Project Implementation Specification

Team Name : MASK
- **Akshat Goyal : 2018101075**
- **Kanish Anand : 2018101025**
- **Manish : 2018101073**
- **Swastik : 2018101022**

We tried various data structures for implementations :

1. Trie
2. Trie with BST
3. Red Black Tree

Best Option : Red Black Tree

Explanation of Various Methods :

Trie :

A trie, also called digital tree or prefix tree, is a kind of search tree —an ordered tree data structure used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Keys tend to be associated with leaves, though some inner nodes may correspond to keys of interest. Hence, keys are not necessarily associated with every node.

Benefits of Trie over Hash Tables :
The following are the main advantages of tries over hash tables:

- Tries support ordered iteration, whereas iteration over a hash table will result in a pseudorandom order given by the hash function (also, the order of hash collisions is implementation defined), which is usually meaningless.
- Tries facilitate longest-prefix matching, but hashing does not, as a consequence of the above. Performing such a “closest fit” find can, depending on implementation, be as quick as an exact find.
- Tries tend to be faster on average at insertion than hash tables because hash tables must rebuild their index when it becomes full – a very expensive operation. Tries therefore have much better bounded worst case time costs, which is important for latency sensitive programs.
- By avoiding the hash function, tries are generally faster than hash tables for small keys like integers and pointers.

Memory Requirement of Trie :
Using a simple prefix tree, the space requirement should be O(N*C) where C is the average number of characters per word and N is the number of words. This is because in the worst case, a Trie will store every character in each word. So a fair estimate would be 2.8 GB as we will store 10 million number of words and average number of characters is 64 plus memory for storing the value of the key is 256 per word so total would be 10million*64 + 10million*256 = 320*10million ~ 3*10^9 which sums to 2.8 G approx. Moreover, each location will need a lot of pointers to be stored, moreover, we will need additional integers to augment the trie nodes.Therefore, we are expected to reach upto 4GB of memory usage.

Analysis of Complexity of Operations of Trie :

- m : length of the string : 64
- s : total number of character : 52 [{a-z},{A-Z}]
- N : total number of Keys : 1e7

a. **get(key)** : returns value for the key
   i. The complexity is O(m) using trie as we traverse down the trie matching 1 letter at a time to the depth m.

b. **put(key,value)** : insert key,value in storage

i. The complexity is O(m) using trie as we need to traverse upto length m and go till the end of the word.

c. **delete(key)** : delete the key in exists
   i. The complexity is O(m) using trie as we traverse down the trie matching 1 letter at a time to the depth m and if Key exists, Value is deleted and Trie is traversed back decreasing the frequency of character by 1.
d. **get(n)** : returns value of the n th key sorted in lexicographic order.
   i. The complexity is O(s*m) using trie as we need to check atmax all s childrens per node and traverse down according to value of n till we get nth key value.

e. **delete(n)** : delete Nth key-value pair
   i. The complexity is O(s*m) using Trie as we traverse down to depth m and at each depth we look for the frequencies of s characters to get Nth key.

Problems faced with Trie :

We tried to run code for 1e7 inputs with trie but code did not work on laptop because of large memory requirements of the trie. Due to large differences in memory requirements as compared to expected value we had to switch from trie to some kind of balanced tree.

Time & Memory :

Time for 1e5 inputs : 0.3 sec

Memory for 1e5 inputs : 3 %

Trie with BST:

In the previous implementation we were storing 52 pointers for each node which took a lot of memory as well as search operation was very costly due to this. So in this approach we stored pointers to those children which really existed. This helped us to decrease memory requirements of the trie. Also searching became much faster. So this code worked on laptop for 1e6 inputs and time recorded were :

Analysis of Complexity of Operations of Trie with BST:

- m : length of the string : 64
- s : total number of character : 52 [{a-z},{A-Z}]
- N : total number of Keys : 1e7

f. **get(key)** : returns value for the key
   i. The complexity is O(m) using trie as we traverse down the trie matching 1 letter at a time to the depth m.

g. **put(key,value)** : insert key,value in storage
   i. The complexity is O(m) using trie as we need to traverse upto length m and go till the end of the word.

h. **delete(key)** : delete the key in exists
   i. The complexity is O(m) using trie as we traverse down the trie matching 1 letter at a time to the depth m and if Key exists, Value is deleted and Trie is traversed back decreasing the frequency of character by 1.

i. **get(n)** : returns value of the n th key sorted in lexicographic order.
   i. The complexity is O(s*m) using trie as we need to check atmax all s childrens per node and traverse down according to value of n till we get nth key value.
   ii. : key-value pair

iii. The complexity is O(s*m) using Trie as we traverse down to depth m and at each depth we look for the frequencies of s characters to get Nth key.

Problems faced with Trie with BST:
We tried to run code for 1e7 inputs with trie but code did not work on laptop because of large memory requirements of the trie. Code worked for 1e6 inputs but did not run for 1e7 inputs. Whereas trie was not able to run with both 1e6 and 1e7 inputs. Also time for 1e6 inputs was not too good. So next we tried Red Black Tree.

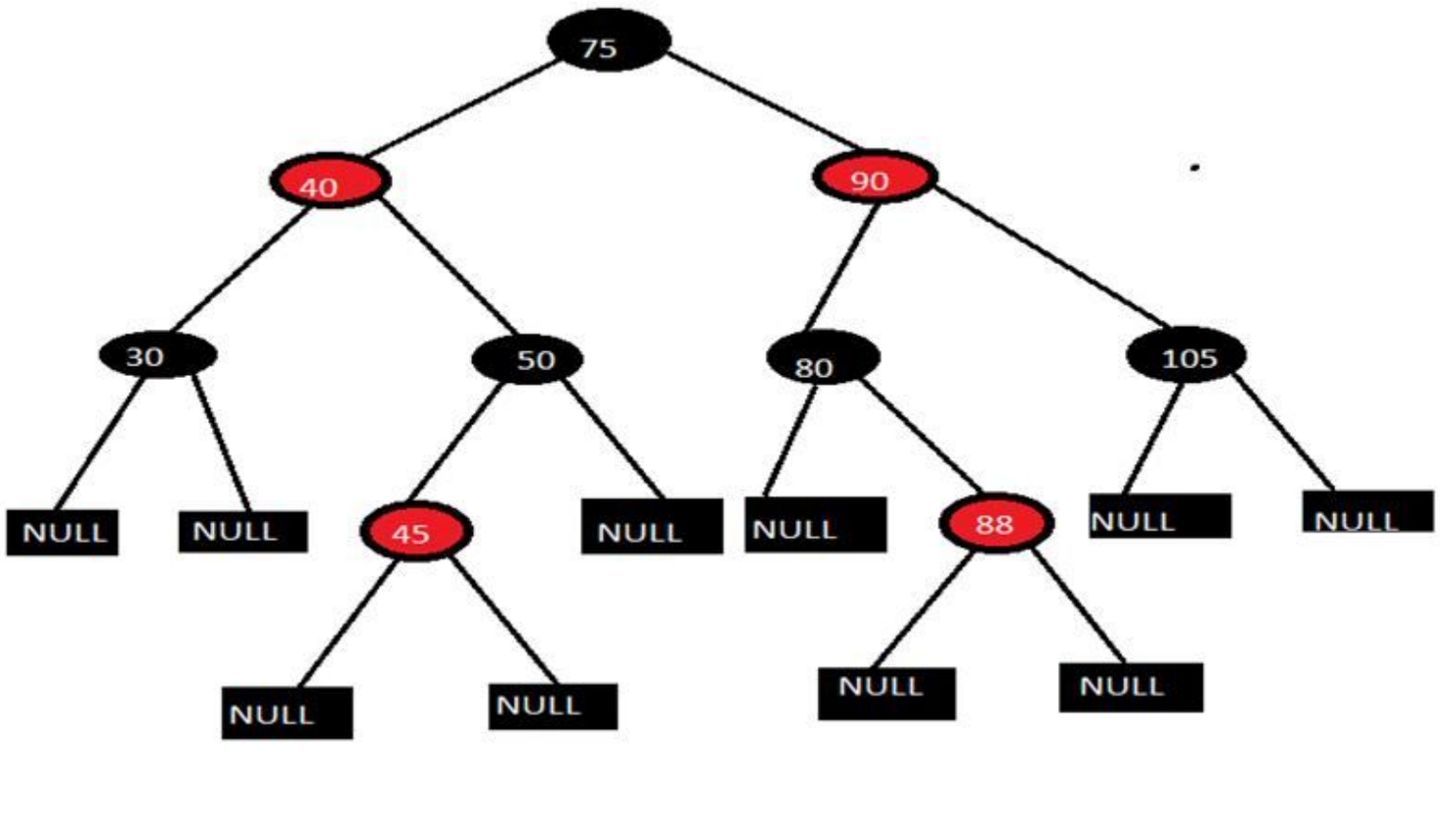
Time & Memory :

Time for 1e6 inputs : 4 sec
Memory for 1e6 inputs : 40 %

Red Black Tree :

A red–black tree is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Balance is preserved by painting each node of the tree with one of two colors in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.



Memory Requirement of Red Black Tree:
So in Red Black Tree we are storing nodes with key and value. And each node has at most 2 children so pointers to these children are stored.

Analysis of Complexity of Operations of Red Black Tree:

- m : length of the string : 64
- s : total number of character : 52 [{a-z},{A-Z}]
- N : total number of Keys : 1e7

j. **get(key)** : returns value for the key

i. The complexity is O(logn) using RBT as it is a balanced binary tree.

k. **put(key,value)** : insert key,value in storage
   i. The complexity is O(logn) using RBT as it is a balanced binary tree.

l. **delete(key)** : delete the key in exists
   i. The complexity is O(logn) using RBT as it is a balanced binary tree.

m. **get(n)** : returns value of the n th key sorted in lexicographic order.
   i. The complexity is O(logn) using RBT as it is a balanced binary tree. With each node we also store the number of descendants. So we binary search for n in the tree.

n. **delete(n)** : delete Nth key-value pair
   i. The complexity is O(logn) using RBT as it is a balanced binary tree. With each node we also store the number of descendants. So we binary search for n in the tree.

Time & Memory :

Time for 1e6 inputs : 0.11 sec
Memory for 1e6 inputs : 7 %
Time for 1e7 inputs : 22 sec
Memory for 1e7 inputs : 71 %

ADVANTAGES of RED BLACK TREE :

1. Memory Consumption is very less as compared to trie because there are only two children per node so there are no unused pointers.
2. Because it is a balanced binary tree, the height of the tree is O(logn).
3. They have very low constants as compared to other balanced trees.