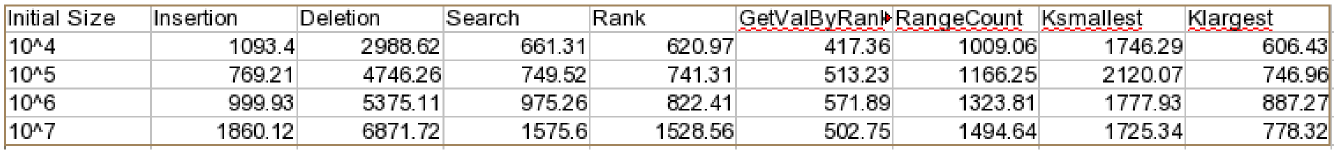
Pre-Submission Note:

Hey, I just wanted to let you know that the reason I turned this part of the project in late is because I was at a conference in Washington D.C. this previous weekend (11/03/14) and had a significantly less amount of free time than I thought that I would.

Gray Houston
ghousto
0026483532

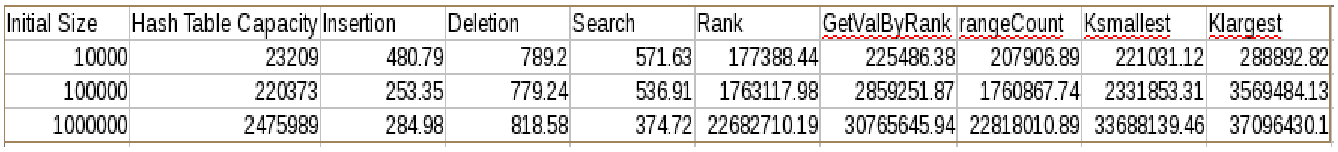
Project 3 Analysis/Report

Report 1: RB Tree Performance



Initial Size	Insertion	Deletion	Search	Rank	GetValByRan	RangeCount	Ksmallest	Klargest
10^4	1093.4	2988.62	661.31	620.97	417.36	1009.06	1746.29	606.43
10^5	769.21	4746.26	749.52	741.31	513.23	1166.25	2120.07	746.96
10^6	999.93	5375.11	975.26	822.41	571.89	1323.81	1777.93	887.27
10^7	1860.12	6871.72	1575.6	1528.56	502.75	1494.64	1725.34	778.32

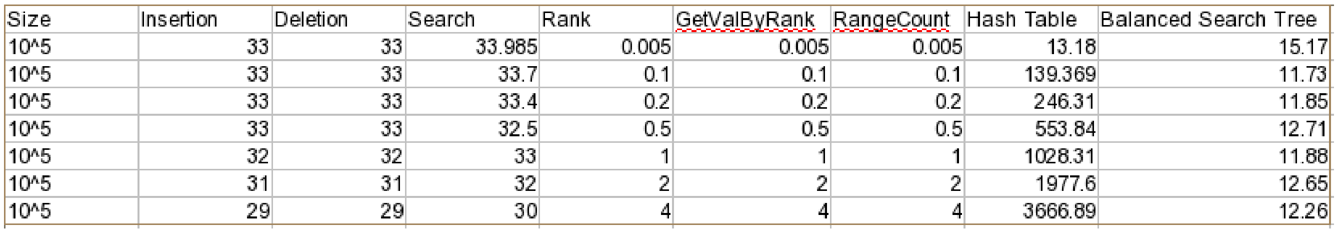
Report 2: Linear Probing Hash Table Performance



Initial Size	Hash Table Capacity	Insertion	Deletion	Search	Rank	GetValByRank	rangeCount	Ksmallest	Klargest
10000	23209	480.79	789.2	571.63	177388.44	225486.38	207906.89	221031.12	288892.82
100000	220373	253.35	779.24	536.91	1763117.98	2859251.87	1760867.74	2331853.31	3569484.13
1000000	2475989	284.98	818.58	374.72	22682710.19	30765645.94	22818010.89	33688139.46	37096430.1

Report 3: Hashing vs. BST

- functions are represented in percentage of total calls



Size	Insertion	Deletion	Search	Rank	GetValByRank	RangeCount	Hash Table	Balanced Search Tree
10^5	33	33	33.985	0.005	0.005	0.005	13.18	15.17
10^5	33	33	33.7	0.1	0.1	0.1	139.369	11.73
10^5	33	33	33.4	0.2	0.2	0.2	246.31	11.85
10^5	33	33	32.5	0.5	0.5	0.5	553.84	12.71
10^5	32	32	33	1	1	1	1028.31	11.88
10^5	31	31	32	2	2	2	1977.6	12.65
10^5	29	29	30	4	4	4	3666.89	12.26

- data structure performance averaged where n = 10

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.
The data shown in table 1 and table 2 shows the mean performance of each function in nanoseconds on different sizes of data averaged over 1000 iterations (n = 1000) for each function for either RB Trees (table 1) or Hash Tables (table 2). For both Reports 1 and 2, the running times shown in the data appear to empirically confirm the worst case running times for all the functions proven in the questions above.

For RB Trees, you can infer that all the project implemented functions run as fast if not faster than insert, delete, and search. In comparison, these same functions for Hash Tables run orders of magnitudes slower. Therefore, in a situation where you will need to get ranges in the data or determine rank or get the k largest or smallest elements, you will want to use a RB Tree.

However, the cost for the basic (e.g. insert, search, delete) functions in RB Trees is proportional to the number of elements in the set. Therefore, in problems where you only need to check if a certain element is within the data set, you will want to use a Hash Table since the performance of these basic functions is independent of the size of the data, as the functions run in O(1) time on average.

6.
Table 3 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

7.
While the ECE students are correct that the **worst case** performance of hash tables is O(*n*), they do not consider the expected frequency of the worst case nor the expected (average) performance of hash tables. For use in a real project, you must look at these factors as well as worst case performance to decide whether or not to use a certain data structure. Through analysis of these factors, I will show that the ECE students are wrong.

To determine the average case performance of a linear probing hash table, you must address what happens in each possible case of operation. Assuming a dynamically resizing hash table, the two possible cases are that the hash table does or doesn’t need to be resized. If the table is resized, a new array must be allocated and every key must be re-hashed into the new array. This whole operation runs in O(*n*) time. However, this case occurs infrequently (only when $\lambda \geq ½$).

Furthermore, this “heavy” cost can be distributed across all the following operations before the next array resizing. When an operation occurs that doesn’t result in array resizing, it runs in constant time. In an amortized analysis where you distribute the present cost of resizing the hash table across future operations, it can be proven that the average performance of the basic functions is O(1), constant time.

The flaw in the ECE students’ logic is that they don’t take into account the average performance. While the worst case is O(*n*), this case occurs infrequently and the expected performance of any arbitrary function call is O(1). Therefore, on average, the performance of these functions is O(1). Furthermore, you could simply initialize the hash table to be more than double the possible number of elements that will be inserted into it. Therefore, the table will never be resized and the performance of the specified functions will always be O(1). Thereby, you completely eliminate the possibility of the worst case occurring.

8.
In a project where there will be any number of calls the RangeCount, I would suggest using a Balanced BST instead of a hash table. I would use a BBST because RangeCount's worst case performance for a BBST is O(log *n*) where a Hash Table's worst case is O(*h*). This difference is very significant. As shown in Reports 1 & 2, the performance of RangeCount in a BBST is orders of magnitude faster than Hash Tables, even when the BBST is operating on data sets orders of magnitude larger than the data sets in the Hash Table. For this reason, I would suggest that the CS 240 student use a BBST instead of a Hash Table.

9.
For this question, I will assume we use a hash function that perfectly distributes the elements uniformly across the range of index values in the hash table.

For Separate Chaining, the average number of probes in a successful search is $1 + (\lambda/2)$. We can use this equation to determine the necessary size of the hash table:

$$\begin{aligned}
&1 + (\lambda/2) = 2 \\
&\lambda = 2 \\
&\lambda = N / M \\
&M = N / \lambda \\
&M = 1500 / 2 \\
&M = 750
\end{aligned}$$

Therefore, to produce an average number of probes in a successful search using separate chaining for about 1500 elements, you should make a hash table for about 750 elements long.

For Linear Probing, the average number of probes in a successful search is defined by the equation $\frac{1}{2} * (1 + 1 / (1 - \lambda))$. We can use this equation to determine the correct size of the hash table:

$$\begin{aligned}
&\tfrac{1}{2} * (1 + 1 / (1 - \lambda)) = 2 \\
&\lambda = 2/3 \\
&\lambda = N / M \\
&M = N / \lambda \\
&M = 1500 / (2/3) \\
&M = 2250
\end{aligned}$$

Therefore, for parameters specified in the problem, we should use a hash table that is about 2250 elements long.