Assignment 6

Due: March 3, 2020

Kanyid, Bradon
bradon.kanyid@pdx.edu

Reimer, Daniel
daniel.reimer@pdx.edu

Problem 1: Longest-Probe Bound for Hashing Suppose we use an open-addressed hash table (section 11.4 in CLRS) of size m to store $n \leq \frac{m}{2}$ items.

(a) [10 points] Assuming simple uniform hashing, show that for $i = 1, 2, ..., n$, the probability is at most 2^{-k} that the ith insertion requires strictly more than k probes.

Solution

Under the assumption of uniform hashing, the likelihood for any insertion to have a colision is n/m where n is the number of probes is m is the size. We know that our open-addressed table has $n < m/2$, so the load factor is known to be strictly $< 1/2$.

The probability of needing more than k probes will be some value $(1/2) * (1/2) * ... * (1/2)$ for k times, or $(1/2)^k$. So $(n/m)^k < (1/2)^k = 2^{-k}$.

(b) [10 points] Assuming simple uniform hashing, show that for $i = 1, 2, ..., n$, the probability is $O(\frac{1}{n^2})$ that the ith insertion requires more than $2 \lg n$ probes.

Solution

Under the assumption of uniform hashing, the likelihood for any insertion to have a colision is n/m where n is the number of probes is m is the size. We know that our open-addressed table has $n < m/2$, so the load factor is known to be strictly $< 1/2$. Similar to 1a, a probe that takes more than $2 \lg n$ probes means $(1/2) * (1/2) * ... * (1/2)$ for $2 \ln n$ times, or stated another way, $(1/2)^{2 \lg n}$ times.

$$\begin{aligned}(1/2)^{(2 \lg n)} &== 2^{(-2 \lg n)} \\ &== 1/(2^{\lg n})^2 \\ &== 1/n^2\end{aligned}$$

Therefore the probability of needing more than $2 \lg(n)$ probes will be $1/n^2$.

Problem 2: Building a Queue using Stacks It is possible to build a *queue* (FIFO) using two stacks. Assume that the stacks have three operations, *push*, *pop*, and *isEmpty*, each with cost 1. A queue can be implemented as follows:

- *enqueue:* push item x onto stack 1
- *dequeue:* if stack 2 is empty then pop the entire contents of stack 1 pushing each element in turn onto stack 2. Now pop from stack 2 and return the result.

A conventional worst-case analysis would establish that *dequeue* takes $O(n)$ time, but this is clearly a weak bound for a sequence of operations, because very few dequeues will actually take that long. To simplify your analysis only consider the cost of the push and pop operations.

(a) [10 points] Using the aggregate method show that the amortized cost of each *enqueue* and *dequeue* is constant.

Solution

Even though a single dequeue operation can be quite expensive (in the case where many enqueues have happened in a row without a dequeue), any sequence of n enqueue and dequeue operations on an initially empty pair of stacks can cost at most $O(n)$.

Enqueue is constant work, always pushing one item onto the first stack. Dequeue has two paths:

- If stack 2 is non-empty, the work is constant and pops one element from the second stack.
- If stack 2 is empty, there are n pops and pushs from stack 1 to stack 2, where n is the number of elements on stack 1 prior to the operation.

We can dequeue each object from the stack at most once for each time we have enqueued it onto the stack. Therefore, the number of times that dequeue can be called on a nonempty stack is at most the number of enqueue operations, which is at most n. For any value of n, any sequence of n enqueue and dequeue operations takes a total of $O(n)$ time. The average cost of an operation is $O(n)/n == O(1)$.

(b) [10 points] Using the accounting method show that the amortized cost of each *enqueue* and *dequeue* is constant.

Solution

We are given the actual costs of the three operations, push, pop, and isEmpty, each with cost 1. We will define our own amortized costs for the larger operations:

- *Enqueue:* amortized cost of 5
- *Dequeue:* amortized cost of 1

As in 2a, there's 3 meta-operations that can happen:

- *Enqueue*
- *Dequeue (Empty Stack 2)*
- *Dequeue (Non-Empty Stack 2)*

Let's break down the cost of each operation.

Enqueue:

- 1 push to stack 1

An enqueue uses 1 credit of the 5 credits it is given, leaving 4 credits remaining.

Dequeue (Empty Stack 2):

- 1 isEmpty (paid by the item being dequeued)
- 1 pop from stack 1 (paid by every item on stack 1)
- 1 push to stack 2 (paid by every item on stack 1)
- 1 pop from stack 2 (paid by the item being dequeued)

The cost per item is 2 for every item moving from stack 1 to stack 2, leaving a credit of 2 for those items. The item destined to be dequeued has also paid for the isEmpty operation, as well as the cost of the pop from stack 2. The final result is that stack 1 is empty, and the remaining items on stack 2 have a credit of 2, which brings us to the last case...

Dequeue (Non-Empty Stack 2):

- 1 isEmpty
- 1 pop from stack 2

A non-empty stack 2 will quickly remove the top item from stack 2, and that item pays its two remainig credits for the two operations.

Overall

The general idea here is that every item enqueued will go through 4 more operations to be dequeued. At some point, it will be popped from stack 1, pushed to stack 2, popped from stack 2, and be the item that required an isEmpty operation. Therefore overall, the amortized cost is $5 \in O(1)$.

(c) [10 points] Using the potential method show that the amortized cost of each *enqueue* and *dequeue* is constant.

Solution

This solution follows much of the intuition of the accounting method. However, instead of each item keeping its own credits, we come up with an equation that describes the system's overall credit.

Instead of having an item credit of 4, enqueuing increases the potential of the system by 4, for example. That's enough credit to maintain the lifetime operations that any item goes through: pop from stack 1, push to stack 2, pop from stack 2, and isEmpty check concurrent with pop from stack 2.

You can describe the potential function as $p(j, k) = 4j + 2k$, where j is the number of items on stack 1, and k is the number of items on stack 2.

Based on this potential function, the effect of any meta-operation is:

- *Enqueue:* potential augmented by 4

- *Dequeue (Empty Stack 2):* potential augmented by -2j-2
- *Dequeue (Non-Empty Stack 2):* potential augmented by -2

Enqueue:

- 1 push to stack 1

An enqueue costs 1 push to perform, and increases potential by 4, and so has an amortized cost of 5, which is constant $O(1)$

Dequeue (Empty Stack 2):

- 1 isEmpty
- j pops from stack 1
- j pushes to stack 2
- 1 pop from stack 2

If the dequeue happens when the stack is empty, you pay the operation cost of $1 + 2j + 1$. The amortized cost of this operation is 0, because it changes the potential by $-2j - 2$. This is because you decreased potential by 4 for every item you removed from stack 1, but increased potential by 2 for every item you added to stack 2, and also descreased overall potential by 2 for the actual dequeue pop from stack 2 and the isEmpty operation.

The potential function shows that potential changes by $-2j - 2$, and we also showed there are $2j + 2$ operations done in a dequeue with an empty stack. So the amortized cost of the operation is $1 + 2j + 1 - 2j - 2 == 0 \in O(1)$

Dequeue (Non-Empty Stack 2):

- 1 isEmpty
- 1 pop from stack 2

The potential function shows that one item is removed from stack 2, so k decreases by 1, for an overall potential change of -2. There are 2 operations done in a dequeue with a non-empty stack. So the amortized cost is $2 + -2 == 0 \in O(1)$

Overall:

All operations are amortized to $O(1)$.