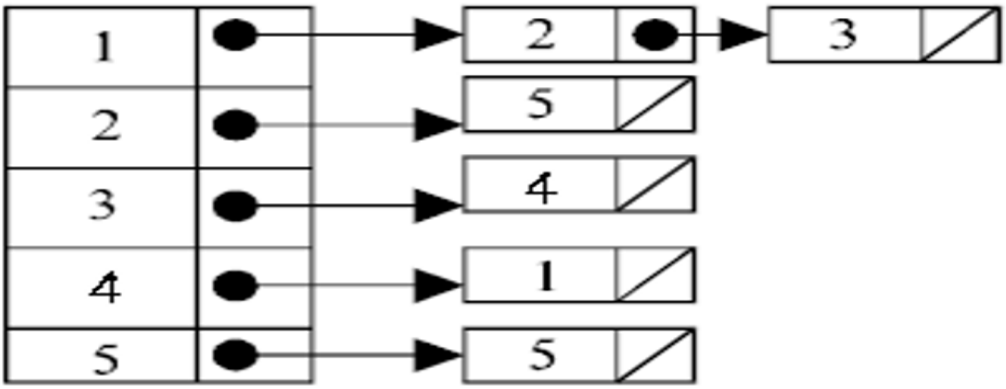


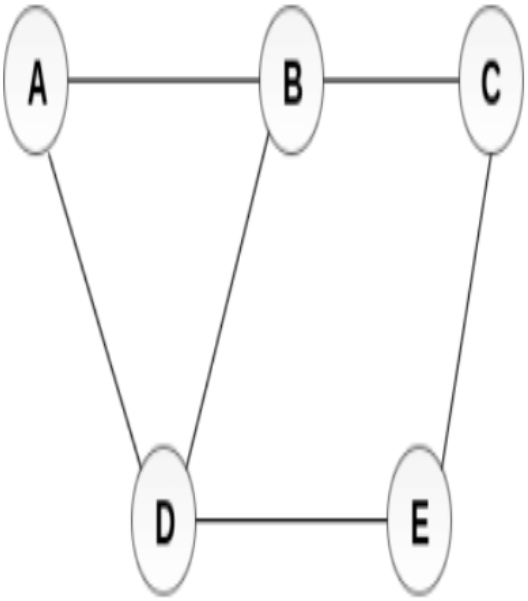
i \ j	1	2	3	4	5
1	- 1	5	3	- 1	- 1
2	- 1	- 1	- 1	- 1	7
3	- 1	- 1	- 1	4	- 1
4	2	- 1	- 1	- 1	- 1
5	- 1	- 1	- 1	- 1	0

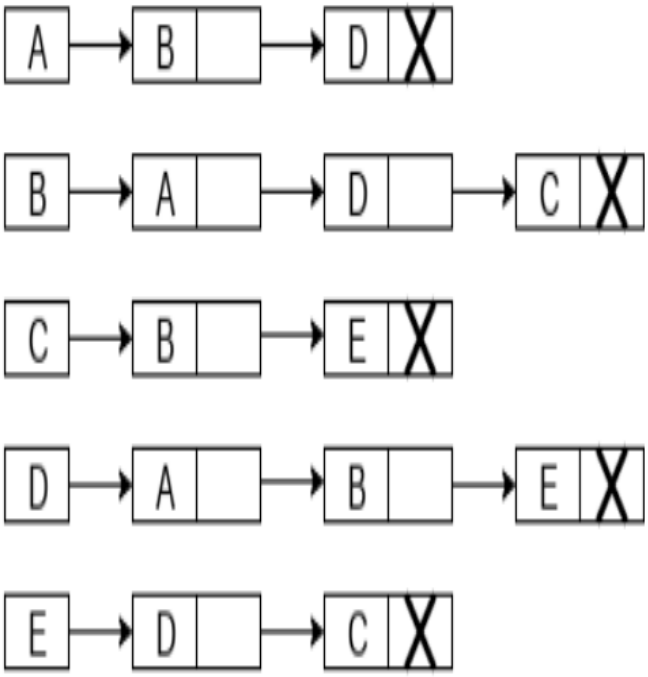
Adjacency (Linked) List:

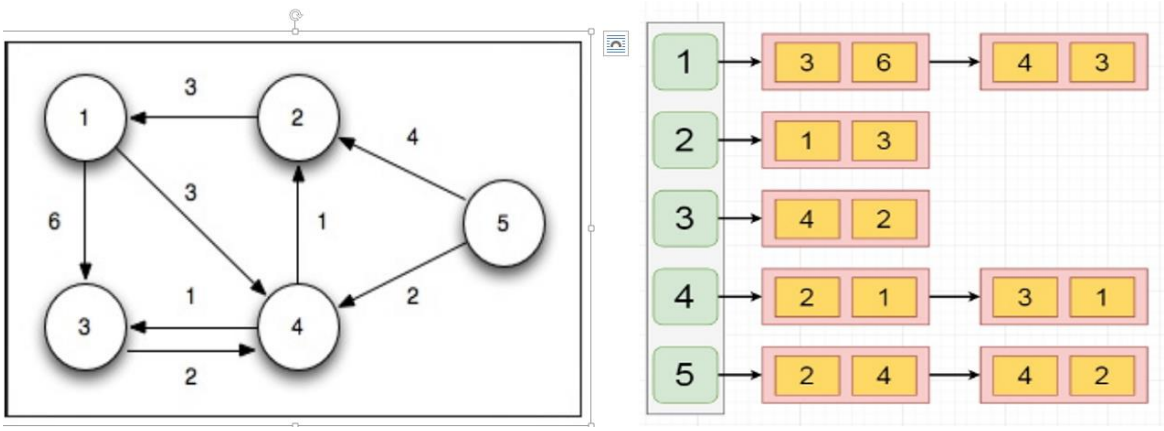
Each vertex has a list of adjacent vertices (or neighbors). This is a space-efficient representation, especially for sparse graphs (where the number of edges is much less than the maximum possible).


1
2
3
4
5


1
2
3
4
5
2
3
5
4
1
5


A
B
C
D
E


A
B
D
B
A
D
C
C
B
E
D
A
B
E
E
D
C



C++ Code: Graph Representation Using Adjacency List

```
#include <iostream>
#include <list>
using namespace std;
class Graph {
    int V;              // Number of vertices
    list<int>* adjList;   // Pointer to adjacency list
public:
    // Constructor
    Graph(int V) {
        this->V = V;
        adjList = new list<int>[V];  // Allocate memory for adjacency list
    }
    // Function to add an edge to the graph
    void addEdge(int v, int w) {
        if(v >= V || w >= V) {
            cout << "Error: Vertex does not exist!" << endl;
            return;
        }
        adjList[v].push_back(w);  // Add w to v's adjacency list
        adjList[w].push_back(v);  // Since the graph is undirected, add v to w's adjacency list
    }
```

```
    // Function to display the adjacency list representation
    void displayGraph() {
        for (int v = 0; v < V; ++v) {
            cout << "Vertex " << v << " is connected to: ";
            for (list<int>::iterator it = adjList[v].begin(); it != adjList[v].end(); ++it) {
                cout << *it << " ";  // Access the element pointed by the iterator
            }
            cout << endl;
        }
    }
    // Destructor to free memory
    ~Graph() {
        delete[] adjList;
    }
};
int main() {
    int V, E;
    // Prompt the user to enter the number of vertices
    cout << "Enter the number of vertices: ";
    cin >> V;
    // Create a graph with V vertices
    Graph g(V);
    // Prompt the user to enter the number of edges
    cout << "Enter the number of edges: ";
    cin >> E;
    // Add edges as per user input
    for (int i = 0; i < E; ++i) {
        int u, v;
        cout << "Enter edge (u v): ";
        cin >> u >> v;
        g.addEdge(u, v);  // Add the edge between vertex u and vertex v
    }
    // Display the graph
    g.displayGraph();
    return 0;
}
```

Explanation:

Graph Class:
The graph is represented using an adjacency list, where adjList is an array of lists. Each index in adjList represents a vertex, and the corresponding list contains the vertices that it is connected to.
The function addEdge(int v, int w) adds an undirected edge between vertices v and w.

User Input:
The program prompts the user to input the number of vertices and edges.
For each edge, the program asks for two integers representing the vertices that the edge connects.

Graph Display:
The program displays the adjacency list representation of the graph, showing which vertices are connected to each other.

Sample Output:
Enter the number of vertices: 5
Enter the number of edges: 6
Enter edge (u v): 0 1
Enter edge (u v): 0 4
Enter edge (u v): 1 2
Enter edge (u v): 1 3
Enter edge (u v): 1 4
Enter edge (u v): 3 4

Vertex 0 is connected to: 1 4
Vertex 1 is connected to: 0 2 3 4
Vertex 2 is connected to: 1
Vertex 3 is connected to: 1 4
Vertex 4 is connected to: 0 1 3

Below is a C++ program that represents a graph using an **adjacency matrix** and traverses the graph using both **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**. The program prompts the user to specify the number of vertices and edges, allows the user to input the edges, and then performs the graph traversal.

```
#include <iostream>

#include <queue>

#include <stack>

using namespace std;
```

```
class Graph {
    int V;              // Number of vertices
    int** adjMatrix;    // Pointer to adjacency matrix
public:
    // Constructor
    Graph(int V) {
        this->V = V;
        // Allocate memory for the adjacency matrix
        adjMatrix = new int*[V];
        for (int i = 0; i < V; i++) {
            adjMatrix[i] = new int[V];
            // Initialize all values to 0 (no edges between vertices)
            for (int j = 0; j < V; j++) {
                adjMatrix[i][j] = 0;
            }
        }
    }
    // Function to add an edge to the graph
    void addEdge(int u, int v) {
        if (u >= V || v >= V) {
            cout << "Error: Vertex does not exist!" << endl;
            return;
        }
        adjMatrix[u][v] = 1;  // Set 1 for an edge between u and v
        adjMatrix[v][u] = 1;  // For undirected graph, set 1 for v to u as well
    }
    // Function to display the adjacency matrix
```

```
    void displayAdjMatrix() {
        cout << "Adjacency Matrix:" << endl;
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                cout << adjMatrix[i][j] << " ";
            }
            cout << endl;
        }
    }
    // DFS Traversal
    void DFS(int startVertex) {
        bool* visited = new bool[V]; // Track visited vertices
        for (int i = 0; i < V; i++) {
            visited[i] = false;
        }
        stack<int> stack;  // Stack to manage DFS traversal
        stack.push(startVertex);
        cout << "DFS Traversal: ";
        while (!stack.empty()) {
            int vertex = stack.top();
            stack.pop();
            if (!visited[vertex]) {
                cout << vertex << " ";
                visited[vertex] = true;
            }
            // Traverse adjacent vertices
            for (int i = V - 1; i >= 0; i--) {
```

```
            if (adjMatrix[vertex][i] == 1 && !visited[i]) {
                stack.push(i);
            }
        }
    }
    cout << endl;
    delete[] visited;  // Free allocated memory
}
// BFS Traversal
void BFS(int startVertex) {
    bool* visited = new bool[V]; // Track visited vertices
    for (int i = 0; i < V; i++) {
        visited[i] = false;
    }
    queue<int> q;  // Queue to manage BFS traversal
    visited[startVertex] = true;
    q.push(startVertex);
    cout << "BFS Traversal: ";
    while (!q.empty()) {
        int vertex = q.front();
        q.pop();
        cout << vertex << " ";
        // Traverse adjacent vertices
        for (int i = 0; i < V; i++) {
            if (adjMatrix[vertex][i] == 1 && !visited[i]) {
                q.push(i);
                visited[i] = true;
```

```
            }
        }
    }
    cout << endl;
    delete[] visited;  // Free allocated memory
}
// Destructor to free memory
~Graph() {
    for (int i = 0; i < V; i++) {
        delete[] adjMatrix[i];  // Free each row
    }
    delete[] adjMatrix;  // Free the main pointer
}
};
int main() {
    int V, E;
    // Prompt the user to enter the number of vertices
    cout << "Enter the number of vertices: ";
    cin >> V;
    // Create a graph with V vertices
    Graph g(V);
    // Prompt the user to enter the number of edges
    cout << "Enter the number of edges: ";
    cin >> E;
    // Add edges as per user input
    for (int i = 0; i < E; ++i) {
        int u, v;
```

```
        cout << "Enter edge (u v): ";
        cin >> u >> v;
        g.addEdge(u, v);  // Add the edge between vertex u and vertex v
    }
    // Display the adjacency matrix
    g.displayAdjMatrix();
    // Perform DFS traversal starting from vertex 0
    g.DFS(0);
    // Perform BFS traversal starting from vertex 0
    g.BFS(0);
    return 0;
}
```

Explanation:

1. **Graph Representation (Adjacency Matrix)**:
   - The graph is represented using a 2D array (adjMatrix). If there's an edge between two vertices u and v, adjMatrix[u][v] and adjMatrix[v][u] are set to 1.
2. **DFS (Depth-First Search)**:
   - Implemented using a **stack** to traverse the graph. The algorithm visits a vertex, pushes adjacent unvisited vertices to the stack, and repeats the process until all vertices have been visited.
3. **BFS (Breadth-First Search)**:
   - Implemented using a **queue** to explore the graph level by level. The algorithm visits a vertex, enqueues its adjacent unvisited vertices, and processes the queue until all vertices have been visited.
4. **User Input**:
   - The program prompts the user to input the number of vertices and edges, and then allows the user to specify the edges.
5. **Traversal Output**:
   - The program performs DFS and BFS starting from vertex 0, but this can be changed as per the user's preference.

Sample Output:

```
Enter the number of vertices: 8
Enter the number of edges: 9
Enter edge (u v): 7 0
Enter edge (u v): 3 2
Enter edge (u v): 2 4
Enter edge (u v): 6 1
Enter edge (u v): 1 0
Enter edge (u v): 0 2
Enter edge (u v): 4 0
Enter edge (u v): 4 3
Enter edge (u v): 5 4
Adjacency Matrix:
0 1 1 0 1 0 0 1
1 0 0 0 0 0 1 0
1 0 0 1 1 0 0 0
0 0 1 0 1 0 0 0
1 0 1 1 0 1 0 0
0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
DFS Traversal: 0 1 6 2 3 4 5 7
BFS Traversal: 0 1 2 4 7 6 3 5

Process returned 0 (0x0)   execution time : 120.594 s
Press any key to continue.
```

Graph Traversal Algorithms

Graph traversal refers to the process of visiting all the vertices in a graph.

1. **Depth-First Search (DFS)**:
    - DFS explores as far down a branch as possible before backtracking.
    - It uses a stack (or recursion) to explore the graph.
    - **Applications**: Detecting cycles, finding connected components, solving mazes, and topological sorting (for DAGs).

2. **Breadth-First Search (BFS)**:
    - BFS explores all the neighbors of a vertex before moving to the next level. It uses a queue to manage the vertices to be visited.
    - **Applications**: Shortest path in unweighted graphs, finding the shortest route in a network, level-order traversal in trees.

Applications of Graphs

1. **Social Networks**:
    - Vertices represent users, and edges represent relationships (friendship, following).
    - Example: Facebook, Twitter, LinkedIn.
2. **Web Search Engines**:
    - The web is modeled as a graph where pages are vertices, and hyperlinks between them are directed edges. Search engines use graph traversal algorithms like **PageRank** to rank web pages.
3. **Routing Algorithms**:
    - Road networks or transportation systems are modeled as weighted graphs, where intersections are vertices and roads are edges with weights (distance or time).
    - Algorithms like **Dijkstra's shortest path algorithm** or **A* algorithm** are used to find the shortest or fastest path.
4. **Recommendation Systems**:
    - In e-commerce websites or streaming services, graphs can be used to recommend items based on user interactions.
    - Vertices represent users and items, and edges represent interactions such as viewing, purchasing, or liking.
5. **Scheduling**:
    - DAGs are used in project management (e.g., in the **Critical Path Method**) where tasks are represented as vertices and dependencies between tasks as directed edges.
6. **Computer Networks**:
    - The internet is a giant graph where vertices represent routers or computers, and edges represent physical or logical connections between them.
    - Routing protocols like **OSPF (Open Shortest Path First)** use graph algorithms to compute the shortest path for data packets.

Advantages of Graphs

- **Versatile**: Graphs are highly flexible and can represent a wide variety of relationships.
- **Efficient for Networking**: Graphs are particularly useful in network modeling and routing algorithms.

- **Modeling Real-world Problems**: Many real-world problems naturally fit into graph structures, making it an ideal data structure for such problems.

Disadvantages of Graphs

- **Memory Consumption**: For large, dense graphs, the memory overhead can be significant, especially when using an adjacency matrix.
- **Complexity**: Graph algorithms can be complex to implement, particularly when working with large datasets and ensuring efficient performance.

ALGORITHMS

- Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.
- We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

Algorithm analysis is the process of determining the efficiency and performance of an algorithm in terms of time and space complexity. It helps in evaluating the feasibility of the algorithm for large inputs and comparing it with alternative approaches. The analysis is generally based on two key aspects:

1. **Time Complexity**: The amount of time an algorithm takes to complete as a function of the input size.
2. **Space Complexity**: The amount of memory an algorithm requires as a function of the input size.

1. Time Complexity

Time complexity refers to the amount of computational time an algorithm takes to execute, typically measured by counting the number of fundamental operations (like comparisons or swaps) as a function of the size of the input.

- **Worst-case Time Complexity (O)**: The maximum time an algorithm can take to complete for any input of size n. This is often the primary measure for comparing algorithms because it provides an upper bound on performance.
- **Best-case Time Complexity (Ω)**: The minimum time an algorithm can take to complete for any input of size n.
- **Average-case Time Complexity (Θ)**: The average time an algorithm takes across all possible inputs of size n.

2. Space Complexity

Space complexity refers to the amount of memory or space required by an algorithm as a function of the input size. It includes:

- **Auxiliary Space**: Extra space or temporary space used by the algorithm, not counting the space for the input itself.
- **Total Space**: Includes the space taken by the input data and auxiliary space.

Importance of algorithm analysis

The most straightforward reason for analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application. The analysis of an algorithm can help us understand it better, and can suggest informed improvements. Algorithms tend to become shorter, simpler, and more elegant during the analysis process.

Why analyze algorithms

- To estimate how long a program will run.
- To estimate the largest input that can reasonably be given to the program.
- To compare the efficiency of different algorithms.
- To choose an algorithm for an application.
- To help focus on the parts of code that are executed the largest number of times.

Asymptotic Notations

Asymptotic notations describe the behavior of the time or space complexity as the input size grows. They are essential for algorithm analysis because they abstract away constant factors and smaller terms, focusing on the dominant term for large input sizes.

- **Big O Notation (O)**: Describes the upper bound of the time complexity. It expresses the worst-case scenario.
    - Example: $O(n^2)$ means that the time taken grows quadratically with the size of the input.
- **Omega Notation (Ω)**: Describes the lower bound of the time complexity. It expresses the best-case scenario.
    - Example: $\Omega(n)$ means that the time taken grows linearly with the size of the input in the best case.
- **Theta Notation (Θ)**: Describes the tight bound, meaning the time complexity is bounded from above and below. It gives a more precise analysis of an algorithm's complexity.

- Example: Θ(n log n) means the time complexity is logarithmic in both the best and worst cases.

Steps to Analyze an Algorithm

1. **Identify Basic Operations**: These are the core steps that contribute significantly to the running time (e.g., comparisons, swaps).
2. **Determine Input Size**: Denote the size of the input as n (or another variable depending on the nature of the algorithm).
3. **Estimate the Number of Operations**: Based on the algorithm's structure, calculate how the number of basic operations grows as a function of n.
4. **Express the Complexity**: Use asymptotic notation (O, Ω, Θ) to represent the time or space complexity.

Factors that Affect Algorithm Efficiency

1. **Input Size (n)**: Larger input sizes generally increase the time and space requirements.
2. **Type of Input Data**: In some cases, the structure of the input (sorted, reverse-sorted, random) can significantly affect performance.
3. **Operations in the Algorithm**: The number and type of operations (arithmetic, comparisons, memory access) play a role in determining efficiency.
4. **Hardware**: While the theoretical analysis is platform-independent, real-world performance can be influenced by hardware like processors and memory architecture.

Trade-offs in Algorithm Analysis

- **Time vs. Space**: Some algorithms use more memory to reduce execution time (e.g., dynamic programming). Others optimize for minimal memory use at the expense of speed.
- **Ease of Implementation vs. Efficiency**: Sometimes simpler algorithms (e.g., Bubble Sort) are easier to implement but less efficient, while more complex algorithms (e.g., Quick Sort) may require additional effort but are more performant.

Conclusion

Algorithm analysis is crucial for choosing the most efficient algorithm for a given problem. Understanding both time and space complexity helps in predicting how an algorithm will scale with larger inputs and whether it can be optimized further.

Sorting algorithms

Are used to arrange data in a specific order (either ascending or descending). Different sorting algorithms vary in efficiency based on time complexity, space complexity, and ease of implementation. Here's an overview of some common sorting algorithms:

Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is sorted.

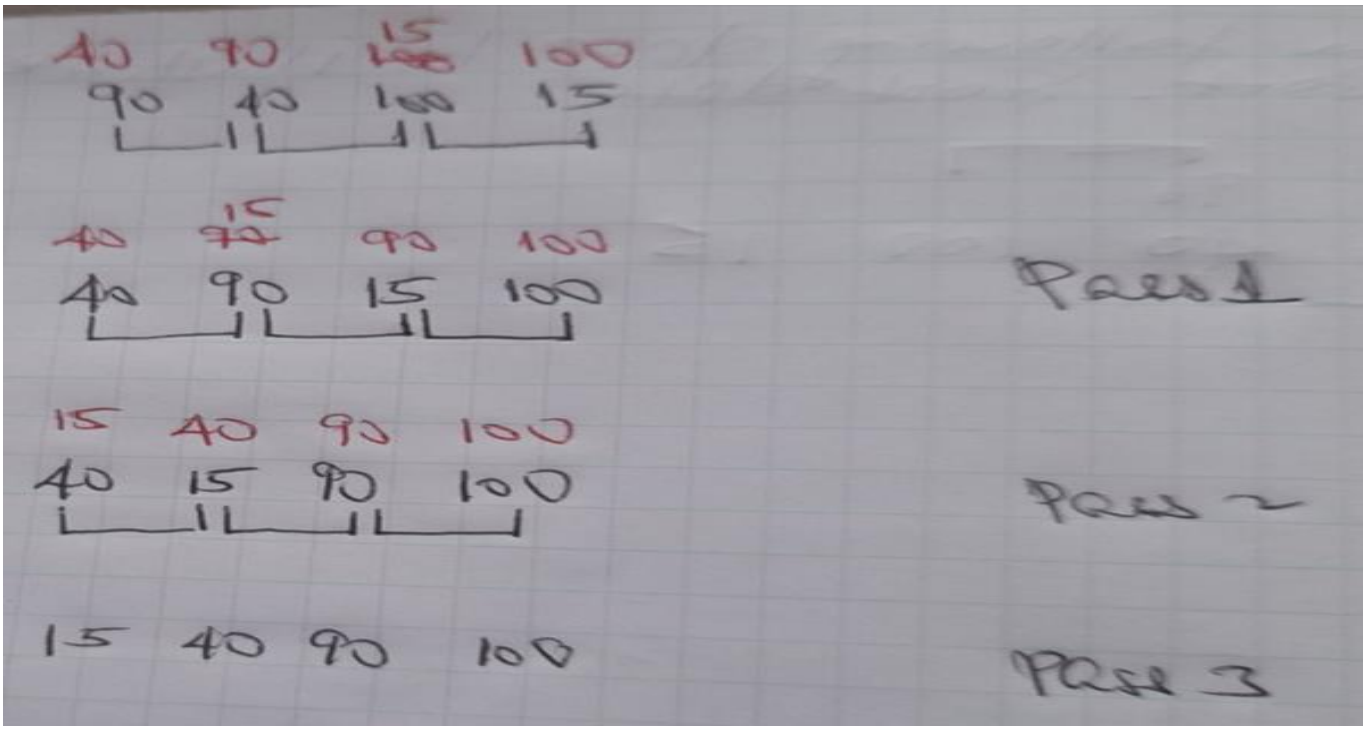
Working of Bubble Sort

1. **Pass through the list**: Starting from the first element, compare the current element with the next one.
2. **Swap if necessary**: If the current element is greater than the next, swap them.
3. **Repeat the process**: After each pass, the largest unsorted element "bubbles up" to its correct position.
4. **Continue passes**: Repeat this process for the remaining unsorted elements.

Let's illustrate **Bubble Sort** step-by-step on the array [90, 40, 100, 15].

Initial array:

[90, 40, 10, 15]



In **Bubble Sort**, the number of passes required to sort the data depends on the size of the array (denoted by n) and how the data is arranged.

- **Worst-case scenario**: In the worst case, where the data is in reverse order
- **Best-case scenario**: If the array is already sorted, Bubble Sort can terminate early with just 1 pass, if optimized with a "swapped" flag to detect when no swaps were made.

General Rule:

- **Maximum number of passes required**: n - 1, where n is the number of elements in the array.

C++ program that prompts a user to enter 10 values in an array, sorts them using Bubble Sort in descending order, and then displays the values before and after sorting:

```
#include <iostream>

using namespace std;

// Function to perform Bubble Sort in descending order

void bubbleSortDescending(int arr[], int n) {

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

```
        bool swapped = false;
        // Last i elements are already sorted
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] < arr[j+1]) {
                // Swap if the current element is smaller than the next one
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = true;
            }
        }
        // If no two elements were swapped, break the loop
        if (!swapped) break;
    }
}
// Function to display the array
void displayArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
int main() {
    const int SIZE = 10;
    int arr[SIZE];
    // Prompting user to enter 10 values
    cout << "Enter 10 values:" << endl;
```

```
    for (int i = 0; i < SIZE; i++) {
        cout<< "Enter value in cell :" <<i<< "\n";
        cin >> arr[i];
    }
    // Display values before sorting
    cout << "Array before sorting: ";
    displayArray(arr, SIZE);
    // Sort the array in descending order using Bubble Sort
    bubbleSortDescending(arr, SIZE);
    // Display values after sorting
    cout << "Array after sorting in descending order: ";
    displayArray(arr, SIZE);
    return 0;
}
```

Explanation:

- The program uses a **Bubble Sort** algorithm, but it compares elements in such a way that the larger elements are placed at the beginning, thus sorting the array in descending order.
- The bubbleSortDescending function handles the sorting logic.
- The displayArray function is used to print the array values before and after sorting.

```
Enter 10 values:
Enter value in cell :0
67
Enter value in cell :1
86
Enter value in cell :2
12
Enter value in cell :3
43
Enter value in cell :4
90
Enter value in cell :5
14
Enter value in cell :6
3
Enter value in cell :7
87
Enter value in cell :8
54
Enter value in cell :9
41
Array before sorting: 67 86 12 43 90 14 3 87 54 41
Array after sorting in descending order: 90 87 86 67 54 43 41 14 12 3
```

Advantages of Bubble Sort

1. **Simple to understand and implement**.
2. **In-place sorting**: Requires no additional space beyond the input array.
3. **Stable sorting**: It preserves the relative order of equal elements.

Disadvantages of Bubble Sort

1. **Inefficient for large datasets**: The time complexity is $O(n^2)$, which makes it impractical for large arrays.
2. **Not adaptive**: Even if the array is partially sorted, Bubble Sort will still make unnecessary comparisons in its standard form.

Bubble Sort is mostly used for small datasets or when simplicity is preferred over performance.

Selection Sort

Selection Sort is a simple comparison-based sorting algorithm that works by repeatedly selecting the smallest (or largest, depending on sorting order) element from the unsorted portion of the array and swapping it with the first unsorted element. This process is repeated until the array is sorted.

Working of Selection Sort:

1. **Find the minimum (or maximum)** element in the unsorted part of the array.

2. **Swap** it with the first unsorted element.
3. **Repeat** the process for the remaining unsorted part of the array.

Example:

Let's sort the array [29, 10, 14, 37, 13] in ascending order using Selection Sort:

Initial array: [29, 10, 14, 37, 13]

1st Pass:

- Find the minimum value in the array: 10
- Swap it with the first element (29): [10, 29, 14, 37, 13]

2nd Pass:

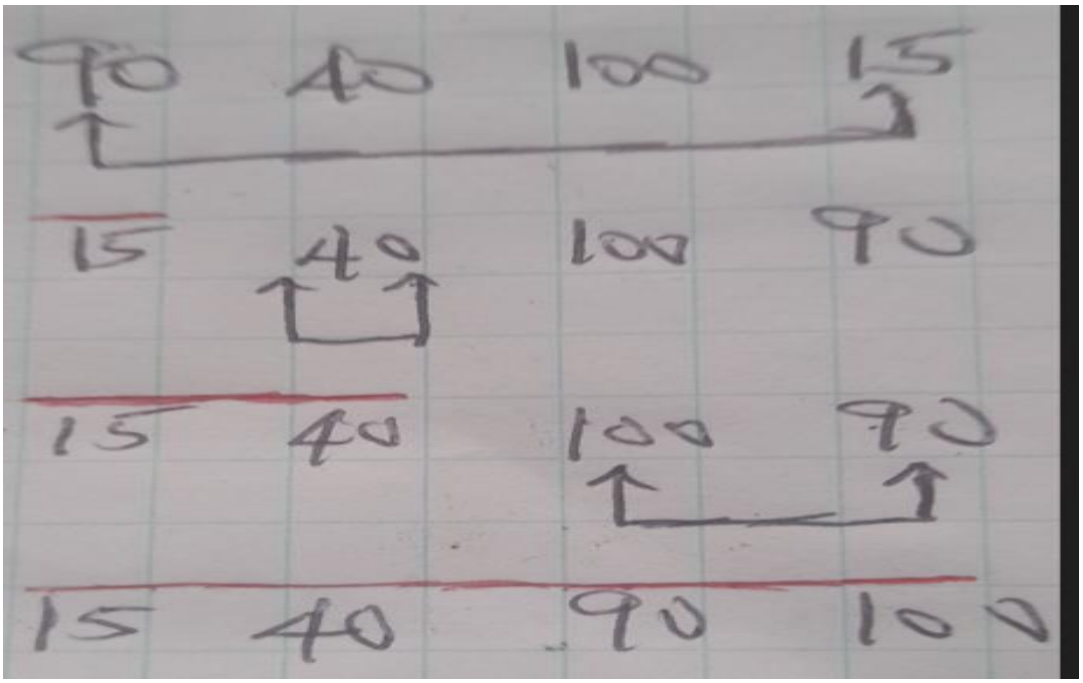
- Find the minimum in the remaining array ([29, 14, 37, 13]): 13
- Swap it with the second element (29): [10, 13, 14, 37, 29]

3rd Pass:

- Find the minimum in the remaining array ([14, 37, 29]): 14 (already in the correct position)

4th Pass:

- Find the minimum in the remaining array ([37, 29]): 29
- Swap it with 37: [10, 13, 14, 29, 37]



Time and Space Complexity:

- **Time Complexity**:
  - **Worst-case**: $O(n^2)$ – The algorithm always performs the same number of comparisons, regardless of the array's initial order.
  - **Best-case**: $O(n^2)$ – Even if the array is already sorted, it still performs $O(n^2)$ comparisons.
  - **Average-case**: $O(n^2)$ – On average, it takes quadratic time.
- **Space Complexity**: $O(1)$ – Selection Sort is an **in-place sorting algorithm**, which means it doesn't require extra space apart from the input array.

Advantages of Selection Sort:

1. **Simple to understand and implement**.
2. **In-place sorting**: Requires no additional memory beyond the input array.

Disadvantages of Selection Sort:

1. **Inefficient for large datasets**: $O(n^2)$ time complexity makes it impractical for sorting large datasets.
2. **Not adaptive**: Selection Sort does not take advantage of existing order in the array. Even if the array is partially sorted, it still performs all comparisons.

C++ program that prompts the user to enter 10 values into an array, sorts them using Selection Sort in ascending order, and then displays the array before and after sorting.

```
#include <iostream>
using namespace std;
// Function to perform Selection Sort in ascending order
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        // Find the minimum element in the unsorted part
        int minIndex = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap the found minimum element with the first element of the unsorted part
        if (minIndex != i) {
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}
// Function to display the array
void displayArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
```

```
}
int main() {
    const int SIZE = 10;
    int arr[SIZE];
    // Prompting the user to enter 10 values into the array
    cout << "Enter 10 values: " << endl;
    for (int i = 0; i < SIZE; i++) {
        cout<< "\n Enter value in cell   :"<<i<< "\n";
        cin >> arr[i];
    }
    // Display the array before sorting
    cout << "Array before sorting: ";
    displayArray(arr, SIZE);
    // Sort the array using Selection Sort
    selectionSort(arr, SIZE);
    // Display the array after sorting
    cout << "Array after sorting in ascending order: ";
    displayArray(arr, SIZE);
    return 0;
}
```

Explanation:

- The selectionSort function sorts the array in **ascending order** using the selection sort algorithm. It finds the minimum value in each pass and swaps it with the first unsorted element.
- The displayArray function prints the array's values.
- The program prompts the user to input 10 values, sorts them, and displays the result before and after sorting.

Sample Input and Output:

Enter 10 values:
12 45 23 89 5 67 10 34 56 78

Output

Array before sorting: 12 45 23 89 5 67 10 34 56 78
Array after sorting in ascending order: 5 10 12 23 34 45 56 67 78 89

Quick Sort

Quick Sort is an efficient, in-place sorting algorithm that uses the divide-and-conquer approach to sort an array. It works by selecting a pivot element and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

Steps in Quick Sort:

1. **Choose a pivot** (the first, last, or a random element).
2. **Partition the array**: Rearrange the array such that all elements less than the pivot are on the left side, and all elements greater than the pivot are on the right.
3. **Recursively apply Quick Sort** to the left and right sub-arrays.
4. Combine the results.

Let's illustrate **Quick Sort** in ascending order using the elements 90, 40, 100, 15.

Given Array:
[90, 40, 100, 14]
Step-by-step Process:

1. **Step 1: Choose a Pivot**
    - Let's choose the last element 15 as the pivot.
2. **Step 2: Partitioning**
    - We start by comparing each element with the pivot 15.
    - First, compare 90 with 14: Since 90 > 15, no swap.
    - Next, compare 40 with 14: Since 40 > 15, no swap.
    - Compare 100 with 14: Since 100 > 15, no swap.
    - Now, place the pivot 14 in its correct position by swapping it with the first element (90):

    After the first partition:

    [15, 40, 100, 90]

    Pivot 15 is now correctly placed.

3. **Step 3: Recursively Apply Quick Sort**
    - Now, we quick sort the left sub-array [] (empty) and the right sub-array [40, 100, 90].

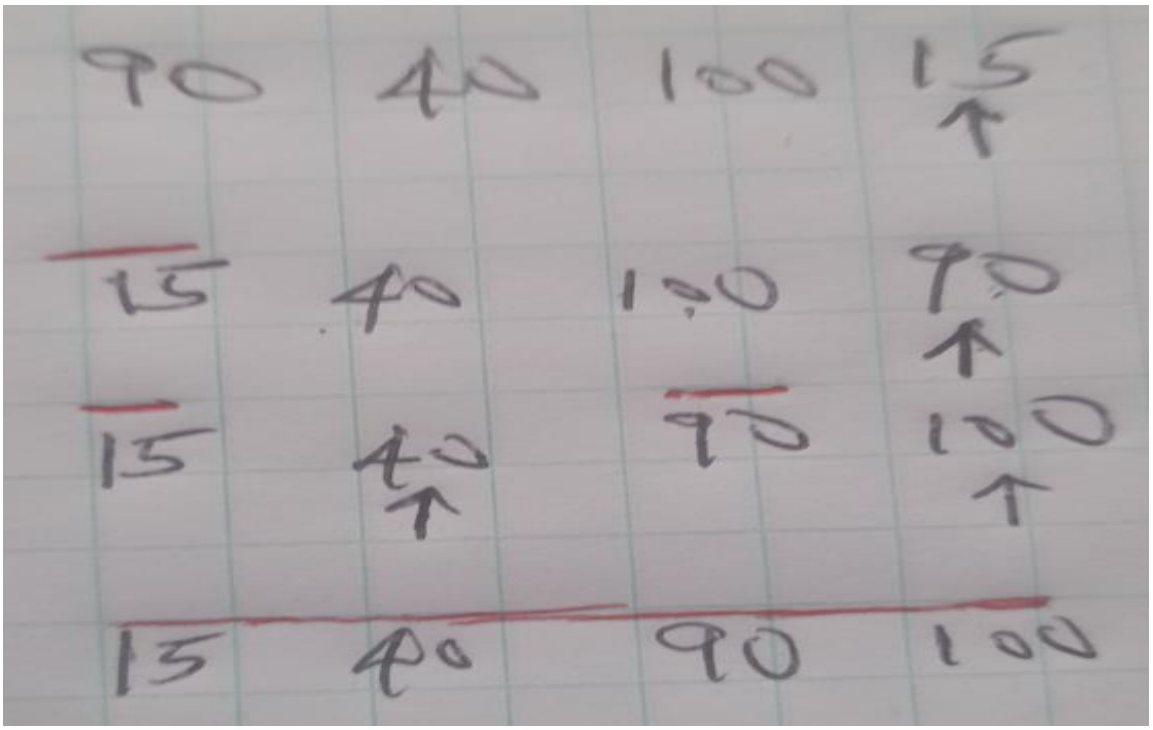
4. **Step 4: Sort Right Sub-array [40, 100, 90]**
    - Choose the last element 90 as the pivot.
    - Compare 40 with 90: Since 40 < 90, swap 40 with itself.
    - Compare 100 with 90: Since 100 > 90, no swap.
    - Place the pivot 90 in its correct position by swapping it with the element after 40:

    After partitioning:

    [15, 40, 90, 100]

5. **Step 5: Recursively Apply Quick Sort**
    - Quick sort the left sub-array [40] (already sorted) and the right sub-array [100] (already sorted).
6. **Final Sorted Array**:

    csharp
    Copy code
    [15, 40, 90, 100]



Time and Space Complexity of Quick Sort:

- **Time Complexity**:
  - **Best-case**: O(n log n) – when the pivot divides the array evenly.
  - **Worst-case**: O(n^2) – when the pivot is the smallest or largest element repeatedly (this can be mitigated using random pivots or better pivot strategies).
  - **Average-case**: O(n log n) – typically efficient in practice.
- **Space Complexity**: O(log n) – due to recursive function calls.

Advantages of Quick Sort:

1. **Efficient**: O(n log n) average time complexity, making it faster than algorithms like Bubble Sort and Insertion Sort.
2. **In-place sorting**: Requires only a small, constant amount of extra storage.
3. **Widely used in practice**: Quick Sort is one of the most efficient sorting algorithms for large datasets.

Disadvantages of Quick Sort:

1. **Worst-case performance**: O(n^2) can occur when the pivot choice is poor (e.g., always the smallest or largest element).
2. **Not stable**: It does not preserve the relative order of equal elements (though this can be mitigated with modifications).
3. **Recursive nature**: For very large datasets, the recursive approach may lead to stack overflow in some systems.

Quick Sort is preferred for **large datasets** due to its speed and efficiency in the average case

Heap Sort

Heap Sort Algorithm

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It is an in-place sorting technique and is not stable.

To sort data in ascending order, the algorithm first builds a **max heap** from the input data, then repeatedly extracts the maximum element from the heap and places it at the end of the sorted list.

To sort data in descending order, the algorithm first builds a **min heap** from the input data, then repeatedly extracts the minimum element from the heap and places it at the end of the sorted list.

Steps of Heap Sort: Ascending order

1. **Build a Max Heap**: Rearrange the array into a max heap, where the largest element is at the root (index 0).
2. **Swap and Heapify**: Swap the root (maximum value) with the last element, reduce the heap size by one, and then heapify the root to maintain the max heap property.
3. **Repeat**: Continue this process until all elements are sorted.

Heap Sort Example (Using array: 90, 40, 100, 15):
Initial Array:

[90, 40, 100, 15]
Step-by-step Process:

1. **Step 1: Build Max Heap**
    - Start by building a max heap from the unsorted array.
    - The array after the first heapification will look like this:

    [100, 40, 90, 15]

    - 100 is the largest element and is now the root of the heap.
2. **Step 2: Swap the Root and Last Element**
    - Swap the root element (100) with the last element (15):

    [15, 40, 90, 100]

    - Now, 100 is in its correct final position.
3. **Step 3: Heapify the Reduced Heap**
    - Heapify the remaining heap [15, 40, 90]:
    - Swap 15 with 90 (the larger child):

    [90, 40, 15, 100]

4. **Step 4: Swap the Root and Last Element**
    - Swap the root element (90) with the second-last element (15):

    [15, 40, 90, 100]

    - Now, 90 is in its correct final position.
5. **Step 5: Heapify the Reduced Heap**

- Heapify the remaining heap [15, 40]:
- Swap 14 with 40:

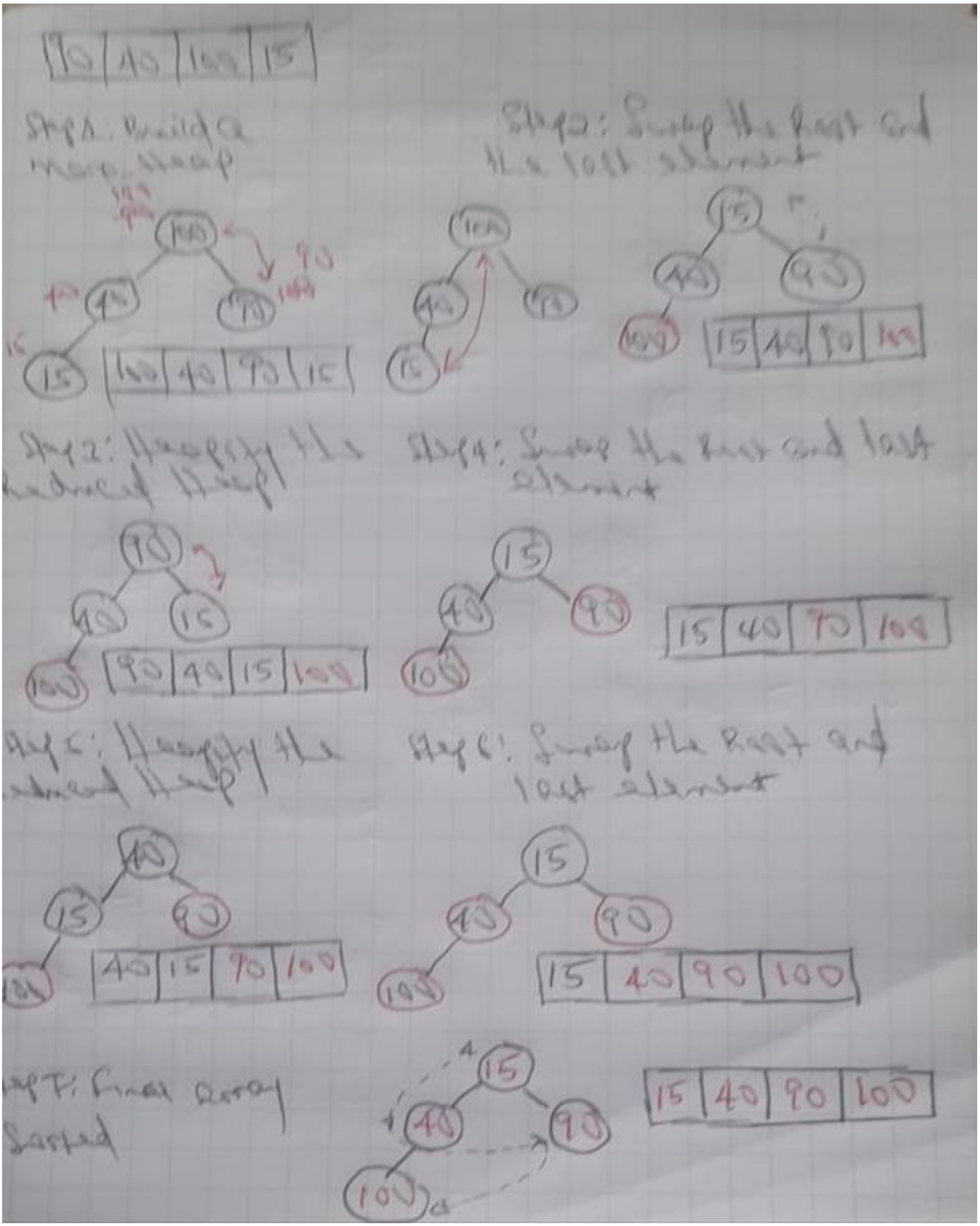
[40, 15, 90, 100]

6. **Step 6: Swap the Root and Last Element**
   - Swap the root element (40) with the last remaining element (15):

   [15, 40, 90, 100]

7. **Step 7: Final Sorted Array**
   - The array is now sorted:

   [15, 40, 90, 100]



Comparison of Sorting Algorithms:

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Stable
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	No
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No

Conclusion:

The choice of sorting algorithm depends on the data size, the distribution of data, and whether or not you need stability. For most practical purposes, **Quick Sort** is often the fastest in average cases. **Heap Sort** is useful when in-place sorting with O(n log n) time complexity is needed.