DESIGN AND ANALYSIS OF ALGORITHMS PROJECT

IMPLEMENTATION OF SEARCH ALGORITHMS

LINEAR SEARCH | BINARY SEARCH | BINARY SEARCH TREE | RED BLACK TREE

Submitted By Ankith Reddy Avula UTA ID – 1002090721 and Sravan Chandaka UTA ID – 1002059166

UNIVERSITY OF TEXAS ARLINGTON

Submitted as a part of the course project for CSE 5311 Design and Analysis of Algorithms under Dr. Junzhou Huang

Search Algorithms:

Linear Search:

Linear search is a search algorithm where an element is searched by traversing through the array of numbers and comparing each element of the array for the searched element if element is found in the array then the index value of the element is returned. Since the algorithm searches the entire array, the time complexity of the algorithms is O(n).

Algorithm:

```
def linear_search(arr, ele):
    """Linear Search Implementation

    Args:
        arr (list[int]): An array to be searched for
        ele (int): Element to be searched

    Returns:
        int: If the element is found returns the index of element in arr else returns -1
    """
    idx = 0
    for i in arr:
        idx += 1
        if i == ele:
            return idx
    return -1
```

Binary Search:

Binary Search algorithm is implemented when the given search array is a sorted array. The element which is to be searched is compared to the median of the array and then accordingly the comparison goes to left or right half of the array depending on whether the element is smaller or larger than the element. If the search element is equal to median, then the index value of mid is returned. The time complexity of this algorithm is O(lg(n)).

Algorithm:

```python
def binary_search(arr, low, high, k):
    """Binary Search implementation

    Args:
        arr (list[int]): Sorted array
        low (int): Lowest index in partition of Binary Search Array
        high (int): Highest index in partition of Binary Search Array
        k (int): Value to be found in array arr

    Returns:
        int: index of the searched value if found else returns -1
    """
    if high >= low:
        mid = (low + high) // 2
        mid = low + (high - low) // 2
        if arr[mid] == k:
            return mid
        elif arr[mid] > k:
            return binary_search(arr, low, mid-1, k)
        else:
            return binary_search(arr, mid+1, high, k)
    else:
        return -1
```

Binary Search Tree:

Binary Search Tree is a data structure made from an array. The tree has nodes with at most two leaves. The values on left side of the parent node are smaller or equal to value of the parent node. The values on the right side of parent node are greater than the value in the parent node. The algorithm does not require a sorted array. The time complexity to search for an element is at most O(lg(n)) i.e., traversing through the entire height of the tree. The Binary Search Algorithms is implemented using a class Node where it has the left, right pointers and value of the node. The root node is initialized in the beginning of the program and all other nodes are linked to the root node forming a tree.

Algorithm:

Basic Structure of a node in Binary Search Tree

```
class Node:
    """Basic Structure of a Binary Search Tree Node
    """

    def __init__(self, data) -> None:
        self.left = None
        self.right = None
        self.value = data
        self.parent = None
```

Insertion Algorithm for a Binary Search Tree

```
def insertIntoBST(root, data):
    """This is an insert function for Binary search tree it inserts the values less than or equal to root in the left of root
and values greater than root into right of root.

    Args:
        root (Node): Root Node of Binary Search Tree
        data (int): Value to be inserted in binary search tree
    """
    x = root
    y = None
    z = Node(data)

    while(x != None):
        y = x
        if z.value <= x.value:
            x = x.left
        else:
            x = x.right

    if z.value <= y.value:
        y.left = z
    else:
        y.right = z
    z.parent = y
```

Search Algorithm for a Binary Search Tree

```
def searchTree(root, data):
    """Searches the BST for given value data and returns the index if found else returns -1

    Args:
        root (Node): Root Node of BST
        data (int): Data Value to be searched

    Returns:
        int: Index if value is found else returns -1
    """
    if root is None:
        return False
    if root.value == data:
        return True
    if root.value < data:
        return searchTree(root.right, data)
    else:
        return searchTree(root.left, data)
```

Red-Black Tree:

A Red-Black Tree is a variant of binary tree where it follows certain properties to have it height maintained so that the worst-case scenario search complexity is always O(lg(n)). The Red-Black Tree has nodes of two colors red and black. The root node is always colored red and leaf nodes including the NULL nodes are colored black. The nodes are colored in a manner where two red nodes are not found consecutively. The Basic Node Structure in a Red Black is same with an additional attribute color for the node. The Search Algorithm for a Binary Search Tree and Red Black Tree is similar. The Insertion algorithm is also similar until the last step where the tree balancing happens according to the different cases on how to arrange the coloring and values in the Red Black Tree.

Algorithm:

Basic Node Structure in Red Black Tree:

```
class Node():
  def __init__(self, data) -> None:
    self.value = data
    self.parent = None
    self.left = None
    self.right = None
    self.color = 1
```

Basic Red Black Tree Structure:

```
class RBTree():
  def __init__(self) -> None:
    self.temp = Node(0)
    self.root = self.temp
```

Insertion Algorithm:

```
def insert(self, key):
    """Insertion Algorithm for a Red Black Tree followed by a insertion algorithm fixup.

    Args:
      key (int): value of new node to be inserted
    """
    node = Node(key)
    node.parent = None
    node.item = key
    node.left = self.temp
    node.right = self.temp
    node.color = 1

    y = None
    x = self.root

    while x != self.temp:
      y = x
      if node.item < x.item:
        x = x.left
      else:
        x = x.right

    node.parent = y
    if y == None:
      self.root = node
    elif node.item < y.item:
      y.left = node
    else:
            y.right = node

        if node.parent == None:
            node.color = 0
            return

        # if node.parent.parent == None:
        #     return

        self.fix_insert(node)
```

Tree Balancer (Insertion Fixup):

```
def fix_insert(self, k):
        """Tree Balancing Algorithm

        Args:
            k (Node): Node inserted during insertion Algorithm
        """
        while k.parent.color == 1:
            if k.parent == k.parent.parent.right:
                u = k.parent.parent.left
                # case 1 in rb fixup
                if u.color == 1:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    # case 2 in rb fixup
                    if k == k.parent.left:
                        k = k.parent
                        self.right_rotate(k)
                    # case 3 in rb fixup
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.left_rotate(k.parent.parent)
            else:
                u = k.parent.parent.right
                # case 1 in rb fixup
                if u.color == 1:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    # case 2 in rb fixup
                    if k == k.parent.right:
                        k = k.parent
                        self.left_rotate(k)
                    # case 3 in rb fixup
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.right_rotate(k.parent.parent)
            if k == self.root:
                break
        self.root.color = 0
```

Left rotation and right rotation Algorithms:

```
def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.temp:
            y.left.parent = x

        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y

    def right_rotate(self, x):
        y = x.left
        x.left = y.right
        if y.right != self.temp:
            y.right.parent = x

        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y
```

<u>Method, Results, and Observation</u>:

Time Complexity:

	Best Case	**Average Case**	**Worst Case**
Linear Search	O (1)	O(n)	O(n)
Binary Search	O (1)	O (log n)	O (log n)
Binary Search Tree	O (1)	O (log n) or O(h)	O(n)
Red-Black Tree	O (1)	O (log n) or O(h)	O (log n) or O(h)

1) We used a Random integer (random.randint()) generator to randomly generate the values between 0 to 1000000.
2) And also the key Index number would be picked randomly between 0 to 100000.
3) And implemented a condition, that the key should definitely be present in the array index to be searched.
4) Used perf_counter_ns() to calculate the execution time or Time Complexity, perf_counter_ns will help to print the time complexity in nano seconds(1 sec =10^-9 ns)
5) Time complexity of different searching techniques is calculated without including the sorting time calculation for the Binary search, BST, and RBT building time of the tree.
6) Only including their search execution time will clearly depict their significance in the search techniques and we can clearly see with the algorithm performs well in different scenarios.
7) In the process of testing, we considered different array sizes like [100, 500, 1000, 5000, 10000, 50000, 100000, 150000, 2000000] to know the best and worst case possibilities.
8) And also by performing the randomized searching techniques 20 times subsequently, we could able to show the best possible Average case scenario for all 4 searching algorithms.
9) Using Matplotlib, considering the different sizes, and execution times as the X and Y axis, we plotted the graphs of all 4 different search algorithms. Through this, we showed the clear difference between linear search, Binary search, Binary Search Tree, and Red Black Tree.

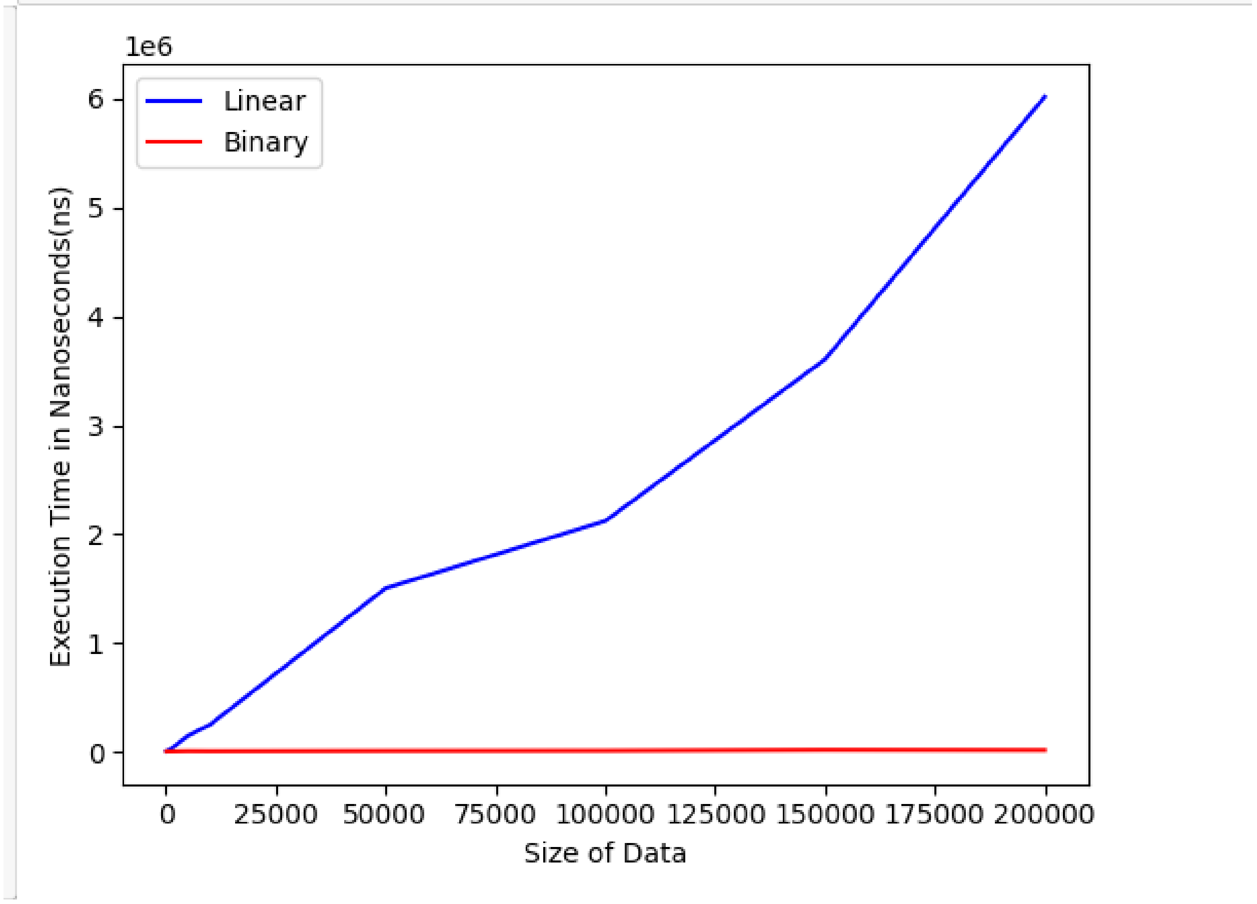
Performance and Comparison:

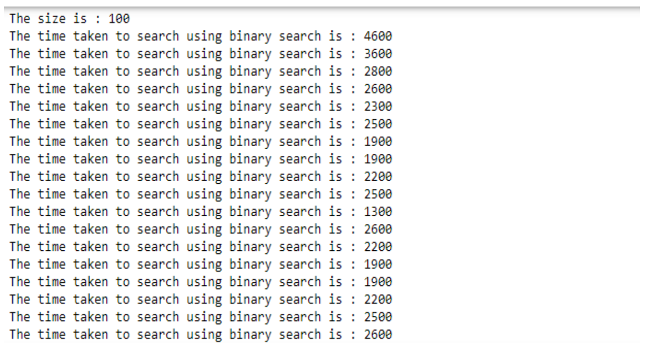
Plotting the graphs using Matplotlib:

Depicting the difference between Binary Search and Linear Search:

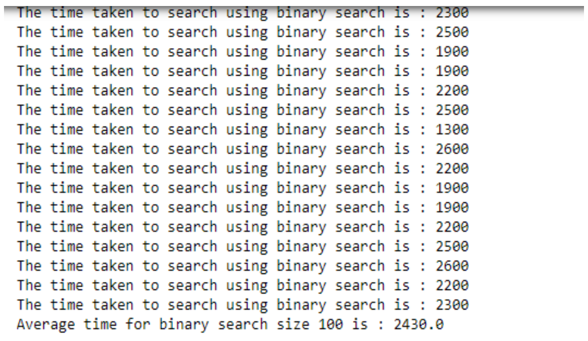
We can see that linear search is taking more time to search the element of different sizes from 100 to 200000.

The X-axis indicates the time in Nano Seconds.
Y-axis indicates the size of the data.
Using legends distinguished the colors between two search algorithms.





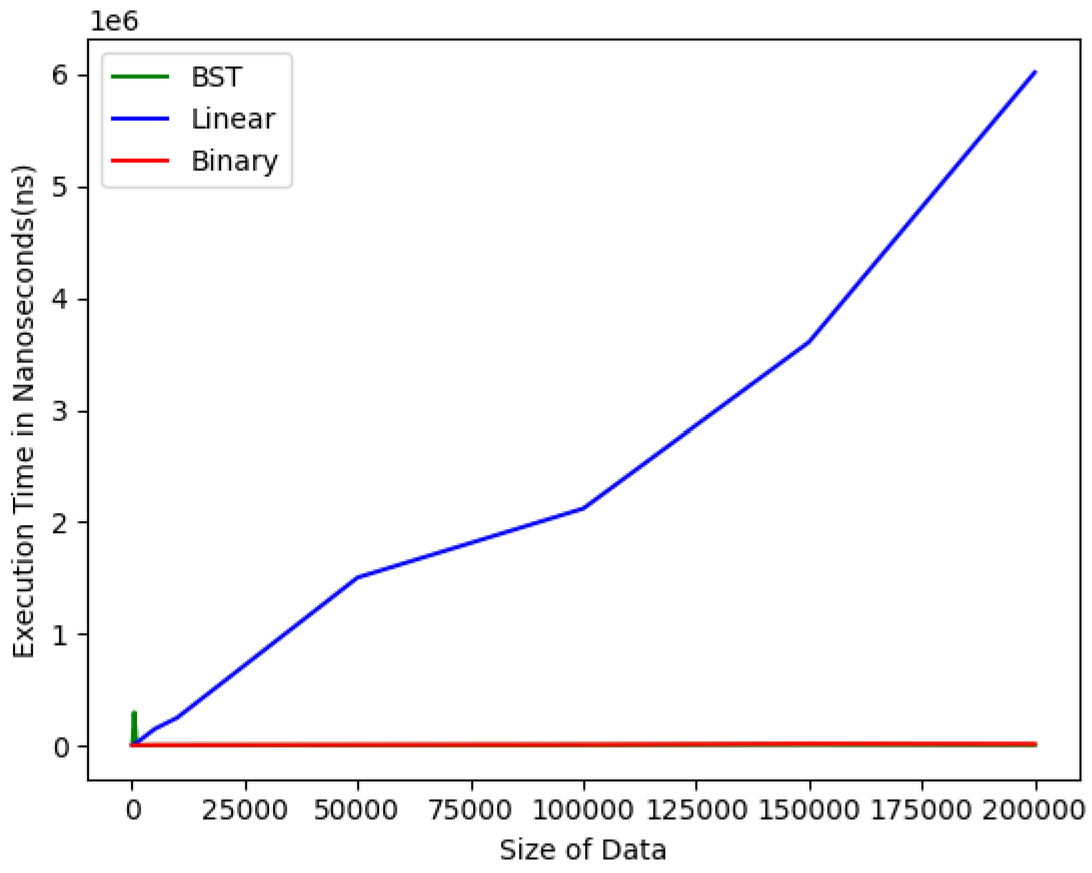
```
The size is : 100
The time taken to search using binary search is : 4600
The time taken to search using binary search is : 3600
The time taken to search using binary search is : 2800
The time taken to search using binary search is : 2600
The time taken to search using binary search is : 2300
The time taken to search using binary search is : 2500
The time taken to search using binary search is : 1900
The time taken to search using binary search is : 1900
The time taken to search using binary search is : 2200
The time taken to search using binary search is : 2500
The time taken to search using binary search is : 1300
The time taken to search using binary search is : 2600
The time taken to search using binary search is : 2200
The time taken to search using binary search is : 1900
The time taken to search using binary search is : 1900
The time taken to search using binary search is : 2200
The time taken to search using binary search is : 2500
The time taken to search using binary search is : 2600
```



```
The time taken to search using binary search is : 2300
The time taken to search using binary search is : 2500
The time taken to search using binary search is : 1900
The time taken to search using binary search is : 1900
The time taken to search using binary search is : 2200
The time taken to search using binary search is : 2500
The time taken to search using binary search is : 1300
The time taken to search using binary search is : 2600
The time taken to search using binary search is : 2200
The time taken to search using binary search is : 1900
The time taken to search using binary search is : 1900
The time taken to search using binary search is : 2200
The time taken to search using binary search is : 2500
The time taken to search using binary search is : 2600
The time taken to search using binary search is : 2200
The time taken to search using binary search is : 2300
Average time for binary search size 100 is : 2430.0
```

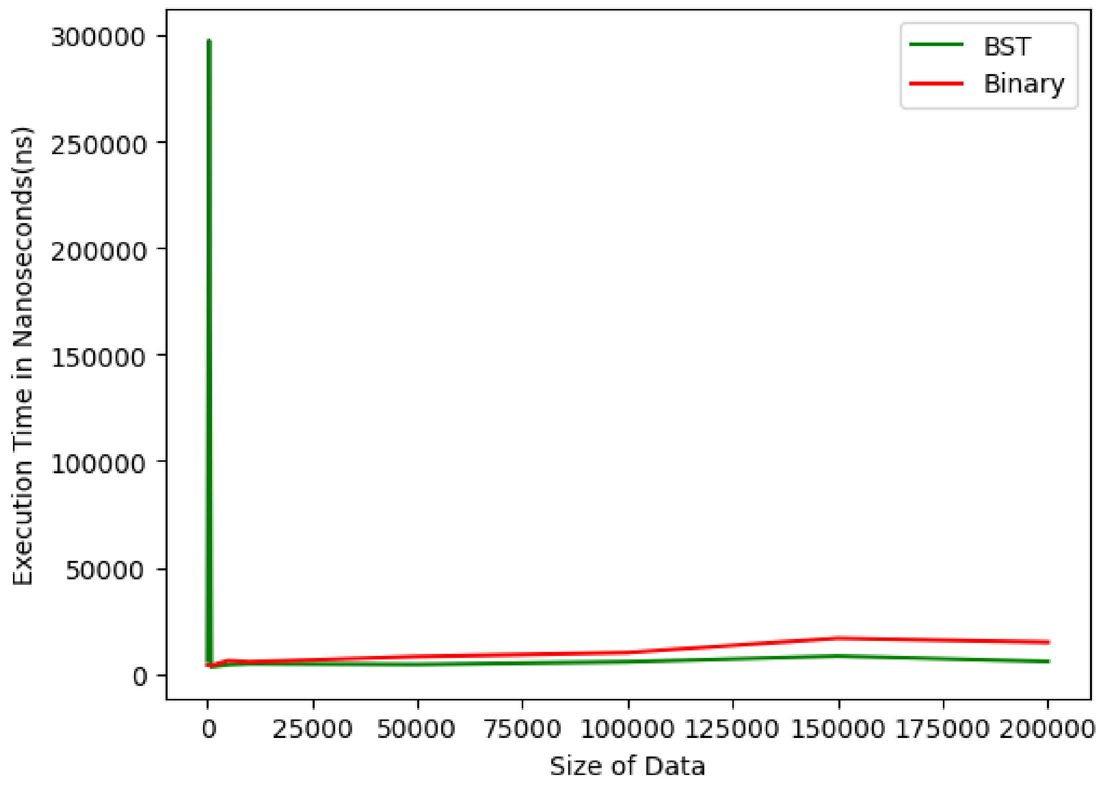
Depicting the difference between BST, Binary and Linear Search:

Clearly shows that BST and Binary search is outperforming linear search.

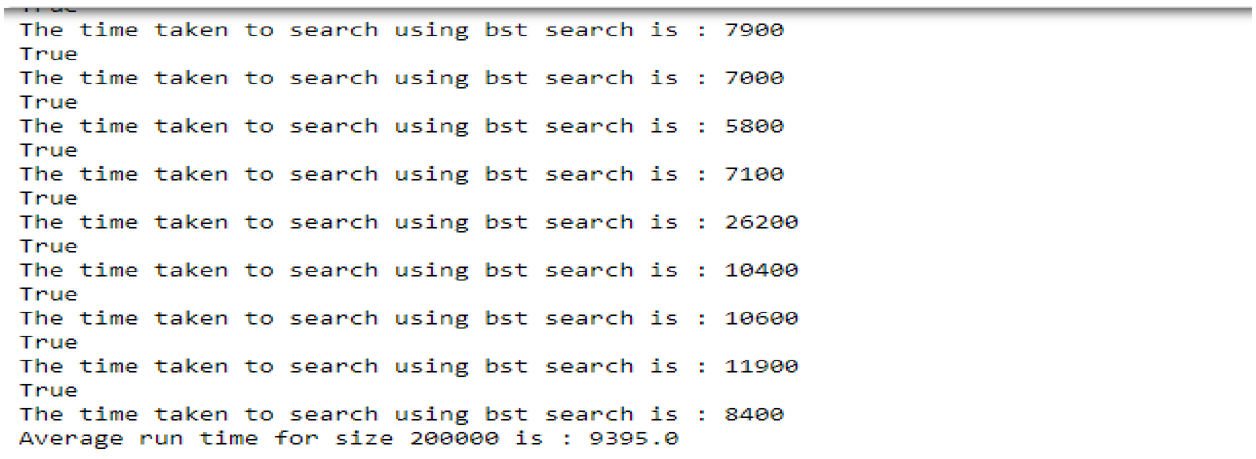


Depicting the difference between BST, and Binary Search:

Shows Binary search Tree is better than Binary Search.



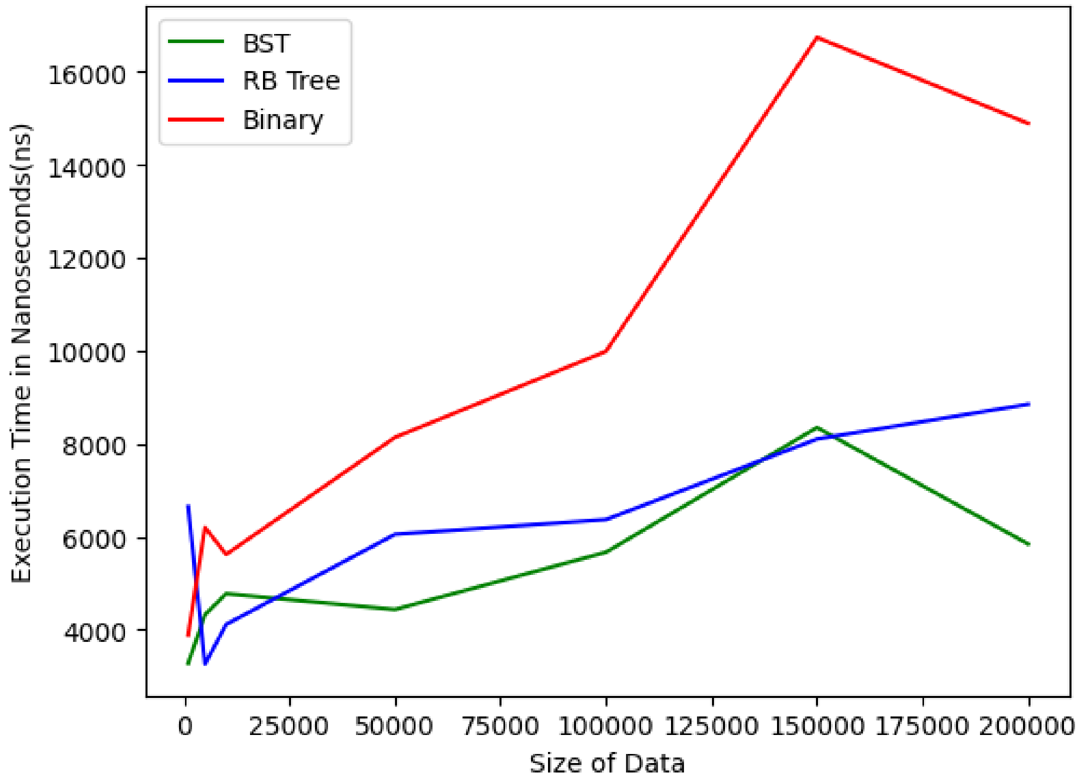
Calculating the BST average execution time for 20 different runs and considering the best possible average case.

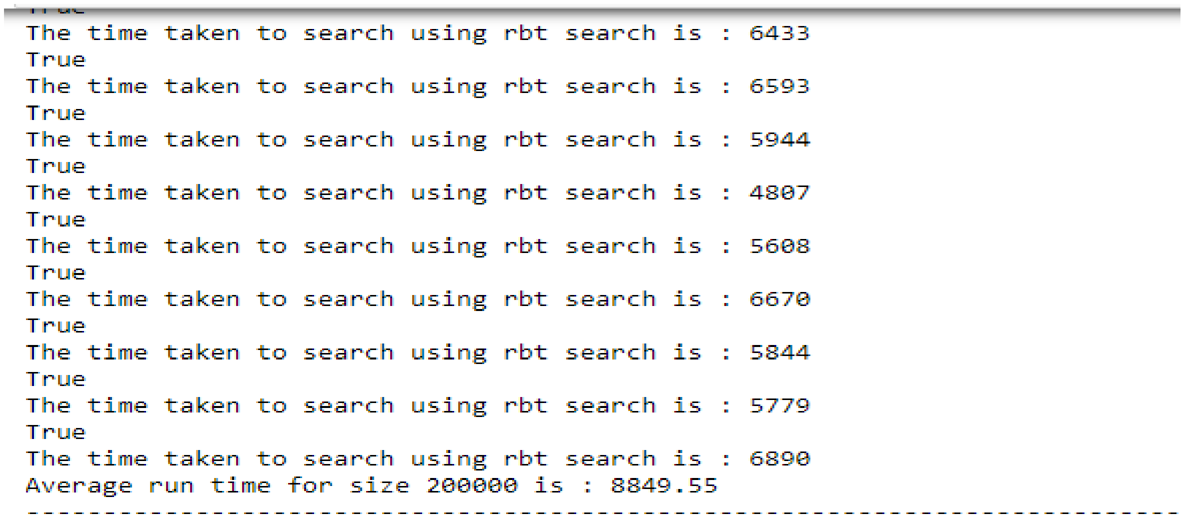


```
The time taken to search using bst search is : 7900
True
The time taken to search using bst search is : 7000
True
The time taken to search using bst search is : 5800
True
The time taken to search using bst search is : 7100
True
The time taken to search using bst search is : 26200
True
The time taken to search using bst search is : 10400
True
The time taken to search using bst search is : 10600
True
The time taken to search using bst search is : 11900
True
The time taken to search using bst search is : 8400
Average run time for size 200000 is : 9395.0
```

Depicting the difference between BST, Binary and Red Black Tree:

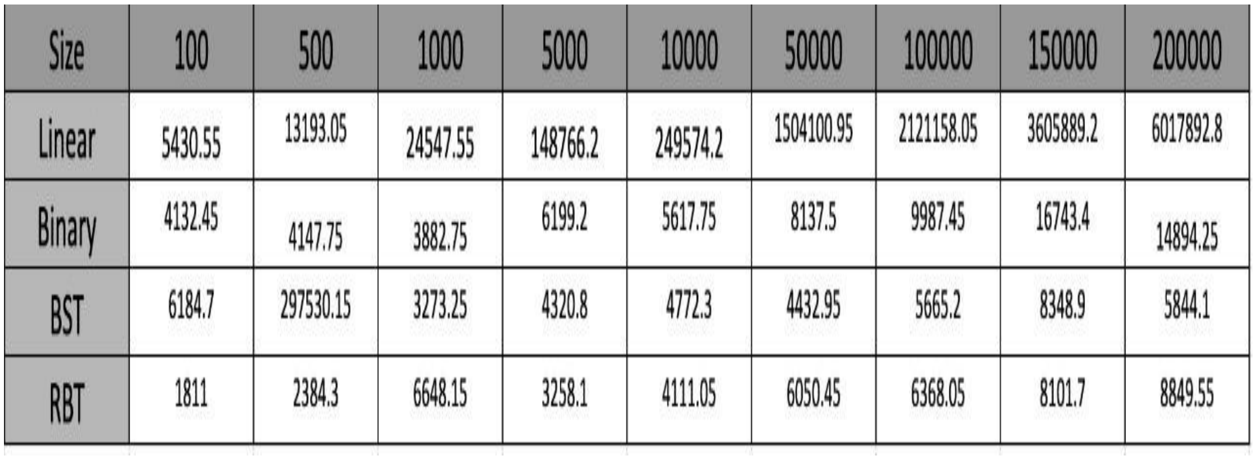
BST performs better than RBT in some scenarios, and it depends on the size of the data. It might varies from time to time.





```
The time taken to search using rbt search is : 6433
True
The time taken to search using rbt search is : 6593
True
The time taken to search using rbt search is : 5944
True
The time taken to search using rbt search is : 4807
True
The time taken to search using rbt search is : 5608
True
The time taken to search using rbt search is : 6670
True
The time taken to search using rbt search is : 5844
True
The time taken to search using rbt search is : 5779
True
The time taken to search using rbt search is : 6890
Average run time for size 200000 is : 8849.55
-----------------------------------------------------------------------------
```

Execution time for different search algorithms of different sizes:-



Size	100	500	1000	5000	10000	50000	100000	150000	200000
Linear	5430.55	13193.05	24547.55	148766.2	249574.2	1504100.95	2121158.05	3605889.2	6017892.8
Binary	4132.45	4147.75	3882.75	6199.2	5617.75	8137.5	9987.45	16743.4	14894.25
BST	6184.7	297530.15	3273.25	4320.8	4772.3	4432.95	5665.2	8348.9	5844.1
RBT	1811	2384.3	6648.15	3258.1	4111.05	6050.45	6368.05	8101.7	8849.55

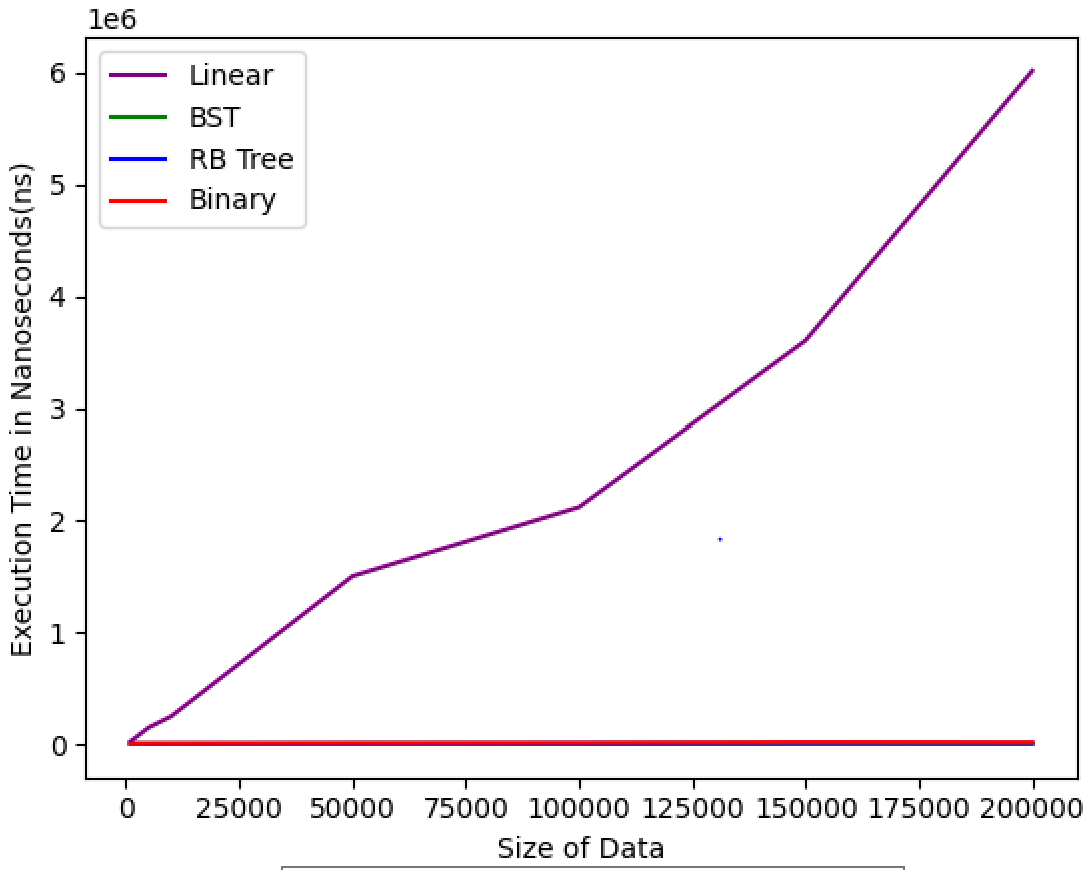
The following table depicts the execution time for different search algorithms, and we can see that Linear search is taking more time when compared to other algorithms, And BST is performing better than any other algorithm when we have a large dataset.

And BST and RBT perform similarly depending on the situation and dataset.

Binary search is outperforming when compared to linear search.

Depicting the difference between all 4 searching techniques:

In Overall we can see that Linear takes more time to search the element for different possible sizes. While BST is better in most of the scenarios compared to other algorithms.



Improvements:

A linear search has a time complexity of O(n), we can try a prune and search algorithm that it prunes and searches the array for the given element using the partition algorithm in quicksort. But it also yields a time complexity of O(n). For a sorted and uniformly distributed array as input the best algorithm we can use is interpolation search since it provides the best case time complexity O(log log(n)) and shorter time duration but its worst case time complexity goes till O(n). A Binary Search Tree in the worst-case scenario yields a linked list of n nodes where search complexity becomes O(n) this is the case where we implement balanced tree algorithms to maintain tree height on both sides like Red Black Tree, AVL Tree, 2-3-4
Tree, etc. AVL trees provide faster lookups than Red-Black Trees because they are more strictly balanced

Contributions:-

Ankith Reddy Avula
I implemented Linear search, Binary search and BST Algorithms and plotted the respective runtimes against data sizes.

Sravan Chandaka
I implemented Red Black Tree search algorithm, and used randomized testing for the algorithms

References:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.
2. https://www.geeksforgeeks.org/insertion-in-red-black-tree/
3. Slides for CSE 5311 course
4. https://www.programiz.com/dsa/red-black-tree
5. https://www.geeksforgeeks.org/g-fact-84/
6. https://en.wikipedia.org/wiki/Interpolation_search/
7. https://www.geeksforgeeks.org/red-black-tree-vs-avl-tree/#:~:text=AVL%20trees%20provide%20faster%20lookups,they%20are%20more%20strictly%20balanced.&text=In%20this%2C%20the%20color%20of,is%20either%20Red%20or%20Black.