PES UNIVERSITY

100 feet Ring Road, BSK 3rd Stage, Bengaluru
560085

UE17CS251

Design and Analysis of Algorithms

Project Report
Data Structure Set

PES1201700986	Pradyumna YM	4’F’
PES1201700924	Anush V Kini	4’F’

Abstract

A set is a well-defined collection of distinct objects. With various applications in various fields,particularly the mathematical field and computer science field, sets are of great significance.

As a consequence, it finds itself as a common data structure in various computer languages such as python, java etc. However, we find that the C language does not have an implementation of the data structure set, which is a very commonly used data structure that has a very efficient underlying data structure such as a hash table or a red black tree, which allows for efficient inserts,deletes and search times. So, we have tried to implement a set data structure, similar to that of C++ standard library.

Problem Statement

In this project we aim to build the data structure set, and to Implement all the set related functions, along with a red-black tree underlying data structure, which will limit the worst case efficiency of the set for various operations in case of skewed sequences. (Increasing/decreasing sequences). This project aims to create a module to mimic the set data structure along with its associated mathematical functions in the C programming language.

In this project, we have implemented the basic mathematical set functions union, intersection, set difference and symmetric difference.Along with these functions, we have implemented other utility functions like kth smallest element, kth largest element among others to act as helper functions.Further, we have implemented other functions which display information about the underlying data structure as a viewpoint for comparison with other data structures such as an ordinary BST and arrays.

Introduction

A set data structure is an ordered list without repeated elements in the C++ standard library, which uses a red black tree as its underlying data structure. A red black tree is a BST that is self-balancing, and hence helps in maintaining the worst case Search/Insert/Delete efficiency O(log(n)). The balancing of the tree is not perfect, but it is good enough to allow it to guarantee searching in O(log(n)) time, where n is the total number of elements in the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in O(log(n)) time. We considered a variety of underlying data structures before arriving at the conclusion to use a red black tree as the underlying data structure. The other data structures considered along with the reason for discarding them are listed below:

- **Arrays**: We also considered using an array to implement the set data structure, but we were convinced that they are not as efficient as the other options. Even if we use a sorted array and insert elements in sorted order, we will have to move all the subsequent elements, which will take us O(n) time at worst, which is not feasible for large number of elements and inserts.
- **Linked list**: An ordered link list can be used to overcome the above demerit. But, a linked list does not support random access, which defeats the original purpose of sorting(log(n) access time using binary search).
- **Binary search tree**: A worst case efficiency of O(n) in searching made us reconsider the use of binary search trees.
- **AVL trees**: Even though AVL trees share the efficiency of O(logn) with red black trees for searching, they suffer during the insertion of elements. During insertion, AVL trees can undergo a lot of rotations which increase the amount of computation required to get it back to AVL form.
- **Heaps**: Heaps are not suitable data structures for our application.(Ordered set).

Testing for ideal data structure

We wrote a program to perform inserts into a red black tree, a binary search tree, and an array and analysed the results. After a number of inserts, we tabulated the height of the red black tree, the height of the binary search tree and the length of the array.

Further, using Scilab we plotted a graph comparing the height of the binary search tree and the red black tree. The red black tree was the clear superior of the three as shown by the table and graph below.

Number of insertions	**Height of Red Black Tree**	**Height of Binary Search Tree**	**Length of Array**
100	7	12	100
1000	11	20	1000
2500	13	23	2500
5000	13	27	5000
10000	15	33	10000
25000	17	37	25000
50000	18	39	50000
100000	20	39	100000
1000000	23	47	1000000
2000000	25	50	2000000
4000000	26	52	4000000
6000000	26	57	6000000
8000000	27	59	8000000
10000000	27	57	10000000

Table 1: A tabulation of inserts performed on various data structures

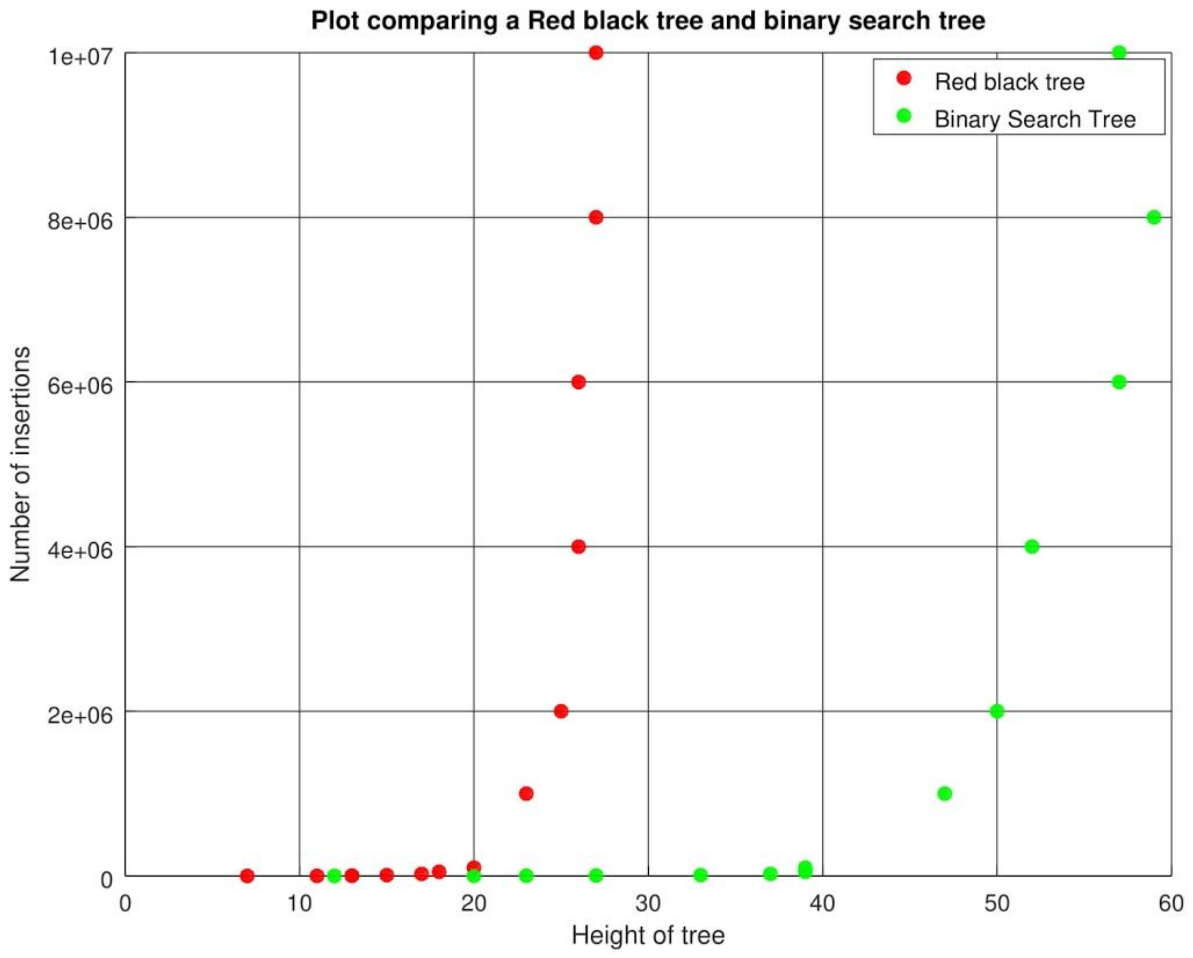


Fig 1: A plot of heights of the red black tree and binary search tree

Hence, after considering our options, we decided to use a red black tree as our underlying data structure for the set.

Design

The basic structure definiton of our set data structure is as follows:

```
typedef struct node{
    int value;
    int color;
    int deleted;
    struct node *parent;
    struct node *left;
    struct node *right;
} Node,set;
```

The various field and their purposes:

- value: used to store the data value of that particular node.
- color: represents the color of the node in the red-black tree. 0 represents red and 1 represents black.
- deleted:used to represent deleted nodes in the set, used instead of removing the nodes of a set on deletion of a key.
- parent:used to reference the parent node of the current node. This field is required for the various rotations that the red black tree undergoes on insertion of elements.
- left:a pointer to the left subtree of the current node
- Right: a pointer to the right subtree of the current node.

The client creates a NULL pointer to a set structure, which will later serve as a pointer to the root of the red black tree. For each of the set functions, the client passes a reference to this pointer (a double pointer). The functions change the pointer to the root of the tree in case it was NULL, or in case it was changed due to a balancing rotation.

Implementation

We have implemented a terminal based interface to demonstrate the working of the set data structure. We have implemented the following set functions in our program:

1)Insert
2)Delete
3)Search
4)Kth Largest/smallest
5)Union
6)Intersection
7)Set difference
8)Symmetric difference

We have also implemented various functions for red black trees such as left rotate,right rotate, the balance function that balances the red black tree based on the structure of the tree.We have also implemented a function that can be used to compare the height of this self balancing red black tree against a normal BST for the same sequence of input.

A brief description of the logic of the functions is given below:

1. Insert Into red black tree:

a)Perform Normal BST insertion. By default, the inserted node is always red.
b)If the inserted node is the root node, Recolor it to black.
c)if the parent node is red:

i)If the uncle node is red

Change the color of Uncle and parent to Black.
repeat b) and c) for grandparent

ii)If the uncle is black

There are 4 cases:

LL case: Swap the color of grandparent and parent, and right rotate grandparent

LR case: Swap the color of grandparent and x, LR rotate the grandparent.

RL case: Swap the color of grandparent and x, RL rotate the grandparent.

RR case: Swap the color of the grandparent and p, Right rotate the grandparent.

2. kth smallest and kth largest element:

We perform iterative in-order traversal of the BST and keep a count of the number of elements using a stack. Once the number of elements becomes k, the present element to be popped is returned as the answer.The same goes for kth largest element as well, where we perform the same operations, but in the opposite direction(right).Hence, this operation takes a time of O(k)

3. Union of two sets:

We use a third set in order to perform union of two sets. Both the sets are traversed and their elements are added to the third set. Due to the property of sets, duplicates are automatically handled, and we get the third set.

4. Intersection of two sets:

We have traversed through one of the two sets, and performed binary search for each element in the second tree. If the element is found, it is inserted into the answer set, that is, the intersection of the two sets.

5.Set difference and symmetric difference:

Set difference and symmetric difference is implemented using their mathematical relationship with the above two set operations.(Union and intersection)

6.Height difference between the Red-Black tree and a normal binary search tree:

We have maintained a normal binary search tree for the same sequence of insertions in the interactive client program. Using this regular BST and the underlying BST of the set, we calculate the height Difference between the two implementations.

7.Searching an element:

Since this is a BST, standard BST search has been implemented for the set data structure.

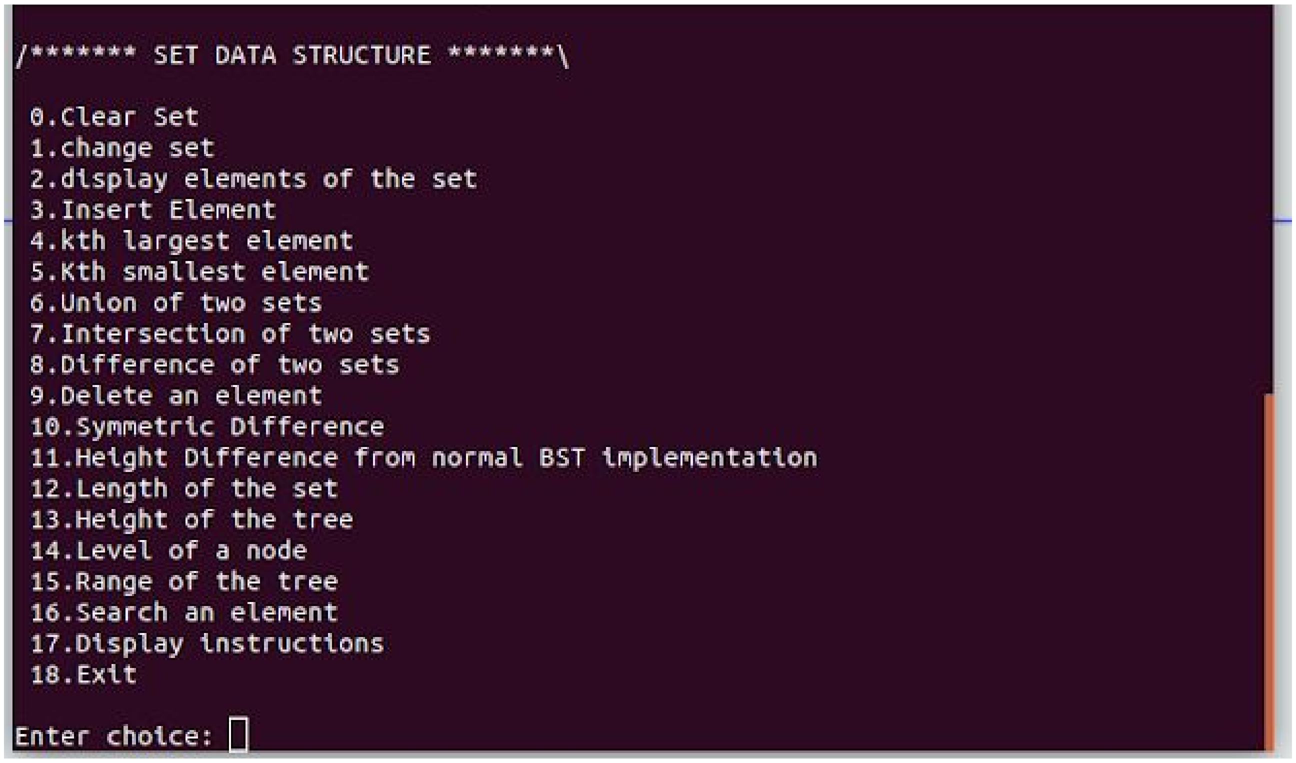
8.Height of the tree:

It is a recursive function that computes the height of the tree by traversing the entire tree.

9.Displaying the set:

We have implemented an in-order traversal of the tree. Using this order, e print the set in the appropriate set notation.

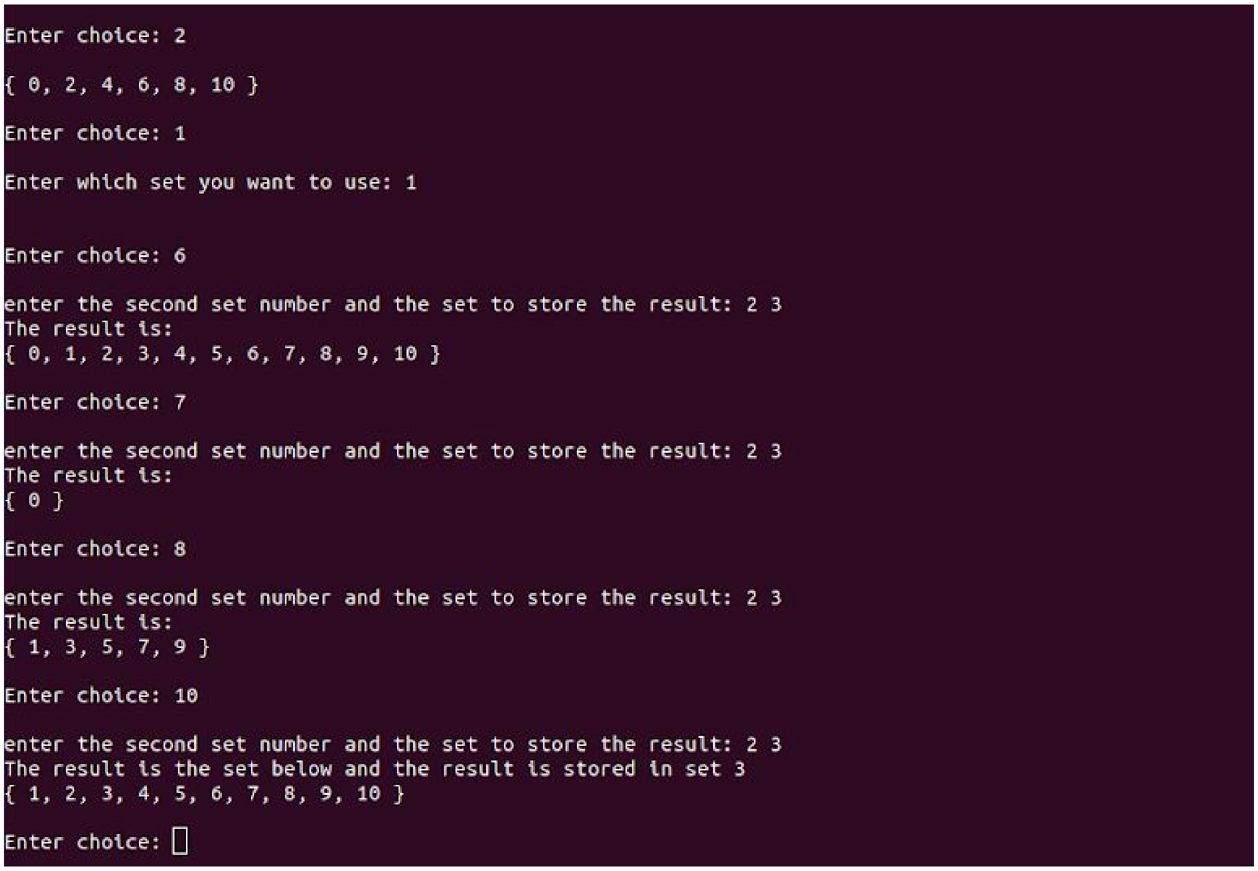
Input/Output



```
/******* SET DATA STRUCTURE *******\

 0.Clear Set
 1.change set
 2.display elements of the set
 3.Insert Element
 4.kth largest element
 5.Kth smallest element
 6.Union of two sets
 7.Intersection of two sets
 8.Difference of two sets
 9.Delete an element
 10.Symmetric Difference
 11.Height Difference from normal BST implementation
 12.Length of the set
 13.Height of the tree
 14.Level of a node
 15.Range of the tree
 16.Search an element
 17.Display instructions
 18.Exit

Enter choice: 
```

Fig 2: Terminal Interface of the program

The following actions were made using sets A - {0,1,3,5,7,9} and B - {0,2,4,6,8,10} to act as demo sets.



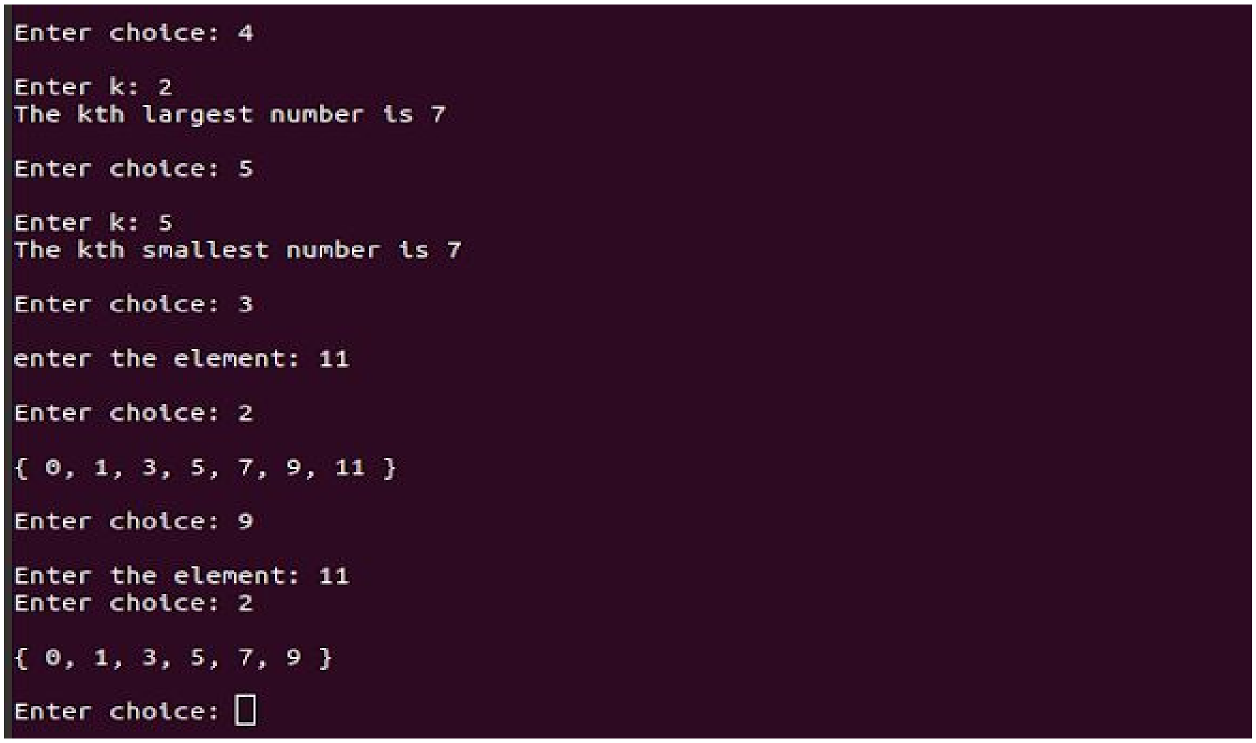
```
Enter choice: 2

{ 0, 2, 4, 6, 8, 10 }

Enter choice: 1

Enter which set you want to use: 1


Enter choice: 6

enter the second set number and the set to store the result: 2 3
The result is:
{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

Enter choice: 7

enter the second set number and the set to store the result: 2 3
The result is:
{ 0 }

Enter choice: 8

enter the second set number and the set to store the result: 2 3
The result is:
{ 1, 3, 5, 7, 9 }

Enter choice: 10

enter the second set number and the set to store the result: 2 3
The result is the set below and the result is stored in set 3
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

Enter choice: 
```

Fig 3: A demo of the mathematical functions incorporated in our program

Operation	Expected Result
Union	{1,2,3,4,5,6,7,8,9,10}
Intersection	{0}
Difference (A - B)	{1,3,7,5,9}
Symmetric Difference	{1,2,3,4,5,6,7,8,9,10}

Table 2: Expected results of the demo of mathematical functions (Fig 3)

We are using the sets A - {0,1,3,5,7,9} as the demo set.



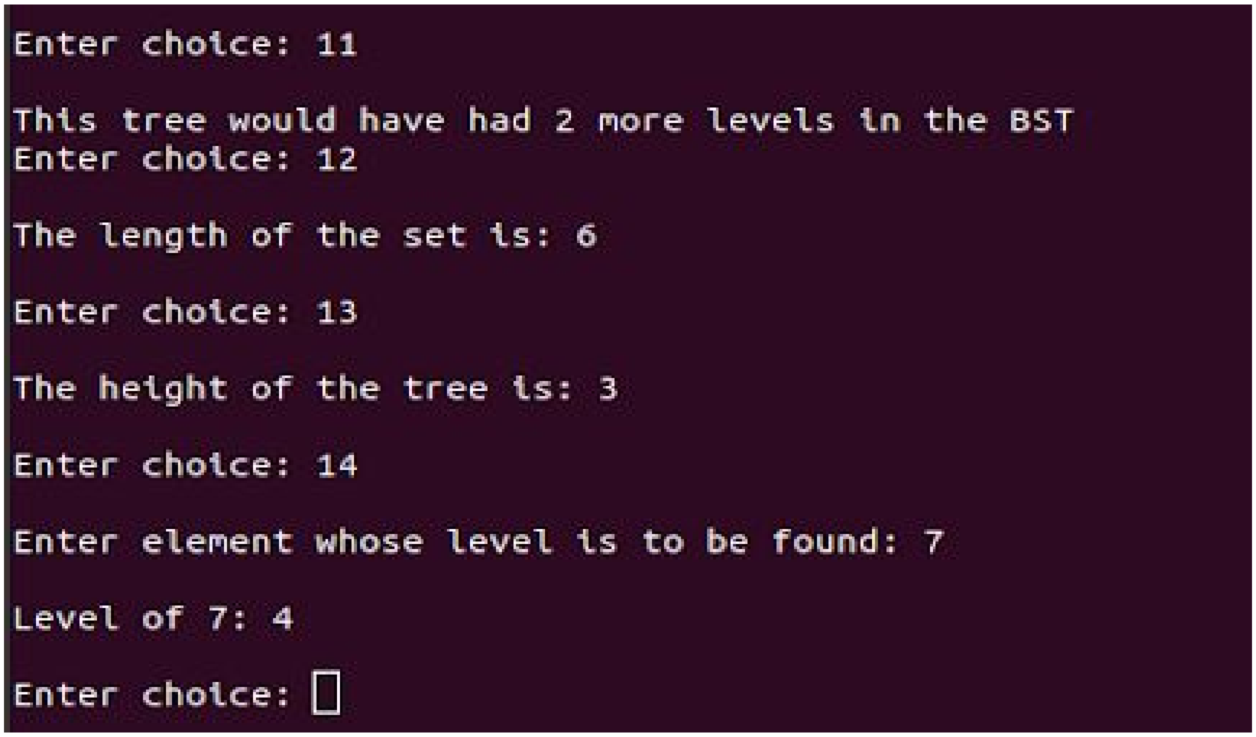
```
Enter choice: 4

Enter k: 2
The kth largest number is 7

Enter choice: 5

Enter k: 5
The kth smallest number is 7

Enter choice: 3

enter the element: 11

Enter choice: 2

{ 0, 1, 3, 5, 7, 9, 11 }

Enter choice: 9

Enter the element: 11
Enter choice: 2

{ 0, 1, 3, 5, 7, 9 }

Enter choice: 
```

Fig 4: A demo of the helper functions (1)

Operation	Expected Result
kth largest number (k = 2)	7
kth smallest number (k = 5)	7
Inserting element 11	{0,1,3,7,5,9,11}
Deleting element 11	{0,1,3,7,5,9}

Table 3: Expected results of the demo of helper functions(Fig 4)

We are using the sets A - {0,1,3,5,7,9} as the demo set.



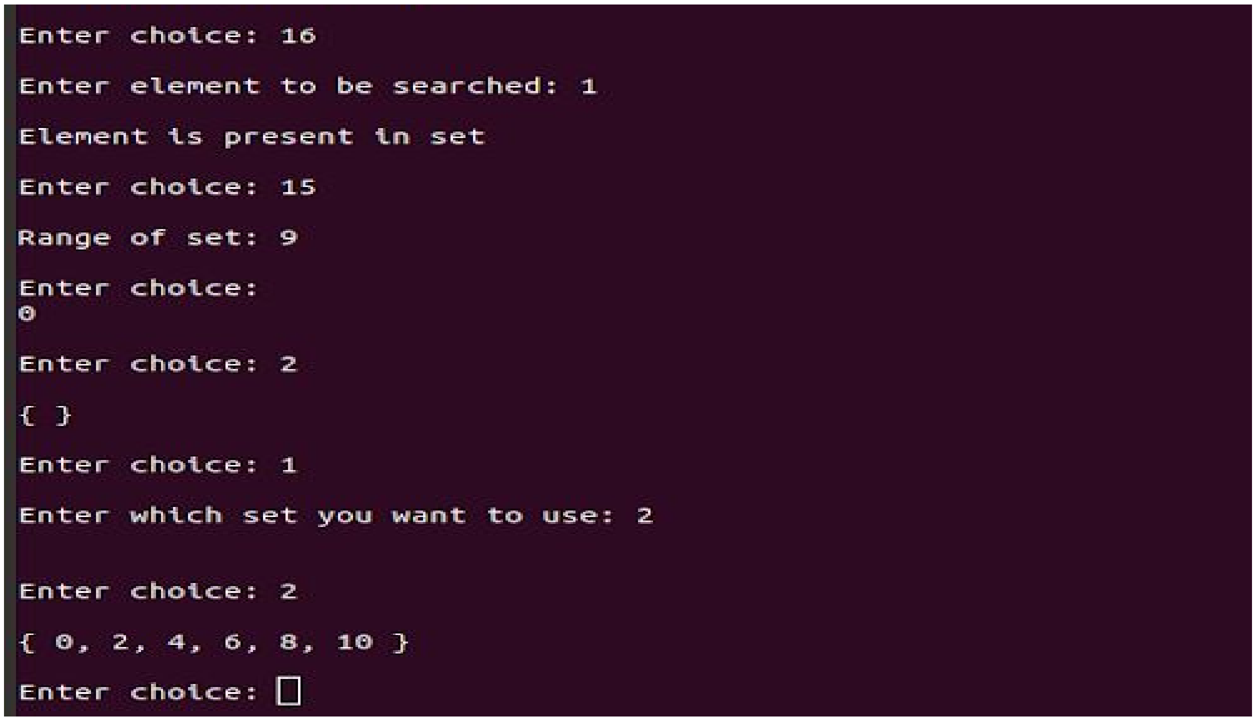
```
Enter choice: 11

This tree would have had 2 more levels in the BST
Enter choice: 12

The length of the set is: 6

Enter choice: 13

The height of the tree is: 3

Enter choice: 14

Enter element whose level is to be found: 7

Level of 7: 4

Enter choice: 
```

Fig 5: A demo of the helper functions (2)

Operation	Expected Result
Height difference from normal BST implementation	2
length of the set	6
Height of the tree	3
Level of a node (node - 7)	4

Table 4: Expected results of the demo of helper functions(Fig 5)

We are using the sets A - {0,1,3,5,7,9,11} and B - {0,2,4,6,8,10} as the demo set.



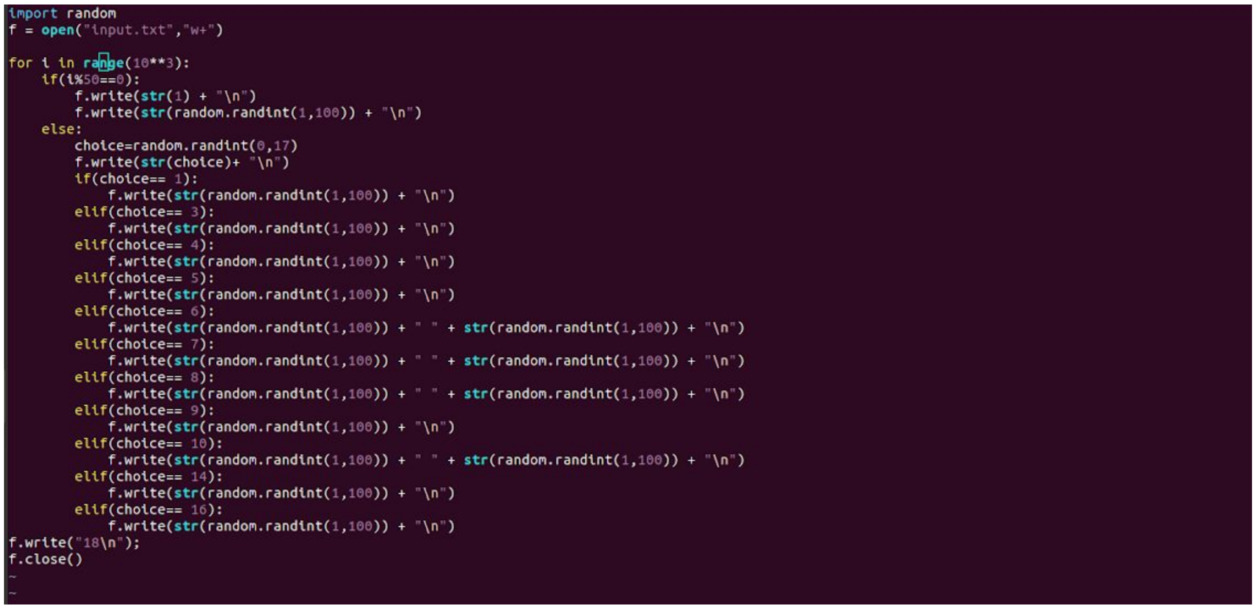
```
Enter choice: 16

Enter element to be searched: 1

Element is present in set

Enter choice: 15

Range of set: 9

Enter choice:
0

Enter choice: 2

{ }

Enter choice: 1

Enter which set you want to use: 2


Enter choice: 2

{ 0, 2, 4, 6, 8, 10 }

Enter choice: 
```

Fig 6: A demo of the miscellaneous functions

Operation	Expected Result
Search for element 1	Element is present in set
Range of the set	9
Clear the set	Resultant display shows a null set
Change set to set 2	Resultant display shows the set B = {0,2,4,6,8,10}

Table 5: Expected results of the demo of helper functions(Fig 6)

Crash Testing



```
import random
f = open("input.txt","w+")

for i in range(10**3):
    if(i%50==0):
        f.write(str(1) + "\n")
        f.write(str(random.randint(1,100)) + "\n")
    else:
        choice=random.randint(0,17)
        f.write(str(choice)+ "\n")
        if(choice== 1):
            f.write(str(random.randint(1,100)) + "\n")
        elif(choice== 3):
            f.write(str(random.randint(1,100)) + "\n")
        elif(choice== 4):
            f.write(str(random.randint(1,100)) + "\n")
        elif(choice== 5):
            f.write(str(random.randint(1,100)) + "\n")
        elif(choice== 6):
            f.write(str(random.randint(1,100)) + " " + str(random.randint(1,100)) + "\n")
        elif(choice== 7):
            f.write(str(random.randint(1,100)) + " " + str(random.randint(1,100)) + "\n")
        elif(choice== 8):
            f.write(str(random.randint(1,100)) + " " + str(random.randint(1,100)) + "\n")
        elif(choice== 9):
            f.write(str(random.randint(1,100)) + "\n")
        elif(choice== 10):
            f.write(str(random.randint(1,100)) + " " + str(random.randint(1,100)) + "\n")
        elif(choice== 14):
            f.write(str(random.randint(1,100)) + "\n")
        elif(choice== 16):
            f.write(str(random.randint(1,100)) + "\n")
f.write("18\n");
f.close()
~
~
```

- We also created a python program which randomises and assigns one of the 17 functions available to the program.
- We performed tests where we gave 10^7 function calls to our set program to check for crashes in the form of memory leaks and segmentation faults.
- This procedure was repeated 10 times and the program was checked for any faults, and the program ran successfully all the 10 times.

Conclusion

- We successfully implemented the data structure set and the underlying red black tree, along with many functions that might be necessary in order to use the data structure efficiently.
- A time complexity of O(logn) was achieved for the operations insert,search ,delete.kth smallest and kth largest element functions were implemented, which take O(k) time complexity.
- We have also compared the efficiency of the red black tree in terms of its self balancing property against a regular Binary search tree, and found out that the former can be very efficient in most cases as compared to regular BST.
- We have created a vast library of mathematical functions to be run on the set data structure along with other helper functions.
- The set data structure that we have implemented successfully mimics most of the functionalities that the C++ STL set data structure offers.

Acknowledgement

We would like to thank our Professor, Prof. Dinesh Singh for giving us this opportunity to do this project, and also for his invaluable inputs which helped us in completing this project successfully.

References

1. C++ STL set data structure
2. Red Black tree visualisation tool: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
3. Introduction to Design and Analysis of Algorithms (Anna University) 2014. by Anany Levitin.