NAME -AADITA GARG
ROLL NO – 1024030461
SUB GROUP – 2C32

DATA STRUCTURES AND ALGORITHMS ASSIGNMENT – 8 – BINARY SEARCH TREES

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

```
#include <iostream>
using namespace std;
struct Node {
   int data;
   Node* left;
   Node* right;
Node (int val){
   data = val;
   left = nullptr;
   right = nullptr;
}
};
void preorder (Node* root){
   if (root == nullptr)
   return ;
      cout << root -> data;
      preorder (root->left);
      preorder (root->right);

}
void inorder (Node* root){
   if (root==nullptr)
```

```
    return;
      inorder (root->left);
      cout << root->data;
 inorder (root->right); }
void postorder (Node* root ){
    if (root==nullptr)
    return;
    postorder (root->left);
    postorder (root->right);
    cout << root -> data;
}
int main ()
  { 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 << "Preorder Traversal: ";
    preorder(root);
    cout << endl;
    cout << "Inorder Traversal: ";
    inorder(root);
    cout << endl;
    cout << "Postorder Traversal: ";
    postorder(root);
    cout << endl; }
```

```
Preorder Traversal: 1245367
Inorder Traversal: 4251637
Postorder Traversal: 4526731
```

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

```
#include <iostream>
using namespace std;
struct Node {
   int data;
   Node* left;
   Node* right;

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

Node* insert (Node* root, int val){
if (root == nullptr)
{
   return new Node (val);
}
if (val<root->data){
   root -> left = insert (root->left,val);
}
else if (val> root-> data){
   root -> right = insert (root->right, val);
}
return root;
}

Node* search (Node* root, int key){
   if (root==nullptr || root->data == key){
      return root;
   }
   if (key<root->data){
      return search (root->left, key);}
```

```
    else if (key> root->data){
        return search (root->right, key);}
        return root;
}

Node* iterative (Node* root, int key){
    Node* current = root;
    while (current!=nullptr){
        if (key == current ->data)
         return current;
        else if (key<current->data)
         current = current -> left;
         else current = current -> right;
    }
    return nullptr;
}

int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);

    int key;
    cout << "Enter element to search: ";
    cin >> key;

    Node* result1 = search(root, key);
    if (result1)
        cout << "Recursive: Element " << key << " found in BST.\n";
```

```
    else
        cout << "Recursive: Element " << key << " NOT found in BST.\n";


    Node* result2 = iterative (root, key);
    if (result2)
        cout << "Non-Recursive: Element " << key << " found in BST.\n";
    else
        cout << "Non-Recursive: Element " << key << " NOT found in
BST.\n";

    return 0;
}
```

```
Enter element to search: 50
Recursive: Element 50 found in BST.
Non-Recursive: Element 50 found in BST.
```

(b) Maximum element of the BST

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;

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

Node* insert (Node* root, int val){
```

```
if (root == nullptr)
{
   return new Node (val);
}
if (val<root->data){
   root -> left = insert (root->left,val);
}
else if (val> root-> data){
   root -> right = insert (root->right, val);
}
return root;
}

Node* max (Node* root){
   if (root == nullptr)
   return nullptr;

   while (root -> right! = nullptr)
      root = root->right;
      return root;
}

int main() {
   Node* root = nullptr;
   root = insert(root, 50);
   insert(root, 30);
   insert(root, 70);
   insert(root, 20);
   insert(root, 40);
   insert(root, 60);
   insert(root, 80);

Node* maxNode = max(root);
if (maxNode)
   cout << "Maximum element in BST: " << maxNode->data << endl;
```

```
else
   cout << "BST is empty.\n";


return 0;
}
```

```
Maximum element in BST: 80
```

(C) Minimum element of the BST

```
#include <iostream>
using namespace std;
struct Node {
   int data;
   Node* left;
   Node* right;

Node (int val){
   data = val;
   left = right = nullptr;
}
};
Node* insert (Node* root, int val){
if (root == nullptr)
{
   return new Node (val);
}
if (val<root->data){
   root -> left = insert (root->left,val);
}
else if (val> root-> data){
   root -> right = insert (root->right, val);
```

```
}
return root;
}

Node* min (Node* root){
   if (root == nullptr)
   return nullptr;

   while (root->left!=nullptr)
      root = root->left;
      return root;

}

int main() {
   Node* root = nullptr;
   root = insert(root, 50);
   insert(root, 30);
   insert(root, 70);
   insert(root, 20);
   insert(root, 40);
   insert(root, 60);
   insert(root, 80);
Node* minNode = min(root);
if (minNode)
   cout << "Minimum element in BST: " << minNode->data << endl;
else
   cout << "BST is empty.\n";
return 0;
}
```

```
Minimum element in BST: 20
```

(D) In-order successor 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 = nullptr;
}
};

Node* insert (Node* root, int val){
if (root == nullptr)
{
   return new Node (val);
}
if (val<root->data){
   root -> left = insert (root->left,val);
}
else if (val> root-> data){
   root -> right = insert (root->right, val);
}
return root;
}

Node* min (Node* root){
   if (root == nullptr)
   return nullptr;

   while (root->left!=nullptr)
      root = root->left;
      return root;
```

```
}

Node* successor (Node* root, Node* target){
   if (target == nullptr)
   return nullptr;

   if (target -> right!= nullptr)
   return min(target->right);

   Node* successor = nullptr;
   Node* ancestor = root;
   while (ancestor!= nullptr){
      if (target->data < ancestor->data){
         successor = ancestor;
         ancestor = ancestor -> left;
      }
      else if (target -> data > ancestor -> data){
         ancestor = ancestor -> right;
      }
      else break;
   }
   return successor;
}

int main() {
   Node* root = nullptr;
   root = insert(root, 50);
   insert(root, 30);
   insert(root, 70);
   insert(root, 20);
   insert(root, 40);
   insert(root, 60);
   insert(root, 80);
   Node* target = root->left;
   Node* succ = successor(root, target);
```

```
    if (succ)
        cout << "Inorder Successor of " << target->data << " is " << succ-
>data << endl;
    else
        cout << "No Inorder Successor exists for " << target->data <<
endl;


return 0;
}
```

```
Inorder Successor of 30 is 40
```

(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 = nullptr;
    }
};

Node* insert(Node* root, int val) {
    if (root == nullptr)
        return new Node(val);
```

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

    return root;
}

Node* min(Node* root) {
    while (root && root->left != nullptr)
        root = root->left;
    return root;
}

Node* max(Node* root) {
    while (root && root->right != nullptr)
        root = root->right;
    return root;
}

Node* predecessor(Node* root, Node* target) {
    if (!target) return nullptr;

    if (target->left)
        return max(target->left);

    Node* pred = nullptr;
    Node* ancestor = root;
    while (ancestor) {
        if (target->data > ancestor->data) {
            pred = ancestor;
            ancestor = ancestor->right;
        } else if (target->data < ancestor->data) {
            ancestor = ancestor->left;
```

```
        } else {
            break;
        }
    }
    return pred;
}

int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);

    Node* target = root->left;
    Node* pred = predecessor(root, target);

    if (pred)
        cout << "Inorder predecessor of " << target->data << " is " <<
pred->data << endl;
    else
        cout << "No Inorder Predecessor exists for " << target->data <<
endl;

    return 0;
}
```

```
Inorder predecessor of 30 is 20
```

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

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

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

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


// (A) Insert a node in BST
Node* insert(Node* root, int val) {
    if (root == nullptr) {
        return new Node(val);
    }
    if (val < root->data) {
        root->left = insert(root->left, val);
    } else if (val > root->data) {
        root->right = insert(root->right, val);
    } else {
        cout << "Duplicate value " << val << " not allowed." << endl;
    }
    return root;
}

Node* findMin(Node* root) {
    while (root && root->left != nullptr)
        root = root->left;
    return root;
```

```
}
```

// (B) Delete a node in a BST

```
Node* deleteNode(Node* root, int val) {
   if (!root) return nullptr;

   if (val < root->data) {
      root->left = deleteNode(root->left, val);
   } else if (val > root->data) {
      root->right = deleteNode(root->right, val);
   } else {
      if (!root->left) {
         Node* temp = root->right;
         delete root;
         return temp;
      } else if (!root->right) {
         Node* temp = root->left;
         delete root;
         return temp;
      } else {
         Node* temp = findMin(root->right);
         root->data = temp->data;
         root->right = deleteNode(root->right, temp->data);
      }
   }
   return root;
}
```

// (C) Maximum depth of BST

```
int maxDepth(Node* root) {
   if (!root) return 0;
   int leftDepth = maxDepth(root->left);
   int rightDepth = maxDepth(root->right);
```

```
    return 1 + max(leftDepth, rightDepth);
}
```

// (d) Minimum depth of BST

```
int minDepth(Node* root) {
    if (!root) return 0;

    if (!root->left) return 1 + minDepth(root->right);
    if (!root->right) return 1 + minDepth(root->left);

    return 1 + min(minDepth(root->left), minDepth(root->right));
}

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

int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);

    cout << "Inorder traversal: ";
    inorder(root);
    cout << endl;
```

```
    // Delete element
    root = deleteNode(root, 70);
    cout << "After deleting 70: ";
    inorder(root);
    cout << endl;

    // Maximum and minimum depth
    cout << "Maximum depth of BST: " << maxDepth(root) << endl;
    cout << "Minimum depth of BST: " << minDepth(root) << endl;

    return 0;
}
```

```
Inorder traversal: 20 30 40 50 60 70 80
After deleting 70: 20 30 40 50 60 80
Maximum depth of BST: 3
Minimum depth of BST: 3
```

(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 = nullptr;
```

```
    }
};

Node* insert(Node* root, int val) {
    if (!root) return new Node(val);
    if (val < root->data) root->left = insert(root->left, val);
    else if (val > root->data) root->right = insert(root->right, val);
    return root;
}

bool isBSTInorder(Node* root, Node*& prev) {
    if (root == nullptr) return true;

    if (!isBSTInorder(root->left, prev)) return false;

    if (prev != nullptr && root->data <= prev->data) return false;
    prev = root;

    return isBSTInorder(root->right, prev);
}

bool isBST(Node* root) {
    Node* prev = nullptr;
    return isBSTInorder(root, prev);
}

int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);
```

```
    if (isBST(root))
        cout << "The tree is a BST." << endl;
    else
        cout << "The tree is NOT a BST." << endl;

    return 0;
}
```

```
The tree is a BST.
```

(5) Implement Heapsort (Increasing/Decreasing order).

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

// MAX HEAPIFY
void maxHeapify(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]);
        maxHeapify(arr, n, largest);
    }
}
```

```
// MIN HEAPIFY
void minHeapify(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]);
        minHeapify(arr, n, smallest);
    }
}

// HEAPSORT INCREASING (max-heap)
void heapSortIncreasing(int arr[], int n) {
    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        maxHeapify(arr, n, i);

    // Extract elements
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        maxHeapify(arr, i, 0);
    }
}

// HEAPSORT DECREASING (min-heap)
void heapSortDecreasing(int arr[], int n) {
    // Build min heap
```

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

    // Extract elements
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        minHeapify(arr, i, 0);
    }
}

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

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

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

    if (choice == 1)
        heapSortIncreasing(arr, n);
    else
        heapSortDecreasing(arr, n);

    cout << "\nSorted Array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}
```

```
Enter number of elements: 5
Enter elements: 8 2 4 1 5

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

Sorted Array: 1 2 4 5 8
```

```
Enter number of elements: 5
Enter elements: 8 2 4 1 5

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

Sorted Array: 8 5 4 2 1
```

(6) Implement priority queues using heaps.

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

class MaxHeap {
   int arr[100];
   int size;

public:
   MaxHeap() { size = 0; }

   int parent(int i) { return (i - 1) / 2; }
```

```
int left(int i) { return 2 * i + 1; }
int right(int i) { return 2 * i + 2; }

// INSERT
void insert(int key) {
    arr[size] = key;
    int i = size;
    size++;

    while (i != 0 && arr[parent(i)] < arr[i]) {
        swap(arr[i], arr[parent(i)]);
        i = parent(i);
    }
}

// GET MAX
int getMax() {
    if (size == 0) {
        cout << "Heap is empty\n";
        return -1;
    }
    return arr[0];
}

// HEAPIFY
void heapify(int i) {
    int largest = i;
    int l = left(i);
    int r = right(i);

    if (l < size && arr[l] > arr[largest])
        largest = l;

    if (r < size && arr[r] > arr[largest])
        largest = r;
```

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

    // EXTRACT MAX
    int extractMax() {
        if (size <= 0)
            return -1;
        if (size == 1)
            return arr[--size];

        int root = arr[0];
        arr[0] = arr[size - 1];
        size--;

        heapify(0);
        return root;
    }

    // DISPLAY
    void display() {
        for (int i = 0; i < size; i++)
            cout << arr[i] << " ";
        cout << endl;
    }
};


int main() {
    MaxHeap pq;

    while (true) {
```

```
        cout << "Enter choice: ";

        int choice, key;
        cin >> choice;

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

            case 2:
                cout << "Maximum element: " << pq.getMax() << endl;
                break;

            case 3:
                cout << "Extracted: " << pq.extractMax() << endl;
                break;

            case 4:
                pq.display();
                break;

            case 5:
                return 0;

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

```
Enter choice: 1
Enter value: 3
Enter choice: 1
Enter value: 2
Enter choice: 1
Enter value: 7
Enter choice: 1
Enter value: 8
Enter choice: 2
Maximum element: 8
Enter choice: 3
Extracted: 8
Enter choice: 4
7 2 3
Enter choice: 6
Invalid choice
```