

## Lab Assignment 8

### Binary Search Trees and Heap

Name- Ujjwal Pant    Batch- 2C24    Rollno. 1024030370

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

```
1 #include <iostream>
2 using namespace std;
3
4 class Node {
5 public:
6     int data;
7     Node* left;
8     Node* right;
9
10    Node(int value) {
11        data = value;
12        left = right = nullptr;
13    }
14 };
15
16 void preorder(Node* root) {
17     if (root == nullptr) return;
18     cout << root->data << " ";
19     preorder(root->left);
20     preorder(root->right);
21 }
22
23 void inorder(Node* root) {
24     if (root == nullptr) return;
25     inorder(root->left);
26     cout << root->data << " ";
27     inorder(root->right);
28 }
29
30 void postorder(Node* root) {
31     if (root == nullptr) return;
32     postorder(root->left);
33     postorder(root->right);
34     cout << root->data << " ";
35 }
36
37 int main() {
38     Node* root = new Node(1);
39     root->left = new Node(2);
40     root->right = new Node(3);
41     root->left->left = new Node(4);
42     root->left->right = new Node(5);
43     root->right->right = new Node(6);
44
45     cout << "Pre-order traversal: ";
46     preorder(root);
47     cout << endl;
48
49     cout << "In-order traversal: ";
50     inorder(root);
51     cout << endl;
52
53     cout << "Post-order traversal: ";
54     postorder(root);
55     cout << endl;
56
57     return 0;
58 }
```

Pre-order traversal: 1 2 4 5 3 6  
In-order traversal: 4 2 5 1 3 6  
Post-order traversal: 4 5 2 6 3 1

=== Code Execution Successful ===

## 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

```
1 #include <iostream>
2 using namespace std;
3
4 class Node {
5 public:
6     int data;
7     Node* left;
8     Node* right;
9
10    Node(int value) {
11        data = value;
12        left = right = nullptr;
13    }
14 };
15
16 Node* insert(Node* root, int value) {
17     if (root == nullptr) return new Node(value);
18     if (value < root->data)
19         root->left = insert(root->left, value);
20     else if (value > root->data)
21         root->right = insert(root->right, value);
22     return root;
23 }
24
25 bool searchRecursive(Node* root, int key) {
26     if (root == nullptr) return false;
27     if (key == root->data) return true;
28     if (key < root->data)
29         return searchRecursive(root->left, key);
30     else
31         return searchRecursive(root->right, key);
32 }
33
34 bool searchIterative(Node* root, int key) {
35     while (root != nullptr) {
36         if (key == root->data) return true;
37         else if (key < root->data) root = root->left;
38         else root = root->right;
39     }
40     return false;
41 }
42
43 Node* findMax(Node* root) {
44     if (root == nullptr) return nullptr;
45     while (root->right != nullptr)
46         root = root->right;
47     return root;
48 }
49
50 Node* findMin(Node* root) {
51     if (root == nullptr) return nullptr;
52     while (root->left != nullptr)
53         root = root->left;
54     return root;
55 }
56
57 Node* inorderSuccessor(Node* root, Node* n) {
58     if (n->right != nullptr)
59         return findMin(n->right);
60
61     Node* succ = nullptr;
62     while (root != nullptr) {
63         if (n->data < root->data) {
64             succ = root;
65             root = root->left;
66         } else if (n->data > root->data)
67             root = root->right;
68         else
69             break;
70     }
71     return succ;
72 }
73
74 Node* inorderPredecessor(Node* root, Node* n) {
75     if (n->left != nullptr)
76         return findMax(n->left);
77
78     Node* pred = nullptr;
79     while (root != nullptr) {
80         if (n->data > root->data) {
81             pred = root;
82             root = root->right;
83         } else if (n->data < root->data)
84             root = root->left;
85         else
86             break;
87     }
88     return pred;
89 }
90
```

```
91 int main() {
92     Node* root = nullptr;
93
94     int values[] = {5, 3, 2, 4, 8, 9, 10, 16};
95     for (int v : values)
96         root = insert(root, v);
97
98     cout << "In-order Traversal of BST:\n";
99     // (Assumed present in screenshot)
100    // inorderTraversal(root);
101
102    int key = 40;
103    cout << "Recursive search for " << key << ": ";
104    cout << (searchRecursive(root, key) ? "Found" : "Not Found") << endl;
105
106    key = 4;
107    cout << "Recursive search: " << (searchRecursive(root, key) ?
108        "Found" : "Not Found") << endl;
109    cout << "Iterative search: " << (searchIterative(root, key) ?
110        "Found" : "Not Found") << endl;
111
112    Node* maxNode = findMax(root);
113    if (maxNode) cout << "Maximum element: " << maxNode->data << endl;
114
115    Node* minNode = findMin(root);
116    if (minNode) cout << "Minimum element: " << minNode->data << endl;
117
118    Node* node = root->left->right;
119    Node* succ = inorderSuccessor(root, node);
120    if (succ)
121        cout << "In-order Successor of " << node->data << ": " << succ
122            << endl;
123
124    Node* pred = inorderPredecessor(root, node);
125    if (pred)
126        cout << "In-order Predecessor of " << node->data << ": " << pred
127            << endl;
128
129    return 0;
130 }
```

```
In-order Traversal of BST:
Recursive search for 40: Not Found
Recursive search: Found
Iterative search: Found
Maximum element: 16
Minimum element: 2
In-order Successor of 4: 5
In-order Predecessor of 4: 3
```

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 BST

```
1 #include <iostream>
2 using namespace std;
3
4 class Node {
5 public:
6     int data;
7     Node* left;
8     Node* right;
9
10    Node(int v) {
11        data = v;
12        left = right = nullptr;
13    }
14 };
15
16 class BST {
17 public:
18     Node* insert(Node* root, int key) {
19         if (root == nullptr) return new Node(key);
20         if (key < root->data)
21             root->left = insert(root->left, key);
22         else if (key > root->data)
23             root->right = insert(root->right, key);
24         return root;
25     }
26
27     Node* findMinNode(Node* root) {
28         while (root->left != nullptr)
29             root = root->left;
30         return root;
31     }
32
33     Node* deleteNode(Node* root, int key) {
34         if (root == nullptr) return root;
35
36         if (key < root->data)
37             root->left = deleteNode(root->left, key);
38         else if (key > root->data)
39             root->right = deleteNode(root->right, key);
40         else {
41             if (!root->left) return root->right;
42             else if (!root->right) return root->left;
43
44             Node* t = findMinNode(root->right);
45             root->data = t->data;
46             root->right = deleteNode(root->right, t->data);
47         }
48         return root;
49     }
50
51     int maxDepth(Node* root) {
52         if (root == nullptr) return 0;
53         int leftDepth = maxDepth(root->left);
54         int rightDepth = maxDepth(root->right);
55         return 1 + max(leftDepth, rightDepth);
56     }
57
58     int minDepth(Node* root) {
59         if (root == nullptr) return 0;
60         if (root->left == nullptr) return 1 + minDepth(root->right);
61         if (root->right == nullptr) return 1 + minDepth(root->left);
62         return 1 + min(minDepth(root->left), minDepth(root->right));
63     }
64 };
65
66 int main() {
67     BST t;
68     Node* root = nullptr;
69
70     int arr[] = {8, 3, 1, 6, 4, 7, 10, 14, 13};
71     for (int x : arr)
72         root = t.insert(root, x);
73
74     cout << "Max Depth: " << t.maxDepth(root) << endl;
75     cout << "Min Depth: " << t.minDepth(root) << endl;
76
77     root = t.deleteNode(root, 6);
78     cout << "After deletion, Max Depth: " << t.maxDepth(root) << endl;
79
80     return 0;
81 }
```

Max Depth: 4  
Min Depth: 3  
After deletion, Max Depth: 4

=== Code Execution Successful ===

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

```
1  #include <iostream>
2  #include <climits>
3  using namespace std;
4
5  class Node {
6  public:
7      int data;
8      Node* left;
9      Node* right;
10
11     Node(int v) {
12         data = v;
13         left = right = nullptr;
14     }
15 };
16
17 bool isBST(Node* root, int minval, int maxval) {
18     if (root == nullptr) return true;
19     if (root->data <= minval || root->data >= maxval)
20         return false;
21     return isBST(root->left, minval, root->data) &&
22         isBST(root->right, root->data, maxval);
23 }
24
25 int main() {
26     Node* root = new Node(10);
27     root->left = new Node(5);
28     root->right = new Node(20);
29     root->left->left = new Node(2);
30     root->left->right = new Node(8);
31
32     if (isBST(root, INT_MIN, INT_MAX))
33         cout << "It IS a BST";
34     else
35         cout << "It is NOT a BST";
36
37     return 0;
38 }
```

It IS a BST

=== Code Exec

## 5. Implement heap sort (increasing/decreasing)

```
1  #include <iostream>
2  using namespace std;
3
4  void heapifyMax(int arr[], int n, int i) {
5      int largest = i;
6      int left = 2*i + 1;
7      int right = 2*i + 2;
8
9      if (left < n && arr[left] > arr[largest]) largest = left;
10     if (right < n && arr[right] > arr[largest]) largest = right;
11
12     if (largest != i) {
13         swap(arr[i], arr[largest]);
14         heapifyMax(arr, n, largest);
15     }
16 }
17
18 void heapifyMin(int arr[], int n, int i) {
19     int smallest = i;
20     int left = 2*i + 1;
21     int right = 2*i + 2;
22
23     if (left < n && arr[left] < arr[smallest]) smallest = left;
24     if (right < n && arr[right] < arr[smallest]) smallest = right;
25
26     if (smallest != i) {
27         swap(arr[i], arr[smallest]);
28         heapifyMin(arr, n, smallest);
29     }
30 }
31
32 void heapSortIncreasing(int arr[], int n) {
33     for (int i = n/2 - 1; i >= 0; i--)
34         heapifyMax(arr, n, i);
35
36     for (int i = n-1; i >= 0; i--) {
37         swap(arr[0], arr[i]);
38         heapifyMax(arr, i, 0);
39     }
40 }
41
42 void heapSortDecreasing(int arr[], int n) {
43     for (int i = n/2 - 1; i >= 0; i--)
44         heapifyMin(arr, n, i);
45
46     for (int i = n-1; i >= 0; i--) {
47         swap(arr[0], arr[i]);
48         heapifyMin(arr, i, 0);
49     }
50 }
51
52 int main() {
53     int arr[] = {3, 5, 2, 9, 1, 8, 10};
54     int n = 7;
55
56     cout << "Sorted (Increasing): ";
57     heapSortIncreasing(arr, n);
58     for (int i = 0; i < n; i++) cout << arr[i] << " ";
59
60     cout << "\nSorted (Decreasing): ";
61     heapSortDecreasing(arr, n);
62     for (int i = 0; i < n; i++) cout << arr[i] << " ";
63
64     return 0;
65 }
```

Sorted (Increasing): 1 2 3 5 8 9 10  
Sorted (Decreasing): 10 9 8 5 3 2 1

=== Code Execution Successful ===

## 6. Implement priority queues using heaps

```
1  #include <iostream>
2  using namespace std;
3
4  class PriorityQueue {
5  public:
6      int arr[100];
7      int size;
8
9      PriorityQueue() { size = 0; }
10
11     int parent(int i) { return (i - 1) / 2; }
12     int left(int i) { return 2*i + 1; }
13     int right(int i) { return 2*i + 2; }
14
15     void insert(int x) {
16         arr[size] = x;
17         int i = size;
18         size++;
19
20         while (i > 0 && arr[parent(i)] < arr[i]) {
21             swap(arr[i], arr[parent(i)]);
22             i = parent(i);
23         }
24     }
25
26     int getMax() {
27         if (size == 0) return -1;
28         return arr[0];
29     }
30
31     void heapify(int i) {
32         int largest = i;
33         int l = left(i);
34         int r = right(i);
35
36         if (l < size && arr[l] > arr[largest]) largest = l;
37         if (r < size && arr[r] > arr[largest]) largest = r;
38
39         if (largest != i) {
40             swap(arr[i], arr[largest]);
41             heapify(largest);
42         }
43     }
44
45     int extractMax() {
46         if (size <= 0) return -1;
47         if (size == 1) return arr[--size];
48
49         int root = arr[0];
50         arr[0] = arr[size - 1];
51         size--;
52         heapify(0);
53
54         return root;
55     }
56
57     bool isEmpty() {
58         return size == 0;
59     }
60 };
61
62 int main() {
63     PriorityQueue pq;
64
65     pq.insert(40);
66     pq.insert(10);
67     pq.insert(30);
68     pq.insert(50);
69     pq.insert(60);
70
71     cout << "Max element: " << pq.getMax() << endl;
72
73     cout << "Extracting elements: ";
74     while (!pq.isEmpty())
75         cout << pq.extractMax() << " ";
76     cout << endl;
77
78     return 0;
79 }
```

Max element: 60  
Extracting elements: 60 50 40 30 10

=== Code Execution Successful ===