Ontario
Canada

March 11, 2022

Mississauga
Canada

Dear Riley:

Thanks for your letter. It is a pleasure for me to join this interview, and i am looking forward to join the Super-Awesome-Org. And here is my solution to the problem you give me, and i will explain my solution in this letter at the same time. There are mainly three different functions in this operation.
The first function is the insert function. Since the logic for this function is too complicated, i prepare a helper function which is called inserthelper. In the function inserthelper, it firstly gets the value hash(k, capacity = none), which k(an integer) and function hash have already been provided in the previous function, and we call this value as c(an integer). If the value in the list of class HashTable at index c is none or delete, adding a Node with key k(an integer) and value v(this attribute could be any type, according to what is given) to the list at index c. If the value on the index c of the list is not none or Delete, we firstly change the value of the capacity in the function hash, using a loop to search a place on the list that the value on the index of Hash(k, capacity) is None or Delete; since the size of the HashTable must less than 75 percent of its capacity, thus we will always find a place satisfy. After that, insert the Node(k, v) to this place. And for the whole function insert, fisrtly, since we adding a node to the HashTable, the size of HashTable will increase by one logically. After increasing its size, if the size of the HashTable has already be larger than 75 percent of the capacity right now, we have to increase the capacity of the the HashTable by doubling it, and insert every node already in the list to the new list by using the inserthelper function, and finally insert the new node, otherwise, we just need to insert the new node to the list. In the worst case of this function. In the worst case of this function, assume when we adding this new node to the array would cause the capacity of the HashTable become 2 times of its orginal, and we have to add all of the nodes into the new list, so we will use n times inserthelper. And for the worst case of inserthelper, when we are trying to add the node to the HashTble, in the first n - 1 tries, there always be a node in the list at Hash(k, capacity)'s index, so we have to run the nth try to find the place the new node could fit in; thus, the time complicity of inserthelper is O(n). And since the worst case of insert will run n times inserthelper, the time for the worst case of insert is n $\times$ n = n^2. Thus, the time complicity of insert is O(n^2).
The second function is the search function. In this function, we firstly calculate

Hash(k, capacity = none), and if the key of the node at this index in the list of the HashTable is same with the k we are searching, then it means that this node is absolutely the node we are looking for, we just return this node's value, and this simple . Otherwise, according the capacity of the HashTable, we have to check each node or value in the list at the index Hash(k, capacity), where capacity is a constant number from 0 to the capacity of the HashTable - 1. If any of them's key is same with k, then it is the node we are looking for; otherwise, it means that the k we are searching is not in the list, thus, return a False. In the worst case, the list is full of nodes, however, none of them has a key equals to k. In this case, the function will check each element in the list and return a false, so the run time for this case is n. Thus, the time complicity of search is O(n).

The third function is the Delete function. I use function search and two other functions as its helper functions. The first helper function i called it as Deletehelper1, this function is use for finding the actual index of the node with key k in the list of the HashTable. In this function, i assume that there must be a node in the list that has a key is k. Firstly, this code calculate the value of hash(k, capacity = none), and if the key of node in the HashTable's array at this index is equals to k, then we return this integer; Otherwise, the code will go through each node at Hash(k, capacity = [0: HashTable's Capacity - 1] to check these nodes' key by a loop, and after finding the node with key k, return the index of that node. In the function "Delete", firstly we use a search function, if it returns false, it means that none of the node in the list has a key=k, so we do not need to delete anything. Otherwise, we can confirm that we should delete a node in the list, so it will reduce the size of the HashTable by 1. For deleting, we firstly finding the index of the node in the list, after that change it into a Delete. And in the end, we should check whether we should change the capacity of the HashTable, and if we have to do so, we need to add the current nodes into the new list at the same time. By achieveing this movement, i prepare a helperfunction which is called Deletehelper2. In this function, i firstly use two if statement to check whether the size of the HashTable is less than 25 percent of its capacity and the size is larger than its initial capacity, the HashTable's capacity will shrink by half at first, after that, the function will add every node in the current list after delete into the new HashTable's list by using inserthelper. To calculate the time complicity of this function, firstly i do a search, which is O(n). In the worst case, the node we are trying to delete is in the list, so we have to use a Deletehelper1 to find the index of the node we are trying to delete. In the worst case of Deletehelper1, the list is full of nodes, and the node we are looking is the last element to check base on hash(k, capacity), which means the operation runs n times; so the time complicity of Deletehelper1 is 0(n). After the delete, we still need to check whether the capacity need to shrink or not by using a Deletehelper2. And in the worst case of Deletehelper, since the capacity is shrink, we need to insert each node after delete to the new list by using inserthelper, the time it spend is n $\times$ n = n^2. Thus, the time of the worst case for Delete spend is n + n + n^2 = n^2 + 2n, and as a result, the time complicity for function Delete is O(n^2).

For the unit test part, since capacity = 0 or negative numbers are meaningless. I choose two numbers 5 and 1 to be tested and i still keep the test you provide. When initial capacity = 5, when we adding 4 nodes, the size = $4 > 5 \times 0.75$, so the capacity will be double at that time, and we will add two more nodes, the capacity is not change. After that when we delete four of them, the capacity will shrink and back to 5. And i still do check for deleting a k not belongs to any Node in the previous HashTable, and there should be no changes, and i still do one more delete test, after this deletion, although the size is less than 25 percent of capacity, however, since the current capacity is equal to the initial capacity, the capacity should not change. And i still check a case when initial capacity = 1, when we adding a node to it, the size is equal to current capacity, the capacity will double, and when we delete one it will shrink.

Futhermore, for the question you provided, providing an analysis of the amortized runtime if we were to start with a hash table with initial capacity of 10, insert N elements, and then delete all N elements; i would like to use aggregate method to do this amortized analysis. Assuming the cost for doing one Hash(k, capacity) is 1. In the worst case, each time when we are trying to add a new element to the current array, it always fail k times, where k equals the total number of element we have already added. And the cost sequence will be 1, 2, 3, 4, 5, 6, 7, 36, 9, 10, 11, 12, 13, 14, 120, 16... we can summarize that $\text{cost}_i = \begin{cases} \sum_{x=1}^{i} x & \text{if i = floor}(0.75 \times 10 \times 2^k \text{ for some k} \in \text{Z}) \\ i & \text{otherwise} \end{cases}$

Since we insert N element, the total runtime for insert is $\sum_{i=1}^{n} cost_i = \sum_{i=1}^{n} i + \sum\sum_{x=1}^{x=i-1} x = \frac{n(n+1)}{2}$ (by the formula of the sum of arithmetic sequence+ $\sum_{j=1}^{log2^n} 0.75 \times 10 \times 2^j = \frac{n^2+n}{2} + 7.5 \sum_{j=1}^{log2^n} 2^j = \frac{n^2+n}{2} + 7.5 \times [2 \times \sum_{j=1}^{log2^n} 2^j$ - $\sum_{j=1}^{log2^n} 2^j] = \frac{n^2+n}{2} + 7.5 \times (2(\log_2 n + 1) - 2) = \frac{n^2+n}{2} + 7.5 \times (2n\text{-}2) = \frac{n^2+n}{2} +$ 15n - 15. Thus, by the amortized analysis, we can concluded that the runtime of insert n nodes is $O(n^2)$.

And for delete n elements, the cost sequence will be n, n - 1, n - 2, n - 3, ..., $\sum_{x=1}^{i} n - x + 1$, ...

we can summarize that $\text{cost}_i = \begin{cases} \sum_{x=1}^{i} n - x + 1 & \text{if i = floor}(0.25 \times 10 \times 2^k \text{ for some k} \in \text{Z}) \\ n - i + 1 & \text{otherwise} \end{cases}$

Thus, the total cost is $\sum_{i=1}^{n} cost_i = \sum_{i=1}^{n} n - i + 1 + \sum\sum_{x=1}^{x=i-1} n - x + 1 = \sum_{i=1}^{n} i + \sum_{j=1}^{log2^n} 0.25 \times 10 \times 2^{n-j+1} = \frac{n^2+n}{2} + 2.5 \sum_{j=1}^{log2^n} 2^{n-j+1} = \frac{n^2+n}{2} + 2.5 \sum_{k=1}^{log2^n} 2^k = \frac{n^2+n}{2} + 2.5 \times (2^{\log_2 n+1} - 2) = n^2 + n_{\frac{}{2}} + 2.5 \times (2n - 2) = \frac{n^2+n}{2} +$ 5n - 5.

And this is my response, and thanks for reviewing my code and explanations. If you have any problem with this solution, please call me. It is my pleasure to join this interview, and i am looking forward to have the interview and join the

Super-Awesome-Org. Thank you very much.

Yours Faithfully,

Sheng, Chi