## **Lab 11**

**Implementation of Binary Search Trees**

#include <iostream>

#include <fstream>

using namespace std;

template <class ItemType>

struct TreeNode {

ItemType info; // Data member

TreeNode<ItemType>\* left; // Pointer to left child

TreeNode<ItemType>\* right; // Pointer to right child

};

template <class ItemType>

class TreeType {

public:

TreeType(); // Constructor

~TreeType(); // Destructor

bool IsEmpty() const;

void InsertItem(ItemType item);

void RetrieveItem(ItemType& item, bool& found);

void PrintTree(ofstream& outFile);

private:

TreeNode<ItemType>\* root;

//This is the root of the binary search tree. It is a pointer to the topmost TreeNode in the tree. All operations (insertion, deletion, retrieval) are initiated from this root pointer.

void InsertHelper(TreeNode<ItemType>\*& ptr, ItemType item);

//A helper function for the public InsertItem function.Recursively navigates the tree to find the correct position to insert the item. The \*& ensures that changes to the pointer (e.g., creating a new node) affect the tree's structure.

void RetrieveHelper(TreeNode<ItemType>\* ptr, ItemType& item, bool& found);

// A helper function for the public RetrieveItem function.

Recursively searches for the item in the tree.

Sets found to true if the item is present, otherwise false.

Uses the pointer ptr to traverse the tree but does not modify the structure.

void PrintHelper(TreeNode<ItemType>\* ptr, ofstream& outFile);

// A helper function for the public PrintTree function.

Recursively performs an in-order traversal (left, root, right) to output tree elements to the provided ofstream.

Operates on ptr without modifying the tree structure

void DestroyHelper(TreeNode<ItemType>\* ptr);

};

//A helper function for the destructor ~TreeType.

Recursively deallocates memory for all nodes in the tree to avoid memory leaks.

Uses a post-order traversal (left, right, root) to delete nodes safely.

template <class ItemType>

TreeType<ItemType>::TreeType() {

root = nullptr;

}

// Purpose: Initializes an empty binary search tree.

Implementation: The root pointer is set to nullptr, indicating the tree has no nodes at the start.

Default State: After calling this constructor, the tree is empty.

template <class ItemType>

bool TreeType<ItemType>::IsEmpty() const {

return root == nullptr;

}

// Purpose: Checks if the tree is empty.

Implementation:

If root is nullptr, the tree has no nodes, and the method returns true.

Otherwise, it returns false.

Const Method: The const keyword ensures that this method does not modify the state of the TreeType object.

template <class ItemType>

void TreeType<ItemType>::InsertItem(ItemType item) {

InsertHelper(root, item);

}

//**Purpose:** Adds a new item to the tree.

**Implementation:**

* Calls the private helper function InsertHelper to perform the recursive insertion.
* Starts the recursive insertion process from the root.

template <class ItemType>

void TreeType<ItemType>::InsertHelper(TreeNode<ItemType>\*& ptr, ItemType item) {

if (ptr == nullptr) {

ptr = new TreeNode<ItemType>;

ptr->info = item;

ptr->left = nullptr;

ptr->right = nullptr;

} else if (item < ptr->info) {

InsertHelper(ptr->left, item);

} else if (item > ptr->info) {

InsertHelper(ptr->right, item);

}

}

// **Base Case: if (ptr == nullptr)**

* **Condition:** Checks if the current pointer (ptr) is nullptr.
* **Action:** If true, this means we’ve reached an empty spot in the tree where the new node can be inserted.
  + Allocates memory for a new TreeNode.
  + Sets its value to item (ptr->info = item).
  + Initializes the left and right child pointers to nullptr.

 **Recursive Case: else if (item < ptr->info)**

* **Condition:** If the item is smaller than the current node's value (ptr->info).
* **Action:** Recursively calls InsertHelper on the left subtree (ptr->left) to continue searching for the correct position.

 **Recursive Case: else if (item > ptr->info)**

* **Condition:** If the item is greater than the current node's value (ptr->info).
* **Action:** Recursively calls InsertHelper on the right subtree (ptr->right) to continue searching for the correct position.

 **Duplicates Not Allowed:**

* The implementation skips insertion if item == ptr->info.
* This ensures that duplicate values are not inserted into the tree.

template <class ItemType>

void TreeType<ItemType>::RetrieveItem(ItemType& item, bool& found) {

RetrieveHelper(root, item, found);

}

//  **Recursive Traversal:** The RetrieveHelper function ensures the tree is traversed efficiently based on the binary search tree property.

 **Boolean Feedback:** The found parameter provides a clear indication of success or failure.

template <class ItemType>

void TreeType<ItemType>::RetrieveHelper(TreeNode<ItemType>\* ptr, ItemType& item, bool& found) {

if (ptr == nullptr) {

found = false;

} else if (item < ptr->info) {

RetrieveHelper(ptr->left, item, found);

} else if (item > ptr->info) {

RetrieveHelper(ptr->right, item, found);

} else {

item = ptr->info;

found = true;

}

}

1. **Base Case: if (ptr == nullptr)**
   * If the current node is nullptr, it means the search has reached the end of a path without finding the item.
   * Sets found = false and stops recursion.
2. **Recursive Cases:**
   * **if (item < ptr->info)**:
     + The item being searched for is smaller than the current node's value.
     + Traverses the left subtree by recursively calling RetrieveHelper(ptr->left, item, found).
   * **if (item > ptr->info)**:
     + The item being searched for is larger than the current node's value.
     + Traverses the right subtree by recursively calling RetrieveHelper(ptr->right, item, found).
3. **Item Found: else**
   * The item matches the current node's value (item == ptr->info).
   * Updates item (if needed) with the value from the tree: item = ptr->info.
   * Sets found = true.

template <class ItemType>

void TreeType<ItemType>::PrintTree(ofstream& outFile) {

PrintHelper(root, outFile);

}

//  **Purpose:** Provides an interface to print the contents of the binary search tree to a file in sorted order.

 **Parameters:**ofstream& outFile: A reference to an output file stream where the tree's contents will be written.

 **Implementation:**Calls the private helper function PrintHelper starting at the root of the tree.

template <class ItemType>

void TreeType<ItemType>::PrintHelper(TreeNode<ItemType>\* ptr, ofstream& outFile) {

if (ptr != nullptr) {

PrintHelper(ptr->left, outFile);

outFile << ptr->info << endl;

PrintHelper(ptr->right, outFile);

}

}

// Recursively traverses the tree in **in-order** sequence and writes each node's data to a file stream.

 Ensures that the contents of the BST are written to the file in sorted order, as dictated by the properties of a BST.

template <class ItemType>

TreeType<ItemType>::~TreeType() {

DestroyHelper(root);

}

template <class ItemType>

void TreeType<ItemType>::DestroyHelper(TreeNode<ItemType>\* ptr) {

if (ptr != nullptr) {

DestroyHelper(ptr->left);

DestroyHelper(ptr->right);

delete ptr;

}

}

// Cleans up all dynamically allocated nodes in the tree when a TreeType object is destroyed to prevent memory leaks.

 Calls the private helper function DestroyHelper, starting at the root node of the tree.

 After execution, all nodes in the tree are deleted, and the root becomes a dangling pointer.

// Driver Program

int main() {

TreeType<int> tree;

ofstream out("tree.txt");

for (int i = 1; i <= 10; i++) {

tree.InsertItem(i);

}

tree.PrintTree(out);

out.close();

int item = 5;

bool found = false;

tree.RetrieveItem(item, found);

cout << "Item found: " << (found ? "Yes" : "No") << endl;

return 0;

}

//  **Tree Creation:**

* A TreeType<int> object named tree is created, which is a binary search tree capable of holding integers.

 **Inserting Items:**

* A loop inserts integers from 1 to 10 into the tree. The tree will automatically arrange these numbers in sorted order due to the properties of a binary search tree
* <https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/>
* <https://www.programiz.com/dsa/binary-search-tree>

**Task 1:**

**Problem Statement:**

**You are given a Binary Search Tree (BST) class with the following structure:**

template <class ItemType>

struct TreeNode {

ItemType info; // Data stored in the node

TreeNode<ItemType>\* left; // Pointer to left child

TreeNode<ItemType>\* right; // Pointer to right child

};

template <class ItemType>

class TreeType {

public:

TreeType(); // Constructor to initialize the tree

bool IsEmpty() const; // Returns true if the tree is empty

void InsertItem(ItemType item); // Insert an item into the tree

void RetrieveItem(ItemType& item, bool& found); // Retrieve an item

void RetrieveHelper(TreeNode<ItemType>\* ptr, ItemType& item, bool& found); // Helper function for retrieving

void PrintTree(ofstream& outFile); // Print the tree contents

// Destructor and other helper functions...

};

You are required to implement a function that searches for a specific item in the binary search tree and returns whether the item is found.