Billy Hegarty

CS300 Assignment 5: Binary Search Tree

## Code Reflection

The code “BinarySearchTree.cpp” reads in data from a .csv file and provides several data engagement choices to the user via a numerical menu. The .csv file location can be specified as a command line argument, otherwise it will default to a file named “eBid\_Monthly\_Sales\_Dec\_2016.csv" located in the same local file as BinarySearchTree.exe. The .csv file contains rows of data. Each row corresponds to an individual bid and auction for an item, such as computer parts or office furniture. The program organizes each bid in a data structure named “Bid” and assigns attributes bidId, title, fund, and amount. These attributes are automatically read in from the .csv file using the provided “CSVparser.cpp” file by selecting menu option #1.

When the program runs, the main() function declares a BinarySearchTree object named “bst” that will hold all bid data from the .csv file. Each node in the BST is initialized to hold a default node construction, a default bid, and “left” and “right” null pointers.

Menu option #1, “Load Bids”, reads in one line at a time from the .csv file. After reading in each line, the bid data is inserted into the BST using the loadBids and Insert() methods. The loadBids() method reads in the bidId, title, fund, and amount, and creates a Bid object to store the attributes. It then passes this Bid object to the Insert() method. The Insert() method first determines if the BST is empty, indicated by a null root node. If it is empty, the passed Bid becomes the root. Otherwise, the Bid gets passed via the addNode() method.

The addNode() method compares the bidId of the new Bid to the bidId that exists at the current node (starting at the root). If the bidId is less than the root bidId, then addNode() recursively compares the new bidId to each node in the left subtree until reaching a larger bidId with no left child. At this point, we can assign the new Bid as the left child of the current bid. A similar process occurs if the bidId is larger than the root bidId, except it recursively traverses the right subtree.

Menu option #2, “Display All Bids”, prints a header line with column titles, then prints bid data for each node in the BST. It calls the inOrder() method, which recursively traverses down the leftmost subtree until finding a null pointer. After encountering a null left pointer, the recursion ends and the inOrder() method prints bid data for the node with the null left pointer. The method then attempts similar recursive diving down the right subtree until hitting a null pointer. Thus, the method recursively finds the left most leaf node, then the “next right”, or “second left-most” leaf node, and unwinds in the reverse path until returning to the root node. This ensures that the bid data is printed in the correct order.

Menu option #3 is Find Bid. The program will search for a bid with a particular bidID, either specified in as a command line argument or by the default ID value 98024. To accomplish this function, the code performs the Search() method on the BST. The Search() method starts at the root and checks if the bidId in the root node matches the target bidId. If it matches, the current (root) node is returned. Otherwise, if the target bidId is less than the current (root) node, the method compares the target bidId to the left child bidId (or right child, if greater than the current node). This process continues until the current pointer returns null. If a matching bidId exists in the BST, it will satisfy the first comparison and the method will return the node containing the matching bidId. If no match is found, the method will return a node containing a default Bid. The menu case will recognize that the returned Bid is default and will print a message saying that the target bid could not be found.

Menu option #4 is Remove Bid. Similar to Find Bid, this method uses the particular bidID passed as a command line argument, or the default 98024 value. The menu case calls the Remove() method, which in turn calls the removeNode() method. The removeNode() method recursively travels down the left or right subtree, depending on if the passed bidId is less than or greater than the bidId in the current node. If bidId is less than the bidId in the current node, then the method recursively calls removeNode() on the subsequent left subtree. This continues until the bidId is equal to the bidId in the current node. Now there are two scenarios: the current node has 1 or 0 children, or it has 2 children. These scenarios are implemented in series but “return” statements ensure that the method ends when it is appropriate. In the first scenario, 1 or 0 children, the left and/or right pointers will be null. The program creates a temporary node and copies the data from the non-null child to it. The target (matching bidId) node is then deleted, and the temporary node (remaining descendants of the deleted node) is returned to preserve the structure of the subtree. If neither the left nor the right children are null (2 child scenario), the program creates a temporary node and copies the right subtree to it. The program then finds the in-order successor, i.e. the next-smallest (left-most) bidId that exists in the right subtree. After finding the in-order successor node, its contents are copied to overwrite the contents of the target (matching bidId) node. Then, the in-order successor node is deleted, eliminating duplication from the BST while preserving the descendants of the target bidId node.

I used a JetBrains IDE called CLion instead of Microsoft Visual Studio. CLion is user friendly and provides visual indications of errors, warning, and typos throughout the code. Since I had previously made housekeeping edits to the “CSVparser.cpp” and “CSVparser.h” files during the last assignment, I reused those clean files instead of using the helper files provided for this assignment.

The implementation and practical application of the binary search tree was (surprisingly) easier than the hash table assignment from last week. I found that navigating the BST was intuitive, and it was especially easy to draw out nodes on paper and visually trace the methods when I was writing the code. Binary search trees seem to lend themselves to recursion, which is an area I used to shy away from. However, completing this week’s assignment gave me confidence to employ recursion (when it makes sense). The only roadblock I encountered when writing this program was the same one I have seen in prior weeks. When I load the SNHU-provided .cpp code into CLion, the program will compile and run normally in the IDE. However, When I try to run the corresponding .exe. file, I get an error message. I followed the same steps this week as in the past: start from scratch with a safe “Hello World” file, and confirm that it will execute normally in the command prompt, then slowly copy and paste methods from the SNHU file into the “Hello World” file. I think the error message I am seeing is related to a difference between the runtime environment in CLion and Windows File Explorer. I tried updating CLion, downloading the latest Visual C++ redistributable files, and building a new project in CLion that manually called on a freshly installed Min-GW 64 directory (verified by checking PATH environment variables) instead of using the default CLion version. I’m still not exactly sure what the problem is, but at the end of the day I was able to come up with a functional .exe. file that satisfies all the assignment requirements.

## Pseudocode

* Define a structure “Bid” with attributes that correspond to the column headers in the data .csv file.
* Define a structure “Node”, which will contain Bid data as well as left and right pointers to link together the BST parent/child nodes
  + Define default constructor for Node, and populate it was a default (empty) Bid
* Declare functions that will be defined later
  + strToDouble
* Define the BinarySearchTree class
  + Private members
    - Node root
    - void addNode(Node\* node, const Bid& bid) : Helper method to recursively add a bid node
    - void inOrder(Node\* node) : Helper method to recursively traverse the tree in order
    - void postOrder(Node\* node) : Helper method to recursively traverse the tree in post-order
    - void preOrder(Node\* node) : Helper method to recursively traverse the tree in pre-order
    - Node\* removeNode(Node\* node, const string& bidId) : Helper method to recursively remove a bid node via Remove method
    - void removeSubtree(Node\* node) : Helps Destructor method to recursively remove the entire subtree by drilling down the left, then right subtrees, and deleting a node when both children are null
  + Public members
    - BinarySearchTree() : Constructor, populates a single node with root nullptr
    - ~BinarySearchTree() : Destructor
    - void InOrder() : Traverse the tree in-order
    - void PostOrder() : Traverse the tree in post-order
    - void PreOrder() : Traverse the tree in pre-order
    - void Insert(Bid bid) : Insert a bid into the tree
    - void Remove(string bidId) : Remove a bid from the tree
    - Bid Search(string bidId) : Search for a bid in the tree
* Define InOrder()
  + Call inOrder and pass root
* Define PostOrder()
  + Call postOrder and pass root
* Define PreOrder()
  + Call preOrder and pass root
* Define Insert()
  + If BST is null, insert Bid as root
  + Else call addNode()
* Define Remove()
  + Call removeNode()
* Define removeNode()
  + Copmpare current node bidID to target bidID
  + Recursively traverse left (less) or right (greater) subtrees until finding matching node
  + If match node has 1 or 0 children
    - Pop non-null child to current node, delete matching node
  + Else
    - Copy in-order successor (left-most of the right subtree) to matching node
    - Delete in-order successor (to eliminate duplicate nodes)
* Define Search()
  + Start at root node
  + While current node is not null
    - Return current node if it contains matching bidID
    - Else traverse left (less) or right (greater)
  + If entire BST traversed and matching node not returned, return a default (empty) Bid
* Define addNode()
  + Compare new bidId to current node (starting at root)
    - If bidId is less than current node bidId
      * Recursively call addNode to left child
    - Else
      * Recursively call addNode to right child
    - When encountering null child, assign bid to child node
* Define inOrder()
  + Base case: check if current node is null. If so, return.
  + Recursive steps
    - Traverse left subtree until encountering null left child (left-most leaf node)
    - Print bid data
    - Traverse right subtree of current node
    - As soon as it hits a null node, it will “undo” one level of recursion, ensuring that the BST data gets printed in true order (least to greatest bidId)
* Define postOrder()
  + Base case: check if current node is null. If so, return.
  + Recursive steps
    - Call postOrder to traverse left subtree of current node
    - Next, call postOrder to traverse right subtree of current node
    - Print data for the current node
    - This ensures that, starting from the bottom-left most node (smallest bidId), the left child, then right child, then parent will be printed out. Then it will move to that parent’s “right” sibling, printing the sibling’s left child, then right child, then sibling.
    - postOrder returns all nodes, reading back one complete “parent-children” group at a time.
      * Left subtree
      * Right subtree
      * Current (parent) node
* Define preOrder()
  + Base case: check if current node is null. If so, return.
  + Recursive steps
    - Print data for current node
    - Call preOrder to traverse left subtree
    - (continue until encountering null left child, indicating bottom-left most node reached)
    - Call preOrder to traverse right subtree
    - preOrder will print:
      * root
      * left subtree
      * right subtree
* Define displayBid()
  + Print bidID, title, amount, and fund
* Define a method to print all bids in the BST
  + Call InOrder()
* Define a function strToDouble that strips an unwanted character from a string and then converts the string to a double
* Define a function called “loadBids”
  + Input parameter string for the CSV file path
  + Create bid objects for each line in the CSV file
  + Insert the bids into the BST using Insert()
* Create a main function:
  + Process command line arguments, such as the filepath of the CSV file
  + Create a Binary Search Table that will contain bids
  + Define a timer variable
  + Display a menu and get user choices
  + Perform actions based on user choice
    - Load Bids
    - Display All Bids
    - Find Bid
    - Remove Bid
    - Exit
  + Display a goodbye message after exiting