Republic of the Philippines
BICOL UNIVERSITY POLANGUI
Polangui, Albay

Members:
ChristineJoy L. Bande
Magdaline S. Infante
Christine Jade P. Ondis
Course, Year& Section: BSIS-2A

Project Overview:

This code provides an implementation for Binary Trees, Binary Search Trees (BSTs), and Heap operations. It includes functions to perform basic operations like insertion, deletion, search, and traversal for trees and heaps.

Description of Each Functionality

Includes and using namespace std

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;
```

- **#include <iostream>**: Enables input/output operations using cin, cout.
- **#include <vector>**: Used for dynamic arrays.
- **#include <queue>**: Allows the use of queues (FIFO) and priority queues.
- **#include <algorithm>**: Provides utility functions like swap.
- **using namespace std;**: Avoids prefixing std:: for standard functions like cout.

Node Structure

```cpp

// Node structure
struct Node {
    int data;
    Node *left, *right;
    Node(int data) : data(data), left(nullptr), right(nullptr) {}
};
```

- **data**: Stores the value of the node.
- **left, right**: Pointers to the left and right children.
- **Constructor**: Initializes data and sets left and right to nullptr.

Binary Tree Operations

Insert a node

```cpp

// Function to insert a node into a Binary Tree
void insertBinaryTree(Node*& root, int data) {
    if (root == nullptr) {
        root = new Node(data);
        return;
    }
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* temp = q.front();
        q.pop();
        if (!temp->left) {
            temp->left = new Node(data);
            break;
        } else
            q.push(temp->left);

        if (!temp->right) {
            temp->right = new Node(data);
            break;
        } else
            q.push(temp->right);
    }
}
```

- Inserts a new node in the first available position using level-order traversal.
- If root is nullptr, create a new root node.
- Otherwise, use a queue to traverse the tree level by level and insert the node in the first available position.

Tree Traversals

Pre-order Traversal: Root → Left → Right

```
// Preorder Traversal for Binary Tree
void preorderBinaryTree(Node* root) {
    if (root) {
        cout << root->data << " ";
        preorderBinaryTree(root->left);
        preorderBinaryTree(root->right);
    }
}
```

In-order Traversal: Left → Root → Right

```
// Inorder traversal for Binary Tree (Sorted order)
void inorderBinaryTree(Node* root) {
    if (root) {
        inorderBinaryTree(root->left); // Recurse on left subtree
        cout << root->data << " "; // Visit the root
        inorderBinaryTree(root->right); // Recurse on right subtree
    }
}
```

Post-order Traversal: Left → Right → Root

```
// Postorder Traversal for Binary Tree
void postorderBinaryTree(Node* root) {
    if (root) {
        postorderBinaryTree(root->left);
        postorderBinaryTree(root->right);
        cout << root->data << " ";
    }
}
```

Search in a Binary Tree

```
// Search in Binary Tree
Node* searchBinaryTree(Node* root, int data) {
    if (!root) return nullptr;
    if (root->data == data) return root;
    Node* leftSearch = searchBinaryTree(root->left, data);
    if (leftSearch) return leftSearch;
    return searchBinaryTree(root->right, data);
}
```

- Searches for a node with the given data using recursion.

Delete a Node

Delete the deepest node

```
// Delete the deepest node
void deleteDeepestNode(Node* root, Node* deepNode) {
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* temp = q.front();
        q.pop();
        if (temp->left) {
            if (temp->left == deepNode) {
                delete temp->left;
                temp->left = nullptr;
                return;
            }
            q.push(temp->left);
        }
        if (temp->right) {
            if (temp->right == deepNode) {
                delete temp->right;
                temp->right = nullptr;
                return;
            }
            q.push(temp->right);
        }
    }
}
```

Delete a specific node

```
// Delete a node in Binary Tree
void deleteNodeBinaryTree(Node*& root, int data) {
    if (!root) return;
    if (!root->left && !root->right) {
        if (root->data == data) {
            delete root;
            root = nullptr;
        }
        return;
    }
    queue<Node*> q;
    q.push(root);
    Node* targetNode = nullptr;
    Node* temp = nullptr;
    while (!q.empty()) {
        temp = q.front();
        q.pop();
        if (temp->data == data) targetNode = temp;
        if (temp->left) q.push(temp->left);
        if (temp->right) q.push(temp->right);
    }
    if (targetNode) {
        targetNode->data = temp->data;
        deleteDeepestNode(root, temp);
    }
}
```

- Finds the deepest node and swaps its value with the node to delete.
- Deletes the deepest node.

Binary Search Tree (BST) Operations

Insert a Node

```
// Insert into Binary Search Tree
void insertBST(Node*& root, int data) {
    if (!root) {
        root = new Node(data);
        return;
    }
    if (data < root->data)
        insertBST(root->left, data);
    else
        insertBST(root->right, data);
}
```

- Places smaller values in the left subtree and larger values in the right subtree.

Delete a Node

```
// Delete a node in BST
Node* deleteNodeBST(Node* root, int data) {
    if (!root) return nullptr;
    if (data < root->data)
        root->left = deleteNodeBST(root->left, data);
    else if (data > root->data)
        root->right = deleteNodeBST(root->right, data);
    else {
        if (!root->left) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (!root->right) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
```

```
        Node* successor = root->right;
        while (successor->left) successor = successor->left;
        root->data = successor->data;
        root->right = deleteNodeBST(root->right, successor->data);
    }
    return root;
}
```

- Handles three cases: node with no child, one child, or two children.

Heap Operations

Heapify Function

```
// Custom heapify function
void heapify(vector<int>& arr, int n, int i) {
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // Left child
    int right = 2 * i + 2; // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected subtree
        heapify(arr, n, largest);
    }
}
```

- This function is used to maintain the heap property (for a max-heap: every parent node must be larger than its children).
- arr: The array representing the heap.
- n: Size of the heap (or sub-heap).
- i: The current root node being checked.

Key Steps:

1. Assume the current node (i) is the largest.
2. Check its left (2*i+1) and right (2*i+2) children, updating largest if a child is larger.
3. If the largest value is not at the root, swap and recursively call heapify.

Build Heap

```
}

void buildHeap(vector<int>& arr) {
    int n = arr.size();
    // Index of the last non-leaf node
    int startIdx = (n / 2) - 1;

    // Perform reverse level order traversal
    // from the last non-leaf node and heapify each node
    for (int i = startIdx; i >= 0; i--) {
        heapify(arr, n, i);
    }
}
```

- Non-leaf nodes start at index (n/2 - 1).
- Heapify every node from the last non-leaf node to the root.

Heapify Operations

```cpp

void heapifyOperations() {
    vector<int> values;
    int value;
    cout << "Enter values to heapify (enter -1 to stop): ";
    while (cin >> value && value != -1) {
        values.push_back(value);
    }

    buildHeap(values);

    cout << "Heapified array: ";
    for (int v : values) {
        cout << v << " ";
    }
    cout << endl;
}
```

- Reads input values, builds a max-heap, and prints the heapified array.
- Input values are collected in a vector.
- buildHeap is called to convert the array into a max-heap.
- The resulting array is printed.

Min-Heap and Max-Heap Operations

Min-Heap

```cpp
// Min-Heap Operations
void minHeapOperations() {
    priority_queue<int, vector<int>, greater<int>> minHeap;
    int value;
    cout << "Enter values for Min-Heap (enter -1 to stop): ";
    while (cin >> value && value != -1) {
        minHeap.push(value);
    }
    cout << "Min-Heap elements: ";
    while (!minHeap.empty()) {
        cout << minHeap.top() << " ";
        minHeap.pop();
    }
    cout << endl;
}
```

Max-Heap

```cpp

// Max-Heap Operations
void maxHeapOperations() {
    priority_queue<int> maxHeap;
    int value;
    cout << "Enter values for Max-Heap (enter -1 to stop): ";
    while (cin >> value && value != -1) {
        maxHeap.push(value);
    }
    cout << "Max-Heap elements: ";
    while (!maxHeap.empty()) {
        cout << maxHeap.top() << " ";
        maxHeap.pop();
    }
    cout << endl;
}
```

- Min-Heap: Maintains the smallest element at the root.
- Max-Heap: Maintains the largest element at the root.
- priority_queue is used to efficiently manage heaps in C++.
- greater<int> creates a min-heap; the default is a max-heap.

Main Function

```
int main() {
    Node* binaryTreeRoot = nullptr;
    Node* bstRoot = nullptr;
    int input;
```

- Perform operations for binary trees, BSTs, and heaps interactively.
- Users input data for each structure and view the results.

Binary Tree Operations

Building the Binary Tree

```
    // Binary Tree Operations
    cout << "Build your Binary Tree:\n";
    while (cout << "Enter node data (-1 to stop): ", cin >> input, input != -1) {
        insertBinaryTree(binaryTreeRoot, input);
    }
```

- This loop prompts the user to enter node data for the Binary Tree until the user enters -1.
- insertBinaryTree(binaryTreeRoot, input) is a function (not defined in the provided code) that inserts the input value into the Binary Tree.

Traversal of the Binary Tree

```
    cout << "In-order (Binary Tree): ";
    inorderBinaryTree(binaryTreeRoot);
    cout << endl;
    cout << "Preorder (Binary Tree): ";
    preorderBinaryTree(binaryTreeRoot);
    cout << "\nPostorder (Binary Tree): ";
    postorderBinaryTree(binaryTreeRoot);
    cout << endl;
```

These are three different tree traversal methods:

- In-order: Left subtree, root, right subtree.
- Preorder: Root, left subtree, right subtree.
- Postorder: Left subtree, right subtree, root.

The functions inorderBinaryTree(), preorderBinaryTree(), and postorderBinaryTree() would perform the respective traversals on the Binary Tree and print the results.

Searching for a Value in the Binary Tree

```
    // Search for a value in the binary tree
    cout << "Enter value to search in Binary Tree: ";
    cin >> input;
    Node* result = searchBinaryTree(binaryTreeRoot, input);
    if (result) {
        cout << "Value found in the Binary Tree!\n";
    } else {
        cout << "Value not found in the Binary Tree.\n";
    }
```

- The user is prompted to enter a value to search for in the Binary Tree.
- searchBinaryTree(binaryTreeRoot, input) searches the Binary Tree for the value entered. If the value is found, a message is displayed; otherwise, a different message is shown.

Deleting a Value from the Binary Tree

```
    // Delete a value from the binary tree
    cout << "Enter value to delete from Binary Tree: ";
    cin >> input;
    deleteNodeBinaryTree(binaryTreeRoot, input);

    cout << "Binary Tree after deletion: ";
    preorderBinaryTree(binaryTreeRoot);
    cout << endl;
```

- The user is prompted to enter a value to delete from the Binary Tree.
- deleteNodeBinaryTree(binaryTreeRoot, input) deletes the node with the specified value from the Binary Tree.
- After deletion, the Binary Tree is displayed using a preorder traversal.

Binary Search Tree (BST) Operations

Building the BST

```
// BST Operations
cout << "\nBuild your Binary Search Tree:\n";
while (cout << "Enter node data (-1 to stop): ", cin >> input, input != -1) {
    insertBST(bstRoot, input);
}
```

- Similar to the Binary Tree, the user is prompted to enter data to build the Binary Search Tree (BST) until -1 is entered.
- insertBST(bstRoot, input) inserts the input into the BST.

Traversal of the BST

```
cout << "Inorder (BST): ";
inorderBST(bstRoot);
cout << "\nPreorder (BST): ";
preorderBST(bstRoot);
cout << "\nPostorder (BST): ";
postorderBST(bstRoot);
cout << endl;
```

- The same traversal methods are used for the BST (inorderBST(), preorderBST(), postorderBST()), displaying the results for the BST.

Searching for a Value in the BST

```
// Search for a value in the BST
cout << "Enter value to search in BST: ";
cin >> input;
result = searchBST(bstRoot, input);
if (result) {
    cout << "Value found in the BST!\n";
} else {
    cout << "Value not found in the BST.\n";
}
```

- The user is prompted to enter a value to search for in the BST.
- searchBST(bstRoot, input) performs the search, and if the value is found, a message is displayed.

Deleting a Value from the BST

```
// Delete a value from the BST
cout << "Enter value to delete from BST: ";
cin >> input;
bstRoot = deleteNodeBST(bstRoot, input);
cout << "BST after deletion: ";
preorderBST(bstRoot);
cout << endl;
```

- The user is prompted to enter a value to delete from the BST.
- deleteNodeBST(bstRoot, input) deletes the node with the specified value.
- After deletion, the BST is displayed using a preorder traversal.

Heap Operations

```
    // heapify/Heap Operations
    heapifyOperations();
    minHeapOperations();
    maxHeapOperations();
```

These functions are placeholders for heap-related operations. Typically, these could involve:

- heapifyOperations(): Operations related to transforming an array into a heap.
- minHeapOperations(): Operations related to maintaining a Min-Heap (e.g., inserting or deleting nodes).
- maxHeapOperations(): Operations related to maintaining a Max-Heap (e.g., inserting or deleting nodes).

Return Statement

```
    return 0;
}
```

- The program ends by returning 0, signaling successful execution.

Sample Output

```
C:\Users\TinTin\OneDrive\De:
Build your Binary Tree:
Enter node data (-1 to stop): 1
Enter node data (-1 to stop): 2
Enter node data (-1 to stop): 3
Enter node data (-1 to stop): 4
Enter node data (-1 to stop): 5
Enter node data (-1 to stop): -1
In-order (Binary Tree): 4 2 5 1 3
Preorder (Binary Tree): 1 2 4 5 3
Postorder (Binary Tree): 4 5 2 3 1
Enter value to search in Binary Tree: 2
Value found in the Binary Tree!
Enter value to delete from Binary Tree: 3
Binary Tree after deletion: 1 2 4 5

Build your Binary Search Tree:
Enter node data (-1 to stop): 1
Enter node data (-1 to stop): 2
Enter node data (-1 to stop): 3
Enter node data (-1 to stop): 4
Enter node data (-1 to stop): 5
Enter node data (-1 to stop): -1
Inorder (BST): 1 2 3 4 5
Preorder (BST): 1 2 3 4 5
Postorder (BST): 5 4 3 2 1
Enter value to search in BST: 4
Value found in the BST!
Enter value to delete from BST: 2
BST after deletion: 1 3 4 5
Enter values to heapify (enter -1 to stop): 1
2
3
4
5
-1
Heapified array: 5 4 3 1 2
Enter values for Min-Heap (enter -1 to stop): 1
2
3
4
5
-1
Min-Heap elements: 1 2 3 4 5
Enter values for Max-Heap (enter -1 to stop): 1
2
3
4
5
-1
Max-Heap elements: 5 4 3 2 1

Process returned 0 (0x0)   execution time : 50.182 s
Press any key to continue.
```