Data Structure Using C
Question Paper Pattern and Questions

1. Data Structured Types

- NOT YET

2. Time Complexity

-

1)Time complexity measures how long an algorithm takes to run based on the size of its input. It shows the efficiency of an algorithm.

2) It helps us know how fast an algorithm runs and Useful to compare algorithms.

3) It also predicts performance in the best, average, and worst cases.

4) Types of Time Complexities:-

1. **O(1): Constant Time**
   - Takes the same time regardless of the input size.
   - Example: Accessing an array element by index.
2. **O(log n): Logarithmic Time**
   - Time grows slowly as the input size increases.
   - Example: Binary Search.
3. **O(n): Linear Time**
   - Time grows directly with the input size.
   - Example: Traversing an array or linked list.
4. **O(n log n): Log-Linear Time**
   - Example: Merge Sort and Quick Sort.
5. **O(n^2): Quadratic Time**
   - Time grows quickly as input size increases.
   - Example: Bubble Sort, Selection Sort.
6. **O(2^n): Exponential Time**
   - Time doubles with each increase in input size.
   - Example: Solving problems with brute-force methods.
7. **O(n!): Factorial Time**
   - Example: Generating all permutations of input

5)Time Complexities of Common Data Structures are –

a) Arrays

- **Access:** O(1)
- **Search:** O(n)
- **Insert/Delete:**
    - At the beginning or middle: O(n) (requires shifting).
    - At the end: O(1)

b) Linked Lists

- **Search:** O(n)
- **Insert/Delete:**
    - At the beginning: O(1)
    - At the middle or end: O(n)

c) Stacks

- **Push, Pop, Peek:** O(1)

d) Queues

- **Enqueue, Dequeue:** O(1)
- **Search:** O(n)

e) Trees

- **Search, Insert, Delete (Balanced BST):** O(log n)
- **Traversal (Inorder, Preorder, Postorder):** O(n)

f) Graphs

- **Adjacency Matrix:**
    - Space Complexity: $O(V^2)$
    - Traversal: $O(V^2)$
- **Adjacency List:**
    - Space Complexity: O(V + E)
    - Traversal: O(V + E)

5. Common Algorithms and Their Time Complexities

Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$

Key Points to Remember

- Time complexity measures how efficient an algorithm is.
- Lower time complexity (like O(1) or O(log n)) is better for large inputs.
- Choose the best algorithms and data structures for solving problems efficiently.

EXAMPLE :

Binary Search in C

```
#include <stdio.h>

int binarySearch(int arr[], int size, int key) {

   int low = 0, high = size - 1;

   while (low <= high) {

      int mid = (low + high) / 2;

      if (arr[mid] == key)

         return mid;

      else if (arr[mid] < key)

         low = mid + 1;

      else

         high = mid - 1;
```

```
    }
    return -1; // Key not found
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key = 30;
    int result = binarySearch(arr, size, key);
    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");
    return 0;
}
```

3. Breadth First Search (BFS)

•

BFS is a graph or tree traversal algorithm that explores nodes **level by level**, starting from a source node and moving to its connected neighbors before moving deeper.It is particularly useful for finding the shortest path in unweighted graphs. BFS is simple to implement and efficient for finding the shortest path in an unweighted graph. It also guarantees that all the vertices in the graph are visited. For example, Imagine you are exploring a family tree starting from yourself. You first visit all your immediate relatives (siblings, parents), then move to the next level (your cousins, grandparents), and so on.

Key Points of BFS:

1.BFS uses a **queue** data structure (First-In-First-Out).

2.It explores nodes in levels.

3.BFS works for both graphs and trees.

4.BFS ensures that nodes are visited in the order they are discovered.

How BFS Works:

1. Start from a starting node (source node).
2. Visit all the nodes connected to the current node level by level.
3. Use a Queue to keep track of nodes to visit next.
4. Mark visited nodes to avoid revisiting them.

Example BFS

If the graph looks like this (starting from node 0):

```
  0

 / \

1   2

|   |

3   4
```

- Start at 0.
- Visit 1 and 2 (level 1).
- Then visit 3 and 4 (level 2).

BFS Traversal Order: 0 → 1 → 2 → 3 → 4.

Example of BFS in C:

```
#include <stdio.h>

#include <stdbool.h>

#define MAX 100

int queue[MAX], front = 0, rear = 0;

bool visited[MAX];

void enqueue(int value) {

   queue[rear++] = value;

}

int dequeue() {

   return queue[front++];

}

bool isEmpty() {

   return front == rear;

}

void BFS(int graph[MAX][MAX], int vertices, int startNode) {

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

      visited[i] = false; // Initialize all nodes as unvisited

   }

   enqueue(startNode);

  visited[startNode] = true;
```

```
    printf("BFS Traversal: ");
    while (!isEmpty()) {
        int currentNode = dequeue();
        printf("%d ", currentNode);
        for (int i = 0; i < vertices; i++) {
            if (graph[currentNode][i] == 1 && !visited[i]) {
                enqueue(i);
                visited[i] = true;
            }
        }
    }
    printf("\n");
}
int main() {
    int graph[MAX][MAX] = {
        {0, 1, 1, 0, 0},
        {1, 0, 1, 1, 0},
        {1, 1, 0, 0, 1},
        {0, 1, 0, 0, 1},
        {0, 0, 1, 1, 0}
    };
 int vertices = 5;
    int startNode = 0;
    BFS(graph, vertices, startNode);
    return 0; }
```

4. Priority Queue

-

1. A **Priority Queue** is a special type of data structure that operates similarly to a regular queue but with an important difference: each element in a priority queue has a "priority" associated with it.

2. Elements with higher priority are served before elements with lower priority.

3. If two elements have the same priority, they are served according to their order in the queue (this is known as a "first-in, first-out" or FIFO order).

4. When you add an element to the priority queue, you also specify its priority. The queue will arrange itself so that the highest priority elements are at the front.

5. We can implement a Priority Queue in multiple ways Like **Using Arrays**, **Using Linked Lists** or **Using Heaps** (most efficient).

6. The ordering of elements in a Priority Queue is done dynamically.

7. How Priority Queue Works:

   1. Assign Priority: Each element has an associated priority (for example, numbers like 1, 2, 3).
   2. Enqueue (Insert): Add the element into the queue based on its priority.
      - High priority elements are placed earlier.
   3. Dequeue (Remove): Remove and return the element with the highest priority first.
   4. Maintain Order: If elements have the same priority, they are processed in the order they were added (FIFO).

8. Example of it in C

```
#include <stdio.h>

#include <limits.h> // For INT_MAX

#define MAX 10

int priorityQueue[MAX]; // Array to hold the elements

int size = 0;          // Current size of the queue

// Function to insert an element into the priority queue

void insert(int value) {

   if (size == MAX) {

      printf("Priority Queue is full!\n");

      return;

   }

   priorityQueue[size] = value; // Add the value at the end

   size++;

}

// Function to remove and return the element with the highest priority (smallest value)

int deleteHighestPriority() {

   if (size == 0) {

      printf("Priority Queue is empty!\n");

      return -1;

   }

   // Find the smallest value (highest priority)

   int minIndex = 0;

   for (int i = 1; i < size; i++) {

      if (priorityQueue[i] < priorityQueue[minIndex]) {
           minIndex = i;
      }
   }
   int highestPriority = priorityQueue[minIndex]; // Store the highest priority element
   // Shift elements to fill the gap
   for (int i = minIndex; i < size - 1; i++) {
      priorityQueue[i] = priorityQueue[i + 1];
   }
   size--;
   return highestPriority;
}
// Function to display the priority queue
void display() {
   if (size == 0) {
      printf("Priority Queue is empty!\n");
      return;
   }
   printf("Priority Queue: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", priorityQueue[i]);
   }
   printf("\n");
}
int main() {
   insert(30);
   insert(10);

   insert(20);

   insert(5);

   printf("Before deletion:\n");

   display();

   printf("Deleted highest priority element: %d\n", deleteHighestPriority());

   printf("After deletion:\n");

display();

   return 0;

}
```

5. Depth First Search (DFS)

->

1. **Depth First Search (DFS)** is a fundamental algorithm used for traversing or searching through data structures like trees and graphs.

2. The main idea behind DFS is to explore as far as possible along each branch before backtracking.

3. This means that it goes deep into one path before trying another path.

4. **Graph Representation**: A graph can be represented using an adjacency list or an adjacency matrix. In an adjacency list, each node has a list of nodes it is connected to.

5. DFS starts at a selected node (often called the "root" in trees) and explores as far as possible down one branch before backtracking to explore other branches.

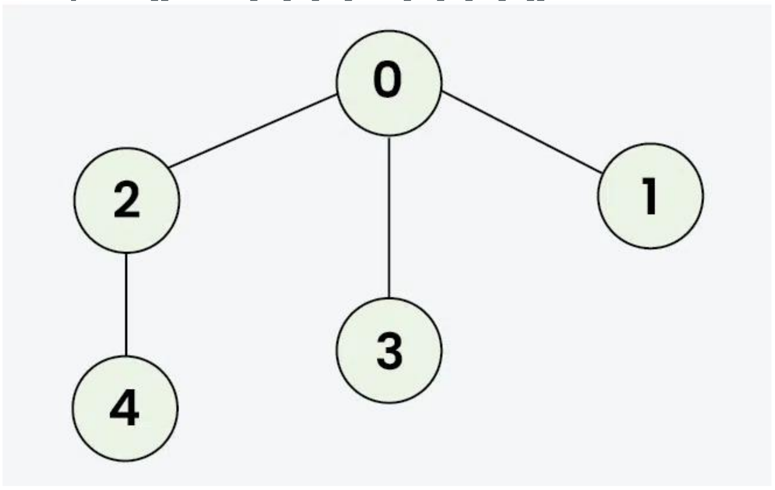
6. DFS can be implemented using a stack data structure. This can be done either explicitly (using a stack data structure) or implicitly (using recursion, which uses the call stack).

7. To avoid visiting the same node multiple times, we maintain a record of visited nodes.

8. How DFS Works:

1. **Start at the source node** (root).
2. Visit the current node and mark it as **visited**.
3. **Explore each neighbor** of the current node that has not yet been visited.
4. Move deeper into the graph along a path until no further nodes can be visited.
5. Backtrack and repeat the process for unvisited nodes.

9. ***Input:*** *[[2,3,1], [0], [0,4], [0], [2]]*



Output: *0 2 4 3 1*
Explanation: *DFS Steps:*

- *Start at 0: Mark as visited. Output: 0*
- *Move to 2: Mark as visited. Output: 2*
- *Move to 4: Mark as visited. Output: 4 (backtrack to 2, then backtrack to 0)*
- *Move to 3: Mark as visited. Output: 3 (backtrack to 0)*
- *Move to 1: Mark as visited. Output: 1*

10. **Example of DFS in C:**

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 100
bool visited[MAX];
```

```
// Function to perform DFS
void DFS(int graph[MAX][MAX], int vertices, int currentNode) {
   printf("%d ", currentNode);
   visited[currentNode] = true;
   for (int i = 0; i < vertices; i++) {
      if (graph[currentNode][i] == 1 && !visited[i]) {
         DFS(graph, vertices, i);
      }
   }
}
int main() {
   int graph[MAX][MAX] = {
      {0, 1, 1, 0, 0},
      {1, 0, 1, 1, 0},
      {1, 1, 0, 0, 1},
      {0, 1, 0, 0, 1},
      {0, 0, 1, 1, 0}
   };
   int vertices = 5;
   // Initialize visited array
   for (int i = 0; i < vertices; i++) {
      visited[i] = false;
   }
   printf("DFS Traversal: ");
   DFS(graph, vertices, 0); // Start DFS from node 0
   return 0;
}
```

6. Algorithm Characteristics
->

1. An algorithm in data structures is a step-by-step procedure or formula for solving a problem or performing a task.

2. It consists of a sequence of instructions that can be executed to achieve a specific outcome.

3. Algorithms are essential in data structures because they define how data is manipulated, processed, and stored.

4. Algorithm Characteristics are :-

1. Finiteness

An algorithm must always terminate after a finite number of steps.

2. Definiteness

Each step of an algorithm must be precisely defined.

3. Input

An algorithm should accept zero or more inputs

4. Output

An algorithm should produce one or more outputs

5. Effectiveness

The operations to be performed in the algorithm should be basic enough that they can be done in a finite amount of time.

6. Generality

It should be able to solve a class of problems or perform a set of operations on a variety of inputs.

7. Efficiency

This characteristic often involves analyzing time and space complexity.

8. Correctness

An algorithm must produce the correct output for all valid inputs.

7. DEQUE, IRDEQ and ORDEQ

->

***Deque (Double-Ended Queue):-**

1. A Deque (pronounced "deck") is a data structure that allows insertion and deletion of elements from both ends, i.e., the front and the back.
2. It is a generalized version of a queue where you can add or remove elements from either end.
3. You can insert and delete elements from both the front and the back.
4. The size of a deque can grow and shrink as needed, unlike static data structures like arrays.
5. Deques can be used to implement both stacks and queues.
6. Characteristics of DEQUE:
    1. **Double-ended operations** – You can insert and delete from both the front and rear.
    2. **Dynamic Size** – The size of the DEQUE can grow or shrink as needed.
    3. **Supports Two Types of Operations**:
        - Insert at front, delete at front
        - Insert at rear, delete at rear
7. Basic Operations you can perform is as follows :-
    1. **Insert Front**: Add an element to the front of the deque.
    2. **Insert Rear**: Add an element to the back of the deque.
    3. **Delete Front**: Remove an element from the front of the deque.
    4. **Delete Rear**: Remove an element from the back of the deque.
    5. **Peek Front**: View the front element without removing it.
    6. **Peek Rear**: View the rear element without removing it.

***IRDEQ (Input Restricted Deque)**

1. An Input Restricted Deque is a type of deque where insertion is allowed only at one end (either front or rear), but deletion can occur from both ends.
2. This structure is useful when you want to maintain the flexibility of removing elements from either end while restricting where new elements can be added.
3. Characteristics of IRDEQ:
    1. **Input Restriction**: New elements can only be added at one end.
    2. **Flexible Deletion**: Elements can be removed from both ends.

3. **Dynamic Size**: Like a regular deque, it can grow and shrink dynamically.

***ORDEQ (Output Restricted Deque)**
1. An Output Restricted Deque is a type of deque where deletion is allowed only at one end (either front or rear), but insertion can occur at both ends.
2. This structure is useful when you want to maintain the flexibility of adding elements from either end while restricting where elements can be removed.
3. Characteristics of ORDEQ:
   1. **Output Restriction**: Elements can only be removed from one end.
   2. **Flexible Insertion**: New elements can be added at both ends.
   3. **Dynamic Size**: Like a regular deque, it can grow and shrink dynamically.

8. Define degree of a tree, generation, pendent node, depth of a tree ->

1. Degree of a Tree

- **Definition:** The degree of a tree is defined as the maximum number of children that any node in the tree can have. In other words, it is the highest number of direct descendants (children) of any node in the tree.
- **Example:** In a binary tree, the degree is 2 because each node can have at most two children (left and right). In a ternary tree, the degree is 3 because each node can have up to three children.

2. Generation

- **Definition:** The generation of a node in a tree refers to the level or depth of that node in relation to the root node. The root node is considered to be at generation 0, its children are at generation 1, their children are at generation 2, and so on.
- **Example:** In a tree where the root is at generation 0, if a node is a child of the root, it is at generation 1. If it has children, those children are at generation 2.

3. Pendent Node

- **Definition**: A pendent node (also known as a leaf node) is a node in a tree that does not have any children. In other words, it is a node that is at the end of a branch and does not lead to any further nodes.
- **Example**: In a binary tree, if a node has no left or right child, it is considered a pendent node. For instance, in the tree below, nodes 4, 5, and 3 are pendent nodes:

```
    1
   / \
  2   3
 / \
4   5
```

4. Depth of a Tree

- **Definition**: The depth of a tree is defined as the length of the longest path from the root node to any leaf node. It is also referred to as the height of the tree. The depth is measured in terms of the number of edges in the longest path from the root to a leaf.
- **Example**: In the tree below, the depth (or height) is 2 because the longest path from the root (1) to the deepest leaf (4 or 5) consists of two edges:

```
    1
   / \
  2   3
 / \
4   5
```

Summary

- **Degree of a Tree:** Maximum number of children any node can have.
- **Generation:** Level of a node in relation to the root (root is generation 0).
- **Pendent Node:** A node with no children (leaf node).
- **Depth of a Tree:** Length of the longest path from the root to a leaf, measured in edges.

9. Explain Omega, Theta and Big Oh

->

In computer science, particularly in the analysis of algorithms, we use **Big O**, **Theta (Θ)**, and **Omega (Ω)** notations to describe the performance or complexity of an algorithm
These notations help us understand how the runtime or space requirements of an algorithm grow as the size of the input increases.

1. Big O Notation (O)

- **Definition**: Big O notation is a way to **measure how fast** an algorithm is or how much **space** it needs as the input size grows. It helps us understand the worst-case scenario like:-
- How much time an algorithm will take when the input is very large.
- How much memory (space) it will need.
- **Purpose**: It helps to understand the maximum amount of time or space an algorithm will require.
- **Example**: If an algorithm has a time complexity of O(n), it means that in the worst case, the time it takes to complete will grow linearly with the size of the input (n). If n doubles, the time taken will also approximately double.
- **Example in C:**

```
void printArray(int arr[], int n) {
 for (int i = 0; i < n; i++) { // O(n)
 printf("%d ", arr[i]);
}
}
```

2. Theta Notation (Θ)

- **Definition**: Theta (Θ) notation describes the **exact growth rate** of an algorithm. It gives us a **tight bound** on how much time or space an algorithm will need. It tells us that the time complexity or space requirement will grow **at the same rate** in **both the best and worst cases**.
- If Big O is like the **worst-case estimate**, Theta (Θ) is like knowing the **exact growth behavior** of the algorithm—it can't grow faster or slower beyond this rate.
- **Purpose:** It helps to understand the average-case scenario for the growth rate of an algorithm.

- **Example:** If an algorithm has a time complexity of Θ(n), it means that the time taken will grow linearly with the size of the input, both in the best and worst cases.
- **Example in C:**

```
void printArray(int arr[], int n) {

  for (int i = 0; i < n; i++) { // Θ(n)

    printf("%d ", arr[i]);

  }

}
```

3. Omega Notation (Ω)

- **Definition**: Omega (Ω) notation describes the **minimum time** or **space** an algorithm will need. It gives a **lower bound** on how well an algorithm can perform.
- It helps us understand the best-case scenario:
- How fast the algorithm can run.
- The least amount of resources it needs.
- It’s like saying, **"This is the minimum time the algorithm will take, no matter how lucky or efficient things are."**
- **Purpose:** It helps to understand the minimum amount of time or space an algorithm will require.
- **Example:** If an algorithm has a time complexity of Ω(n), it means that in the best case, the time it takes to complete will grow linearly with the size of the input.
- **Example in C:**

```
int findMax(int arr[], int n) {
  if (n == 0) return -1; // Ω(1)
  int max = arr[0]; // Ω(1)
  for (int i = 1; i < n; i++) { // Ω(n)
    if (arr[i] > max) {
      max = arr[i];
    }
  }
  return max; // Ω(1)
}
```

Summary of Notations

- **Big O (O)**: Describes the upper bound (worst-case scenario) of an algorithm's time or space complexity. It tells us the maximum time or space an algorithm will take.
- **Theta (Θ)**: Describes a tight bound (exact growth rate) on an algorithm's time or space complexity. It tells us the average-case scenario.
- **Omega (Ω)**: Describes the lower bound (best-case scenario) of an algorithm's time or space complexity. It tells us the minimum time or space an algorithm will take.

10. What is stack ? Write Operation on stack with example (Program)

->

1. A **stack** is a linear data structure that follows the Last In First Out (LIFO) principle, meaning that the last element added to the stack is the first one to be removed.
2. It can be visualized as a stack of plates where you can only add or remove the top plate.
3. Basic Operations on Stack

    1. **Push**: Add an element to the top of the stack.
    2. **Pop**: Remove the top element from the stack.
    3. **Peek/Top**: Retrieve the top element without removing it.
    4. **isEmpty**: Check if the stack is empty.
    5. **Size**: Get the number of elements in the stack.

4. Example of stack:

```
#include <stdio.h>
#define MAX 5 // Maximum size of the stack
int stack[MAX];
int top = -1;
// Push operation
void push(int value) {
   if (top == MAX - 1) {
      printf("Stack Overflow!\n");
   } else {
      stack[++top] = value;
      printf("%d pushed onto the stack.\n", value);
```

```
    }
}

// Pop operation
void pop() {
    if (top == -1) {
        printf("Stack Underflow!\n");
    } else {
        printf("%d popped from the stack.\n", stack[top--]);
    }
}
// Peek operation
void peek() {
    if (top == -1) {
        printf("Stack is empty.\n");
    } else {
        printf("Top element is %d.\n", stack[top]);
    }
}
// Main function
int main() {
    push(10);
    push(20);
    push(30);
    peek();
    pop();
    peek();
    return 0;
}
```

Output:

```
10 pushed onto the stack.
20 pushed onto the stack.
30 pushed onto the stack.
Top element is 30.
30 popped from the stack.
Top element is 20.
```

11. Explain different types of AVL Rotation with an example

->

1. AVL trees are a type of self-balancing binary search tree where the difference in heights between the left and right subtrees (the balance factor) is at most 1 for every node.

2. When an insertion or deletion operation causes the tree to become unbalanced, rotations are performed to restore balance.

4. There are four types of rotations used in AVL trees:

   1. **Right Rotation (Single Rotation)**
   2. **Left Rotation (Single Rotation)**
   3. **Left-Right Rotation (Double Rotation)**
   4. **Right-Left Rotation (Double Rotation)**

1. Right Rotation (Single Rotation)

A right rotation is performed when a node becomes unbalanced due to an insertion in the left subtree of its left child.

EXAMPLE:-

Consider inserting the following values in order: 30, 20, 10.

- After inserting 30:

```
    30
```

- After inserting 20:

```
    30
   /
 20
```

- After inserting 10:

```
      30
     /
   20
  /
10
```

The tree is unbalanced at node 30 (balance factor = 2). A right rotation is performed around node 30.

After Right Rotation:

```
   20
  /  \
10    30
```

2. Left Rotation (Single Rotation)

A left rotation is performed when a node becomes unbalanced due to an insertion in the right subtree of its right child.

Example:

Consider inserting the following values in order: 10, 20, 30.

- After inserting 10:

```
10
```

- After inserting 20:

```
10
  \
   20
```

- After inserting 30:

```
10
  \
   20
     \
      30
```

The tree is unbalanced at node 10 (balance factor = -2). A left rotation is performed around node 10.

After Left Rotation:

```
   20
  /  \
10   30
```

3. Left-Right Rotation (Double Rotation)

A left-right rotation is performed when a node becomes unbalanced due to an insertion in the right subtree of its left child.

Example:

Consider inserting the following values in order: 30, 10, 20.

- After inserting 30:

```
30
```

- After inserting 10:

```
    30
   /
10
```

- After inserting 20:

```
    30
   /
 10
   \
    20
```

The tree is unbalanced at node 30 (balance factor = 2). A left-right rotation is performed, which consists of a left rotation on node 10 followed by a right rotation on node 30.

After Left Rotation on 10:

```
      30
     /
   20
  /
10
```

After Right Rotation on 30:

```
    20
   /  \
 10   30
```

4. Right-Left Rotation (Double Rotation)

A right-left rotation is performed when a node becomes unbalanced due to an insertion in the left subtree of its right child.

Example:

Consider inserting the following values in order: 10, 30, 20.

- **After inserting 10:**

```
10
```

- **After inserting 30:**

```
10
  \
   30
```

- **After inserting 20:**

```
  10
    \
     30
    /
  20
```

The tree is unbalanced at node 10 (balance factor = -2). A right-left rotation is performed, which consists of a right rotation on node 30 followed by a left rotation on node 10.

After Right Rotation on 30:

```
10
  \
   20
     \
      30
```

After Left Rotation on 10:

```
  20
 /  \
10   30
```

Summary of Rotations

- **Right Rotation:** Used when a left-heavy subtree is unbalanced due to a left child insertion.
- **Left Rotation:** Used when a right-heavy subtree is unbalanced due to a right child insertion.
- **Left-Right Rotation:** Used when a left-heavy subtree is unbalanced due to a right child insertion in the left child.
- **Right-Left Rotation:** Used when a right-heavy subtree is unbalanced due to a left child insertion in the right child.

These rotations help maintain the balance of the AVL tree, ensuring that operations such as insertion, deletion, and lookup remain efficient.

Imbalance Type	**Rotation Type**	**Steps**
Left-Left (LL)	Right Rotation	Single right rotation
Right-Right (RR)	Left Rotation	Single left rotation
Left-Right (LR)	Left-Right	Left rotation + Right rotation
Right-Left (RL)	Right-Left	Right rotation + Left rotation

12. What is Graph ? Explain Adjacency list of graph ,adjacency matrix with example

->

1. A graph is a data structure consisting of:

- Vertices (or nodes): Points in the graph.
- Edges: Connections between the vertices.

2. Graphs can be:

1. Directed: Edges have a direction.
2. Undirected: Edges have no direction.
3. Weighted: Edges have weights or costs associated with them.
4. Unweighted: Edges have no weights.

3. Representation of a Graph

Two common ways to represent a graph are:

1. Adjacency List
2. Adjacency Matrix

4. Adjacency List

Definition: An adjacency list represents a graph as a collection of lists. Each vertex has a list of all its connected vertices.

Efficient for Sparse Graphs: It uses less space when the graph has fewer edges.

Example:

Consider the following undirected graph:

```
  A
 / \
B   C
 \ /
  D
```

The adjacency list representation of this graph would look like this:

A: [B, C]

B: [A, D]

C: [A, D]

D: [B, C]

In this representation:

- Vertex A is connected to vertices B and C.
- Vertex B is connected to vertices A and D.
- Vertex C is connected to vertices A and D.
- Vertex D is connected to vertices B and C.

2. Adjacency Matrix

An adjacency matrix is a 2D array (or matrix) used to represent a graph. The rows and columns of the matrix represent the vertices, and the value at the intersection of row (i) and column (j) indicates whether there is an edge between vertex (i) and vertex (j). If there is an edge, the value is typically 1 (or the weight of the edge), and if there is no edge, the value is 0.

Example:

Using the same graph as before:

```
   A
  / \
 B   C
  \ /
   D
```

The adjacency matrix representation of this graph would look like this:

```
     A  B  C  D

A: [0, 1, 1, 0]

B: [1, 0, 0, 1]

C: [1, 0, 0, 1]

D: [0, 1, 1, 0]
```

In this matrix:

- The row and column headers represent the vertices A, B, C, and D.
- A value of 1 indicates an edge between the corresponding vertices, while a value of 0 indicates no edge.

Summary

Adjacency List: Efficient for sparse graphs, where each vertex maintains a list of its neighbors. It uses less memory compared to an adjacency matrix, especially when the number of edges is much lower than the square of the number of vertices.

Adjacency Matrix: Suitable for dense graphs, where the number of edges is close to the maximum possible. It provides quick access to check if an edge exists between any two vertices but consumes more memory, as it requires (O(V^2)) space, where (V) is the number of vertices.

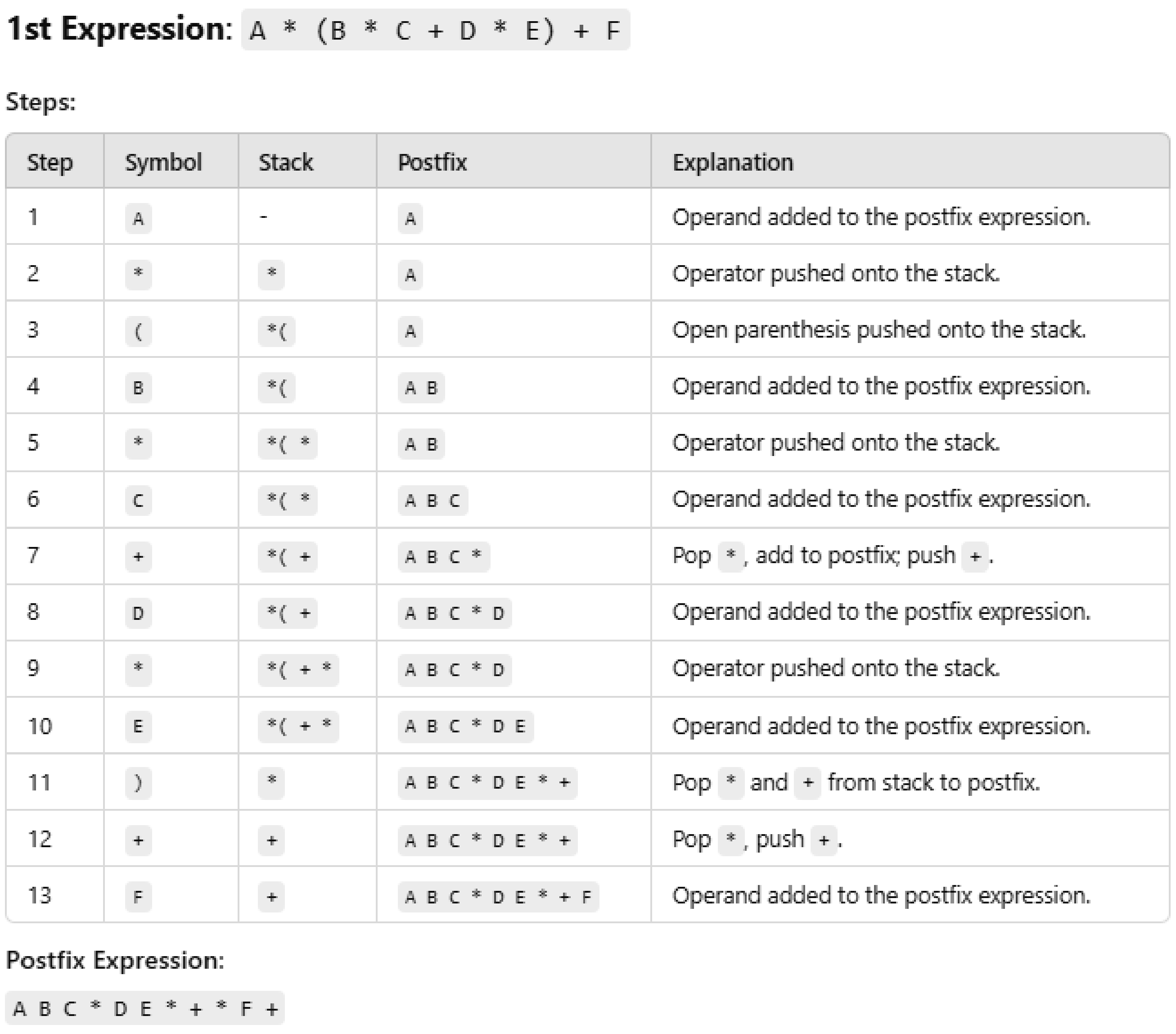
13. Convert the following Infix expression into postfix using a table.

1. A*(B*C+D*E)+F

2. A ^ (B – C * D /E) + F

->

1. A*(B*C+D*E)+F



1st Expression: `A * (B * C + D * E) + F`

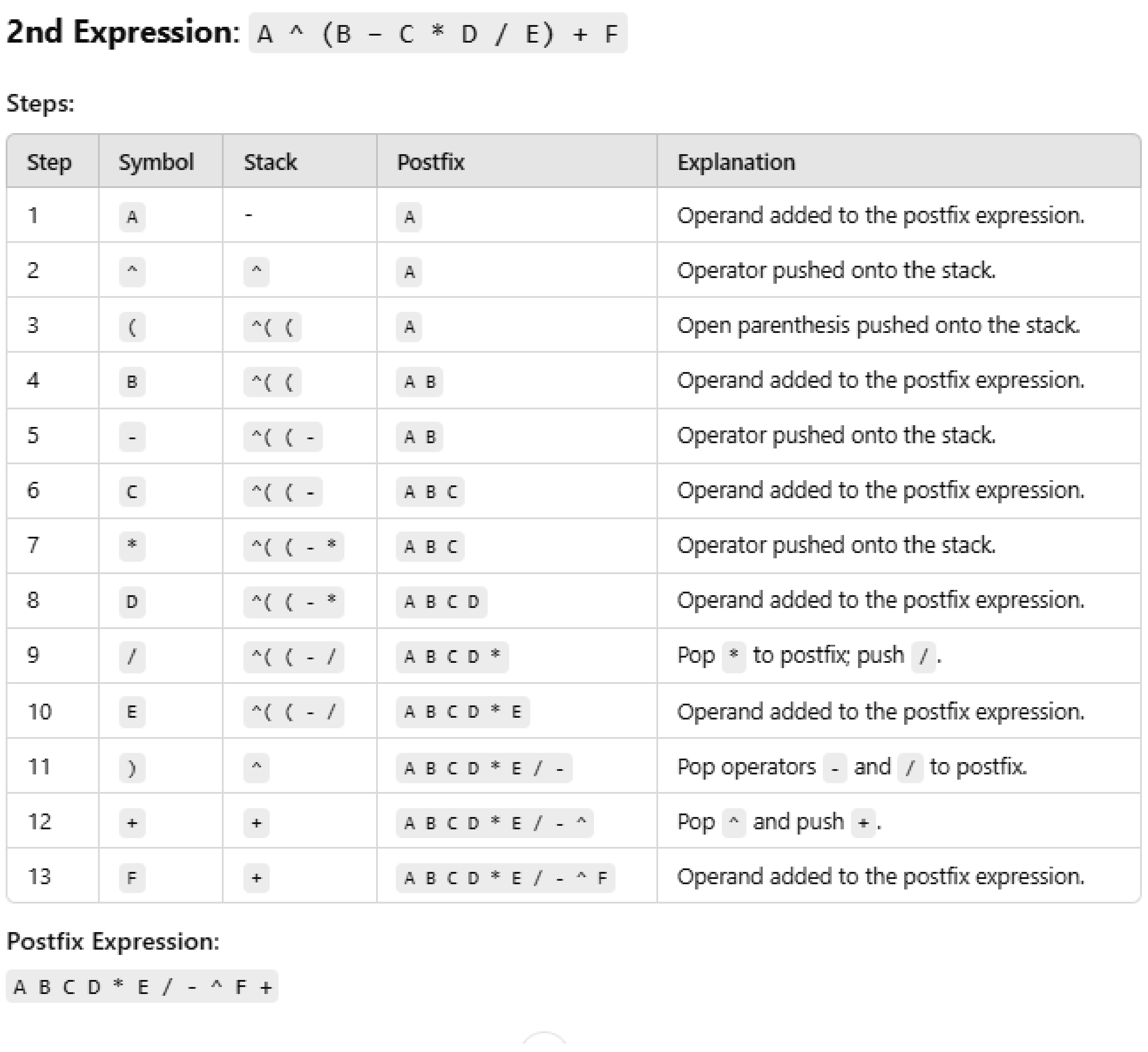
Steps:

Step	Symbol	Stack	Postfix	Explanation
1	`A`	-	`A`	Operand added to the postfix expression.
2	`*`	`*`	`A`	Operator pushed onto the stack.
3	`(`	`*(`	`A`	Open parenthesis pushed onto the stack.
4	`B`	`*(`	`A B`	Operand added to the postfix expression.
5	`*`	`*( *`	`A B`	Operator pushed onto the stack.
6	`C`	`*( *`	`A B C`	Operand added to the postfix expression.
7	`+`	`*( +`	`A B C *`	Pop `*`, add to postfix; push `+`.
8	`D`	`*( +`	`A B C * D`	Operand added to the postfix expression.
9	`*`	`*( + *`	`A B C * D`	Operator pushed onto the stack.
10	`E`	`*( + *`	`A B C * D E`	Operand added to the postfix expression.
11	`)`	`*`	`A B C * D E * +`	Pop `*` and `+` from stack to postfix.
12	`+`	`+`	`A B C * D E * +`	Pop `*`, push `+`.
13	`F`	`+`	`A B C * D E * + F`	Operand added to the postfix expression.

Postfix Expression:

`A B C * D E * + * F +`

2. A ^ (B – C * D /E) + F



2nd Expression: `A ^ (B − C * D / E) + F`

Steps:

Step	Symbol	Stack	Postfix	Explanation
1	`A`	-	`A`	Operand added to the postfix expression.
2	`^`	`^`	`A`	Operator pushed onto the stack.
3	`(`	`^( (`	`A`	Open parenthesis pushed onto the stack.
4	`B`	`^( (`	`A B`	Operand added to the postfix expression.
5	`-`	`^( ( -`	`A B`	Operator pushed onto the stack.
6	`C`	`^( ( -`	`A B C`	Operand added to the postfix expression.
7	`*`	`^( ( - *`	`A B C`	Operator pushed onto the stack.
8	`D`	`^( ( - *`	`A B C D`	Operand added to the postfix expression.
9	`/`	`^( ( - /`	`A B C D *`	Pop `*` to postfix; push `/`.
10	`E`	`^( ( - /`	`A B C D * E`	Operand added to the postfix expression.
11	`)`	`^`	`A B C D * E / -`	Pop operators `-` and `/` to postfix.
12	`+`	`+`	`A B C D * E / - ^`	Pop `^` and push `+`.
13	`F`	`+`	`A B C D * E / - ^ F`	Operand added to the postfix expression.

Postfix Expression:

`A B C D * E / - ^ F +`

14. Convert the following Infix expression into prefix using a table.

(A+B)/(C+D*E)

->

Steps for Conversion

We will use the following rules:

1. Start from the innermost parentheses.
2. Follow the operator precedence and associativity rules.
3. Convert the expression inside parentheses to prefix.
4. Move outward until the entire expression is converted.

Step-by-Step Conversion

Step	Symbol	Prefix	Explanation
1	`A + B`	`+ A B`	Convert the expression inside the left parentheses.
2	`D * E`	`* D E`	Convert the multiplication inside the right parentheses.
3	`C + (* D E)`	`+ C * D E`	Combine `C` and `* D E` as `+ C * D E`.
4	`(A + B) / (C + D * E)`	`/ + A B + C * D E`	Combine the left and right results with `/`.

Final Prefix Expression

`/ + A B + C * D E`

Verification

To ensure correctness, let's evaluate the order of operations:

1. Compute `+ A B` (add `A` and `B`).
2. Compute `* D E` (multiply `D` and `E`).
3. Compute `+ C * D E` (add `C` to the result of `D * E`).
4. Compute `/ + A B + C * D E` (divide the result of `A + B` by the result of `C + D * E`).

The prefix expression is correct and follows the order of operations.

15. Draw BST for the following and traverse inorder, preorder and postorder

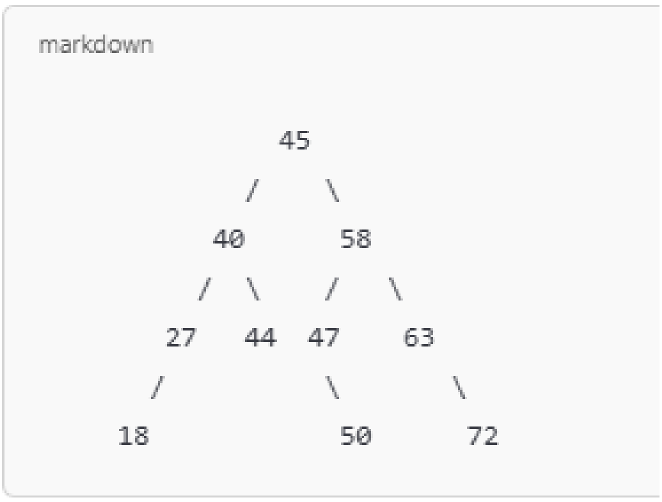
45, 40, 27, 58, 63, 72, 18, 44, 47, 50

->

Steps to Insert Nodes into the BST

1. **Start with 45**: It becomes the root.
2. **Insert 40**: Less than 45 → Goes to the left of 45.
3. **Insert 27**: Less than 45 → Left of 45. Less than 40 → Left of 40.
4. **Insert 58**: Greater than 45 → Right of 45.
5. **Insert 63**: Greater than 45 → Right of 45. Greater than 58 → Right of 58.
6. **Insert 72**: Greater than 45 → Right of 45. Greater than 58 → Right of 58. Greater than 63 → Right of 63.
7. **Insert 18**: Less than 45 → Left of 45. Less than 40 → Left of 40. Less than 27 → Left of 27.
8. **Insert 44**: Less than 45 → Left of 45. Greater than 40 → Right of 40.
9. **Insert 47**: Greater than 45 → Right of 45. Less than 58 → Left of 58. Less than 50 → Left of 50.
10. **Insert 50**: Greater than 45 → Right of 45. Less than 58 → Left of 58. Greater than 47 → Right of 47.

Final BST Diagram



markdown

```
                45
              /    \
            40      58
           /  \    /   \
         27    44 47     63
        /          \       \
      18            50      72
```

Traversals

1. Inorder Traversal (Left, Root, Right):

- Visit the nodes in ascending order.
- Result: `18, 27, 40, 44, 45, 47, 50, 58, 63, 72`

2. Preorder Traversal (Root, Left, Right):

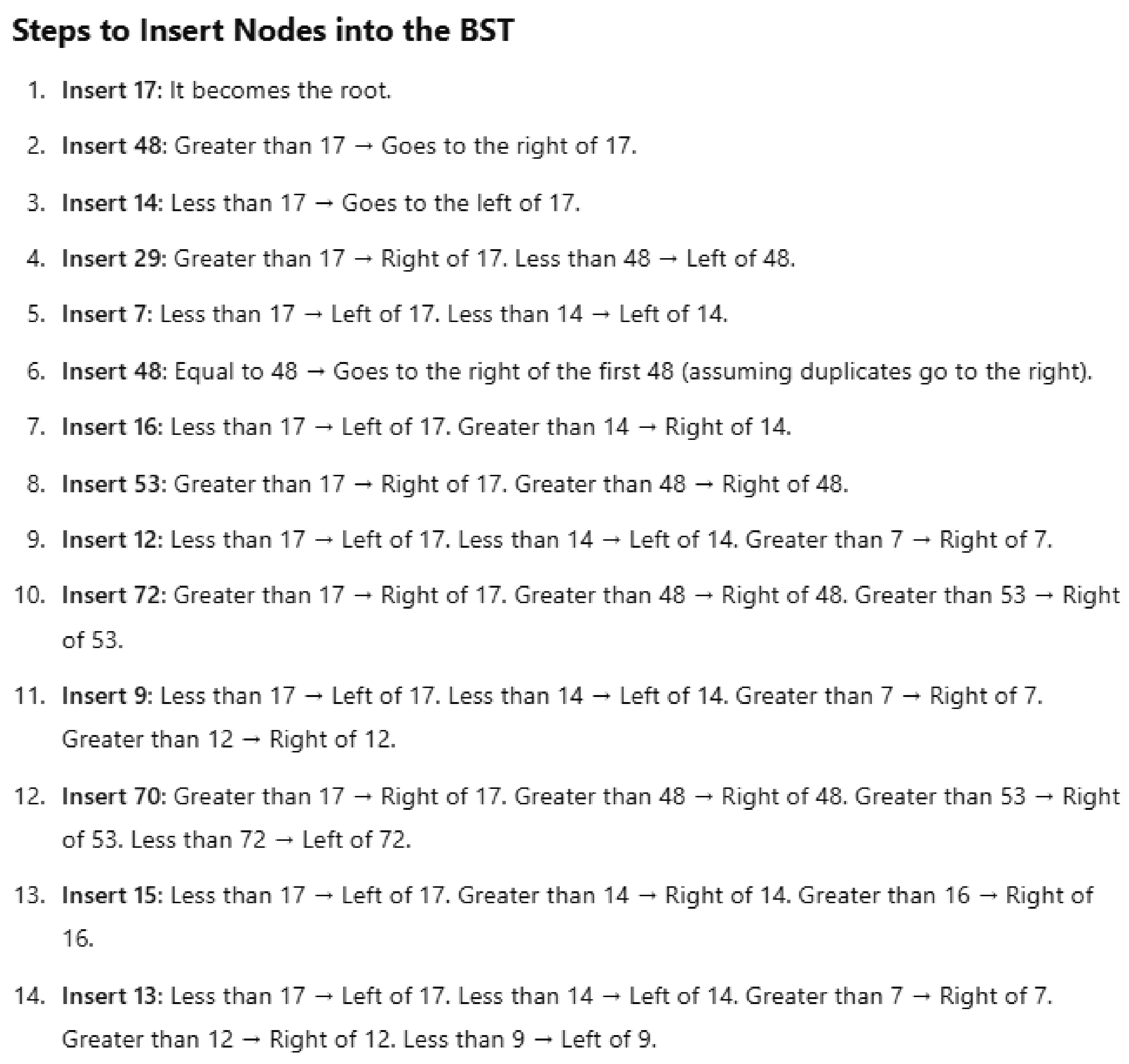
- Visit the root first, then left subtree, and finally the right subtree.
- Result: `45, 40, 27, 18, 44, 58, 47, 50, 63, 72`

3. Postorder Traversal (Left, Right, Root):

- Visit the left subtree first, then the right subtree, and finally the root.
- Result: `18, 27, 44, 40, 50, 47, 72, 63, 58, 45`

16. Draw BST for the and traverse inorder ,preorder and postorder 17,48,14,29,7,48,16,53,12,72,9,70,15,13

->



Steps to Insert Nodes into the BST

1. **Insert 17**: It becomes the root.
2. **Insert 48**: Greater than 17 → Goes to the right of 17.
3. **Insert 14**: Less than 17 → Goes to the left of 17.
4. **Insert 29**: Greater than 17 → Right of 17. Less than 48 → Left of 48.
5. **Insert 7**: Less than 17 → Left of 17. Less than 14 → Left of 14.
6. **Insert 48**: Equal to 48 → Goes to the right of the first 48 (assuming duplicates go to the right).
7. **Insert 16**: Less than 17 → Left of 17. Greater than 14 → Right of 14.
8. **Insert 53**: Greater than 17 → Right of 17. Greater than 48 → Right of 48.
9. **Insert 12**: Less than 17 → Left of 17. Less than 14 → Left of 14. Greater than 7 → Right of 7.
10. **Insert 72**: Greater than 17 → Right of 17. Greater than 48 → Right of 48. Greater than 53 → Right of 53.
11. **Insert 9**: Less than 17 → Left of 17. Less than 14 → Left of 14. Greater than 7 → Right of 7. Greater than 12 → Right of 12.
12. **Insert 70**: Greater than 17 → Right of 17. Greater than 48 → Right of 48. Greater than 53 → Right of 53. Less than 72 → Left of 72.
13. **Insert 15**: Less than 17 → Left of 17. Greater than 14 → Right of 14. Greater than 16 → Right of 16.
14. **Insert 13**: Less than 17 → Left of 17. Less than 14 → Left of 14. Greater than 7 → Right of 7. Greater than 12 → Right of 12. Less than 9 → Left of 9.

Final Bst DIG

```
           17
          /  \
        14    48
       /  \  /  \
      7   16 48  53
       \    \     \
        12   15    72
       /  \          /
      9   13        70
```

Traversals

1. Inorder Traversal (Left, Root, Right):

- Visit the nodes in ascending order.
- Result: `7, 9, 12, 13, 14, 15, 16, 17, 29, 48, 48, 53, 70, 72`

2. Preorder Traversal (Root, Left, Right):

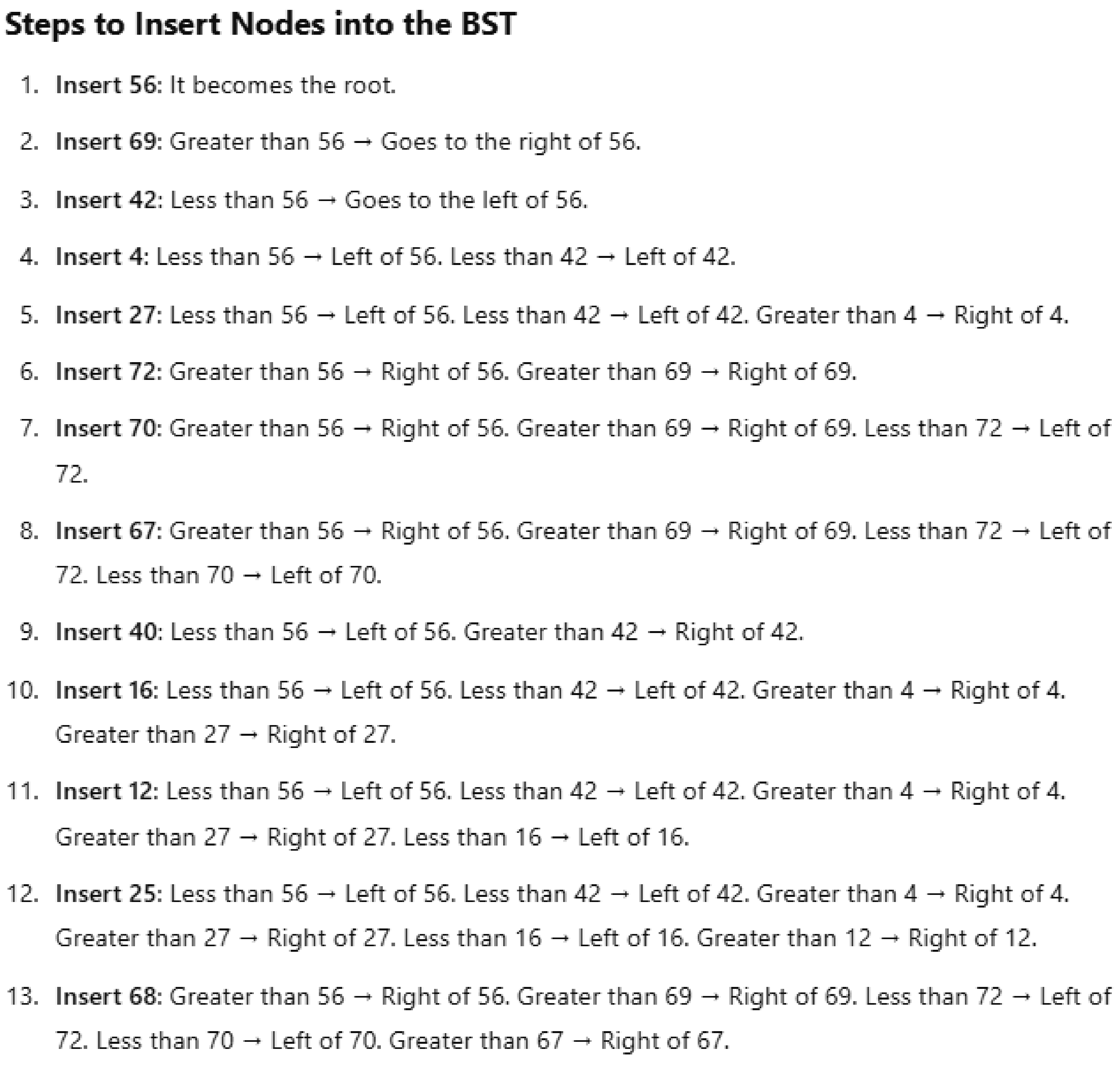
- Visit the root first, then left subtree, and finally the right subtree.
- Result: `17, 14, 7, 12, 9, 13, 16, 15, 48, 48, 53, 72, 70, 29`

3. Postorder Traversal (Left, Right, Root):

- Visit the left subtree first, then the right subtree, and finally the root.
- Result: `9, 13, 12, 7, 15, 16, 14, 29, 70, 72, 53, 48, 48, 17`

17. Draw BST for the and traverse inorder ,preorder and postorder 56,69,42,4,27,72,70,67,40,16,12,25,68

->



Steps to Insert Nodes into the BST

1. **Insert 56**: It becomes the root.
2. **Insert 69**: Greater than 56 → Goes to the right of 56.
3. **Insert 42**: Less than 56 → Goes to the left of 56.
4. **Insert 4**: Less than 56 → Left of 56. Less than 42 → Left of 42.
5. **Insert 27**: Less than 56 → Left of 56. Less than 42 → Left of 42. Greater than 4 → Right of 4.
6. **Insert 72**: Greater than 56 → Right of 56. Greater than 69 → Right of 69.
7. **Insert 70**: Greater than 56 → Right of 56. Greater than 69 → Right of 69. Less than 72 → Left of 72.
8. **Insert 67**: Greater than 56 → Right of 56. Greater than 69 → Right of 69. Less than 72 → Left of 72. Less than 70 → Left of 70.
9. **Insert 40**: Less than 56 → Left of 56. Greater than 42 → Right of 42.
10. **Insert 16**: Less than 56 → Left of 56. Less than 42 → Left of 42. Greater than 4 → Right of 4. Greater than 27 → Right of 27.
11. **Insert 12**: Less than 56 → Left of 56. Less than 42 → Left of 42. Greater than 4 → Right of 4. Greater than 27 → Right of 27. Less than 16 → Left of 16.
12. **Insert 25**: Less than 56 → Left of 56. Less than 42 → Left of 42. Greater than 4 → Right of 4. Greater than 27 → Right of 27. Less than 16 → Left of 16. Greater than 12 → Right of 12.
13. **Insert 68**: Greater than 56 → Right of 56. Greater than 69 → Right of 69. Less than 72 → Left of 72. Less than 70 → Left of 70. Greater than 67 → Right of 67.

FINAL BST DIGRAM

```
        56
      /    \
    42       69
   /  \        \
 4     27       72
  \   /  \     /
  16  25   40  70
 /    \        /
12     4      68
```

Traversals

1. Inorder Traversal (Left, Root, Right):

- Visit the nodes in ascending order.
- Result: `4, 12, 16, 25, 27, 40, 42, 56, 67, 68, 69, 70, 72`

2. Preorder Traversal (Root, Left, Right):

- Visit the root first, then left subtree, and finally the right subtree.
- Result: `56, 42, 4, 27, 16, 12, 25, 40, 69, 67, 70, 72, 68`

3. Postorder Traversal (Left, Right, Root):

- Visit the left subtree first, then the right subtree, and finally the root.
- Result: `4, 12, 25, 16, 40, 27, 42, 68, 70, 67, 72, 69, 56`

18. Draw AVL tree of following data 20,10,30,5,15,25,35,13,17

->

Steps to Insert Nodes into the AVL Tree

1. **Insert 20**: This is the root.
2. **Insert 10**: Less than 20 → Goes to the left of 20.
3. **Insert 30**: Greater than 20 → Goes to the right of 20.
4. **Insert 5**: Less than 20 → Left of 20. Less than 10 → Left of 10.
   - **Balance Check:** The tree is unbalanced, with a balance factor of `2` at node 20 (left-heavy).
   - **Rotation:** Perform a **right rotation** on node 20. After the rotation, 10 becomes the root.
5. **Insert 15**: Greater than 10 → Goes to the right of 10.
6. **Insert 25**: Less than 30 → Left of 30.
7. **Insert 35**: Greater than 30 → Right of 30.
8. **Insert 13**: Less than 15 → Left of 15. Greater than 10 → Right of 10.
9. **Insert 17**: Greater than 15 → Right of 15. Less than 20 → Left of 20.

AVL Tree After Insertions and Rotations

Let's break down the tree after each insertion:

1. **Initial Insertions** (Before balancing):
   - **Insert 20** → Root: 20
   - **Insert 10** → 10 becomes left child of 20
   - **Insert 30** → 30 becomes right child of 20
   - **Insert 5** → 5 becomes left child of 10 (After right rotation, tree becomes balanced)
   - **Insert 15** → 15 becomes right child of 10
   - **Insert 25** → 25 becomes left child of 30
   - **Insert 35** → 35 becomes right child of 30
   - **Insert 13** → 13 becomes left child of 15
   - **Insert 17** → 17 becomes right child of 15

Final Balanced AVL Tree

```
        15
       /    \
    10       30
  /   \    /   \
 5     13  25     35
           /
        20
```

Traversals

1. Inorder Traversal (Left, Root, Right):

- Result: `5, 10, 13, 15, 17, 20, 25, 30, 35`

2. Preorder Traversal (Root, Left, Right):

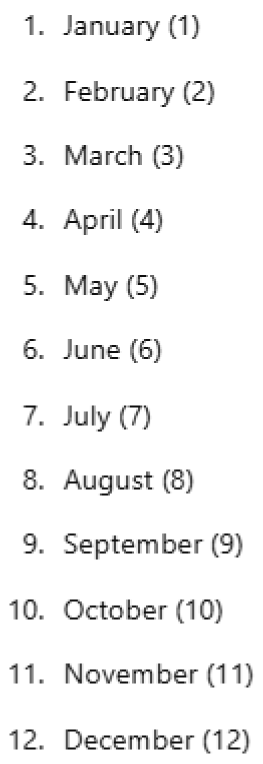
- Result: `15, 10, 5, 13, 17, 30, 25, 20, 35`

3. Postorder Traversal (Left, Right, Root):

- Result: `5, 13, 17, 10, 20, 25, 35, 30, 15`

19. Draw AVL tree for months of a year

->



1. January (1)
2. February (2)
3. March (3)
4. April (4)
5. May (5)
6. June (6)
7. July (7)
8. August (8)
9. September (9)
10. October (10)
11. November (11)
12. December (12)

Insertions and Balancing

1. Insert 1 (January): Root node.
2. Insert 2 (February): Greater than 1 → Goes to the right of 1.
3. Insert 3 (March): Greater than 1 → Right of 1. Greater than 2 → Right of 2.
   - Unbalanced at Node 1 (Right-heavy).
     Balance Factor: -2.
     Rotation: Perform left rotation at node 1.
     - New Root: 2.
4. Insert 4 (April): Greater than 2 → Goes to the right of 3.
5. Insert 5 (May): Greater than 3 → Goes to the right of 4.
   - Unbalanced at Node 3 (Right-heavy).
     Rotation: Perform left rotation at node 3.
     - New Root: 4 for the subtree rooted at 3.
6. Insert 6 (June): Greater than 4 → Goes to the right of 5.
7. Insert 7 (July): Greater than 5 → Right of 6.
   - Unbalanced at Node 5 (Right-heavy).
     Rotation: Perform left rotation at node 5.
     - New Root: 6 for the subtree rooted at 5.
8. Insert 8 (August): Greater than 6 → Goes to the right of 7.
9. Insert 9 (September): Greater than 7 → Goes to the right of 8.
   - Unbalanced at Node 7 (Right-heavy).
     Rotation: Perform left rotation at node 7.
     - New Root: 8 for the subtree rooted at 7.
10. Insert 10 (October): Greater than 8 → Goes to the right of 9.
11. Insert 11 (November): Greater than 9 → Goes to the right of 10.
12. Insert 12 (December): Greater than 10 → Goes to the right of 11.
    - Unbalanced at Node 10 (Right-heavy).
      Rotation: Perform left rotation at node 10.
      - New Root: 11 for the subtree rooted at 10.

FINAL AVL TREE STRUCTURE

```
          6
       /     \
     3         9
   /  \      /   \
  2    4    8      11
 /       \        /  \
1          5      10   12
             \
              7
```

Traversals

1. Inorder Traversal (Left, Root, Right):

- Result: `1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12`
  (Months in ascending order)

2. Preorder Traversal (Root, Left, Right):

- Result: `6, 3, 2, 1, 4, 5, 9, 8, 7, 11, 10, 12`

3. Postorder Traversal (Left, Right, Root):

- Result: `1, 2, 5, 4, 3, 7, 8, 10, 12, 11, 9, 6`

PROGRAMS

20. Write a program to add two polynomials using 1D Arrays

->

```c
#include <stdio.h>
#define MAX 100  // Maximum degree of the polynomial

void addPolynomials(int poly1[], int poly2[], int result[], int degree1, int
degree2) {
   int maxDegree = (degree1 > degree2) ? degree1 : degree2;

   // Initialize result array
   for (int i = 0; i <= maxDegree; i++) {
      result[i] = 0;
   }

   // Add coefficients of the first polynomial
   for (int i = 0; i <= degree1; i++) {
      result[i] += poly1[i];
   }

   // Add coefficients of the second polynomial
   for (int i = 0; i <= degree2; i++) {
      result[i] += poly2[i];
   }
}
void printPolynomial(int poly[], int degree) {
   for (int i = 0; i <= degree; i++) {
      if (poly[i] != 0) {
         printf("%d", poly[i]);
         if (i > 0) {
            printf("x^%d", i);
         }
         if (i < degree) {
            printf(" + ");
         }
      }
   }
   printf("\n");
}
int main() {
   int poly1[MAX] = {0}, poly2[MAX] = {0}, result[MAX];
```

```
    int degree1, degree2;

    printf("Enter the degree of the first polynomial: ");
    scanf("%d", &degree1);

    printf("Enter the coefficients of the first polynomial:\n");
    for (int i = 0; i <= degree1; i++) {
        printf("Coefficient of x^%d: ", i);
        scanf("%d", &poly1[i]);
    }
    printf("Enter the degree of the second polynomial: ");
    scanf("%d", &degree2);
    printf("Enter the coefficients of the second polynomial:\n");
    for (int i = 0; i <= degree2; i++) {
        printf("Coefficient of x^%d: ", i);
        scanf("%d", &poly2[i]);
    }
    addPolynomials(poly1, poly2, result, degree1, degree2);

    printf("Resultant polynomial after addition:\n");
    printPolynomial(result, (degree1 > degree2) ? degree1 : degree2);

    return 0;
}
```

INPUT:

```
Enter the degree of the first polynomial: 2
Enter the coefficients of the first polynomial:
Coefficient of x^0: 3
Coefficient of x^1: 4
Coefficient of x^2: 5

Enter the degree of the second polynomial: 3
Enter the coefficients of the second polynomial:
Coefficient of x^0: 1
Coefficient of x^1: 2
Coefficient of x^2: 3
Coefficient of x^3: 4
```

Output:
Resultant polynomial after addition:
4 + 6x^1 + 8x^2 + 4x^3

21. Write a Program to reverse a string using stack

->

```
#include <stdio.h>
#include <string.h>
#define MAX 100

char stack[MAX];
int top = -1;

void push(char ch) {
   stack[++top] = ch;
}

char pop() {
   return stack[top--];
}

void reverseString(char str[]) {
   int n = strlen(str);

   // Push characters onto the stack
   for (int i = 0; i < n; i++) {
      push(str[i]);
   }

   // Pop characters and overwrite the string
   for (int i = 0; i < n; i++) {
      str[i] = pop();
   }
}

int main() {
   char str[MAX];

   printf("Enter a string: ");
```

```
    scanf("%s", str);

    reverseString(str);

    printf("Reversed string: %s\n", str);

    return 0;
}
```

INPUT
Enter a string: world

OUTPUT
Reversed string: dlrow

22. Write a Program Implementation of a BST create, insert ,countleaf,Countnodes

->

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure of a BST node
typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

// Function to create a new node
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function to insert a value into the BST
Node* insert(Node* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }
    return root;
}

// Function to count the total number of nodes
int countNodes(Node* root) {
    if (root == NULL) {
        return 0;
    }
    return 1 + countNodes(root->left) + countNodes(root->right);
}

// Function to count the number of leaf nodes
int countLeafNodes(Node* root) {
    if (root == NULL) {
        return 0;
    }
    if (root->left == NULL && root->right == NULL) {
        return 1;
    }
    return countLeafNodes(root->left) + countLeafNodes(root->right);
}

// Inorder traversal (optional, to display the tree structure)
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Main function
int main() {
    Node* root = NULL;
```

```
  int choice, value;

  while (1) {
    printf("\n1. Insert\n2. Count Nodes\n3. Count Leaf Nodes\n4. Display
Inorder\n5. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
      case 1:
        printf("Enter value to insert: ");
        scanf("%d", &value);
        root = insert(root, value);
        break;

      case 2:
        printf("Total nodes: %d\n", countNodes(root));
        break;

      case 3:
        printf("Leaf nodes: %d\n", countLeafNodes(root));
        break;

      case 4:
        printf("Inorder Traversal: ");
        inorder(root);
        printf("\n");
        break;

      case 5:
        exit(0);

      default:
        printf("Invalid choice! Try again.\n");
    }
  }
  return 0;
}
```

OUTPUT

```
1. Insert
2. Count Nodes
3. Count Leaf Nodes
4. Display Inorder
5. Exit
Enter your choice: 1
Enter value to insert: 50

Enter your choice: 1
Enter value to insert: 30

Enter your choice: 1
Enter value to insert: 70

Enter your choice: 4
Inorder Traversal: 30 50 70

Enter your choice: 2
Total nodes: 3

Enter your choice: 3
Leaf nodes: 2
```

23. Write a Program to Bubble Sort
->
```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            // Swap arr[j] and arr[j + 1]
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}

void printArray(int arr[], int n) {
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}

int main() {
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("Original array: ");
   printArray(arr, n);

   bubbleSort(arr, n);

   printf("Sorted array: ");
   printArray(arr, n);

   return 0;
}
```

INPUT:-
64 34 25 12 22 11 90
OUTPUT:-
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90

24. Write a Program Linear search on sorted data
->

```
#include <stdio.h>

int linearSearch(int arr[], int n, int key) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == key) {
         return i; // Return index if key is found
      }
      if (arr[i] > key) {
         break; // Stop searching as the data is sorted
      }
   }
   return -1; // Return -1 if key is not found
}

int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int n = sizeof(arr) / sizeof(arr[0]);
   int key;

   printf("Enter the element to search: ");
   scanf("%d", &key);

   int result = linearSearch(arr, n, key);
   if (result != -1) {
      printf("Element %d found at index %d.\n", key, result);
   } else {
      printf("Element %d not found.\n", key);
   }

   return 0;
}
```

Input:-

Enter the element to search: 30

Output:-
Element 30 found at index 2.

25. Write a Program Linear search on unsorted data
->

```
#include <stdio.h>

int linearSearch(int arr[], int n, int key) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == key) {
         return i; // Return index if key is found
      }
   }
   return -1; // Return -1 if key is not found
}

int main() {
   int arr[] = {34, 23, 12, 45, 56};
   int n = sizeof(arr) / sizeof(arr[0]);
   int key;

   printf("Enter the element to search: ");
   scanf("%d", &key);

   int result = linearSearch(arr, n, key);
   if (result != -1) {
      printf("Element %d found at index %d.\n", key, result);
   } else {
      printf("Element %d not found.\n", key);
   }

   return 0;
}
```

INPUT:
Enter the element to search: 45
Output:
Element 45 found at index 3.

26. Write a Program linked list create ,insert,first .

->

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a linked list node
struct Node {
   int data;
   struct Node* next;
};

// Function to insert a new node at the beginning of the list
void insertFirst(struct Node** head, int newData) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = newData;
   newNode->next = *head;
   *head = newNode;
}

// Function to print the linked list
void printList(struct Node* head) {
   struct Node* temp = head;
   while (temp != NULL) {
      printf("%d -> ", temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
}

int main() {
   struct Node* head = NULL;

   insertFirst(&head, 10);
   insertFirst(&head, 20);
   insertFirst(&head, 30);

   printf("Linked List: ");
   printList(head);

   return 0;
}
```

OUTPUT:-

```
Linked List: 30 -> 20 -> 10 -> NULL
```

27. Write a program to multiply two polynomials. (1 dimensional)

->

```c
#include <stdio.h>
void multiplyPolynomials(int A[], int B[], int result[], int n, int m) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         result[i + j] += A[i] * B[j]; // Multiply and accumulate
      }
   }
}
void printPolynomial(int poly[], int n) {
   for (int i = 0; i < n; i++) {
      printf("%d ", poly[i]);
   }
   printf("\n");
}

int main() {
   int A[] = {5, 0, 10, 6}; // Polynomial 5 + 0x + 10x^2 + 6x^3
   int B[] = {1, 2, 4};     // Polynomial 1 + 2x + 4x^2
   int n = sizeof(A) / sizeof(A[0]);
   int m = sizeof(B) / sizeof(B[0]);
   int result[n + m - 1];   // Result will have (n + m - 1) terms

   // Initialize result array to zero
   for (int i = 0; i < n + m - 1; i++) {
      result[i] = 0;
   }

   multiplyPolynomials(A, B, result, n, m);
   printf("Product of the polynomials: ");
   printPolynomial(result, n + m - 1);

   return 0;
}
```

OUTPUT:-
Product of the polynomials: 5 10 34 44 28

28. Write a Program in a circular queue

->
```
#include <stdio.h>
#define MAX 5  // Maximum size of the circular queue

// Define the Circular Queue structure
struct Queue {
   int arr[MAX];
   int front, rear;
};

// Initialize the queue
void initializeQueue(struct Queue* q) {
   q->front = -1;
   q->rear = -1;
}

// Check if the queue is full
int isFull(struct Queue* q) {
   return ((q->rear + 1) % MAX == q->front);
}

// Check if the queue is empty
int isEmpty(struct Queue* q) {
   return (q->front == -1);
}

// Enqueue operation (insert)
void enqueue(struct Queue* q, int value) {
   if (isFull(q)) {
      printf("Queue is full!\n");
      return;
   }
   if (isEmpty(q)) {
      q->front = 0;
   }
   q->rear = (q->rear + 1) % MAX;
   q->arr[q->rear] = value;
   printf("Enqueued: %d\n", value);
```

```
}

// Dequeue operation (delete)
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return -1;
    }
    int value = q->arr[q->front];
    if (q->front == q->rear) {  // Only one element was in the queue
        q->front = q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX;
    }
    return value;
}

// Display the queue elements
void display(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue elements: ");
    int i = q->front;
    while (i != q->rear) {
        printf("%d ", q->arr[i]);
        i = (i + 1) % MAX;
    }
    printf("%d\n", q->arr[q->rear]);
}

int main() {
    struct Queue q;
    initializeQueue(&q);

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);
    enqueue(&q, 50);  // Queue becomes full here
    display(&q);
    dequeue(&q);
    dequeue(&q);
    display(&q);

    enqueue(&q, 60);  // Circular insertion
    display(&q);

    return 0;
}
```

OUTPUT:-

```
Enqueued: 10

Enqueued: 20

Enqueued: 30

Enqueued: 40

Enqueued: 50

Queue elements: 10 20 30 40 50

Dequeued: 10

Dequeued: 20

Queue elements: 30 40 50

Enqueued: 60

Queue elements: 30 40 50 60
```

29. Write a program to Selection Sort

->

```
#include <stdio.h>

void selectionSort(int arr[], int n) {
   for (int i = 0; i < n - 1; i++) {
      int minIndex = i;
      for (int j = i + 1; j < n; j++) {
         if (arr[j] < arr[minIndex]) {
            minIndex = j;  // Find the minimum element
         }
      }
      // Swap the found minimum element with the first element
      int temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
   }
}

void printArray(int arr[], int n) {
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}

int main() {
   int arr[] = {64, 25, 12, 22, 11};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("Original array: ");
   printArray(arr, n);

   selectionSort(arr, n);

   printf("Sorted array: ");
   printArray(arr, n);

   return 0;
}
```

OUTPUT:-

Original array: 64 25 12 22 11

Sorted array: 11 12 22 25 64

30. Write a program to implement Binary Search on Arrays

->

```
#include <stdio.h>

int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        // Check if key is present at mid
        if (arr[mid] == key)
            return mid;

        // If key is greater, ignore left half
        if (arr[mid] < key)
            low = mid + 1;

        // If key is smaller, ignore right half
        else
            high = mid - 1;
    }

    return -1; // Key not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key;

    printf("Enter the element to search: ");
    scanf("%d", &key);
    int result = binarySearch(arr, n, key);
    if (result != -1) {
        printf("Element %d found at index %d.\n", key, result);
    } else {
        printf("Element %d not found.\n", key);
    }

    return 0;
}
```

INPUT:
Enter the element to search: 30

OUTPUT:
Element 30 found at index 2.

INPUT:
Enter the element to search: 15

OUTPUT:
Element 15 not found.

31. Write a program to implement a Dynamic Queue(Use pointer,linked list)
->

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a queue node
struct Node {
    int data;
    struct Node* next;
};

// Define the structure for the queue
struct Queue {
    struct Node* front;
    struct Node* rear;
};

// Function to initialize the queue
void initializeQueue(struct Queue* q) {
    q->front = q->rear = NULL;
}

// Function to check if the queue is empty
int isEmpty(struct Queue* q) {
    return q->front == NULL;
}

// Enqueue operation (insert at the rear)
void enqueue(struct Queue* q, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (q->rear == NULL) {
        q->front = q->rear = newNode;  // If the queue is empty, both front and
rear point to the new node
    } else {
        q->rear->next = newNode;  // Add new node at the rear
        q->rear = newNode;
    }
    printf("Enqueued: %d\n", value);
}

// Dequeue operation (remove from the front)
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return -1;
    }

    struct Node* temp = q->front;
    int value = temp->data;
    q->front = q->front->next;

    if (q->front == NULL) {
        q->rear = NULL;  // If the queue becomes empty, set rear to NULL
    }

    free(temp);
    return value;
}
```

```
// Display the queue elements
void displayQueue(struct Queue* q) {
   if (isEmpty(q)) {
      printf("Queue is empty!\n");
      return;
   }

   struct Node* temp = q->front;
   printf("Queue: ");
   while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->next;
   }
   printf("\n");
}

int main() {
   struct Queue q;
   initializeQueue(&q);

   enqueue(&q, 10);
   enqueue(&q, 20);
   enqueue(&q, 30);
   displayQueue(&q);

   printf("Dequeued: %d\n", dequeue(&q));
   displayQueue(&q);

   printf("Dequeued: %d\n", dequeue(&q));
   displayQueue(&q);

   return 0;
}
```

OUTPUT:

```
Enqueued: 10

Enqueued: 20

Enqueued: 30

Queue: 10 20 30

Dequeued: 10

Queue: 20 30

Dequeued: 20

Queue: 30
```

32. Write a program to implement a dynamic stack(Use pointer,linked list)

->

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a stack node
struct Node {
   int data;
   struct Node* next;
};

// Define the structure for the stack
struct Stack {
   struct Node* top;
};

// Function to initialize the stack
void initializeStack(struct Stack* s) {
   s->top = NULL;
}
```

```
// Check if the stack is empty
int isEmpty(struct Stack* s) {
   return s->top == NULL;
}

// Push operation (insert at the top)
void push(struct Stack* s, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = s->top;
   s->top = newNode;  // The new node becomes the top of the stack
   printf("Pushed: %d\n", value);
}

// Pop operation (remove from the top)
int pop(struct Stack* s) {
   if (isEmpty(s)) {
      printf("Stack is empty!\n");
      return -1;
   }

   struct Node* temp = s->top;
   int value = temp->data;
   s->top = s->top->next;

   free(temp);  // Free the memory of the popped node
   return value;
}

// Display the stack elements
void displayStack(struct Stack* s) {
   if (isEmpty(s)) {
      printf("Stack is empty!\n");
      return;
   }

   struct Node* temp = s->top;
   printf("Stack: ");
   while (temp != NULL) {
      printf("%d ", temp->data);
      temp = temp->next;
```

```
    }
    printf("\n");
}

int main() {
    struct Stack s;
    initializeStack(&s);

    push(&s, 10);
    push(&s, 20);
    push(&s, 30);
    displayStack(&s);

    printf("Popped: %d\n", pop(&s));
    displayStack(&s);

    printf("Popped: %d\n", pop(&s));
    displayStack(&s);

    return 0;
}
```

OUTPUT:

```
Pushed: 10

Pushed: 20

Pushed: 30

Stack: 30 20 10

Popped: 30

Stack: 20 10

Popped: 20

Stack: 10
```

33. Write a function for the transpose matrix (full program)

->

```
#include <stdio.h>

void transposeMatrix(int rows, int cols, int matrix[rows][cols], int
transpose[cols][rows]) {
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         transpose[j][i] = matrix[i][j];
      }
   }
}

void printMatrix(int rows, int cols, int matrix[rows][cols]) {
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
   }
}

int main() {
   int rows, cols;

   printf("Enter the number of rows and columns of the matrix: ");
   scanf("%d %d", &rows, &cols);

   int matrix[rows][cols], transpose[cols][rows];

   printf("Enter the elements of the matrix:\n");
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         printf("Element [%d][%d]: ", i, j);
         scanf("%d", &matrix[i][j]);
      }
   }

   printf("\nOriginal Matrix:\n");
   printMatrix(rows, cols, matrix);
```

```
    transposeMatrix(rows, cols, matrix, transpose);

    printf("\nTranspose Matrix:\n");
    printMatrix(cols, rows, transpose);

    return 0;
}
```

Input

```
Enter the number of rows and columns of the matrix: 2 3

Enter the elements of the matrix:

Element [0][0]: 1

Element [0][1]: 2

Element [0][2]: 3

Element [1][0]: 4

Element [1][1]: 5

Element [1][2]: 6
```

OUTPUT

```
Original Matrix:

1 2 3

4 5 6


Transpose Matrix:

1 4

2 5

3 6
```

34. Write a function to insert and delete elements in a circular queue.
->

```
#include <stdio.h>
#define MAX 5 // Maximum size of the circular queue

typedef struct {
   int items[MAX];
   int front, rear;
} CircularQueue;

// Initialize the queue
void initializeQueue(CircularQueue* q) {
   q->front = -1;
   q->rear = -1;
}

// Check if the queue is full
int isFull(CircularQueue* q) {
   return ((q->rear + 1) % MAX == q->front);
}

// Check if the queue is empty
int isEmpty(CircularQueue* q) {
   return (q->front == -1);
}

// Insert an element into the circular queue
void enqueue(CircularQueue* q, int value) {
   if (isFull(q)) {
      printf("Queue is full! Cannot enqueue %d.\n", value);
      return;
   }

   if (isEmpty(q)) { // First element
      q->front = 0;
   }

   q->rear = (q->rear + 1) % MAX;
```

```c
    q->items[q->rear] = value;
    printf("Enqueued: %d\n", value);
}

// Delete an element from the circular queue
int dequeue(CircularQueue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty! Cannot dequeue.\n");
        return -1;
    }

    int value = q->items[q->front];
    if (q->front == q->rear) { // Only one element left
        q->front = q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX;
    }

    printf("Dequeued: %d\n", value);
    return value;
}

// Display the elements in the circular queue
void display(CircularQueue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }

    printf("Queue elements: ");
    int i = q->front;
    while (1) {
        printf("%d ", q->items[i]);
        if (i == q->rear) break;
        i = (i + 1) % MAX;
    }
    printf("\n");
}

int main() {
    CircularQueue q;
    initializeQueue(&q);
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    enqueue(&q, 40);
    enqueue(&q, 50); // Queue becomes full here
    display(&q);

    dequeue(&q);
    dequeue(&q);
    display(&q);

    enqueue(&q, 60);
    enqueue(&q, 70); // Circular insertion
    display(&q);

    return 0;
}
```

OUTPUT

```
Enqueued: 10
Enqueued: 20
Enqueued: 30
Enqueued: 40
Enqueued: 50
Queue elements:
10 20 30 40 50
Dequeued: 10
Dequeued: 20
Queue elements:
30 40 50
Enqueued: 60
Enqueued: 70
Queue elements:
30 40 50 60
```

35. Write a Function Preorder,Postorder,Inorder Traversal

->

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a binary tree node
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};

// Function to create a new node
struct Node* newNode(int data) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->data = data;
   node->left = node->right = NULL;
   return node;
}

// Preorder Traversal (Root, Left, Right)
void preorder(struct Node* root) {
   if (root != NULL) {
      printf("%d ", root->data);
      preorder(root->left);
      preorder(root->right);
   }
}

// Inorder Traversal (Left, Root, Right)
void inorder(struct Node* root) {
   if (root != NULL) {
      inorder(root->left);
      printf("%d ", root->data);
      inorder(root->right);
   }
}

// Postorder Traversal (Left, Right, Root)
void postorder(struct Node* root) {
   if (root != NULL) {
      postorder(root->left);
      postorder(root->right);
      printf("%d ", root->data);
   }
}

int main() {
   // Creating a simple binary tree
   struct Node* root = newNode(1);
   root->left = newNode(2);
   root->right = newNode(3);
   root->left->left = newNode(4);
   root->left->right = newNode(5);
   root->right->left = newNode(6);
   root->right->right = newNode(7);

   // Print tree traversals
   printf("Preorder Traversal: ");
   preorder(root);
   printf("\n");

   printf("Inorder Traversal: ");
   inorder(root);
   printf("\n");

   printf("Postorder Traversal: ");
   postorder(root);
   printf("\n");

   return 0;
}
```

OUTPUT:

```
Preorder Traversal: 1 2 4 5 3 6 7

Inorder Traversal: 4 2 5 1 6 3 7

Postorder Traversal: 4 5 2 6 7 3 1
```

TREE STRUCTURE:

```
        1
      /   \
     2     3
    / \   / \
   4   5 6   7
```

36. write a function to print the minimum value from a BS

->

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a BST node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Function to find the minimum value in the BST
int findMin(struct Node* root) {
    struct Node* current = root;
    // Keep traversing the left child until we reach the leftmost node
    while (current != NULL && current->left != NULL) {
        current = current->left;
    }
    return current ? current->data : -1;  // Return the minimum value
}

int main() {
    // Creating a simple BST
    struct Node* root = newNode(50);
    root->left = newNode(30);
    root->right = newNode(70);
    root->left->left = newNode(20);
    root->left->right = newNode(40);
    root->right->left = newNode(60);
    root->right->right = newNode(80);

    // Print the minimum value in the BST
    printf("The minimum value in the BST is: %d\n", findMin(root));

    return 0;
}
```

OUTPUT;
The minimum value in the BST is: 20

TREE STRUCTURE:

```
         50
        /   \
      30      70
     /  \    /  \
   20    40 60   80
```

37. Solve :- insert a node at the start of a DLL

->

```
    // Function to insert a node at the beginning of the doubly linked list
struct Node* insertAtBeginning(struct Node* head, int value)
    {
    // Step 1: Create a new node with the given value
     struct Node* newNode = createNode(value);
     // Step 2: If the list is not empty, update the current head's previous
    pointer
    if (head != NULL)
    {
    head->prev = newNode;
     }
    // Step 3: Make the new node's next pointer point to the current head
     newNode->next = head;
    // Step 4: Update the head to point to the new node
     head = newNode;
     // Return the new head of the list
    return head;
    }
```

38. Reverse print a DLL without using recursion

->

```
  d_rdisplay( struct dnode *q )
                    {
                    Struct dnode *temp=*q;

while ( temp->next != NULL )

//checks if the current node (temp) has a next node (i.e., temp->next is not
NULL).

temp = temp->next;

//temp = temp->next;: This moves the pointer temp to the next node in the list.

                         while(temp!=NULL)
                         {
                              printf ( "%d ", temp -> data ) ;
                              temp = temp -> next ;
                         }
                    }
```

The code loops through the linked list, printing the data of each node, and moves to the next node until it reaches the end (when temp is NULL).

39. Write a function to sort a DLL

->

```
sort(struct node **head)
//pointer to the pointer of the head node of the linked list.
 {
 struct node *n, *c;
//declares two pointers to nodes in a linked list.
 int x;
for (c = *head; c-> next ! = NULL, c = c->next)

                    //  c = *head: Start at the first node of the list.

                     // c->next != NULL: Check if the current node has a next node;
                    continue if true.

                    //  c = c->next: Move to the next node in the list.
                     for(n = c->next; n!=NULL; n=n->next)
                     {
                     //compare (swap logic)
                       If(c->data > n->data)
                     {
                     X = c->data;
                     c->data = n->data;
                     n->data = x;
                     }
                     }
                      }
```

40. What is difference between stack and queue

->

Feature	Stack	Queue
Order of Access	Last In, First Out (LIFO)	First In, First Out (FIFO)
Main Operations	Push, Pop, Peek	Enqueue, Dequeue, Peek
Element Removal	From the top of the stack	From the front of the queue
Typical Use	Function calls, expression evaluation, undo operations	Task scheduling, print queues, BFS
Structure	Operations on one end (top)	Operations on both ends (front and rear)

5. Visual Representation:

- **Stack:**

```css
Top -> [ 10 ] <- Bottom
       [ 20 ]
       [ 30 ]
```

Push 10, then 20, then 30. Pop removes 30, then 20, then 10.

- **Queue:**

```css
Front -> [ 10 ] [ 20 ] [ 30 ] <- Rear
```

Enqueue 10, then 20, then 30. Dequeue removes 10, then 20, then 30.