

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 9 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.

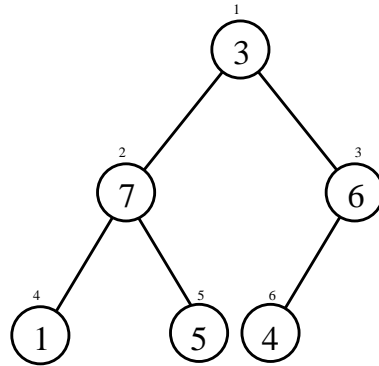


Figure 1: The heap A , on which to execute HEAPIFY .

Answer a) First 3 are swapped with 7, next 3 with 5 after which the algorithm stops.

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.

Answer b) The pivot-element is 7. The result is thus $A = \langle 6, 5, 4, 9, 8, 7 \rangle$

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.

Answer c) CHECK(A)

1. **for** $i = 2$ **to** LENGTH(A) **do**
2. **if** $A[i - 1] \leq A[i]$ **return false**
3. **return true**

SORT1(A)

1. **if** CHECK(A) **return** A
2. **else** MERGESORT(A)

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.

Answer d) SORT2(A, p, r)

1. $d \leftarrow r - p$
2. **if** $d \geq c$ **then**
3. $q \leftarrow$ PARTITION(A, p, r)
4. SORT2(A, p, q)
5. SORT2($A, q + 1, r$)
6. **else** INSERTIONSORT(A, p, r)

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

Answer e) $O(n)$ time, since each call to HEAPIFY takes $O(1)$ time in this case.

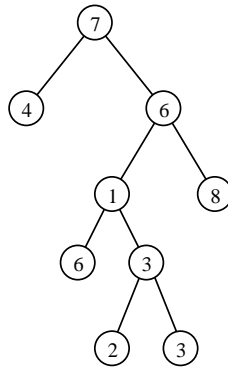
Exercise 2

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.

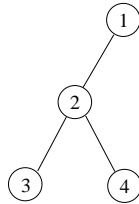
The *weight-depth* of a tree is the value of a longest weighted path among the paths from the root to a leaf in the tree. For example the weight depth is 21 for the tree in figure 2.



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 .

Answer a) See figure 3



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 procedure $\text{WEIGHTTOROOT}(v)$, 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 .

Answer b) $\text{WEIGHTTOROOT}(v)$

1. $s \leftarrow key[v]$
2. **while** $p[v] \neq \text{nil}$ **do**
3. $v \leftarrow p[v]$
4. $s \leftarrow s + key[v]$
5. **return** s .

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

Answer c) $\text{WEIGHTDEPTH}(v)$

1. **if** $v = \text{nil}$ **return** 0
2. **else return**
3. $\max(\text{WEIGHTDEPTH}(left[v]), \text{WEIGHTDEPTH}(right[v])) + key[v]$

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) Describe 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 .

Answer d) We add a field $weight[v]$ to each node v , which contains the actual weight depth of the subtree with root v . The operations are as follows:

ACTUALWEIGHTDEPTH(): Returns $weight[v]$.

UPDATE(v, x): Let r be the root of the tree. Set $weight[v] \leftarrow x$ and for all nodes w on the path from v to r set

$$weight[w] \leftarrow \max(weight(left[w]), weight(right[w])) + key[v]$$

We are now interested in another expansion 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:

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

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

e) Describe 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.

Answer e) We use the Union-find datastructure. For each node v we associate an element $e(v)$ and a weight $weight[e(v)]$. First we initialize n sets for each element with MAKE-SET operations. All the weights are initially set to 1. The operations are now:

SETZERO(v): Set $weight[e(v)] \leftarrow 0$. For all nodes w incident to v , if $weight[e(w)] = 0$ run UNION($e(v), e(w)$).

ZEROPATH(u, v): Returns true, if FIND-SET(u)=FIND-SET(v). Otherwise false is returned.

We have decided to let the length of the weighted path from a node to itself be 0. It is trivial to change the datastructure if one doesn't want that. Because $O(n)$ MAKE-SET, UNION and FIND-SET can be executed in $O(n \lg^* n)$ time, the datastructure has the wanted time complexities.

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.

Answer a) Sort the table and investigate in linear time, whether two elements $A[i - 1]$ and $A[i]$ are equal for $2 \leq i \leq n$.

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?

Answer b) By using for instance COUNTING-SORT, A can be sorted in time $O(n + 3n) = O(n)$.

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 $O(\lg n)$.

Answer c) Perform a binary search for the element.

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 $O(n * k)$.

Answer d) $\text{NEXT}(n - 1)$ returns $n - 1$ per definition and as a consequence takes $O(1)$ time. The complexity for $\text{AMOR}(k)$ is $O(n + k)$ because we at most runs through A once and at the most executes k calls to NEXT

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. $i \leftarrow i + A[i]$

HALVE(k)

1. **for** $j \leftarrow 1$ **to** k
2. **do** 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)?

Answer e) After a call to NEWNUMBERS is $A = \langle 2, 1, 2, 1, 2, 1, 2, 1, 2 \rangle$. A call to HALVE results in k calls to NEWNUMBERS. NEWNUMBERS doubles the length of the interval it jumps each time it is called. Therefore the time complexity of HALVE(k) is $= O(n/2 + n/4 + \dots) = O(n)$.