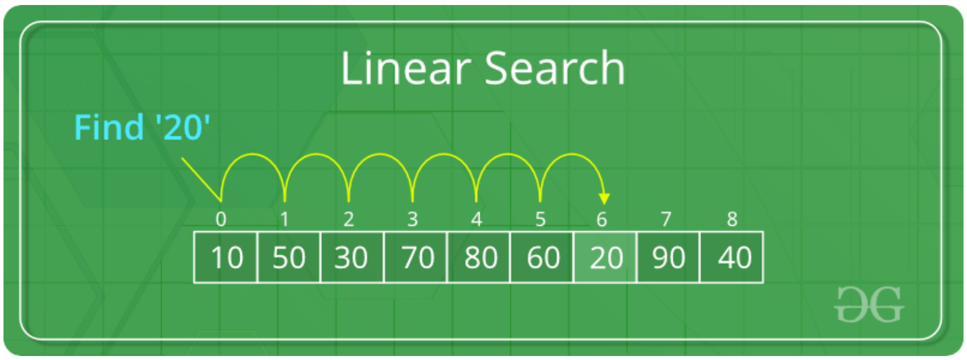
Shivani Panchiwala
UTA ID: 1001982478

Project Report on Search Algorithms

Linear Search Algorithm:

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.



Follow the below idea to solve the problem:

Iterate from 0 to N-1 and compare the value of every index with value if they match return index

Follow the given steps to solve the problem:

- Start from the leftmost element of arr[] and one by one compare value with each element of arr[]
- If value matches with an element, return the index.
- If value doesn't match with any of the elements, return -1.

Pseudocode of Linear Search Algorithm

Start
linear_search (Array , value)
For each element in the array
If (searched element == value)
Return's the searched lament location
end if
end for
end

Below is the implementation of the above approach:

```
import time
def linear_search(arr, value):
  t1_start = time.perf_counter()
  for i in range(len(arr)):

    if (arr[i] == value):
      t1_stop = time.perf_counter()
      #print("--- %s seconds ---" % str(t1_stop - t1_start))
      e1 = str(t1_stop - t1_start) + "s"  # Get running time
      #return True
      return (i, e1)
  t1_stop = time.perf_counter()
  e1 = str(t1_stop - t1_start) + "s"
  #print("--- %s seconds ---" % str(t1_stop - t1_start))
  #return False
  return (-1, e1)
```

Complexity

Best Case Complexity

- The element being searched could be found in the first position.
- In this case, the search ends with a single successful comparison.
- Thus, in the best-case scenario, the linear search algorithm performs O(1) operations.

Worst Case Complexity

- The element being searched may be at the last position in the array or not at all.
- In the first case, the search succeeds in ‘n’ comparisons.
- In the next case, the search fails after ‘n’ comparisons.
- Thus, in the worst-case scenario, the linear search algorithm performs O(n) operations.

Average Case Complexity

When the element to be searched is in the middle of the array, the average case of the Linear Search Algorithm is O(n).

Space Complexity

The linear search algorithm takes up no extra space; its space complexity is O(n) for an array of n elements.

Data Structure

It is used to search for any element in a linear data structure like arrays and linked lists.

References

https://www.geeksforgeeks.org/python-program-for-linear-search/

https://www.geeksforgeeks.org/linear-search/

https://www.simplilearn.com/tutorials/data-structure-tutorial/linear-search-algorithm

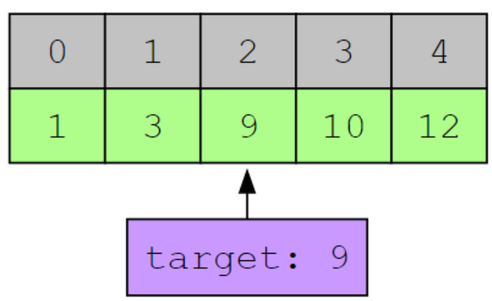
Binary Search Algorithm on Sorted Array:

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you have narrowed down the possible locations to just one.

Given an array of integers, nums, sorted in ascending order, and an integer value, target. If the target exists in the array, return its index. If the target does not exist, then return -1.

The binary search divides the input array by half at every step. After every step, either we find the index we are looking for, or we discard half of the array.

Given the following sorted array, if the target's value is 9, the binary search returns 2.



In the approach, here is how the algorithm works:

- At each step, consider the array between low and high indices.
- Calculate the mid index.
- If the element at the mid index is equal to the target value, we return mid.
- If the element at mid is greater than the target:
    - Change the index high to mid - 1.
    - The value of low remains the same.
- If the element at mid is less than the target:
    - Change low to mid + 1.
    - The value of high remains the same.

- When the value of low is greater than the value of high, this indicates that the target does not exist. Hence, -1 is returned.

```
import time

def binary_search_rec(nums, target, low, high):
  if low > high:
    return -1

  # Finding the mid using floor division
  mid = low + ((high - low) // 2)

  # Target value is present at the middle of the array
  if nums[mid] == target:
    return mid

  # Target value is present in the low subarray
  elif target < nums[mid]:
    return binary_search_rec(nums, target, low, mid - 1)

  # Target value is present in the high subarray
  else:
    return binary_search_rec(nums, target, mid + 1, high)


def binary_search(nums, target):
  arr_sort = sorted(nums)  # Sorting array
  t2_start = time.perf_counter()
  result = binary_search_rec(arr_sort, target, 0, len(nums) - 1)
  t2_stop = time.perf_counter()
  #print("--- %s seconds ---" % str(t2_stop - t2_start))
  t2 = str(t2_stop-t2_start)+"s"
  if result != -1:
    return (result, arr_sort, t2)
  else:
    return (-1, arr_sort, t2)
```

Complexity

Best Case Time Complexity of Binary Search
The best case of Binary Search occurs when:

- The element to be search is in the middle of the list

In this case, the element is found in the first step itself and this involves 1 comparison.

Therefore, Best Case Time Complexity of Binary Search is O(1).

Average Case Time Complexity of Binary Search
Let there be N distinct numbers: a1, a2, ..., a(N-1), aN

We need to find element P.

There are two cases:

Case 1: The element P can be in N distinct indexes from 0 to N-1.
Case 2: There will be a case when the element P is not present in the list.
There are N case 1 and 1 case 2. So, there are N+1 distinct cases to consider in total.

If element P is in index K, then Binary Search will do K+1 comparisons.

This is because:

The element at index N/2 can be found in 1 comparison as Binary Search starts from middle.

Similarly, in the 2nd comparisons, elements at index N/4 and 3N/4 are compared based on the result of 1st comparison.

On this line, in the 3rd comparison, elements at index N/8, 3N/8, 5N/8, 7N/8 are compared based on the result of 2nd comparison.

Based on this, we know that:

- Elements requiring 1 comparison: 1
- Elements requiring 2 comparisons: 2
- Elements requiring 3 comparisons: 4

Therefore, Elements requiring I comparisons: 2^(I-1)

The maximum number of comparisons = Number of times N is divided by 2 so that result is 1 = Comparisons to reach 1st element = logN comparisons

I can vary from 0 to logN

Total number of comparisons = 1 * (Elements requiring 1 comparison) + 2 * (Elements requiring 2 comparisons) + ... + logN * (Elements requiring logN comparisons)

Total number of comparisons = 1 * (1) + 2 * (2) + 3 * (4) + ... + logN * (2^(logN-1))

Total number of comparisons = 1 + 4 + 12 + 32 + ... = 2^logN * (logN - 1) + 1

Total number of comparisons = N * (logN - 1) + 1

Total number of cases = N+1

Therefore, average number of comparisons = (N * (logN - 1) + 1) / (N+1)

Average number of comparisons = N * logN / (N+1) - N/(N+1) + 1/(N+1)

The dominant term is N * logN / (N+1) which is approximately logN. Therefore, Average Case Time Complexity of Binary Search is O(logN).

Analysis of Worst Case Time Complexity of Binary Search
The worst case of Binary Search occurs when:

- The element is to search is in the first index or last index

In this case, the total number of comparisons required is logN comparisons.

Therefore, Worst Case Time Complexity of Binary Search is O(logN).

Analysis of Space Complexity of Binary Search
In an iterative implementation of Binary Search, the space complexity will be O(1).

This is because we need two variables to keep track of the elements to be checked. No other data is needed.

In a recursive implementation of Binary Search, the space complexity will be O(logN).

Data Structure:

The Heap data structure to implement Prim's Algorithm.

References:

https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search

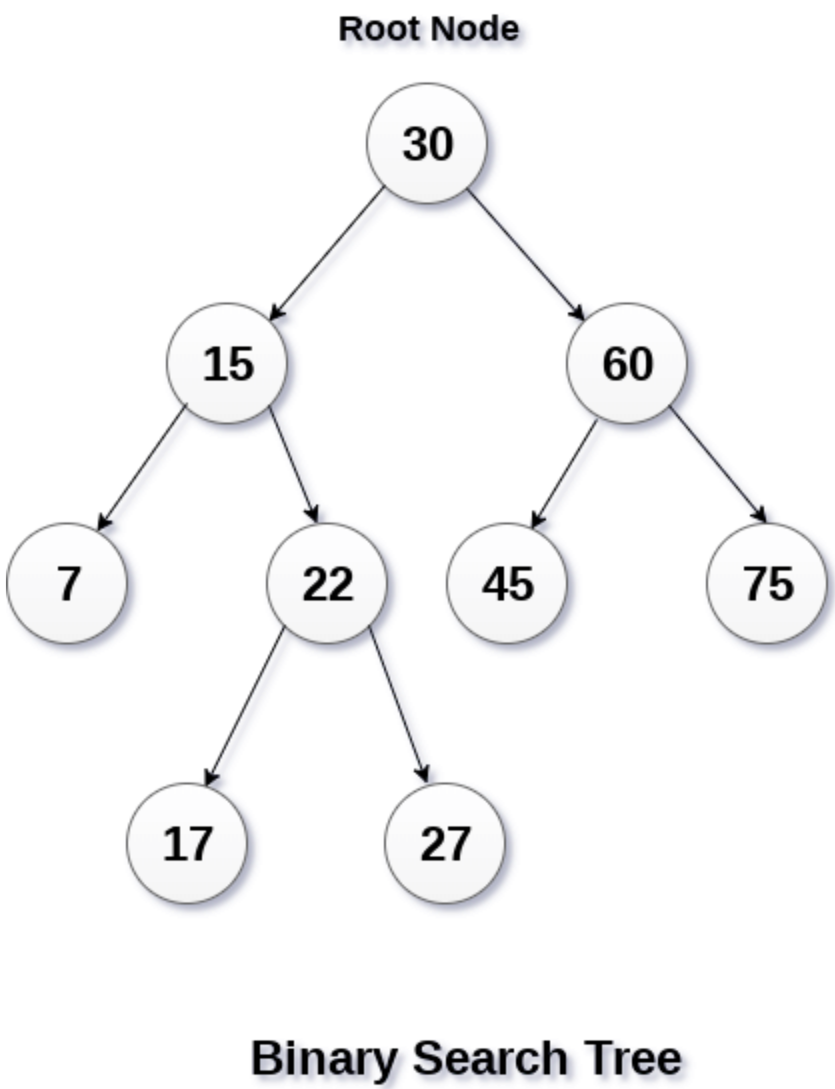
https://iq.opengenus.org/time-complexity-of-binary-search/

Binary Search Tree:

A binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers. It is composed of nodes, which store data and link to up to two other child nodes. It is the relationship between the leaves linked to and the linking leaf, also known as the parent node, which makes the binary tree such an efficient data structure.

For a binary tree to be a binary search tree, the data of all the nodes in the left sub-tree of the root node should be less than the data of the root. The data of all the nodes in the right subtree of the root node should be greater than equal to the data of the root. As a result, the leaves on the farthest left of the tree have the lowest values, whereas the leaves on the right of the tree have the greatest values.

(i)It is called a binary tree because each tree node has a maximum of two children.
(ii)It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.



<u>Binary Search Tree Algorithm works:</u>

- Compare the element with the root of the tree.

- If the item is matched then return the location of the node.
- Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
- If not, then move to the right sub-tree.
- Repeat this procedure recursively until match found.
- If element is not found then return NULL.

Implementation:

```
import time
class Binary_Search_Tree:
    """
    Constructor with value we are going
    to insert in tree with assigning
    left and right child with default None
    """

    def __init__(self, data):
        self.data = data
        self.Left_child = None
        self.Right_child = None

    """
    If the data we are inserting already
   present in tree it will not add it
   to avoid the duplicate values
    """

    def Add_Node(self, data):
        if data == self.data:
            return  # node already exist

        """
        If the data we are inserting is Less
        than the value of the current node, then
        data will insert in Left node
        """

        if data < self.data:
            if self.Left_child:
                self.Left_child.Add_Node(data)
            else:
```

```
                self.Left_child = Binary_Search_Tree(data)

            """
            If the data we are inserting is Greater
            than the value of the current node, then
            data will insert in Right node
            """

        else:
            if self.Right_child:
                self.Right_child.Add_Node(data)
            else:
                self.Right_child = Binary_Search_Tree(data)

    def Find_Node(self, val):

        """
        If current node is equal to
        data we are finding return true
        """

        if self.data == val:
            return True

        """
        If current node is lesser than
       data we are finding we have search
       in Left child node
        """

        if val < self.data:
            if self.Left_child:
                return self.Left_child.Find_Node(val)
            else:
                return False

        """
        If current node is Greater than
       data we are finding we have search
       in Right child node
        """

        if val > self.data:
```

```
            if self.Right_child:
                return self.Right_child.Find_Node(val)
            else:
                return False

    """
    First it will visit Left node then
    it will visit Root node and finally
   it will visit Right and display a
    list in specific order
    """

    def In_Order_Traversal(self):
        elements = []
        if self.Left_child:
            elements += self.Left_child.In_Order_Traversal()

        elements.append(self.data)

        if self.Right_child:
            elements += self.Right_child.In_Order_Traversal()

        return elements

    """
    First it will visit Left node then
    it will visit Right node and finally
   it will visit Root node  and display a
    list in specific order
    """

    def Post_Order_Traversal(self):
        elements = []
        if self.Left_child:
            elements += self.Left_child.Post_Order_Traversal()
        if self.Right_child:
            elements += self.Right_child.Post_Order_Traversal()

        elements.append(self.data)

        return elements

    """
```

```
    First it will visit Root node then
    it will visit Left node and finally
    it will visit Right node  and display a
    list in specific order
    """

    def Pre_Order_Traversal(self):
        elements = [self.data]
        if self.Left_child:
            elements += self.Left_child.Pre_Order_Traversal()
        if self.Right_child:
            elements += self.Right_child.Pre_Order_Traversal()

        return elements

    """
    This method will give
    the Max value of tree
    """

    def Find_Maximum_Node(self):
        if self.Right_child is None:
            return self.data
        return self.Right_child.Find_Maximum_Node()

    """
    This method will give
    the Min value of tree
    """

    def Find_Minimum_Node(self):
        if self.Left_child is None:
            return self.data
        return self.Left_child.Find_Minimum_Node()

    """
    This method will give
    the Total Sum value of tree
    """

    def calculate_Sum_Of_Nodes(self):
        left_sum = self.Left_child.calculate_Sum_Of_Nodes() if self.Left_child else 0
        right_sum = self.Right_child.calculate_Sum_Of_Nodes() if self.Right_child else 0
```

```
            return self.data + left_sum + right_sum


    """
    This method helps to build the tree
    whith the element we inserted in it
    """


    def Build_Tree(elements,x):
        root = Binary_Search_Tree(elements[0])

        for i in range(1, len(elements)):
            root.Add_Node(elements[i])
        s3 = time.perf_counter()
        result = root.Find_Node(x)
        e3 = time.perf_counter()  # end counter
        t3 = str(e3 - s3) + "s"
        print("element found in array : ", result)
        if result == None:  #element is not found in the binary search tree
            return (-1, t3)
        else:
            return (1, t3)
        #return root
```

Traversals

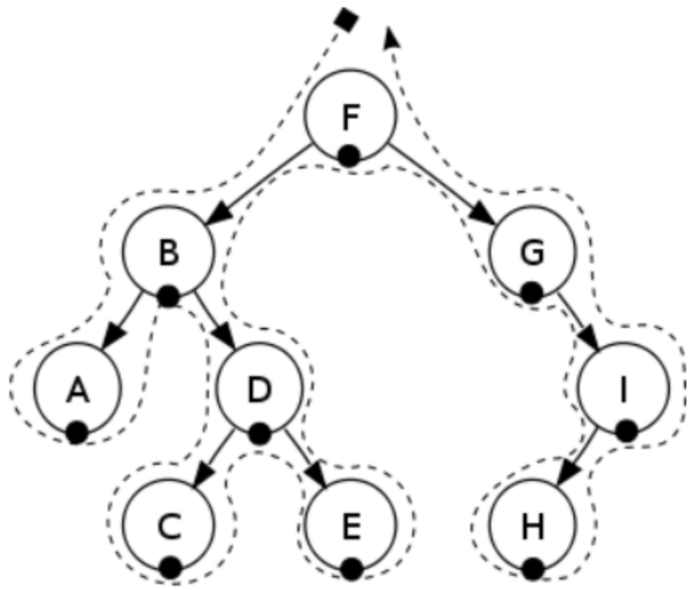
There are three ways to traverse a tree: pre-order traversal, in-order traversal, and post-order traversal. The traversals are mostly implemented in the Node class.

In-Order Traversal

An in-order traversal does the steps in the following order:

- Traverse the left Subtree
- Handle the current Node
- Traverse the right Subtree

This is best seen in the following diagram:



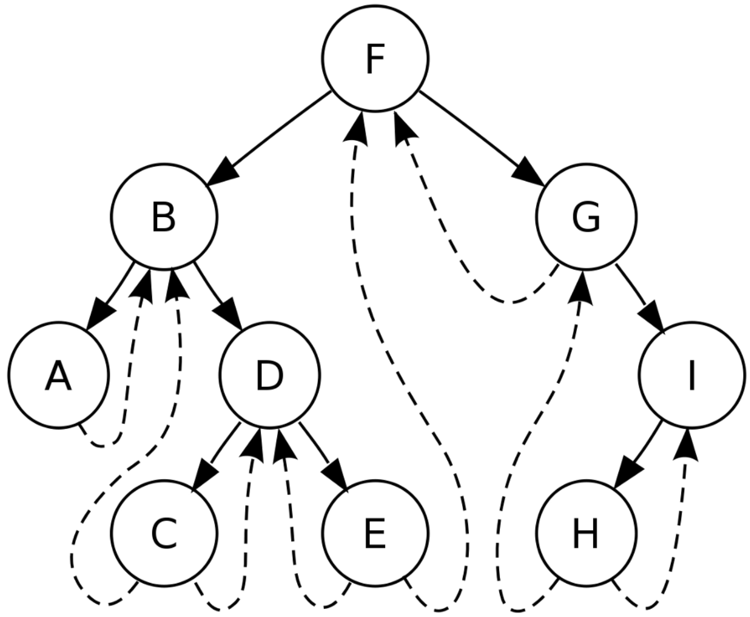
In-Order Traversal

Pre-order Traversal

A pre-order traversal does the above steps in the following order:

- Handle the current Node
- Traverse the left Subtree
- Traverse the right Subtree

This is best seen in the following diagram:



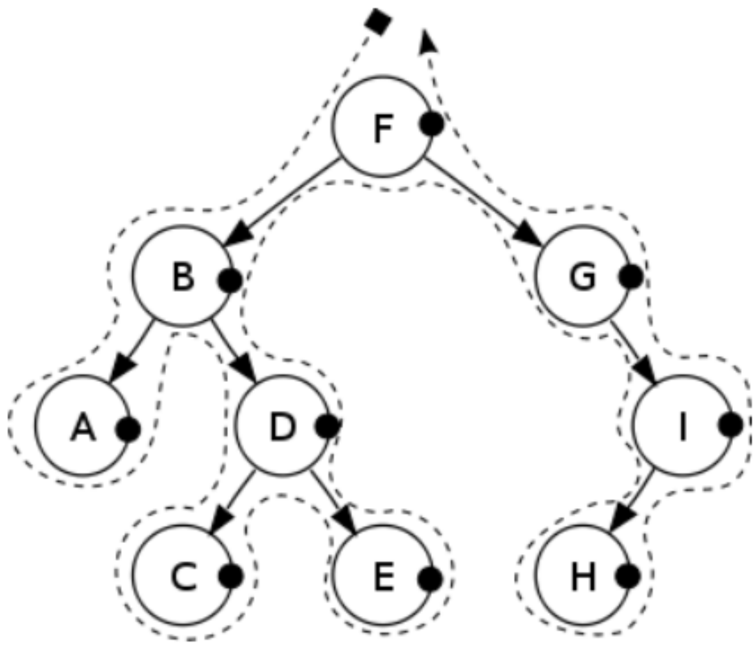
Pre-Order Traversal

Post-Order Traversal

You can guess in what order a post-order traversal accomplishes its tasks:

- Traverse the left Subtree
- Traverse the right Subtree
- Handle the current Node

This is best seen in the following diagram:



Post-Orde Traversal

Complexity

Best case: When the tree is balanced, we must traverse through a node after making h comparisons for searching a node which takes time which is directly proportional to the height of the tree (logN) and then copying the contents and deleting it requires constant time, so the overall time complexity is O(log N) which is the best-case time complexity.

Average case: Average case time complexity is same as best case so the time complexity in deleting an element in binary search tree is O(log N).

Note: Average Height of a Binary Search Tree is 4.31107 ln(N) - 1.9531 lnln(N) + O(1) that is O(logN).

Worst case: When we are given a left skewed or a right skewed tree(a tree with either no right subtree or no left subtree), then we have to traverse from root to last leaf node and the perform deletion process, so it takes O(n) time as height of the tree becomes 'n' in this case. So overall time complexity in worst case is O(n).

Space complexity: The space complexity of this algorithm would be O(n) with 'n' being the depth of the tree since at any point of time maximum number of stack frames that could be present in memory is 'n'.

Data Structure:

A linked list is a type of data structure which is connected via links. Each linked list has a data element which contains a connection to another data element in the form of a pointer. We implement the linked lists using nodes, as Python does not have library for linked list. Hence, we have used node data structure for implementation of linked list for build a binary search tree.

References:

https://medium.com/odscjournal/binary-search-tree-implementation-in-python-5f8a50341eaf

https://iq.opengenus.org/time-and-space-complexity-of-binary-search-tree/

Red Black Tree:

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

A red-black tree satisfies the following properties:

1. Red/Black Property: Every node is colored, either red or black.
2. Root Property: The root is black.
3. Leaf Property: Every leaf (NIL) is black.
4. Red Property: If a red node has children, then, the children are always black.
5. Depth Property: For each node, any simple path from this node to any of its descendant leaf has the same black depth (the number of black nodes).

Algorithm to insert a node

Following steps are followed for inserting a new element into a red-black tree:

1. Let y be the leaf (ie. NIL) and x be the root of the tree.
2. Check if the tree is empty (ie. whether x is NIL). If yes, insert newNode as a root node and color it black.
3. Else, repeat steps following steps until leaf (NIL) is reached.
    a. Compare newKey with rootKey.
    b. If newKey is greater than rootKey, traverse through the right subtree.
    c. Else traverse through the left subtree.
4. Assign the parent of the leaf as a parent of newNode.

5. If leafKey is greater than newKey, make newNode as rightChild.
6. Else, make newNode as leftChild.
7. Assign NULL to the left and rightChild of newNode.
8. Assign RED color to newNode.
9. Call InsertFix-algorithm to maintain the property of red-black tree if violated.

Algorithm to maintain red-black property after insertion

This algorithm is used for maintaining the property of a red-black tree if the insertion of a newNode violates this property.

1. Do the following while the parent of newNode p is RED.
2. If p is the left child of grandParent gP of z, do the following.

   Case-I:

   a. If the color of the right child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.

   b. Assign gP to newNode.

   Case-II:

   c. Else if newNode is the right child of p then, assign p to newNode.

   d. Left-Rotate newNode.

   Case-III:

   e. Set color of p as BLACK and color of gP as RED.

   f. Right-Rotate gP.

3. Else, do the following.

   g. If the color of the left child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.

   h. Assign gP to newNode.

I. Else if newNode is the left child of p then, assign p to newNode and Right-Rotate newNode.

j. Set color of p as BLACK and color of gP as RED.

k. Left-Rotate gP.

4. Set the root of the tree as BLACK.

```
# Implementing Red-Black Tree in Python


import time
import sys



# Node creation
class Node():
    def __init__(self, item):
        self.item = item
        self.parent = None
        self.left = None
        self.right = None
        self.color = 1



class RedBlackTree():
    def __init__(self):
        self.TNULL = Node(0)
        self.TNULL.color = 0
        self.TNULL.left = None
        self.TNULL.right = None
```

```
        self.root = self.TNULL

    # Preorder
    def pre_order_helper(self, node):
        if node != self.TNULL:
            sys.stdout.write(node.item + " ")
            self.pre_order_helper(node.left)
            self.pre_order_helper(node.right)

    # Inorder
    def in_order_helper(self, node):
        if node != self.TNULL:
            self.in_order_helper(node.left)
            sys.stdout.write(node.item + " ")
            self.in_order_helper(node.right)

    # Postorder
    def post_order_helper(self, node):
        if node != self.TNULL:
            self.post_order_helper(node.left)
            self.post_order_helper(node.right)
            sys.stdout.write(node.item + " ")

    # Search the tree
    def search_tree_helper(self, node, key):
        if node == self.TNULL or key == node.item:
            return node
```

```
        if key < node.item:
            return self.search_tree_helper(node.left, key)
        return self.search_tree_helper(node.right, key)

    # Balancing the tree after deletion
    def delete_fix(self, x):
        while x != self.root and x.color == 0:
            if x == x.parent.left:
                s = x.parent.right
                if s.color == 1:
                    s.color = 0
                    x.parent.color = 1
                    self.left_rotate(x.parent)
                    s = x.parent.right

                if s.left.color == 0 and s.right.color == 0:
                    s.color = 1
                    x = x.parent
                else:
                    if s.right.color == 0:
                        s.left.color = 0
                        s.color = 1
                        self.right_rotate(s)
                        s = x.parent.right

                    s.color = x.parent.color
                    x.parent.color = 0
                    s.right.color = 0
```

```
                self.left_rotate(x.parent)
                x = self.root
        else:
            s = x.parent.left
            if s.color == 1:
                s.color = 0
                x.parent.color = 1
                self.right_rotate(x.parent)
                s = x.parent.left

            if s.right.color == 0 and s.right.color == 0:
                s.color = 1
                x = x.parent
            else:
                if s.left.color == 0:
                    s.right.color = 0
                    s.color = 1
                    self.left_rotate(s)
                    s = x.parent.left

                s.color = x.parent.color
                x.parent.color = 0
                s.left.color = 0
                self.right_rotate(x.parent)
                x = self.root
    x.color = 0

def __rb_transplant(self, u, v):
```

```
        if u.parent == None:
            self.root = v
        elif u == u.parent.left:
            u.parent.left = v
        else:
            u.parent.right = v
        v.parent = u.parent

    # Node deletion
    def delete_node_helper(self, node, key):
        z = self.TNULL
        while node != self.TNULL:
            if node.item == key:
                z = node

            if node.item <= key:
                node = node.right
            else:
                node = node.left

        if z == self.TNULL:
            print("Cannot find key in the tree")
            return

        y = z
        y_original_color = y.color
        if z.left == self.TNULL:
            x = z.right
```

```
            self.__rb_transplant(z, z.right)
        elif (z.right == self.TNULL):
            x = z.left
            self.__rb_transplant(z, z.left)
        else:
            y = self.minimum(z.right)
            y_original_color = y.color
            x = y.right
            if y.parent == z:
                x.parent = y
            else:
                self.__rb_transplant(y, y.right)
                y.right = z.right
                y.right.parent = y

            self.__rb_transplant(z, y)
            y.left = z.left
            y.left.parent = y
            y.color = z.color
        if y_original_color == 0:
            self.delete_fix(x)

    # Balance the tree after insertion
    def fix_insert(self, k):
        while k.parent.color == 1:
            if k.parent == k.parent.parent.right:
                u = k.parent.parent.left
                if u.color == 1:
```

```
            u.color = 0
            k.parent.color = 0
            k.parent.parent.color = 1
            k = k.parent.parent
        else:
            if k == k.parent.left:
                k = k.parent
                self.right_rotate(k)
            k.parent.color = 0
            k.parent.parent.color = 1
            self.left_rotate(k.parent.parent)
    else:
        u = k.parent.parent.right

        if u.color == 1:
            u.color = 0
            k.parent.color = 0
            k.parent.parent.color = 1
            k = k.parent.parent
        else:
            if k == k.parent.right:
                k = k.parent
                self.left_rotate(k)
            k.parent.color = 0
            k.parent.parent.color = 1
            self.right_rotate(k.parent.parent)
    if k == self.root:
        break
```

```
        self.root.color = 0

    # Printing the tree
    def __print_helper(self, node, indent, last):
        if node != self.TNULL:
            sys.stdout.write(indent)
            if last:
                sys.stdout.write("R----")
                indent += "     "
            else:
                sys.stdout.write("L----")
                indent += "|    "

            s_color = "RED" if node.color == 1 else "BLACK"
            print(str(node.item) + "(" + s_color + ")")
            self.__print_helper(node.left, indent, False)
            self.__print_helper(node.right, indent, True)

    def preorder(self):
        self.pre_order_helper(self.root)

    def inorder(self):
        self.in_order_helper(self.root)

    def postorder(self):
        self.post_order_helper(self.root)

    def searchTree(self, k):
```

```
        return self.search_tree_helper(self.root, k)

    def minimum(self, node):
        while node.left != self.TNULL:
            node = node.left
        return node

    def maximum(self, node):
        while node.right != self.TNULL:
            node = node.right
        return node

    def successor(self, x):
        if x.right != self.TNULL:
            return self.minimum(x.right)

        y = x.parent
        while y != self.TNULL and x == y.right:
            x = y
            y = y.parent
        return y

    def predecessor(self,  x):
        if (x.left != self.TNULL):
            return self.maximum(x.left)

        y = x.parent
        while y != self.TNULL and x == y.left:
```

```
            x = y
            y = y.parent

        return y

    def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.TNULL:
            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.TNULL:
            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

    def insert(self, key):
        node = Node(key)
        node.parent = None
        node.item = key
        node.left = self.TNULL
        node.right = self.TNULL
        node.color = 1

        y = None
        x = self.root

        while x != self.TNULL:
            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)

    def get_root(self):
        return self.root

    def delete_node(self, item):
        self.delete_node_helper(self.root, item)

    def print_tree(self):
        self.__print_helper(self.root, "", True)

def red_black_tree(arr,x):
    rbt = RedBlackTree()
    for i in arr:
```

```
        rbt.insert(i)
    t4_start=time.perf_counter()
    result=rbt.searchTree(x)
    t4_stop=time.perf_counter()
    t4=str(t4_stop-t4_start)+"s"
    print(result.item)
    if(result.item==0):
        print("Key Not Found")
        #print("--- %s seconds ---" % str(t4_stop - t4_start))
        return (-1,t4)
    else:
        print("Key Found",result.item)
        #print("--- %s seconds ---" % str(t4_stop - t4_start))
        return (1,t4)
```

Complexity:

OPERATION	AVERAGE CASE	WORST CASE
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Insertion Time

Best Case: In the best case, there is no rotation. Only recoloring takes place. The time complexity is O(log n).

Worst case: RB trees require a constant (at most 2 for insert) number of rotations. So in the worst case, there will be 2 rotations while insertion. The time complexity is O(log n).

Average Case: Since the average case is the mean of all possible cases, the time complexity of insertion in this case too is O(log n).

Deletion Operation

Best Case: In the best case, there is no rotation. Only recoloring takes place. The time complexity is O(log n).

Worst case: RB trees require a constant (at most 3 for deletion) number of rotations. So in the worst case, there will be 3 rotations while deletion. The time complexity is O(log n).

Average Case: Since the average case is the mean of all possible cases, the time complexity of deletion in this case too is O(log n).

Space Complexity

The average and worst space complexity of a red-black tree is the same as that of a Binary Search Tree and is determined by the total number of nodes: O(n) because we don't need any extra space to hold duplicate data structures. We arrive to this conclusion because each node has three pointers: left child, right child, and parent. Each node takes up O(1) space. As a result, if the tree has n total nodes, the space complexity is n times O(1), which is O(n).

Data Structure:

A linked list is a type of data structure which is connected via links. Each linked list has a data element which contains a connection to another data element in the form of a pointer. We implement the linked lists using nodes, as Python does not have library for linked list. Hence, we have used node data structure for implementation of linked list for build a Red Black tree.

References:

https://www.programiz.com/dsa/red-black-tree

https://iq.opengenus.org/time-and-space-complexity-of-red-black-tree/

Experimental Results:

How their running times change with respect to data size

In my implementation,

Entered array size: 15 and selected random series.

[59, 14, 89, 1, 35, 15, 44, 28, 54, 26, 12, 73, 76, 72, 25]

Your series is: 59,14,89,1,35,15,44,28,54,26,12,73,76,72,25

5

number value is : 54

random list is [59, 14, 89, 1, 35, 15, 44, 28, 54, 26, 12, 73, 76, 72, 25]

element found in array : True

54

Key Found 54

Linear Search 5.899999997893701e-06s

Binary Search 4.600000000465343e-06s

Binary Seach Tree 2.69999999957804e-06s

Red Black Tree 4.200000002896331e-06s

As per the comparison of runtime and input size is 15 we can see that binary search tree taking less time to find key compare than other algorithms.

Entered array size: 25 and selected random series

Your series is: 44,6,65,11,5,72,17,70,7,79,59,87,71,66,74,55,22,48,49,27,89,34,37,30,78

5

number value is : 55

random list is [44, 6, 65, 11, 5, 72, 17, 70, 7, 79, 59, 87, 71, 66, 74, 55, 22, 48, 49, 27, 89, 34, 37, 30, 78]

element found in array : True

55

Key Found 55

Linear Search 5.999999999062311e-06s

Binary Search 1.1400000001771105e-05s

Binary Seach Tree 2.500000000793534e-06s

Red Black Tree 5.700000002661909e-06s

As per the comparison of runtime and input size is 25, we can see that binary search takes less time to find key compared than other algorithms.

Entered array size: 50 and selected random series

Your series is:
46,31,53,65,40,43,70,84,27,25,54,89,12,91,11,18,93,64,44,52,97,1,57,28,29,42,88,80,86,47,13,82,90,83,49,24,26,2,5,15,48,41,36,61,58,99,95,72,22,30

5

number value is : 80

random list is [46, 31, 53, 65, 40, 43, 70, 84, 27, 25, 54, 89, 12, 91, 11, 18, 93, 64, 44, 52, 97, 1, 57, 28, 29, 42, 88, 80, 86, 47, 13, 82, 90, 83, 49, 24, 26, 2, 5, 15, 48, 41, 36, 61, 58, 99, 95, 72, 22, 30]

element found in array : True

80

Key Found 80

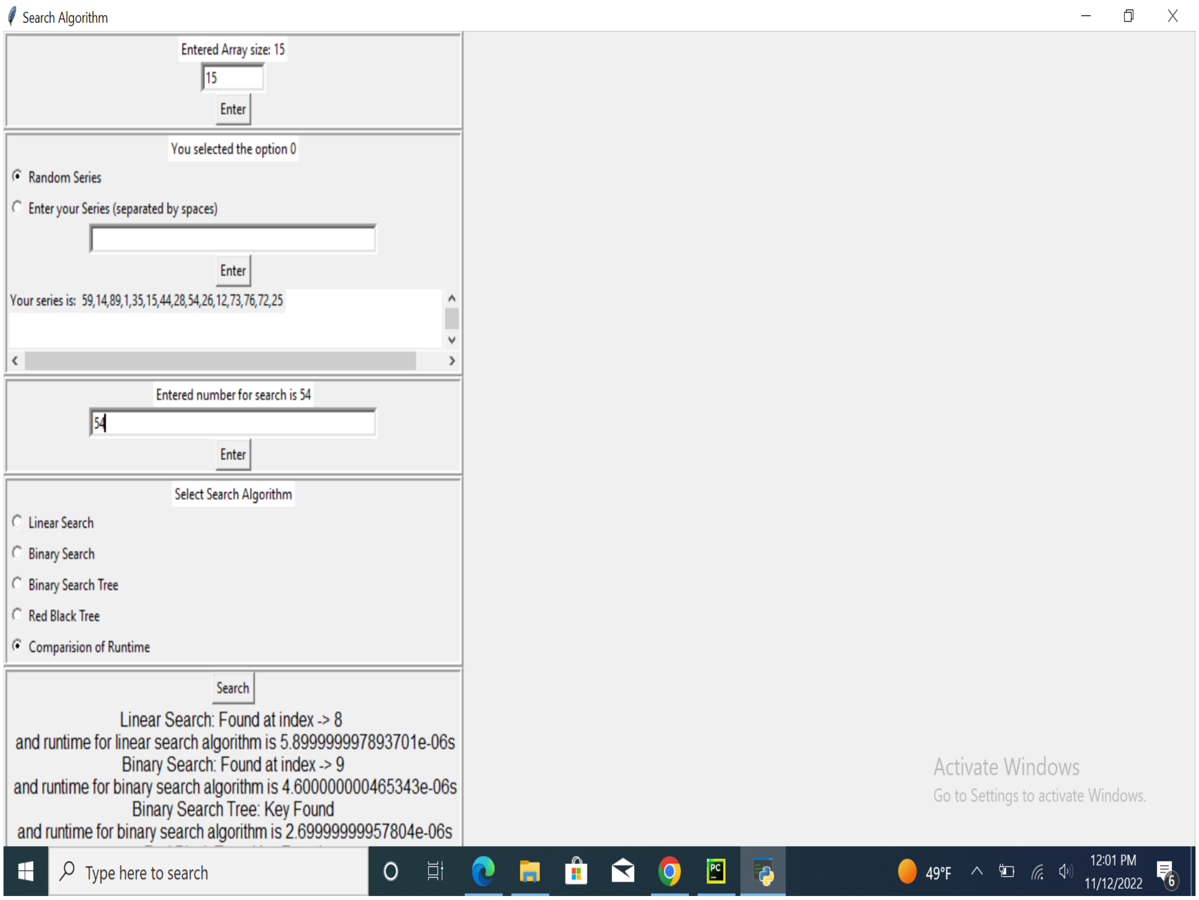
Linear Search 6.599999998968542e-06s

Binary Search 3.800000001774606e-06s

Binary Seach Tree 3.5999999994373866e-06s

Red Black Tree 4.400000001680837e-06s

As per the comparison of runtime and input size is 50, we can see that binary search tree takes less time to find key compared than other algorithms.



Screenshot of implementation

Which one is better in terms of what conditions?

Based on experimental Results, Binary search Tree is taking less time to find out the key compared than other algorithms. Implementing a binary search tree is useful in any situation where the elements can be compared in a less than / greater than manner. A binary search tree is a data structure that allows for fast insertion, removal, and lookup of items while offering an efficient way to iterate them in sorted order. And its worst time complexity is higher than other. So, I think BST is better for searching Algorithms.

Design of the user-interface:

For the design of the user interface, I used tkinter in ptyhon.