SOFTWARE PROGRAMMING FOR PERFORMANCE

ALPHIES and QWERTYUIOP

Anishka Sachdeva (2018101112)
Astitva Gupta (2018101085)
Satyam Viksit Pansari (2018101088)
Trusha Sakharkar (2018101093)

Project Problem Statement : Build a In-Memory Key-Value Storage Software in C++

Data Structure decided in the implementation spec: **TRIES**

New data structure used for the project: **RED BLACK TREE**

Reason for change of data structure:

- Tries give the best runtime for each API call.
- We started implementing tries but faced **memory constraints** which consisted of many segfaults for large numbers of entries.
- Hence we switched to a balanced tree named Red Black Tree.
- Also we thought of using AVL trees but insertions and deletions are very heavy due to many rotations, whereas get requests are very fast compared to Red Black Trees.
- Red Black Tree promises better performance with frequent insertion and deletion queries, as we wanted to concentrate on making an optimized implementation based on techniques learned in this course rather than algorithmic optimization.

WHY RED BLACK TREE:

The options that we looked at for implementing the dictionary were Tries,Red-Black Trees,BST, AVL Trees, Compressed Tries and hash + tree Implementation. We left Tries for reasons of high memory usage. When comparing AVL Trees and Red-Black Trees, we found that AVL trees are more strictly

balanced so although searching is faster due to less tree height, insertion and deletion operations are much slower. We have noticed that the Red-Black Tree takes slightly more space while storing the dictionary as compared to AVL tree, however, we notice that the performance of Red-Black Tree is better as compared to the AVL tree and the difference becomes highly visible as the size of the dictionary increases.

These were the reasons for choosing red black tree over other trees :

- 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
    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.

COMPARISONS WITH OTHER DATA STRUCTURES:

	Insertion	Searching	Deletion
Red Black Tree	O(ln n)	O(ln n)	O(ln n)
B Tree	O(ln n)	O(ln n)	O(ln n)
AVL Tree	O(ln n)	O(ln n)	O(ln n)
Tries	O(mn)	O(mn)	O(mn)

1. **About Red Black Trees**

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

A red-black tree satisfies the following properties:

1. **Red/Black Property**: Every node is colored, either red or black.
2. **Root Property:** The root is black.
3. **Leaf Property:** Every leaf (NIL) is black.
4. **Red Property:** If a red node has children then, the children are always black.
5. **Depth Property:** For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).
6. The path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf.

2. Initial 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.

3. <u>Final Implementation details of the APIs:</u>

Let n = number of entries(key-value pairs).

- **get (key) - Returns value for the key**
  Implementation:

  Worst Case Complexity = O(log(n))
  The get function takes the key-value pair as input and checks for its existence in the tree. Returns true if the key-value pair is found.

- **put (key, value) - Add key-value, overwrite existing value**
  Implementation:

  Worst Case Complexity : O(log(n))
  The put function adds a key-value pair to the tree and rebalances it using the function for fixing consecutive red nodes to help retain the basic properties of the red-black tree.

- **delete (key) - Deletes a key along with its value**
  Implementation:

  Worst Case Complexity = O(log(n))
  The above function checks for the existence of the given key and then removes that key-value pair from the tree. The function is followed by a function that rebalances the tree and retains the properties of the red-black tree.

- **get(int N) - Returns Nth key-value pair**
  Implementation:

  Worst Case Complexity = O(log(n))

The above function finds the existence and then returns the Nth key-value pair observed in the tree when all key-value pairs are stored lexicographically in the tree.

- **delete(int N) - Delete Nth key-value pair**
  Implementation:

  Worst Case Complexity = O(log(n))
  The delete function deletes the Nth key-value pair from the tree when the entire data in the tree is arranged in lexicographical order.

The insert is done checking for the double red errors while the comparatively complex delete operation balances the tree by keeping the depth property true always. The search in RB tree is easy as we have already stored the sum of keys in the left child as well or the right child in each node.We check these values before proceeding.

Optimizations:

- We plan to introduce pointer references in the code to make it run even faster.

Memory Allocations:

- We are dynamically adding memory when required using **new()** and dellocating or freeing it using **free()** in c++.

Locks:

- We have used mutex locks to take care of proper synchronization and avoiding errors while using multiple threads.

References:

- Geeks for Geeks

- www.programiz.com/dsa/red-black-tree
- A Comparative Study on AVL and Red-Black Trees Algorithm