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.

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: $h(k,i) = (h(k) + i) \text{mod } 7 \qquad i = 0, 1, 2, 3, ...$

a) This claim is true. This is because:

say we have some key k_j; where h(k_j) = h_j and there is atleast one free slot. By taking mod 7 of ($h_j + i$), we get only values from 0 to $m - 1$ which are valid indexes. As i increases by 1 each time, this allows us to iterate through every position of the list. Because once we get to index $m - 1$ (where n is our current i value):

$$h_j + n \text{ mod } 7 = m - 1$$

Incrementing n by one results in:

$$h_j + (n + 1) \text{ mod } 7 = 0$$

Which has in effect looped around the list to the first element. As we can always get to the next position in the list by incrementing i, and the fact that at the last position, incrementing i brings us to the front of the list, we can always insert a key if a spot is free.

b) This claim is false.

Say we have an array that has been filled completely except at position 3. We now insert a key k_j. Say for simplicity h(k_j) = 0. This means (h(k_j), i) = (h(k_j)+i^2) mod 7, for $i = 0, 1, 2 \ldots$ simplifies to i^2mod 7 for i = 0,1,2 ...

If a solution to i^2mod7 = 3 exists, we can insert into the position 3 (our only free position). If no solution exists, then we can't insert.

we will use Euler's Criterion to check if solution exists:

$$a^{\frac{p-1}{2}} \equiv \begin{cases} 1(\text{mod p}) & \text{if there is an integer x such that a} \equiv x^2(\text{mod p}) \\ -1(\text{mod p}) & \text{if there is no such integer} \end{cases}$$

(our p is odd and prime, and our a and p are coprime, so we can use this method.)

$3^{\frac{7-1}{2}} = 3^3 = 27$

$27 \equiv -1 \text{ (mod 7)}$

$\Rightarrow$ no solution exists

As no solution exists, then we cannot insert, which verifies our assertion that the claim is false.

c)

`Sources used: https://en.wikipedia.org/wiki/Quadratic_probing , under "Limitations"`

Proof by contradiction:

Assume that we have less than $\left[\frac{m}{2}\right]$ keys and that our next insertion fails. $h(k,i) = (h(k) + i^2)$ mod m for i = 0,1,2... Say our failed insertion is of key k. This means there are at most $\left[\frac{m}{2}\right] - 1$ distinct probing locations. So eventually, as we continue probing, we loop around and probe the same spot. Let $x, y \in \mathbb{N}$ such that $0 \leq x, y < \left[\frac{m}{2}\right]$ and $x \neq y$ (these restrictions on x and y were taken from the source provided). This means:

$$h(k) + x^2 \text{ mod } m = h(k) + y^2 \text{ mod } m$$

(Which is to say different values of i has resulted in same location).

$$\implies x^2 \text{ mod } m = y^2 \text{ mod } m$$

$$\implies x^2 - y^2 \text{ mod } m = 0$$

$$\implies (x + y)(x - y) \text{ mod } m = 0$$

The following argument was also taken from the source:

Because m is an odd prime greater than 3, this means that m must divide $(x + y)$ or $(x - y)$ evenly (Theorem). But because $0 \leq x, y < \left[\frac{m}{2}\right]$ and $x \neq y$.

$$\implies 0 < (x + y), (x - y) < m$$

But if $(x + y)$ and $(x - y)$ are both between 0 and m (exclusive) then it follows that they cannot be evenly divided by m. Hence a contradiction. Therefore the insertion must 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)Let C_i be the cost of i 'th insertion.

$$c_i = \begin{cases} 1 & if \quad i \leq n - [log_2 n] + 1 \\ k & \text{otherwise} \end{cases}$$

Worst sequence is n - $[log_2 n]$+1 inserts, followed by $[log_2 n]$-1 print-cuts.

$\Gamma(n) = \sum_{i=1}^{n} C_i$

$= \sum_{i=1}^{n-[log_2 n]-1} 1 + (n - [log_2 n] + 1) \sum_{i=0}^{[log_2 n]-1} 2^{-i}$

$= n - [log_2 n] - 1 + (n - [log_2 n] + 1)\frac{n-2}{n}$

$= \frac{n^2 - n[log_2 n] - n + n^2 - 2n - n[log_2 n] + 2[log_2 n] + n - 2}{n}$

$= \frac{2n^2 - 2n[log_2 n] - 3n + 2[log_2 n] - 1}{n}$

$\leq \frac{2n^2 + 2[log_2 n]}{n} = 2n + \frac{2log_2 n}{n}$

Notice as n approaces inf, $\frac{2log_2 n}{n}$ approaches 0. So as we are finding the upperbound, we can ignore this term.

$\Gamma(n) = 2n$

Cost per operation : $\frac{2n}{n} = 2$

b) The cost per operation is 2. If We do n [logn] + 1 insertions, we have n - [logn] + 1 tokens remaining for the print-and-cut operations.

The cost of print-and-cut is :

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

$= \frac{n^2-2n-n[log_2 n]+2[log_2 n]+n-2}{n}$

$= \frac{n^2-n-n[log_2 n]+2[log_2 n]-2}{n}$

$= n - 1 - [log_2 n] + \frac{2[log_2 n]}{n} - \frac{2}{n}$

notice as n $\to$ inf, $\frac{2[log_2 n]}{n} \to 0$ and $\frac{2}{n} \to 0$ so we can simplify to :

n - $[log_2 n] - 1$

Which is less than how many tokens we set asside for its, Therefore the cost of 2 covers all operations.

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.

   SOLUTION:

   Our solution works by using a hash function that sums up the integer rankings of the pizza. By doing this, we ensure that each equivalent pizza will always end in the same bucket. We check to see if a bucket is empty, in that case we insert the pizza in the bucket and increase our total number of unique pizzas. In the case we try to insert a pizza into a non-empty bucket, we check to see if it is unique to all other pizzas in the bucket, if it is, we add it to the bucket

and increase the total number of unique pizzas. After we have done this for all pizzas, we are left with the number of unique pizzas and return it.

We check to see if two pizzas are identical by concatenating one of the pizzas with itself, we then check every 5 item sublist in the concatenated list and compare it to the non-concatenated pizza. The idea to concatenate a sequence with itself to contain every possible rotation was found here: https://stackoverflow.com/questions/18330401/offset-independent-hash-function

Correctness:

We begin first by creating a list of empty buckets equal to the number of pizzas inputted. We then calculate the hash of each pizza. We calculate the hash of each pizza by summing the rankings of each slice. This ensures that every rotation of the same pizza will end up at the same bucket. Each pizza has two paths:

path 1:
There is no pizza in the current bucket, in this case we add the current pizza to the bucket, and increment our total number of unique pizzas by one.

path 2:
There is atleast one pizza in the current bucket. In this case we call our function unique on the current pizza to be added, and the bucket we will add to if the pizza is unique. The function unique compares the current pizza to every possible rotation of each pizza in the bucket already. The function returns true if the pizza is unique. If the pizza is unique, the current pizza is added to the bucket, and the number of unique pizzas increases by one. In the case that it is not unique, we do nothing and move on to the next pizza.

Once we have done this for all pizzas inputted, we return the total number of unique pizzas. This is a valid return based on the fact we only increment the total number of unique pizzas when it is a unique pizza.

Run Time:

We assume Simple Uniform Hashing. Let n represent the number of pizzas. Because we create n buckets, this means that our load factor is: $\frac{n}{n} = 1$. This means the average length of each bucket is 1. If this is the case, then each insertion took constant time. Each insertion takes constant time because it inserts to an empty bucket. This operation involves calling hash function (constant), checking value of an index in a list (constant), setting a value in a list (constant), and incrementing an integer (constant). As each operation takes constant time (adding constant operations together results in a constant), and there are n operations, the run time of this function is linear.