**Binary Search Tree: Code Reflection and Pseudocode**

Justin Starr

Department of STEM

CS 300 – DSA: Analysis and Design

Professor Ricardo Scarello

June 9, 2024

**Binary Search Tree: Code Reflection and Pseudocode**

**Code Reflection**

The purpose of this code was to be able to load a list of bids from a CSV file and insert the bids into a binary search tree data structure. Each bid is a node in the data structure which contains all the data for each bid. When the function loadBids is called, all bids are parsed from a CSV file and the Insert function is called to load the bids into the BST data structure. In the insert function, if the root of the BST is nullptr, the tree is empty and the root node is created with the passed-in bid. Otherwise, function addNode is called to determine where in the BST the node should be added. In the addNode function, the left side of the BST is checked first if the passed-in bidId is smaller than the root node’s bidId. A traversal of the left side of the BST continues until there is no left node, and then the new node is added to the BST. Similarly, if the passed-in bidId is greater than the root node’s bidId, a right-side traversal occurs until there is no right node and the new node is created with the passed-in bid.

Next, the user is able to display a list of all bids by calling function InOrder and sending the function the BST. InOrder calls function inOrder and passes in the root node. Function inOrder checks that the root node is not equal to nullptr and if not it recursively calls the inOrder function until it has traversed to the smallest node on the left side of the BST. Once it finds the smallest node, it displays the bid information for that node, then recursively traverses back up the BST checking the right side of every left side node and displaying the node's bid information along the way if the node is the next smallest node. This type of traversal ensures that the nodes are printed in sorted order, from least to greatest.

The user can also choose to find a bid, which when used the BST’s search function is called with a particular bidId. The search function begins by assigning the current node equal to the root node. While the current node is not equal to nullptr, the traversal continues unless the bid is found. If the current node’s bidId is the same as the passed-in bidId, the current bid is “found” and returned. Otherwise, traversal occurs by comparing the left and right side nodes. If the passed-in bidId is smaller than the current node’s bidId, then the left side is traversed. Otherwise, the right side of the BST is traversed.

The code further continues to demonstrate an understanding of Binary Search Tree algorithms, such as the ability to remove a node from a BST and traverse a BST in post-order and pre-order traversal. It also demonstrates how efficient search, insertion, and deletion operations can be when using a BST data structure, especially when the BST is balanced.

The biggest challenge I had while working on this assignment was actually implementing the destructor for the BinarySearchTree. The starter code did not provide a recursive function for deleting the BST, however, the comments in the destructor said to recurse from root deleting every node. To complete this portion of the code, I created the function “void BinarySearchTree::DeleteRecursive(Node\* &node) that when called from the BinarySearchTree destructor, would recursively delete all nodes in the BST. Following the already written in-line comments, utilizing the resources provided in this week’s resources, and conducting my own research helped me complete my work and overcome any challenges I faced.

**Pseudocode**

The following is the pseudocode for the main function and all of the functions that required fixing:

BinarySearchTree:: BinarySearchTree(), BinarySearchTree::~ BinarySearchTree(),

void BinarySearchTree::DeleteRecursive(Node\* node), void BinarySearchTree::InOrder(),

void BinarySearchTree:: BinarySearchTreePostOrder(), void BinarySearchTree::PreOrder(),

void BinarySearchTree::Insert(Bid bid), void BinarySearchTree::Remove(string bidId),

Bid BinarySearchTree::Search(string bidId), void BinarySearchTree::addNode(Node\* node, Bid, bid), void BinarySearchTree::inOrder(Node\* node), void BinarySearchTree::postOrder(Node\* node), void BinarySearchTree::preOrder(Node\* node), Node\* BinarySearchTree::removeNode(Node\* node, string bidId)

**int main(int argc, char\* argv[])** function **{**

DEFINE csvPath (type string) and searchValue (type string)

SWITCH (argc) {

case 2:

STORE csvPath = argv[1]

STORE searchValue = “98223”

BREAK

END case 2

case 3:

STORE csvpath = argv[1];

STORE searchValue = argv[2]

BREAK

END case 3

default:

STORE csvPath with the name of the csv file

STORE searchValue = “98109”

END DEFAULT

DEFINE ticks (type clock\_t)

DEFINE bst (type BinarySearchTree) and INITIALIZE the new hash table

DEFINE bid (type Bid)

DEFINE choice (type int) and INITIALIZE to zero

WHILE LOOP choice not equal to 9 {

DISPLAY “MENU:” and ENDLINE

DISPLAY “1. Load Bids” and ENDLINE

DISPLAY “2. Display all Bids” and ENDLINE

DISPLAY “3. Find Bid” and ENDLINE

DISPLAY “4. Remove Bid” and ENDLINE

DISPLAY “9. Exit” and ENDLINE

DISPLAY “ Enter Choice: “ and ENDLINE

INPUT choice

SWITCH (choice) {

case 1:

STORE ticks = CALL clock() // current clock ticks

CALL loadBids(csvPath, bst)

STORE ticks = CALL clock() – ticks // current clock ticks

minus starting clock ticks

DISPLAY “time:”, ticks, “ clock ticks”

DISPLAY “time:”, ticks \* 1.0 / CLOCKS\_PER\_SEC, “ seconds”

BREAK

END case 1

case 2:

CALL InOrder for bst

BREAK

END case 2

case 3:

STORE ticks = CALL clock() // current clock ticks

STORE bid = CALL Search(searchValue) in bst

STORE ticks = CALL clock() – ticks // current clock ticks minus

starting clock ticks

IF the returned bid is not empty {

CALL displayBid(bid)

} END IF

ELSE {

DISPLAY “Bid Id”, searchValue, “not found”

} END ELSE

DISPLAY “time:”, ticks, “ clock ticks”

DISPLAY “time:”, ticks \* 1.0 / CLOCKS\_PER\_SEC, “seconds”

BREAK

END case 3

case 4:

CALL Remove(searchValue) on bst

BREAK

END case 4

} END SWITCH

} END WHILE LOOP

DISPLAY “Good bye.”

RETURN 0

} END main function

**BinarySearchTree:: BinarySearchTree()** function {

ASSIGN root equal to nullptr

} END function

**BinarySearchTree::~ BinarySearchTree()** function {

CALL DeleteRecursive function and PASS the root node

} END function

**void BinarySearchTree::DeleteRecursive(Node\* &node)** function {

IF node is nullptr {

RETURN

} END IF

ELSE {

CALL DeleteRecursive and PASS node’s left node

CALL DeleteRecursive and PASS node’s right node

DELETE node

ASSIGN node equal to nullptr

} END ELSE

} END function

**void BinarySearchTree::InOrder()** function {

CALL inOrder function and pass root

} END function

**void BinarySearchTree::PostOrder()** function {

CALL postOrder function and pass root

} END function

**void BinarySearchTree::PreOrder()** function {

CALL preOrder function and pass root

} END function

**void BinarySearchTree::Insert(Bid bid)** function {

IF root is equal to nullptr {

ASSIGN root equal to new node and PASS in the bid

} END IF

ELSE {

POINT this to CALL addNode and PASS root and bid

} END ELSE

**void BinarySearchTree::Remove(string bidId)** function {

POINT this to CALL removeNode and PASS root and bidId

} END function

**Bid BinarySearchTree::search(string bidId)** function {

ASSIGN current node equal to root node

LOOP WHILE current node is not nullptr {

IF the current node’s bidId is equal to bidId {

RETURN the current node’s bid

} END IF

IF the passed in bidId is smaller than the current node’s bidId {

// traverse left

ASSIGN current node equal to current’s left node

} END IF

ELSE larger so traverse right {

ASSIGN current node equal to current’s right node

} END ELSE

} END WHILE LOOP

// if the program reaches here the bid was not found

CREATE a new empty bid

RETURN the empty bid

} END function

**void BinarySearchTree::addNode(Node\* node, Bid bid)** function {

IF the passed in node’s bidId is larger than the passed in bid’s bidId {

IF the passed in node’s left node is nullptr {

ASSIGN node’s left node to a new node with the passed in bid

} END IF

ELSE recurse down the left node {

POINT this to CALL addNode and PASS node’s left node and the passed

in bid

} END ELSE

} END IF

} END function

**void BinarySearchTree::inOrder(Node\* node)** function {

IF the passed in node is not equal to nullptr {

CALL inOrder and pass node’s left node

DISPLAY node’s bid.bidId, title, amount, and fund

CALL inOrder and pass node’s right node

} END IF

} END function

**void BinarySearchTree::postOrder(Node\* node)** function {

IF the passed in node is not equal to nullptr {

CALL postOrder and PASS the node’s left node

CALL postOrder and PASS the node’s right node

DISPLAY the node’s bid.bidId, title, amount, and fund

} END IF

} END function

**void BinarySearchTree::preorder(Node\* node)** function {

IF the passed in node is not equal to nullptr {

DISPLAY the node’s bid.bidId, title, amount, and fund

CALL postOrder and PASS the node’s left node

CALL postOrder and PASS the node’s right node

} END IF

} END function

**Node\* BinarySearchTree::removeNode(Node\* node, string bidId)** function {

IF the passed in node is equal to nullptr {

RETURN the passed in node

} END IF

IF the passed in bidId is less than the passed in node’s bid.bidId {

ASSIGN the passed in node’s left node equal to CALL removeNode and PASS node’s left node and bidId

} END IF

ELSE IF the passed in bidId is greater than the passed in node’s bid.bidId {

ASSIGN the passed in node’s right node equal to CALL removeNode and PASS node’s right node and bidId

} END ELSE IF

ELSE node is a leaf node {

IF node’s left node equals nullptr AND node’s right node equals nullptr {

DELETE node

ASSIGN node equal to nullptr

} END IF

ELSE IF node’s left node AND node’s right node are not equal to nullptr {

ASSIGN temp (type node) equal to node

ASSIGN node equal to node’s left node

DELETE temp node

} END ELSE IF

ELSE IF node’s left node is equal to nullptr and node’s right node is not equal to

nullptr {

ASSIGN temp (type node) equal to node

ASSIGN node equal to node’s right node

DELETE temp node

} END ELSE IF

ELSE there is more than one child so find the minimum {

ASSIGN temp (type node) equal to node’s right node

LOOP WHILE temp’s left node is not equal to nullptr {

ASSIGN temp node with temp’s left node

} END WHILE LOOP

ASSIGN node’s bid with temp node’s bid

ASSIGN node’s right node by CALLING removeNode and PASS node’s

right node and temp node’s bid.bidId

} END ELSE

} END ELSE

RETURN node

} END function