**5-2 Assignment: Binary Search Tree**

Nicholas Shaner

SNHU

CS-300-18278-M01 DSA: Analysis and Design

Professor Griffith

April 6, 2025

For this week’s assignment, we were once again asked to work with our supplied datasets of monthly sales records. The data structure being utilized was a tree, specifically a Binary Search Tree (BST). In this structure, data is organized in a hierarchal manner similar to what is typically imaged as a family tree. The top of the tree which anchors the structure is the ‘root’ node typically used as the access point to the tree. In a BTS, each node on the tree can have up to two successor nodes (child nodes), a left node and a right node. At the end of each branch of the tree, there are nodes with no successors called ‘leaf’ nodes. Each branch of the tree is strategically organized in which the key of the left child is lower than that of the parent node, and the right node is greater. Utilizing this form of structuring allows very quick operations like inserting, removing, and searching. This is due to the fact that at the end of each operational iteration the number of potential matching nodes is cut in half. Because of this logical breaking down of the structure the best-case runtime of these operations is O(logN). In a worst-case scenario in which the tree becomes completely unbalanced, it essentially turns into a linked list with a runtime of O(N). In this scenario, every node of the list may need to be iterated through.

As this data structure utilizes code and techniques similar to linked lists, and each node is clearly indicated with pointers to both the left and right child nodes, traversing the tree was relatively easy. The major challenge I ran into with this assignment, which was pointed out in Zybook readings this week, was removing an internal node that had pointers to two not null child nodes. The main challenge of this operation was due to the complexity of the code that included nested ‘if’ statements as well as a nested ‘while’ loop. Additionally, the pre-supplied instructional notes were not very clear. To overcome this, I actually moved the ‘removeNode’ function to an alternate file which allowed me to create a handful of simple nodes and work with the function piece by piece and manually select nodes to remove depending on location in the tree. Using this tactic allowed me to segregate the code with breakpoints and output statements to test the code operation at different points. Once I knew the code was working, I made a couple adjustments to the function parameters and moved the function back into the main ‘.cpp’ file.

**PSEUDOCODE**

BinarySearchTree() //Default constructor for BST, used to hold pointer to ‘root’ node

**SET** root equal to *nullptr*

virtual ~BinarySearchTree() //Default destructor for BST, used to delete root and BST

**DELETE** *root*

void inOrder(Node\* node) //Function call to traverse BST in order from lowest to highest

**IF** *root* node equals *nullptr* //validation that tree is not empty

**PRINT** message indicating empty tree

**RETURN** //break statement

**IF** *node* is equal to *nullptr* //base case escape function

**CALL** inOrder(node-> left) //recursive call to move current node to left child

**PRINT** *node* content //print bidId, title, amount, fund

**CALL** inOrder(node-> right) //recursive call to move current node to right child

void postOrder()

**IF** *root* node equals *nullptr* //validation that tree is not empty

**PRINT** message indicating empty tree

**RETURN** //break statement

**IF** node is equal to nullptr //base case escape function

**CALL** postOrder( node-> left)

**CALL** postOrder( node-> right)

**PRINT** *node* content //print bidId, title, amount, fund

void Insert(Bid bid)

**IF** *root* equals nullptr //indicates tree is empty so new node is now the root node

**SET** *root* to new bid

**ELSE**

**CALL** addNote() passing the *root* node, and new *bid* object as parameters

void Remove(string bidId)

**CALL** removeNode() passing *root* node and designated *bidId* as parameters

Bid Search(string bidId)

**INITIALIZE** new node object *currNode* //used to iterate comparisons

**SET** *currNode* to equal *root* node

**WHILE** *currNode* not equal to *nullptr* //iterates to end of branch

**IF** *currNode bidId* equal to requested *bidId* //match found

**RETURN** *currNode* bid object

**ELSE IF** requested *bidId* less than *currNode bidId*

**SET** *currNode* equal to *currNode* left child

**ELSE** //indicates requested *bidId* more than *currNode bidId*

**SET** *currNode* equal to *currNode* right child

**DECLARE** empty Bid object *bid* //called when no match found

**PRINT** message indicating no match found

**RETURN** *bid* //returns empty bid item (required for function return value)

Void addNode(Node\* node, Bid bid)

**IF** *node* is equal to *nullptr* //verification if tree empty

**SET** *root* node to new node passing *bid* argument //sets root to new node

**IF** *bidId* less than node *bidId*

**IF** *node* left child equal to *nullptr*

**SET** *node* left child to new node passing *bid* argument

**ELSE** //left child not empty

//recursive call on left child node

**CALL** addNode() passing current *node* left child and *bid* object arguments

**ELSE** //bidId is greater than current node bidId

**IF** *node* right child equal to *nullptr*

**SET** *node* right equal to new *node* passing *bid* argument

**ELSE** //right child not equal to nullptr

//recursive call on right child node

**CALL** addNode() passing *node* right child and *bid* object arguments

Node\* removeNode(Node\* node, string bidId)

**IF** *node* equal to *nullptr* //base case: indicated end of branch reached w/ no match

**IF** designated *bidId* less than current node *bidId*

**CALL** *removeNode* passing left child and designated *bidId* arguments

**ELSE IF** designated *bidId* more than current *node* *bidId*

**CALL** *removeNode* passing left child and designated *bidId* arguments

**ELSE** //match found, handles shifting remaining nodes

//matching node to remove is leaf node

**IF** *node* left child equal to *nullptr* AND *node* right child equal to *nullptr*

**DELETE** *node*

**PRINT** return statement to user

**RETURN** *nullptr* //required for function return value

//matching node to remove has only left child node

**ELSE IF** *node* left child not *nullptr* AND *node* right child equal to *nullptr*

**INITIALIZE** new *node* object *temp* and **SET** to *node*

**SET** *node* equal to *node* left child

**DELETE** *temp*

**PRINT** return statement to user

**RETURN** *node*

**ELSE IF** *node* left child equal *nullptr* AND *node* right child not *equal* nullptr

**INITIALIZE** new *node* object *temp* and **SET** to *node*

**SET** *node* equal to *node* right child

**DELETE** *temp*

**PRINT** return statement to user

**RETURN** *node*

**INITIALIZE** new *node* object *tempNode* and **SET** to *node* right child

**WHILE** *tempNode* left child not equal *nullptr*

**SET** *tempNode* equal to *tempNode* right child

**SET** node *bid* equal to *tempNode* bid

**SET** *root* right child to **CALL** removeNode(*node* right child, *tempNode* *bidId*

**RETURN** *node*