MC57 Intra-term Exam 2

Serial #:

Note added in 2011: This test from 1994 is offered as an aid in studying for the 2011 test. (MC57 in 1994 was the same course as is numbered 2011. This was the pervious time the course was offered by the same instructor.) The topics covered between the time of test 1 and test 2 were largely the same in 1994 as in 2011; binomial heaps were covered in 1994 but didn't show up on this test at all. (Augmented data structures, skip lists, and disjoint set structures weren't covered in 1994.) I've tried to include notes where the change in textbook edition matter, although I might have missed some.

This exam is an open-book, open-notes, open-computer, individual-work exam. You may may work on it anywhere you like at any time you like, but you may work for *no longer than two hours* and must return it within 24 hours of picking it up from me.

Do any three problems. If you do any work on all four, mark one "don't grade." Do the best you can in the available time and then stop worrying about it. I take your name on this test to be a pledge of honor that you have worked on the test no longer than two hours.

Please write your name below. Be sure to look at all problems before deciding which one to do first. Some problems are easier than others, so plan your time accordingly.

Write the answer to each problem on the page on which that problem appears. You may also use the back of that same sheet if you need additional space, or attach additional paper identified with problem number and test serial number.

Name: ______________________________

Problem	Page	Possible	Score
1	2	12	
2	3	12	
3	4	12	
4	5	12	
Total		36	

1. [**12 Points**] A compiler needs to keep track of information about symbols. We'll model this by treating each symbol as an object with a property called *info*, i.e., *info*[*sym*], where *sym* is a symbol.

   Note added in 2011: The old textbook edition used the above syntax with brackets for properties of an object. Our current edition would use *sym.info*.

   We'll assume that the value of each symbol's *info* property is a stack, and that these stacks start out empty. For convenience, we'll presume that every datum pushed onto the *info* stacks has a *depth* property available for us to use. We'll also use an additional stack, not associated with any symbol, called *scopeStack*, which also starts out empty, and a global *depth* counter that starts out at 0. Do an amortized analysis of the following operations, assuming that the actual cost is 1 per Push or Pop done. (All the other steps are free.)

```
Enter-Scope()
  depth $\leftarrow$ depth + 1
  Push(scopeStack, NIL)

Declare-Info(sym, information)
  if not Stack-Empty(info[sym])
     then x $\leftarrow$ Pop(info[sym])
          if depth[x] $\neq$ depth
             then Push(info[sym], x)
                  Push(scopeStack, sym)
     else Push(scopeStack, sym)
  depth[information] $\leftarrow$ depth
  Push(info[sym], information)

Get-Info(sym)
  if Stack-Empty(info[sym])
     then error "undeclared symbol"
     else x $\leftarrow$ Pop(info[sym])
          Push(info[sym], x)
          return x

Exit-Scope()
  depth $\leftarrow$ depth - 1
  sym $\leftarrow$ Pop(scopeStack)
  while sym $\neq$ NIL
     do Pop(info[sym])
        sym $\leftarrow$ Pop(scopeStack)
```

2. [**12 Points**] Do exercise 17.2-2 on page 336 of the text, i.e., give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack. The definition of the 0-1 knapsack problem is on page 335; note in particular that it stipulates that W is an integer, and so are the weights and values of each of the n items.

   Note added in 2011: The exercise is now number 16.2-2. The page numbers have also changed. (I don't know at the moment what the new page numbers are.)

3. [**12 Points**] Since deleting from a B-tree is complex, consider the following alternative. When a request is made to delete an item from the tree, normally it is just marked as "deleted," but physically left in the tree. The search and insertion procedures are very slightly modified, so that they still use "deleted" keys in internal nodes to decide which child to move to, but otherwise ignore the "deleted" keys. Counts are kept of how many real items the tree contains (n) and how many "deleted" items (k). When a deletion request results in $k \geq n$, so that half or more of the items in the tree are marked as "deleted," then a tree-walk of the entire B-tree is done, and all the non-deleted items found in this tree-walk are re-built into a brand new B-tree with n real items but no "deleted" items. The tree walk can be done in time $\Theta(n + k)$ time, and it is possible to build the new tree in $\Theta(n)$ time, rather than the $\Theta(n \lg n)$ you might naively expect for n insertions into an empty tree.

   (a) What impact does the presence of the "dead wood" in the B-tree have on the performance of the search and insert operations? Can you give a worst-case performance bound for each of those operations that is in terms only of n, not also of k?

   (b) What, in terms only of n, is an asymptotically-tight worst-case time for the delete operation (including the possible tree copy)?

   (c) Show that the deletion operation, even including the tree copying, can be done in $O(1)$ amortized time.

4. [**12 Points**] (This is a slight variant on parts a and c of problem 17-1 on page 353 of the text.)

   Note added in 2011: The problem is now number 16.1. The page number has also changed. (I don't know at the moment what the new page number is.)

   Consider the problem of making change for n cents using the least number of coins.

   (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

   (c′) Give a set of coin denominations and a number of cents, n, such that it is possible to choose coins with denominations from your set that total to n, but the greedy algorithm fails to do so. That is, the greedy algorithm fails to make change for the specified amount, optimally or otherwise.

   (c″) Give a set of coin denominations and a number of cents, n, such that the greedy algorithm succeeds in making change (i.e., choosing coins that total to n), put uses more coins than the optimal solutions.