

Homework 5 (due before class, Monday, November 12, 2001)

1 Amortized Analysis [Chapter 17 in CLRS]

1. Suppose we want to implement a stack using a dynamic array, but instead of doubling the size when the array is full, we decide to multiply the size by $3/2$. Assume that increasing the size of the array from n to $\frac{3n}{2}$ costs n . Thus, the *actual* cost of a push is 1 if the array was not full, and $(1 + n)$ if it was.

- What is the amortized cost of a push?
- Suppose we want to make sure the array never becomes less than $2/3$ full. One way to do this is to shrink the array from size n to $\frac{2n}{3}$ whenever we pop from an array that is $2/3$ full. Does this yield an amortized cost of $O(1)$ per operation?
- Another approach is to reduce the size from n to $\frac{5n}{6}$ whenever we pop from an array that is $2/3$ -full. Does this yield an amortized cost of $O(1)$?

2. [CLRS 17.3-6] Show how to implement a queue with two ordinary stacks so that the amortized cost of each *Enqueue* and each *Dequeue* operation is $O(1)$.

3. Show how to implement a dynamic set that efficiently supports the FIFO queue operations *Enqueue* and *Dequeue*, as well as *Minimum*. Solve this problem, but follow the time constraints given in each part.

- Implement *Enqueue* and *Dequeue* in $O(1)$ time, and *Minimum* in $O(n)$.
- Implement *Enqueue* and *Dequeue* in $O(\lg n)$ time, and *Minimum* in $O(1)$.
- Implement all three operations in $O(1)$ amortized time.

4. Suppose we have a binary counter such that the cost to increment the counter is equal to the number of bits that need to be flipped. We saw in class that if the counter begins at 0, and we increment the counter n times, the amortized cost per increment is just $O(1)$. Equivalently, the total cost to perform all n increments is $O(n)$. Suppose that we want to be able to both increment *and* decrement the counter.

- Show that even without making the counter go negative, it is possible for a sequence of n operations, allowing both increments and decrements, to cost as much as $\Omega(\lg n)$ amortized per operation (i.e. $\Omega(n \lg n)$ total cost).
- To fix the problem from part (a), consider the following *redundant ternary number system*. A number is represented by a sequence of *trits*, each of which is 0, +1, or -1. The value of the number t_{k-1}, \dots, t_0 (where each t_i is a trit) is defined to be $\sum_{i=0}^{k-1} t_i 2^i$. The process of incrementing a ternary number is analogous to that operation on binary numbers. One is added to the low order trit. If the result is 2, then it is changed to

0, and a carry is propagated to the next trit. This process is repeated until no carry results. Decrementing a number is similar. One is subtracted from the low order trit. If it becomes -1 , then it is replaced by a 0, and a borrow is propagated.

The cost of an increment or decrement is the number of trits that change in the process. Starting from 0, a sequence of n increments and decrements are done. Give a clear, coherent proof that with this representation, the amortized cost per operation is $O(1)$ (i.e. the total cost for the n operations is $O(n)$). Hint: Think about a "bank account" or "potential function" argument.

2 Heaps [Chapter 19, 20 in CLRS]

5. [CLRS 19-1] Describe this problem at a higher level. Omit part (f).

3 External Memory Algorithms

6. A professor has developed a hardware priority queue for his computer. The priority queue device can store up to p records, each consisting of a key and a small amount of data (such as a pointer). The computer to which it is attached can perform *Insert* and *ExtractMin* operations on the priority queue, each of which takes $O(1)$ time, no matter how many records are stored in the device. The professor wishes to use the hardware priority queue to help implement a sorting algorithm on his computer. He has n records stored in the primary memory of his machine. If $n \leq p$, the professor can certainly sort the keys in $O(n)$ time by first inserting them into the priority queue, and then repeatedly extracting the minimum. Design an efficient algorithm for sorting $n > p$ items using the hardware priority queue. Analyze your algorithm in terms of both n and p .