SAURABH KALA
SAP ID – 590014968

Lab Experiment: 09 **Batch: 1**
Subject: Data Structures Lab **MCA**
Semester: 1st

Objective:

To understand the structure and implementation of binary trees using arrays and linked lists.

• To perform various tree traversal techniques (in-order, pre-order, post-order, and level-order).

• To implement heap sorting using a binary tree structure.

Instructions:

Implement the following tasks in C. Use appropriate data structures (array or linked list) to create the binary tree and demonstrate traversal methods and heap sorting. 1st Assignment:

Binary Tree Creation Using Arrays:

• Represent a complete binary tree using an array.

• Note that for a node at index i:

• The left child is at 2 * i + 1

• The right child is at 2 * i + 2 Using Linked Lists:

• Represent a binary tree where each node contains data and pointers to its left and right children.

• Include functions to create and insert nodes in the binary tree

Binary Tree Using Arrays

main.c

Share Run Output

```c
#include <stdio.h>

#define MAX_SIZE 100

int tree[MAX_SIZE];
int n = 0; // Current size of the tree

// Function to insert an element into the tree
void insert(int value) {
    if (n < MAX_SIZE) {
        tree[n] = value;
        n++;
    }
}

// In-order traversal
void inorder(int index) {
    if (index >= n) return;
    inorder(2 * index + 1); // Left child
    printf("%d ", tree[index]);
    inorder(2 * index + 2); // Right child
}

// Pre-order traversal
void preorder(int index) {
```

```
/tmp/4VfyOlU8Or.o
In-order Traversal: 4 2 5 1 3
Pre-order Traversal: 1 2 4 5 3
Post-order Traversal: 4 5 2 3 1
Level-order Traversal: 1 2 3 4 5


=== Code Execution Successful ===
```

main.c

Share Run Output

```c
// Pre-order traversal
void preorder(int index) {
    if (index >= n) return;
    printf("%d ", tree[index]);
    preorder(2 * index + 1);
    preorder(2 * index + 2);
}

// Post-order traversal
void postorder(int index) {
    if (index >= n) return;
    postorder(2 * index + 1);
    postorder(2 * index + 2);
    printf("%d ", tree[index]);
}

// Level-order traversal
void levelOrder() {
    for (int i = 0; i < n; i++) {
        printf("%d ", tree[i]);
    }
    printf("\n");
}
```

```
/tmp/4VfyOlU8Or.o
In-order Traversal: 4 2 5 1 3
Pre-order Traversal: 1 2 4 5 3
Post-order Traversal: 4 5 2 3 1
Level-order Traversal: 1 2 3 4 5


=== Code Execution Successful ===
```

main.c

Share Run

```
int main() {
    // Inserting elements into the binary tree
    insert(1);
    insert(2);
    insert(3);
    insert(4);
    insert(5);

    printf("In-order Traversal: ");
    inorder(0);
    printf("\n");

    printf("Pre-order Traversal: ");
    preorder(0);
    printf("\n");

    printf("Post-order Traversal: ");
    postorder(0);
    printf("\n");

    printf("Level-order Traversal: ");
    levelOrder();

    return 0;
```

Output

```
/tmp/4VfyOlU8Or.o
In-order Traversal: 4 2 5 1 3
Pre-order Traversal: 1 2 4 5 3
Post-order Traversal: 4 5 2 3 1
Level-order Traversal: 1 2 3 4 5


=== Code Execution Successful ===
```

Binary Tree Using Linked Lists

main.c

Share Run

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

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

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

// Function to insert nodes in a binary tree (manually for simplicity)
struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL) return createNode(data);

    if (data < root->data)
        root->left = insertNode(root->left, data);
    else
        root->right = insertNode(root->right, data);
```

Output

```
/tmp/S5236UxPzh.o
In-order Traversal: 4 6 8 10 12 15 18
Pre-order Traversal: 10 6 4 8 15 12 18
Post-order Traversal: 4 8 6 12 18 15 10


=== Code Execution Successful ===
```

main.c

Share Run

```c
    return root;
}

// In-order traversal
void inorder(struct Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

// Pre-order traversal
void preorder(struct Node* root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

// Post-order traversal
void postorder(struct Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}
```

Output

```
/tmp/S5236UxPzh.o
In-order Traversal: 4 6 8 10 12 15 18
Pre-order Traversal: 10 6 4 8 15 12 18
Post-order Traversal: 4 8 6 12 18 15 10


=== Code Execution Successful ===
```

main.c

Share Run

```
53
54  int main() {
55      struct Node* root = NULL;
56
57      root = insertNode(root, 10);
58      insertNode(root, 6);
59      insertNode(root, 15);
60      insertNode(root, 4);
61      insertNode(root, 8);
62      insertNode(root, 12);
63      insertNode(root, 18);
64
65      printf("In-order Traversal: ");
66      inorder(root);
67      printf("\n");
68
69      printf("Pre-order Traversal: ");
70      preorder(root);
71      printf("\n");
72
73      printf("Post-order Traversal: ");
74      postorder(root);
75      printf("\n");
76
77      return 0;
```

Output

```
/tmp/S5236UxPzh.o
In-order Traversal: 4 6 8 10 12 15 18
Pre-order Traversal: 10 6 4 8 15 12 18
Post-order Traversal: 4 8 6 12 18 15 10


=== Code Execution Successful ===
```

Heap Sort using Binary Tree

main.c

Share Run Output

```c
#include <stdio.h>

#define MAX_SIZE 100

int heap[MAX_SIZE];
int size = 0;

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Heapify function to maintain max-heap property
void heapify(int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < size && heap[left] > heap[largest])
        largest = left;
    if (right < size && heap[right] > heap[largest])
        largest = right;

    if (largest != i) {
        swap(&heap[i], &heap[largest]);
        heapify(largest);
    }
}

// Insert an element into the heap
void insertHeap(int value) {
    if (size < MAX_SIZE) {
        heap[size] = value;
        int i = size;
        size++;

        while (i != 0 && heap[(i - 1) / 2] < heap[i]) {
            swap(&heap[(i - 1) / 2], &heap[i]);
            i = (i - 1) / 2;
        }
    }
}

// Perform heap sort
void heapSort() {
    for (int i = size - 1; i >= 0; i--) {
        swap(&heap[0], &heap[i]);
        size--;
```

```
/tmp/rg92H5wqn3.o
Heap Sort: 1 2 3 4 5 6


=== Code Execution Successful ===
```

main.c

Share Run Output

```c
    if (largest != i) {
        swap(&heap[i], &heap[largest]);
        heapify(largest);
    }
}

// Insert an element into the heap
void insertHeap(int value) {
    if (size < MAX_SIZE) {
        heap[size] = value;
        int i = size;
        size++;

        while (i != 0 && heap[(i - 1) / 2] < heap[i]) {
            swap(&heap[(i - 1) / 2], &heap[i]);
            i = (i - 1) / 2;
        }
    }
}

// Perform heap sort
void heapSort() {
    for (int i = size - 1; i >= 0; i--) {
        swap(&heap[0], &heap[i]);
        size--;
```

```
/tmp/rg92H5wqn3.o
Heap Sort: 1 2 3 4 5 6


=== Code Execution Successful ===
```

main.c

Share Run Output

```c
// Perform heap sort
void heapSort() {
    for (int i = size - 1; i >= 0; i--) {
        swap(&heap[0], &heap[i]);
        size--;
        heapify(0);
    }
}

int main() {
    insertHeap(3);
    insertHeap(1);
    insertHeap(6);
    insertHeap(5);
    insertHeap(2);
    insertHeap(4);

    printf("Heap Sort: ");
    heapSort();
    for (int i = 0; i < 6; i++) {
        printf("%d ", heap[i]);
    }
    printf("\n");

    return 0;
```

```
/tmp/rg92H5wqn3.o
Heap Sort: 1 2 3 4 5 6


=== Code Execution Successful ===
```

Explanation:

- **Array Representation**: Simple complete binary tree using an array.
- **Linked List Representation**: Binary tree using nodes with pointers.
- **Heap Sort**: Uses a max-heap to perform sorting.

2nd Assignment:

Tree Traversal Methods

Implement the following traversal methods: In-order Traversal:

• Traverse the left subtree, visit the root node, then traverse the right subtree.

Pre-order Traversal:

• Visit the root node, traverse the left subtree, then traverse the right subtree.

Post-order Traversal:

• Traverse the left subtree, traverse the right subtree, then visit the root node.

Level-order Traversal:

• Traverse the nodes level by level, starting from the root.

Implement each traversal function and test them with the binary tree created above.

main.c

Share Run Output

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

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

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

// Function to insert nodes in a binary tree (manual insertion for testing)
struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL)
        return createNode(data);

    // Simple binary search tree (BST) insertion for demonstration
    if (data < root->data)
        root->left = insertNode(root->left, data);
```

```
/tmp/bagqM5ppo5.o
In-order Traversal: 4 6 8 10 12 15 18
Pre-order Traversal: 10 6 4 8 15 12 18
Post-order Traversal: 4 8 6 12 18 15 10
Level-order Traversal: 10 6 15 4 8 12 18


=== Code Execution Successful ===
```

main.c

Share Run

```
        root->left = insertNode(root->left, data);
    else
        root->right = insertNode(root->right, data);

    return root;
}

// In-order traversal: Left -> Root -> Right
void inorder(struct Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

// Pre-order traversal: Root -> Left -> Right
void preorder(struct Node* root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

// Post-order traversal: Left -> Right -> Root
void postorder(struct Node* root) {
```

Output

```
/tmp/bagqM5ppo5.o
In-order Traversal: 4 6 8 10 12 15 18
Pre-order Traversal: 10 6 4 8 15 12 18
Post-order Traversal: 4 8 6 12 18 15 10
Level-order Traversal: 10 6 15 4 8 12 18

=== Code Execution Successful ===
```

main.c

Share Run

```
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}

// Level-order traversal using a queue
void levelOrder(struct Node* root) {
    if (root == NULL) return;

    // Queue implementation using an array
    struct Node* queue[100];
    int front = 0, rear = 0;

    // Enqueue the root node
    queue[rear++] = root;

    while (front < rear) {
        // Dequeue the front node
        struct Node* current = queue[front++];
        printf("%d ", current->data);

        // Enqueue the left child if it exists
        if (current->left != NULL)
            queue[rear++] = current->left;
```

Output

```
/tmp/bagqM5ppo5.o
In-order Traversal: 4 6 8 10 12 15 18
Pre-order Traversal: 10 6 4 8 15 12 18
Post-order Traversal: 4 8 6 12 18 15 10
Level-order Traversal: 10 6 15 4 8 12 18

=== Code Execution Successful ===
```

main.c

Share Run

Output

```
75
76          // Enqueue the right child if it exists
77          if (current->right != NULL)
78              queue[rear++] = current->right;
79      }
80  }
81
82  int main() {
83      struct Node* root = NULL;
84
85      // Manually creating a binary tree for testing
86      root = insertNode(root, 10);
87      insertNode(root, 6);
88      insertNode(root, 15);
89      insertNode(root, 4);
90      insertNode(root, 8);
91      insertNode(root, 12);
92      insertNode(root, 18);
93
94      printf("In-order Traversal: ");
95      inorder(root);
96      printf("\n");
97
98      printf("Pre-order Traversal: ");
99      preorder(root);
```

```
/tmp/bagqM5ppo5.o
In-order Traversal: 4 6 8 10 12 15 18
Pre-order Traversal: 10 6 4 8 15 12 18
Post-order Traversal: 4 8 6 12 18 15 10
Level-order Traversal: 10 6 15 4 8 12 18


=== Code Execution Successful ===
```

main.c

Share Run

Output

```
86      root = insertNode(root, 10);
87      insertNode(root, 6);
88      insertNode(root, 15);
89      insertNode(root, 4);
90      insertNode(root, 8);
91      insertNode(root, 12);
92      insertNode(root, 18);
93
94      printf("In-order Traversal: ");
95      inorder(root);
96      printf("\n");
97
98      printf("Pre-order Traversal: ");
99      preorder(root);
100     printf("\n");
101
102     printf("Post-order Traversal: ");
103     postorder(root);
104     printf("\n");
105
106     printf("Level-order Traversal: ");
107     levelOrder(root);
108     printf("\n");
109
110     return 0;
```

```
/tmp/bagqM5ppo5.o
In-order Traversal: 4 6 8 10 12 15 18
Pre-order Traversal: 10 6 4 8 15 12 18
Post-order Traversal: 4 8 6 12 18 15 10
Level-order Traversal: 10 6 15 4 8 12 18


=== Code Execution Successful ===
```

Explanation of the Code:

1. **Node Structure:**
    - **Each node contains:**
        - **An integer data**
        - **A pointer to the left child**
        - **A pointer to the right child**
2. **Node Insertion:**
    - **The insertNode function inserts nodes into a Binary Search Tree (BST).**
3. **Traversal Functions:**
    - **In-order Traversal (inorder): Recursively traverses the left subtree, visits the root, then traverses the right subtree.**
    - **Pre-order Traversal (preorder): Visits the root, then recursively traverses the left and right subtrees.**
    - **Post-order Traversal (postorder): Recursively traverses the left and right subtrees, then visits the root.**
    - **Level-order Traversal (levelOrder): Uses a queue to traverse the nodes level by level.**
4. **Level-order Traversal Implementation:**
    - **We use a simple array-based queue to implement level-order traversal**