# **Objectives**

The goal of this assignment was to write a program that would read a CSV file and implement an algorithm that would add the items from the CSV into a binary search tree with respect to each item’s unique UPC key and the binary search tree properties. Another algorithm was required to traverse the binary search tree using in-order traversal, printing the attributes of the item at each node including its UPC key, amount, and name. The last algorithm that was to be implemented was supposed to use the binary search tree properties to locate an item in the tree with a specific UPC key. Each item’s UPC key, along with the item’s amount and name, could be found in another file titled “input.dat” which also needed to be read in order to compare the key of the target item to keys of those in the binary search tree.

# **Program Design**

Because this assignment required the implementation of a binary search tree and Java is an object oriented programming language, the first thing implemented was a TreeNode class that represented the attributes of the items in the CSV file. A BinaryTree class was then created that contains the necessary functions to create the binary search tree, print each node using in-order traversal, and search the tree for a particular key. Lasty, the HW4 class contains the main() function that is used to call the functions from the other classes and test the algorithms. The following functions are contained within these classes:

**TreeNode()**

TreeNode() is a basic constructor for the TreeNode class that establishes each object’s key, amount, name, and parent. The amount and name are of the String type, while the key is of the type Long, and the parent (or p) of the node is of the type TreeNode as well. The amount is a String instead of an integer because the amounts of each item in the csv file are of different types and most of them contain alphabetical characters. These amounts also do not need to be operated on, only printed. Lastly, all keys within these classes are of the type Long because there are items in the csv file with a UPC key larger than the maximum value possible for a signed 32-bit integer. The type of unsigned integer also would have worked as a solution, as it also can represent values above 2,147,483,647.

**insert()**

Constructs the binary search tree by inserting each TreeNode object created based on the attributes read from the UPC.csv file. Each node is inserted with respect to its unique UPC key and the binary search tree properties. An iterative solution had to be implemented using while loop’s because a recursive solution would result in a stack overflow at runtime due to the function hitting recursion depth. The function determines the correct place for the node to be placed in the tree by iterating through the tree with a while loop, testing an if condition with each loop. The loop continues if the current node is not null, meaning the loop will stop once it has reached a leaf. The if condition tests if the key of the node to be added is less than that of the current node. If so, it sets the current node to the left node, and if not, it sets it to the right node. A second set of if statements outside the while loop tests determine if the node to be added is the left or right child of this node. The first if statement tests if the parent of the node to be added is null, making it the root if so. The second tests if the parent’s key is less node’s key, making it the left child if so and the right child if not. This ensures that less than the node to be added is to the left and any node greater than the node is on the right, following the binary search tree property.

**printInOrderTraversal()**

Uses in-order traversal to visit every node contained in the binary search tree. This code uses a stack to simulate the recursive in-order traversal of a binary search tree. The reason that a recursive solution could not be implemented is because it would result in a stack overflow at runtime due to the function hitting recursion depth. The function starts from the leftmost node and works its way up to the root while printing the nodes in ascending order of their keys. Before the in-order traversal, an if statement tests if the tree is empty and returns null if so, preventing the code from crashing. The function first initializes this stack then continues a while loop if the current node is not null and the stack is greater than zero. A nested while loop traverses the leftmost subtree and pushes it onto the stack. After the nested while loop exits the current node is popped from the stack, and a series of if statements print the node’s key, amount if applicable, and name. Finally, the current node is set to the right child to ensure that the right children nodes are printed as well.

**searchTree()**

Takes advantage of the already established binary search tree properties to locate a specific node using its unique UPC key. The function iterates through the tree using a while loop, continuing if the current node is not null and the target key does not match that of the current node’s key. An if statement tests if the target key is less than the node’s key, assigning the current node to the left node for the next iteration if so, and assigning it to the right node if not. Due to the binary search tree properties, the key must be to the left if it is less and to the right if it is greater, as each nodes’ left child is less and right child is greater. Once the while loop exits, the target key must match the key of the current node, and the function returns this node.

**main()**

This function reads the UPC.csv and input.dat files and calls the functions necessary to build the binary search tree, print each node using in-order traversal, and search for a particular node's key and print the results. First, the function creates a BinaryTree object so that the other functions can be called on it later. Then a scanner object is initialized and called on the CSV file. A while loop iterates through each line of the CSV file. The resulting string is split at each comma using a regex, and each element of the string array is assigned to an attribute of the TreeNode object and constructed. The key attribute is first converted to a long using the parseLong() function. The TreeNode object is then added to items array list. While the insert() function could have been called on each object directly, it was found to be much slower in testing. The arraylist is then iterated over using a for loop, and the insert() function is called on each object. The printInOrderTraversal() is then called on the tree to print every node’s attributes. Next, a new scanned is instantiated and called on the input.dat file. Using a similar method to before, the resulting string is split at each comma using a regex, the first element of each array is converted to a long, and then added to an array list. A for loop then iterates through this array list, calling the searchTree() function on each key and the root of the tree. A series of print statements then print out the result’s key, amount if applicable, and name. An if statement was also added to test if the result is null, ensuring that the code does not crash if the key was not found. The function also records the search and insert functions’ execution time and prints it in nanoseconds, milliseconds, and seconds, then prints the results.

**Code Screenshots:**

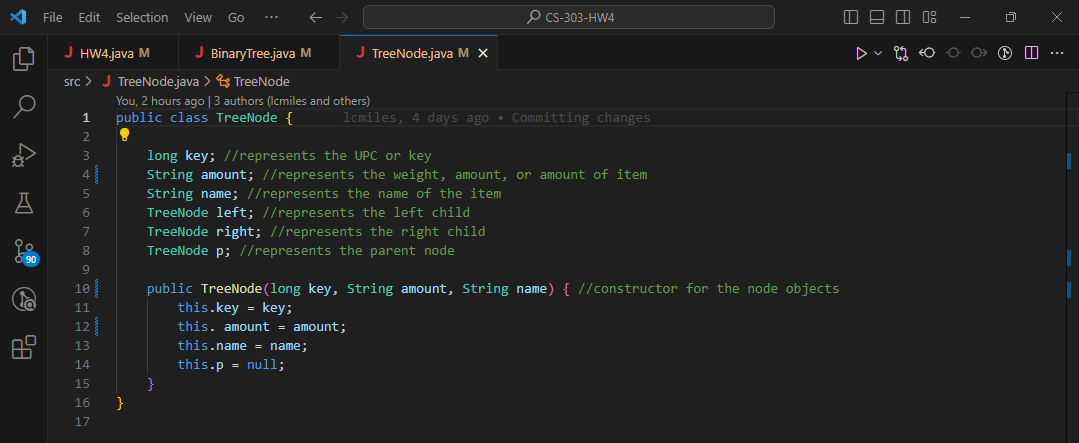


Figure : TreeNode.java

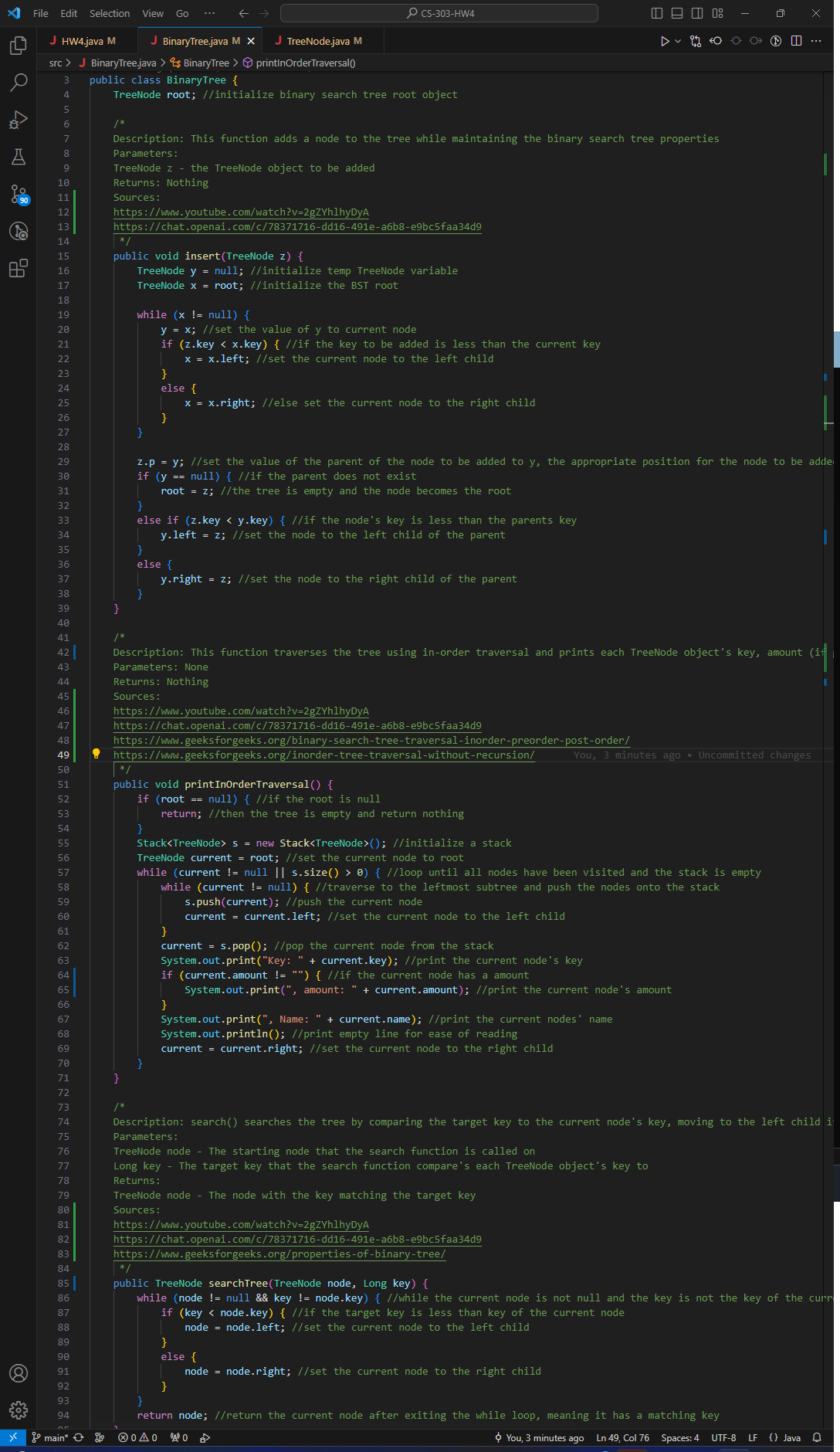


Figure : BinaryTree.java

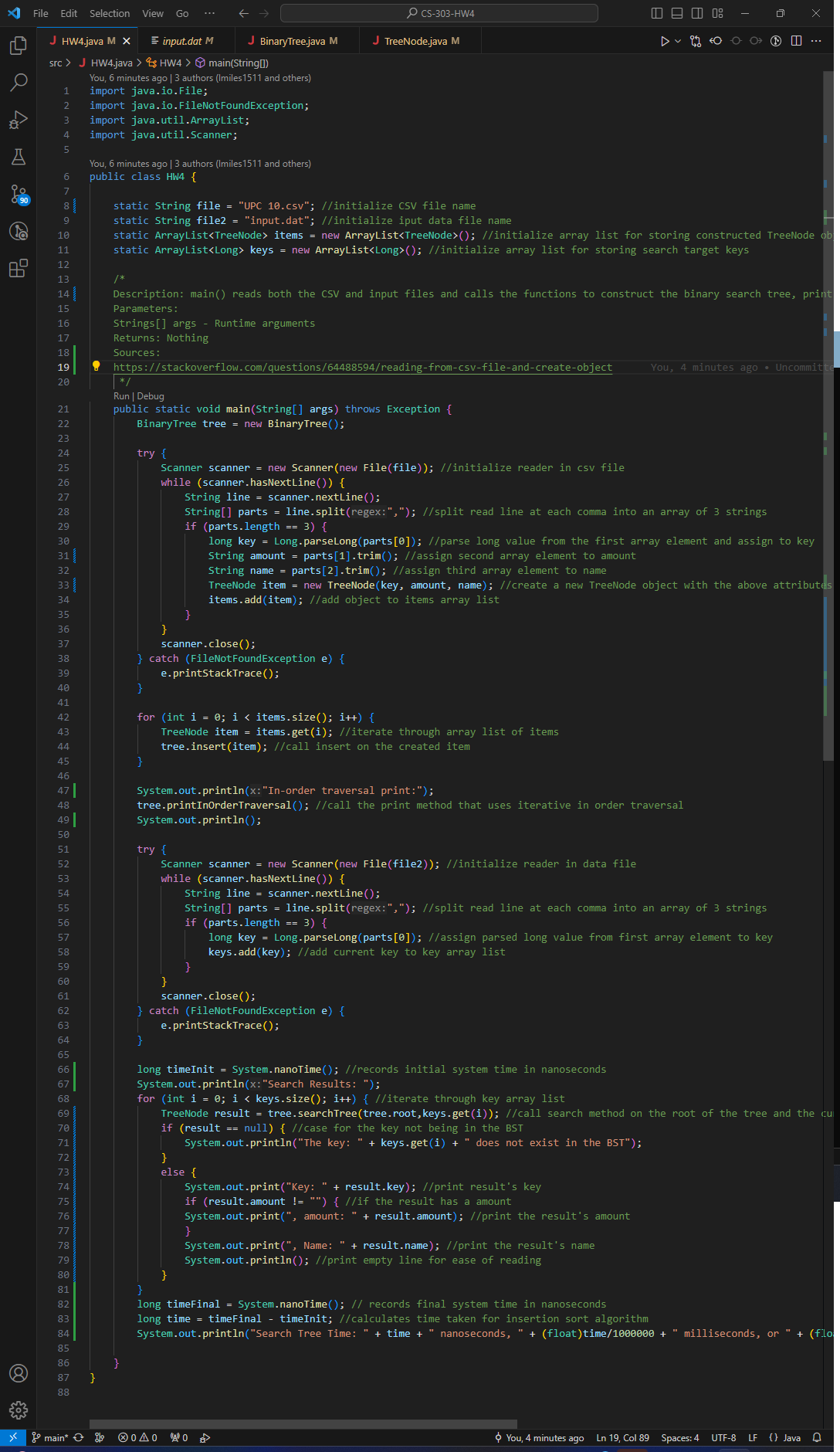


Figure : HW4.java

# **Testing**

The algorithms in this program that were tested include the insert() function, which is the algorithm that builds the binary search tree, and the searching algorithm contained in searchTree(). Both functions were tested using several different versions of the same CSV file, each containing a different number of elements:

1. UPC 10 - 10 elements
2. UPC 100 - 100 elements
3. UPC 1K - 1000 elements
4. UPC 10K - 10000 elements
5. UPC 100K - 100000 elements
6. UPC - The Original CSV file containing 177650 elements

**Screenshots of testing outputs:**

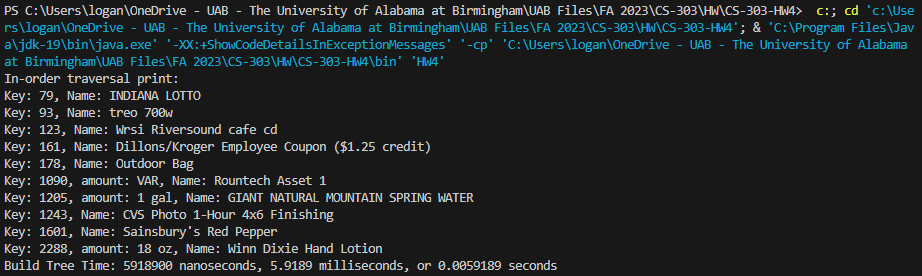


Figure : Insert() UPC 10

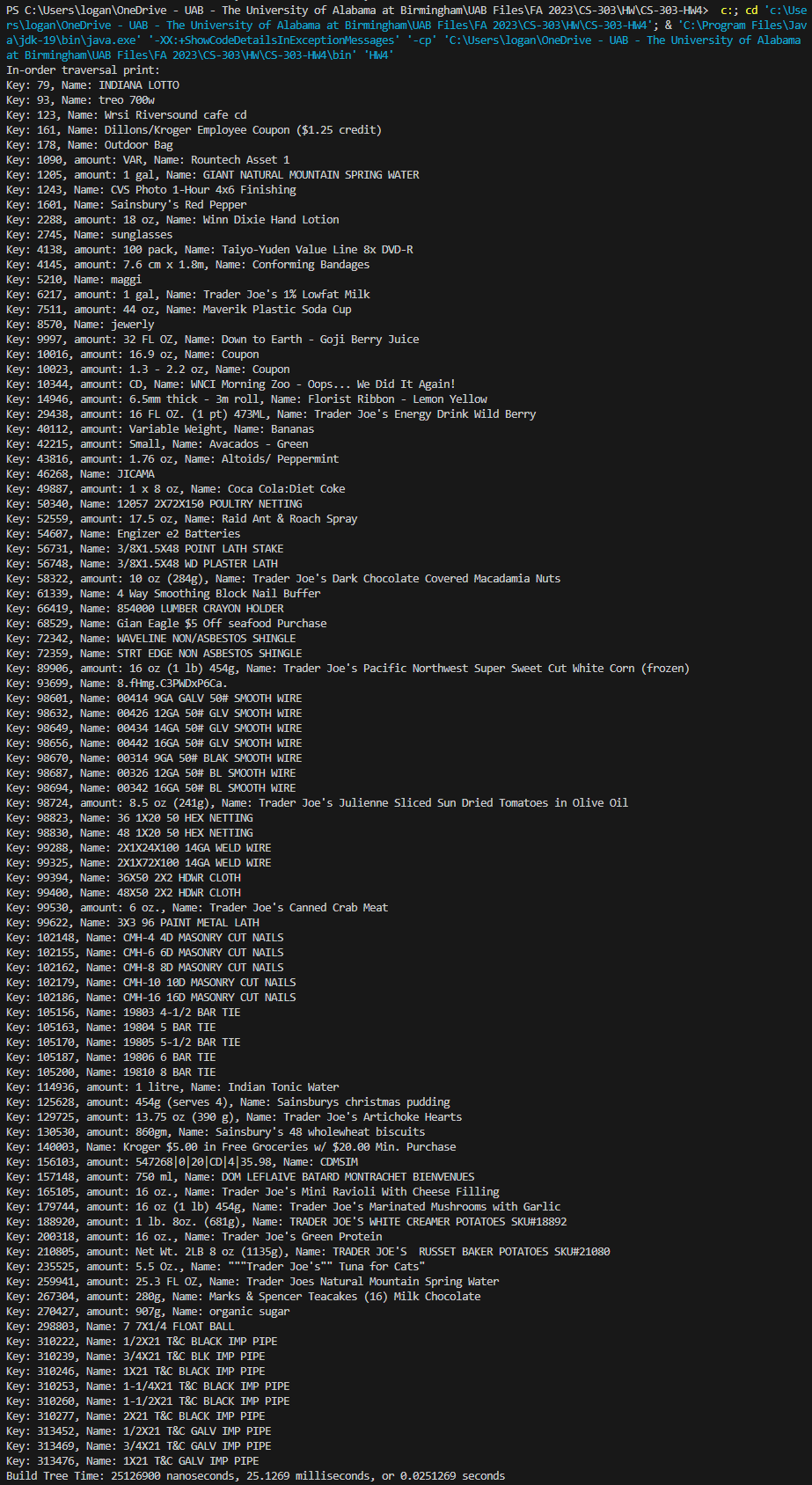


Figure : Insert() UPC 100

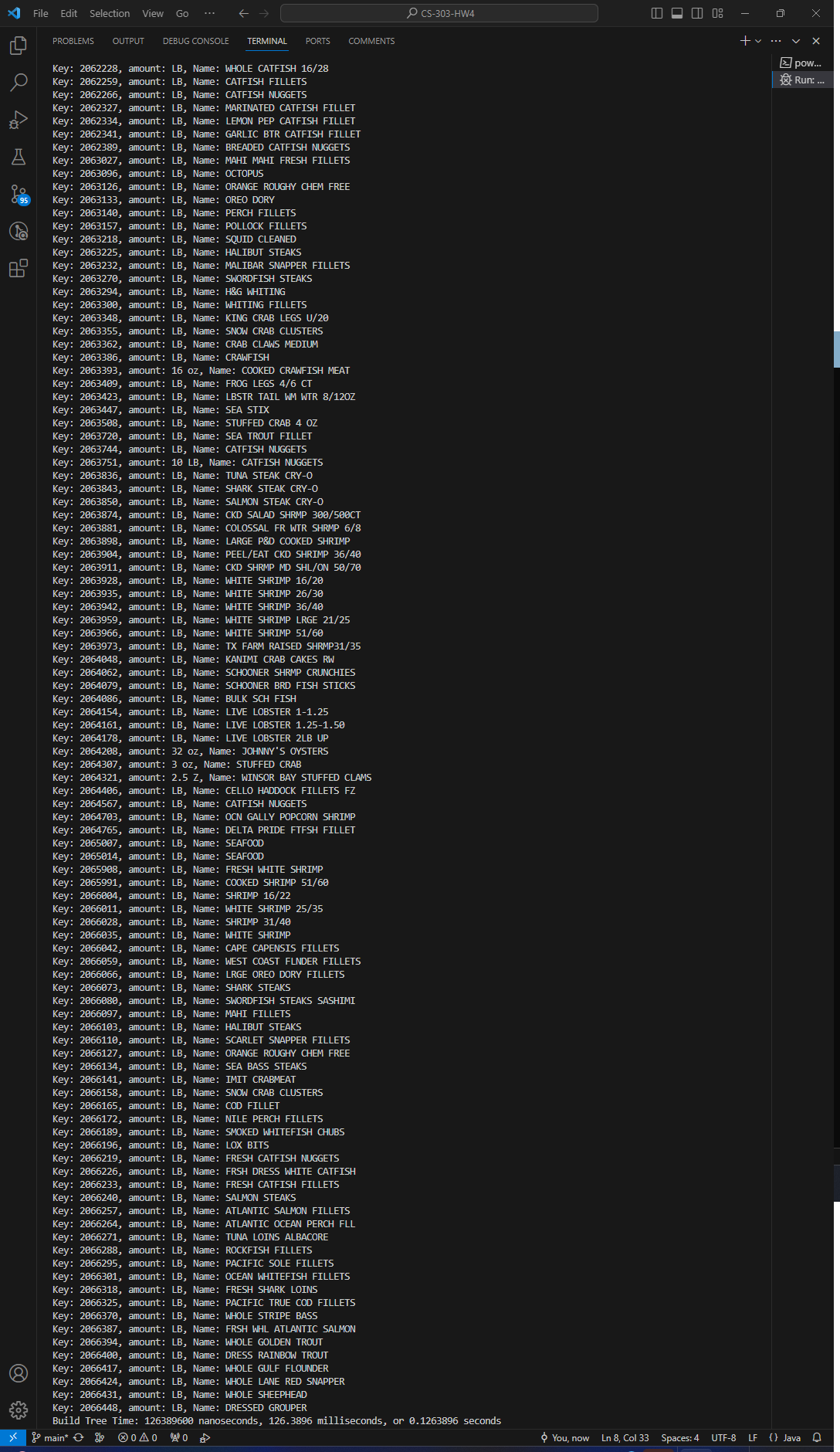


Figure : Insert() UPC 1K

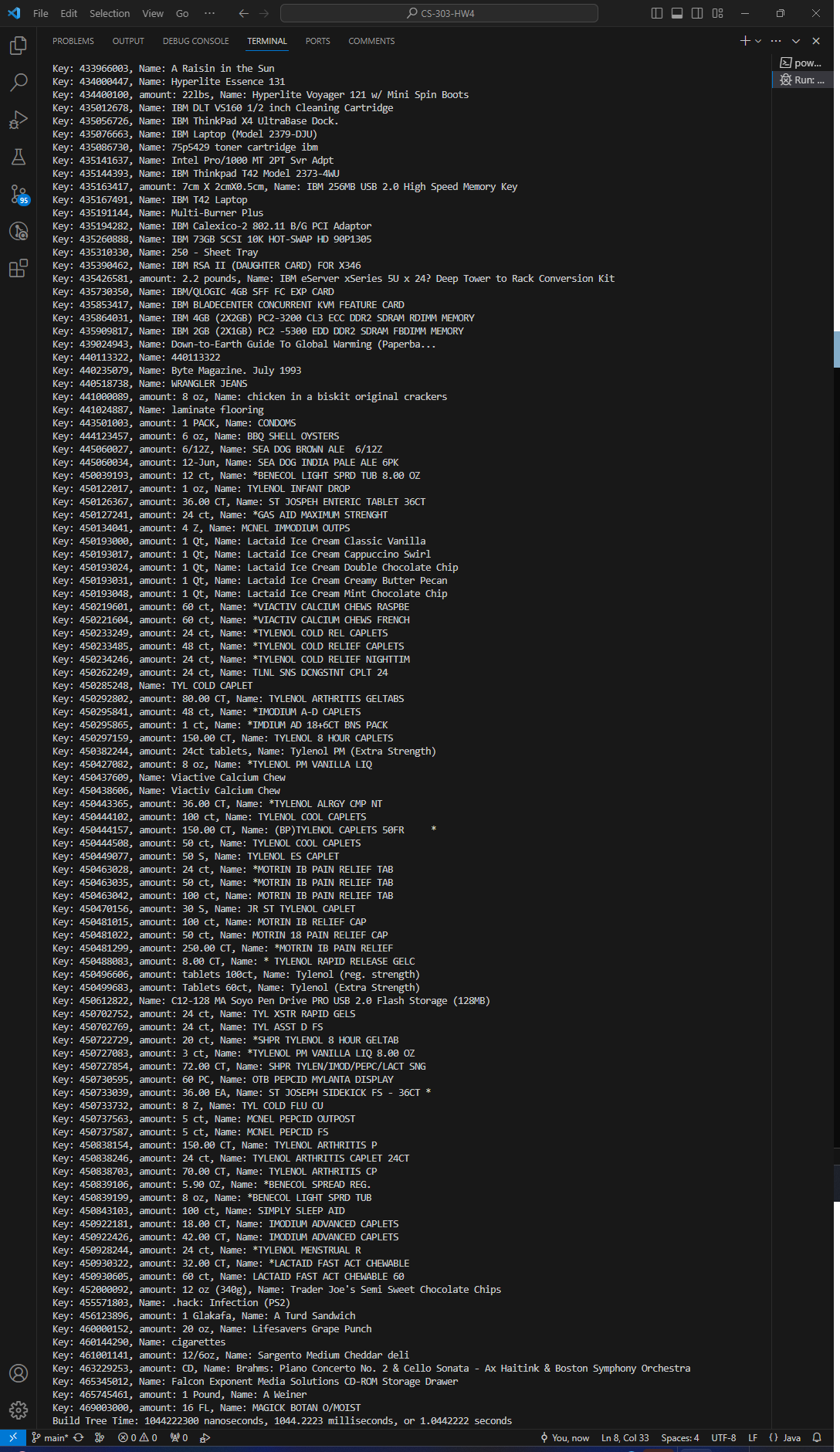


Figure : Insert() UPC 10K

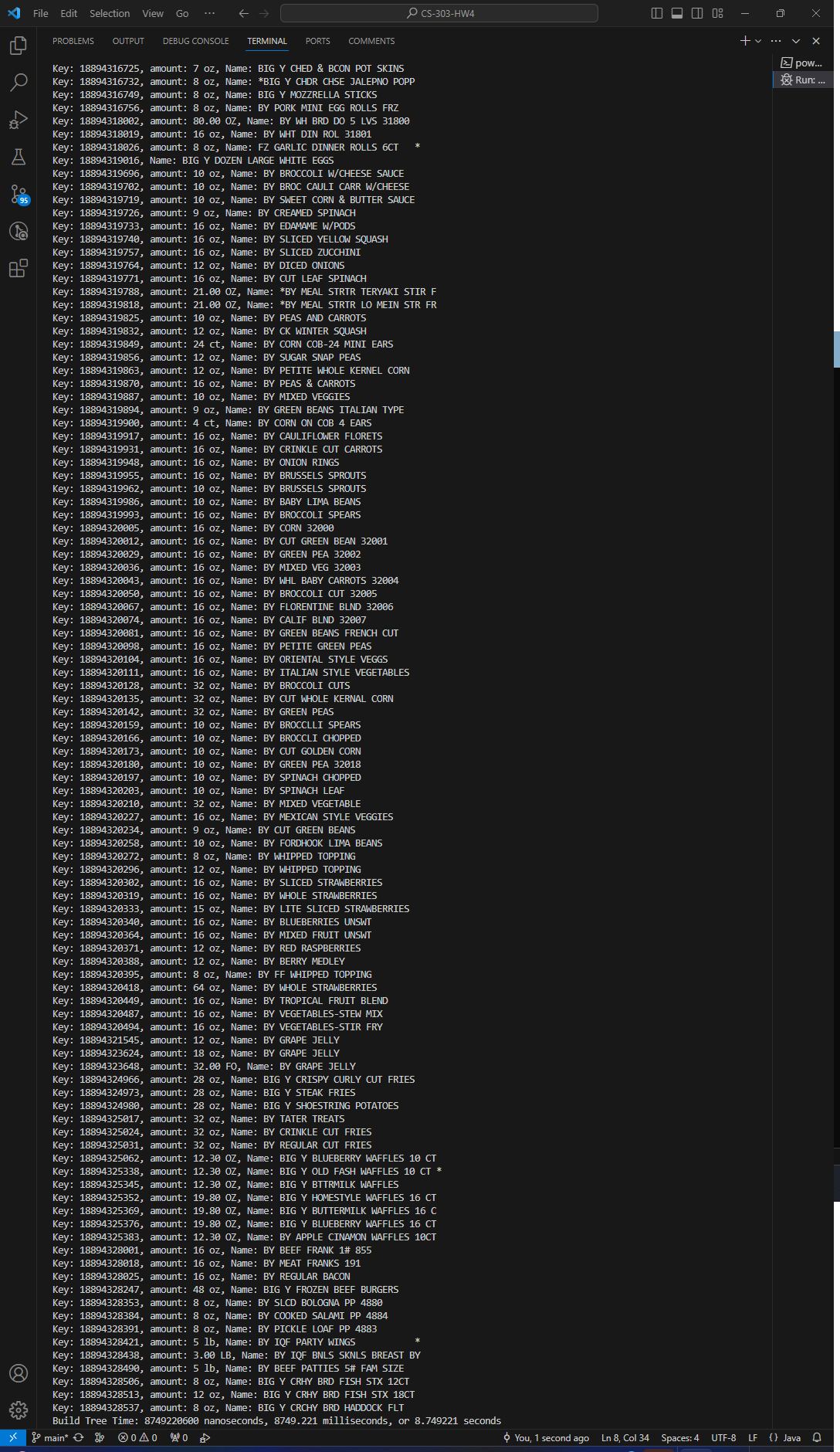


Figure : Insert() UPC 100K

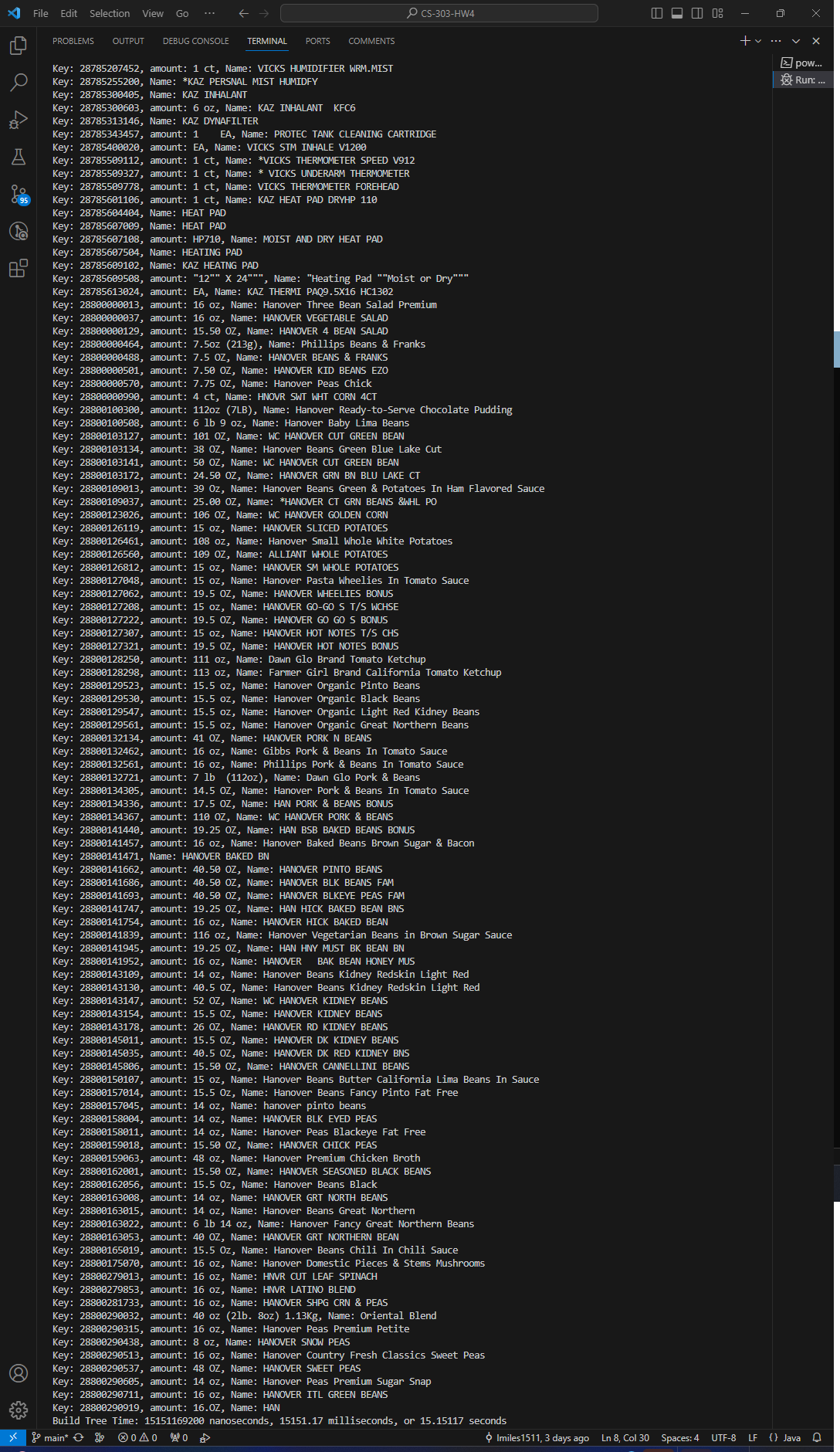


Figure : Insert() UPC

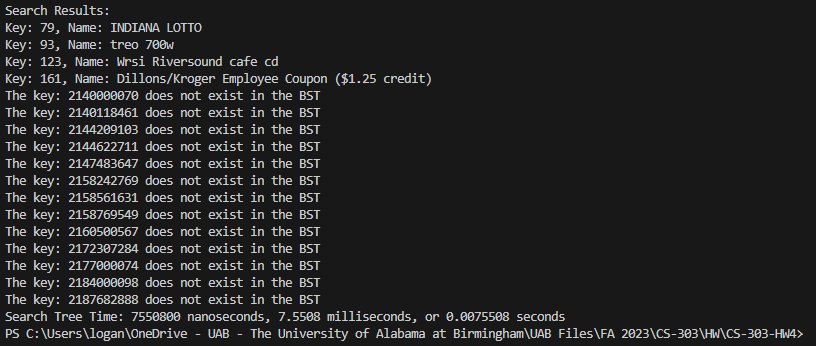


Figure : searchTree() UPC 10

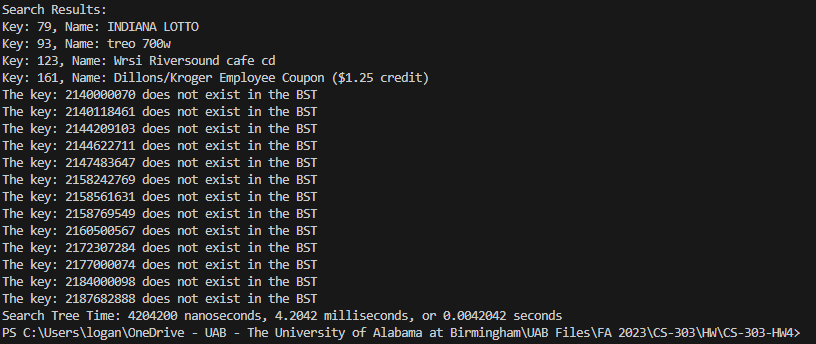


Figure : searchTree() UPC 100

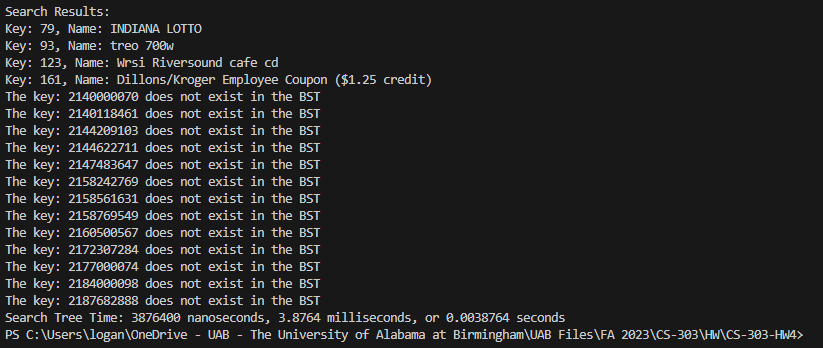


Figure : searchTree() UPC 1K

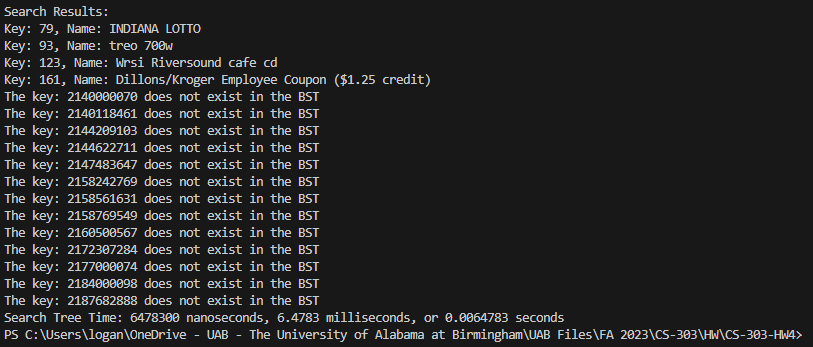


Figure : searchTree() UPC 10K

A screenshot of a computer

Description automatically generated

Figure : searchTree() UPC 100K

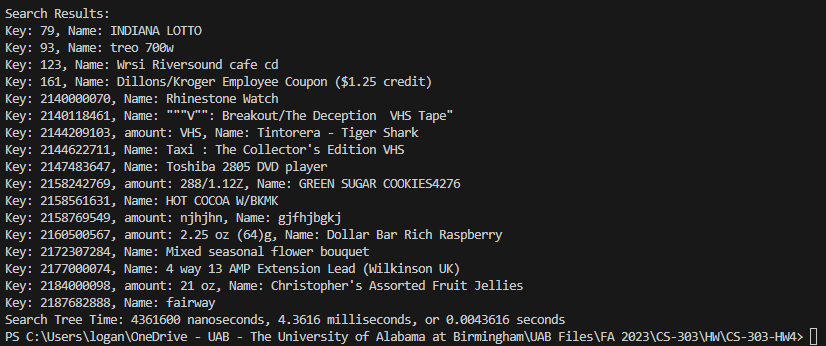
Figure : searchTree() UPC

Figure : insert() Execution Time vs Number of Elements in CSV

Figure : searchTree() Execution Time vs Number of Elements in CSV

The data above clearly shows that the time complexity of the insert() function is linear, or O(n). The time complexity of the search() function is not so easily discernable, however, as just by looking at the graph it would seem there is no connection between the execution time of the algorithm and the number of elements operated on. This isn’t reflective of the algorithm’s actual time complexity, which is O(log(n)). This is because the algorithm is similar to binary search, in that it divides the tree that needs to be searched in half with each iteration, resulting in a time complexity of O(log(n)) on average and O(n) in the worst case. The reason that the graph poorly reflects the actual time complexity of the algorithm could be due to the scaling of the datasets or the fact that the same search keys were used for each test. In the future, it could be more effective to randomize the keys with each test.

# **Sources**

<https://stackoverflow.com/questions/64488594/reading-from-csv-file-and-create-object>

<https://www.youtube.com/watch?v=2gZYhlhyDyA>

<https://www.geeksforgeeks.org/binary-search-tree-traversal-inorder-preorder-post-order/>

<https://www.geeksforgeeks.org/properties-of-binary-tree/>

<https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/>

<https://chat.openai.com/c/78371716-dd16-491e-a6b8-e9bc5faa34d9>