ADS 2021: Week 11 Solutions

Solutions for week 11 of Algorithms and Data Structures.

From Thore's notes - Green

a)
- Worst case: 4 km/day
- Amortised: $\frac{4km*5days}{7days}$ = 2.9 km/day

b)
(a) Expected number: 3.5 km/day. With enchanted die: 1.0 km/day. With cursed die: 6.0 km/day.

(b) Worst case: 6 km/day

(c) Assuming he also runs on weekends: 6 km/day. If he does not, $\frac{6km*5days}{7days}$ = 4.3 km/day

c)
- Worst case: Signs contract in December, pays 100 DKK for contract and an additional 1,100 DKK for the part of the voice call over the 10 included minutes.
- Amortised assuming contract is signed in January: Thore saves up 10 minues of voice call each month until he calls his mother on Christmas day and his amortised cost per month is therefore $\frac{100DKK*12months}{12months}$ = 100 DKK.

d)
- Worst case single ride: 200 DKK
- Amortised cost for a ride: Could be considered 200 DKK as there is no way to "save" up credit before paying 200 DKK for the multiride ticket. Alternatively, for $N0 > 10$ the amortised cost is $\frac{200kr+200kr}{11rides}$ = 36.4 kr/ride.

Old exam set 110530: 2 - Green Note that rebuild() never changes x, so the data structure as given on the exam will crash with IndexOutOfBoundsException if insert() ever calls rebuild().

a) - [A] Stack

b) - [D] return cap-x; Note that the sentence "Let us agree that N denotes the number of elements in the data structure." directly below the code does not change the code and [A] can therefore not be correct.

c) - [C] $\sim 4N$ because rebuild() first makes a new array of objects with a cost of $2N$ (note that the new cap is twice the number of elements N). It then loops over half of these 2N elements doing two array accesses per visited index for a total cost of $\sim 4N$.

d) - [B] constant. "Charge" 9 array access for each insert. 4 for its own rebuild, 4 for the rebuild of an existing element and 1 for the array access in insert().

Mehlhorn-Sanders book: 3.9 - Green

a) Do n pushback(e) operations so the array is full and will be resized on the next pushback(e) call.

b) Alternate between 1 pushback(e) operation, which resizes the array for cost $2n$, and 1 popback() operation which resizes the array for cost n.

Mehlhorn-Sanders book: 3.13 - Yellow Set $n = n - k$ and resize the array if necessary. The running time is still amortised constant time as the potiential resizing has already been "paid for" and is independent of k.

MultiPush - Yellow We could be clever by resizing only once, by first calculating what size the array should be after inserting the k elements. This will be faster, but not asymptotically faster than simply calling pushBack(e) k times and letting the pushBack() method resize the array as necessary. So it will still be amortized constant time for each of the k operations and therefore amortised linear in k.

Old exam set 120531: 2 - Yellow

a) - [B] Queue

b) - [A]. [D] is incorrect as hi may be smaller than lo.

c) - [A]

d) - A = [3, null], hi = 1, lo = 0, N = 1.

e) - [B]

f) - [A] $\sim 4N$. A call to the public method insert(Key in) costs 1 array access but may in the worst case call the private method rebuild() which first makes a new array of objects with a cost of $2N$. It then loops over N elements doing two array accesses per visited index for a total cost of $\sim 4N$.

g) - [B]

h) - [B], it is just a longer sequence of the operations given in d). Note that it is asking "per operation".

i) - False. remove() never calls rebuild, so the space used by Y is linear in the largest value N has had, not in the current value of N.

Set Union - Red

a) Worst case running time:

- Union(A,B): A and B both consist of $N/2$ elements, resulting in a recoloring of $N/2$ elements and $O(N)$.
- SameSet(x, y): Check the color of x and compare to the color of y in $O(1)$.

b) Amortized cost: As we are recoloring the elements in the smaller set, for each k elements being recolored the resulting set has grown to a size $>= 2k$ and at least k elements did not need to be recolored. A set may at most grow $>= 2k$ with $k >= 1$ at most $\log n$ times, meaning each element may at most be recolored $\log n$ times. As $\log n$ is an upped bound for the number of recolorings per element, the upper bound for m union operations is $O(m \log n)$ and with s added constant time SameSet operation it is $O(m \log n + s)$.

Mehlhorn-Sanders book: 3.14 - Red At size array.length/2, allocate a new array of size array.length*2. At each new insert, insert the element at the same position in both arrays, and copy an element from the old array into the new array. When the original array is full, simply change the pointer from the original array to the new array.

To ensure linear space, when the array is half full, we allocate a new array of size array.length/2. For each pop, we then copy an element, starting from index 0, from the original array over to the new array. We have n/2 elements to copy, and n/2 pop operations to do so, resulting in constant time. When the original array is 1/4 full (and the new one is 1/2 full), we change the pointer from the original array to the new array.

In Java or Python, one can consider a single-threaded environment where the data structure starts adding elements to the new, larger array in position n and then moves the elements in a[0..n-1] to the new array between calls. But this fails when the calls are continuous and no background work can be done, meaning the real-time application will no longer be real-time.

One can also consider doing this array allocation and moving of elements in a background thread, but that would require a guarantee that the background thread completes its work before it is needed by the main thread. Which as far I am aware is not possible to guarantee. Doing this efficiently, without the constant time requirement, would be an interesting problem for PCPP students.

Creative students may realise that pushback and popBack is constant time in a linked-list, although that breaks the requirement of using an array, as well as constant time indexing.