CSC263H5 Problem Set 3

Robert Motrogeanu (1008070169/motroge1)

March 12th 2023

Dear Riley,

I have attached the code as the `csc263_ps3.py` file below. The correctness of the `HashTable` class can be divided up into three parts, `INSERT(k, v)`, `SEARCH(k)` and `DELETE(k)`.

`INSERT(k, v)` correctness:

- For `INSERT(k, v)`, we first use the provided hash function to generate an index, and then check if it maps to free slots in the `HashTable` (either `None` or `"DELETED"`). If so, we insert the (k, v) pair at that index. Otherwise, we use a linear probing sequence to find the next free slot.
- After the insertion, we check if the `capacity` attribute of the `HashTable` is $\geq 50\%$. If so, we create a new `HashTable` with doubled size, and re-hash all (k, v) pairs from the old `HashTable` into the new one. This is to ensure the hashing function is consistent, as the function is dependent on the `capacity` attribute. Had we not re-hashed all old (k, v) pairs into a new table, all (k, v) pairs would have indexes generated with the old `capacity` attribute from the hashing function, breaking our `SEARCH` algorithm.
- We always double the capacity at $\geq 50\%$ capacity to ensure that we are always guaranteed free slots.

`SEARCH(k)` correctness:

- For `SEARCH(k)`, we generate an index and probe until a free slot. While probing, we check if the index contains the key we are searching for. The `SEARCH` algorithm terminates once we encounter an empty slot (a `None` type slot).
- We do not stop at `"DELETED"` slots because our insertion algorithm maps elements into the `HashTable` in clusters. If we delete an element in the middle of a cluster, then the `SEARCH` algorithm would give up on a `"DELETED"` slot even though the key being searched could be further in the cluster.

- The reasoning above (clustering) is also why we stop probing upon a `None` slot.

`DELETE(k)` correctness:

- For `DELETE(k)`, we use the `SEARCH` algorithm to determine if the key being searched for exists. Upon existence, we mark the slot with the string `"DELETED"`, delete the (k, v) pair at the slot, and then decrement the size of the `HashTable`.
- Marking deleted slots in the `HashTable` is necessary to prevent the `SEARCH` algorithm from breaking, as it terminates when it reaches a `None` slot.
- If the size of the `HashTable` is $\leq 25\%$, we halve the `HashTable`'s capacity and re-hash all (k, v) pairs into a new `HashTable` with the appropriate capacity to maintain hashing function consistency.

Next, let's analyze the amortized run-time using aggregate analysis. Suppose that n worst-case `INSERT` operations occur, followed by n worst-case `DELETE` operations. Assume that our hashing function always collides with taken slots, meaning we probe as much as possible. Let $T(n)$ represent the total operations, assuming that probing once, inserting a (k, v) pair, and deleting a (k, v) pair take one step each. Then,

$$T(n) = n + \sum_{i=1}^{n} i + \sum_{i=1}^{\lfloor log_2 \frac{n}{5} + 1 \rfloor} (5 \cdot 2^{i-1} + \sum_{j=1}^{5 \cdot 2^{i-1}} j) + (1 + 2 + 3) + 3$$

where the first n term represents the total insertions, and the first sum represents the total probes. The second sum represents the number of times we resize the hash table, accompanied by the amount of inserts and probes in the new table, where $5 \cdot 2^{i-1}$ is the size of the new table. The last terms account for the first resizing, where there are $(1 + 2 + 3)$ probes and 3 insertions in the new table.

Notice that:

$$T(n) = n + \sum_{i=1}^{n} i + \sum_{i=1}^{\lfloor \log_2 \frac{n}{5} + 1 \rfloor} (5 \cdot 2^{i-1} + \sum_{j=1}^{5 \cdot 2^{i-1}} j) + (1 + 2 + 3) + 3$$

$$T(n) \leq n + \sum_{i=1}^{n} i + \sum_{i=1}^{\log_2 n + 1} (5 \cdot 2^{i} + \sum_{j=1}^{5 \cdot 2^{i}} j) + (1 + 2 + 3) + 3$$

$$T(n) \leq \frac{403n^2}{6} + \frac{63n}{2} - \frac{68}{3n}$$

$$\frac{T(n)}{n} \leq \frac{\frac{403n^2}{6} + \frac{63n}{2} - \frac{68}{3n}}{n}$$

$$\frac{T(n)}{n} \leq \frac{403n^2 + 189n - 136}{6n} \qquad \text{(computed using Wolfram Alpha)}$$

$$\frac{T(n)}{n} \in \Theta(n)$$

Thus, the amortized cost per operation for `INSERT` is $\Theta(n)$. However, the sum of operations for n worst-case `DELETE` operations is the same as $T(n)$. This is because `DELETE` re-hashes into a new `HashTable` as well, meaning we re-size, probe, and insert the same amount of times as if we went backwards from the n insertions we did earlier, starting from n insertions. Thus, the amortized cost per operation is also $\Theta(n)$ as well for `DELETE`.

I loved working on this problem, and I'd love to hear your thoughts on my solution, and how we can move forward in the hiring process at Super-Awesome-Org. I am looking forward to your response!

Thanks,
Robert