CS 225: Data Structures

Homework 6

Group D1

Last Modified on April 1, 2022

Members	
Name	**Student ID**
Li Rong	3200110523
Zhong Tiantian	3200110643
Zhou Ruidi	3200111303
Jiang Wenhan	3200111016

Please turn over for our answers.

Exercise 4 is written by Li Rong & Jiang Wenhan.

Ex. 1 **OUR ANSWER.** (This part is written by Zhou Ruidi)

Define $E(n)$ as the expectation of the number of non-empty lists after n insertions. In this way, if we have inserted $(n-1)$ elements, then the expectation is $E(n-1)$.

Initial condition. Inserting the first element,

$$E(0) = 1 \times \frac{1}{m} = \frac{1}{m}$$

Induction. Inserting the n-th element.

(a) If the element should be inserted into a non-empty list, then the probability we insert it here is

$$P_{\text{non-empty}}(n) = \frac{E(n-1)}{m} \tag{1}$$

(b) If the element should be inserted into an empty list, then the probability we insert it here is

$$P_{\text{empty}}(n) = 1 - P_{\text{non-empty}}(n) = 1 - \frac{E(n-1)}{m} \tag{2}$$

Hence the expectation after inserting n-th element is

$$E(n) = P_{\text{non-empty}}(n) \times [E(n-1)+1] + P_{\text{empty}}(n) \times E(n-1) \tag{3}$$

With Equation 1, 2, 3 we solve for a general representation of $E(n)$:

$$E(n) = m \cdot \left(1 - \frac{1}{m}\right)^n \tag{4}$$

Ex. 2 **Our Answer.** (This part is written by Jiang Wenhan)

(i) To calculate the total payment for one customer, we only need to traverse all triples and add up all prices whenever the customer ID fits the one we need. So the time complexity is $O(n)$. (n refers to the number of triples)

(ii) Assume the size of represented array is n, and the size of stored value is m. For hashing with chaining, we assume the hashing is perfect where no empty entry exist. We need to scan all entries to get all return values. So the running time is $O(m)$. For hashing with linear probing, there exist empty entries. But we can not make assertion for each entry whether it is empty or not. So we have to scan all entries, which takes $O(n)$.

Ex. 3 **Our Answer.** (This part is written by Zhong Tiantian)

(i) Focusing on condition $h_j(e) \leq M-1$. Assuming we have an element e. Every time if the calculated hash value is larger than M, then increase the size of the table by

$$M' \leftarrow M + m \cdot 2^j$$

.

Obviously, for current j we must have

$$M \leq h_{j-1}(e) < h_j(e) \leq M'$$

in order to satisfy the properties of the hash function.

With the assumptions above and suppose we are using the j-function h_j:

Hash value 1. It's possible to calculate the current hash value with j-th hash function $h_j(e)$. Check if the "bucket" for $h_j(e)$ has been full (i.e. the length of "bucket" exceeds limit l); if not, put e to the corresponding position of $h_j(e)$. This is the first hash value.

Hash value 2. If bucket with key $h_j(e)$ has been full, calculate the next hash function $h_{j'}(e)$ with $j' = j+1$. From the assumption we made at the beginning, $h_{j'}(e) > M$ must hold. Thus the bucket for $h_{j'}(e) > M$ must be empty, which means we can safely put the element into this position. After that an increase to the table size by

$$M \leftarrow M + m \cdot 2^j$$

can maintain our assumption.

And this also proves that two hash value for an element e suffices.

(ii) Define

- a dynamically-sized hash table with M elements, where the size of the table is controlled by M.
- the hash function as `Hash(`j`, `e`)` which is an implementation of $h_j(e)$.
- lists `HashList[`i`]` as the list of element whose hash value equals to i.

The algorithm `rehashing` looks like Algorithm 1.

(iii) Rewrite Algorithm 1 to obtain Algorithm 2, where we add a checking process to ensure the corresponding list of an element is only splitted if the original list is full, to avoid unnecessary splitting.

Algorithm 1: Rehashing v1

```
Input  : a set of elements $\mathscr{T} = \{e_1, e_2, \cdots, e_n\}$, number of buckets $m$
Output: a hash table.
$M \leftarrow m$
forall $e \in \mathscr{T}$ do
    if HashList[Hash($j$,$e$)] is full then
        $j \leftarrow j + 1$
        $M \leftarrow M + m \cdot 2^{j-1}$
        Put $e$ into bucket HashList[Hash($j$, $e$)]
    end
    else
        Put $e$ into bucket HashList[Hash($j$, $e$)]
    end
end
return HashList
```

Algorithm 2: Rehashing v2

```
Input  : a set of elements $\mathscr{T} = \{e_1, e_2, \cdots, e_n\}$, number of buckets $m$
Output: a hash table.
M ← m, j ← 0
forall $e \in \mathscr{T}$ do
    /* Find the last bucket that has been used for current element */
    while ListUsed[HashList[Hash($j$,$e$)]]=False and $j > 0$ do
        $j \leftarrow j - 1$
    end
    /* If the last-used bucket is full then split it */
    if HashList[Hash($j$,$e$)] is full then
        $j \leftarrow j + 1$
        $M \leftarrow M + m \cdot 2^{j-1}$
        Put $e$ into bucket HashList[Hash($j$, $e$)]
        ListUsed[HashList[Hash($j$, $e$)]] ← True
    end
    else
        Put $e$ into bucket HashList[Hash($j$, $e$)]
    end
end
return HashList
```