CSC263 – Problem Set 3

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.

Authors: Junwen Shen(1004299190), Mengning Yang(1002437552), Jianhao Tian(1001354465)

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.

   Answer:
   This claim is true. We know that $h(k)$ as a hash function, will give output from 0 to m-1 (a key that does not exist in the table), and we have i = 0, 1, 2..., so $(h(k) + i)$ will be a sequence of consecutive numbers by adding i each time. So, the result of $(h(k) + i) \bmod m$ will also be a sequence of consecutive numbers, so that it will occupy the entire hash table until there's no free space.
   Alternatively, we know that $h(k)$ will give output from 0 to m-1, and if a slot is already occupied, we just keep adding one and move to the next slot, so eventually, 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.

   (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.

   Answer: This claim is false.
   $h(k, i) = (h(k) + i^2) \bmod m$
   $= (h(k) \bmod m + i^2 \bmod m) \bmod m$ (by the property of module)
   $=(h(k) + (i \bmod m)^2) \bmod m$ (by the property of module)
   let j $= i \bmod m = 1, 2, 3, 4, 5, 6$
   So we have $(h(k) + j^2 \bmod m) \bmod m$

Then, let's see the value of j^2 mod m where $m = 7$
j^2 = 0,1,4,9,16,25,36 and j^2 mod m = 0,1,4,2,2,4,1

Suppose $h(k)$ is a bad hash function, let $h(k) = 0$,
Then, we have $(h(k) + j^2 \bmod m) \bmod m$
$=(0 + j^2 \bmod m) \bmod m$ where $j^2 \bmod m = 0, 1, 2, 4$.
So, $(0 + j^2 \bmod m) \bmod m$ won't be able to produce a sequence of consecutive numbers that will occupy each slot in the hash table. The image of $h(k, i) = (h(k) + i^2) \bmod m$ is $0, 1, 2, 4$.
Therefore, 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.
Answer:
We will prove that there will be more than half of size free spaces, so it is guaranteed that if the hash table contains less than $\lfloor m/2 \rfloor$ keys, the insertion will be successful.
Let p be the number of keys in the hash table that is a prime number greater than 3.
$h(k, i) = (h(k) + i^2) \bmod p$
$= (h(k) \bmod p + i^2 \bmod p) \bmod p$ (by the property of module)
$=(h(k) + (i \bmod p)^2) \bmod p$ (by the property of module)
let j $= i \bmod p = i \bmod p = 1, 2, \ldots, p - 1$
So we have $(h(k) + j^2) \bmod p$
$=(h(k) \bmod p + j^2 \bmod p) \bmod p$

Now we want to make sure that $(h(k) \bmod p + j^2 \bmod p) \bmod p$ will produce more than half size of hash table number of distinct keys, therefore if the hash table contains less than $\lfloor m/2 \rfloor$ keys, the insertion will be successful. We also know that h(k) here is a unknown hash function, so it could be really bad that only produces one key. So we have to make sure that $j^2 \bmod p$ will have more than half size of hash table number of distinct keys. We will need to check is $j^2 \bmod p$ has more than $\frac{p}{2}$ values.

j^2 = {0,1,4,9,16,$\ldots$,$\frac{p-1}{2}^2$, $\frac{p+1}{2}^2$,$\ldots$,$(p-2)^2$,$(p-1)^2$}
Here, note that 1 mod $m = (p-1)^2 \bmod m$
2 mod $m = (p-2)^2 \bmod m$ and so on...
Based on above pattern, we got $i^2 = (p-i)^2 \bmod m$
$j^2 \bmod p \in$ {0 mod p,1 mod p,$\ldots$,$\ldots$,((p-1)/2)^2 mod p}
So, there are (1+p)/2 number of elements in $j^2 \bmod p = \lceil m/2 \rceil$
Now we know that we will have more than half of size values produced by $j^2 \bmod p$, we will now check that if elements in $j^2 \bmod p$ are distinct.

We use contradiction to prove:
Suppose $j_1, j_2 \in \{0, \ldots, \frac{p-1}{2}\}, where$ j$_1 \neq$j$_2$, and $j_1 \bmod p = j_2 \bmod p$
we get $j_1 = j_2 \bmod p$ (suppose $j_1 > j_2$)
According to the property of module, $j_1^2 = kp + j_2^2$ where k is a positive integer.
we get $j_1^2 - j_2^2 = kp = (j_1 + j_2)(j_1 - j_2) = kp$
But p is a prime number, and $j_1, j_2 \in \{0, \ldots, \frac{p-1}{2}\}, so$ j$_1$+$j_2 < p$ and $j_1 - j_2 < p$, so the product of $(j_1 + j_2)(j_1 - j_2)$ is impossible to be kp.

So by contradiction, $j_1 \bmod p \neq j_2 \bmod p$, Therefore, the elements in $j^2 \bmod p$ are distinct.
Conclusion: if the hash table contains less than $\lfloor m/2 \rfloor$ keys, then 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:
from Dan:
(a) Aggregate method: The worst-case sequence of n operations is the sequence with $n - (floor(logn) + 1)$ INSERT followed by $floor(logn) + 1$ PRINT-AND-CUT. The total cost of the INSERT operations is $n - (floor(logn) + 1) <= n$, and the total cost of the PRINT-AND-CUT operations is
$\sum_{i=0}^{floor(logn)} n/2^i$ ¡= $n \sum_{i=0}^{\infty} 1/2^i = 2n$.
Therefore, the total cost of the sequence is upper-bounded by $n + 2n = 3n$, and the amortized cost per operation is upper-bounded by $3n/n = 3$.

(b) Accounting method: Charge each INSERT with 3 dollars. The first dollar is for the cost of the insertion, the second dollar is for covering each element's own cost of print. The third dollar is a recharge dollar, i.e., the half of the elements that are cut in a PRINT-AND-CUT operation use the recharge dollar to recharge the half of the elements that are not cut, so that their next cost of print is covered. With the operations charged in this way, all cost is any sequence of INSERT and PRINT-AND-CUT operations is fully covered. Therefore, the amortized cost per operation is upper-bounded by 3.

from Jianhao:
(a)
Worst-case sequence: first execute INSERT n times, where k = n. Second start to execute PRINT-AND-CUT until k = 0.

Aggregate method:
cost of INSERT = 1
cost of PRINT-AND-CUT = k
total cost of execute INSERT n times = n

times that PRINT-AND-CUT can execute after n times INSERT = log n
sequence of cost of PRINT-AND-CUT: n, n/2, n/4,..., 1.
total cost of execute PRINT-AND-CUT log n times = $n*(1-0.5^{logn})/(1-0.5) = 2n*(1-2^{logn^{-1}}) = 2n-2n*1/n = 2n-2$

total cost of n times INSERT and log n times PRINT-AND-CUT = n + 2n - 2 = 3n - 2

total operation times = n + log n

amortized cost per operation = (3n - 2)/(n + log n), when n become infinity, log n decrease to 0, therefore, the upper bound of amortized cost per operation = 3n/2 = 3.

(b)
let the Charge amount be x.
1. when x = \$1, each INSERT will cost \$1, then after n times INSERT, we have no money left, there are no money for PRINT-AND-CUT.

2. when x = \$2, each INSERT will cost \$1, then after n times INSERT, we have \$n remaining, and then after the first PRINT-AND-CUT, which costs \$n to do so, we have n - n + 1 = \$1, there are no money for next PRINT-AND-CUT.

3. when x = \$3, each INSERT will cost \$1, then after n times INSERT, we have \$2n remaining
and after the first time PRINT-AND-CUT, we have 2n - n + 1 = \$(n + 1) remaining,
after the second time PRINT-AND-CUT, we have n + 1 - n/2 + 1 = \$(n/2 + 2) remaining,
after the third time PRINT-AND-CUT, we have n/2 + 2 - n/4 + 1 = \$(n/4 + 3) remaining,
therefore, we always have enough money for the next PRINT-AND-CUT operation.
Therefore, the charged amount for each insert 3, will be the upper-bound on the amortized cost per operation.

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.

from Dan:
Use a hash table. Make sure that two pizzas are hashed to the same slot if it is possible for them to be equivalent. For two pizzas that are hashed to the same slot, try all ways of rotations to make sure if the two are actually equivalent. Increment the counter if a new kind of pizza is found (not equivalent to any existing pizza in the hash table).

In the average case, each insert of pizza takes $O(1)$ time. Comparing all rotations is also $O(1)$. Therefore, going through n pizzas takes $O(n)$ time in the average case.

Explanation:
To build a hash table, we create a new LinkedList class, the Node class, and its hash function. To check whether two pizzas are the same kind, we build a checkEquivalent function. In the main function, first, we create a list with None objects depends on the number of pizzas. This step will take $\mathcal{O}(n)$ time complexity. Then, for each pizza in the given list, use hash function to determine its index of the bucket. Next, If the bucket is empty, just put the pizza at the head of the linked list and stored the linked list in the bucket. However, if it is not empty, check the head pizza of the linked list, if they are same, then insert this pizza to the head of the linked list, otherwise, use linear probing to find an empty bucket for this pizza. Since the number of the buckets equals to the number of pizzas, this pizza is guaranteed to have an empty bucket. Finally, the number of kinds is just the number of linked list in the buckets.Since every loop takes $\mathcal{O}(n)$, and there are three loops, the total time complexity is still $\mathcal{O}(n)$