Project 2 (Search Algorithms)

Implement and compare the following search algorithm:
● Linear search
● Binary search in sorted array.
● Binary search tree
● Red-Black Tree

Table of Contents

Main data structures:

1. **Linear search data structure:** For Linear search in this project, random input array (generated using input size or array given by user) is used.

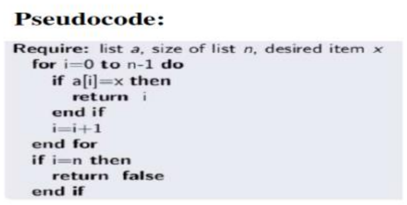
2. **Binary search in sorted array**: For Binary search in this project, random input array (generated using input size or array given by user) is first sorted using inbuilt python sort and then it is passed to search algorithm function.

3. **Binary search tree data structure**: For Binary search tree in this project, random input array (generated using input size or array given by user) is used for creating tree data structure. The first step is to create a tree node using a class data structure which has 3 elements, node value, left child and right child. This tree node is applied on each array element to create individual tree nodes which attaches to left or right child node based on its value, if it is lesser than current node value then it is attached to left child otherwise to right child.

4. **Red-Black Tree data structure:** For Red black tree in this project, random input array (generated using input size or array given by user) is used for creating tree data structure. The first step here is to create a tree node using a class data structure which has 5 elements - node value, left child, right child, parent of node and color of node. Just like BST, this tree node is applied on each array element to create individual tree nodes which attaches to left or right child node based on its value, if it is lesser than current node value then it is attached to left child otherwise to right child. As red-black tree is a binary search tree with one extra attribute for each node: the color, which is either red or black. We also need to keep track of the parent of each node.

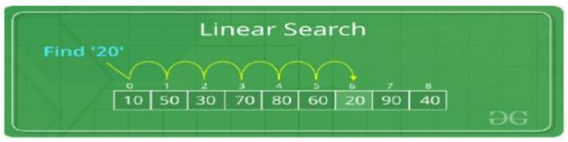
Details about main data structures and time complexity:

1. Linear search: 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.



Pseudocode:

```
Require: list $a$, size of list $n$, desired item $x$
  for i=0 to n-1 do
    if a[i]=x then
      return i
    end if
    i=i+1
  end for
  if i=n then
    return false
  end if
```



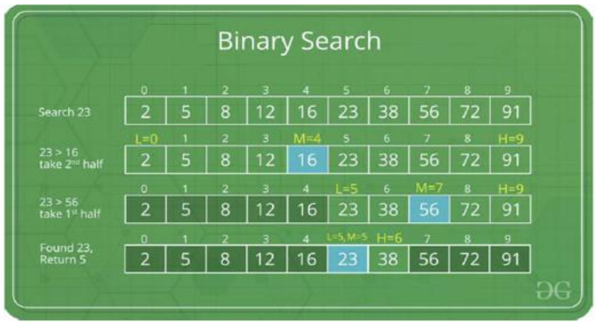
- Best case complexity: When the element to be searched is found at first position. The best case time complexity is O(1).
- Average case complexity: When the element to be searched is found in the middle of array. The average case time complexity is O(n).
- Worst case complexity: When the element to be searched is found at the last position in array or not found at all. The worst-case time complexity is O(n).

2. Binary search: This algorithm is specifically designed for searching in sorted data-structures. 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 in 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

Pseudocode:

```
BinarySearch(list[], min, max, key)
while min ≤ max do
   mid = (max+min) / 2
   if list[mid] >key then
      max = mid-1
   else if list[mid] <key then
      min = mid+1
   else
      return  mid
   end if
end while
return  false
```



- Best case complexity: When the element to be searched is found at middle position. The best case time complexity is O(1).
- Average case complexity: When the element to be searched is found at any random position in the array. The average case time complexity is O(logN).
- Worst case complexity: When the element to be searched is found at the first index or last index of array. The worst-case time complexity is O(logN).

3. Binary Search Tree **(BST)**: It is a node-based binary tree data structure which has the following properties:
   - The number of children in any node should be at most 2
   - The left subtree of a node contains only nodes with keys lesser than the node’s key.
   - The right subtree of a node contains only nodes with keys greater than the node’s key.
   - The left and right subtree each must also be a binary search tree.

   If we need to search for a number in BST, then we start with root and if the target is equal to value of root, then we found the element in BST and done with search. If target is smaller than root, then we go to left subtree and if it is greater. We go to right subtree and recursively perform these steps until we find the element.

- Best case complexity: When the element to be searched is found at root value. The best case time complexity is O(1).
- Average case complexity: When the element to be searched is somewhere between root element and maximum or minimum element of BST. The average case time complexity is O(logN).
- Worst case complexity: When the tree is a skewed binary search tree (left or right skewed) and the element you are searching is minimum or maximum. The worst-case time complexity is O(N).

Algorithm for creating BST:

```
// Adds a new node and returns the root of the updated tree.
Algorithm insert(node, root)
if root is empty
return node
if node.value = root.value
// do nothing, element already in tree
else if node.value < root.value
root.left ← insert(node, root.left)
else
root.right ← insert(node, root.right)
return root
```

Algorithm for searching in BST:

```
// Returns true if the value is found in the tree.
Algorithm find(value, root)
if root is empty
return false
if value = root.value
return true
if value < root.value
return find(value, root.left)
else
return find(value, root.right)
```

4. Red Black Tree:

A red-black tree is a binary search tree with one extra attribute for each node: the color, which is either red or black. We also need to keep track of the parent of each node.
A red-black tree is a binary search tree which has the following red-black properties:

- Every node is either red or black.
- The root node is always black.
- If a node is red, then both its children are black.
- Every simple path from a node to a descendant leaf contains the same number of black nodes.

The goal of the insert operation is to insert key K into tree T, maintaining T's red-black tree properties. A special case is required for an empty tree. If T is empty, replace it with a single black node containing K. This ensures that the root property is satisfied.
If T is a non-empty tree, then we do the following:

- use the BST insert algorithm to add K to the tree
- color the node containing K red
- restore red-black tree properties (if necessary)

Restoring red black properties:

- If tree is empty, create a new node as root node and color it black and exit
- If tree is not empty create a new node as leaf node and color it red.
- If parent of new node is black, then exit
- If parent of new node is red, then check color of parent’s sibling of new node:
  a) If color is black or null, then do suitable rotation and recolor parent and grandparent
  b) If color is Red then recolor (parent and also check if parent’s parent of new node is not root node then recolor it and check again.

Searching in RBT:
Search operation in RBT is same as BST but it is kind of self-balancing BST so searching is faster in RBT.

- Best case complexity: When the element to be searched is found at root value. The best case time complexity is O(1).
- Average case complexity: When the element to be searched is somewhere between root element and maximum or minimum element of RBT. The average case time complexity is O(logN).
- Worst case complexity: When the element to be searched is at extreme left or right.The worst case time complexity is O(logN).

Time Complexity	Best case	Average case	Worst case
Linear search	O(1)	O(n)	O(n)
Binary search	O(1)	O(logn)	O(logn)
Binary search tree	O(1)	O(logn)	O(n)
Red-black tree	O(1)	O(logn)	O(logn)

Main components of the algorithm (different functions and what they do):

1. ### Linear search:
   Function : linear_search(inputarray, searchvalue)
   This is a function for performing search operation in array.
   In this function we are passing input array and search value given by user as parameters to function and inside this function a for loop is running and if value matches the value of input array then index of that value is returned else false is returned.

2. ### Binary search:
   Function: binary_search(inputarray, searchvalue):
   This is a function for performing search operation in sorted array.
   - Inside this function we have declared start, mid as 0 initially and end is length of array-1.
   - A while loop runs if start is less than end
   - Inside while loop we calculate the value of mid(index position) which is average value of start and end
   - Then we check 3 conditions:
     a) If the value to be searched is equal to middle index value, then return mid value
     b) If search value is less than value at middle index, then keep the start as it is and set end =mid-1.
     c) If search value is greater than value at middle index, then keep the end as it is and set start =mid+1.
     d) If the value is found, we return true else false

3. Binary search tree:

Data structure

- **Class nodeBST**
  This class holds the node value, left child and right child.

Functions:

- **__init__(data)**
  This function for BST is defined in class nodeBST and the parameters passed inside this is data. This function creates a tree node with data as node value with left and right child null.

- insert(NodeBST, data):This function is defined outside the class nodeBST and it is used to insert the data given by user. It takes 2 parameters NodeBST and data. In this function first we check if the tree is empty and if it is empty then it creates an object of class nodeBST. Otherwise, we further check the data and if it less than the root value then a new node is created, and it is attached to left subtree otherwise attached to right subtree.

- bstSearch(rootBST, key):This is used to search for data in BST and values passed are root and the search element given by user. This function returns true if key element exists in tree otherwise false.

- preorderTraversalBST(rootBST): This function returns preorder traversal of Binary search tree.

4. Red black tree:

Class:

- **RBTNode**
  This class holds the node value, left child and right child, parent, color.

Functions:

- def __init__(data): This function initialize RBTNode with data, color as black and left child, right child and parent as none to create an individual red black tree node.

Class:

- RBTree
  This class holds the RBTNode, left child and right child, color.

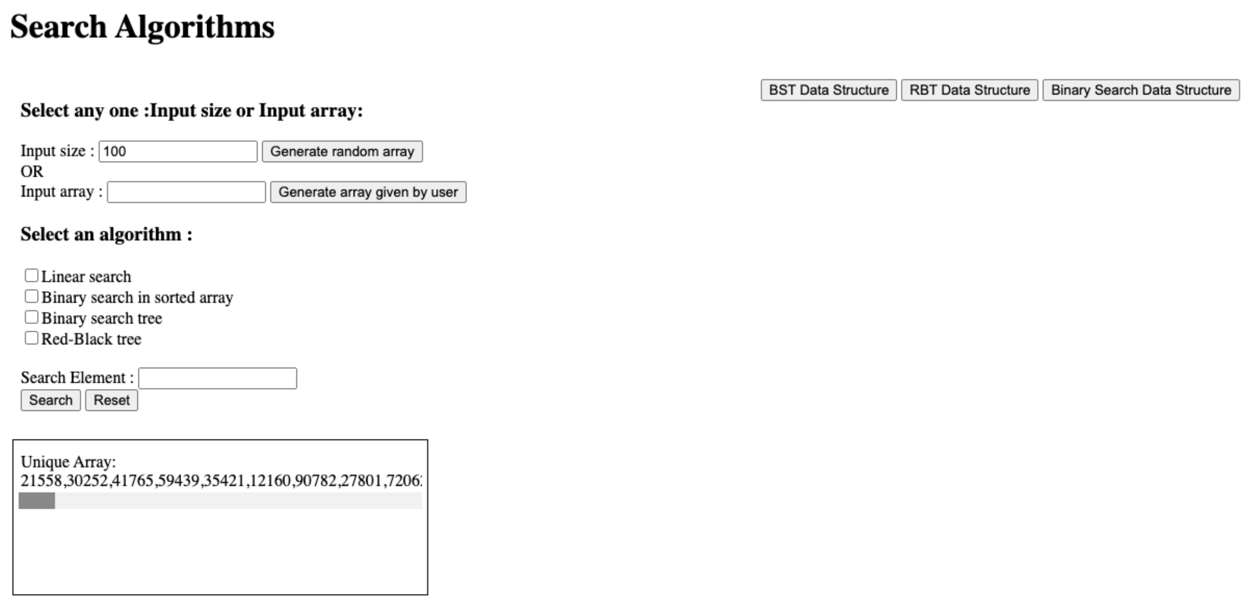
Functions:

- def __init__(): To initialize the RBTree with RBTNode with 0 value and color to black.
- def insertRBTNode(key): This function inserts a new value in RBTree DS. It utilizes RBTNode to create a tree node with key as value and its color as red, then it inserts this new node based on its value to left or right position (to left if key value is smaller than node value otherwise to the right child). At the end it calls the fixPosition to balance the tree according to Red Black tree property.
- def fixPosition(fixNode): This function is used to fix new node position in red black tree if new node violate the red black tree property. This function calls left rotate or right rotate based on red black rotation algorithms.
- def rotateLeft (newNode): In this function the arrangements of the nodes in right are transformed to arrangements on the left node.
- def rotateRight (newNode): In this function the arrangements of the nodes in left are transformed to arrangements on the right node.
- def preorderTraversalRBT(): This function returns preorder traversal of RBTree
- def search(key): This function returns true if key element exists in tree otherwise false.

Design of the user-interface:

User needs to select either of the 2 options (Input size or input array).

1. If user gives input size, then need to click on “Generate random array” button and a unique array is generated:



Search Algorithms

Select any one :Input size or Input array:

Input size : 100

OR

Input array :

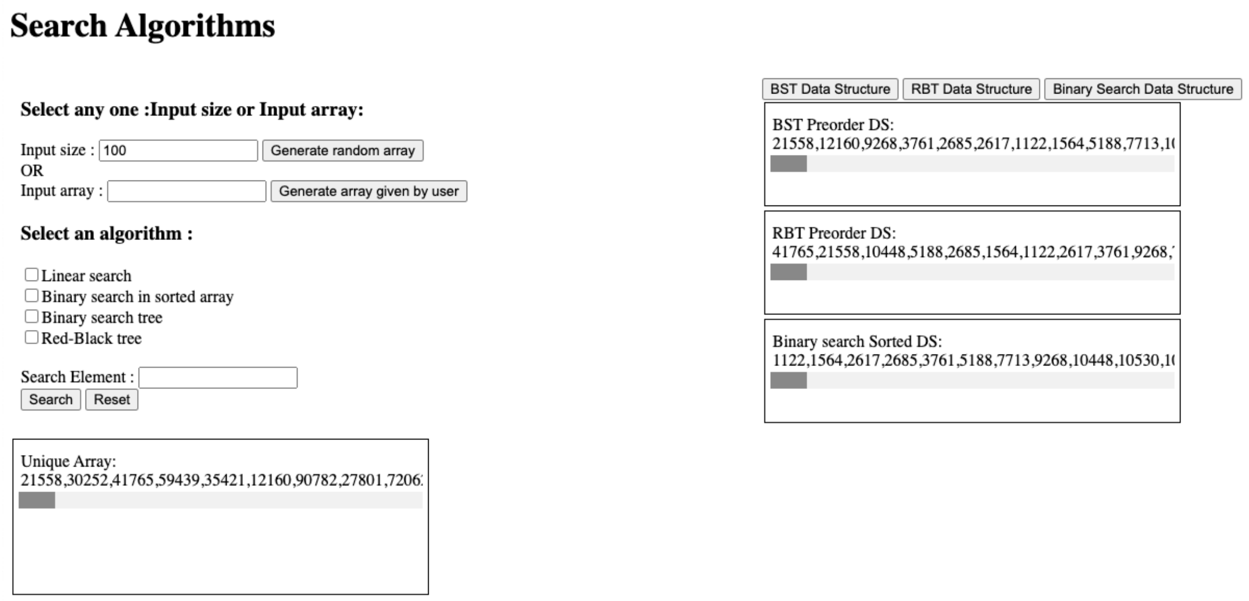
Select an algorithm :

- Linear search
- Binary search in sorted array
- Binary search tree
- Red-Black tree

Search Element :

Unique Array:
21558,30252,41765,59439,35421,12160,90782,27801,7206

If user further clicks on buttons “BST data structure”,”RBT data structure” and “binary search data structure “then these data structures are displayed:



Search Algorithms

Select any one :Input size or Input array:

Input size : 100

OR

Input array :

Select an algorithm :

- Linear search
- Binary search in sorted array
- Binary search tree
- Red-Black tree

Search Element :

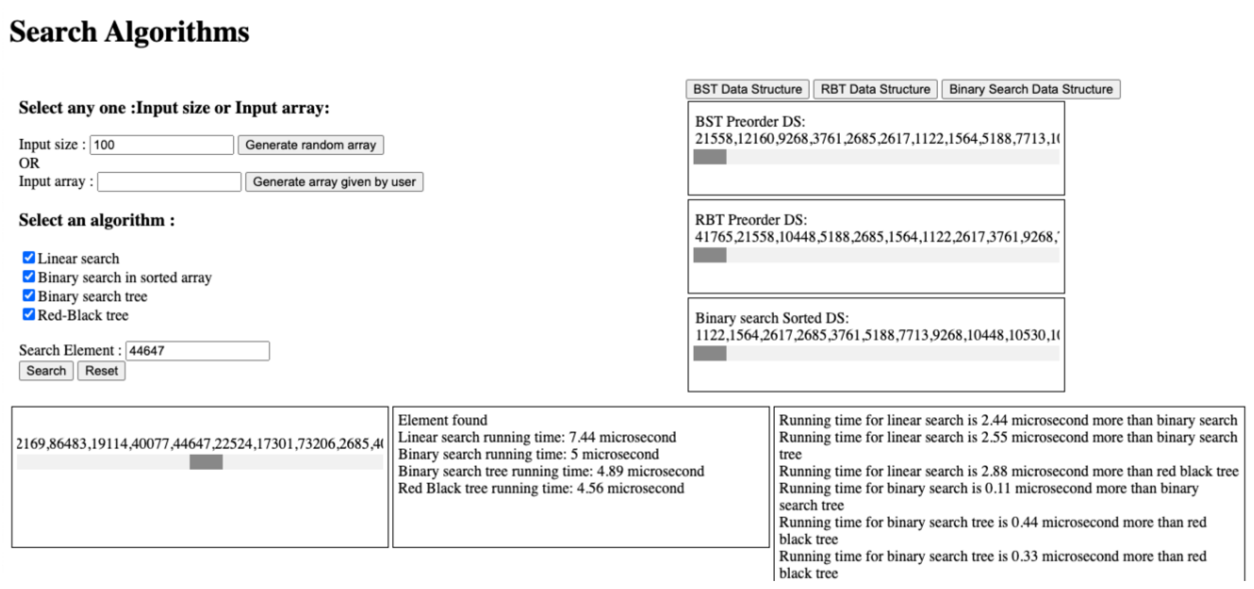
Unique Array:
21558,30252,41765,59439,35421,12160,90782,27801,7206

BST Preorder DS:
21558,12160,9268,3761,2685,2617,1122,1564,5188,7713,1

RBT Preorder DS:
41765,21558,10448,5188,2685,1564,1122,2617,3761,9268,

Binary search Sorted DS:
1122,1564,2617,2685,3761,5188,7713,9268,10448,10530,1

Then user needs to select the checkboxes of algorithm which needs to be compared and then give the search element and click on search. This will display the running time of all algorithms and their comparison with each other:



Search Algorithms

Select any one :Input size or Input array:

Input size : 100 Generate random array
OR
Input array : Generate array given by user

Select an algorithm :

- [✓] Linear search
- [✓] Binary search in sorted array
- [✓] Binary search tree
- [✓] Red-Black tree

Search Element : 44647
Search Reset

BST Preorder DS:
21558,12160,9268,3761,2685,2617,1122,1564,5188,7713,1(

RBT Preorder DS:
41765,21558,10448,5188,2685,1564,1122,2617,3761,9268,

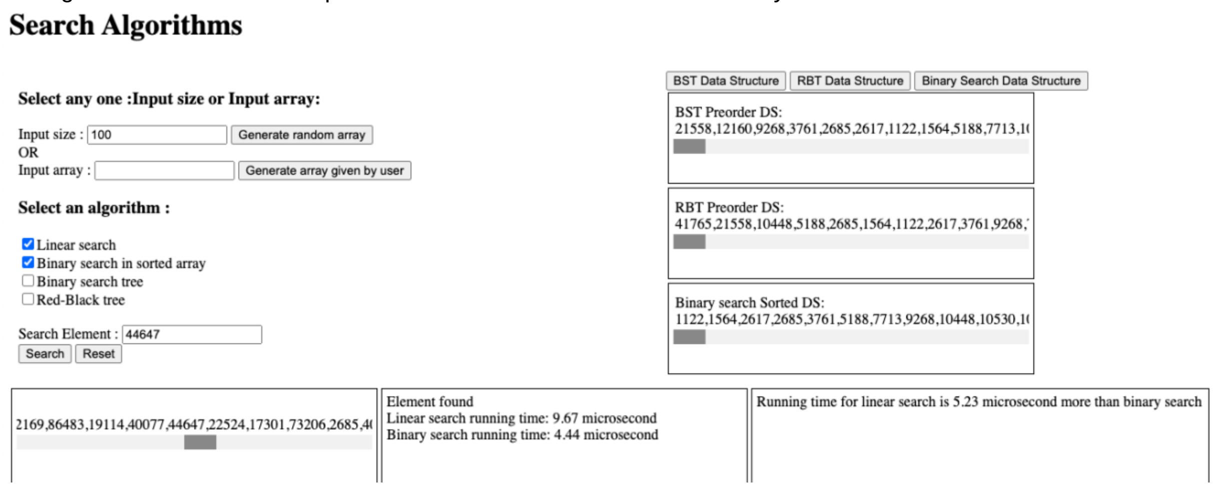
Binary search Sorted DS:
1122,1564,2617,2685,3761,5188,7713,9268,10448,10530,1(

2169,86483,19114,40077,44647,22524,17301,73206,2685,4(

Element found
Linear search running time: 7.44 microsecond
Binary search running time: 5 microsecond
Binary search tree running time: 4.89 microsecond
Red Black tree running time: 4.56 microsecond

Running time for linear search is 2.44 microsecond more than binary search
Running time for linear search is 2.55 microsecond more than binary search tree
Running time for linear search is 2.88 microsecond more than red black tree
Running time for binary search is 0.11 microsecond more than binary search tree
Running time for binary search tree is 0.44 microsecond more than red black tree
Running time for binary search tree is 0.33 microsecond more than red black tree

If 2 algorithms need to be compared then user need to select those 2 only:



Search Algorithms

Select any one :Input size or Input array:

Input size : 100 Generate random array
OR
Input array : Generate array given by user

Select an algorithm :

- [✓] Linear search
- [✓] Binary search in sorted array
- [ ] Binary search tree
- [ ] Red-Black tree

Search Element : 44647
Search Reset

BST Preorder DS:
21558,12160,9268,3761,2685,2617,1122,1564,5188,7713,1(

RBT Preorder DS:
41765,21558,10448,5188,2685,1564,1122,2617,3761,9268,

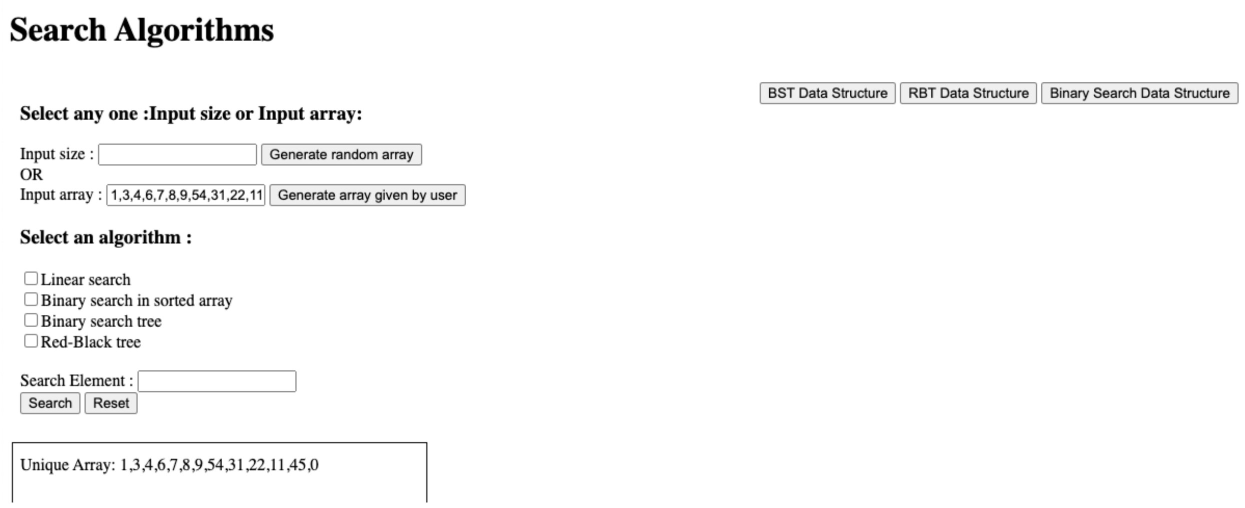
Binary search Sorted DS:
1122,1564,2617,2685,3761,5188,7713,9268,10448,10530,1(

2169,86483,19114,40077,44647,22524,17301,73206,2685,4(

Element found
Linear search running time: 9.67 microsecond
Binary search running time: 4.44 microsecond

Running time for linear search is 5.23 microsecond more than binary search

2. User can also select the option to give the input array and then click onbutton "Generate array given by user":



Search Algorithms

Select any one :Input size or Input array:

Input size : Generate random array
OR
Input array : 1,3,4,6,7,8,9,54,31,22,11 Generate array given by user

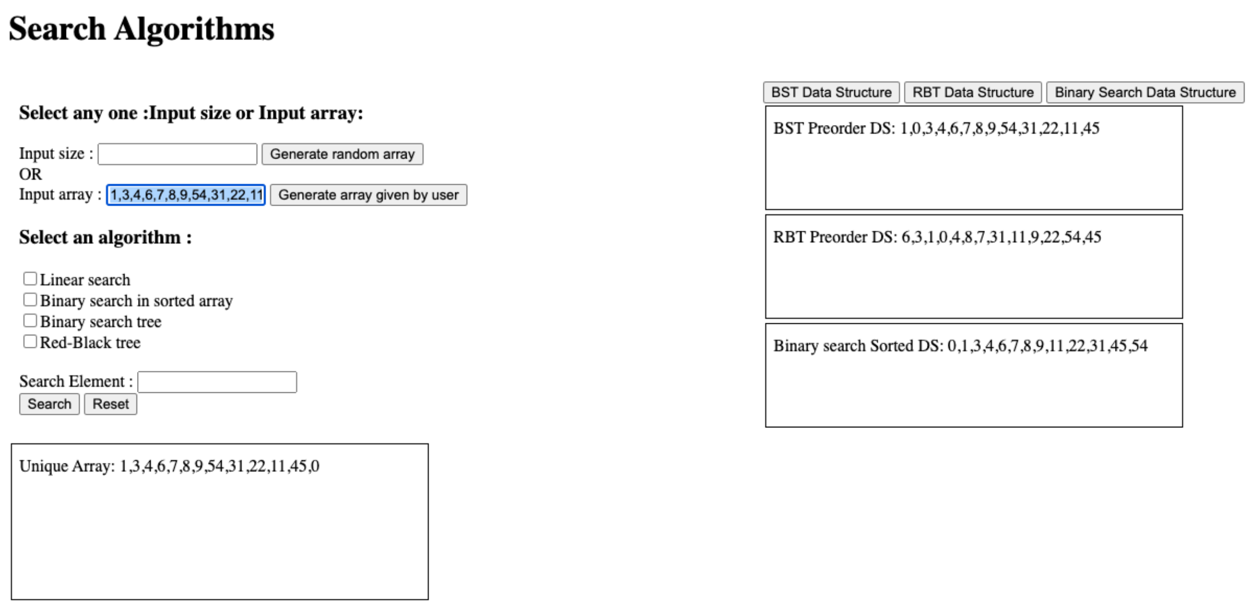
Select an algorithm :

- [ ] Linear search
- [ ] Binary search in sorted array
- [ ] Binary search tree
- [ ] Red-Black tree

Search Element :
Search Reset

Unique Array: 1,3,4,6,7,8,9,54,31,22,11,45,0

User can again see BST, RBT and Binary search data structure by clicking on respective buttons:



Search Algorithms

Select any one :Input size or Input array:

Input size :
OR
Input array : 1,3,4,6,7,8,9,54,31,22,11

Select an algorithm :

☐ Linear search
☐ Binary search in sorted array
☐ Binary search tree
☐ Red-Black tree

Search Element :

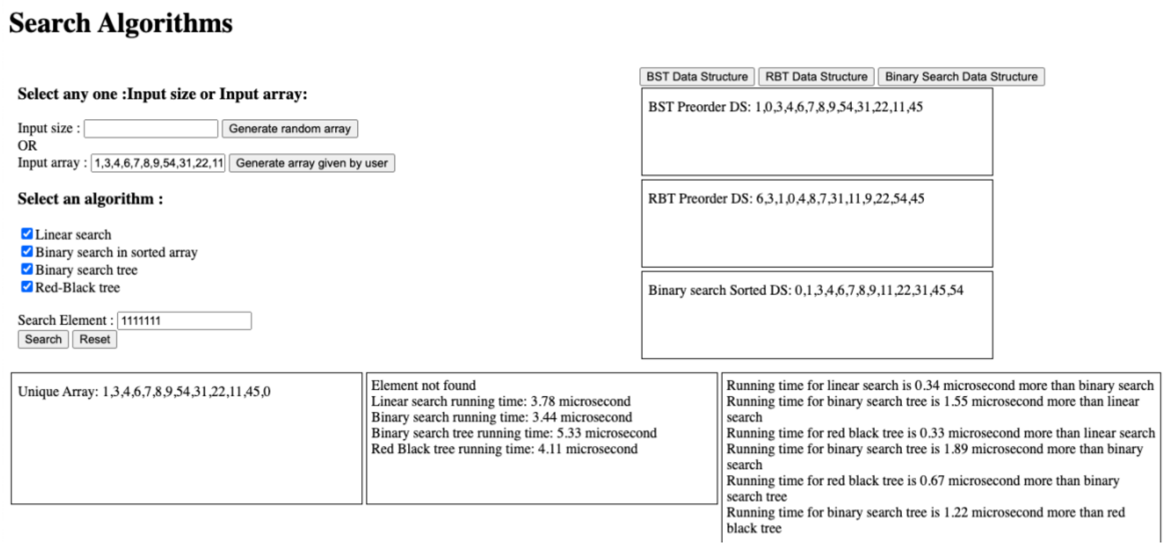
```
Unique Array: 1,3,4,6,7,8,9,54,31,22,11,45,0
```

```
BST Preorder DS: 1,0,3,4,6,7,8,9,54,31,22,11,45
```

```
RBT Preorder DS: 6,3,1,0,4,8,7,31,11,9,22,54,45
```

```
Binary search Sorted DS: 0,1,3,4,6,7,8,9,11,22,31,45,54
```

User can further select the algorithms for which results are required and give the search element, if the element is found then in the output window "Element found" is displayed else "element not found" is displayed and along with this running time and comparison between running time is displayed:



Search Algorithms

Select any one :Input size or Input array:

Input size :
OR
Input array : 1,3,4,6,7,8,9,54,31,22,11

Select an algorithm :

✓ Linear search
✓ Binary search in sorted array
✓ Binary search tree
✓ Red-Black tree

Search Element : 1111111

```
Unique Array: 1,3,4,6,7,8,9,54,31,22,11,45,0
```

```
Element not found
Linear search running time: 3.78 microsecond
Binary search running time: 3.44 microsecond
Binary search tree running time: 5.33 microsecond
Red Black tree running time: 4.11 microsecond
```

```
Running time for linear search is 0.34 microsecond more than binary search
Running time for binary search tree is 1.55 microsecond more than linear
search
Running time for red black tree is 0.33 microsecond more than linear search
Running time for binary search tree is 1.89 microsecond more than binary
search
Running time for red black tree is 0.67 microsecond more than binary
search tree
Running time for binary search tree is 1.22 microsecond more than red
black tree
```

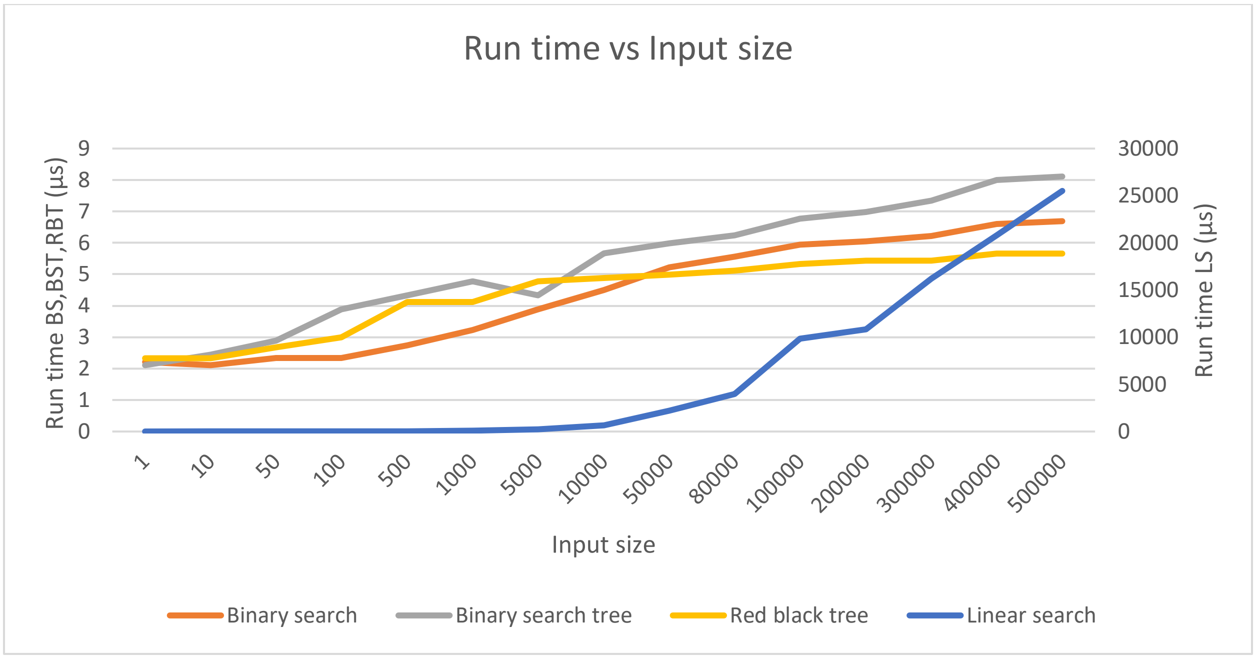
```
BST Preorder DS: 1,0,3,4,6,7,8,9,54,31,22,11,45
```

```
RBT Preorder DS: 6,3,1,0,4,8,7,31,11,9,22,54,45
```

```
Binary search Sorted DS: 0,1,3,4,6,7,8,9,11,22,31,45,54
```

Charts of running time versus input size:

Considering average case of all searching algorithm below is the chart for run time as input size grows:


Run time vs Input size
Run time BS,BST,RBT (µs)
Run time LS (µs)
Input size
Binary search
Binary search tree
Red black tree
Linear search