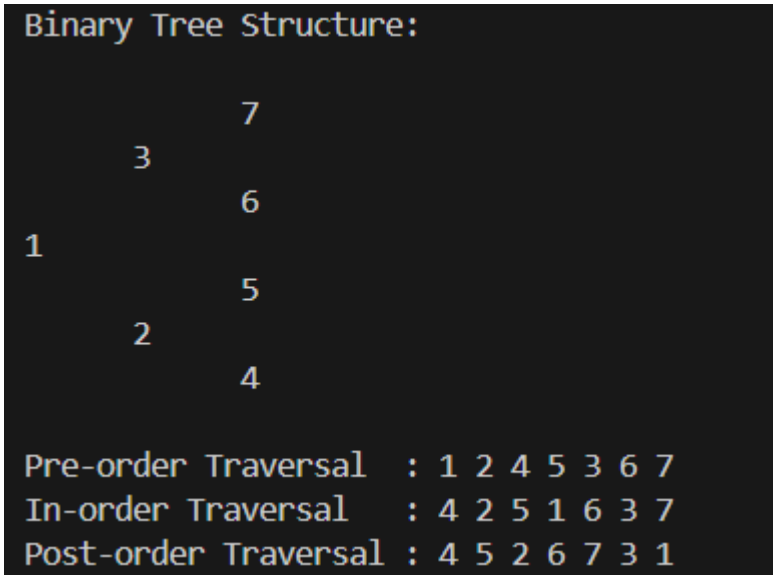
ASSIGNMENT 8

1. Write program using functions for binary tree traversals: Pre-order, In-order and Post. order using a recursive approach.

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

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = right = NULL;
    }
};

void printTree(Node* root, int space = 0) {
    if (root == NULL) return;

    space += 6;

    printTree(root->right, space);
```

```
    cout << endl;
    for (int i = 6; i < space; i++)
        cout << " ";
    cout << root->data;

    printTree(root->left, space);
}

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

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

void postorder(Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}
```

```
int main() {
    // Creating the standard binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    root->right->right = new Node(7);

    cout << "Binary Tree Structure:\n";
    printTree(root);

    cout << "\n\nPre-order Traversal  : ";
    preorder(root);

    cout << "\nIn-order Traversal   : ";
    inorder(root);

    cout << "\nPost-order Traversal : ";
    postorder(root);

    cout << endl;
    return 0;
}
```


Binary Tree Structure:
7
3
6
1
5
2
4
Pre-order Traversal : 1 2 4 5 3 6 7
In-order Traversal : 4 2 5 1 6 3 7
Post-order Traversal : 4 5 2 6 7 3 1

2. Implement following functions for Binary Search Trees

(a) Search a given item (Recursive & Non-Recursive)

(b) Maximum element of the BST

(c) Minimum element of the BST

(d) In-order successor of a given node the BST

(e) In-order predecessor of a given node the BST

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

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = right = NULL;
    }
};

Node* insertNode(Node* root, int val) {
    if (root == NULL)
        return new Node(val);
```

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

    return root;
}

// (a) Recursive Search
Node* searchRec(Node* root, int key) {
    if (root == NULL || root->data == key)
        return root;

    if (key < root->data)
        return searchRec(root->left, key);
    return searchRec(root->right, key);
}

// (a) Non-recursive Search
Node* searchIter(Node* root, int key) {
    while (root != NULL) {
        if (root->data == key)
            return root;
        else if (key < root->data)
            root = root->left;
        else
            root = root->right;
    }
```

```
    return NULL;
}

// (b) Maximum element in BST
Node* findMax(Node* root) {
    if (root == NULL) return NULL;

    while (root->right != NULL)
        root = root->right;

    return root;
}

// (c) Minimum element in BST
Node* findMin(Node* root) {
    if (root == NULL) return NULL;

    while (root->left != NULL)
        root = root->left;

    return root;
}

// (d) In-order Successor
Node* inorderSuccessor(Node* root, int key) {
    Node* curr = searchIter(root, key);
    if (curr == NULL) return NULL;
```

```
    // Case 1: Right subtree exists → successor = min(right)
    if (curr->right != NULL)
        return findMin(curr->right);

    // Case 2: No right subtree → find ancestor
    Node* successor = NULL;
    Node* ancestor = root;

    while (ancestor != curr) {
        if (curr->data < ancestor->data) {
            successor = ancestor;
            ancestor = ancestor->left;
        } else {
            ancestor = ancestor->right;
        }
    }
    return successor;
}

// (e) In-order Predecessor
Node* inorderPredecessor(Node* root, int key) {
    Node* curr = searchIter(root, key);
    if (curr == NULL) return NULL;

    // Case 1: Left subtree exists → predecessor = max(left)
    if (curr->left != NULL)
        return findMax(curr->left);
```

```
    // Case 2: No left subtree → find ancestor
    Node* predecessor = NULL;
    Node* ancestor = root;

    while (ancestor != curr) {
        if (curr->data > ancestor->data) {
            predecessor = ancestor;
            ancestor = ancestor->right;
        } else {
            ancestor = ancestor->left;
        }
    }
    return predecessor;
}

// In-order Print for reference
void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

int main() {
    Node* root = NULL;

    // Constructing BST
    int arr[] = {20, 10, 30, 5, 15, 25, 40};
```

```
for (int x : arr)
    root = insertNode(root, x);

cout << "BST (In-order): ";
inorder(root);
cout << endl;

int key = 15;

// Search
cout << "\nSearching for " << key << " (Recursive): ";
cout << (searchRec(root, key) ? "Found" : "Not Found");

cout << "\nSearching for " << key << " (Iterative): ";
cout << (searchIter(root, key) ? "Found" : "Not Found");

// Min & Max
cout << "\n\nMinimum Element: " << findMin(root)->data;
cout << "\nMaximum Element: " << findMax(root)->data;

// Successor & Predecessor
Node* succ = inorderSuccessor(root, key);
Node* pred = inorderPredecessor(root, key);

cout << "\n\nIn-order Successor of " << key << ": ";
if (succ) cout << succ->data; else cout << "None";

cout << "\nIn-order Predecessor of " << key << ": ";
```

```
    if (pred) cout << pred->data; else cout << "None";


    cout << endl;

    return 0;
}
```

```
BST (In-order): 5 10 15 20 25 30 40

Searching for 15 (Recursive): Found
Searching for 15 (Iterative): Found

Minimum Element: 5
Maximum Element: 40

In-order Successor of 15: 20
In-order Predecessor of 15: 10
```

3. Write a program for binary search tree (BST) having functions for the following

operations:

(a) Insert an element (no duplicates are allowed),

(b) Delete an existing element,

(c) Maximum depth of BST

(d) Minimum depth of

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

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = right = NULL;
    }
};

Node* insertNode(Node* root, int val) {
    if (root == NULL)
        return new Node(val);
```

```
    if (val < root->data)
        root->left = insertNode(root->left, val);
    else if (val > root->data)
        root->right = insertNode(root->right, val);
    else
        cout << "Duplicate value, not inserted.\n";

    return root;
}

Node* findMin(Node* root) {
    while (root->left != NULL)
        root = root->left;
    return root;
}

Node* deleteNode(Node* root, int key) {
    if (root == NULL)
        return NULL;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        if (root->left == NULL && root->right == NULL) {
            delete root;
            return NULL;
```

```
        }
        else if (root->left == NULL) {
            Node* t = root->right;
            delete root;
            return t;
        }
        else if (root->right == NULL) {
            Node* t = root->left;
            delete root;
            return t;
        }
        else {
            Node* t = findMin(root->right);
            root->data = t->data;
            root->right = deleteNode(root->right, t->data);
        }
    }
    return root;
}

int maxDepth(Node* root) {
    if (root == NULL) return 0;
    return 1 + max(maxDepth(root->left), maxDepth(root->right));
}

int minDepth(Node* root) {
    if (root == NULL) return 0;
    if (root->left == NULL)
```

```
        return 1 + minDepth(root->right);
    if (root->right == NULL)
        return 1 + minDepth(root->left);
    return 1 + min(minDepth(root->left), minDepth(root->right));
}

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

int main() {
    Node* root = NULL;
    int choice, val;

    while (true) {
        cout << "\n--- BST MENU ---\n";
        cout << "1. Insert Node\n";
        cout << "2. Delete Node\n";
        cout << "3. Display Inorder\n";
        cout << "4. Maximum Depth\n";
        cout << "5. Minimum Depth\n";
        cout << "6. Exit\n";
        cout << "Enter choice: ";
        cin >> choice;
```

```
switch (choice) {
case 1:
    cout << "Enter value to insert: ";
    cin >> val;
    root = insertNode(root, val);
    break;

case 2:
    cout << "Enter value to delete: ";
    cin >> val;
    root = deleteNode(root, val);
    break;

case 3:
    cout << "Inorder Traversal: ";
    inorder(root);
    cout << endl;
    break;

case 4:
    cout << "Maximum Depth: " << maxDepth(root) << endl;
    break;

case 5:
    cout << "Minimum Depth: " << minDepth(root) << endl;
    break;

case 6:
```

```
            return 0;

        default:
            cout << "Invalid choice!\n";
        }
    }

    return 0;
}
```

```
--- BST MENU ---
1. Insert Node
2. Delete Node
3. Display Inorder
4. Maximum Depth
5. Minimum Depth
6. Exit
Enter choice: 3
Inorder Traversal: 1 2 3 4 5 6 7 8 9

--- BST MENU ---
1. Insert Node
2. Delete Node
3. Display Inorder
4. Maximum Depth
5. Minimum Depth
6. Exit
Enter choice: 1
Enter value to insert: 0

--- BST MENU ---
1. Insert Node
2. Delete Node
3. Display Inorder
4. Maximum Depth
5. Minimum Depth
6. Exit
Enter choice: 2
Enter value to delete: 5
```

4. Write a program to determine whether a given binary tree is a BST or not.

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

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = right = NULL;
    }
};

// Function to check BST property using min/max range
bool isBSTUtil(Node* root, int minVal, int maxVal) {
    if (root == NULL)
        return true;

    if (root->data <= minVal || root->data >= maxVal)
        return false;

    return isBSTUtil(root->left, minVal, root->data) &&
           isBSTUtil(root->right, root->data, maxVal);
```

```
}

bool isBST(Node* root) {
    return isBSTUtil(root, INT_MIN, INT_MAX);
}

// Inorder print for verifying tree contents
void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

int main() {

    /*
       Creating a general binary tree
              8
             / \
            3   10
           / \   \
          12  6    14

       This is **NOT** a BST (12 is in the left subtree of 8)
    */

    Node* root = new Node(8);
```

```
    root->left = new Node(3);
    root->right = new Node(10);
    root->left->left = new Node(12);   // breaks BST rule
    root->left->right = new Node(6);
    root->right->right = new Node(14);

    cout << "Inorder Traversal of the Tree: ";
    inorder(root);
    cout << endl;

    if (isBST(root))
        cout << "The tree IS a BST.\n";
    else
        cout << "The tree is NOT a BST.\n";

    return 0;
}
```

```
Inorder Traversal of the Tree: 12 3 6 8 10 14
The tree is NOT a BST.
```

5. Implement Heapsort (Increasing/Decreasing order).

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

void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSortIncreasing(vector<int>& arr) {
    int n = arr.size();

    for (int i = n/2 - 1; i >= 0; i--)
```

```
        heapify(arr, n, i);

    for (int i = n - 1; i >= 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

// For decreasing: build MIN-heap
void heapifyMin(vector<int>& arr, int n, int i) {
    int smallest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;

    if (left < n && arr[left] < arr[smallest])
        smallest = left;

    if (right < n && arr[right] < arr[smallest])
        smallest = right;

    if (smallest != i) {
        swap(arr[i], arr[smallest]);
        heapifyMin(arr, n, smallest);
    }
}

void heapSortDecreasing(vector<int>& arr) {
    int n = arr.size();
```

```
    for (int i = n/2 - 1; i >= 0; i--)
        heapifyMin(arr, n, i);

    for (int i = n - 1; i >= 0; i--) {
        swap(arr[0], arr[i]);
        heapifyMin(arr, i, 0);
    }
}

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Enter elements:\n";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    int choice;
    cout << "\n1. Sort Increasing\n2. Sort Decreasing\nEnter choice: ";
    cin >> choice;

    if (choice == 1)
        heapSortIncreasing(arr);
    else if (choice == 2)
        heapSortDecreasing(arr);
```

```
    else
        cout << "Invalid choice!\n";

    cout << "\nSorted Array: ";
    for (int x : arr)
        cout << x << " ";
    cout << endl;

    return 0;
}
```

```
Enter number of elements: 5
Enter elements:
5 2 0 8 4

1. Sort Increasing
2. Sort Decreasing
Enter choice: 1

Sorted Array: 0 2 4 5 8
```

6. Implement priority queues using heaps.

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

class PriorityQueue {
public:
    vector<int> heap;

    // Function to swap two values
    void swapVals(int &a, int &b) {
        int temp = a;
        a = b;
        b = temp;
    }

    // Push an element (insert)
    void push(int val) {
        heap.push_back(val);
        int i = heap.size() - 1;

        // Up-heap bubbling
        while (i > 0) {
            int parent = (i - 1) / 2;
            if (heap[parent] < heap[i]) {
                swapVals(heap[parent], heap[i]);
```

```
            i = parent;
        } else break;
    }
}

// Pop the highest priority element
void pop() {
    if (heap.empty()) {
        cout << "Priority Queue is empty.\n";
        return;
    }

    // Move last element to root
    heap[0] = heap.back();
    heap.pop_back();

    // Down-heap trickle
    heapify(0);
}

// Heapify function (max-heap)
void heapify(int i) {
    int size = heap.size();
    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) {
        swapVals(heap[i], heap[largest]);
        heapify(largest);
    }
}

// Return highest priority element
int top() {
    if (heap.empty()) {
        cout << "Priority Queue is empty.\n";
        return -1;
    }
    return heap[0];
}

// Display the heap
void display() {
    if (heap.empty()) {
        cout << "Priority Queue is empty.\n";
        return;
    }
    cout << "Priority Queue (Heap): ";
    for (int x : heap)
```

```
            cout << x << " ";
        cout << endl;
    }
};

int main() {
    PriorityQueue pq;
    int choice, val;

    while (true) {
        cout << "\n--- PRIORITY QUEUE MENU ---\n";
        cout << "1. Insert (Push)\n";
        cout << "2. Delete Highest Priority (Pop)\n";
        cout << "3. Peek (Top Element)\n";
        cout << "4. Display\n";
        cout << "5. Exit\n";
        cout << "Enter choice: ";
        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Enter value to insert: ";
            cin >> val;
            pq.push(val);
            break;

        case 2:
            pq.pop();
```

```
            break;

        case 3:
            cout << "Top element: " << pq.top() << endl;
            break;

        case 4:
            pq.display();
            break;

        case 5:
            return 0;

        default:
            cout << "Invalid choice!\n";
        }
    }
}
```

```
--- PRIORITY QUEUE MENU ---
1. Insert (Push)
2. Delete Highest Priority (Pop)
3. Peek (Top Element)
4. Display
5. Exit
Enter choice: 1
Enter value to insert: 6
```