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`.

`find` is a method where given an array `arr`, its capacity c, and a key k, returns the index of where this key should be. 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 if it has already been occupied by the key k, then the method will return the index $h(k)$. Otherwise, if the address is occupied by another key, 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 program stops when it finds an empty address or an address that contains the key k, and returns that address. 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 `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 inserts the given key and value into the hash table by finding a suitable address for insertion using `find` and placing the key-value pair `Node`(k, v) in that address.

After the insertion, if the capacity of the hash table exceeds half, the method creates a new array with double the capacity of the current array, and iterates through the current array to reinsert the existing elements into the new array, following the steps outlined above. The only difference is when hashing each key, the hash function is called with the capacity parameter set to twice the current capacity. The implementation is fairly straightforward. The insertion is done with a standard quadratic probing approach, and rehashing necessarily needs to iterate through every key, so a simple loop through the array gets the job done.

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. The address is guaranteed to either be empty or contain a key-value pair with the key being equal to 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. Otherwise, the method does nothing.

Afterwards, it checks if the capacity has gone below one quarter. If it has, and the maximum capacity is greater than the initial capacity, the method will initialize a new array with half the current capacity, and rehash all the existing elements into the new array using the same method described in `insert`.

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.

The first situation begins with an empty hash table with intial capacity 10, where N elements are inserted, and then N elements are deleted. I will analyse the amortized complexity using the

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—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, then

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

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. There is a

symmetry between the runtime of the insert and delete operations, namely if b_i represents the ith delete operation, then

$$b_i = a_{N-i}.$$

Thus

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

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 any operation is $\mathcal{O}(N)$.

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