SPP Implementation Specification

Team Members: Dolton Fernandes (2018111007)
Tummala Shiva Prasad (2018101047)
S.shreevignesh (2018111019)
B Giri Prasath (2018111006)

Data Structure which will be used to implement the project: AVL Tree with some modifications

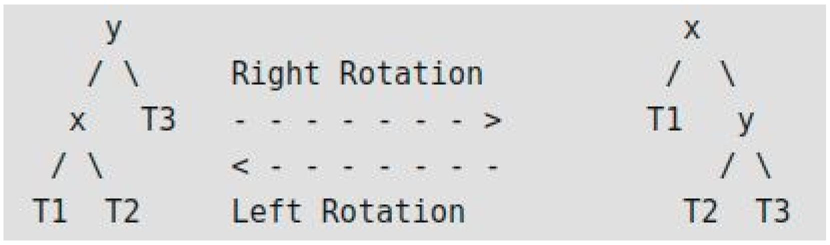
AVL Tree :

- AVL tree is a self-balancing Binary Search Tree where the difference between heights of left and right subtrees cannot be more than one for all nodes.

- Tree rotation is an operation that changes the structure without interfering with the order of the elements on an AVL tree. It moves one node up in the tree and one node down. It is used to change the shape of the tree, and to decrease its height by moving smaller subtrees down and larger subtrees up, resulting in improved performance of many tree operations. The direction of a rotation depends on the side which the tree nodes are shifted upon whilst others say that it depends on which child takes the root's place.

Insertion :

- To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).
    - Left Rotation
    - Right Rotation



```
     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - >        T1   y
  / \       < - - - - - - -            / \
 T1  T2     Left Rotation            T2  T3
```

Steps to follow for insertion :

Let the newly inserted node be w

1) Perform standard BST insert for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

1) y is left child of z and x is left child of y (Left Left Case)
2) y is left child of z and x is right child of y (Left Right Case)
3) y is right child of z and x is right child of y (Right Right Case)
4) y is right child of z and x is left child of y (Right Left Case)

1. Left Left case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

2. Left right case

```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

3. Right right case

```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

4. Right left case

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - - ->  / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                              /  \
T2   T3                           T3   T4
```

Deletion :

- To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

  1) Left Rotation
  2) Right Rotation

```
T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)
                y                               x
               / \     Right Rotation          /  \
              x   T3   – - – - – - – >        T1   y
             / \       < - - - - - - -            / \
            T1  T2     Left Rotation            T2  T3
Keys in both of the above trees follow the following order
      keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

Let w be the node to be deleted

1) Perform standard BST delete for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in

4 ways - Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case)
b) y is left child of z and x is right child of y (Left Right Case)
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)

Left left case:

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

Left right case:

```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

Right right case:

```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

Right left case:

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - - ->  / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                              /  \
T2   T3                           T3   T4
```

Finding kth element in the hash table :

In each node the count of nodes in its subtree will be stored.

At a particular node if the count of nodes in it’s left subtree is less than k then we move to the right node else to the left node.

If the count of nodes in left subtree + 1 is equal to k then the current node is the kth element to be found.

Structure of AVL :

The nodes of the tree will be pointers which will point to the left and right nodes respectively.

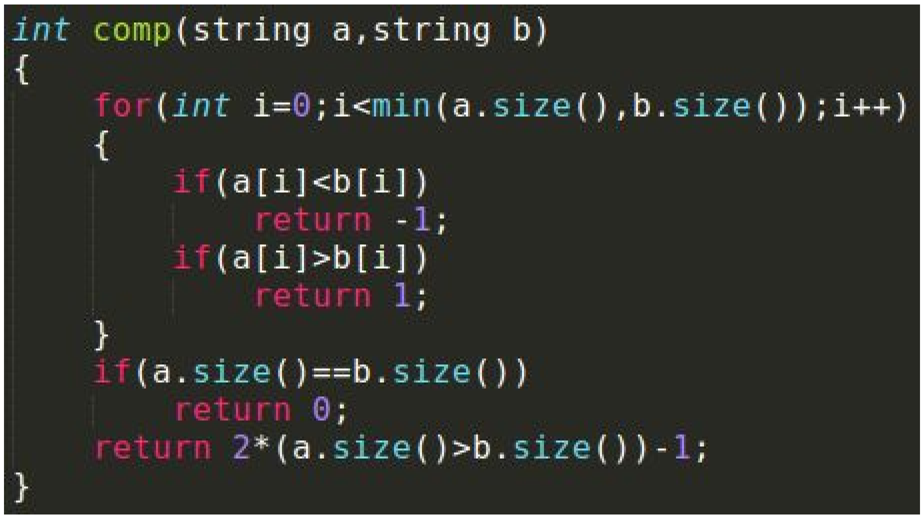
Each node will consist of pointer to left node , pointer to right node , count of nodes in subtree and Slice struct.

```
struct Slice{
    uint8_t size;
    char* data;
}
```

Comparing nodes in AVL operations :

We compare the string from first character to last character. The string with the first smaller character is the lexicographically smaller one.

Pseudocode:



```
int comp(string a,string b)
{
    for(int i=0;i<min(a.size(),b.size());i++)
    {
        if(a[i]<b[i])
            return -1;
        if(a[i]>b[i])
            return 1;
    }
    if(a.size()==b.size())
        return 0;
    return 2*(a.size()>b.size())-1;
}
```

This function returns 0 if s1 is lexicographically smaller than s1 , 1 otherwise.

Multithreading :

We can create threads when many get() requests are being made since each get() operation doesn’t affect the other get() requests. This will improve the time factor by reducing it.

For other operations like put(), del() we can’t do this since they depend on the other.
How ? put() and del() leads to addition, deletion in memory and rearrangement in the tree structure.

To synchronize all these events , we use mutex locks provided by the pthread library.

Modifications for better Time , Space complexity :

Instead of creating one single AVL Tree for all nodes , we can create some x (optimally chosen) AVL Trees and put the keys that satisfy a common property together.

The way of doing things can be changed to maximise cache efficiency.

Complexity Analysis:

Insertion , Deletion , Search takes worst case time complexity of O(64*log(n)) where n is the number of keys present.

The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of AVL insert remains the same as BST insert which is O(h) where h is the height of the tree. Since the AVL tree is balanced, the height is O(Logn). So the time complexity of AVL insert is O(Logn)!