Dear Riley,

Thank you for reaching out to me and I hope you are doing well. I am writing to inform you that I have finished the task that you assigned to me. Attached to this email is the Python code I wrote to solve this problem. As well, I have provided an explanation for my code below, followed by the amortized complexity analyses for both the scenarios described in your email.

Within the starter code I was given, I implemented the provided `insert`, `search`, and `delete` methods, where all of them use a helper method which I named `find`. I also use another helper which I named `rehash`.

`find` is a method where given an array `arr`, its capacity c, and a key k, returns the index of this key if it already exists in the hash table, or an empty index to insert k into if it is not already in the table. First, it hashes the key using the given hash function, whose `capacity` parameter is set to c. I will denote this hashed value as $h(k)$. Then, the function checks `arr` at index $h(k)$. If the address is unoccupied or contains the same key k, then the method will return the index $h(k)$. If not, it must be true that either another key or the DELETED string is present at this index. The reason why the method does not return the index of a DELETED string immediately is because k can potentially lie at another index, as a result of quadratic probing. Thus it must first confirm whether k exists before deciding what to return. In this case, the method keeps track of the index of the first DELETED occurence encountered so that it knows what to return if k is not in the hash table. Then, using quadratic probing, the method continually performs the same checks at indices $h_i = h(k) + i^2 \pmod{c}$, where i is a positive integer, starting at one, that increments at each iteration. The loop stops when it finds an empty address or an address that contains the key k. Note that this algortihm is guaranteed to terminate given that we know capacity of the hash table is less than half, although proving that is the case is quite lengthy. However, I am willing to elaborate if need be.

The other helper method, `rehash`, given a new capacity c, creates a new array with capacity c, and iterates through the current array to reinsert the existing elements into the new array by finding a suitable address for each element using `find`, and assigning the key to that address. Particularly, `find` is called with each of its arguments being the new array, capacity c, and some key k from the current hash table, respectively.

The `insert` method, given a key k and value v, inserts the key and value into the hash table with capacity c in two steps. To start, it finds a suitable address for insertion using `find` and inserting the key-value pair `Node`(k, v) in that address. If the node was inserted into an address that did not previously contain k, the size attribute is incremented. If the capacity of the hash table exceeds half, the method will call `rehash` with the capacity argument equal to twice the current capacity in order to expand the hash table.

Next, the `search` method takes a key k as input and attempts to return the value associated with that key. If the hash table cannot find the key, the method returns None. This method uses `find` to find the address where k would be inserted. From the implementation of `find`, the address returned is guaranteed to either be empty, DELETED, or contain the key k. If the address does not contain a node, then the method returns None. Otherwise, the method will return the value in the node.

Finally, the `delete` method deletes the key k from the hash table. The implementation is very similar to `insert`. First, it attempts to find k with `find`. If k is inside the address found with `find`, replace it with the DELETED string decrement the size attribute. Otherwise, the method does nothing. Afterwards, it checks if the capacity is below one quarter. If it has, and the maximum capacity is greater than the initial capacity, the method will call `rehash` with new capacity equal to half of the current capacity to shrink the hash table.

That concludes the explanations for the implementation of the hash table. I will continue on with the amortized analyses of the two situations described in your email. In both situations, I will use the aggregate method to find the amortized complexity.

The first situation begins with an empty hash table with intial capacity 10, where N elements are inserted, and then N elements are deleted.

Focusing on each insert operation first, on the ith insert, there are going to be $i-1$ elements in the hash table already, and the worst-case situation is if all i elements—the $i-1$ previous insertions plus the one to be inserted next—hash to the same address. This means that the `insert` method must iterate through all of the previous elements in order to find an available address. In this case, the number of array accesses is i. After the insertion, if the hash table is half full, the method will need to reinsert each element into a new array with double the capacity. If this occurs, the number of accesses to the original array is $2i$ and in the worst-case where every element hashes to the same address in the new array, inserting the jth element will take j array accesses. Our possible capacities, given that the initial capacity is 10, can be represented by $10 \cdot 2^n$ with n being a non-negative integer. The rehashing only happens if $i = 5 \cdot 2^n$, which is half of some potential capacity. Then if a_i is the number of array accesses made in the ith insertion operation, we can write

$$a_i = \begin{cases} i + 2i + \displaystyle\sum_{j=1}^{i} j, & \text{if } i = 5 \cdot 2^n, \text{ for some } n \geq 0, \\ i, & \text{otherwise.} \end{cases}$$

Note that $5 \cdot 2^n < N$, so n must be between 0 and $\lfloor \log N - \log 5 \rfloor$, where the logarithm is in base 2. Summing up all N insertions, we have that the total worst-case runtime of the insertions is

$$\begin{aligned} \sum_{i=1}^{N} a_i &= \sum_{i=1}^{N} i + \sum_{n=0}^{\lfloor \log N - \log 5 \rfloor} \left(2 \cdot 5 \cdot 2^n + \sum_{j=1}^{5 \cdot 2^n} j \right) \\ &= \sum_{i=1}^{N} i + \sum_{n=0}^{\lfloor \log N - \log 5 \rfloor} 2 \cdot 5 \cdot 2^n + \sum_{n=0}^{\lfloor \log N - \log 5 \rfloor} \sum_{j=1}^{5 \cdot 2^n} j \\ &= \sum_{i=1}^{N} i + \sum_{n=0}^{\lfloor \log N - \log 5 \rfloor} 2 \cdot 5 \cdot 2^n + \frac{1}{2} \sum_{n=0}^{\lfloor \log N - \log 5 \rfloor} 5 \cdot 2^n (5 \cdot 2^n + 1) \\ &= \sum_{i=1}^{N} i + \frac{25}{2} \sum_{n=0}^{\lfloor \log N - \log 5 \rfloor} 2^n + \frac{25}{2} \sum_{n=0}^{\lfloor \log N - \log 5 \rfloor} 4^n \\ &= \frac{1}{2} N(N+1) + \frac{25}{2} \cdot (2^{\lfloor \log N - \log 5 \rfloor + 1} - 1) + \frac{25}{2} \cdot \frac{4^{\lfloor \log N - \log 5 \rfloor + 1} - 1}{3} \end{aligned}$$

Removing the floor operator will obtain an upper bound for the runtime:

$$\begin{aligned} \sum_{i=1}^{N} a_i &\leq \frac{1}{2} N(N+1) + \frac{25}{2} \cdot (2^{\log N - \log 5 + 1} - 1) + \frac{25}{2} \cdot \frac{4^{\log N - \log 5 + 1} - 1}{3} \\ &= \frac{1}{2} N(N+1) + \frac{25}{2} \cdot \left(2 \cdot \frac{N}{5} - 1 \right) + \frac{25}{2} \cdot \frac{4 \cdot \frac{N^2}{25} - 1}{3} \\ &= \frac{7}{6} N^2 + \frac{11}{2} N - \frac{25}{2} - \frac{25}{6} \in \mathcal{O}(N^2) \end{aligned}$$

We can perform a very similar analysis on the N delete operations. The worst-case runtime is when every element hashes to the same address at every capacity, and each element is deleted in backwards order of the most recent rehashing. In particular, with my implementation, rehashing is done linearly, so the last key in the hash table will be inserted last, and due to open addressing, must iterate through the address of every key inserted before it. This guarantees that every other element must be checked before the method is able to access the element to be deleted. For similar reasons, if b_{N-i+1} represents the $N-i+1$th delete operation, then

$$b_{N-i+1} = \begin{cases} i + 4i + \sum_{j=1}^{i} j, & \text{if } i = 5 \cdot 2^n,\ n \in \mathbb{N} \\ i, & \text{otherwise.} \end{cases}$$

This is because after $N-i$ deletions, there are i elements remaining in the hash table, so the subsequent deletion without accounting for array shrinking will have a runtime of i. As well, when rehashing, i is a quarter of the maximum capacity, and the rehashing algorithm needs to iterate through the entire array, so there are actually $4i$ array accesses rather than just $2i$. Moreover, the resizing only happens if i is a quarter of the maximum capacity and the current maximum capacity is not the initial capacity. Namely, when $i = 5 \cdot 2^n$. Thus we can do a very similar calculation as before to obtain that

$$\sum_{i=1}^{N} b_i \in \mathcal{O}(N^2).$$

Adding the aggregated sum of the insert and delete operations together will still give us a cost in $\mathcal{O}(N^2)$, so the amortized cost of the sequence in the first situation is $\mathcal{O}(N)$.

In the second situation, every insert and delete operation will require a rehashing. This is because the capacity of the hash table is one insertion away from half capacity, so after the insertion the table will be expanded. After this expansion, the capacity is at exactly one quarter, and the current capacity is not the initial capacity (which must be less than or equal to $2N$), and deleting an element will cause the table to shrink. After these two operations, the table "returns" to the same state as how it started, so the next pair of insertion and deletion will do the same thing. Again, the worst-case input for this situation is if every element hashes to the same address, and `delete` is called on the element that is inserted last in the rehashing. This being the case, every pair of insert and delete operations will have the runtime

$$t = \left(N + 2N + \sum_{j=1}^{N} j\right) + \left(N + 4N + \sum_{j=1}^{N} j\right) = 8N + 2\sum_{j=1}^{N} j = 8N + N(N+1) = N^2 + 9N.$$

Since there are N pairs in total, the total cost of all operations is

$$\begin{aligned}\sum_{i=1}^{N} t &= \sum_{i=1}^{N} N^2 + 9N \\ &= N^3 + 9N^2 \in \mathcal{O}(N^3).\end{aligned}$$

Thus the amortized runtime in the second situation is $\mathcal{O}(N^2)$, which completes both the complexity analyses.

I hope that my explanations and analyses were satisfactory, and I look forward to hearing back from you soon. Thank you very much.

Ethan Hua