Software Programming for Performance

In-Memory Key Value storage

Peppermint

- *Yash Bhansali*(2018101068)
- *Arohi Srivastav*(2018111018)
- *Shikhar Garg*(2018101101)
- *M Sridhar*(2018111021)

OverView

The project requires us to build an In-Memory Key-Value Storage data structure in C++.

A data structure needs to be built which stores these Key-Value pairs and facilitates the following operations:

1. **get(key): Return value for the key**
2. **put(key, value): Insert key-value, overwrite value if it already exists.**
3. **delete(key) : Delete the key-value from the data structure.**
4. **get(int N): Return the Nth key-value pair**
5. **delete(int N): Delete the Nth key-value pair from the data structure.**

The project demands these operations on the data structure to be fast, efficient and consuming less memory and CPU while having a high transaction rate.

Ideas We Thought

- ***Tries***- Is time efficient on all operations.We need to store additional information about the number of keys starting with each letter in all the nodes.The only concern is high memory usage. If at some later time we can optimize on memory we may try out this data structure.

- ***Fst***- Current implementations of FST are not good for alpha-numeric keys.Also does not support changing the key value pair at a later time.

- ***Hybrid data structure (Hashtable for storing key - value pairs and balanced tree for ordering)*** - This is what we decided for after looking at all other structures.

Chosen Data Structure - Hybrid

Details

First Implementation :

- Use a **hash table** for storing key-value pair.Helps us in storing and getting key value pairs in constant time. Put(key,value) , get (key) and delete (key) can be done in a constant amount of time with overhead of book-keeping on the balanced tree.
- Use **Red-Black tree** for ordering the keys.Will be able to do get(N),delete(N) in logarithmic time.We will look up the key using this tree and delete it from the hash table.

Second Implementation :

- Follow a two level hashing. The keys are first hashed by their length which helps in memory and time optimization.So we have different hash tables for keys of different lengths with the keys further hashed based on their first character.Now each row of this hash table has a corresponding red black tree with the further data stored in it.

Implementation Details -

- We tried with tries first as the time complexity was better for tries. But the memory used by tries was very high.We were getting memory errors for inputs of the order 100000.We tried to optimize on memory but it increased the time overhead and even after optimization memory usage was very high.
- We then moved to our other implementation of two-level hashing.We figured out if we hash with respect to length, lexicographical ordering of the keys will be an issue.So we did two level hashing with respect to the first two letters of the keys. The key value pair was further stored in a red black tree.For the implementation of the red black tree we looked at this website.
- We compared the time taken by two-level hashing and one level hashing and figured out two level hashing is better.
- For now we are doing get(n) query in order of n using inorder traversal.We will look into enhancing the red black tree structure such that we are able to do this in order of log(n). We will look into further optimizations in the later stages of this project.

Final Phase

In the final implementation, these are the time complexities of various operations where n is the number of key-value pairs.

Operations	*Time Complexity*
1. Put	O(log(n))
2. Get(key)	O(log(n))
3.Delete(key)	O(log(n))
4.Get(n)	O(log(n))
5.Delete(n)	O(log(n))

Why Red Black Tree

- Every node which needs to be inserted should be marked as red.
- Not every insertion causes imbalance but if imbalance occurs then it can be removed, depending upon the configuration of the tree before the new insertion is made.
- In Red black tree if imbalance occurs then for removing it two methods are used that are: **1) Recoloring** and **2) Rotation**
- Red Black Trees provide faster insertion and removal operations than AVL trees as fewer rotations are done due to relatively relaxed balancing.
- AVL trees store balance factors or heights with each node, thus requires storage for an integer per node whereas Red Black Tree requires only 1 bit of information per node.

Thank YOU.