HW4 Solutions: Preliminary version

March 13, 2001

Note that Don graded 7.5-5, 13.2-4, and the extra problem. Bo graded the rest.

7.1-1 The maximum number of elements is $2^{h+1}-1$, if the bottom row is "full"; the minimum number is 2^h, if there is only one element in the bottom row.

Comments: One point was deducted if the superscript was off by 1. One point was also deducted if there was no attempt to explain where the numbers are coming from.

7.1-2 Note that $2^{\lfloor \lg n \rfloor} \le 2^{\lg n} = n < 2^{\lfloor \lg n \rfloor + 1}$. Thus, by 7.1-1, a heap with n elements must have height $\lfloor \lg n \rfloor$.

7.1-3 A formal argument would proceed by induction on height. It's trivially true for a heap of height 1. Suppose it's true for a heap of height h and we are given a heap of height $h+1$. Notice that, by the heap property, the root is larger than its children. If we consider the heap rooted at a child, it has height $\le h$; therefore, by the induction hypothesis, each child of the root is larger than any element in the subtree rooted at that child. Since the root is larger than its children, the result follows. [Induction on the number of elements in the heap is also OK; it's also OK to do a proof by contradiction.]

7.1-4 The smallest element must be a leaf (since a nonleaf must be greater than its children), but it can be any leaf. Thus, in an n-element heap, the smallest element can be any element between $2^{\lfloor \lg n \rfloor}$ and n.

Comments: 1 point was deducted if you didn't say that the smallest element is a leaf. Another point was deducted if you just said it was between $\lceil n/2 \rceil$ and n.

7.1-5 Yes an array in reverse sorted order is a heap, since $A[i] \ge A[2i]$ and $A[i] \ge A[2i+1]$, so it satisfies the heap property.

7.2-2 HEAPIFY(A, i) does nothing if $A[i]$ is larger than its children. (More accurately, we'll end up with $\mathit{largest} = i$.)

7.5-5 Roughly speaking, HEAP-DELETE fills the vacated spot with the “last” element of the heap (much as in HEAP-EXTRACT-MAX), and then adjusts the structure to restore the heap property. Using a process similar to HEAP-INSERT, we move the formerly last element up in the tree until it is less than or equal to its parent. Finally, we heapify the subtree rooted at the new location of that element. Here’s the code:

HEAP-DELETE(A, i)

```
if $heap\text{-}size[A] < i$
  then error
$key \leftarrow A[heap\text{-}size[A]]$
$heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
$n \leftarrow i$
while $n > 1$ and $A[\text{PARENT}(n)] < key$ do
  $A[n] \leftarrow A[\text{PARENT}(n)]$
  $n \leftarrow \text{PARENT}(n)$
$A[n] \leftarrow key$
HEAPIFY$(A, n)$
```

Comments: There were very few totally complete solutions to this problem. Most were along the lines of: (1) Switch item to be deleted (call it item D) with the “last” item in the heap (call it item L); (2) Lower the heap size so that item D is now excluded; (3) Heapify at the new location of item L. This is almost correct, and I suspect that most students tried it out on a few heaps. But consider the heap stored as [6 3 5 2 1 3 4]. If one attempts to delete 1 from this heap, 4 goes into the place previously held by the 1, the 1 is excluded from the heap, and HEAPIFY is run at the new location of the 4. However, 4’s parent is now 3! This violates the heap property. See the solution above for an algorithm that deals with this problem.

13.1-2 The BST property says nodes are ordered inorder. That is, all the nodes in the left subtree of x are smaller than x, all the nodes in the right subtree of x are larger than x. The heap property says that the children of a parent are smaller than the parent, but puts no constraints on the relative order of the children.

You can’t use a heap to print out nodes in sorted order in time $O(n)$. We know it only takes time $O(n)$ to build the heap (see Section 7.3), so if we could sort in time $O(n)$ given a heap, we could sort in time $O(n)$, which can’t be done.

[Grading: 1 point each for stating the BST property and heap property. 2 points for saying that a heap can be used to sort and justifying it.]

13.1-5 Given a binary search tree, we can sort in time $O(n)$ in the comparison-based model (without doing any comparisons at all). Since sorting must take $O(n \lg n)$ comparison, building the tree must have taken $O(n \lg n)$ comparisons.

13.2-3 There are two cases. First suppose that x has a right successor. Then TREE-SUCCESSOR(x) returns the minimum element in the subtree rooted at *right*(x). Suppose this element is y. Clearly y is the successor of x in the subtree rooted at x. We still have to argue that there is no element z such that $x < z < y$ anywhere else in the tree. Let $p^0[x]$ be x, let $p^1[x]$ be the parent of x, $p^2[x]$ be the grandparent of x, etc. The root of the tree is $p^m[x]$ for some m. We show by induction on k that y is the successor of x in the subtree rooted at $p^k[x]$, for $k = 0, \dots, m$. We've already shown it for $k = 0$. Suppose we've shown it for k; we show it for $k+1$. If $p^k[x]$ is the left child of $p^{k+1}[x]$. Then $p^{k+1}[x]$ and all the nodes in its right subtree must all be bigger than y, so y is still the successor of x in the tree rooted at $p^{k+1}[x]$. If $p^k[x]$ is the right child of $p^{k+1}[x]$, then $p^{k+1}[x]$ and all the nodes in its left subtree are smaller than x, so y must also be the successor of x in the subtree rooted at $p^{k+1}[x]$. This completes the inductive step. It follows that y is the successor of x in the tree rooted at $p^m[x]$, i.e., in the whole tree. Next suppose that x has no right child. Then TREE-SUCCESSOR goes up the tree until it finds the first ancestor $p^k[x]$ such that $p^k[x]$ is the left child of $p^{k+1}[x]$. In that case, we return $p^{k+1}[x]$. (Note that if there is no such ancestor, x is the rightmost node in the tree, and hence the largest node.) We want to show that $p^{k+1}[x]$ is the successor of x in the tree. Since $p^k[x]$ is a left child of $p^{k+1}[x]$, $p^k[x]$ and all the nodes below it (including x) are smaller than $p^{k+1}[x]$, so that $p^{k+1}[x] > x$. Moreover, since x is the rightmost node in the subtree rooted at $p^k[x]$, it is the largest node in that subtree. Thus, $p^{k+1}[x]$ is the successor of x is the subtree rooted at $p^{k+1}[x]$. We now do exactly the same inductive argument as before to show that $p^{k+1}[x]$ is still the successor of x in the subtree rooted at $p^{k'}[x]$ for all k' up to m.

Grading scheme: 4 points for each of the two cases. First part: -1 if did not prove (by induction) that no other element in the tree can be a successor. Second part: -1 if similar inductive argument not mentioned.

13.2-4 A priori it seems that this algorithm would take time $O(n \lg n)$, since each call to TREE-SUCCESSOR could take time $O(\lg n)$. However, it's not as bad as that. It's easy to see that the sequence of calls to TREE-SUCCESSOR traverses the tree following an inorder tree walk. We now show by strong induction on the size of the tree that when we do a walk on a BST, starting at the root, from there going to the smallest element, then the next smallest element, and so on, then going from the largest element back up to the root, we traverse each edge of the tree twice, once going down and once going up. This depends crucially on the BST property. Since a tree with n nodes has $n-1$ edges (with each node other than the root, we can associate the edge going up to its parent; this takes care of all the edges in the tree), this shows that the walk takes time $O(n)$.

The base case of the induction is immediate. Now suppose we have a BST with n nodes. Let r be the root of the BST, and let x and y be the left and right children of r, respectively (one of these may be NIL). We do our walk as follows:

- we first go from r to its left child x (if there is one)
- we then walk through the tree T_x rooted at x (since all the nodes in T_x are less than x
- the next node we visit is r, which means we need to come back to x, then go from x to r
- we then go from r to y and traverse the subtree T_y rooted at y.
- we then go from the largest node in T_y to r, which mean we must pass y.

It is clear from this description that the edges from r to x and from r to y are each traversed twice, once going down and once going up. The induction hypothesis assures us the the walks through T_x and T_y, starting and ending at x (resp., y) results in going through each edge in T_x and T_y twice. This completes the inductive step of the proof.

Grading criteria: pointing out the fact that each edge is traversed twice will get 2 points, to show the walk takes time $O(n)$ with the fact and the BST property will get 1 point. The induction proof of the fact will take 5 points (Base case: 1 point, inductive case: 4 points). Several students used the substitution method directly and got full marks.

Comments: There were a few serious deficiencies in students' solutions to this problem. First, induction on tree structures. An important thing to keep in mind when formulating an induction strategy for trees is that your inductive step must ensure that *all possible trees* can be constructed. An example of a deficient inductive hypothesis/step is:

Basis: ...P holds for a tree of one node.

Inductive Hypothesis: Assume that for all trees with n nodes, P holds.

Step Case: By I.H., P holds for any tree with n nodes. Attach a left child to the smallest node in the tree... P is maintained for this tree. Therefore P holds for a tree of $n+1$ nodes.

While this proves P for a subset of all possible trees, it fails on the overwhelming majority of them! Try to prove P on a full tree of height 2 using this base case and inductive step.

The other major problem related to the use of Θ-notation. A common induction claim was: "Let $P(n)$ mean that SUCCESSOR-INORDER runs in time $\Theta(n)$ on a tree of n nodes." While it is possible to express correct reasoning within a proof of this sort, the proof doesn't quite work. Θ-notation deals exclusively with the *asymptotic* behaviour of a function. To say that for $n = 4$, some $f(n) = \Theta(n)$, goes nowhere toward proving that $f(n) = \Theta(n)$. When proving a bound, it is far more correct to guess a function bound (such as "Let $P(n)$ mean that SUCCESSOR-INORDER runs in time between $c_1 n$ and $c_2 n$ on a tree of n nodes.", where you specify c_1 and c_2.) It is then trivial to show that the running time is $\Theta(n)$.

Extra problem 1: SKIPDELETE(S, k)

```
1  $y \leftarrow$ SKIP-SEARCH$(S, k)$
2  if $key[y] \neq k$
3    then return "$k$ is not in list"
4    else while $key[y] = k$
5           do while $y \neq$ NIL
6                do $after[before[y]] \leftarrow after[y]$
7                   $before[after[y]] \leftarrow before[y]$
8                   $y \leftarrow above[y]$
9           $y \leftarrow$ SKIP-SEARCH$(S, k)$
```

Comment: there were a number of bugs in the preliminary solution to the problem. Note that the while loop in line 3–9 is necessary to delete *all* ooccurrences of items with key k. Thus, after one such element is deleted, we do another Skip-Search to see if there are more. Thiw while loop is unncessary if we know that keys are unique.