

# Introduction to algorithms and datastructures

## Solutions

IT University of Copenhagen

June 13, 2001

This exam consists of 3 exercises containing in total 13 subexercises. Each of these 13 subexercises are given the same weight in the evaluation. You have 4 hours to complete the exam. Remember to number the pages and write your name and CPR number on every page. The exam consists of 7 numbered pages.

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

For exercises in which algorithms must be specified, the asymptotic time complexity of the specified solution will be taken into account when grading. Exercises asking for time complexity must be answered using O-notation. It is weighted in the evaluation that growth rates in the O-notation are expressed with least possible asymptotic growth.

## Exercise 1

This exercise is about  $O$ -notation.

a) What is  $O(\log n) + O(1) + O(n)$ ?

**Answer a)**  $O(n)$ .

Let there be given a heap, which supports the following operations BUILD-HEAP, EXTRACT-MIN and DECREASE-KEY.

b) Assume that the heap above supports EXTRACT-MIN in  $O(\log \log n)$  time, DECREASE-KEY in  $O(1)$  time and BUILD-HEAP in linear time. Give the time complexity for Dijkstra's algorithm in a graph with  $m$  edges and  $n$  nodes if the above heap is used.

**Answer b)** There are  $m$  DECREASE-KEY and  $n$  EXTRACT-MIN operations. This gives a running time of  $O(m + n \log \log n)$ .

Look at the following method:

```
F1( $n$ )
1  if  $n > 2$ 
2     then return (F1( $n - 1$ ) + F1( $n - 2$ )) mod  $n$ 
3  else return 2
```

c) Explain how F1 can be implemented with a running time of  $O(n)$ .

**Answer c)** We use dynamic programming. A bottom-up or a top-down approach can be applied, using a table. Let  $F1[n]$  be a table with 0-values initially. First we set  $F1[1] = F1[2] = 2$ . Then for  $i = 3, \dots, n$  we calculate  $F1[i] = F1[i - 1] + F1[i - 2] \pmod n$ . This takes  $O(n)$  time and  $f1(n) = F1[n]$ .

Look at the following three methods:

F2( $n$ )

```
1 if  $n > 0$ 
2   then F2( $\lfloor n/2 \rfloor$ )
3     F2( $\lfloor n/2 \rfloor$ )
```

F3( $n$ )

```
1 if  $n > 0$ 
2   then for  $i \leftarrow 1$  to  $n$ 
3     do  $j \leftarrow 1$ 
4     F3( $\lfloor n/2 \rfloor$ )
5     F3( $\lfloor n/2 \rfloor$ )
```

F4( $n$ )

```
1 if  $n > 0$ 
2   then for  $i \leftarrow 1$  to  $n$ 
3     do  $j \leftarrow 1$ 
4     F4( $\lfloor n/2 \rfloor$ )
```

**d)** Assume that the procedures are only called with an integer  $n$ . For each of the four procedures above give the time complexity in  $n$ .

**Answer d)** Let  $t(n)$  denote the time complexity for the procedure in question.

F1( $n$ ):  $t(n) = 2 \cdot t(n/2)$  and  $t(0) = 1$ . Therefore is  $t(n) = O(2^{\log n}) = O(n)$ .

F2( $n$ ):  $t(n) = 2 \cdot t(n/2) + n$  as in for instance MERGESORT.  $t(n) = O(n \log n)$ .

F3( $n$ ):  $t(n) = n + n/2 + n/4 + \dots = O(n)$ .

## Assignment 2

This assignment is about heaps and amortized analysis. A heap is defined as a data structure, which supports the operations `EXTRACT-MAX` and `INSERT`.

`INSERT(A, k)` : Inserts the value  $k$  in the heap  $A$ .

`EXTRACT-MAX(A)` : Removes and returns a maximum value from the heap  $A$ .

Let these two operations be implemented as described in CLR chapter 7.

**a)** Illustrate the execution of `EXTRACT-MAX(A)` for the heap  $A$  in figure ???. Use the style from figure 7.5 in CLR page 151. Remark that in this figure insertion is shown. You have to show extraction.

**Answer a)** At first 9 and 4 are inserted at 9's old place. The following call to `HEAPIFY` makes 4 swap place with 7, whereafter the heap property is restored and the algorithm stops.

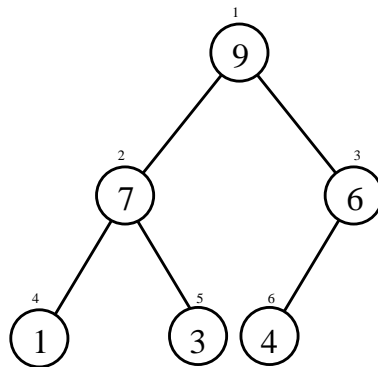


Figure 1: The heap  $A$ , which `EXTRACT-MAX` are to be executed at.

Now the heap must support another operation:

`REMOVELARGEST(A, k)` : Removes the  $k$  largest elements from  $A$ . If  $A$  has less than  $k$  elements,  $A$  are emptied completely.

**b)** Assume that the heap can contain up to  $n$  elements. Describe how `REMOVELARGEST(A, k)` can be supported with a worst case running time of  $O(k \log n)$ .

**Answer b)** Call `EXTRACT-MAX`  $\min(n, k)$  times. this takes  $O(k \log n)$  time.

The heap must now support another operation:

**EXTRACTMORETHAN**( $A, x$ ) : Removes and returns all values from the heap  $A$ , which are larger than  $x$ .

**c)** Assume that the heap can contain up to  $n$  values. Describe how the operations **INSERT**, **EXTRACT-MAX** and **EXTRACTMORETHAN** can be implemented in such a way, that the heap operations has an amotized running time of  $O(\log n)$ . That is  $m$  heap operations are done in time  $O(m \log n)$

**Answer c)** **EXTRACTMORETHAN**( $A, x$ ) can be implemented calling **EXTRACT-MAX**( $A$ ) until the maximum element in the heap ( $A[1]$ ) are less than or equal to  $x$ .  $m$  **INSERT** and **EXTRACT-MAX** operations takes at most  $O(m \log n)$  time. **EXTRACTMORETHAN** makes at most as many **EXTRACT-MAX** as there has been insertions. This means that  $m$  **EXTRACTMORETHAN** and **INSERT** operations takes at most  $O(m \log n)$  time.

The heap must now support another operation:

**EXTRACT-MIN**( $A$ ) : Removes and retrns a minimal value from the heap  $A$ .

**d)** Assume that the heap can contain up to  $n$  values. Describe how the operations **INSERT**, **EXTRACT-MAX** and **EXTRACT-MIN** can be implemented in such a way that the heap operations has a worst case time complexity of  $O(\log n)$  per operation. This means, that each of the operations must be supported in time  $O(\log n)$ .

**Answer d)**

**INSERT**( $x$ ): We insert  $x$  in two objects and connect these with pointers. The one object are inserted into a min-heap and the other in a max-heap.

**EXTRACT-MAX**: We extract an element from the max-heap and finds the copy of it in the min-heap, which we call **DELETE** on.

**EXTRACT-MIN**: As in **EXTRACT-MAX**.

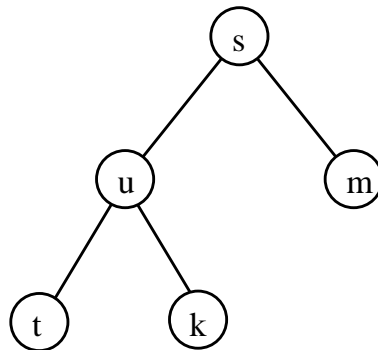
All operations runs in  $O(\log n)$  time.

### Exercise 3

This exercise is about recursion and rooted binary trees. Each node has either two or no children. The node  $x$ 's left child is  $left[x]$ , and its right child is  $right[x]$ . If the node  $x$  doesn't have any children,  $left[x]$  and  $right[x]$  has the special value NIL and  $x$  is called a leaf. If  $x$  has any children it is called an internal node. In case the root node for a tree are NIL, the tree is empty. For each node in the tree a color are assigned; the node  $x$  has color  $color[x]$ .

Look at the following procedure:

```
PRINT( $x$ )
1  if  $x \neq$  NIL
2    then print  $color[x]$ .
3      if  $left[x] \neq$  NIL
4        then PRINT( $left[x]$ )
5          PRINT( $right[x]$ )
```



Figur 2: Tree With nodes, that has colors denoted by letters.

Let  $x$  be the root node for the tree in figure ?? . A call to the method PRINT( $x$ ) will print the color sequence “sutkm”.

a) Change the method PRINT, in such a way that the recursive run-through by the call to PRINT( $x$ ) for the root node  $x$  in the tree from figure ?? prints the color sequence “smukt’ in stead.

**Answer a)** Swap line 4 and 5.

With the definition of binary trees we use in this exercise, we can calculate the number of nodes and leafs in a tree using the following two procedures:

NODES( $x$ )

```
1 if  $x = \text{NIL}$ 
2   then return 0
3   else return 1+NODES(left[ $x$ ]) + NODES(right[ $x$ ])
```

LEAVES( $x$ )

```
1 if  $x = \text{NIL}$ 
2   then return 0
3   else if left[ $x$ ] = NIL
4     then return 1
5     else return LEAVES(left[ $x$ ]) +LEAVES(right[ $x$ ])
```

**b)** An internal node in a tree is a node which aren't a leaf. Construct a method INTERNAL( $x$ ), which given a root node  $x$  for a tree, returns the number of internal nodes in the tree. Give the time complexity of your solution.

**Answer b)**

INTERNAL ( $x$ )

```
1 return NODES( $x$ )–LEAVES( $x$ )
```

Because LEAVES( $x$ ) and NODES( $x$ ) both runs in time  $O(n)$ , INTERNAL( $x$ ) takes  $O(n)$  time.

In the following three subexercises, effective recursive methods are to be constructed.

**c)** A tree has a red root path, if there are a path in the tree from the root  $x$  to a leaf, where all nodes on the path from  $x$  to the leaf has the color red. Construct a method REDROOTPATH( $x$ ), which returns the number 1 if the tree with root  $x$  has a red root path. If such a path doesn't exist, the method must return 0. Give the time complexity of your solution.

**Answer c)** In this and the following answers it is assumed that the algorithms are not called on empty trees.

REDROOTPATH( $x$ )

1 **if**  $left[x] = \text{NIL}$

2     **then return**  $color[x] == \text{"red"}$

3     **else return**  $color[x] == \text{"red"}$  **and**

4          $color[left[x]] == \text{"red"}$  **or**  $color[right[x]] == \text{"red"}$

The algorithm runs in  $O(n)$  time.

**d)** A tree has a red root path, if there are a path between two different leafs in the tree, where all the nodes on this path has the color red. Construct a procedure REDLEAFPATH( $x$ ), which returns the number 1, if the tree with root  $x$  has a red leaf path. If such a path doesn't exist, the method shall return a number different from 1. Give the time complexity of your solution.

**Answer d)**

REDLEAFPATH( $x$ )

```
1 if  $left[x] = \text{NIL}$ 
2   if  $color[x] == \text{"red"}$  return 2
3   else return 3
4 else
5    $rl \leftarrow \text{REDLEAFPATH}(left[x])$ 
6    $rr \leftarrow \text{REDLEAFPATH}(right[x])$ 
7   if  $rl == 1$  or  $rr == 1$  return 1
8   else if  $color[x] == \text{"red"}$ 
9     if  $rl == 2$  and  $rr == 2$  return 1
10    else if  $rl == 2$  or  $rr == 2$  return 2
11    return 3
```

The algorithm returns 1 if  $x$  has a red leaf path, 2 if  $x$  has a red root path and 3 otherwise. Using dynamic programming, where we save the result of calling REDLEAFPATH for all nodes in the tree, the algorithm can be implemented in  $O(n)$  time.

e) A tree is said to be complete if all leafs has the same depth. Construct a method COMPLETE( $x$ ), which given a root node  $x$  for a tree returns  $-1$  if the tree is not complete and otherwise the height of the tree. Give the time complexity of your solution.

**Answer e)**

COMPLETE( $x$ )

1 **if**  $left[x] = \text{NIL}$  **RETURN** 0

2 **else**

3      $kl \leftarrow \text{COMPLETE}(left[x])$

4      $kr \leftarrow \text{COMPLETE}(right[x])$

5     **if**  $kl \neq -1$  **and**  $kl == kr$  **return**  $kl + 1$

6     **else return**  $-1$

As before the algorithm can be implemented so that it runs in  $O(n)$  time.