Southern New Hampshire University

Chris Trimmer

CS-300-T1159 DSA: Analysis and Design

Assignment 5-2: Binary Search Tree Reflection

22EW1 – 09/27/2022

The purpose of this document is to provide a code reflection of assignment 5-2. Assignment 5-2 is focused on the binary search tree (BST) data structure. In previous assignments, we primarily focused on linear data structures. A BST is considered a hierarchical tree structure such that each node in the tree can have none, one, or two child nodes. The top-most node is considered the root node, and any nodes that do not have child nodes are called leaves. As nodes are inserted and removed from the data structure, the BST forms one of the most efficient data structures for searching, inserting, and deleting, as those operation can be performed in O(logN) time.

Code Reflection

Our code for this assignment includes the core operations that are part of a BST, which include: insert, remove, search, and three traversal operations (inorder, preorder, and postorder). All these functions also have helper functions. At a high level, the insertion process works by inserting nodes based on whether the key is greater or less than previously inserted nodes. The first node becomes the root node. After that, when a node is inserted, we recursively determine if the node should be inserted to the left or right side the nodes in the tree. The determination is based on whether the key is less than or greater than the current node being compared. The process of comparing keys and traversing to the left or right of a parent node works for the search process, as well as the removal process.

Overall, I really enjoyed working with BSTs. I was able to get all functions working recursively! The function I had the most trouble with was the GetParent function, but I ended up getting it to work. Our Zybook provided an overall good process and pseudocode for this, but I ended up making a lot of mistakes in how I assigned the pointers in the various functions. In our Zybook, they use three pointer variables in the helper function, and I spent a lot of time trying to get it to work with only two. Within the helper function, they create a pointer to hold the successor node. I tried to create a helper function to get the successor, but I must have been doing that one wrong. So, I defaulted back to the way that Zybook provide pseudocode for. I also added a function to get the size of the tree.

For most of the other functions, I just followed the steps outlined in the assignment and tested as I coded. This helped ensure that I found bugs and corrected any problems before moving to each subsequent step.

Chris Trimmer

CS-300-T1159 DSA: Analysis and Design

Assignment 5-2: Binary Search Tree Pseudocode

22EW1 – 09/27/2022

The purpose of this document is to provide pseudocode of the main functions used in Assignment 5-2. Assignment 5-2 is focused on the binary search tree (BST) data structure. In previous assignments, we primarily focused on linear data structures. A BST is considered a hierarchical tree structure, such that each node in the tree can have none, one, or two child nodes. The topmost node is considered the root node, and any nodes that do not have child nodes are called leaves. As nodes are inserted and removed from the data structure, the BST forms one of the most efficient data structures for searching, inserting, and deleting, as those operation can be performed in O(logN) time.

The following is the pseudocode for the core functions of the BST:

Pseudocode

```
// Constructor
BinarySearchTree::BinarySearchTree() {
  Set root to nullptr
}


// Destructor
BinarySearchTree::~BinarySearchTree() {
  Call ClearTree helper function with the root as argument

}

// ClearTree
Void BinarySearchTree::ClearTree(Node* node) {
  If node is null, then just return as the tree is empty

  Recursively call ClearTree on the left side of node
  Recursively call ClearTree on the right side of node
  Delete the node
}

// Insertion helper function
Void BinarySearchTree::addNode(Node* node, Bid bid) {

  If node bidId is greater than the incoming bidId
    If the left node is empty, then set the left node to the new node
    Else, recursively call addNode with the left side

  Else
```

If the right side is empty, then set the right side to the new node
    Else, recursively call addNode with the right side
}

// **Insertion**
Void BinarySearchTree::Insert(Bid bid) {

  If root is null, then set root to the new node and return

  Else call the addNode helper function with root and bid object

}

// **Search**
Node* BinarySearchTree::Search(string bidId) {

  Call the search helper, and return the result
}

// **Search helper**
Node* BinarySearchTree::SearchHelperA(Node* node, string bidId) {

  If the node is null, or the node bidId is equal to the incoming bidId,
    Then return the node

  If the node bidId is greater than the incoming bidId
    Then recursively call the SearchHelper on the left side and return

  If it less than,
    Then recursively call the SearchHelper on the right side
}

// **Removal**
Void BinarySearchTree::Remove(string bidId) {

  Create pointer node by searching for the bidId

  Create pointer parentNode by calling GetParent with the node

  Call the RemoveNode helper function with the root, parent, node
}

// Removal helper function

```
bool BinarySearchTree::removeNodeRecur(Node* tree, Node* parent, Node* node) {

  if node is null, then just return as there is nothing to delete

  // case 1: if the node has two children
  If node left is not null AND node right is not null {
    Create pointer as successorNode to node->right
    Create pointer as successorParent to node

    While successorNode->left is not null
      Set succParent to succNode
      Set succNode to succNode->left

    Set the node->bid to the succNode->bid

    Call removeNodeRecur recursively with tree, succParent, succNode arguments

  } // end case 1


  // case 2: if the node is the root
  Else if node equals root {

    If node->left is not null
      Set root to node->left
    Else
      Set root to node->right

  } // end case 2

  // case 3: if only a left child
  Else if node->left not equal to nullptr {
    If parent->left equals node
      Set parent->left to node->left
    Else
      Set parent->right to node->left

  } // end case 3

  // case 4: if only a right child or leaf
```

```
    Else {
      If parent->left equals node
        Set parent->left to node->right
      Else
        Set parent->right to node->right

  } // end case 4

}
```

// InOrder traversal helper

```
Void BinarySearchTree::inOrder(Node* node) {
  If node is empty, then return to caller immediately

  Recursively call inOrder using node->left
  Print the node->bid
  Recursively call inOrder using node->right

}
```

// InOrder traversal

```
Void InOrder() {

  Call inOrder with root node

}
```

// PreOrder traversal helper

```
Void BinarySearchTree::preOrder(Node* node) {

  If node is empty, then return to caller immediately

  Print the node->bid
  Recursively call preOrder using node->left
  Recursively call preOrder using node->right

}
```

// PreOrder traversal

```
Void PreOrder() {

  Call preOrder with root node
```

}

// **PostOrder traversal helper**
Void BinarySearchTree::postOrder(Node* node) {
  If node is empty, then return to caller immediately

  Recursively call postOrder using node->left
  Recursively call postOrder using node->right
  Print the node->bid

}

// **PostOrder traversal**
Void PostOrder() {

  Call postOrder with root node

}

// **Get Size of tree**
Size_t BinarySearchTree::GetSize() {
  Call size helper with root as argument, and return result
}

// **Get size helper function**
Size_t BinarySearchTree::size(Node* node) {
  If node is nullptr
    Return

  Recursively call size of left side and add
    Recursive call of size of right side and add 1, and return
}

// **Get parent**
Node* BinarySearchTree::GetParent(Node* node) {
  Return the result of get parent helper function, passing in root and node as arguments
}

// Get parent helper
Node* BinarySearchTree::GetParentHelper(Node* tree, Node* node) {

  If tree is null, then return null as the tree is empty

  If tree->left equals node OR tree->right equals node
    Then return the tree node

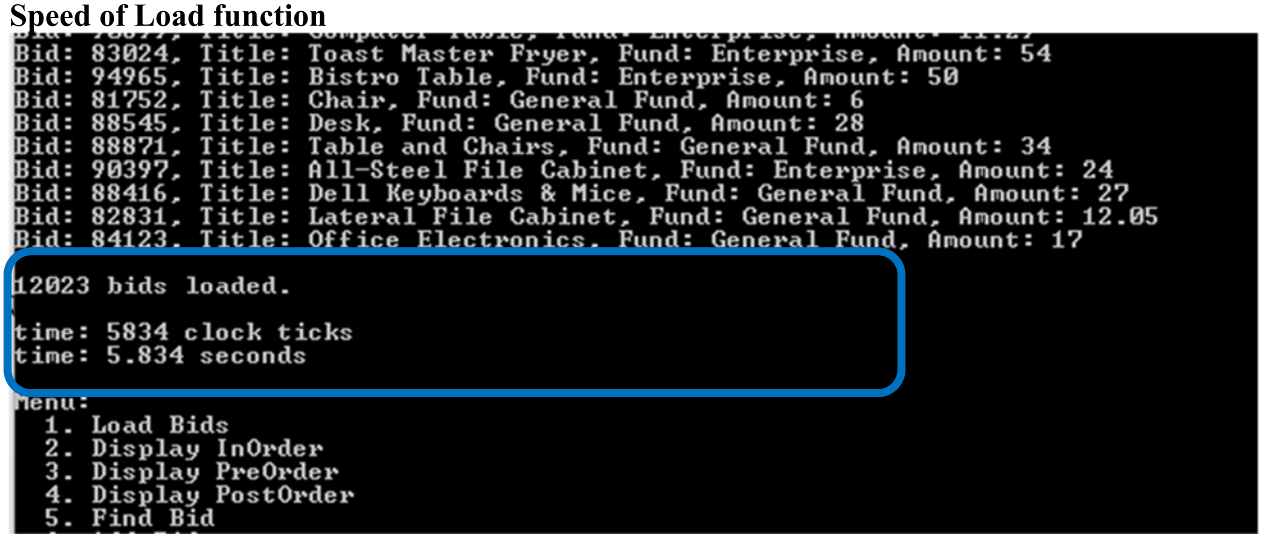
  If the node bidId is less than the tree bidId
    Then recursively call GetParentHelper with tree->left and node as arguments and return

  Otherwise recursively call GetParentHelper with tree->right and node as arguments and return

}

<u>Screenshots</u>

Speed of Load function



```
Bid: 83024, Title: Toast Master Fryer, Fund: Enterprise, Amount: 54
Bid: 94965, Title: Bistro Table, Fund: Enterprise, Amount: 50
Bid: 81752, Title: Chair, Fund: General Fund, Amount: 6
Bid: 88545, Title: Desk, Fund: General Fund, Amount: 28
Bid: 88871, Title: Table and Chairs, Fund: General Fund, Amount: 34
Bid: 90397, Title: All-Steel File Cabinet, Fund: Enterprise, Amount: 24
Bid: 88416, Title: Dell Keyboards & Mice, Fund: General Fund, Amount: 27
Bid: 82831, Title: Lateral File Cabinet, Fund: General Fund, Amount: 12.05
Bid: 84123, Title: Office Electronics, Fund: General Fund, Amount: 17

12023 bids loaded.

time: 5834 clock ticks
time: 5.834 seconds
Menu:
  1. Load Bids
  2. Display InOrder
  3. Display PreOrder
  4. Display PostOrder
  5. Find Bid
```

Snippet of InOrder traversal

```
98897: 1 Lot of 50 Chairs, Enterprise, 106.01
98898: 1 Lot of 50 Chairs, Enterprise, 145
98899: 1 Lot of 50 Chairs, Enterprise, 106
98900: 1 Lot of 50 Chairs, Enterprise, 52
98901: 1 Lot of 3 Chairs, General Fund, 7.05
98902: Chair, General Fund, 57
98905: Dell Laptop, General Fund, 72.5
98907: File Cabinet, General Fund, 21
98911: Dell Laptop, General Fund, 115

time: 2314 clock ticks
time: 2.314 seconds

Bid count: 12023

Menu:
  1. Load Bids
  2. Display InOrder
  3. Display PreOrder
  4. Display PostOrder
  5. Find Bid
  6. Add Bid
```

Speed of Search function

```
Menu:
  1. Load Bids
  2. Display InOrder
  3. Display PreOrder
  4. Display PostOrder
  5. Find Bid
  6. Add Bid
  7. Remove Bid
  9. Exit
Enter choice: 5


Enter bidId: 98899

98899: 1 Lot of 50 Chairs | 106 | Enterprise

time: 1 clock ticks
time: 0.001 seconds
```

Speed of Removal

Southern New Hampshire University

```
Menu:
  1. Load Bids
  2. Display InOrder
  3. Display PreOrder
  4. Display PostOrder
  5. Find Bid
  6. Add Bid
  7. Remove Bid
  9. Exit
Enter choice: 7

Enter bidId: 98899

Bid has been removed.
time: 0 clock ticks
time: 0 seconds
Menu:
  1. Load Bids
  2. Display InOrder
  3. Display PreOrder
  4. Display PostOrder
  5. Find Bid
  6. Add Bid
  7. Remove Bid
  9. Exit
Enter choice: 5


Enter bidId: 98899
98899 was not found
```