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

Cristiano Miranda

Southern New Hampshire University

CS-300: Analysis and Design

Professor Muthukudage

5 October 2025

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

Code Review:

            The purpose of this program was to enable users to load bids from a CSV file, allowing them to display all bids, search for a specific bid, or remove a certain bid. To accomplish this, the program utilizes a Binary Search Tree data structure to house the various bids as node elements along the various tree paths. This situation provided an opportunity to implement the Binary Search Tree data structure in a way that maximized its utility. While the insertion and removal of the Binary Search Tree data structure is similar to that of the singly linked list data structure, its utility is evident when searching through the tree to retrieve a specific bid. The Binary Search Tree stores the bids in nodes, which are then set as either the left or right child of parent nodes, except for the first node in the tree, which is the root and has no parent elements. The decision to place a node either to the left or right of another node is determined by comparing the value in the parent node to that of the new node. If the parent node's value is greater than the new node's value, the new node is placed to the left, and if the parent node's value is less than or equal to the new node's value, it is put to the right. By following these rules, it simplifies and reduces the number of comparisons required when searching the tree for a specific bid. In this program's case, the bid ID is used as the comparison value. Thus, when a search is performed, rather than comparing all bids within the list to the bid ID, the algorithm starts at the root of the tree and compares the desired bid's ID to the root ID. Then, it follows the appropriate tree path after making the comparison. Again, if the value of the passed-in bid ID is less than the current node, the left path is followed, and if the value is greater than the current node, the right path is followed. This pattern continues as the tree is traversed, moving from node to node. At any point, if the current node's value matches the passed-in bid ID, then that node is the sought-after node and is returned. Traversing the tree in this manner significantly reduces the average time required to search through it, regardless of its size, as fewer comparisons are needed since only a specific path will be searched, depending on the value to be found. At the same time, there is still the possibility for the worst-case scenario to require searching through the entire Binary Search Tree, like if all values added were less than their parent node, in which the Binary Search Tree would be skewed and essentially be a singly linked list, the run-time complexity would still be O(N). Still, having the potential to reduce the time spent searching for elements within a data structure can significantly improve the user experience, as even if users begin to add a larger amount of data to a program, having an efficient searching algorithm can be the deciding factor for whether a user continues to use the program.

            While the Binary Search Tree is more complex to implement, it remains relatively straightforward to code, as separating the various logic greatly helps reduce the additional complexities. Even with the addition of an extra pointer to each node element, it was simple enough to account for the added pointer when implementing the Binary Search Tree. Still, in this specific instance, we utilized recursion to design the logic for the Binary Search Tree. While recursion is helpful for reducing the amount of dry code and allowing solutions to problems to be more manageable, it does increase the complexity of designing the solution and the potential for errors. Most of the errors I encountered while coding this program, and specifically implementing the Binary Search Tree class, were caused by incorrectly calling a method recursively without properly exiting the function. My errors caused multiple program crashes, and since these crashes were caused by the recursive method, it made it difficult to debug the problem, as it became hard to identify when the crash occurred. To overcome these issues, I utilize pseudocode and walk through the recursive function step-by-step to properly understand what my code was executing at each step, identifying when the crash or errors occurred. In doing this, it made it easier to spot my mistakes and to properly understand what caused them. Rather than fixing the error and moving on, performing this process helped reinforce my understanding of using recursive functions, which proved helpful once I began coding the other methods for the class. Overall, while recursion is more challenging to debug and has a higher chance of errors occurring during development, preplanning and designing the recursive function can help mitigate potential issues. Additionally, walking through each step of the code's execution can aid in finding and debugging issues caused by recursion.

Pseudocode:

* STRUCT Bid
  + INIT bidId
  + INIT title
  + INIT fund
  + INIT amount
  + CONSTRUCTOR ()
    - SET amount = 0.0
  + ENDCONSTRUCTOR
* ENDSTRUCT
* STRUCT Node
  + INIT bid
  + INIT POINTER left
  + INIT POINTER right
  + CONSTRUCTOR ()
    - START
    - POINT left TO null
    - POINT right TO null
    - END
  + ENDCONSTRUCTOR
  + CONSTRUCTOR (aBid)
    - START
    - SET bid = aBid
    - CALL Constructor () // Call default constructor
    - END
  + ENDCONSTRUCTOR
* ENDSTRUCT
* CLASS BinarySearchTree
  + DEFINE PRIVATE POINTER root
  + DEFINE PUBLIC CONSTRUCTOR ()
    - START
    - POINT root TO null
    - END
  + ENDCONSTRUCTOR
  + DEFINE PUBLIC DECONSTRUCTOR ()
    - START
    - INIT POINTER currNode
    - POINT currNode TO root
    - WHILE currNode IS NOT POINTING TO null
      * CALL removeNode (currNode, currNode->bid.bidId) AND POINT currNode TO result
    - ENDWHILE
    - END
  + ENDDECONSTRUCTOR
  + PUBLIC METHOD InOrder ()
    - START
    - CALL inOrder (root)
    - END
  + ENDMETHOD
  + PUBLIC METHOD PostOrder ()
    - START
    - CALL postOrder (root)
    - END
  + ENDMETHOD
  + PUBLIC METHOD PreOrder ()
    - START
    - CALL preOrder (root)
    - ENDF
  + ENDMETHOD
  + PUBLIC METHOD Insert (bid)
    - START
    - IF root IS POINTING TO null THEN
      * INIT new Node (bid)
      * POINT root TO new Node
    - ELSE
      * CALL addNode (root, bid)
    - ENDIF
    - END
  + ENDMETHOD
  + PUBLIC METHOD Remove (bidId)
    - START
    - CALL removeNode (root, bidId)
    - END
  + ENDMETHOD
  + PUBLIC METHOD Search (bidId)
    - START
    - INIT POINTER currNode
    - POINT currNode TO root
    - WHILE currNode IS NOT POINTING TO null
      * IF currNode->bid.bidId IS EQUAL TO bidId THEN
        + RETURN currNode->bid
      * ELSE IF currNode->bid.bidId IS GREATER THAN bidId THEN
        + POINT currNode TO currNode->left
      * ELSE
        + POINT currNode TO currNode->right
      * ENDIF
    - ENDWHILE
    - INIT new Bid
    - RETURN new Bid
    - END
  + ENDMETHOD
  + PRIVATE METHOD addNode (node, bid)
    - START
    - IF node->bid.bidId IS GREATER THAN bid.bidId THEN
      * IF node->left IS POINTING TO null THEN
        + INIT new Node (bid)
        + POINT node->left TO new Node
      * ELSE
        + CALL addNode (node->left, bid)
      * ENDIF
    - ELSE
      * IF node->right IS POINTING TO null THEN
        + INIT new Node (bid)
        + POINT node->right TO new Node
      * ELSE
        + CALL addNode (node->right, bid)
      * ENDIF
    - ENDIF
    - END
  + ENDMETHOD
  + PRIVATE METHOD inOrder (node)
    - START
    - IF node IS POINTING TO null THEN
      * RETURN // Exit the method as the end of the tree root is reached
    - ENDIF
    - CALL inOrder (node->left)
    - CALL printNode (node)
    - CALL inOrder (node->right)
    - END
  + ENDMETHOD
  + PRIVATE METHOD postOrder (node)
    - START
    - IF node IS POINTING TO null THEN
      * RETURN // Exit the method as the end of the tree root is reached
    - ENDIF
    - CALL postOrder (node->left)
    - CALL postOrder (node->right)
    - CALL printNode (node)
    - END
  + ENDMETHOD
  + PRIVATE METHOD preOrder (node)
    - START
    - IF node IS POINTING TO null THEN
      * RETURN // Exit the method as the end of the tree root is reached
    - ENDIF
    - CALL printNode (node)
    - CALL preOrder (node->left)
    - CALL preOrder (node->right)
    - END
  + ENDMETHOD
  + PRIVATE METHOD removeNode (node, bidId)
    - START
    - IF node IS POINTING TO null THEN
      * RETURN node
    - ELSEIF node->bid.bidId IS GREATER THAN bidId THEN
      * CALL removeNode (node->left, bidId) AND POINT node->left TO result
    - ELSEIF node->bid.bidId IS LESS THAN bidId THEN
      * CALL removeNode (node->right, bidId) AND POINT node->right TO result
    - ELSE
      * IF node->left IS POINTING TO null AND node->right IS POINTING TO null THEN
        + Delete node
        + POINT node to NULL
      * ELSE IF node->left IS NOT POINTING TO null AND node->right IS POINTING TO null THEN
        + INIT POINTER removedNode
        + POINT removedNode TO node
        + POINT node TO node->left
        + DELETE removedNode
        + POINT removedNode TO null
      * ELSE IF node->left IS POINTING TO null AND node->right IS not POINTING TO null THEN
        + INIT POINTER removedNode
        + POINT removedNode TO node
        + POINT node TO node->right
        + DELETE removedNode
        + POINT removedNode TO null
      * ELSE
        + INIT POINTER succNode
        + POINT succNode TO node->right
        + WHILE succNode->left IS NOT POINTING TO null

POINT succNode TO succNode->left

* + - * + ENDWHILE
        + INIT succNodeBidData
        + SET succNodeBidData = succNode->bid
        + SET node->bid = succNodeBidData
        + CALL removeNode (node->right, succNodeBidData.bidId) AND POINT node->right TO result
      * ENDIF
    - ENDIF
    - RETURN node
    - END
  + ENDMETHOD
  + PRIVATE METHOD printNode (node)
    - START
    - INIT bid
    - SET bid = node->bid
    - DISPLAY bid.bidId + “: ” + bid.title + “ | ” + bid.amount + “ | ” + bid.fund
    - RETURN // exit the method
    - END
  + ENDMETHOD
* ENDCLASS
* FUNCTION displayBid (bid)
  + START
  + DISPLAY bid.bidId + “: ” + bid.title + “ | ” + bid.amount + “ | ” + bid.fund
  + RETURN // exit the method
  + END
* ENDFUNCTOIN
* FUNCTION loadBids (csvPath, bst)
  + START
  + DISPLAY “Loading CSV file ” + csvPath
  + CALL csv::Parser (csvPath) // loads the csv file at the location specified by csvPath
  + INIT VECTOR header
  + SET header = file.getHeader()
  + FOR EACH heading IN header
    - DISPLAY heading + “ | ”
  + ENDFOR
  + TRY
    - FOR index from 0 TO file.rowCount ()
      * INIT bid
      * SET bid.bidId = file[index][1]
      * SET bid.title = file[index][0]
      * SET bid.fund = file[index][8]
      * SET bid.amount = file[index][4]
      * INSERT bid INTO bst
    - ENDFOR
  + CATCH (error)
    - DISPLAY error
  + ENDTRY
  + END
* ENDFUNCTION
* FUNCTION strToDouble (str, ch)
  + START

// Move the unwanted character to the start of the string

* + CALL str.remove (str.begin (), str.end (), ch)

// Remove unwanted characters the start of the string

* + CALL str.erase (remove result, str.end ())
  + RETURN str
  + END
* ENDFUNCTION
* FUCNTION main (argc, argv)
  + START
  + INIT csvPath
  + INIT bidKey
  + IF argc IS EQUAL TO 2 THEN
    - SET csvPath = element AT argv[1]
    - SET bidKey = “98223”
  + ELSE IF argc IS EQUAL TO 3 THEN
    - SET csvPath = element AT argv[1]
    - SET bidKey = element AT argv[2]
  + ELSE
    - SET csvPath = “eBid\_Monthly\_Sales.csv”
    - SET bidKey = “98223”
  + ENDIF
  + INIT ticks
  + INIT bst
  + SET bst = new BinarySearchTree ()
  + INIT bid
  + INIT choice
  + SET choice = 0
  + WHILE choice IS NOT EUQAL TO 9
    - DISPLAY “Menu:

1. Load Bids
2. Display All Bids
3. Find Bid
4. Remove Bid
5. Exit

Enter choice: ”

* + - GET choice
    - IF choice IS EQUAL TO 1 THEN
      * SET ticks = clock () // get current time value
      * CALL loadBids (csvPath, bst)
      * SET ticks = clock () – ticks
      * DISPLAY “time: ” + ticks + “ clock ticks”

// Calculate and display seconds using a predefined conversion value for ticks to seconds

* + - * DISPLAY “time: ” + ticks \* 1.0 / CLOCKS\_PER\_SEC + “ clock ticks”
    - ELSE IF choice IS EQUAL TO 2 THEN
      * CALL bst->InOrder ()
    - ELSE IF choice IS EQUAL TO 3 THEN
      * SET ticks = clock () // get current time value
      * CALL bst->Search (bidKey) AND SET bid = result
      * SET ticks = clock () – ticks
      * IF bid.bidId IS NOT empty THEN
        + CALL displayBid (bid)
      * ELSE
        + DISPLAY “Bid Id ” + bidKey + “ not found.”
      * ENDIF
      * DISPLAY “time: ” + ticks + “ clock ticks”

// Calculate and display seconds using a predefined conversion value for ticks to seconds

* + - * DISPLAY “time: ” + ticks \* 1.0 / CLOCKS\_PER\_SEC + “ clock ticks”
    - ELSE IF choice IS EQUAL TO 4 THEN
      * CALL bst->Remove (bidKey)
    - ELSE IF choice IS EQUAL TO 9 THEN
      * BREAK
    - ELSE
      * DISPLAY “Invalid Input!”
      * SET choice = 0
    - ENDIF
  + ENDWHILE
  + DISPLAY “Good bye.”
  + RETURN
  + END
* ENDFUNCTION