Project Implementation Specification

Team Name : AK
- **Akshat Goyal : 2018101075**
- **Kanish Anand : 2018101025**

Various Alternatives :

1. **BST , B-Tree, Red-Black Tree, AVL**
2. **Trie**
3. **Hash Table + Trees**

Best Option : 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 BST :

The following are the main advantages of Tries over Binary search trees (BSTs):

- Looking up keys is faster. Looking up a key of length m takes worst case O(m) time. A BST performs O(log(n)) comparisons of keys, where n is the number of elements in the tree, because lookups depend on the depth of the tree, which is logarithmic in the number of keys if the tree is balanced. Hence in the worst case, a BST takes O(m log n) time. Moreover, in the worst case log(n) will approach m. Also, the simple operations tries use during lookup, such as array indexing using a character, are fast on real machines.

- **Tries** are more **space efficient** when they contain a large number of short keys, because nodes are shared between keys with common initial subsequences.
- Tries facilitate longest-prefix matching, helping to find the key sharing the longest possible prefix of characters all unique.
- The number of internal nodes from root to leaf equals the length of the key. Balancing the tree is therefore no concern.

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.

Implementation of cache:
Idea:
We can cache the most frequently accessed prefixes of keys, and store direct pointers to them in a cache pool. The prefixes can be of length upto 5 or 6, and this will probably save us 5 to 6 iterations of having to traverse the trie.
Implementation:
We will create an organized cache pool of size 52^4 times 8 for prefixes of size 4. Then, it's only a matter of some simple bit shifts and in one line of C code we can get to the concerned pointer.

Multithreading ideas:
We intend to keep a mutex to avoid segmentation fault, eg if there are two simultaneous queries one for get and second for delete for the same key then it may happen that get thread might use the pointer which is already freed by the delete thread leading to segmentation fault.
We plan on implementing a **reader-writer like mechanism** - multiple threads may be performing `get` operations on a store but only one thread may be performing a `put` operation at any given instant. If there are multiple gets, and a put is requested, the put (and any other subsequently arriving operations) is added to a queue until the gets complete.

Second Option : Balanced Tree
A balanced tree is a tree where every leaf is “not more than a certain distance” away from the root than any other leaf. The various balancing schemes give actual definitions for “not more than a certain distance” and require different efforts to keep the trees balanced:
• AVL trees
• Red-black trees.
Inserting into, and deleting from, a balanced binary search tree involves transforming the tree if its balancing property — which is to be kept invariant — is violated. These re-balancing transformations should also take O(log2n) time, so that the effort is worth it. These transformations are built from operators that are independent from the balancing scheme.

Analysis of Complexity of Operations of Balanced Tree:

- 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(log(N) * m) as the tree is balanced so max height is O(log(N)) and at each level string of length m is compared.

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

i. The complexity is O(log(N) * m) as the tree is balanced so max height is O(log(N)) and at each level string of length m is compared.

c. **delete(key)** : delete the key in exists

i. The complexity is O(log(N) * m) as the tree is balanced so max height is O(log(N)) and at each level string of length m is compared and if key exists node is deleted and tree is balanced again.

d. **get(n)** : returns value of the n th key sorted in lexicographic order.

i. The complexity is O(log(N) * m) as the tree is balanced so max height is O(log(N)) and at each level string of length m is compared. We will store the number of subtrees of each node in it and will use this to reach till the nth key. If the value of n is less than the number of subtrees of the right child then we will traverse to the left, otherwise we will traverse to the right path.

e. **delete(n)** : delete nth key-value pair

i. The complexity is O(log(N) * m) as the tree is balanced so max height is O(log(N)) and at each level string of length m is compared. We will store the number of subtrees of each node in it and will use this to reach till the nth key. If the value of n is less than the number of subtrees of the right child then we will traverse to the left, otherwise we will traverse to the right path.

Source :

https://thenextcode.wordpress.com/2015/04/12/trie-vs-bst-vs-hashtable/

http://user.it.uu.se/~justin/Teaching/PK2/Slides/AVLtrees.pdf

https://en.wikipedia.org/wiki/Trie