DSC IA3

M3Q1) Write functions for Circular Linked List: a) Insertfront(), b) Insertend(), c) Del_front(), d) Del_End()

a) `Insertfront()` – Insertion at the Front End

Logic:

- Allocate memory for a new node.
- If the list is empty:
  - Point `newnode→next` to itself.
  - Set `head` to `newnode`.
- Else:
  - Traverse to the last node.
  - Make last node's `next` point to `newnode`.
  - Set `newnode→next` to head.
  - Update `head` to `newnode`.

Code:

```
void insertfront(int item) {
    NODE *newnode, *temp;
    newnode = (NODE *)malloc(sizeof(NODE));
    newnode→info = item;

    if (head == NULL) {
        newnode→next = newnode;
        head = newnode;
    } else {
        temp = head;
        while (temp→next != head)
            temp = temp→next;
        temp→next = newnode;
```

```
        newnode→next = head;
        head = newnode;
    }
}
```

b) `Insertend()` – Insertion at the Rear End

Logic:

- Allocate memory for new node.
- If the list is empty:
  - Set `newnode→next` to point to itself and `head = newnode`.
- Else:
  - Traverse to last node.
  - Insert `newnode` after last node and point it to `head`.

Code:

```
void insertend(int item) {
    NODE *newnode, *temp;
    newnode = (NODE *)malloc(sizeof(NODE));
    newnode→info = item;

    if (head == NULL) {
        newnode→next = newnode;
        head = newnode;
    } else {
        temp = head;
        while (temp→next != head)
            temp = temp→next;
        temp→next = newnode;
        newnode→next = head;
    }
}
```

c) `Del_front()` – Deletion from Front End

Logic:

- If the list is empty: print underflow.
- If only one node: free and set `head = NULL`.
- Else:
  - Traverse to last node.
  - Make last node point to second node.
  - Free head node and update head.

Code:

```
void del_front() {
    NODE *temp, *last;
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    if (head->next == head) {
        free(head);
        head = NULL;
        return;
    }
    last = head;
    while (last->next != head)
        last = last->next;
    temp = head;
    head = head->next;
    last->next = head;
    free(temp);
}
```

d) `Del_end()` – Deletion from Rear End

Logic:

- If list is empty: print underflow.
- If only one node: free and set `head = NULL`.
- Else:

- Traverse to the second last node.
- Update its `next` to head.
- Free last node.

Code:

```
void del_end() {
    NODE *temp, *prev;
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    if (head->next == head) {
        free(head);
        head = NULL;
        return;
    }
    temp = head;
    while (temp->next != head) {
        prev = temp;
        temp = temp->next;
    }
    prev->next = head;
    free(temp);
}
```

M3Q2) Write a C function to perform the following: 1) Reversing a single linked list, 2) Concatenating single linked list, 3) Finding the length of the list

1) Reversing a Singly Linked List

Logic:

- Traverse the list using three pointers: `prev`, `curr`, and `next`.
- At each step, reverse the `next` pointer.
- Finally, update `head` to `prev`.

Code:

```
void reverse() {
    NODE *prev = NULL, *curr = head, *next;
    while (curr != NULL) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    head = prev;
}
```

2) Concatenating Two Singly Linked Lists

Logic:

- Traverse to the end of the first list.
- Set the `next` of last node in list1 to point to the head of list2.

Code:

```
NODE* concatenate(NODE *head1, NODE *head2) {
    NODE *temp = head1;
    if (head1 == NULL) return head2;
    if (head2 == NULL) return head1;

    while (temp->next != NULL)
        temp = temp->next;

    temp->next = head2;
    return head1;
}
```

3) Finding the Length of the List

Logic:

- Initialize a counter.
- Traverse the list and increment the counter until the end is reached.

Code:

```
int length(NODE *head) {
    int count = 0;
    NODE *temp = head;
    while (temp != NULL) {
        count++;
        temp = temp→next;
    }
    return count;
}
```

M4Q1) Define the following: 1) Binary tree 2) Complete BT, 3) Almost Complete BT, 4) Binary Search Tree, 5) Depth of a tree, 6) Siblings of a tree

1) Binary Tree

A **Binary Tree** is a finite set of elements that is either empty or consists of a root and two disjoint binary trees called the left and right subtrees.

Each node can have **at most two children**.

2) Complete Binary Tree

A **Complete Binary Tree (CBT)** is a binary tree in which:

- All levels are completely filled **except possibly the last**.
- The last level has all keys as **left as possible**.

3) Almost Complete Binary Tree

An **Almost Complete Binary Tree** is another name sometimes used for a **Complete Binary Tree**, particularly emphasizing that all levels are filled except the last, which is filled from left to right.

4) Binary Search Tree (BST)

A **Binary Search Tree** is a binary tree in which:

- The **left subtree** contains nodes with values **less than** the root.

- The **right subtree** contains nodes with values **greater than** the root.
- Both left and right subtrees are also BSTs.

5) Depth of a Tree

The **depth (or height)** of a tree is the number of nodes on the longest path from the **root node** to a **leaf node**.

6) Siblings in a Tree

Siblings are nodes that **share the same parent**.

For example, if two nodes have the same immediate ancestor (parent), they are siblings.

M4Q2) What are the different ways of representing a tree? Explain with examples.

1) Sequential Representation (Using Arrays)

- A tree is stored in an array based on the index positions of its nodes.
- **For a node at index *i***:
  - **Left child** is at `2*i`
  - **Right child** is at `2*i + 1`
  - **Parent** is at `i/2` (if not root)
- This is mainly used for **complete binary trees**, such as heaps.

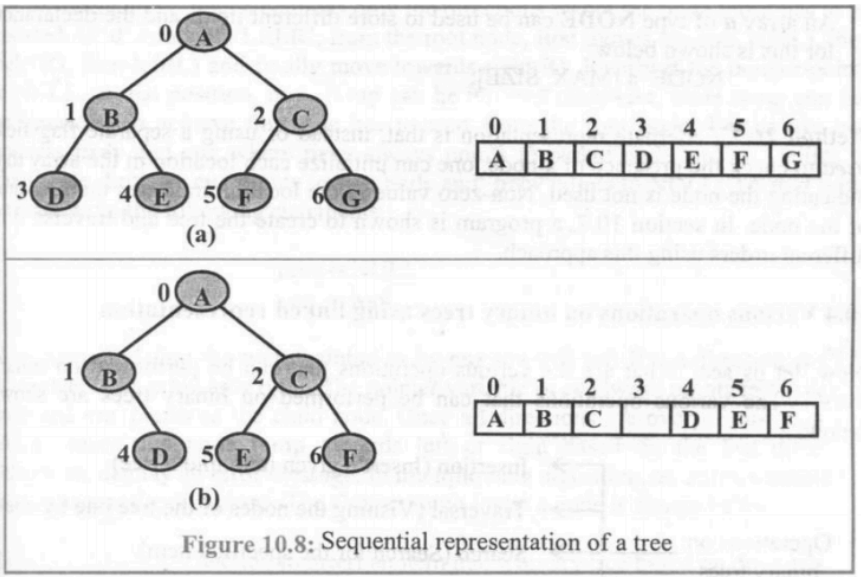


Figure 10.8: Sequential representation of a tree

Advantages: Simple, efficient for complete trees

Disadvantages: Wastes space for sparse trees

2) Linked Representation (Using Structures and Pointers)

- Each node is represented as a structure with:
  - **Data field**
  - **Pointer to left child**
  - **Pointer to right child**

```
struct TreeNode {
    int data;
    struct TreeNode *left, *right;
};
```

- Root is accessed via a pointer (e.g., `TreeNode *root`)
- Used widely for **general trees**, **binary trees**, **BSTs**, etc.

Advantages: Efficient use of memory, works for all tree types

Disadvantages: More complex than arrays

M4Q3) Write C functions for the following tree traversals: a) Inorder, b) Preorder, c) Postorder

a) Inorder Traversal (Left → Root → Right)

Logic:

- Recursively visit the **left subtree**
- Visit the **root**
- Recursively visit the **right subtree**

Code:

```
void inorder(struct TreeNode *root) {
    if (root != NULL) {
        inorder(root→left);
        printf("%d ", root→data);
        inorder(root→right);
    }
}
```

b) Preorder Traversal (Root → Left → Right)

Logic:

- Visit the **root**
- Recursively visit the **left subtree**
- Recursively visit the **right subtree**

Code:

```
void preorder(struct TreeNode *root) {
    if (root != NULL) {
        printf("%d ", root→data);
        preorder(root→left);
        preorder(root→right);
    }
}
```

c) Postorder Traversal (Left → Right → Root)

Logic:

- Recursively visit the **left subtree**

- Recursively visit the **right subtree**
- Visit the **root**

Code:

```
void postorder(struct TreeNode *root) {
    if (root != NULL) {
        postorder(root→left);
        postorder(root→right);
        printf("%d ", root→data);
    }
}
```

M4Q4) Construct a binary trees from the given preorder and inorder sequence: i) Preorder: A B D G C E H I F, ii) Inorder: D G B A H E I C F

M4Q5) Construct an Expression tree for the expression A / B + C * D + E.

Step 1: Understand Operator Precedence

The standard **precedence of operators** is:

```
1) * and / (Left to Right)
2) + and - (Left to Right)
```

So, break down the expression as:

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

Step 2: Build the Tree from Bottom Up

We'll use a **bottom-up recursive approach**:

Step A: Lowest-level operations

- A / B → `/` is the root with A and B as children

- C * D → is the root with C and D as children

Step B: Combine subtrees with `+`

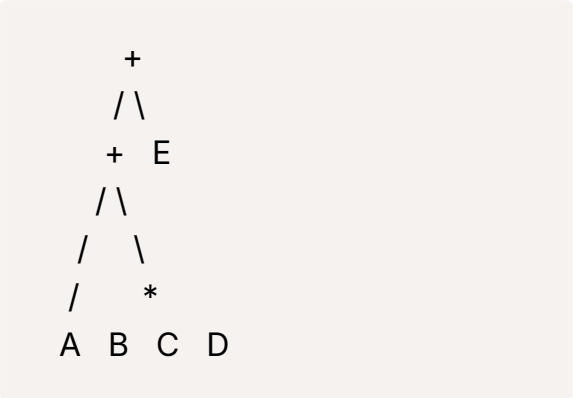
- First +: Left = (A / B), Right = (C * D)

  → `+` becomes new root

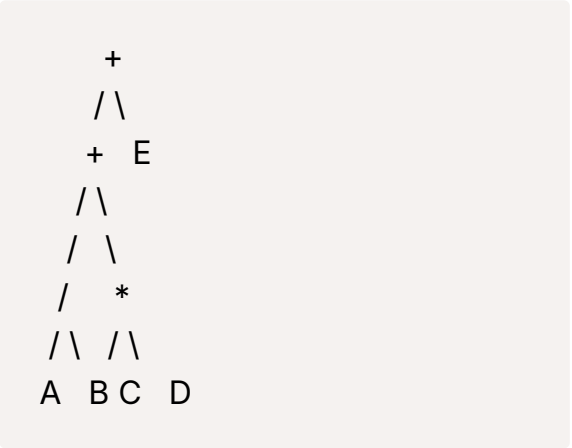
Step C: Add E

- Second +: Left = ((A / B) + (C * D)), Right = E

Final Expression Tree:



This tree corresponds to:



Infix / Postfix / Prefix (for verification)

- Infix: `((A / B) + (C * D)) + E`
- Postfix: `A B / C D * + E +`
- Prefix: `+ + / A B * C D E`

M4Q6) Prove that max no of nodes in a BT of depth K is 2^k-1

In a **Binary Tree (BT)**, each node can have **at most two children**.

The **depth** or **height** of the tree is defined as the **number of levels**, and we start counting levels from **1 (not 0)** as per the textbook convention.

Let's prove that the **maximum number of nodes** in a binary tree of **depth *k*** is:

$$\text{Max Nodes} = 2^k - 1$$

Proof:

Let the depth of the binary tree be k.

Let's count the number of nodes at each level:

- At level 1 (root): $2^{1-1} = 2^0 = 1$ node
- At level 2: $2^{2-1} = 2^1 = 2$ nodes
- At level 3: $2^{3-1} = 2^2 = 4$ nodes
- ...
- At level k: 2^{k-1} nodes

So, the total number of nodes in the tree is:

$1+2 + 4 + \cdots + 2^{k-1}$

This is a **geometric progression (G.P.)** with:

- First term $a = 1$
- Common ratio $r = 2$
- Number of terms = k

The sum of the first k terms of a G.P. is:

$S_k = a \cdot \frac{r^k - 1}{r - 1} = 1 \cdot \frac{2^k - 1}{2 - 1} = 2^k - 1$

Conclusion:

The **maximum number of nodes** in a binary tree of depth k is:

$\boxed{2^k - 1}$

M4Q7) Max no. of nodes on level i of a BT is 2^i-1, given that i>=1 (or) 2^i given that i>=0.

We are to prove:

- **If levels are counted starting from 1** (i.e., root is at level 1), then

  $\text{Max nodes at level } i = 2^{i-1}, \quad \text{for } i \geq 1$

- **If levels are counted starting from 0** (i.e., root is at level 0), then

  $\text{Max nodes at level } i = 2^{i}, \quad \text{for } i \geq 0$

Explanation:

In a **binary tree**, each node can have **at most 2 children**. This gives rise to exponential growth of nodes per level in a **complete binary tree**.

- At **level 1** (or 0-indexed level 0), there is:
  - $2^{1-1} = 2^0 = 1$ node (root)
- At **level 2** (or level 1):
  - $2^{2-1} = 2^1 = 2$ nodes
- At **level 3** (or level 2):
  - $2^{3-1} = 2^2 = 4$ nodes
- ...
- At **level *i***:
  - 2^{i-1} nodes (if levels start from 1)
  - 2^i nodes (if levels start from 0)

This is because **each node at a level can have 2 children**, so the number of nodes doubles from the previous level.

Therefore:

- **Max nodes at level ii (1-based index)**:

  $\boxed{2^{i-1}}, \quad \text{for } i \geq 1$
- **Max nodes at level ii (0-based index)**:

  $\boxed{2^i}, \quad \text{for } i \geq 0$

M4Q8) Prove that no of leaf nodes = no of nodes of degree-2 (or) for any nonempty Binary Tree T, if n0 the number of leaf nodes and N2 no of nodes of degree 2 then N0 = N2 +1.

Let us consider a **non-empty Binary Tree T** with:

- N: Total number of nodes
- N_0: Number of **leaf nodes** (nodes with 0 children)

- N_1: Number of nodes with **1 child**
- N_2: Number of nodes with **2 children**

Then, the **total number of nodes** is:

$N = N_0 + N_1 + N_2 \quad (1)$

In a Binary Tree with N nodes:

Each node (except the root) has **exactly one incoming edge** (from its parent).

So the total number of edges E in a tree is:

$E = N - 1$

Also, each **node of degree 1 contributes 1 edge**, and each **node of degree 2 contributes 2 edges**.

Hence, total number of edges is:

$E = N_1 + 2N_2 \quad (2)$

From (1) and (2), equating total number of edges:

$N - 1 = N_1 + 2N_2$

Substitute $N = N_0 + N_1 + N_2$ into this:

$(N_0 + N_1 + N_2) - 1 = N_1 + 2N_2$

Simplify:

$N_0 + N_1 + N_2 - 1 = N_1 + 2N_2$

Subtract N_1 from both sides:

$N_0 + N_2 - 1 = 2N_2$

Subtract N2N_2 from both sides:

$N_0 - 1 = N_2$

Hence, proved:

$\boxed{N_0 = N_2 + 1}$

Conclusion:

In any non-empty binary tree:

$\boxed{\text{Number of leaf nodes} = \text{Number of nodes with degree } 2 + 1}$

That is:

$$\boxed{N_0 = N_2 + 1}$$

M4Q9) Construct BST for the following: 22, 28,20,25,15,18,10,14

We will insert each element one by one into the Binary Search Tree following the **BST rule**:

> For each node:
>
> - Left child < Node
> - Right child > Node

Step-by-step Insertion:

1. **Insert 22** → Becomes the **root** of the BST
2. **Insert 28** → 28 > 22 → goes to the **right** of 22
3. **Insert 20** → 20 < 22 → goes to the **left** of 22
4. **Insert 25** → 25 > 22 → right of 22 → 25 < 28 → goes to **left of 28**
5. **Insert 15** → 15 < 22 → left of 22 → 15 < 20 → goes to **left of 20**
6. **Insert 18** → 18 < 22 → left of 22 → 18 < 20 → left of 20 → 18 > 15 → goes to **right of 15**
7. **Insert 10** → 10 < 22 → left of 22 → 10 < 20 → left of 20 → 10 < 15 → goes to **left of 15**
8. **Insert 14** → 14 < 22 → left of 22 → 14 < 20 → left of 20 → 14 < 15 → left of 15 → 14 > 10 → goes to **right of 10**

Final BST Structure:

```
      22
    /   \
  20     28
 /      /
15     25
```

```
 / \
10   18
 \
 14
```

M4Q10) Write recursive functions for the following operations on BST: 1) Insert_key(), 2) Delete_key(), 3) Search_key()

Below are the recursive functions in C for BST operations using a typical `struct node`:

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

1. Insert_key():

```
struct node* Insert_key(struct node* root, int key) {
    if (root == NULL) {
        // Allocate new node if tree is empty or reached null
        struct node* newNode = (struct node*) malloc(sizeof(struct node));
        newNode->data = key;
        newNode->left = newNode->right = NULL;
        return newNode;
    }

    if (key < root->data)
        root->left = Insert_key(root->left, key);
    else if (key > root->data)
        root->right = Insert_key(root->right, key);

    return root; // Return unchanged root pointer
}
```

2. Search_key():

```
struct node* Search_key(struct node* root, int key) {
    if (root == NULL || root->data == key)
        return root;

    if (key < root->data)
        return Search_key(root->left, key);
    else
        return Search_key(root->right, key);
}
```

3. Delete_key():

```
struct node* FindMin(struct node* node) {
    while (node->left != NULL)
        node = node->left;
    return node;
}

struct node* Delete_key(struct node* root, int key) {
    if (root == NULL)
        return root;

    if (key < root->data)
        root->left = Delete_key(root->left, key);
    else if (key > root->data)
        root->right = Delete_key(root->right, key);
    else {
        // Node found
        if (root->left == NULL) {
            struct node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
```

```
            return temp;
        }

        // Node with two children
        struct node* temp = FindMin(root->right);
        root->data = temp->data;
        root->right = Delete_key(root->right, temp->data);
    }
    return root;
}
```

M4Q11) Write C functions to perform the following operations on BST: 1) Count the number of nodes, 2) Find the largest and smallest element, 3) Count and display the leaf nodes, 4) Count and display the non-leaf nodes

Assuming the node structure:

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

1. Count the Number of Nodes

```
int countNodes(struct node* root) {
    if (root == NULL)
        return 0;
    return 1 + countNodes(root->left) + countNodes(root->right);
}
```

2. Find the Largest and Smallest Element

Find Smallest (Leftmost node):

```
int findMin(struct node* root) {
    if (root == NULL) {
        printf("Tree is empty\n");
        return -1;
    }
    while (root->left != NULL)
        root = root->left;
    return root->data;
}
```

Find Largest (Rightmost node):

```
int findMax(struct node* root) {
    if (root == NULL) {
        printf("Tree is empty\n");
        return -1;
    }
    while (root->right != NULL)
        root = root->right;
    return root->data;
}
```

3. Count and Display Leaf Nodes

```
int countLeafNodes(struct node* root) {
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL) {
        printf("%d ", root->data);
        return 1;
    }
    return countLeafNodes(root->left) + countLeafNodes(root->right);
}
```

4. Count and Display Non-Leaf Nodes

```
int countNonLeafNodes(struct node* root) {
    if (root == NULL || (root->left == NULL && root->right == NULL))
        return 0;

    printf("%d ", root->data);
    return 1 + countNonLeafNodes(root->left) + countNonLeafNodes(root->ri
ght);
}
```

M4Q12) Construct a binary tree for the following data: 23, 67, 100, 2, 11, and 56,90,34,99. Perform all traversals of the constructed binary tree.

Step 1: Construct a Binary Search Tree (BST)

We insert the elements one by one, following the BST rule:

> Left child < Node < Right child

Insert Order & Placement:

- **23** → root
- **67** → 67 > 23 → right of 23
- **100** → 100 > 23 → right of 23 → 100 > 67 → right of 67
- **2** → 2 < 23 → left of 23
- **11** → 11 < 23 → left of 23 → 11 > 2 → right of 2
- **56** → 56 > 23 → right of 23 → 56 < 67 → left of 67
- **90** → right of 23 → right of 67 → 90 < 100 → left of 100
- **34** → right of 23 → <67 → <56 → left of 56
- **99** → right of 23 → right of 67 → right of 100 → <100 → <90 → right of 90

Final BST Structure:

```
      23
    /    \
   2      67
    \    /  \
    11  56   100
       /    /
      34   90
           \
           99
```

Step 2: Tree Traversals

1. Inorder Traversal (Left → Root → Right)

```
2, 11, 23, 34, 56, 67, 90, 99, 100
```

2. Preorder Traversal (Root → Left → Right)

```
23, 2, 11, 67, 56, 34, 100, 90, 99
```

3. Postorder Traversal (Left → Right → Root)

```
11, 2, 34, 56, 99, 90, 100, 67, 23
```

M4Q13) Construct two binary trees for the given: i) Post order sequence: DJGBHEAFKIC, ii) Inorder sequence: JGDHEBKIFCA

- **Postorder**: `DJGBHEAFKIC`
- **Inorder**: `JGDHEBKIFCA`

We will use **Postorder** and **Inorder** to construct **one unique binary tree**, not two, since for any given Postorder and Inorder (with distinct elements), there is **only one unique binary tree** possible.

tep-by-Step Construction

Step 1: Identify Root from Postorder

- In **Postorder**, the **last node is the root**.
- **Root = A**

Split **Inorder** around `A`:

```
Inorder: J G D H E B K I F C | A
```

- Left Subtree = `J G D H E B K I F C`
- Right Subtree = empty

So, **entire tree lies on the left of A**.

Step 2: Next Root in Postorder = `C`

Now working backwards in Postorder:

Current subtree's Postorder (excluding A):

`D J G B H E F K I C`

- Root = `C`

Split Inorder around `C`:

```
Inorder: J G D H E B K I F | C
```

Left = `J G D H E B K I F`

Right = empty

Step 3: Continue Process

Postorder: `D J G B H E F K I`

Inorder: `J G D H E B K I F`

Next root = `I` (last of current Postorder)

Split Inorder around `I`:

```
Inorder: J G D H E B K | I | F
```

- Left subtree of I: `J G D H E B K`
- Right subtree of I: `F`

Postorder for left subtree: `D J G B H E K`

Postorder for right subtree: `F`

Continue recursively:

Continue this Recursively (abbreviated):

From this step-by-step process, the tree structure will eventually look like this:

```
          A
         /
        C
       /
      I
     / \
    K   F
   /
  B
 /
E
 /
 H
/
D
 \
  G
 /
J
```

M4Q14) Define a binary tree for the expression 3+4*(7-6)/4+3. Traverse the above generated tree using inorder, preorder and postorder. Also write the C function for each.

Step 1: Convert Infix to Binary Expression Tree

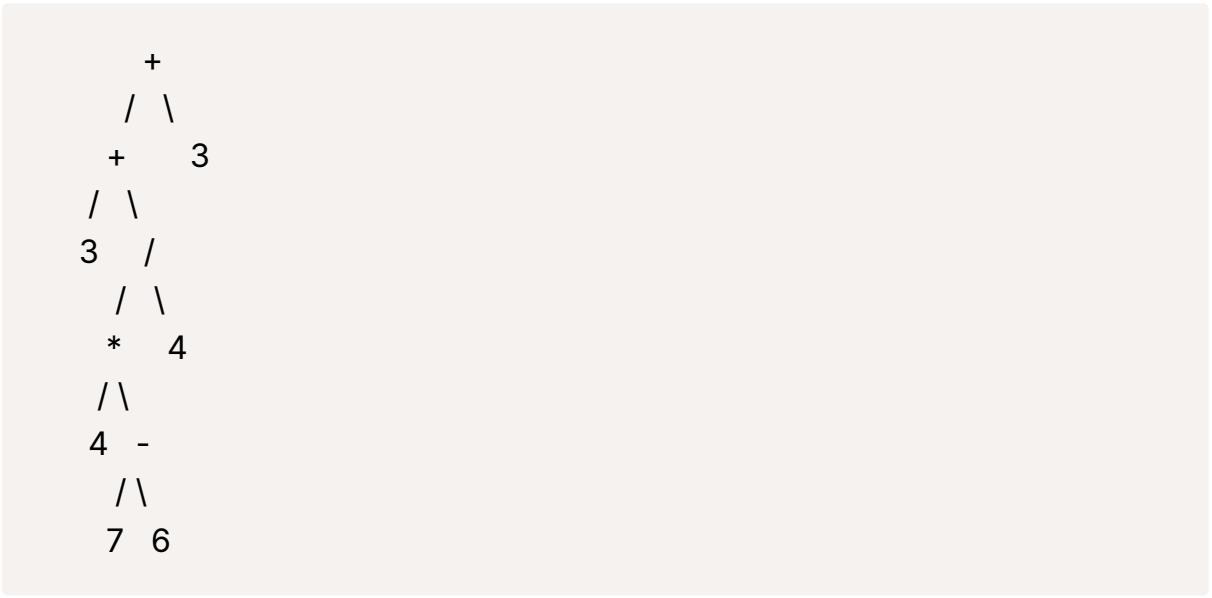
Given Expression:

$3 + \frac{4 \times (7-6)}{4} + 3$

Operator precedence:

1. Parentheses → (7 − 6)
2. Multiplication → 4 × (7 − 6)
3. Division → [4 × (7 − 6)] ÷ 4
4. Addition → 3 + [...] + 3

Constructed Binary Expression Tree



Step 2: Tree Traversals

1. Inorder (Left → Root → Right)

In infix notation (with parentheses for clarity):

```
(((3 + ((4 * (7 - 6)) / 4)) + 3))
```

2. Preorder (Root → Left → Right)

Prefix notation:

```
+ + 3 / * 4 - 7 6 4 3
```

3. Postorder (Left → Right → Root)

Postfix notation:

```
3 4 7 6 - * 4 / + 3 +
```

Step 3: C Functions for Tree Traversals

Inorder Traversal

```
void inorder(struct node* root) {
    if (root != NULL) {
        if (root->left || root->right) printf("(");
        inorder(root->left);
        printf("%c", root->data);
        inorder(root->right);
        if (root->left || root->right) printf(")");
    }
}
```

Preorder Traversal

```
void preorder(struct node* root)
{
    if (root != NULL) {
        printf("%c ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

Postorder Traversal

```
void postorder(struct node* root)
{
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%c ", root->data);
    }
}
```

********** EOF **********