CSC263 – Problem Set 3

Si Tong Liu(1004339628), Jing Huang(1003490705), Yifei Gao(1004152640)

Remember to write your **full name** and **student number** prominently on your submission. To avoid suspicions of plagiarism: at the beginning of your submission, **clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, that you consulted**.

Remember that you are required to submit your problem sets as both LaTeX `.tex` source files and `.pdf` files. There is a 10% penalty on the assignment for failing to submit both the `.tex` and `.pdf`.

Due Feb 25, 2019, 22:00; required files: ps3.pdf, ps3.tex, pizza.py

Answer each question completely, always justifying your claims and reasoning. Your solution will be graded not only on correctness, but also on clarity. Answers that are technically correct that are hard to understand will not receive full marks. Mark values for each question are contained in the [square brackets].

You may work in groups of up to THREE to complete these questions.

1. **[14]** In this question, we will study a pitfall of quadratic probing in open addressing. In the lecture, we showed for quadratic probing that we need to be careful choosing the probe sequence, since it could jump in such a way that some of the slots in the hash table are never reached.

   (a) Suppose that we have an open addressing hash table of size $m = 7$, and that we are using **linear** probing of the following form.

   $$h(k, i) = (h(k) + i) \bmod m, \quad i = 0, 1, 2 \ldots$$

   where $h(k)$ is some arbitrary hash function. We claim that, as long as there is a free slot in this hash table, the insertion of a new key (a key that does not exist in the table) into the hash table is guaranteed to succeed, i.e., we will be able to find a free slot for the new key. Is this claim true? If true, concisely explain why; if not true, give a detailed counterexample and justify why it shows that the claim is false.

   (b) Now, suppose that we have an open addressing hash table of size $m = 7$, and that we are using **quadratic** probing of the following form.

   $$h(k, i) = (h(k) + i^2) \bmod m, \quad i = 0, 1, 2 \ldots$$

   where $h(k)$ is some arbitrary hash function. We claim again that, as long as there is a free slot in this hash table, the insertion of a new key into the hash table is guaranteed to succeed, i.e., we must be able to find a free slot for the new key. Is this claim true? If true, concisely explain why; if not true, give a detailed counterexample and justify why it shows that the claim is false.

   (c) If either of your answers to (a) and (b) is "false", then it means that some of the slots in the hash table are essentially "wasted", i.e., they are free with no key occupying them, but the new keys to be inserted may not be able to use these free slots. In this part, we will show an encouraging result for quadratic probing that says "this waste cannot be too bad".

   Suppose that we have an open addressing hash table whose size m is a prime number greater than 3, and that we are using **quadratic** probing of the following form.

   $$h(k, i) = (h(k) + i^2) \bmod m, \quad i = 0, 1, 2 \ldots$$

   Prove that, if the hash table contains less than $\lfloor m/2 \rfloor$ keys (i.e., the table is less than half full), then the insertion of a new key is guaranteed to be successful, i.e., the probing must be able to reach a free slot.

   Hint: What if the first $\lfloor m/2 \rfloor$ probe locations for a given key are all distinct? Try proof by contradiction.

Solution:

(a) The claim is true. Since h(k) is hash function, it is the index where the key is stored in the given array with size m = 7. So the range of h(k)+i is none negative numbers, which implies that h(k,i) will always in the range of [0,6]. Besides, the key for insertion does not exist in the table, which means there will be no duplicate keys. In conclusion, inserting a new key by using linear probing is guaranteed to be true.

(b) The claim is false. Counterexample: in the question, m = 7, $h(k,i) = (h(k) + i^2)$. The possible numbers we can get after i^2 mod 7 are:

1^2 mod 7 = 1; 2^2 mod 7 = 4;3^2 mod 7 = 2;4^2 mod 7 = 2;5^2 mod 7 = 4;6^2 mod 7 = 1;7^2 mod 7 = 0 => 1, 4, 2, 2, 4, 1, 0. And we cannot reach slots like 3,5 and 6. In conclusion, by using quadratic probing, there will be some slots unreachable in the given array. So it is not guaranteed to succeed to find a free slot for the new key.

(c) assume $0 \leq i, j \leq m/2$ and $i \neq j$(since all elements in the first half array are distinct).Then we prove by contradiction:

we assume that $i \neq j$ and $(h(k) + i^2)$ mod (m/2) = $(h(k) + j^2)$ mod (m/2)

=> i^2 mod (m/2) = j^2 mod (m/2)

=> (i^2 - j^2) mod (m/2) = 0

=> i^2 - j^2 = 0 (basic algebra)

=> (i+j)(i-j) = 0

The contradiction is false, because we assumed $i \neq j$, so both (i+j) or (i-j) can not be 0. Therefore, if $i \neq j$, then $(h(k) + i^2)$ mod (m/2) $\neq$ $(h(k) + j^2)$ mod (m/2), which also means that if the hash table contains less than m/2 keys the insertion of a new key is guaranteed to be successful,

2. **[14]** Suppose that we have an array $A[1, 2, \dots]$ (index starting at 1) that is sufficiently large, and supports the following two operations `INSERT` and `PRINT-AND-CUT` (where k is a global variable initially set to 0):

```
1 def INSERT(x):
2     k = k + 1
3     A[k] = x
4
5 def PRINT-AND-CUT():
6     for i from 1 to k:
7         print A[k]
8     k = k // 2      # integer division
```

We define the cost of the above two operations as follows:

- The cost of `INSERT` is exactly 1.
- The cost of `PRINT-AND-CUT` is exactly the value of k before the operation is executed.

Now consider any sequence of n of the above two operations. Starting with $k = 0$, perform an amortized analysis using the following two techniques.

(a) Use the **aggregate method**: First, describe the worst-case sequence that has the largest possible total cost, then find the upper-bound on the amortized cost per operation by dividing the total cost of the sequence by the number of operations in the sequence.

(b) Use the **accounting method**: Charge each inserted element the smallest amount of "dollars" such that the total amount charged always covers the total cost of the operations. The charged amount for each insert will be an upper-bound on the amortized cost per operation.

Note: Your answer should be in **exact forms** and your upper-bound should be as tight as possible. For example, 7 would be a tighter upper-bound than 8, $\log_2 n$ is tighter than $\sqrt{n}$, and $4n$ is tighter than $5n$. Your upper-bound should also be a simple term like 7, 8 or $3 \log n$, rather than something like $(5n^2 - n \log n)/n$. Make sure your answer is clearly justified.

Solution:

(a) assume the size of the given array is n. Then there will be $\log_2 n$ times that the PRINT-AND-CUT function will be executed. Therefore, the total cost can be written as: T(n) = n + $\sum_{i=0}^{\log_2(n-1)} 2^i$

by geometric, we can get that S(n) = $2^0 + 2^1 + 2^2 + ...2^{\log_2(n-1)}$

$$2S(n) = 2^1 + 2^2 + ...2^{\log_2(n-1)} + 2^{\log_2(n-1)+1}$$

$$=> S(n) = 2^{\log_2(n-1)+1} - 1$$

so T(n) = n + $2^{\log_2(n-1)+1} - 1$

$\leq n + 2^{\log_2(n-1)+1}$

$\leq n + 2^{\log_2(n)+1}$

$= n + 2n = 3n = O(n)$

Total cost is O(n), and we have total number of n operations. Therefore, the average upper-bound = $n/n = 1 = O(1)$

(b) Assume the array is empty now. Since the cost of insert is O(1), for every element, we need 1 dollar to insert. And once we insert, we need one dollar to print the element. Then we need 1 dollar extra to help other cases. So the minimum cost will be 3 dollars. Therefore, the total cost will be 3n(O(n)). So the average upper-bound = $n/n = 1 = O(1)$

Programming Question

The best way to learn a data structure or an algorithm is to code it up. In each problem set, we will have a programming exercise for which you will be asked to write some code and submit it. You may also be asked to include a write-up about your code in the PDF/TEXfile that you submit. Make sure to **maintain your academic integrity** carefully, and protect your own work. The code you submit will be checked for plagiarism. It is much better to take the hit on a lower mark than risking much worse consequences by committing an academic offence.

3. [**12**] Dan's favourite food is pizza. (If you haven't tried the Cow Pie pizza in DH, you should!)

   Imagine that every pizza in the world is a circle, with exactly five slices. For each slice, Dan gives the integer quality rating of the slice. We say that two pizzas are equivalent if one pizza can be rotated so that the quality of each corresponding slice is the same.

   For example, suppose that we had these two pizzas: $(3, 9, 15, 2, 1)$ and $(15, 2, 1, 3, 9)$ These two pizzas are equivalent: the second is a rotation of the first.

   However, the following two pizzas are **not** equivalent: $(3, 9, 15, 2, 1)$ and $(3, 9, 2, 15, 1)$ because no rotation of one pizza can give you the other.

   Here's another example of two pizzas that are **not** equivalent: $(3, 9, 15, 2, 1)$ and $(9, 15, 2, 1, 50)$

   We say that two pizzas are the same **kind** if they are equivalent.

   In Python, a pizza will be represented as a tuple of 5 integers. Your task is to write the function `num_pizza_kinds`, which determines the **number** of different kinds of pizzas in the list.

   Requirements:

   - Your code must be written in Python 3, and the filename must be `pizza.py`.
   - We will grade only the `num_pizza_kinds` function; please do not change its signature in the starter code. include as many helper functions as you wish.
   - You are **not** allowed to use the built-in Python dictionary.
   - To get full marks, your algorithm must have average-case runtime $\mathcal{O}(n)$. You can assume Simple Uniform Random Hashing.

   Write-up: in your `ps3.pdf/ps3.tex` files, include the following: an explanation of how your code works, justification of correctness, and justification of desired $\mathcal{O}(n)$ average-case runtime.

   Justification:

The `compare_two_pizza` function takes two tuples in the pizzas, and make comparison between the two tuples. Inside this function, the two for loop take about O(n) runtime each, since they are basically linear search. So the runtime of `compare_two_pizza` function is O(n). Then in the function `is_pizza_in`, it determines whether the given tuple is in the given list by using `compare_two_pizza` function. So the runtime should be $O(n^2)$. Lastly,in the `num_pizza_kinds`function, everytime we see a different kind of pizza, we append the pizza into an empty list. Then by calling `is_pizza_in` function, we can count the total number of different kinds of pizzas. So the final runtime will be O(n^2*n) = O(n^3). And the total number of operations would be n^2, since we are searching the given list of tuples twice to find all possible pair of tuples. Therefore, the average-case runtime will be $n^3/n^2 = n$ = O(n).