Name: Saarthak Mudigere Girish
UTA ID: 1002119262

Design & Analysis of Algorithms CSE 5311-004

Implementation of Project 2

(Search Algorithm)

Main Data Structures

- **Linear Search**: Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that item is returned, otherwise the search continues till the end of the data collection.

- **Binary Search**: Binary search is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues the remaining half, again taking the middle element to compare to the target value and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array. Binary search runs in logarithmic time in the worst case, making O(log n) comparisons, where n is the number of elements in the array. Binary search is faster than linear search except for small arrays. However, the array must be sorted first to be able to apply binary search.

- **Binary Search Tree**: Binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree data structure whose internal nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtree. A binary tree is a type of data structure for storing data such as numbers in an organized way. Binary search trees allow binary search for fast lookup, addition, and removal of data items, and can be used to implement dynamic sets and lookup tables. The order of nodes in a BST means that each comparison skips about half of the remaining tree, so the whole lookup takes time proportional to the binary logarithm of the number of items stored in the tree.

- **Red-Black Tree**: Red–black tree is a kind of self-balancing binary search tree. Each node stores an extra bit representing "color" ("red" or "black"), used to ensure that the tree remains balanced during insertions and deletions. When the tree is modified, the new tree is rearranged and "repainted" to restore the coloring properties that constrain how unbalanced the tree can become in the worst case. The properties are designed such that rearranging and recoloring can be performed efficiently. The re-balancing is not perfect but guarantees searching in O (log n) time, where n is the number of nodes of the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in O (log n) time. Tracking the color of each node requires only one bit of information per node because there are only two colors. The tree does not contain any other data specific to its being a red–black tree, so its memory footprint is almost identical to that of a classic (uncolored) binary search tree. In many cases, the additional bit of information can be stored at no additional memory cost.

Main components of the algorithm

- **Linear Search:** Linear search is implemented using only one function which takes the array and the key to be searched as the input. Inside the function, a loop is started to iterate through the entire array and each element is checked against the key entered. If there is a match, then the function returns the position and time it took to find the key, if not, then it returns the value -1.

- **Binary Search:** Binary search is also implemented using only one function which takes in input as the array and the key to be found. We loop though the entire array until the condition that the start index is less than or equal to the end index and during each iteration of the loop, we check if the array value of that index is equal to the key, if not we check is it greater than or less than the key. If it is found to be greater than the key then we reassign the end index to the middle -1, if it's found to be lesser than the key then we reassign the start to the middle +1. If there is a match, then the function returns the position and time it took to find the key, if not, then it returns the value -1.

- **Binary Search Tree:** Binary search tree is implemented using two functions. insert(data) is the function used to insert each array value as a node in the tree. We define a class called Node to help us keep track of the left and the right value of each node in the tree. Now when a value is passed into the function, the first check we perform is if there exists a root node or not. If not, then we make the first value as the root node. If it's not the first value, then we check if the entered value is less than or greater than the node at comparison. If is less than the current node, then we move to the left of that node and check if there exists any index there to see if there is already a value present. If no value is present, then we add the value there, if there is any value then we perform the above logic again to see on which side of the node the value must go to. Once the insertion is performed, we have a function called findval(data) which is used to find the value in the tree. The logic is like that of insertion but once we perform a check with the current node, if the value is not found, we check if the value of the node was greater or lesser than the key. If the value was greater, then we move towards the right of the tree and if the value was lesser, then we move towards the left of the tree. We continue till we find the node. If the node is found, we return that node, else we return -1.

- **Red-Black Tree:** The Red Black Tree algorithm is implemented using 5 functions. The insertNode(key) function is used to insert value in the red-black tree configuration which first checks if there exists a root node or not. If not, then it assigns the current value to root node with a black color denoted by 0. If the root value does exist, then we check if the value entered is less than or greater than the current node. If the value is less than the current node, then we insert a node on the left and if the value is greater than the current node, then we insert a node on the right. Then we check if the new node's parent is a root

node or not. If it is a root node, then we move onto the next value to be inserted and if not, then we check the color of the parent. If it is black, then no conflict is present, and we can move onto the next value to be inserted. If it is red, then conflict is present, and we call the function fixInsert(node). Here, we check if the current node's parent is a left child or a right child. Then we check if the current nodes uncle is a black or a red color. If it is a red color, then we recolor the current node's parent and uncle to black color and recheck; and if it is black or null, then we check if the current node is left or right child of its parent. Depending on the nodes, we perform appropriate right or left rotations by calling the LR(node) or RR(node) and recheck. Once the insertion is performed, we have a function called __searchValue(key) which is used to find the value in the tree. The logic is like that of insertion but once we perform a check with the current node, if the value is not found, we check if the value of the node was greater or lesser than the key. If the value was greater, then we move towards the right of the tree and if the value was lesser, then we move towards the left of the tree. We continue till we find the node. If the node is found, we return that node, else we return -1.

Design of the user interface

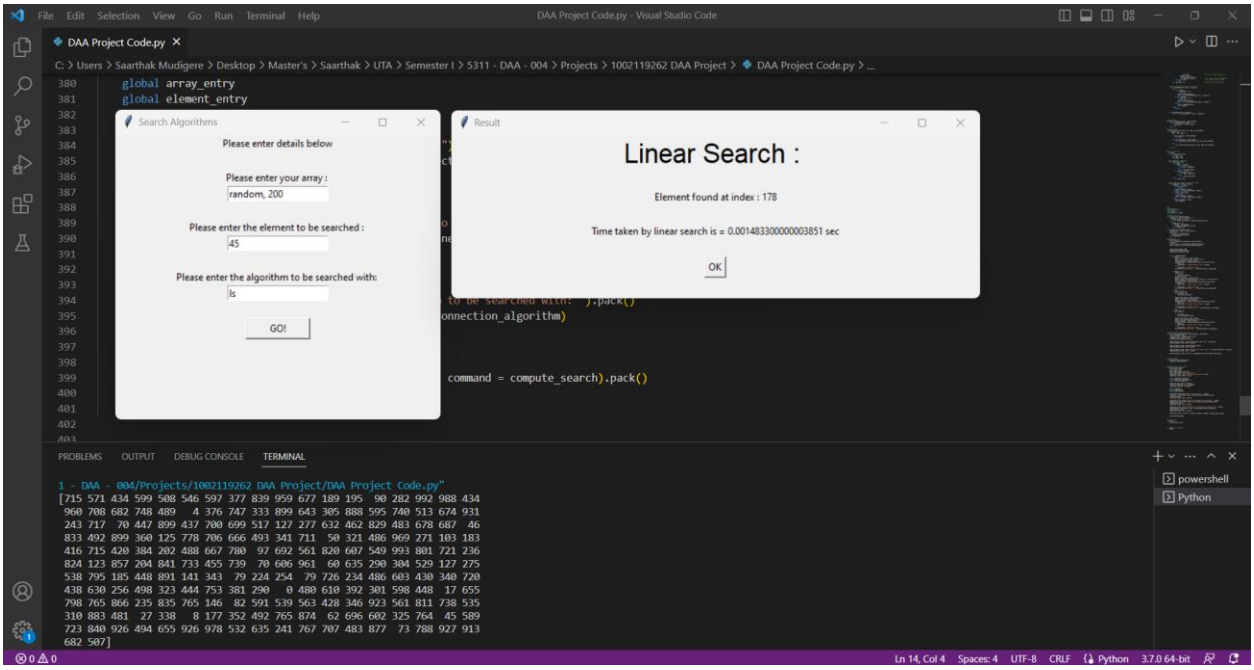
The graphical user interface uses tkinter. The foundational element of a tkinter GUI is the window. Windows are the containers in which all other GUI elements live. These other GUI elements, such as text boxes, labels, and buttons, are known as widgets. Widgets are contained inside of windows. Here, we have created two types of windows with respective widgets:

1. Primary Window (Search Algorithms):
   There are three field in the GUI where the user must input the following:
   a. Enter search array: Here the user must enter "random" followed by array size for random array or must enter the integers on which the search operations have to be performed. They have to be comma separated values, for example: 2,4,10,5.
   b. Enter search number: Here the user must enter the integer value that has to be searched in the above-mentioned array, for example: 5.
   c. Choose algorithm: Here the user must choose algorithm to be used and enter the value in this field. For example:
      i. ls (for linear search)
      ii. bs (for binary search)
      iii. bst (for binary search tree)
      iv. rb (for red-black tree)
      v. ls+bs (for linear search and binary search)
      vi. bst+ls (for binary search tree and linear search)
      vii. bs+bst+rb (for binary search, binary search tree and red-black tree)
      viii. ls+bs+bst (for linear search, binary search, and binary search tree)
      ix. ls+bs+bst+rb (for all)

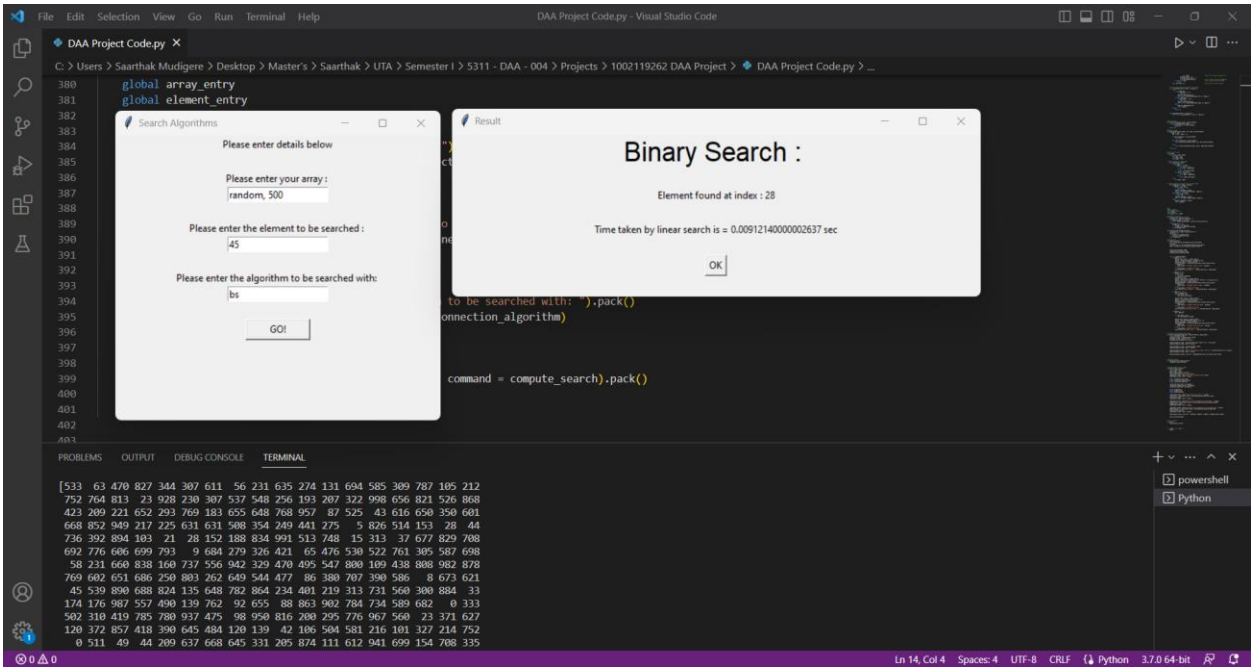
2. Secondary Window (Result):
   There are three field in the GUI:
   a. Algorithm: Here it shows the Algorithm chosen by the user, for example: Binary Search
   b. Output: Here we display if the element is found or not and its appropriate index or path, for example: Element found at index: 5
   c. Time taken: Here the time taken to execute is displayed, for example: 0.002153 sec.

Experimental results

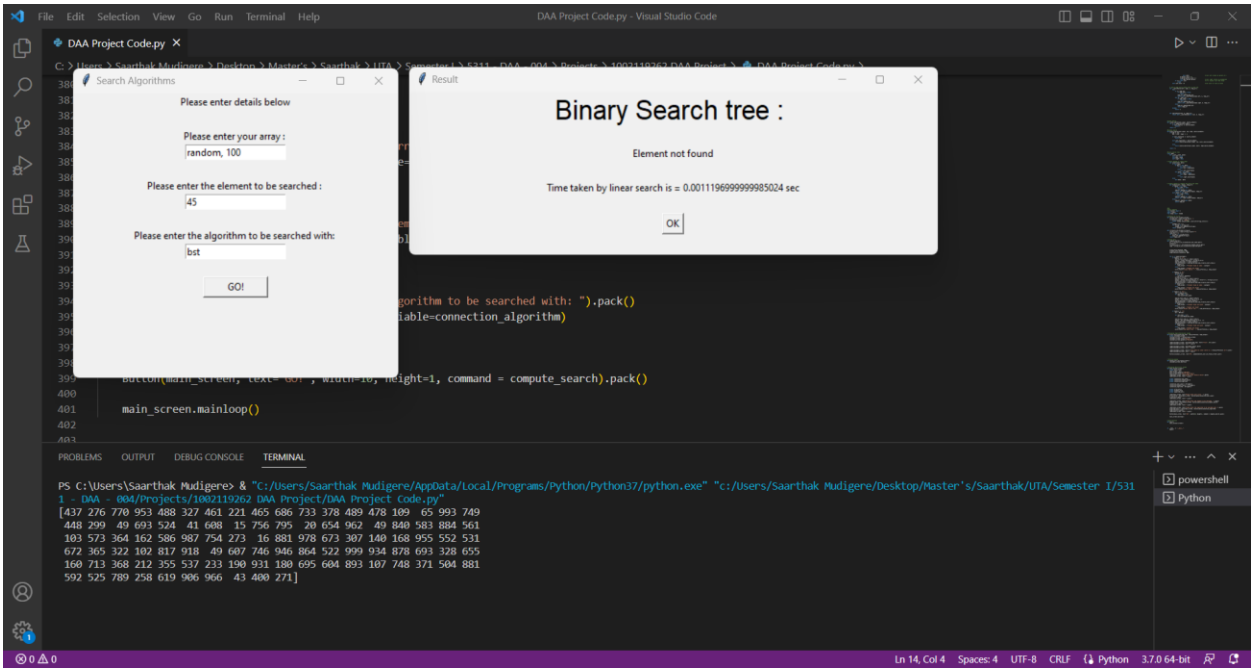
1. Linear search with 200 inputs



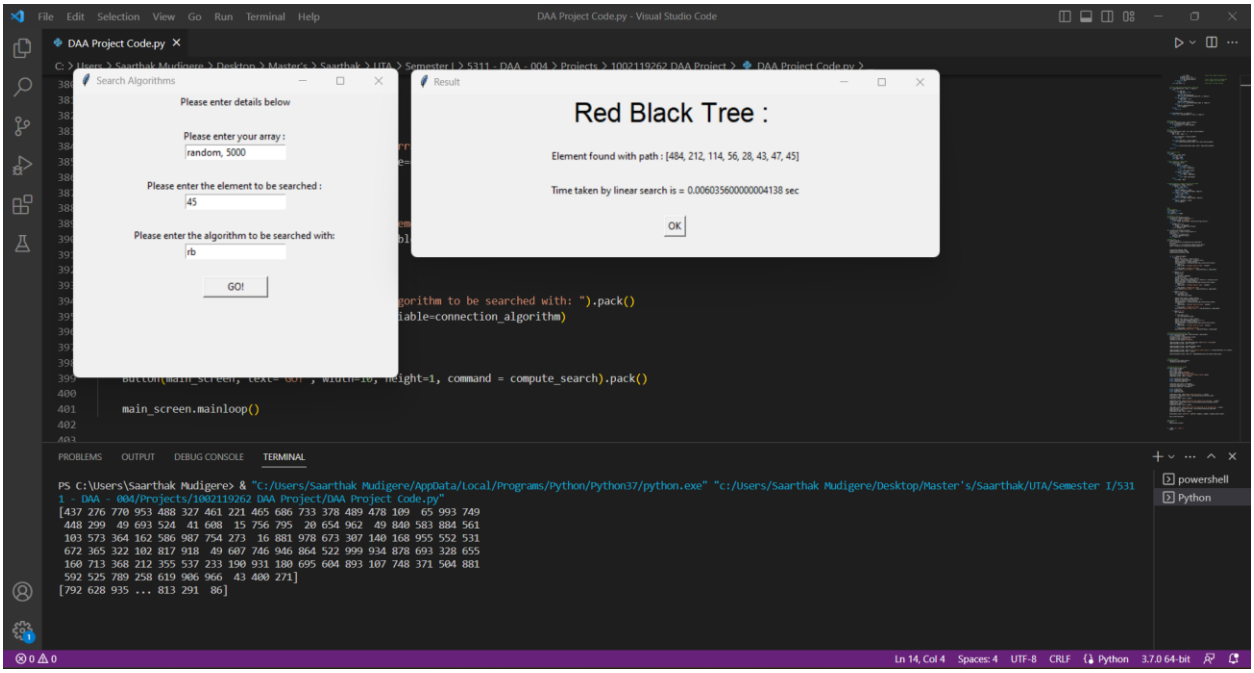
2. Binary search with 500 inputs



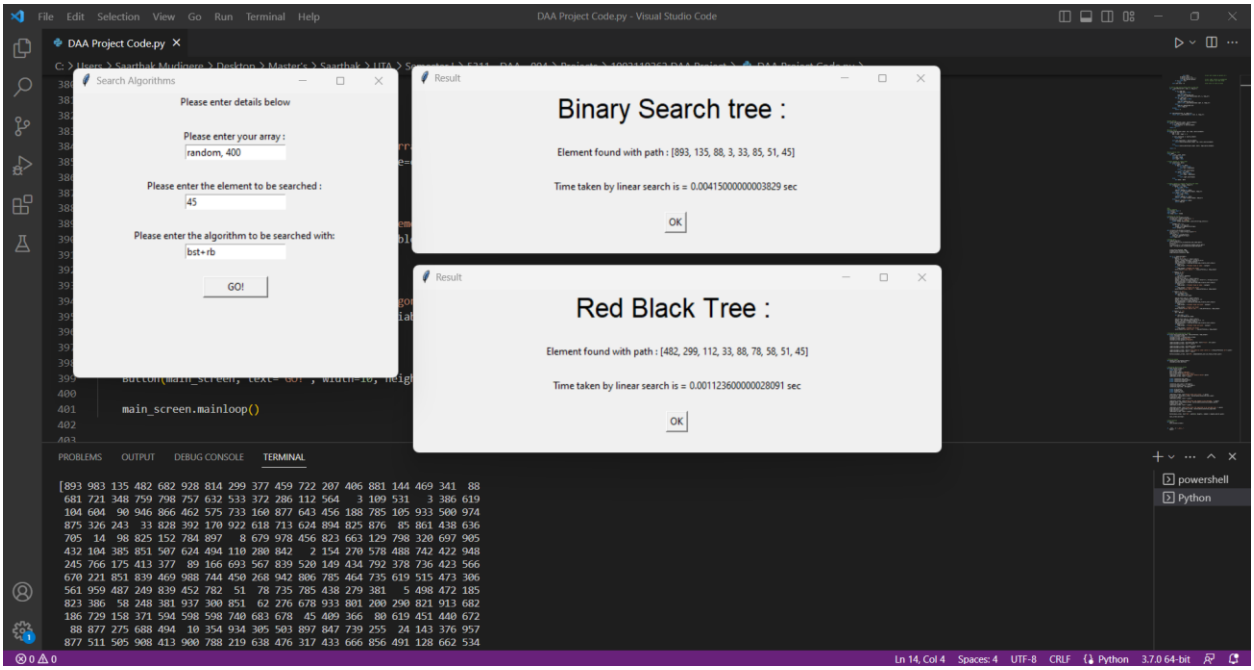
3. Binary search tree with 100 inputs



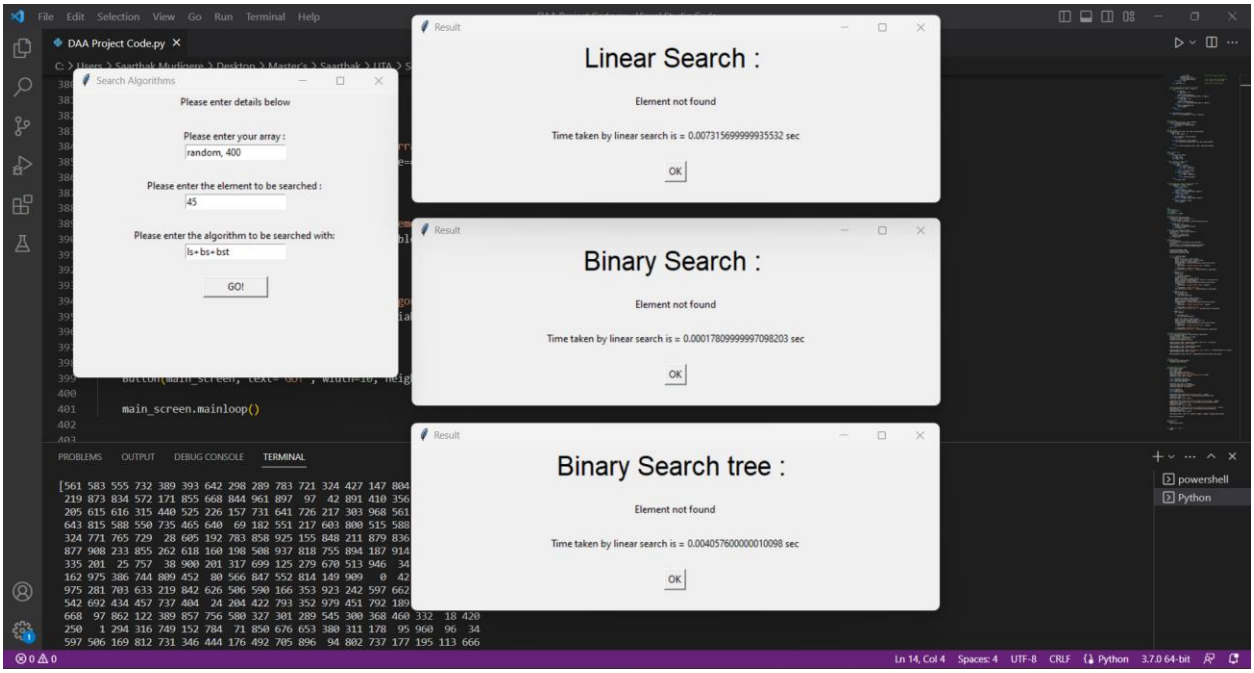
4. Red black tree search with 5000 inputs



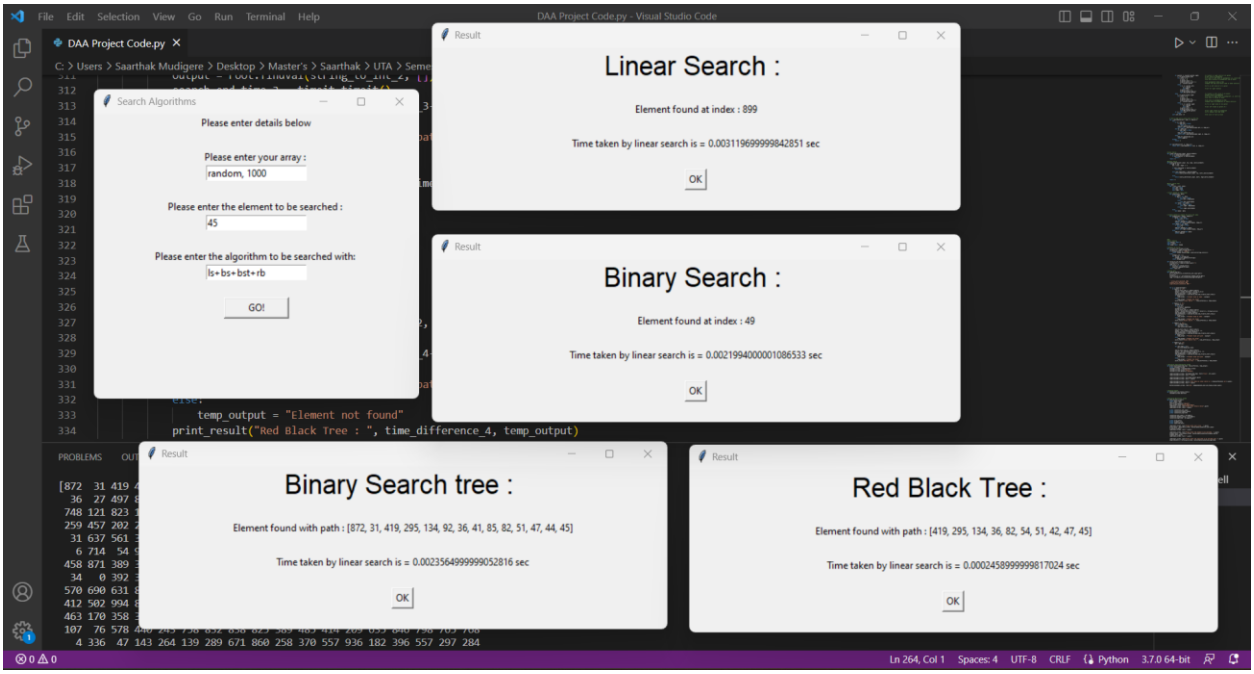
5. Binary search tree and red-black tree comparison with 400 inputs



6. Linear search, Binary search, and Binary search tree comparison for 600 inputs



7. Comparing all algorithms.



Chart

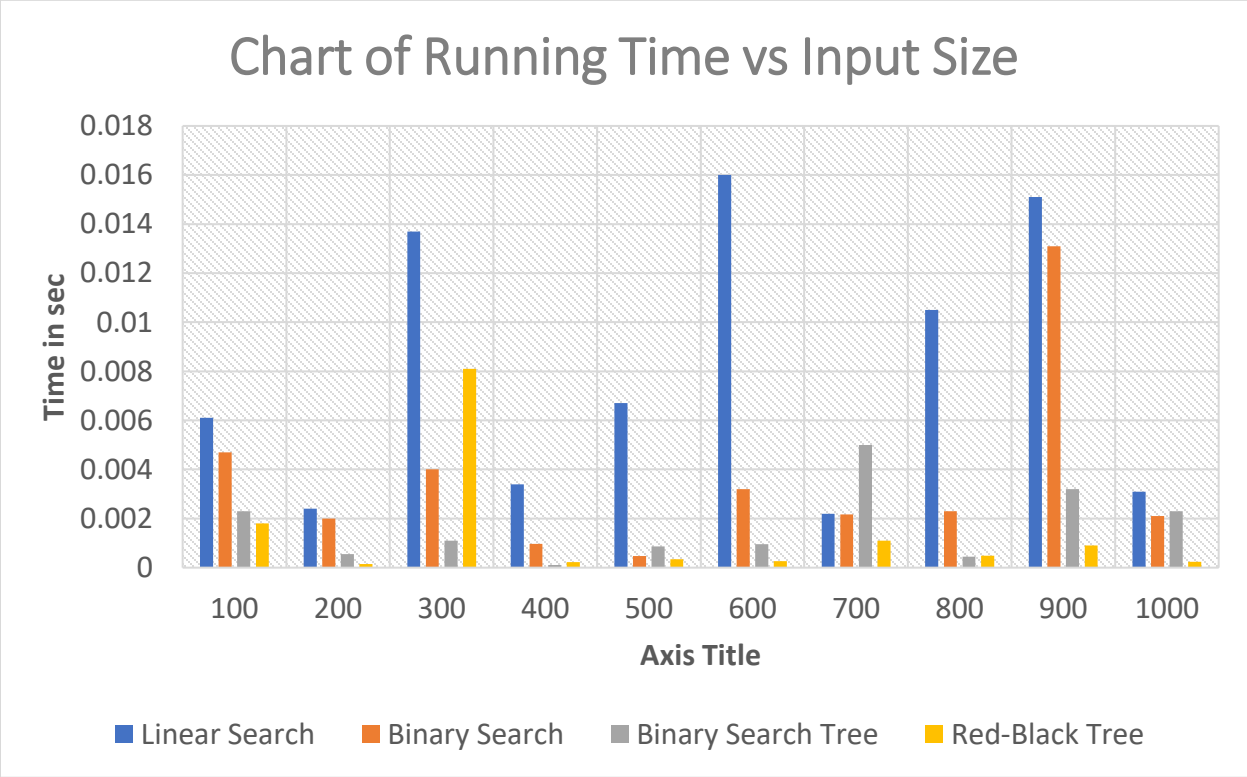


Chart of Running Time vs Input Size
Time in sec
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
100
200
300
400
500
600
700
800
900
1000
Axis Title
Linear Search
Binary Search
Binary Search Tree
Red-Black Tree