COMP 6481 - Assignment 3 - [Part II] - IntelligentSIDC

Kaustav Mandal
Student Id: 40115265

Design Decision:

Since the keys can be maximum of 8 digits, the range of keys is from 00000001 to 99999999.
So below are the eight patterns(based on length of the key) and each key will follow one of those patterns.

1. 0000000X
2. 000000XX
3. 00000XXX
4. 0000XXXX
5. 000XXXXX
6. 00XXXXXX
7. 0XXXXXXX
8. XXXXXXXX

Where X is any digit from 1 to 9.
If there are two keys 6254 and 896547 then the first key follows the 4th Pattern and the second key follows the 6th Pattern.

Now, based one the first non zero digit (X) each pattern can be of 9 types. For example 896547 , 796547 they ~~vault~~ follow the 6th pattern from above but are different since the first digits are different. So, all total there could be ~~81~~ possibilities of keys if we consider the length and first digit of the key (Fig. 1)

From the above understanding if we consider there are 8 boxes and in each box there are 9 AVL trees, then any given key based on its length and first non zero digit; can be put in one AVL tree in one of those 8 boxes.

To illustrate the idea let's take three random keys having a maximum length of 8.
84569, **1**254568, **6**668.
Here the first key has length **5** and the first digit is **8**, so this key will be placed in the **8th** AVL tree of the **5th** Box. Similarly, key 1254568 will be placed in the first AVL tree of the 7th box. The key 6668 will be placed in the 6th AVL tree of the 4th box.

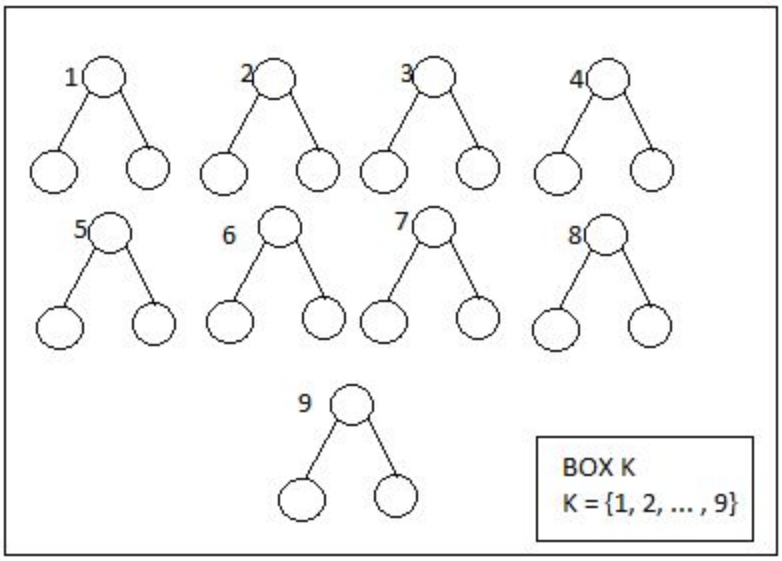

Fig. 1

Overall Architecture:

First one Class named as "OuterDatabase" has been created. Then another class called "InnerDatabase" has been created. OuterDatabase has one one array of InnerDatabase instances and here each InnerDatabase instance can be considered as one of those 8 boxes.

Each InnerDatabase has one array of 9 AVL tree instances.

```
public class OuterDatabase {

    private InnerDatabase [] innerdatabases = new InnerDatabase [8];

    public OuterDatabase()
    {
        for(int i =0; i<8; i++)
        {
            InnerDatabase innerdatabase = new InnerDatabase();
            innerdatabases[i] = innerdatabase;
        }
    }
```

OuterDatabase Structure

```
public class InnerDatabase
{
    final static Logger logger = Logger.getLogger(InnerDatabase.class);
    private AVLTree [] avltrees = new AVLTree[9];

    public InnerDatabase()
    {
        avltrees = new AVLTree[9];
        for(int i = 0; i<9; i++)
        {
            AVLTree tree = new AVLTree();
            avltrees[i] = tree;
        }
    }
```

InnerDatabase Structure

Methods Description:

1. add(int key)

   Since based on the length and first non zero key the key is getting inserted to the right AVL tree, the time complexity would be O(logn). Because to find out the right AVL tree takes constant time and insertion in AVL tree has time complexity of log(n).

2. remove(int key)

   Remove operation also has O(logn) complexity because finding out in which AVL tree the key is present takes constant time and removal from AVL tree operation has time complexity of O(logn).

3. getValue(int key)

   This operation also has time complexity of O(logn).

4. nextKey(int key)

   First, the AVL tree where the key is present is identified and all the nodes are stored in an array after traversing the AVL tree in In-inorder traversal . Inorder traversal of the nodes give us an array of nodes which are already sorted. Then binary search is performed to find the position of the key. Here two cases can take place.

   a. If the key is not the last element of the array then the next key will be the element present at the next position of the key.
   b. If the key is the last element then we need to jump to the next AVL tree in the same box. Here we don't do in order traversal rather there is one attribute in AVLTree class which is **lowestNode** which is returned. The reason behind this is, if we had done InOrder traversal then the first element would be the lowest element because InOrder traversal returns an array in ascending order. But it would cost an extra O(n) time complexity.
   c. Now if all the remaining AVL trees in the box are empty then we need to jump to the next box and repeat the same steps.

So, the time complexity is O(n) since we need to search all the keys in at least one tree.

5. prevKey(int **key**)

Here all the steps are similar to the nextKey(int key) method, however, there is one attribute called **highestNode** that is used if the previous key is not found in the AVL tree where the **key** is present. Here we need to go back to the previous AVL tree and if required in the previous boxes until we find **highestNode** in one of the AVL trees. Time complexity here is O(n)since we need to search all the keys in at least one AVL tree.

6. allKeys()

Since we need to traverse all the AVL trees, the time complexity would be O(n).

7. generate(int length)

Here we take the length from the user, then based on the length we go to the box and check the first AVL tree in that box. Then the **highestNode** is extracted and one new key with value one more than the key value of highest node is generated and which is of course not present in the tree or in the system. So, the time complexity is O(1).

8. rangeKeys(int key1, int key2)

Here allKeys() method has been reused. From the set, all those keys which are in the range between key1 and key2 are returned. So time complexity is O(n).

Drawback:

It may happen that all the keys can get stored into a single AVL tree, if the starting digit is the same for all the keys which have the same length. But then also search or insert operation will have complexity of O(logn) although n value will be much higher. However, it is not very likely to happen and in general if the keys are scattered this Hashing of AVL tree will perform very well.

Conclusion:

1. None of the above discussed methods exceed time complexity of O(n).
2. Also, this IntelligentSIDC performs equally well if the keys list is small also. So there is no need to design a separate system to store the keys for less amount of keys.