CS 373U: Combinatorial Algorithms, Spring 2004
Homework 4

Due Friday, April 2, 2004 at noon

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

- For each numbered problem, if you use more than one page, staple all those pages together. **Please do *not* staple your entire homework together.** This will allow us to more easily distribute the problems to the graders. Remember to print the name and NetID of every member of your group, as well as the assignment and problem numbers, on every page you submit. You do not need to turn in this cover page.
- As with previous homeworks, we strongly encourage you to begin early.

#	1	2	3	4	5	6*	Total
Score							
Grader							

1. 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 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 (like CLRS does); there is a much easier solution.

2. Remember the difference between stacks and queues? Good.

   (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The *only* access you have to the stacks is through the standard subroutines PUSH and POP.

   (b) A *quack* is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

   - **Push:** add a new item to the left end of the list;
   - **Pop:** remove the item on the left end of the list;
   - **Pull:** remove the item on the right end of the list.

   Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any push, pop, or pull operation is $O(1)$. Again, you are *only* allowed to access the stacks through the standard functions PUSH and POP.

3. Some applications of binary search trees attach a *secondary data structure* to each node in the tree, to allow for more complicated searches. Maintaining these secondary structures usually complicates algorithms for keeping the top-level search tree balanced.

   Suppose we have a binary search tree T where every node v stores a secondary structure of size $O(|v|)$, where $|v|$ denotes the number of descendants of v in T. Performing a rotation at a node v in T now requires $O(|v|)$ time, because we have to rebuild one of the secondary structures.

   (a) **[1 pt]** Overall, how much space does this data structure use in the worst case?

   (b) **[1 pt]** How much space does this structure use if the top-level search tree T is balanced?

   (c) **[2 pt]** Suppose T is a splay tree. Prove that the *amortized* cost of a splay (and therefore of a search, insertion, or deletion) is $\Omega(n)$. *[Hint: This is easy!]*

   (d) **[3 pts]** Now suppose T is a scapegoat tree, and that rebuilding the subtree rooted at v requires $\Theta(|v| \log |v|)$ time (because we also have to rebuild all the secondary structures). What is the *amortized* cost of inserting a new element into T?

   (e) **[3 pts]** Finally, suppose T is a treap. What's the worst-case *expected* time for inserting a new element into T?

4. In a *dirty* binary search tree, each node is labeled either *clean* or *dirty*. The lazy deletion scheme used for scapegoat trees requires us to *purge* the search tree, keeping all the clean nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.

   Describe an algorithm to purge an arbitrary n-node dirty binary search tree in $O(n)$ time, using only $O(\log n)$ additional memory. For 5 points extra credit, reduce the additional memory requirement to $O(1)$ *without repeating an old CS373 homework solution.*[1]

5. This problem considers a variant of the lazy binary notation introduced in the extra credit problem from Homework 0. In a *doubly lazy binary number*, each bit can take one of *four* values: -1, 0, 1, or 2. The only legal representation for zero is 0. To increment, we add 1 to the least significant bit, then carry the rightmost 2 (if any). To decrement, we subtract 1 from the lest significant bit, and then borrow the rightmost -1 (if any).

```
LazyIncrement(B[0..n]):
  B[0] ← B[0] + 1
  for i ← 1 to n − 1
      if B[i] = 2
          B[i] ← 0
          B[i + 1] ← B[i + 1] + 1
          return
```

```
LazyDecrement(B[0..n]):
  B[0] ← B[0] − 1
  for i ← 1 to n − 1
      if B[i] = −1
          B[i] ← 1
          B[i + 1] ← B[i + 1] − 1
          return
```

   For example, here is a doubly lazy binary count from zero up to twenty and then back down to zero. The bits are written with the least significant bit (*i.e.*, $B[0]$) on the right. For succinctness, we write $\bar{1}$ instead of -1 and omit any leading 0's.

$$
\begin{aligned}
0 &\xrightarrow{\texttt{++}} 1 \xrightarrow{\texttt{++}} 10 \xrightarrow{\texttt{++}} 11 \xrightarrow{\texttt{++}} 20 \xrightarrow{\texttt{++}} 101 \xrightarrow{\texttt{++}} 110 \xrightarrow{\texttt{++}} 111 \xrightarrow{\texttt{++}} 120 \xrightarrow{\texttt{++}} 201 \xrightarrow{\texttt{++}} 210 \\
&\xrightarrow{\texttt{++}} 1011 \xrightarrow{\texttt{++}} 1020 \xrightarrow{\texttt{++}} 1101 \xrightarrow{\texttt{++}} 1110 \xrightarrow{\texttt{++}} 1111 \xrightarrow{\texttt{++}} 1120 \xrightarrow{\texttt{++}} 1201 \xrightarrow{\texttt{++}} 1210 \xrightarrow{\texttt{++}} 2011 \xrightarrow{\texttt{++}} 2020 \\
&\xrightarrow{\texttt{--}} 2011 \xrightarrow{\texttt{--}} 2010 \xrightarrow{\texttt{--}} 2001 \xrightarrow{\texttt{--}} 2000 \xrightarrow{\texttt{--}} 20\bar{1}1 \xrightarrow{\texttt{--}} 2\bar{1}10 \xrightarrow{\texttt{--}} 2\bar{1}01 \xrightarrow{\texttt{--}} 1100 \xrightarrow{\texttt{--}} 11\bar{1}1 \xrightarrow{\texttt{--}} 1010 \\
&\xrightarrow{\texttt{--}} 1001 \xrightarrow{\texttt{--}} 1000 \xrightarrow{\texttt{--}} 10\bar{1}1 \xrightarrow{\texttt{--}} 1\bar{1}10 \xrightarrow{\texttt{--}} 1\bar{1}01 \xrightarrow{\texttt{--}} 100 \xrightarrow{\texttt{--}} 1\bar{1}1 \xrightarrow{\texttt{--}} 10 \xrightarrow{\texttt{--}} 1 \xrightarrow{\texttt{--}} 0
\end{aligned}
$$

   Prove that for any intermixed sequence of increments and decrements of a doubly lazy binary number, starting with 0, the amortized time for each operation is $O(1)$. Do *not* assume, as in the example above, that all the increments come before all the decrements.

[1] That was for a slightly different problem anyway.

6. [**Extra credit**] My wife is teaching a class[2] where students work on homeworks in groups of exactly three people, subject to the following rule: *No two students may work together on more than one homework.* At the beginning of the semester, it was easy to find homework groups, but as the course progresses, it is becoming harder and harder to find a legal grouping. Finally, in despair, she decides to ask a computer scientist to write a program to find the groups for her.

   (a) We can formalize this homework-group-assignment problem as follows. The input is a graph, where the vertices are the n students, and two students are joined by an edge if they have not yet worked together. Every node in this graph has the same degree; specifically, if there have been k homeworks so far, each student is connected to exactly $n - 1 - 2k$ other students. The goal is to find $n/3$ disjoint triangles in the graph, or conclude that no such triangles exist. Prove (or disprove!) that this problem is NP-hard.

   (b) Suppose my wife had planned ahead and assigned groups for every homework at the beginning of the semester. How many homeworks can she assign, as a function of n, without violating the no-one-works-together-twice rule? Prove the best upper and lower bounds you can. To prove the upper bound, describe an algorithm that actually assigns the groups for each homework.

[2]Math 302: Non-Euclidean Geometry. Problem 1 from last week's homework assignment: "Invert Mr. Happy."