15-451/651 Algorithms, Spring 2019
Recitation #2 Worksheet

Recap of this week's lectures:

- Amortized analysis using the banker's method and the potential function method.
- Examples: Binary counter, and growing/shrinking arrays
- Union-find data structures: list-based and tree-based approaches.
- Recap of Minimum spanning tree algorithms: Kruskal, Prim, Boruvka

---

Slower resizing: Suppose we are growing a table as in lecture. Instead of doubling, we do the following: we begin with an array of size 1. The first time we resize, we add 2 to the length. The second time, we add 3. The third time we add 4. And so on. The cost for resizing the array is the size of the new array. What is (in Θ notation) the amortized cost per operation for n pushes?

Solution: $\Theta(\sqrt{n})$.

If the number of resizes is k, the size of the final cache is $1+2+\ldots+(k+1) = \binom{k+2}{2} = \Theta(k^2)$. Since our final cache has size $\Theta(n)$, the total number of resizes is $\Theta(\sqrt{n})$. Hence the total cost of resizing is $\Theta(n\sqrt{n})$, giving an $\Theta(\sqrt{n})$ bound on the amortized cost (i.e., cost per push).

Another way to argue is to just sum up. The total resizing cost $= 1+(1+2)+(1+2+3)+\ldots+(1+2+3+\ldots+k)$ where $k = \Theta(\sqrt{n})$ is the smallest integer such that $1+2+3+\ldots+k \geq n$ (i.e., $n \approx k^2/2$).

Binary Counter Revisited: Suppose we are incrementing a binary counter, but instead of each bit flip costing 1, suppose flipping the i^{th} bit costs us 2^i. (Flipping the lowest order bit `A[0]` costs $2^0 = 1$, the next higher order bit `A[1]` costs $2^1 = 2$, the next costs $2^2 = 4$, etc.) What is the amortized cost per operation for a sequence of n increments, starting from zero?

Solution: $O(\log n)$. The idea is simple. We flip `A[0]` each time, so pay n over n operations. We flip `A[1]` every other time, so pay $\leq 2 \times n/2 = n$ over n operations, and so on, until `A[`$\lfloor \log_2 n \rfloor$ which gets flipped once for a cost of at most n. Hence $O(n \log n)$ in total, or $O(\log n)$ per operation.

Binary Counter Re-Revisited: Suppose we are incrementing a binary counter, but instead of each bit flip costing 1, suppose flipping the i^{th} bit costs us $i+1$. (Flipping the lowest order bit `A[0]` costs $0+1 = 1$, the next higher order bit `A[1]` costs $1+1 = 2$, the next costs $2+1 = 3$, etc.) What is the amortized cost per operation for a sequence of n increments, starting from zero?

Solution: At most \$4. We will maintain the invariant that any 1 in the binary counter at position i will have \$$(i+3)$ on it. To begin, we flip the lowest order bit to 1 and put \$3 on it, and use

\$1 to pay for the flip. Now when incrementing, suppose there is a zero followed by k ones at the end. Now for each of those ones (say at position i), we use \$$(i+1)$ to pay for the flip, and pick up the remaining two dollars. Finally, we have $2k$ dollars we have picked up, and \$4 more from the current increment. We use \$$(k+1)$ to pay for flipping the zero to one, and put down $(2k+4)-(k+1)=k+3$ dollars on this bit. This maintains the inductive invariant.

For a potential function based proof, if the current counter value is $B = b_{n-1}\cdots b_2 b_1 b_0$ you can use $\Phi(B) = \sum_{i=0}^{n-1} b_i \cdot (i+3)$, and show that (a) $\Phi(0)=0$, Φ is always non-negative, and that the per-step amortized cost $ac_t = c_t + \Phi_t - \Phi_{t-1}$ is always bounded by 4. In this case it's the same proof as above, just packaged differently.

Yet another way to argue: We flip `A[i]` every 2^i operations, so we pay $(i+1)n/2^i$ over n operations for `A[i]` where $0 \le i \le \lfloor \lg n \rfloor$. Then, we can sum up the costs:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} (i+1)\frac{n}{2^i} \le n \sum_{i=1}^{\infty} i(1/2)^{i-1} = \frac{n}{(1-1/2)^2} = 4n$$

(We use the fact that $\sum_{i=0}^{\infty} i x^{i-1} = 1/(1-x)^2$ where $|x|<1$. Please try to prove this for yourself. If you fail please ask on piazza.) Then the total cost is at most $4n$, so the amortized cost is at most 4.

Another Dictionary Data Structure: A "dictionary" data structure supports fast insert and lookup operations into a set of items. Note that a sorted array is good for lookups (binary search takes time only $O(\log n)$) but bad for inserts (takes linear time), and a linked list is good for inserts (takes constant time) but bad for lookups (takes linear time). Here is a simple method that takes $O(\log^2 n)$ search time and $O(\log n)$ amortized cost per insert.

Here, we keep a collection of arrays, where array i has size 2^i. Each array is either empty or full, and each is in sorted order. However, there will be no relationship between the items in different arrays. The issue of which arrays are full and which are empty is based on the binary representation of the number of items we are storing. For example, if we had 11 items (where $11 = 1+2+8$), then the arrays of size 1, 2, and 8 would be full and the rest empty, and the data structure might look like this:

```
A0:  [5]
A1:  [4,8]
A2:  empty
A3:  [2, 6, 9, 12, 13, 16, 20, 25]
```

Lookups. How would you do a lookup in $O(\log^2 n)$ worst-case time?

Solution: Just do binary search in each occupied array. In the worst case, this takes time $O(\log(n) + \log(n/2) + \log(n/4) + \ldots + 1) = O(\log^2 n)$.

Inserts. How would you do inserts? Suppose you wanted to insert an element, you will have 12 items and $12 = 8+4$, you want to have two full arrays in $A2$ and $A3$ and the rest empty. Suggest a way that, if you insert an element 11 into the example above, gives:

```
A0:  empty
A1:  empty
A2:  [4, 5, 8, 11]
A3:  [2, 6, 9, 12, 13, 16, 20, 25]
```

(Hint: merge arrays!)

Solution: Create an array of size 1 that just has this inserted number in it. We now look to see if `A0` is empty. If so we make the new array be `A0`. If not we merge our array with `A0` to create a new array (which in the example would be `[5, 11]`) and look to see if `A1` is empty. If `A1` is empty, we make this array be `A1`. If not, we merge this with `A1` to create a new array and check to see if `A2` is empty, and so on.

Cost of Inserts: Suppose the cost of creating an array of length 1 costs 1, and merging two arrays of length m costs $2m$. So, the above insert had cost $1 + 2 + 4$. Inserting another element would cost 1, and the next insert would cost $1 + 2$.

What is the amortized cost of n inserts?

Solution: With this cost model defined above, it's exactly the same as the binary counter with cost 2^k for the k^{th} bit. So the amortized cost is $O(\log n)$.