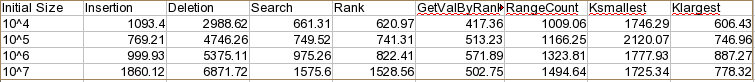
Gray Houston

ghousto

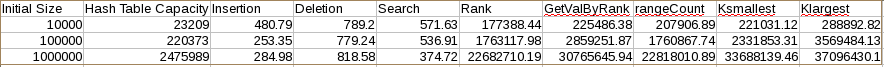
0026483532

Project 3 Analysis/Report

Report 1: RB Tree Performance

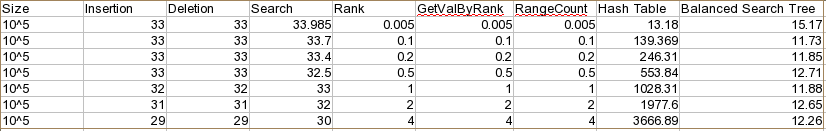


Report 2: Linear Probing Hash Table Performance



Report 3: Hashing vs. BST

* functions are represented in percentage of total calls
* data structure performance averaged where n = 10



The table above shows that when there are fewer calls to Rank, GetValByRank, and RangeCount, the Hash Table's performance will improve will the BST's will become worse. However, the Hash Table only actually surpasses the BST in speed when the rank-related functions are practically never called. As shown in the table, the smallest percentage of calls to non rank-related functions (e.g. Insertion, Deletion) where the Hash Table outperforms the BST is 99.985% . If the percentage becomes lower than this, the BST will outperform the Hash Table.

Therefore, we can infer that if there are no calls made to the rank-related functions, then the Hash Table will outperform the BST. However, if there are any calls to a rank-related function, the BST will perform better. The data structures' relative performance for the rank-related function shown in tables 1 and 2 provide evidence for this theory.

Furthermore, the data does not suggest that there is a noticeable correlation between the distribution of Insert, Delete, and Insert function calls and the performance of either the Hash Table or the BST. The only apparent determining factor is the total percentage of the first three functions to the total percentage of the rank-related functions (when the rank-related percentage is non-trivial). Both the Hash Table and the BST appear to perform worse as the percentage of rank-related functions increases. However, the effect is significantly stronger on the Hash Table than the BST

Analysis Questions

1. For the RB Tree, the worst case performance of both the rank function and the getValByRank function is O(log *n*) , assuming that the Size function runs in O(1).

For Rank, the worst case scenario is that the specified key is not in the tree or that the key is a leaf. This is the worst case because hitting the specified key is a base case that ends the recursive call. Thereby, when Rank ends because it hit the specified key, there may still be child nodes that can be traversed. So, if the key isn't in the tree or if the key is a leaf node, Rank will recurse down the tree until it hits a leaf.

In the worst case for Rank, the function will execute like a binary search as it moves down the tree (semantically, it's searching for the specified key). By using the Size function, no more than O(log *n*) nodes will be traversed. Therefore, assuming that Size runs in O(1) time, Rank will run in O(log *n*) time in the worst case.

For GetValByRank, the worst case is when the specified rank k = n / 2 + 2 . This is the case where the specified rank k is 1 greater than the number of elements in the left sub-tree and the root. In this scenario, the function essentially binary searches for the minimum element in the right sub-tree. Therefore, because the function stops 1 level above the leaf, the worst case running time for GetValByRank is O(log *n* – 1), assuming that Size runs in O(1) time. This running time is asymptotically equal to O(log *n*). Therefore, the worst case running time is O(log *n*).

For both of these functions, because they traverse the tree similarly to a binary search, their worst case performance will be O(log *n*).

2. For Hash Tables, in both Rank and GetValByRank functions, the worst case running time is O(*h*) where *h* is the length of the Hash Table.

The worst case for Rank is that the key is the largest element in the list or larger. In this case, the Hash Table will have to traverse over every node. However, because nodes are spread out non uniformly across the table in clusters, traversing each element will require the Hash Table to traverse every single possible index in the table. Therefore, the worst case running time is O(*h*).

The worst case for GetValByRank is that the specified rank is greater than or equal to the number of nodes in the table. For the same reasons stated in the preceding paragraph, in this worst case, GetValByRank will iterate over every single index in the table. Therefore, it's worst case running time is O(*h*).

3. In a RB Tree, the worst case performance for RangeCount is proportional to O(log *n*) while the worst case for Ksmallest and Klargest is O(*k* log *n*).

For RangeCount, my implementation uses the Rank and Contains functions to compute the specified range. Because there are 2 calls to Rank, and 1 call to Contains, the worst case running time for this function is O(3log *n*). This running time is proportional to O(log *n*). Therefore, the worst case running time for RangeCount is O(log *n*).

Both Ksmallest and KLargest work by essentially binary searching the node that has a rank of *k* + 1 where *k* is the number of smallest/largest elements requested. This binary search will run in O(log ­*n*) in the worst case. Furthermore, the *k* smallest/largest elements are added to an ArrayList by using an in-order traversal. In the worst case, traversing over all of the *k* elements will run in O(*k*) time. Therefore, in the worst case, Ksmallest and Klargest will run in O(*k* + log *n*) time.

4. For the Hash Table, the worst case performance for RangeCount is O(*h*) while the worst case performances for Klargest and Ksmallest are O(*h* + *n* log *k*) where *h* is the size of the hash table, *n* is the number of elements in the hash table, and *k* is the number of smallest/largest elements requested in the function call (e.g. when getting the 5 smallest elements, *k* is 5).

For RangeCount, my implementation, assuming a valid range, essentially loops through the entire hash table, making comparisons. The function loops through all *h* locations in the hash table and checks to see if there is an element at that location, if it is greater than or equal to the lower bound, and if it is less than or equal to the upper bound. If all 3 of these conditions are met, the count is incremented.

Assuming that the hash table would resize if there were *h*/2 + 1 elements, the worst case for RangeCount would be if there are *h*/2 elements that are greater than or equal to the lower bound. In this scenario, the null check would occur O(*h*) times and both the lower and upper bound checks would each occur O(*h*/2) times. Therefore, the running time for the function would be O(*h* + *h*/2 + *h*/2) = O(2*h*), which is asymptotically proportional to O(*h*). Therefore, the worst case running time for RangeCount is O(*h*).

For Ksmallest and Klargest, my implementation, assuming a valid range, first loops through the hash table until *k* elements have been added to either a min or max heap, depending on the function. Then, it peeks on the heap and gets the boundary element, either the max for Ksmallest or min for Klargest. Then, picking up where the last loop ended, you loop through the rest of the hash table and, if an element is smaller or larger than the specified element (depending on the function), you add it to the heap and then delete the head of the heap, which becomes the new boundary element.

Regardless of worst or best case, the entire hash table must be traversed. Therefore, the running time will be O(*h*) in addition to whatever other operations must occur. The worst case for both Ksmallest and Klargest is if every single element in the hash table has to be added to the heap. The case for Ksmallest would be if the hash table was in sorted order and the case for Klargest is if the hash table was in reversed sorted order. The cost of adding a single element to the heap is asymptotically proportional to O(log *k*). Therefore, the cost of inserting every element in the hash table into the heap is asymptotically proportional to O(*n* log *k*). Therefore, considering the traversal of the whole has table, the overall worst case performance for Ksmallest and Klargest is proportional to O(*h* + *n* log *k*).

5.