Design and Analysis of Algorithm

Jilow John (1001775801)

- Linear Search
- Red Black Tree

Kshitij Khaladkar (1001713442)

- Binary Search
- Binary Search Tree

2. Implement and compare the following search algorithm:

- **Linear search**
- **Binary search in sorted array.**
- **Binary search tree**
- **Red-Black Tree**

Linear search

A linear search or sequential search is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.

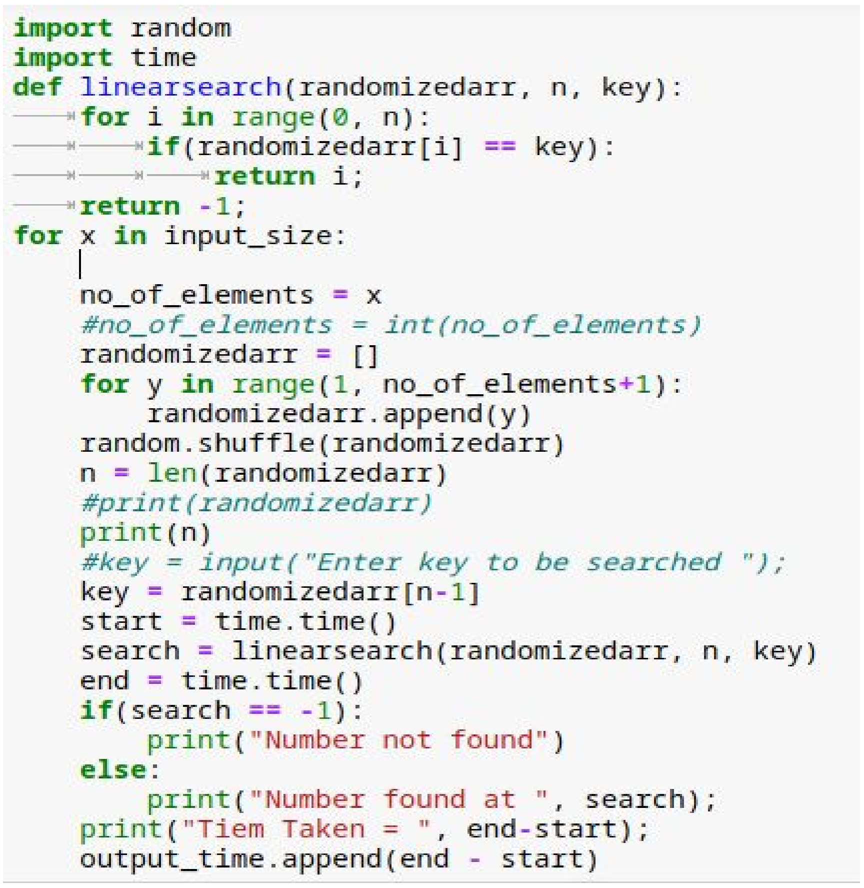
Time Complexity: O(n)

Space Complexity: O(1)

Data structures used : list (for input)

How to improve linear search : We could use 2 pointers to traverse through the list.One would start from the beginning and the other would start from the end.

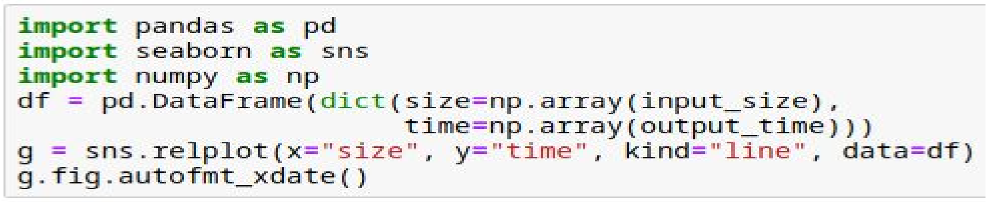
Code



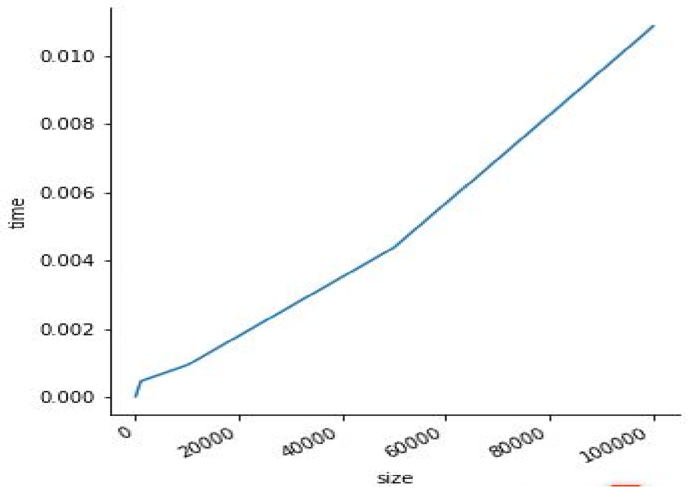
```python
import random
import time
def linearsearch(randomizedarr, n, key):
    for i in range(0, n):
        if(randomizedarr[i] == key):
            return i;
    return -1;
for x in input_size:

    no_of_elements = x
    #no_of_elements = int(no_of_elements)
    randomizedarr = []
    for y in range(1, no_of_elements+1):
        randomizedarr.append(y)
    random.shuffle(randomizedarr)
    n = len(randomizedarr)
    #print(randomizedarr)
    print(n)
    #key = input("Enter key to be searched ");
    key = randomizedarr[n-1]
    start = time.time()
    search = linearsearch(randomizedarr, n, key)
    end = time.time()
    if(search == -1):
        print("Number not found")
    else:
        print("Number found at ", search);
    print("Tiem Taken = ", end-start);
    output_time.append(end - start)
```

Main Component: In the above code the main component is linearsearch() function, where the actual search takes place. It takes an array and the key to be searched as the inputs and finds the element in the unsorted array. Also the time is calculated using the time.time() function for the linear search function.

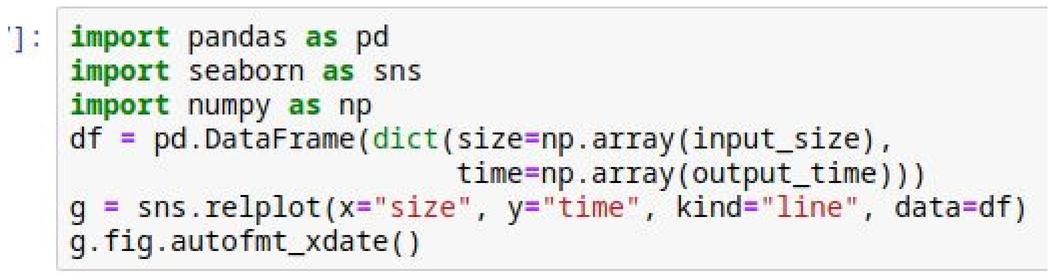
Output Runtime Graphs : (Key found)



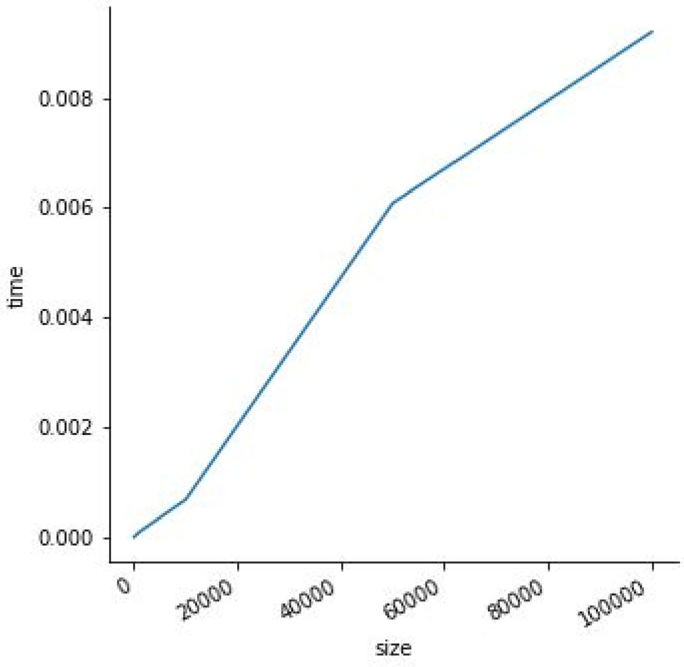
```
import pandas as pd
import seaborn as sns
import numpy as np
df = pd.DataFrame(dict(size=np.array(input_size),
                       time=np.array(output_time)))
g = sns.relplot(x="size", y="time", kind="line", data=df)
g.fig.autofmt_xdate()
```



Output Runtime Graphs : (Key not found)



```
import pandas as pd
import seaborn as sns
import numpy as np
df = pd.DataFrame(dict(size=np.array(input_size),
                       time=np.array(output_time)))
g = sns.relplot(x="size", y="time", kind="line", data=df)
g.fig.autofmt_xdate()
```



This algorithm has a linear time throughout all sizes of data input.Thus this algorithm would be preferred for a small to a moderate sized dataset.

Binary search in sorted array

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 on 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.

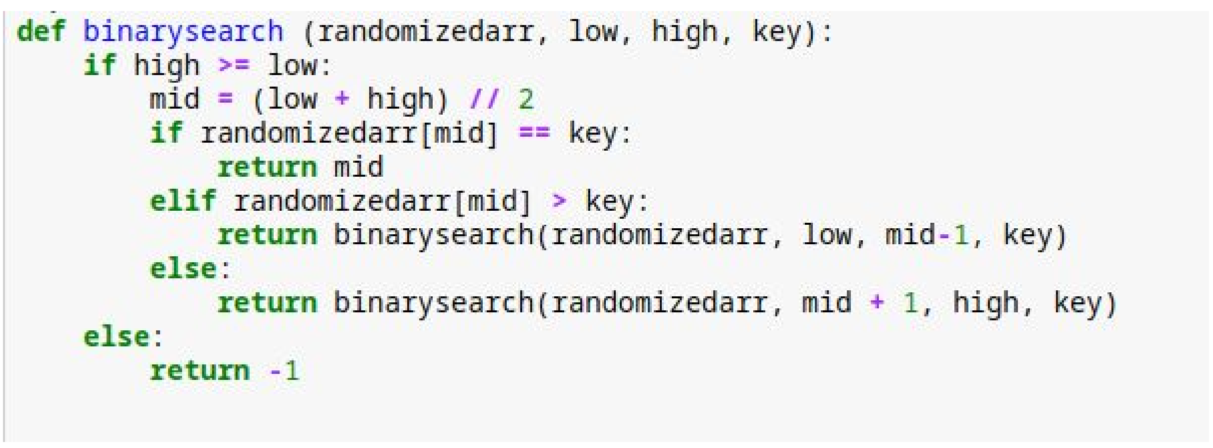
Time Complexity: O(log n)

Space Complexity: O(1)

Data Structure Used : Sorted List (for input)

How to improve Binary Search : We could improve binary search by dividing the array in 3 parts thereby changing the runtime to O($\log_3$n).

Code

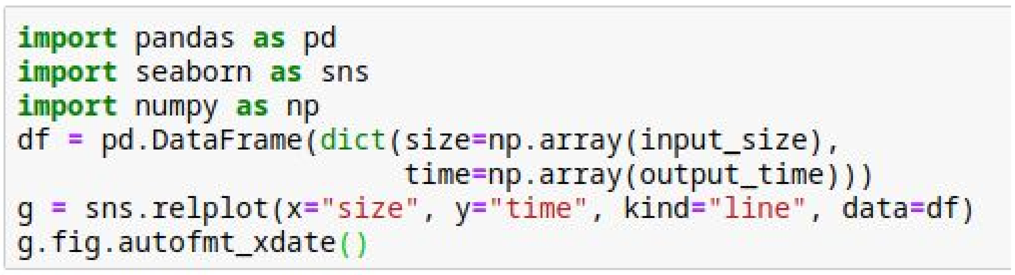


```python
def binarysearch (randomizedarr, low, high, key):
    if high >= low:
        mid = (low + high) // 2
        if randomizedarr[mid] == key:
            return mid
        elif randomizedarr[mid] > key:
            return binarysearch(randomizedarr, low, mid-1, key)
        else:
            return binarysearch(randomizedarr, mid + 1, high, key)
    else:
        return -1
```

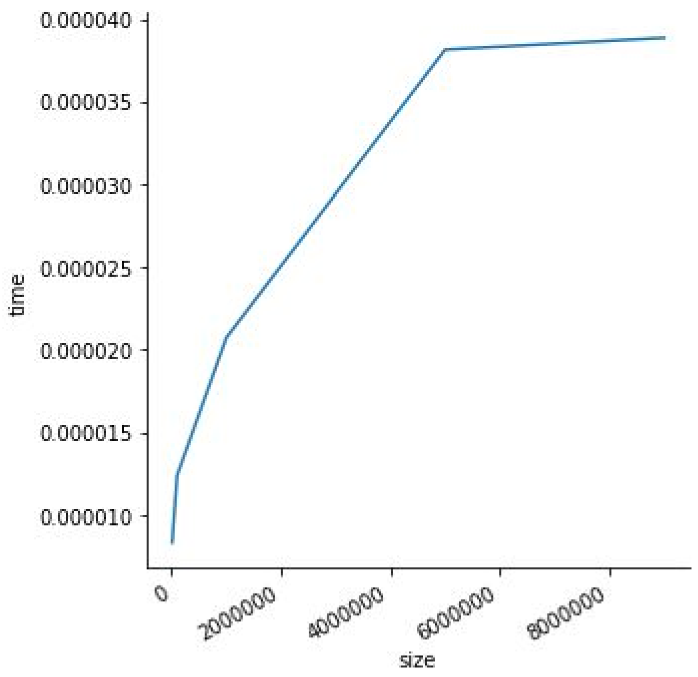
Main Component: The main component in the Binary Search algorithm is binarysearch() function in which we pass an array, key to be searched, low value and the high value from the array. We calculate the mid, using mid we compare it with the key to be found and check whether the key is in the left half of the array or the

right half of the array. Doing this recursively we end up getting the mid and key value as same. In this way the search goes through.
In the above code we generate a random array and sort the array using the sort() function and calculate the time of the search using the time.time() function .

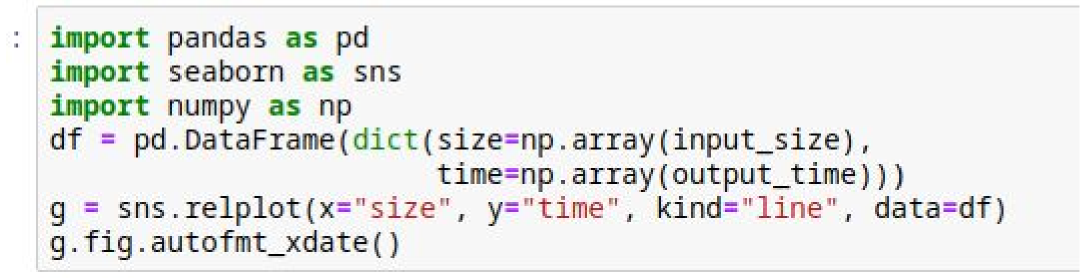
<u>Output runtime Graph (Key Found)</u>



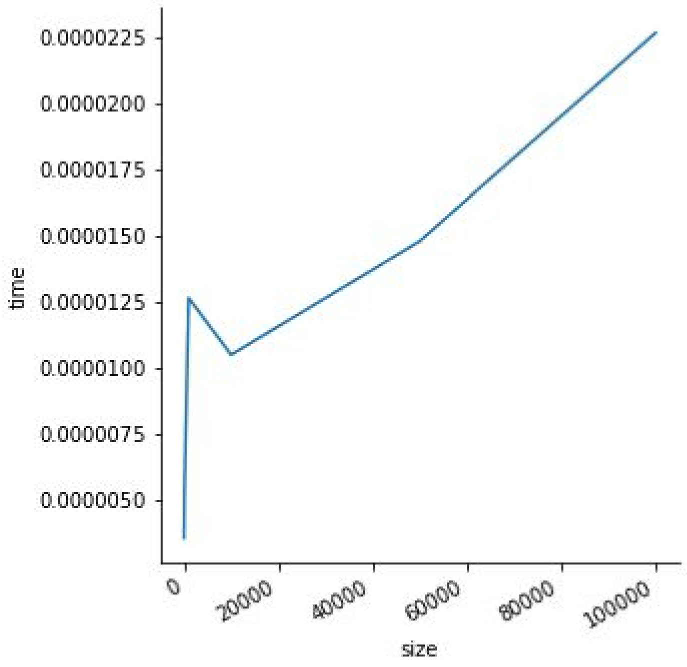
```
import pandas as pd
import seaborn as sns
import numpy as np
df = pd.DataFrame(dict(size=np.array(input_size),
                       time=np.array(output_time)))
g = sns.relplot(x="size", y="time", kind="line", data=df)
g.fig.autofmt_xdate()
```



Output runtime Graph (Key Not Found)



```
: import pandas as pd
  import seaborn as sns
  import numpy as np
  df = pd.DataFrame(dict(size=np.array(input_size),
                         time=np.array(output_time)))
  g = sns.relplot(x="size", y="time", kind="line", data=df)
  g.fig.autofmt_xdate()
```



According to the plotted graph, it can be observed that this algorithm works much better for a large dataset compared to a smaller dataset. Hence it should be used in scenarios where large data is present.

Binary search tree

Binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of container: a data structure that stores "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name).

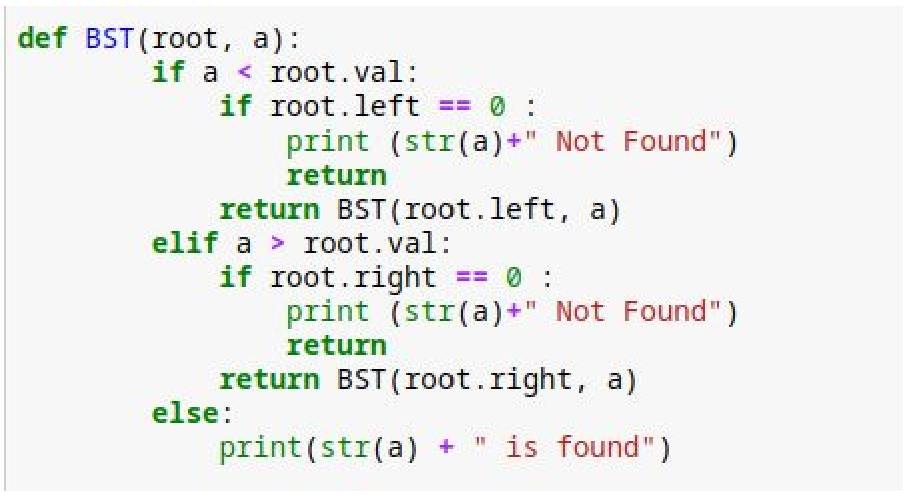
Time Complexity: O(log n) (if tree balanced)
O(n) (if tree unbalanced)

Space Complexity: O(n)

Data Structures used : -Tree (to store values)
-List (to store input)

How to improve the algorithm : Rather than having 2 child nodes for each parent node we could add 3 child nodes per parent thereby decreasing the tree height and helping in easy traversal through the tree.

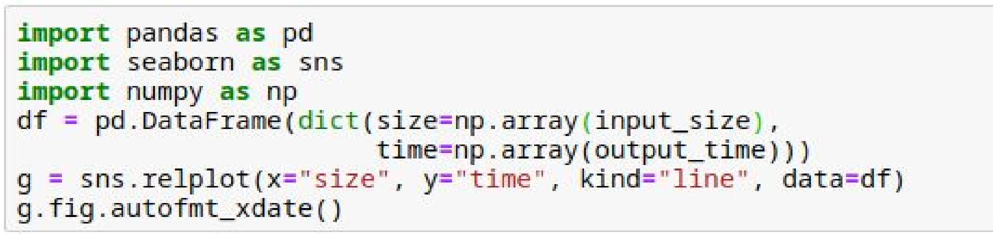
Code



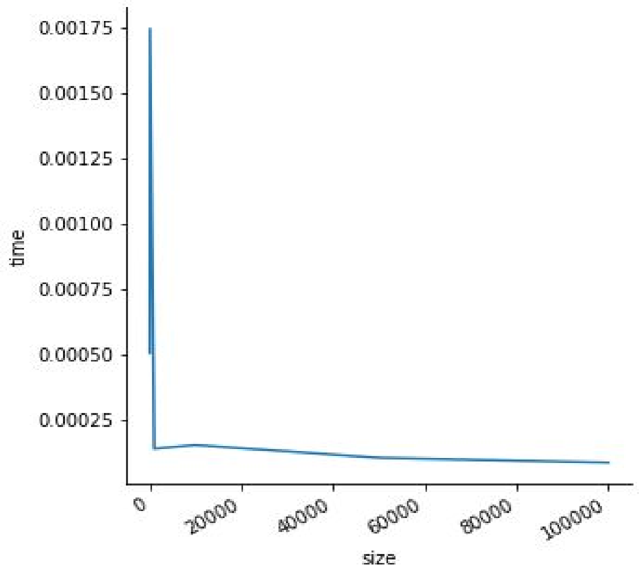
```
def BST(root, a):
        if a < root.val:
            if root.left == 0 :
                print (str(a)+" Not Found")
                return
            return BST(root.left, a)
        elif a > root.val:
            if root.right == 0 :
                print (str(a)+" Not Found")
                return
            return BST(root.right, a)
        else:
            print(str(a) + " is found")
```

Main Component: The main components in the binary search tree algorithm are insert() and BST(). In the insert() function we pass the value of the root and the nodes to be inserted. The tree inserts the values by comparing the node values with the root. If the node value is less than the root value we insert the value to the left half of the tree and if the the node value is greater than the node we insert the value to the right half of the tree.
In the BST() function we pass the value of root and the key to be searched. Here we compare the key value with the root node at first and if it's greater than the root we search the value in the right half of the tree else we search in the left half of the tree. Doing this recursively we find whether the key is present in the tree.

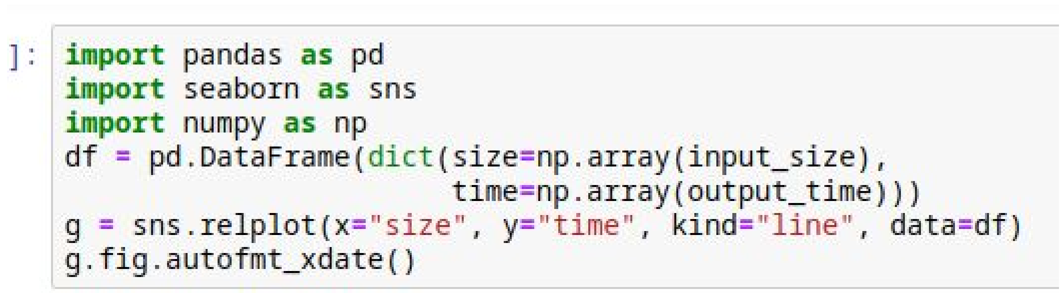
Output Runtime Graph (Key Found)



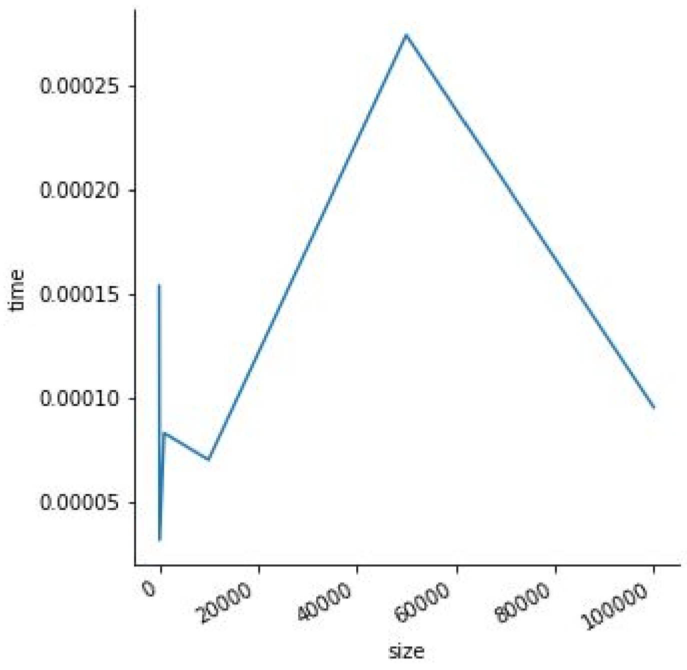
```
import pandas as pd
import seaborn as sns
import numpy as np
df = pd.DataFrame(dict(size=np.array(input_size),
                       time=np.array(output_time)))
g = sns.relplot(x="size", y="time", kind="line", data=df)
g.fig.autofmt_xdate()
```



Output Runtime Graph (Key Not Found)



```
]: import pandas as pd
   import seaborn as sns
   import numpy as np
   df = pd.DataFrame(dict(size=np.array(input_size),
                          time=np.array(output_time)))
   g = sns.relplot(x="size", y="time", kind="line", data=df)
   g.fig.autofmt_xdate()
```



From the graph, it could be observed that it works terribly for small datasets but could work incredibly well for moderate and large datasets

<u>Red-Black Tree :</u>

A red–black tree is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

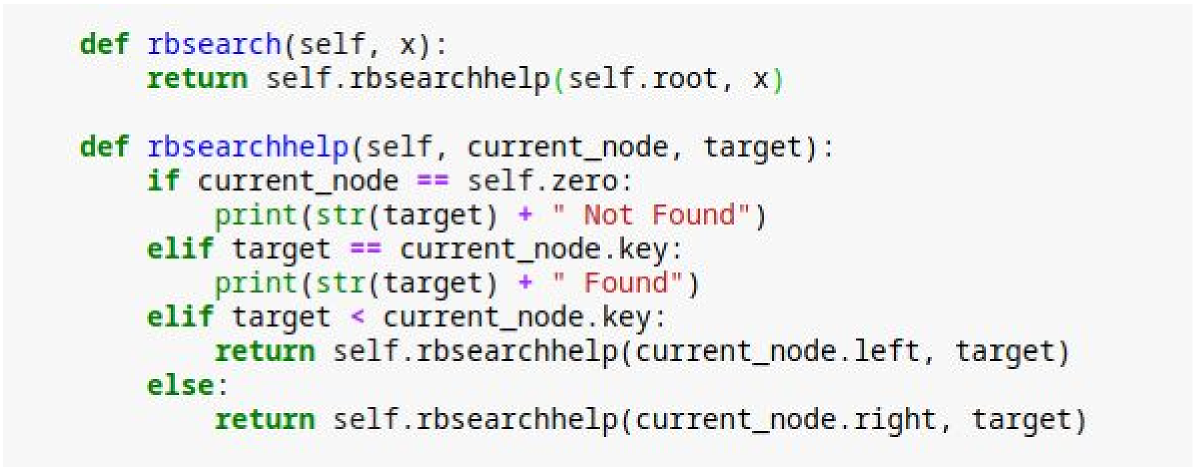
<u>Time Complexity:</u> O(log n)

<u>Space Complexity:</u> O(n)

<u>Data Structure</u> : -List (for input)
                              -Trees

<u>How to improve the algorithm</u> : Could not think any improvements for Red - Black Tree

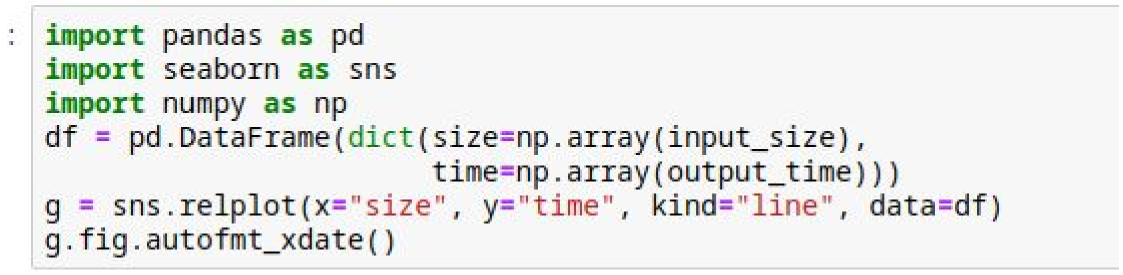
<u>Code :</u>



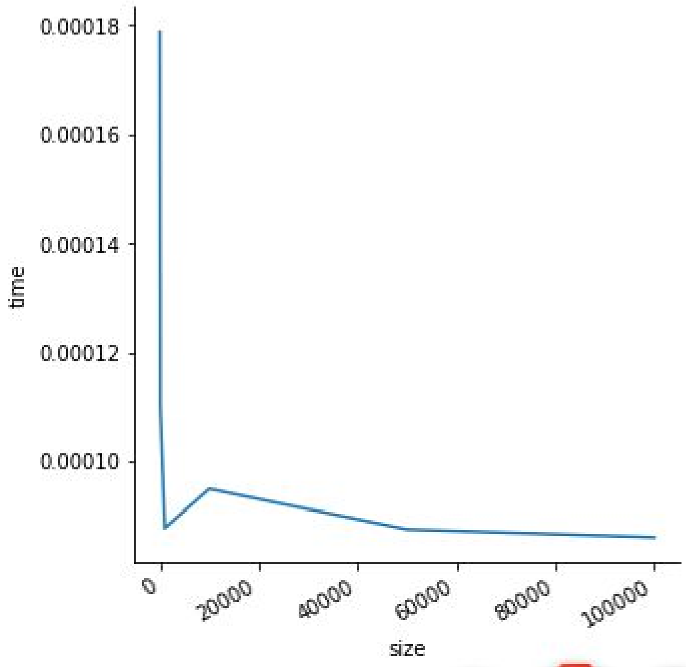
```
def rbsearch(self, x):
    return self.rbsearchhelp(self.root, x)

def rbsearchhelp(self, current_node, target):
    if current_node == self.zero:
        print(str(target) + " Not Found")
    elif target == current_node.key:
        print(str(target) + " Found")
    elif target < current_node.key:
        return self.rbsearchhelp(current_node.left, target)
    else:
        return self.rbsearchhelp(current_node.right, target)
```

Main Component: The main components in the red black tree algorithm are RR(), LL(), maketree(), balance() and rbsearch(). The RR() and LL() functions are used in order to balance the tree by doing Left Rotation and Right Rotation.
maketree() function is used in order to create the red black tree by adding the nodes to the tree.
balance() function checks whether the tree is balanced or not i.e whether the number of black nodes on both the sides of the tree are equal in height. If the tree is not balanced then the appropriate function is called like Left Rotation or Right Rotation.
rbsearch() function gets the key from the user and searches whether the particular key is in the red black tree or not. For search we use the binary search method and find the required key.

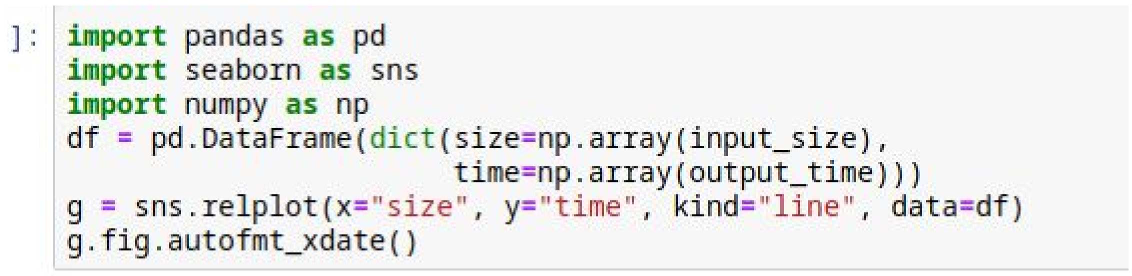
Output Runtime Graph (Key Found)



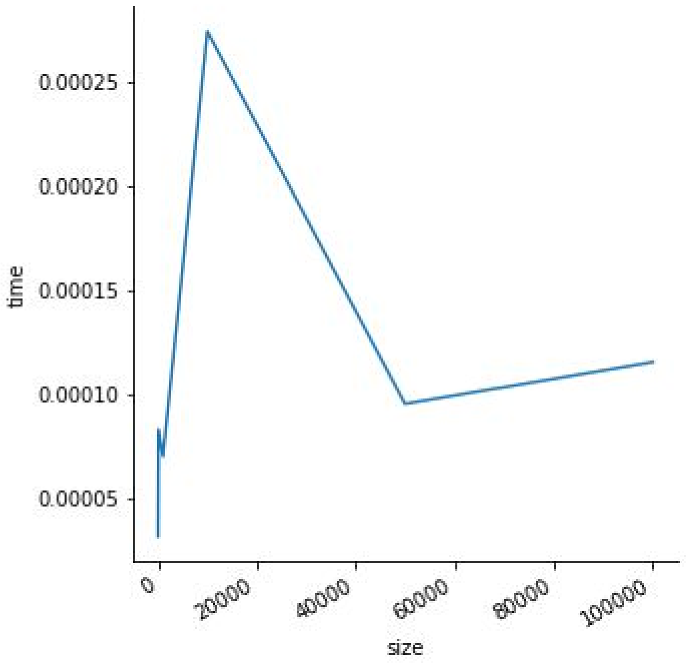
```
import pandas as pd
import seaborn as sns
import numpy as np
df = pd.DataFrame(dict(size=np.array(input_size),
                       time=np.array(output_time)))
g = sns.relplot(x="size", y="time", kind="line", data=df)
g.fig.autofmt_xdate()
```



Output Runtime Graph (Key Not Found)



```
]: import pandas as pd
   import seaborn as sns
   import numpy as np
   df = pd.DataFrame(dict(size=np.array(input_size),
                          time=np.array(output_time)))
   g = sns.relplot(x="size", y="time", kind="line", data=df)
   g.fig.autofmt_xdate()
```



Red Black Tree works exceptionally fine for large input size whereas it is extremely slow for smaller input sizes.