Ryan Hatch  
SNHU  
CS-300

**Code Reflection:**

This project involved implementing a Binary Search Tree (BST) in C++ to manage bid data. The main use case of this project was to demonstrate the concepts of data structures, specifically BSTs, and their operations such as insertion, deletion, searching, and traversal.

**Challenges during the development:**

One of the main issues I faced was handling the removal of nodes with two children in the BST. This required careful consideration of the restructure of the tree in order to maintain the BST properties. I also struggled with the overall structure of the project, for example, how to organize the classes and methods effectively was much more complicated because of the nature of the pre existing structure of the project.

To help me get over my hurdles, I implemented a method to find the minimum value in the right subtree (successor) and replaced the node to be deleted with the successor. This approach ensured that the BST properties were preserved after deletion. I also spent time planning and organizing the project structure to ensure that the classes and methods were logically arranged and easy to understand. This helped me to manage the complexity of the project more effectively, as well as the ability to help other students understand how to properly implement the solutions into their projects.

This project enhanced my understanding of Binary Search Trees and the way that they operate. It helped me with my problem-solving skills, and practicing team communication skills by peer reviewing and sharing my work/ syntax with other students who were facing the exact same issue. I also was able to practice how to structure a project in a way that is easy to understand and navigate, which is good practice that helps to make it easier for the next development team to understand how the program operates.

**Pseudocode:**

**class BinarySearchTree:**

**Node root**  
  
 **// Constructor initializes the tree**

method BinarySearchTree():

root = null  
  
 **// Insert a bid into the tree**

method Insert(Bid bid):

If root is null:

root = new Node(bid)

Else:

Call recursiveInsert(root, bid)  
  
 **// Helper method for inserting a node**

method recursiveInsert(Node node, Bid bid):

If bid.bidId < node.bid.bidId:

If node.left is null:

node.left = new Node(bid)

Else:

Call recursiveInsert(node.left, bid)

Else:

If node.right is null:

node.right = new Node(bid)

Else:

Call recursiveInsert(node.right, bid)  
  
 **// Remove a bid from the tree**

method Remove(bidId):

root = Call recursiveRemove(root, bidId)  
  
 // Helper method for removing a node

method recursiveRemove(Node node, bidId):

If node is null:

return null

If bidId < node.bid.bidId:

node.left = Call recursiveRemove(node.left, bidId)

return node

If bidId > node.bid.bidId:

node.right = Call recursiveRemove(node.right, bidId)

return node

// Node with two children

If node.left is not null and node.right is not null:

Node minNode = findMin(node.right)

node.bid = minNode.bid

node.right = recursiveRemove(node.right, minNode.bid.bidId)

return node

// Node with one or no child

If node.left is not null:

return node.left

Else:

return node.right  
  
 // Find the minimum node

method findMin(Node node):

While node.left is not null:

node = node.left

return node

// Search for a bid in the tree

method Search(bidId):

Node current = root

While current is not null:

If bidId == current.bid.bidId:

return current.bid

ElseIf bidId < current.bid.bidId:

current = current.left

Else:

current = current.right

return null  
  
 // In-order traversal

method InOrder():

Call recursiveInOrder(root)  
  
 // Helper method for in-order traversal

method recursiveInOrder(Node node):

If node is not null:

Call recursiveInOrder(node.left)

Print node.bid

Call recursiveInOrder(node.right)  
  
 // Pre-order traversal

method PreOrder():

Call recursivePreOrder(root)  
  
 // Helper method for pre-order traversal

method recursivePreOrder(Node node):

If node is not null:

Print node.bid

Call recursivePreOrder(node.left)

Call recursivePreOrder(node.right)  
  
 // Post-order traversal

method PostOrder():

Call recursivePostOrder(root)  
  
 // Helper method for post-order traversal

method recursivePostOrder(Node node):

If node is not null:

Call recursivePostOrder(node.left)

Call recursivePostOrder(node.right)

Print node.bid

**Flowchart:**

Start: Begin the operation.

Check if Root is Null: Determine if the tree's root is null.

Yes: Set the root to the new node.

No: Proceed to find the correct insertion point.

Initialize Current to Root: Start from the root to find where the new node should go.

Compare Bid ID:

If Bid ID < Current Node's Bid ID:

Check if Left Child is Null:

Yes: Insert the new node here.

No: Set Current to Left Child and repeat comparison.

If Bid ID >= Current Node's Bid ID:

Check if Right Child is Null:

Yes: Insert the new node here.

No: Set Current to Right Child and repeat comparison.

Insert Node: Place the new node in the identified position.

End: The insertion operation is complete.