Advanced Algorithms, Fall 2012

Prof. Bernard Moret

Homework Assignment #1: Solutions

Question 1.
Implement a queue using two ordinary stacks and analyze the amortized cost of each Insert-Queue and Delete-Queue operation. (Assume the cost of each Pop or Push operation is 1).

Now implement a *double-ended* queue with two ordinary stacks and give a counterexample to show that the amortized cost of its four operations (Insert-Back, Insert-Front, Delete-Back, and Delete-Front) is no longer $O(1)$. A counterexample is a sequence of operations that takes time growing faster than the number of operations in the sequence. Indicate which single operation(s) are such that their removal makes the the amortized cost of the remaining three operations $O(1)$.

Solution.
In a stack, we insert and delete at the same end of a list; in a queue, we insert at one end and delete at the other. So we can use two stacks to implement the queue, say S_1 and S_2. We use S_1 for Insert-Queue, and S_2 for Delete-Queue. Then to do an Insert-Queue, we push that element into S_1. To do a Delete-Queue, we consider S_2: if S_2 is not empty, pop the top element from S_2; otherwise, we pop and push all the elements from S_1 to S_2 one by one, and then pop the top element from S_2.

We define the potential function as: $\Phi = 2n_1$, where n_1 is the number of elements in S_1. So the amortized cost for each Insert-Queue operation is

$$\text{actual cost} + \Delta\Phi = 1 + 2 \leqslant O(1)$$

For each Delete-Queue operation, we need to consider two cases. When S_2 is not empty, the amortized cost for one Delete-Queue operation is:

$$\text{actual cost} + \Delta\Phi = 1 + 0 \leqslant O(1)$$

When S_2 is empty, the amortized cost for one Delete-Queue operation is:

$$\text{actual cost} + \Delta\Phi = 2n_1 + 1 + (0 - 2n_1) \leqslant O(1)$$

So we can conclude that the amortized cost for each Insert-Queue and Delete-Queue is constant.

We use the same setup, with stacks S_1 and S_2, where S_1 corresponds to the "back" of the queue, and S_2 the "front" of the queue. The four operations can be implemented as following:

- insert-back: push the element into S_1.
- insert-front: push the element into S_2.
- delete-back: first check S_1. If S_1 is not empty, pop the top element from S_1 and return; if S_1 is empty, and S_2 is not empty, pop and push all the elements from S_2 to S_1 one by one, then pop the top element from S_1. If both S_1 and S_2 are empty, return error.
- delete-front: is the opposite of delete-back. First check S_2. If S_2 is not empty, pop the top element from S_2 and return; if S_2 is empty, and S_1 is not empty, pop and push all the elements from S_1 to S_2 one by one, then pop the top element from S_2. If both S_1 and S_2 are empty, return error.

Consider the following sequence of operations: starting from an empty queue, after doing n insert-back operations, repeat doing "delete-front, delete-back" until both stacks are empty. The total number of operations is $2n$. The total cost is:

$$n + (2n+1) + [2(n-1)+1] + [2(n-2)+1] + \ldots + 2*1 + 1 = n^2 + 3n$$

which is $\Theta(n^2)$. The first term n in the equation is the cost of inserting n elements. Each following term represents the cost of doing the corresponding delete operation. The cost of this sequence of operations grows faster than the number of operations, so this can be a counterexample to show that the amortized cost of the four operations (insert-back, insert-front, delete-back, and delete-front) is no longer $O(1)$.

Note that in the above counterexample we only used one insert operation. So removing either one of the two insert operations will not prevent such cases from happening. However, if we remove either of the two delete operations, say, we remove delete-back and keep delete-front, we can prove that the amortized cost of the remaining 3 operations (insert-back, insert-front, and delete-front) is $O(1)$. Define the potential function $\Phi = 2n_1$, where n_1 is the number of elements in S_1. Then the amortized cost for one insert-back operation is

$$\text{actual cost} + \Delta\Phi = 1 + 2 \leqslant O(1)$$

The amortized cost for one insert-front operation is

$$\text{actual cost} + \Delta\Phi = 1 + 0 \leqslant O(1)$$

When S_2 is not empty, the amortized cost for one delete-front operation is:

$$\text{actual cost} + \Delta\Phi = 1 + 0 \leqslant O(1)$$

Finally, when S_2 is empty, the amortized cost for one delete-front operation is:

$$\text{actual cost} + \Delta\Phi = 2n_1 + 1 + (0 - 2n_1) \leqslant O(1)$$

Question 2.
Suppose that, instead of using sums of powers of two, we represent integers as sums of Fibonacci numbers. In other words, instead of an array of 0/1 bits, we keep an array of 0/1 "fits", where the i^{th} least significant "fit" indicates whether the sum includes the i^{th} Fibonacci number F_i. For example, the "fitstring" 101110_F represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$.

Verify that a number need not have a unique fitstring representation (in contrast to bitstrings) and describe an algorithm to implement `increase` and `decrease` operations on a single fitstring in constant amortized time.
Solution.
The representation need not be unique. First, the Fibonacci recurrence means that we can always replace two consecutive "1" flits by a single flit one position ahead—the pattern 011 has the same value as the pattern 100. In addition, even if we fixed a canonical representation by insisting that two consecutive ones always be replaced by a carry to the next position, we could still have two representations of the same number: for instance, 21 is the 8th Fibonacci number and so can be represented by the fitstring 10000000, but it can also be written as $F_1 + F_3 + F_5 + F_7 = 1 + 2 + 5 + 13 = 21$ and thus can be represented by the fitstring 1010101. (On the other hand, it is clear that certain numbers have unique representations—for instance, fitstrings of the form $101010\ldots1010$ describe numbers, namely $F_i - 1$, that do not have other representations.)

Now, how do we increment or decrement a fitstring? First, we need to see how to handle carries. If the last fit is a 0, we change it into a 1 and we are done. If the last fit is a 1, we consider the previous fit as well. If the previous (penultimate) fit is a 0—that is, the last two fits in our fitstring are 01—we simply turn it into a 1, so that now the last two fits are 11. (This works because the last two positions both correspond to a value of 1.) On the other hand, if the previous fit is also a 1, so that our fitstring ends with 11, we replace the last two fits by 01, and report a carry to the previous position—what we have done is to convert $x11 = x00 + 011 = x00 + 100 = (x+1)00$ and then added 1 to obtain $(x+1)01$. In the general case, then, we are at position i and have a carry. If the fit at position i is a 0, we simply replace it by a 1 and are done. If, however, the fit is a 1, we now have $2F_i$ to set up in the fitstring and this is not a Fibonacci number. But $2F_i = F_{i+1} + F_{i-2}$, so we can propagate this carry upward and downward, replacing the fit at position i by a 0. Now, notice that upward propagation always follows the pattern $(x+1)0$: the fit to the right of the carry position is always a 0. So we can write our carry rules very simply as follows, showing positions $i+1$, i, and $i-1$, $i > 1$:

$x(0+1)x- > x1x$, $0(1+1)00- > 1001$, $0(1+1)01x- > 1010(x+1)$,
$1(1+1)00- > (1+1)001$, and $1(1+1)01x- > (1+1)010(x+1)$.
The last three rules cause propagation, one upward only, one downward only, and one in both directions. Thus the real cost of an increment is the distance of propagation (the down propagation is not an issue, as it cannot take more work than the up propagation), which is the number of trailing 1s in the fitstring. For instance, incrementing $111111 = 20$ yields $1010101 = 21$. Notice that propagation will create a trailing substring of alternating 0s and 1s, ensuring that a subsequent increment will not require more than constant time. Crucially, a single increment cannot create an arbitrary long sequence of trailing 1s (or trailing 0s), since propagation entails leaving a 0 behind and alternating with a 1; indeed, incrementation can, at most, create one new adjacent pair of 1s. (Contrast this finding with the case of binary representations, where a single incrementation can change a sequence of 1s into a sequence of 0s.)

What about decrement? Because of multiple representations, it need not be symmetric—for instance, to use the previous fitstring, if we increment $111111 = 20$ to get $1010101 = 21$ and then decrement it, the simplest way to do the decrementation is to change the last fit from 1 to 0 to get $1010100 = 20$. Indeed, if the last fit is a 1, we replace it by a 0. If the last fit is a 0, but the previous fit is a 1, we replace the previous fit by a 0. If both last fits are 0, but the previous one is a 1 so that the fitstring ends with 100, we replace these last three fits with 010. If the last three fits are all 0, we enter the more general borrow case. Assume we are a position i with a fit of 1 (otherwise we keep moving up in position) and have a borrow; note that this is a borrow of 1, not of F_i. By inductive hypothesis, we have at least two 0s to the right of fit i, so we have the pattern 100 at positions i, $i-1$, and $i-2$; we replace it by pattern 011 and move the borrow back down to position $i-2$, which has a fit of 1. If we still have two more fits of 0 to the right of this new position, we repeat the operation just described; otherwise we are down to one of our two bases cases and carry out an actual decrement. Hence we get $1000000..00- > 101010..10$ or $1000000..0- > 101010..0$. Thus the real cost of a borrow is proportional to the number of trailing 0s of the fitstring—just as for bitstrings. Notice that the result of a borrow is to replace a string of trailing 0s by a string of alternating 0s and 1s (as in the increment), so that the next decrement will take constant time. Once again, a decrement cannot create an arbitrarily long sequence of 0s—at most, it can eliminate a 1 at the low end and thus create two additional 0-0 adjacencies. (And, once again, contrast this finding with binary representations, where a single decrement can turn a sequence of 0s into a sequence of 1s.)

Now, how does this amortize? The real cost of an increment is proportional to the number of trailing 1s in the fitstring and the real cost of a decrement is proportional to the number of trailing 0s in the fitstring. Suppose we start with k trailing 1s (or 0s); then the next increment (resp., decrement) will take order k time, but it will leave behind a tail of k fits that alternate between 0 and 1. How many operations will it take to return to k trailing 1s (or 0s)? As this seems a rather formidable question, let us ask the simpler question: if we now have i trailing 1s (or 0s), how many operations will it take to produce a fitstring with $i+1$ trailing 1s (or 0s)? Let us start with trailing 0s, as decrement is a bit simpler than increment. Suppose we have no trailing 0, say the number is 11111; then a single decrement gives us one trailing 0, 11110. To get two, we just decrement again, obtaining 11100. But to get three, we need more work, as the next decrement, according to our rules, produces 11010, which is down to just one trailing 0; we need a second decrement to take us to 11000, with 3 trailing 0s. Decrement again: we get 10100, and so have once again lost a trailing 0; decrement a second time: we now get 10010, and have lost another trailing 0; so decrement a third time, and now we get 10000, and have 4 trailing 0s. The cost to go from 3 trailing 0s to 4 trailing 0s was three decrements, and their total cost in terms of propagation was $4+3+2$. With 4 trailing 0s, we get to 5 trailing 0s with five decrement operations:

$$13 = F_7 = F_6 + F_5 = 110000 \rightarrow 101010 \rightarrow 101000 \rightarrow 100100 \rightarrow 100010 \rightarrow 100000 = F_6 = 8$$

and this costs us a total propagation of $5+2+4+3+2$. To go from 7 to 8 trailing 0s is to go from $55 = F_{10} = F_9 + F_8 = 110000000$ down to $34 = F_9 = 100000000$ and so costs us 21 decrements and a total of $8+(3+2)+5+2+4+(3+2)+7+2+4+(3+2)+6+(3+2)+5+2+4+(3+2)$ time units. To turn the $(i+1)$st fit into a 0 first causes us to create $\frac{i}{2}$ new fits with value 1 and so requires us to solve the

same problem recursively for each of these indices. So, finally, we can write the recurrence

$$t(i) = \sum_{j=1}^{\frac{i-1}{2}} t(2j) + \Theta(1)$$

for the number of decrement operations and

$$t(i) = \sum_{j=1}^{\frac{i-1}{2}} t(2j) + \Theta(i)$$

for the number of fits examined or changed. Let us tackle the first recurrence; using our standard telescoping trick, we get

$$t(i) - t(i-2) = t(i-1) + O(1)$$

which is, of course, the Fibonacci recurrence, plus a constant driving term. The important result here is that the growth rate is the same for both recurrences, since the linear driving term in the number of fits examined or changed is dominated by the exponential. Since both recurrences have solution $\Theta(r^i)$, where r is the golden ratio, the amortized cost of a decrement in these sequences is a constant. The same reasoning applies to incrementation. A bit of thought shows that the same reasoning also applies to going from i trailing 0s to i trailing 1s or vice versa.

Now we have analyzed particular sequences of operations, not general ones; how can we be sure that we have the worst-case sequences? We saw that the cost depends on the number of identical trailing fits and analyzed the cost of going from one long sequence of identical trailing fits to another long sequence of identical trailing fits. Can we make this argument more formal with a potential function?

Now that we understand the system, we can indeed devise a suitable potential function. Since the ideal sequence is one that alternates 0s and 1s, our potential will measure the number of adjacent identical fits in the fitstring:

$$\Phi(s) = \Phi(s_n s_{n-1} \dots s_1 s_0) = \sum_{i=1}^{n} (s_i = s_{i-1})$$

Everything now works easily: if the real cost of an operation is i, it is because the fitstring has i identical trailing fits, so that its potential is at least $i-1$. But the operation will then turn these i identical trailing fits into a sequence of i alternating fits, decreasing the potential by $i-1$ and thus paying for the cost of the operation. As we have seen, any increment or decrement can at most create one or two adjacent pairs of identical fits, so an operation that takes constant real time could at most increase the potential by a constant. Overall, the sum of the real time and the potential change remains bounded by a constant—and we are done!

So why is it that we can increment and decrement in constant amortized time, whereas the same is not possible for binary representations? (Just consider starting with the number 2^n and then alternately decrementing and incrementing it: every successive operation flips n bits, so the cost cannot amortize to better than $\Theta(n)$ per operation.) The mechanical reason is that an expensive operation turns a "dangerous" fitstring (with a long trailing sequence of identical fits) into a "friendly" fitstring (with a long trailing sequence of alternating fits)—whereas, in a binary operation, it is possible to turn a dangerous bitstring (a long sequence of 0s) into another dangerous bitstring (a long sequence of 1s). But what made the "friendly" fitstring possible is the presence of alternate representations of the same number, which breaks the symmetry of operations. If all operations were perfectly symmetric, as is the case with binary representation, this could not work. For instance, we could start with $11111\dots11$, increment to get $101010\dots101$, upon which a decrement would return us to the starting string—this would give us a cycle in which half the fits would be flipped at each operation, so that each operation could not be amortized to better than $\Theta(n)$. In that case, the alternating fitstring would not really be "friendly" at all! Viewed differently, the different representations enable us to return to the previous fitstring through a long detour rather than in just one step and so give us a hysteresis cycle. (Indeed, there is classical algorithm in computer arithmetic that shows how to keep two bitstrings per number so as to obtain multiple representations of the same value and thereby amortize the cost of incrementation and decrementation to a constant.)

Question 3.
Using arrays, as in a simple queue or a binary heap, creates a problem: when the needs grow beyond the allocated array size, there is no support for simply increasing the array. Therefore consider the following solution.

- When the next insertion encounters a full array, the array is copied into one twice its size (and the old array returned to free storage) and the insertion then proceeds.
- When the next deletion reduces the number of elements below one quarter of the allocated size, the array is copied into one half its size (and the old array returned to free storage) and the deletion then proceeds.

Assume that returning an array to free storage takes constant time, but that creating a new array takes time proportional to its size.

- Prove that, in the context of a data structure that grows or shrinks one element at a time, the amortized cost per operation of array doubling and array halving is constant.
- Why not halve the array as soon as the number of elements falls below one half of the allocated size?

Solution.
Intuitively, the amortization works because, when we incur a cost of $2N$, where N is the size of the full array, this must have been preceded by at least N insertions to fill the array, and so we can shift the cost of the array doubling onto the preceding insertions. Similarly, when the array is to be halved, we must have had at least $\frac{N}{4}$ deletions (the array could have started only half full, as a result of the last doubling), and again we can shift the halving cost of $\frac{N}{2}$ onto the preceding deletions. Thus our potential needs to grow as the number of items in the array grows closer to N, but also as it shrinks closer to $\frac{N}{4}$.

We shall choose a reference point between $\frac{N}{4}$ and N. For example, the midpoint is $\frac{5N}{8}$, so we define the potential of an array of size N with k elements in it as $c_1(k-\frac{5N}{8})$ when k is larger than (or equal to) the midpoint, and $c_2(\frac{5N}{8}-k)$ otherwise. The main component is the deviation from the midpoint; the multiplicative factors ($c_1 > 0$ and $c_2 > 0$) can be chosen as follows.

Clearly, any insertion (or deletion) that does not incur doubling or halving has constant amortized running time, since both the actual cost and the change of potential are constants. Now we analyze the cases of doubling and halving. A doubling costs us $2N$ in real costs, but brings about a change in potential: the new array will have $N+1$ elements in it, with a midpoint of $\frac{5N}{4}$, and so we gain a potential of $c_2(\frac{N}{4}-1)$, while the old array had a potential of $c_1(N-\frac{5N}{8})$, which we lose. The total loss is thus $(\frac{3}{8}c_1-\frac{c_2}{4})N+c_2$. Adding the real costs and the change in potential gives us an amortized cost of $(2-\frac{3}{8}c_1+\frac{c_2}{4})N-c_2$. Similarly, a halving costs us $\frac{N}{2}$ in real costs. The old array had a potential of $c_2(\frac{5N}{8}-\frac{N}{4})=\frac{3c_2N}{8}$, which we lose while the new array, with $\frac{N}{4}-1$ elements in it and a midpoint of $\frac{5N}{16}$, yields a gain in potential of $c_2(\frac{5N}{16}-(\frac{N}{4}-1))=\frac{c_2N}{16}+c_2$. adding all three components give us an amortized cost of $(\frac{1}{2}-\frac{5c_2}{16})N+c_2$. Since both doubling and halving should have $O(1)$ amortized running time, we have $2-\frac{3c_1}{8}+\frac{c_2}{4}=0$ and $\frac{1}{2}-\frac{5c_2}{16}=0$, which yield $c_2=\frac{8}{5}$ and $c_1=\frac{32}{5}$.

In summary, we can define the potential of an array of size N with k elements in it as $\frac{32}{5}(k-\frac{5N}{8})$ if $k \geq \frac{5N}{8}$ and $\frac{8}{5}(\frac{5N}{8}-k)$ otherwise. The 4:1 ratio between our two factors reflects the fact that the real cost of doubling the array is much higher than that of halving it.

However, we could use the same constant in both cases, in which case we need to adjust the reference point—instead of the midpoint, it will be closer to the halving threshold. If we call the unknown constant multiplier c and the unknown reference point αN, $\frac{1}{4} < \alpha < \frac{1}{2}$ (you can also try $\alpha \geq \frac{1}{2}$, but in that case, you cannot get the same constant), then we have, for doubling, the amortized cost is $2N-c(N-\alpha N)+c((N+1)-2\alpha N)$ and for halving the amortized cost is $\frac{N}{2}-c(\alpha N-\frac{N}{4})+c(\frac{N}{4}-1-\frac{\alpha N}{2})$. Both of them should be $O(1)$, thus we have $\alpha=\frac{2}{5}$ and $c=5$, resulting in a potential definition of $5|k-\frac{2N}{5}|$. (It is worth repeating that the potential is just an accounting mechanism—any choice that works is good enough (there is no such thing as a better or worse choice); conversely, if the amortization does not work with some choice of potential, the only conclusion that can be draw is that this was not a good choice of potential—nothing can be said about the amortization.)

So why not reduce the array size as soon as the number of items falls back below $\frac{N}{2}$? Well, consider the deletion that causes that to happen, then follow it by two insertions, two deletions, two insertions, etc. For the second of each pair of insertions, we have to double the array; for the second of each pair of deletions, we would have to halve the array; we would thus incur the full costs of an array doubling and an array halving with only four other operations over which to spread the cost—unless these other operations each took linear time, this could not be done. This is an illustration of a very basic engineering principle: we need *hysteresis* in a regulatory cycle (here a cycle of doubling and halving) so as not to get a potentially unstable system that constantly switches between the two. (The most familiar hysteresis cycle is probably that of a thermostat; another common one is the cruise control system that keeps a car around a constant speed on the highway.)