CS 373: Combinatorial Algorithms, Fall 2002

Homework 3, due October 17, 2002 at 23:59:59

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Undergrads	Grads

This homework is to be submitted in groups of up to three people. Graduate and undergraduate students are *not* allowed to work in the same group. Please indicate above whether you are undergraduate or graduate students. Only *one* submission per group will be accepted.

Required Problems

1. (a) Prove that only one subtree gets rebalanced in a scapegoat tree insertion.

   (b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree. (Recall that $I(v) = \max\{0, |T| - |s| - 1\}$, where T is the child of greater height and s the child of lesser height, and $|v|$ is the number of nodes in subtree v. A perfectly balanced tree has two perfectly balanced subtrees, each with as close to half the nodes as possible.)

   *(c) Show that you can rebuild a fully balanced binary tree from an unbalanced tree in $O(n)$ time using only $O(\log n)$ additional memory.

2. Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
   - After a deletion, if the table is less than 1/4 full, we we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

   Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do *not* use the potential method (it makes it much more difficult).

3. A stack is a FILO/LIFO data structure that represents a stack of objects; access is only allowed at the top of the stack. In particular, a stack implements two operations:

   - PUSH(x): adds x to the top of the stack.

- POP: removes the top element and returns it.

A queue is a FIFO/LILO data structure that represents a row of objects; elements are added to the front and removed from the back. In particular, a queue implements two operations:

- ENQUEUE(x): adds x to the front of the queue.
- DEQUEUE: removes the element at the back of the queue and returns it.

Using two stacks and no more than O(1) additional space, show how to simulate a queue for which the operations ENQUEUE and DEQUEUE run in constant amortized time. You should treat each stack as a black box (i.e., you may call PUSH and POP, but you do not have access to the underlying stack implementation). Note that each PUSH and POP performed by a stack takes $O(1)$ time.

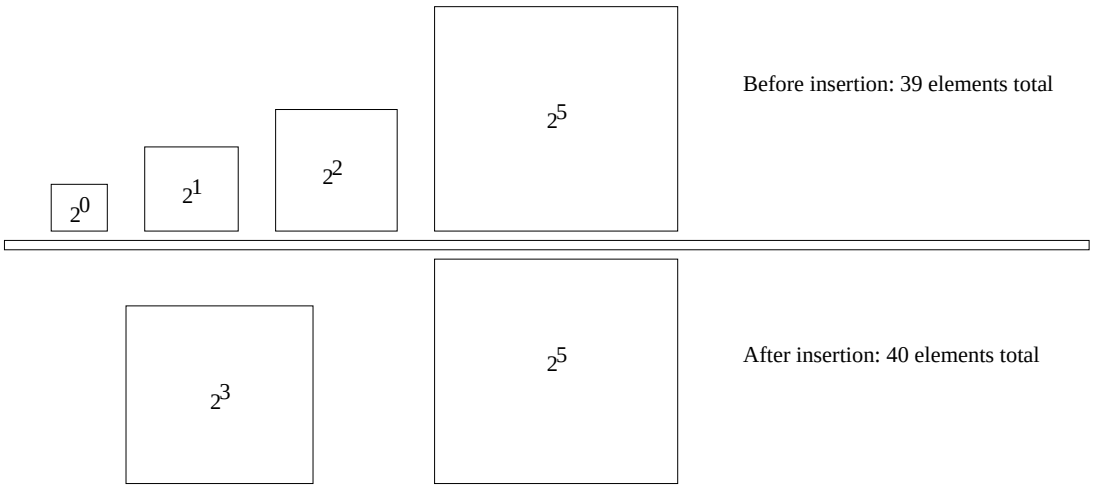
4. A data structure is *insertion-disabled* if there is no way to add elements to it. For the purposes of this problem, further assume that an insertion-disabled data structure implements the following operations with the given running times:

   - INITIALIZE(S): Return an insertion-disabled data structure that contains the elements of S. Running time: $O(n \log n)$.
   - SEARCH(D, x): Return TRUE if x is in D; return FALSE if not. Running time: $O(\log n)$.
   - RETURNALL(D, x): Return an unordered set of all elements in D. Running time: $O(n)$.
   - DELETE(D, x): Remove x from D if x is in D. Running time: $O(\log n)$.

   Using an approach known as the Bentley-Saxe Logarithmic Method (BSLM), it is possible to represent a dynamic (i.e., supports insertions) data structure with a collection of insertion-disabled data structures, where each insertion-disabled data structure stores a number of elements that is a distinct power of two. For example, to store $39 = 2^0 + 2^1 + 2^2 + 2^5$ elements in a BSLM data structure, we use four insertion-disabled data structures with 2^0, 2^1, 2^2, and 2^5 elements.

   To find an element in a BSLM data structure, we search the collection of insertion-disabled data structures until we find (or don't find) the element.

   To insert an element into a BSLM data structure, we think about adding a 2^0-size insertion-disabled data structure. However, an insertion-disabled data structure with 2^0 elements may already exist. In this case, we can combine two 2^0-size structures into a single 2^1-size structure. However, there may already be a 2^1-size structure, so we will need to repeat this process. In general, we do the following: Find the smallest i such that for all nonnegative $k < i$, there is a 2^k-sized structure in our collection. Create a 2^i-sized structure that contains the element to be inserted and all elements from 2^k-sized data structures for all $k < i$. Destroy all 2^k-sized data structures for $k < i$.



Inserting an element into a BSLM data structure.

We delete elements from the BSLM data structure lazily. To delete an element, we first search the collection of insertion-disabled data structures for it. Then we call DELETE to remove the element from its insertion-disabled data structure. This means that a 2^i-sized insertion-disabled data structure might store less than 2^i elements. That's okay; we just say that it stores 2^i elements and say that 2^i is its *pretend* size. We keep track of a single variable, called $Waste$, which is initially 0 and is incremented by 1 on each deletion. If $Waste$ exceeds three-quarters of the total pretend size of all insertion-disabled data structures in our collection (i.e., the total number of elements stored), we rebuild our collection of insertion-disabled data structures. In particular, we create a 2^m-sized insertion-disabled data structure, where 2^m is the smallest power that is greater than or equal to the total number of elements stored. All elements are stored in this 2^m-sized insertion-disabled data structure, and all other insertion-disabled data structures in our collection are destroyed. $Waste$ is reset to $2^m - n$, where n is the total number of elements stored in the BSLM data structure.

Your job is to prove the running times of the following three BSLM operations:

- SEARCHBSLM(D, x): Search for x in the collection of insertion-disabled data structures that represent the BSLM data structure D. Running time: $O(\log^2 n)$ worst-case.
- INSERTBSLM(D, x): Insert x into the collection of insertion-disabled data structures that represent the BSLM data structure D, modifying the collection as necessary. Running time: $O(\log^2 n)$ amortized.
- DELETEBSLM(D, x): Delete x from the collection of insertion-disabled data structures that represent the BSLM data structure D, rebuilding when there is a lot of wasted space. Running time: $O(\log^2 n)$ amortized.

5. Except as noted, the following sub-problems refer to a Union-Find data structure that uses both path compression and union by rank.

   (a) Prove that in a set of n elements, a sequence of n consecutive FIND operations takes $O(n)$ total time.

   (b) Show that any sequence of m MAKESET, FIND, and UNION operations takes only $O(m)$ time if all of the UNION operations occur before any of the FIND operations.

(c) Now consider part b with a Union-Find data structure that uses path compression but does *not* use union by rank. Is $O(m)$ time still correct? Prove your answer.

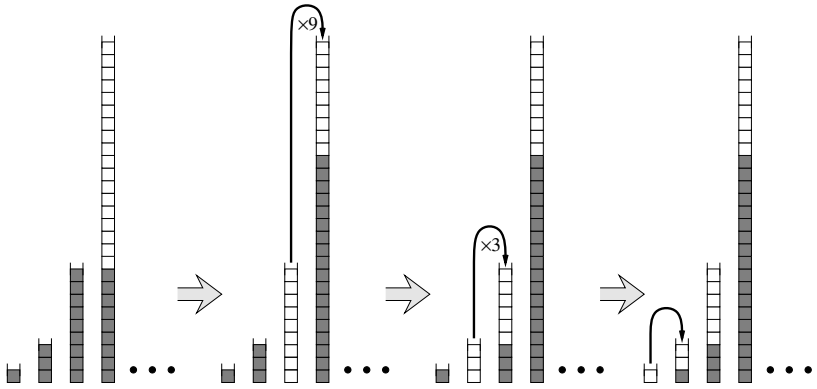
6. *[This problem is required only for graduate students (including I2CS students); undergrads can submit a solution for extra credit.]*

   Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of 'fits', where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string.]

Practice Problems

These remaining practice problems are entirely for your benefit. Don't turn in solutions—we'll just throw them out—but feel free to ask us about these questions during office hours and review sessions. Think of these as potential exam questions (hint, hint).

1. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i, we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2}, and so on. Moving a single element from one stack to the next takes $O(1)$ time.



Making room for one new element in a multistack.

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?

   (b) Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack. You can use any method you like.

2. A hash table of size m is used to store n items with $n \leq m/2$. Open addressing is used for collision resolution.

   (a) Assuming uniform hashing, show that for $i = 1, 2, \ldots, n$, the probability that the i^{th} insertion requires strictly more than k probes is at most 2^{-k}.

   (b) Show that for $i = 1, 2, \ldots, n$, the probability that the i^{th} insertion requires more than $2 \lg n$ probes is at most $1/n^2$.

   Let the random variable X_i denote the number of probes required by the i^{th} insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

   (c) Show that $\Pr\{X > 2 \lg n\} \leq 1/n$.

   (d) Show that the expected length of the longest probe sequence is $E[X] = O(\lg n)$.

3. A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. That is operation i costs $f(i)$, where:

$$f(i) = \begin{cases} i, & i = 2^k, \\ 1, & \text{otherwise} \end{cases}$$

Determine the amortized cost per operation using the following methods of analysis:

(a) Aggregate method

(b) Accounting method

*(c) Potential method