

Introduction to algorithms and data structures.

IT University of Copenhagen.

June 13th, 2000

This exam consists of 3 exercises, each with 5 subexercises. All exercises has the same weight, and all subexercises has the same weight. You have 4 hour to complete the exam. Remember to number the pages and write your study-number and name on all pages. The exam consists of 7 numbered pages.

CLR refers to “Introduction to Algorithms” by Cormen, Leiserson and Rivest, 18. press, 1997.

Exercise 1

This assignment are about heaps and sorting.

a) Illustrate the execution of $\text{HEAPIFY}(A, 1)$ for the heap A in figure 1. Use the style of figure 7.2 in CLR page 143.

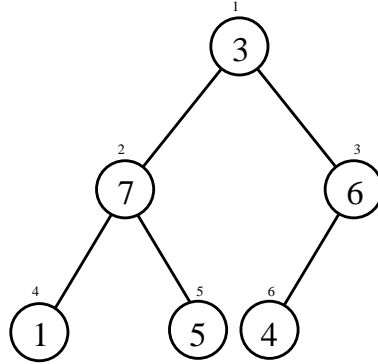


Figure 1: The heap A , on which to execute HEAPIFY .

b) QUICKSORT (CLR page 154) are defined using $\text{PARTITION}(A, p, r)$, which can swap the numbers in the array A from index p to r . Illustrate the calculation of $\text{PARTITION}(A, 1, 6)$, where

$$A = \langle 7, 8, 9, 4, 5, 6 \rangle$$

You can use Fig. 8.1 page 155 in CLR as a model for the illustration.

c) Construct a sorting algorithm, which has linear time complexity if the numbers to sort are already in order; otherwise the complexity $O(n \lg n)$. Write pseudo-code.

QUICKSORT are expectedly more effective than INSERTION-SORT (CLR page 3) for large inputs, but experiments has shown, that INSERTION-SORT are more effective than QUICKSORT , if few numbers are to be sorted.

d) Write pseudo-code for a sorting algorithm, which is a combination of QUICKSORT and INSERTION-SORT . The algorithm must use that INSERTION-SORT for practical purposes is more effective than QUICKSORT , if the numbers to sort are fewer than c numbers for some constant c . It is allowed to refer to code in CLR.

e) What is the time complexity of HEAPSORT if the numbers to sort are identical?

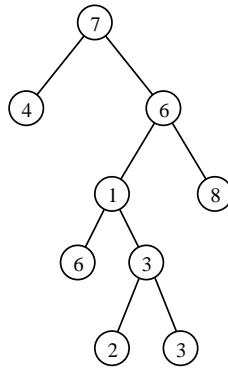
Exercise 2

This exercise is about binary trees with non-negative (≥ 0) integer weights in the nodes.

We define the *weighted length* of a simple path in a tree to be the sum of the weights in the nodes on the path, including end nodes. That is the weighted length from the root in the tree in figure 2 to the leaf with weight 2 has a weighted length of 19.

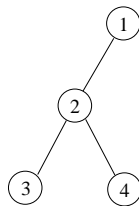
Among several paths in a tree a *weighted longest path* is one of the paths with the largest weighted length. For instance in figure 2 the weighted longest path (among all possible) has the value 25.

The *weight-depth* of a tree is the value of a longest weighted path among the paths from the root to a leaf in the tree. For example the weight depth is 21 for the tree in figure 2.



Figur 2: Binary tree with weights.

a) Show by a counter example, that the weighted longest path in a tree T doesn't necessarily go through the root of T .



Figur 3: Counter example.

In this exercise we consider the usual representation of a binary tree T , which is used in CLR chapter 13. That is to each node v in the tree T we have the fields $left[v]$, $right[v]$, and $p[v]$ for left child, right child, and the parent node respectively. For the root r is $p[r] = \text{NIL}$ and for a leaf w is both $left(w) = \text{NIL}$

and $right(w) = \text{NIL}$. Furthermore we let $key[v]$ denote the *weight* in the node v . Remark that we *doesn't* assume that the weights meets the *binary-search-property* in this assignment.

b) Make a pocedure $\text{WEIGHTTOROOT}(v)$, which calculates the weighted length of the path from the node v to the root of T . The procedure must run in time $O(h)$, where h is the height of T .

c) Construct a recursive procedure WEIGHTDEPTH , which calculates the weight-depth of a binary tree T .

We are now interested in expanding the datastructure for the weighted binary tree T , in such a way that we can find the weight-depth fast, when the weights in the nodes can be updated dynamically. More precisely we are interested in expanding the datastructure for T to be able to support the following two operations:

$\text{UPDATE}(v, x)$: Change T , so node v gets weight x .

$\text{ACTUALWEIGHTDEPTH}()$: Returns the actual weight depth for T .

d) Descripe an expansion of the datastructure for a tree T , which supports the two operations above. The running time for the UPDATE must be $O(h)$, where h is the height of T , while ACTUALWEIGHTDEPTH must run in time $O(1)$.

Hint: Make an extra field for each node v , which contains the actual weight-depth of the subtree with root v .

We are now intereseted in another expnsion of the datastructure for T . First the tree T must be initialized, so all node has weight 1. Then we must handle the two following operations:

$\text{SETZERO}(v)$: Change the weight in node v to 0, that is set $key[v]$ to 0.

$\text{ZEROPATH}(u, v)$: Returns true, if and only if the weighted length of the path from node u to node v is 0.

e) Descripe a datastructure for the above problem. For a tree T with n nodes the solution must give a *total* running time of $O(n \lg^* n)$ for initialization an execution of n SETZERO and ZEROPATH operations.

Exercise 3

For all subexercises in this exercise it is the case that A is an array consisting of n integers. The first integer in the array is $A[0]$ and the last in the table is $A[n - 1]$.

In figure 4 is shown an array with 8 integers. For this array is $n = 8$, $A[0] = 15$ and $A[n - 1] = 314$

15	20	31	150	210	214	217	314
----	----	----	-----	-----	-----	-----	-----

Figur 4: An array with 8 numbers.

a) Let a be a non empty (not necessarily sorted) array with n integers. Describe how to decide in time $O(n \lg n)$ whether there are two equal numbers in the array.

b) Let A be a (not necessarily sorted) array with n integers, where all n numbers has a value between 1 and $3n$. How fast can the n numbers be sorted?

c) Let A be a sorted array with n numbers. Let $\text{SUCC}(x)$ be a procedure, which returns the smallest number y from A , which is greater than x . If such a number doesn't exist ∞ is returned. $\text{SUCC}(170)$ on the array in figure 4 is therefore 210. Describe how to implement SUCC with time complexity $O(\lg n)$.

Let $\text{NEXT}(i)$ be a procedure that given an index i from an array A returns the smallest index j , which is larger than i , where the number $A[j]$ is equal. If such an index doesn't exist $n - 1$ is returned. Consider the array in figure 4. In this case $\text{NEXT}(0)$ will return 1, $\text{NEXT}(1)$ will return 3, and $\text{NEXT}(5)$ will return 7. Assume that if $\text{NEXT}(i)$ returns j , then NEXT has time complexity $O(1 + j - i)$. Consider the following procedure AMOR : $\text{AMOR}(k)$

1. $x \leftarrow 0$
2. **for** $j \leftarrow 1$ **to** k

3. **do** $x \leftarrow \text{NEXT}(x)$

d) Let A be an array with n numbers. What does a call to $\text{NEXT}(n - 1)$ return? What is the time complexity for a call to $\text{NEXT}(n - 1)$? What is the time complexity for a call to $\text{AMOR}(k)$? For each of the following time complexity measures, denote an argument for, whether they express the time complexity of $\text{AMOR}(k)$: $O(n)$, $O(k)$, $O(n + k)$ or $O(n * k)$.

In the next subexercise we will consider an array A , where all numbers in the array are 1 initially. Furthermore we assume that the array contains n numbers, where n can be written as $2^k + 1$ for an integer $k \geq 0$ (that is n is one of the numbers 2, 3, 5, 9, 17, 33, ...).

In figure 5 are shown an array with 9 integers. For this array is $n = 2^3 + 1 = 9$, $k = 3$, and $A[i] = 1$, for all indices i .

1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---

Figur 5: An array with 9 ones.

Consider the following two procedures NEWNUMBERS and HALVE :

$\text{NEWNUMBERS}()$

1. $i \leftarrow 0$
2. **while** $i \leq n - 1$
3. **do** $A[i] \leftarrow A[i] * 2$
4. $i \leftarrow i + A[i]$

$\text{HALVE}(k)$

1. **for** $j \leftarrow 1$ **to** k
2. **do** $\text{NEWNUMBERS}()$

e) Let A be an array with 9 numbers, which are all ones. Draw the situation after a call to `NEWNUMBERS`. Now let A be an array with n numbers, where $n = 2^k + 1$ for an integer $k \geq 0$. What is the time complexity of a call to `HALVE(k)`?