Data Structure and Algorithm Final Project

Aloysious Irish O. Portugal BSCS 2-A

Neo L. Gangawan

Code:

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

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

    Node(int value) {
        key = value;
        left = right = nullptr;
    }
};
```

```
class BinaryTree {
private:
    Node* root;

    void inorder(Node* node) {
        if (node != nullptr) {
            inorder(node->left);
            cout << node->key << " ";
            inorder(node->right);
        }
    }

    void preorder(Node* node) {
        if (node != nullptr) {
            cout << node->key << " ";
            preorder(node->left);
            preorder(node->right);
        }
    }

    void postorder(Node* node) {
        if (node != nullptr) {
            postorder(node->left);
```

```
        postorder(node->right);
        cout << node->key << " ";
    }
}

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

Node* deleteNode(Node* node, int key) {
    if (node == nullptr) return node;
    if (key < node->key) node->left = deleteNode(node->left, key);
    else if (key > node->key) node->right = deleteNode(node->right,
key);
    else {
        if (node->left == nullptr) {
            Node* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
```

```
            Node* temp = node->left;
            delete node;
            return temp;
        }
        Node* temp = findMin(node->right);
        node->key = temp->key;
        node->right = deleteNode(node->right, temp->key);
    }
    return node;
}

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

public:
BinaryTree() { root = nullptr; }

void insert(int key) { root = insert(root, key); }

bool search(int key) {
```

```
        Node* node = root;
        while (node) {
            if (node->key == key) return true;
            node = (key < node->key) ? node->left : node->right;
        }
        return false;
    }

    void deleteKey(int key) { root = deleteNode(root, key); }

    void inorder() { inorder(root); cout << endl; }
    void preorder() { preorder(root); cout << endl; }
    void postorder() { postorder(root); cout << endl; }
};

class BinarySearchTree {
private:
    Node* root;

    Node* insert(Node* node, int key) {
        if (node == nullptr) return new Node(key);
        if (key < node->key) node->left = insert(node->left, key);
        else if (key > node->key) node->right = insert(node->right, key);
```

```
    return node;
}

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

Node* deleteNode(Node* node, int key) {
    if (node == nullptr) return node;
    if (key < node->key) node->left = deleteNode(node->left, key);
    else if (key > node->key) node->right = deleteNode(node->right,
key);
    else {
        if (node->left == nullptr) {
            Node* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            Node* temp = node->left;
            delete node;
            return temp;
```

```
            }
            Node* temp = findMin(node->right);
            node->key = temp->key;
            node->right = deleteNode(node->right, temp->key);
        }
        return node;
    }

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

    void inorder(Node* node) {
        if (node != nullptr) {
            inorder(node->left);
            cout << node->key << " ";
            inorder(node->right);
        }
    }

public:
    BinarySearchTree() { root = nullptr; }
```

```
    void insert(int key) { root = insert(root, key); }
    bool search(int key) { return search(root, key) != nullptr; }
    void deleteKey(int key) { root = deleteNode(root, key); }
    void inorder() { inorder(root); cout << endl; }
};

// Max Heap and Min Heap functions
void insert(vector<int>& heap, int value) {
    heap.push_back(value);
}

void heapifyMax(vector<int>& heap, int index) {
    int parent = (index - 1) / 2;
    while (index > 0 && heap[index] > heap[parent]) {
        swap(heap[index], heap[parent]);
        index = parent;
        parent = (index - 1) / 2;
    }
}

void heapifyMin(vector<int>& heap, int index) {
    int parent = (index - 1) / 2;
```

```
    while (index > 0 && heap[index] < heap[parent]) {
        swap(heap[index], heap[parent]);
        index = parent;
        parent = (index - 1) / 2;
    }
}

void insertMaxHeap(vector<int>& heap, int value) {
    heap.push_back(value);
    heapifyMax(heap, heap.size() - 1);
}

void insertMinHeap(vector<int>& heap, int value) {
    heap.push_back(value);
    heapifyMin(heap, heap.size() - 1);
}

void print(const vector<int>& heap) {
    for (int i = 0; i < heap.size(); i++) {
        cout << heap[i] << " ";
    }
    cout << endl;
}
```

```
int main() {
    vector<int> maxHeap; // Max heap
    vector<int> minHeap; // Min heap
    BinarySearchTree bst;
    BinaryTree bt;
    int choice, key, n;
    vector<int> array;

    cout << "Enter the number of elements to insert into the Binary Tree and BST: ";
    cin >> n;

    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++) {
        cin >> key;
        array.push_back(key);
    }

    // Insert elements into Binary Tree and BST
    for (int num : array) {
        bt.insert(num);
        bst.insert(num);
```

```
}

// Insert elements into heaps
for (int value : array) {
    insertMaxHeap(maxHeap, value);
    insertMinHeap(minHeap, value);
}

cout << "Initial Max Heap: ";
print(maxHeap);

cout << "Initial Min Heap: ";
print(minHeap);

cout << "Binary Tree and Binary Search Tree setup complete.\n";

do {
    cout << "\nChoose an operation:\n";
    cout << "1. Insert into Binary Tree\n";
    cout << "2. Inorder Traversal (Binary Tree)\n";
    cout << "3. Preorder Traversal (Binary Tree)\n";
    cout << "4. Postorder Traversal (Binary Tree)\n";
    cout << "5. Search in Binary Search Tree\n";
```

```
cout << "6. Delete from Binary Search Tree\n";
cout << "7. Display Max Heap\n";
cout << "8. Display Min Heap\n";
cout << "9. Exit\n";
cout << "Enter your choice: ";
cin >> choice;

switch (choice) {
case 1:
    cout << "Enter a number to insert into Binary Tree: ";
    cin >> key;
    bt.insert(key);
    cout << key << " inserted into Binary Tree.\n";
    break;
case 2:
    cout << "Inorder Traversal (Binary Tree): ";
    bt.inorder();
    break;
case 3:
    cout << "Preorder Traversal (Binary Tree): ";
    bt.preorder();
    break;
case 4:
```

```
    cout << "Postorder Traversal (Binary Tree): ";
    bt.postorder();
    break;
case 5:
    cout << "Enter a number to search in BST: ";
    cin >> key;
    if (bst.search(key)) cout << key << " found in BST.\n";
    else cout << key << " not found in BST.\n";
    break;
case 6:
    cout << "Enter a number to delete from BST: ";
    cin >> key;
    bst.deleteKey(key);
    cout << key << " deleted from BST.\n";
    break;
case 7:
    cout << "Max Heap: ";
    print(maxHeap);
    break;
case 8:
    cout << "Min Heap: ";
    print(minHeap);
    break;
```

```
        case 9:
            cout << "Exiting...\n";
            break;
        default:
            cout << "Invalid choice! Please try again.\n";
        }
    } while (choice != 9);


    return 0;
}
```

Overview of the Project:

This C++ program implements a Binary Tree, Binary Search Tree (BST) along with Max Heap and Min Heap data structures. The program allows users to interact with these structures through a simple menu interface, where they can perform common operations such as insertion, search, traversal, and deletion.

How to run the code:

Initialization:

- The program first prompts the user to enter the number of elements they want to insert into the BST and heaps.
- It then takes input for these elements and inserts them into both the BST and the heaps.

Displaying Initial Heaps:

- After inserting the elements, it displays the current state of the Max Heap and Min Heap.

Menu Operations:

- The user is presented with a menu to perform operations. For example, they can select to perform an inorder traversal of the BST:

Search and Delete Operations:

- The user can search for a key or delete a key from the BST. After deletion, the program updates the BST and displays the relevant traversal.
- The user can also delete a key and observe the changes in the tree or heap.

Exit:

- The user selects option "9" to exit the program, and the program terminates.

Description of each functionality:

1. Binary Search Tree (BST)

- Structure: The BST is a tree where each node has a key, a pointer to the left child, and a pointer to the right child. It follows the property that:
  - Left child values are smaller than the parent node.
  - Right child values are greater than the parent node.
- Operations:
  - Insert: Inserts a new value in the appropriate position in the tree, maintaining the BST property.
  - Search: Searches for a specific value in the tree. It returns whether the value exists in the tree or not.
  - Delete: Deletes a node from the tree. It handles three cases:
    1. Node has no children (leaf node).
    2. Node has one child.

3. Node has two children (replaces node with the smallest value in the right subtree).
    - Traversal: The program supports three types of traversal:
        - Inorder: Left subtree → Node → Right subtree (used to display the tree in sorted order).
        - Preorder: Node → Left subtree → Right subtree.
        - Postorder: Left subtree → Right subtree → Node.

2. Heaps (Max Heap and Min Heap)

- **Max Heap**: The value of each parent node is greater than or equal to its children. This is used when we need quick access to the largest element.
- **Min Heap**: The value of each parent node is less than or equal to its children. This is used when we need quick access to the smallest element.
- **Heap Operations**:
    - **Insert**: Inserts a new element while maintaining the heap property (Max Heap or Min Heap). This is done by "heapifying" the tree starting from the newly inserted node to ensure that the heap property is satisfied.
    - **Heapify**: A helper function that moves a node upwards in the heap until the heap property is maintained.
- **Heap Display**: The program provides the ability to print both **Max Heap** and **Min Heap**.

3. Program Flow

- **Input and Initialization**:
    - First, the program asks the user for a number of elements to insert into the **Binary Search Tree (BST)**.
    - The user enters the elements, which are then inserted into the BST.
    - Afterward, the program inserts the same elements into both the **Max Heap** and **Min Heap**.

- **Operations Menu**: After the initial setup, the program enters a loop where