Quintin B. Rozelle
5/28/2025
4-3 Milestone Three
CS-499

- **Briefly describe the artifact. What is it? When was it created?**

The artifact chosen for this enhancement is the binary search tree (BST) assignment from CS-300 Data Structures and Algorithms. The original intent of this code was to showcase the use of a BST in storing, sorting, searching, displaying, and deleting data that was imported from a CSV file. It was one of a series of similar assignments that accomplished the same task but with different data structures so we could learn firsthand how they function and differ from one another.

- **Justify the inclusion of the artifact in your ePortfolio. Why did you select this item? What specific components of the artifact showcase your skills and abilities in algorithms and data structure? How was the artifact improved?**

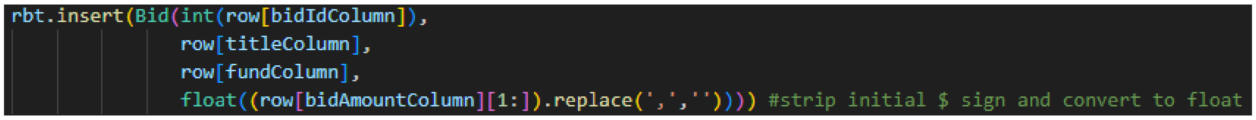
I chose this artifact for this enhancement due to some discoveries I made while researching data structures and algorithms. I was aware that more advanced data structures existed but I didn't know much about them, so I had assumed that BST offered what they did with minimal downsides. I quickly discovered that standard BSTs are not guaranteed to have a time complexity of O(log n) because they have no mechanism for enforcing balance to the tree. In fact, in a BST that has no balance (i.e., all nodes share the same parity in regard to their left verse right child status), operations approach a time complexity of O(n) (GeeksforGeeks, 2025). I further learned that BST data structures that enforce self-balancing retain a guaranteed time complexity of O(log n). Based on this, I became interested exploring one of these.

Using this artifact seemed like the obvious choice since it is originally built with a standard BST and improving it with a self-balancing BST would provide me the opportunity to learn how these work while seeing firsthand how they improve upon a non-self-balancing BST. I settled on implementing a Red Black Tree (RBT) over other self-balancing BSTs (i.e., AVL Trees or Splay Trees) because RBTs are best for situations in which insertion speed is needed more than search speed (GeeksforGeeks, 2023). Since this artifact does more insertions than searches, an RBT was chosen as the best option.

The main improvements for this artifact are the inclusion of an RBT as the data structure used while also building it in a way that the BST that was built for enhancement one didn't break. This was done through separate classes (BST vs RBT) while also modifying the Node class to work with either data structure. In addition to this, I further improved the unit testing to test the new RBT.

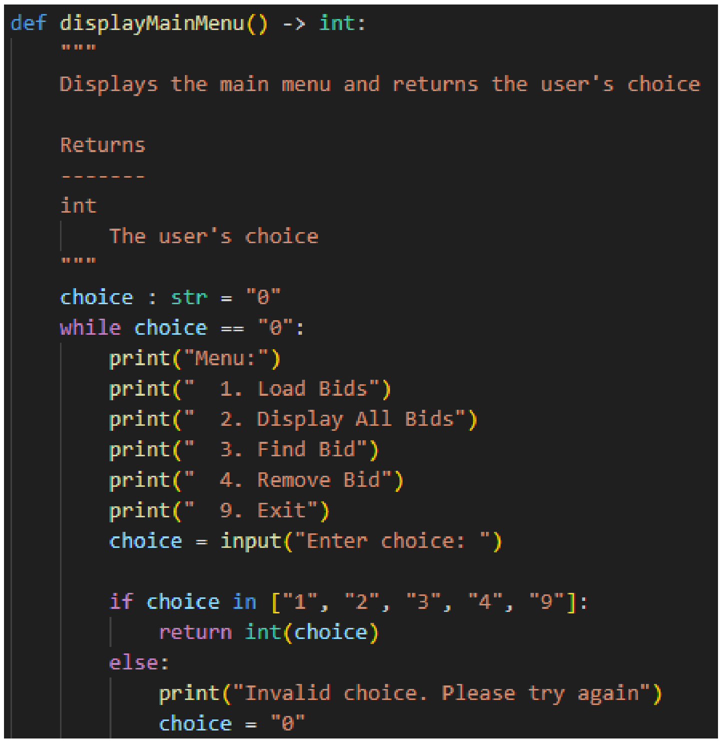
In addition to the implementation of an RBT, I also retained the improved security and stability that was created during the first enhancement. This is accomplished through data validation and exception handling. Examples of data validation include the following:

- Ensuring that Bid data meets the requirements of the Bid class, specifically that the Bid ID and Bid Amount are and int and float respectively:



```
rbt.insert(Bid(int(row[bidIdColumn]),
                row[titleColumn],
                row[fundColumn],
                float((row[bidAmountColumn][1:]).replace(',',''))) #strip initial $ sign and convert to float
```

- Ensuring that input for menus adhere to the set of possible choices. Deviations from this display an error and ask the user to try again:

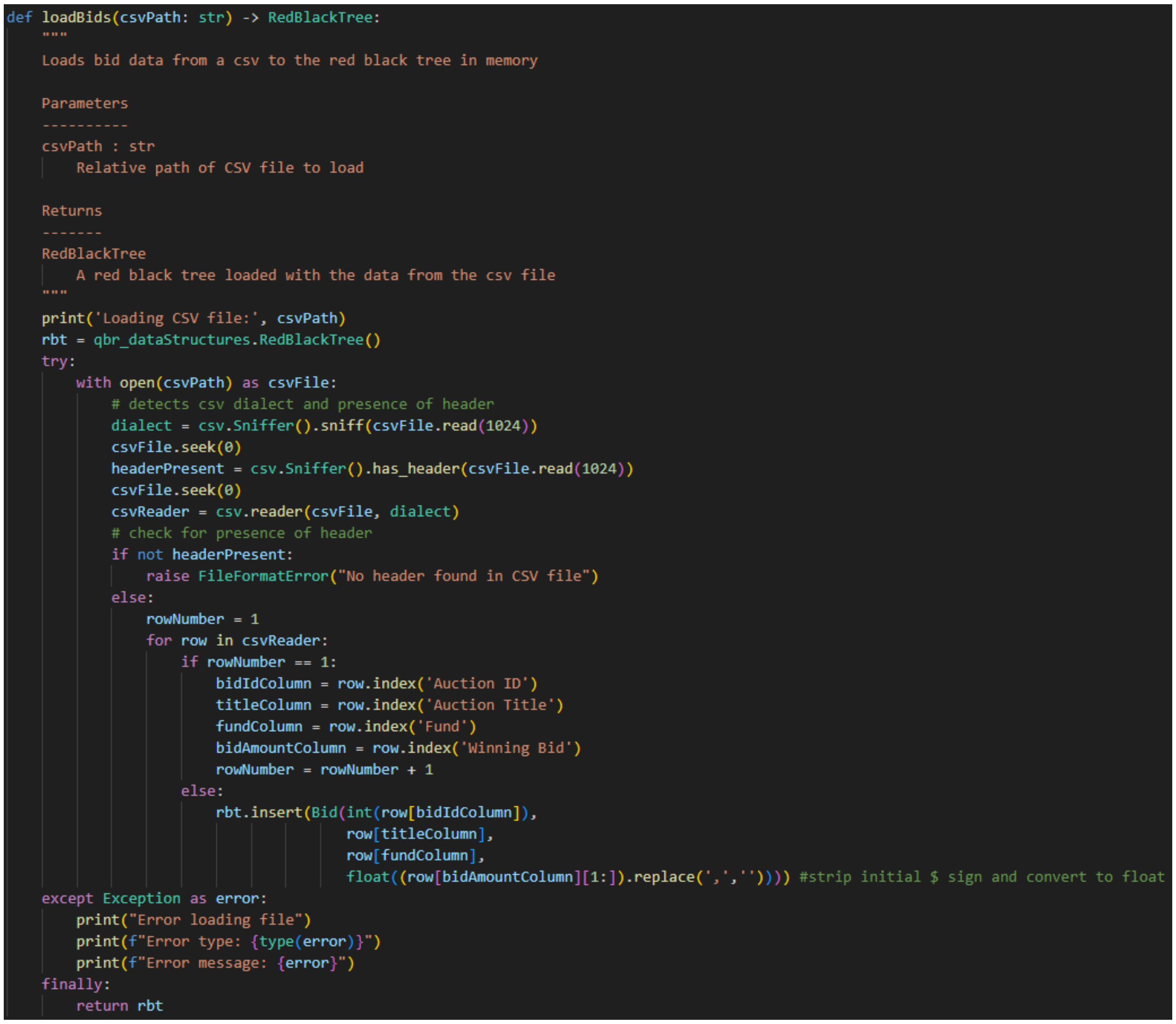


```python
def displayMainMenu() -> int:
    """
    Displays the main menu and returns the user's choice

    Returns
    -------
    int
        The user's choice
    """
    choice : str = "0"
    while choice == "0":
        print("Menu:")
        print("  1. Load Bids")
        print("  2. Display All Bids")
        print("  3. Find Bid")
        print("  4. Remove Bid")
        print("  9. Exit")
        choice = input("Enter choice: ")

        if choice in ["1", "2", "3", "4", "9"]:
            return int(choice)
        else:
            print("Invalid choice. Please try again")
            choice = "0"
```

The main example of exception handling is seen in the loadBids function. This function has the possibility of throwing an exception due to a variety of reasons (e.g., missing file, missing headers in file, data formatted incorrectly, etc.). This function will catch those errors, display an appropriate error message to the user, then return to the main function:



```
def loadBids(csvPath: str) -> RedBlackTree:
    """
    Loads bid data from a csv to the red black tree in memory

    Parameters
    ----------
    csvPath : str
        Relative path of CSV file to load

    Returns
    -------
    RedBlackTree
        A red black tree loaded with the data from the csv file
    """
    print('Loading CSV file:', csvPath)
    rbt = qbr_dataStructures.RedBlackTree()
    try:
        with open(csvPath) as csvFile:
            # detects csv dialect and presence of header
            dialect = csv.Sniffer().sniff(csvFile.read(1024))
            csvFile.seek(0)
            headerPresent = csv.Sniffer().has_header(csvFile.read(1024))
            csvFile.seek(0)
            csvReader = csv.reader(csvFile, dialect)
            # check for presence of header
            if not headerPresent:
                raise FileFormatError("No header found in CSV file")
            else:
                rowNumber = 1
                for row in csvReader:
                    if rowNumber == 1:
                        bidIdColumn = row.index('Auction ID')
                        titleColumn = row.index('Auction Title')
                        fundColumn = row.index('Fund')
                        bidAmountColumn = row.index('Winning Bid')
                        rowNumber = rowNumber + 1
                    else:
                        rbt.insert(Bid(int(row[bidIdColumn]),
                                       row[titleColumn],
                                       row[fundColumn],
                                       float((row[bidAmountColumn][1:]).replace(',',''))))) #strip initial $ sign and convert to float
    except Exception as error:
        print("Error loading file")
        print(f"Error type: {type(error)}")
        print(f"Error message: {error}")
    finally:
        return rbt
```

The skill I have shown with this enhancement is my ability to select and apply advanced algorithms and data structures to a problem when necessary. The thought process for choosing an RBT over another self-balancing BST shows that I can critically think about the best solution for the task at hand and my coding of the RBT shows that I am able to implement that solution.

- **Did you meet the course outcomes you planned to meet with this enhancement in Module One? Do you have any updates to your outcome-coverage plans?**

I planned to apply outcomes 1 and 3 with this enhancement. Like enhancement one, I have made significant progress toward outcome 1 with this enhancement through my inclusion of well written comments and docstrings, though I do not wish to consider it completed until finishing enhancement three since I have also applied outcome 1 to that as well. I have successfully completed outcome 3 with this enhancement though my implementation of an RBT as the data structure for this artifact.

- **Reflect on the process of enhancing and modifying the artifact. What did you learn as you were creating it and improving it? What challenges did you face?**

I learned two major things about advanced data structures and algorithms with this enhancement. First is how much more complex they are than the ones we learned in CS-300 Algorithms and Data Structures. I assumed RBTs to be more complex than standard BSTs, but I didn't realize by how much until I researched them and began to code one. I ran into a few difficulties while coding this RBT which caused it to not work correctly at first. My issue was that I assumed that leaves on an RBT function the same way that they do in a BST, that is that a BSTs leaves are the end of a BST, and their left and right children are null pointers. I originally built the RBT the same without realizing that their leaves have left and right children that are not null pointers but are in fact nodes that have a null key. This seemed like a very minor distinction until my code ran into issues with finding some nodes' sibling and uncle nodes. In my first iteration, these didn't always exist, so my code failed. Once I realized this and fixed the issue, the code worked as intended.

The second major thing I learned is how much of an improvement an RBT is over a standard BST. While testing my code, I used two separate data sets for loading information into the data structures. Both had identical information, but one was randomly sorted while the other was sorted by key. With a BST, the randomly sorted data was inserted into the tree quickly (approximately 0.9 seconds) while the sorted data set took significantly longer (about 13.5 seconds). Using the same data sets on an RBT resulted in a load time of about 0.9 seconds for either sorting method. Enforcing balancing to a tree has a significant improvement to its functionality.

References

GeeksforGeeks. (2023, March 28). *Self-balancing binary search trees*. https://www.geeksforgeeks.org/self-balancing-binary-search-trees/

GeeksforGeeks. (2025, May 16). *Insertion in binary search tree (BST)*. https://www.geeksforgeeks.org/insertion-in-binary-search-tree/