Problem Set 04

Mostafa Touny

Contents

Exercises

Ex. 1

done

Ex. 2

The amortized cost of n operations is upper-bounded by

$$
\begin{aligned}
n &+ \sum_{i=1}^{\lfloor \lg n \rfloor} 2^i \\
&= n + \frac{2(1 - 2^{\lfloor \lg n \rfloor})}{1-2} \\
&\leq n + \frac{2(1-n)}{-1} \\
&= n - 2 + 2n \\
&= 3n - 2 \\
&= \mathcal{O}(n)
\end{aligned}
$$

So the amortized cost of one operation is $\frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$.

Ex. 3

We assign the following amortized costs:

- ith operation isn't a power of 2 $\rightarrow$ 4
- ith operation is an exact power of 2 $\rightarrow$ 0

We prove for each operation 2^i, There's a sufficient balance for it. For $i \geq 2$, There are exactly $2^{i-1} - 1$ non-power operations before 2^i and after 2^{i-1}. It sufficies to show $4(2^{i-1} - 1) \geq 2^i$ which can trivially be proven by induction.

Observe amortized cost $= 4n - 4\lfloor \lg n \rfloor \geq n - \lfloor \lg n \rfloor + 2n \geq n - \lfloor \lg n \rfloor + \sum_{i=1}^{\lfloor \lg n \rfloor} 2^i =$ actual cost. Note by the geometric series $\sum_{i=1}^{\lfloor \lg n \rfloor} 2^i = \frac{2(1 - 2^{\lfloor \lg n \rfloor})}{1-2} \leq 2n$

The amortized cost of n operations is $\mathcal{O}(n)$, and hence the amortzed cost of one operation is $\mathcal{O}(1)$.

Ex. 4

Define potential function $\Phi(D_i)$ to be the number of 1-bits in the binary representation of i. Note $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ which suffices to show the validity of our definition.

Observe the amortized cost of operations:

$$c_i' = \left\{ \begin{array}{ll} i+1-i=1, & \text{i is a power of 2} \\ 1+1=2, & \text{if i is odd} \\ 1+\Delta\Phi(D_i) \leq 1, & \text{if i is even but not power of 2} \end{array} \right\} \tag{1}$$

When i is odd, it has one additional 1-bit over even i-1, due to the right most bit being only flipped from 0 to 1. When i is even, then i-1 is odd, and at least one 1-bit is flipped to zero and at most one 0-bit is flipped to 1. So $\Delta\Phi(D_i) \leq 0$. When $i = 2^k$, a power of two, then $\Phi(D_i) = 1$ because there's exactly one 1-bit. Also, i-1 contains exactly i 1-bits, So $\Phi(D_{i-1}) = i$.

In all cases, the amortized cost of a single operation is $\mathcal{O}(1)$.

Ex. 5

done

Ex. 6

Each element of the array needs to be compared with the *pivot* only once to conclude whether it is greater or less than it.

Ex. 7

Since $0 < \alpha \leq \frac{1}{2}$ branching $1-\alpha$ is greater or equal than branching α. Maximum depth is $\lg_{\frac{1}{1-\alpha}} n = \frac{\lg n}{\lg \frac{1}{1-\alpha}} = \frac{\lg n}{\lg 1 - \lg(1-\alpha)}$ and minimum depth is $\lg_{\frac{1}{\alpha}} n = \frac{\lg n}{\lg \frac{1}{\alpha}} = \frac{\lg n}{\lg 1 - \lg \alpha}$. The fact $\lg 1 = 0$ concludes the intended result.

Ex. 8

Failed to solve.

Through the same reasoning of establishing upper-bound, we derived a lower-bound of $\Omega(\lg n)$.

Problems

Prob. 1

The obvious FIFO queue satisfies the problem's requirements. Think of a list of numbers where integers are *enqueued* to left and *dequeued* from right.

A *list.min* variable is maintained whenever a new integer is added, Checking whether it's less than *list.min* and updating accordingly. Whenever *dequeue* is called, we check whether removed integer is equal to *list.min*. If not, no additional work is done. If yes, we know by the distinctness of integers, that the *list.min* is removed from the list, and hence it must be updated. A linear scan is implemented to update *list.min*.

While the worst-case analysis of *dequeue* is linear, That worst case of removing the *list.min* happens in proportion to the number of integers enqueued, which in turn allows us to conclude an amortized cost of $\mathcal{O}(1)$.

The central key idea is to loop only once on each element, from left to right, storing in each *element.min*, The minimum integer of the sub-array starting from left-most to current element's position. Now whenever we need to loop again to find *list.min*, We do not loop on already-visited elements, but only on newly inserted elements. We assign *list.min* to be the minimum integer of that new sub-array. Observe we can conclude the minimum of the whole list, from *list.min* and right-most *element.min* stored in visited elements. It's basically *min(list.min, element.min)*.

We continue in this manner untill all visited elements are dequeued. Then we are left with a list of totally no visited elements, and *list.min* is the minimum integer of the whole list.

a

- **element** contains *int* holding the integer value and *min* storing the minimum element of a sub-array.
- **list** contains *min* indicating the minimum integer of the unstamped sub-array. That, besides *elements* aforementioned.

b

- **minAllElements** Loop from left to right on the whole list, Maintaining the minimum of the sub-array from left-most to currently visiting element, and storing it in each *element.min*. Reset *list.min* to $+\infty$ so that it considers only newly inserted elements.
- **Enqueue** Append element to the left of the list. If it's less than *list.min*, Update *list.min* to it.
- **Find-Min**
  - (1) No element is visited in a *minAllElements* call before.
    - return *list.min*.
  - (2) Some elements are visited in a *minAllElements* call before.
    - return *min(list.min, element.min)*, where the element here is the right-most one.
- **Dequeue** Assign *localMin=Find-Min()*, and remove the element. For case (1), if removed element is equal to *localMin*, *minAllElements* is called.

c

We skip a proof by invariance is it seems unnecessarily. We believe our discussion suffices to convince the reader our design covers all cases.

d

Trivially, *Enqueue* and *Find-Min* are $\mathcal{O}(1)$, and *minAllElements* is $\theta(n)$. *Dequeue*'s worst-case is $\theta(n)$ due to the call of *minAllElements*. So, m operations are upper-bounded by $\omega(m^2)$.

The goal now, by the *accounting* method, is to show we can pay *minAllElements* by an amortized cost of 2 for *Enqueue*. Note we cannot visit an element unless it's enqueued. We already discussed each element is going to be visited by *minAllElements* at most once, Hence the additional credit for each element accommodates the payment.

Now we have all desired operations to have an amortized cost of $\mathcal{O}(1)$, and a sequence of m operations costs $\mathcal{O}(m)$.

Prob. 2

a

The event is logically equivalent to, assuming x_i is not the pivot the next recursive call containing x_i has a subarray of size at most $3m/4$.

Consider the array's elements ordered as $q_1 < q_2 < \cdots < q_m$. There are three cases for which the event occurs:

- (i) The pivot $z \in \{\lceil m/4 \rceil, \ldots, \lfloor 3m/4 \rfloor + 1\}$. Then x_i is always in a subarray of size at most $3m/4$.
- (ii) $z \in \{1, \ldots, \lceil m/4 \rceil - 1\}$, and x_i is in the left subarray.
- (iii) $z \in \{\lfloor 3m/4 \rfloor + 2, \ldots, m\}$, and x_i is in the right subarray.

We ignore (ii) and (iii) and prove (i) concludes the desired lower-bound of probability 1/2.

Since the pivot is randomly selected, we know the probability of q_i being the pivot is $1/m$. There are exactly $\lfloor 3m/4 \rfloor + 1 - \lceil m/4 \rceil + 1$ elements. So the probability is:

$$\geq \frac{1}{m}\,(\left\lfloor \frac{3m}{4} \right\rfloor + 1 - \left\lceil \frac{m}{4} \right\rceil + 1)$$

$$\geq \frac{1}{m}\,(\frac{3m}{4} - \frac{m}{4})$$

$$= \frac{1}{m} \cdot \frac{m}{2} = \frac{1}{2}$$

b

Assume the algorithm lasted for iteration $3(2+\frac{1}{\log_2 4/3})\log_2 n = 3(\alpha + c)\log_2 n$. By the instructor's claim and exercise a, We know the array size is reduced by a factor of at most $3m/4$ for at least $\frac{1}{\log_2 4/3}\ \log_2 n = \log_{4/3} n$ times. Thus the array size is at most $\frac{n}{(4/3)^{\lg_{4/3} n}} = 1$ and the algorithm terminates. Therefore with probability at least $1 - \frac{1}{n^2}$, The number of comparisons is logarithmic for $d \leq 3(2+\frac{1}{\log_2 4/3})$.

c

Definition 1. Let k_i denote the event, that the total comparisons of x_i with pivots is at most $d \lg n$.

Lemma 2. $prob[\neg k_1 \vee \neg k_2 \vee \cdots \vee \neg k_n] \leq \frac{1}{n}$.

Immediately follows by the fact $prob[\neg k_i] = \frac{1}{n^2}$ and the union bound. Note $\frac{1}{n^2} + \cdots + \frac{1}{n^2} = n\frac{1}{n^2} = \frac{1}{n}$

Corollary 3. $prob[k_1 \wedge \cdots \wedge k_n] \geq 1 - \frac{1}{n}$

The event is the logical negation of the event in **lemma 2**. Hence $prob[k_1 \wedge \cdots \wedge k_n] = 1 - prob[\neg k_1 \vee \neg k_2 \vee \cdots \vee \neg k_n] \geq 1 - \frac{1}{n}$.

d

The procedure of c yields probability $1 - \frac{1}{n^{\alpha-1}} = 1 - \frac{1}{n^1}$ from $\alpha = 2$ in b. But the procedure of b is general enough, So we can select any α instead of just $\alpha = 2$. In other words, For any α we can set $\alpha + 1$ in b and get the desired probability bound.