Universität Innsbruck - Institut für Informatik
Prof. Clemens Ballarin, Robert Binna, Friedrich Neurauter, Francois Scharffe und Sarah Winkler

10 June 2008

Proseminar Algorithmen und Datenstrukturen

Exercise Sheet 11

Exercise 1 (Building Heaps)

A heap can be constructed from an array A of size n in two ways, either top-down or bottom-up.

The *top-down* method starts by considering $A[1]$ as a heap. In the $i+1$-th iteration $A[1,\ldots,i]$ is assumed to be a heap and $A[i+1]$ is added, i.e. pulled up until the heap condition holds. This yields the extended heap $A[1,\ldots,i+1]$. The approach is iterated until $i = n$, so the whole array is turned into a heap.

In the *bottom-up* method, one starts with $i = \lfloor n/2 \rfloor$. In each iteration all subtrees in $A[i+1,\ldots,n]$ are assumed to satisfy the heap condition. $A[i]$ has one or two children at positions $A[2i]$ and $A[2i+1]$, which are by assumption the roots of valid heaps. After sinking $A[i]$, all subtrees in $A[i,\ldots,n]$ are heaps. By decreasing i this approach is repeated until $i = 1$, so the whole array satisfies the heap condition.

Now consider the example array

$$A = [3, 12, 9, 5, 4, 8, 1, 13, 12]$$

and perform the following operations (on paper):

a) Construct the initial heap for A in a *top-down* fashion.

b) Construct the initial heap for A using the *bottom-up* approach.

c) Apply Heapsort to one of the heaps obtained above. Is Heapsort stable?

Exercise 2 (Combining Heaps)

Provide pseudo-code for an algorithm to build one heap that contains all elements of two given heaps with n and m elements, respectively (where n and m are positive). Assume that the heaps are given in a tree representation, i.e. each node has links to its two children. The running time of the algorithm should be $O(\log(n+m))$ in the worst case.

Exercise 3 (Heapsort in C)

Implement functions *sink* and *buildHeap* in C, and use them to incorporate *heapsort* into the framework of Exercise 4 of last week.

Exercise 4 (Lower Bound for Searching)

In the lecture you used decision trees to derive an information-theoretic lower bound for comparison-based sorting: Given a comparison operation that can check for two elements a and b whether $a \leq b$ or $a > b$ holds, it was shown that any sorting algorithm using only such an operation requires $\Omega(n \log n)$ comparisons.

Use the same technique to show that searching a value in a sorted array requires $\Omega(\log(n))$ comparisons.