Hash Map Vs Red Black Tree

Abstract

This report compares the Hash Map and Red-Black Trees data structures, two widely used computer science techniques for organising and accessing data. The study explores the strengths and weaknesses of each data structure in terms of key performance metrics such as Insertion, Searching, and Deletion. Through experiments and analyses, we evaluate the impact of various operations on the performance of Hash Maps and Red-Black Trees.

Hash Map

A hash table is a data structure that stores and retrieves values using a key. It provides constant-time average-case complexity for basic operations like insertions, deletions, and searches under certain conditions.

To initialise the hash table, we need two things: 1. An initial size, i.e., the initial length of the hash table and the load factor.

**Initial Size**: An integer value signifying the initial size of the hash table. Depending on the use case, this value can be any integer; the smaller the size, the more numbers or resize operations are performed, and vice versa.

**Load factor**: a floating-point value used as a threshold to resize the hash table, i.e., when the size of the hash table reaches this threshold, then we increase the length (in the current implementation, we double the size) of the hash table.

The current implementation has used open addressing with quadratic probing for conflict resolution. A conflict occurs when some other value already occupies the hash value position. So, we need to find the next available position. The quadratic probing method of collision resolution searches for the following vacant space by taking steps in the order of *i2* where *i = 1, 2, 3...* So, the table is traversed(probed) in the order *h(k)+1*, *h(k)+4*, *h(k)+9*, *h(k)+16* and so on. Since the step size keeps increasing gradually, the elements are more widely distributed in the table, leading to less clustering.

Red Black Trees

A Red-Black Tree is a self-balancing binary search tree data structure that maintains balance during insertions and deletions. It ensures that the depth of the tree remains logarithmic, preventing performance degradation commonly associated with degenerate cases of unbalanced binary search trees.

Critical characteristics of Red-Black Trees:

1. **Node Colour:**
   * Each node in the tree is assigned one of two colours: red or black.
2. **Balancing Rules:**
   * The tree adheres to specific rules to maintain balance:
     + **Rule 1:** Every node is coloured, either red or black.
     + **Rule 2:** The root is always black.
     + **Rule 3:** All leaves (null or NIL nodes) are considered black.
     + **Rule 4:** If a red node has children, both children are black.
     + **Rule 5:** Every path from a node to its descendant leaves contains the same number of black nodes.
3. **Balancing Operations:**
   * During insertions and deletions, Red-Black Trees employ balancing operations to ensure adherence to the balancing rules.
     + **Recolouring:** Changing the colour of nodes to maintain balance.
     + **Rotation:** Performing left or right rotations to preserve the binary search tree property.
4. **Advantages:**
   * Red-Black Trees guarantee logarithmic height, leading to efficient search, insertion, and deletion operations.
   * They are suitable for scenarios where a balanced tree is essential, such as implementing associative containers like sets and maps.
5. **Complexity:**
   * The time complexity of search, insertion, and deletion operations in a Red-Black Tree is O(log n), where n is the number of nodes in the tree.

Red-Black Trees maintain a relatively balanced structure and the overhead of balancing operations.

Results

In this iteration, the initial size of the hash table was 1, and the load factor was 0.7

1. Time is taken in seconds to create. It took 14 resize operations to insert all the elements completely

{'hash': 0.011687755584716797, 'rbt': 0.013743877410888672}

The hash table was faster by 0.002056121826171875 seconds.

1. Time taken to insert five new elements

{'hash': 6.9141387939453125e-06, 'rbt': 1.1682510375976562e-05}

The hash table was faster by 4.76837158203125e-06 seconds.

1. Time taken to search for five elements

{'hash': 7.486343383789062e-05, 'rbt': 7.510185241699219e-05}

The hash table was faster by 2.384185791015625e-07 seconds.

1. Time taken to delete five elements

{'hash': 5.4836273193359375e-06, 'rbt': 1.5735626220703125e-05}

The hash table was faster by 1.0251998901367188e-05 seconds.

In this iteration, the initial size of the hash table was 1000, and the load factor was 0.7

1. Time is taken in seconds to create. It took four resize operations to insert all the elements completely

{'hash': 0.010606765747070312, 'rbt': 0.013679265975952148}

The hash table was faster by 0.003072500228881836 seconds

1. Time taken to insert five new elements

{'hash': 5.9604644775390625e-06, 'rbt': 1.4066696166992188e-05}

The hash table was faster by 8.106231689453125e-06 seconds.

1. Time taken to search for five elements

{'hash': 3.933906555175781e-05, 'rbt': 8.273124694824219e-05}

The hash table was faster by 4.3392181396484375e-05 seconds.

1. Time taken to delete five elements

{'hash': 5.9604644775390625e-06, 'rbt': 1.5974044799804688e-05}

The hash table was faster by 1.0013580322265625e-05 seconds.

Conclusion

So, from the above results, we can conclude that the hash table/map is better regarding the time complexity or the total time taken for different operations like Insertion, Searching and Deletion.

For the Hash table, we can further reduce the number of resize operations and collisions by increasing the initial size of the hash table.