University of Houston

COSC 3320: Algorithms and Data Structures
Spring 2016

Solutions for Homework 4

1. Given an array $A[1, 2, \ldots, n]$ of n elements, a *majority* element of A is an element occurring at least $\lceil (n+1)/2 \rceil$ times. The elements cannot be ordered or sorted, but can be compared for equality. Design an efficient divide and conquer algorithm that returns a majority element of A (if any), and determine its complexity.

   Solution:

   The algorithm partitions the array A into two halves, and recursively computes the majority (if any) in each half. The algorithm then checks (in linear time) which of these at most two elements is the majority in the overall array A. There are the following three cases:

   - Both recursive calls return "no majority". Then A cannot have a majority element.
   - One of the two halves has a majority element. Then compare every element in the combined array and count the number of elements that are equal to this value. If it is a majority element then return that element, else return null.
   - Both recursive calls return a majority element. Then count the number of elements equal to both of the candidates for majority element. If either is a majority element in the combined array then return it, else return null.

   The pseudocode follows.

```
Majority(A)
input: Array A[1,2,...,n] of n elements
output: A majority element of A it there is one, null otherwise
if (n = 1) then return A[1]
A1 <- A[1,...,\lfloor n/2 \rfloor]        \\ First half of the array
A2 <- A[\lfloor n/2 \rfloor + 1,...,n]    \\ Second half of the array
a <- Majority(A1)
b <- Majority(A2)
x <- 0
y <- 0
for i <- 1 to n do   \\ count the number of occurrences of a and b in A
   if (a = A[i]) then x++
   else if (b = A[i]) then y++
if x >= \lceil (n+1)/2 \rceil then return a
if y >= \lceil (n+1)/2 \rceil then return b
return null
```

   The cost of the divide and of the conquer steps is linear in n. Therefore, the complexity of the above algorithm is $T(n) = 2T(n/2) + O(n)$ when $n > 1$, and $T(n) = 1$ if $n = 1$, which solves (e.g., by the Master Theorem) to $T(n) = O(n \log n)$.

2. Design and analyze and algorithm `preorderNext(T,v)` that, given a binary tree T and a node $v \in T$, returns the node visited immediately after v in the preorder visit of T (and returns `null` if v is the last node visited in the preorder visit of T).

   Solution:

   Observe that if v is an internal node, then its successor in the preorder visit is its left child. Otherwise, we have to go up from v till finding the first ancestor u (that is the deepest ancestor) such that v is in the left subtree of u. In this case, the successor of v in the preorder visit is the right child of u. If such an ancestor u does not exist, it means that v is the leaf that is found going down always on the right starting from the root of T, and therefore it is the last visited node in the preorder visit of T. In this case the algorithm returns null. The pseudocode follows.

```
preorderNext(T,v)
input: Binary tree T, node v of T
output: Successor of v in preorder visit, if it exists, null otherwise
if (T.isInternal(v)) then return T.left(v)
while (!T.isRoot(v)) do
  if (v = T.left(T.parent(v))) then return T.right(T.parent(v))
    else v <- T.parent(v)
return null
```

   The number of iterations of the while is bounded from above by the depth of v, and in each iteration a constant number of operations is executed. The rest is just a constant number of operations. Since the depth of a node is at most the height of the tree, the complexity of the algorithm is $O(h)$, where h is the height of T.

3. Let T be a proper binary tree. Define the *heightsum of* T as the sum of all the heights of the nodes of T.

   (a) Determine an upper bound to the heightsum of a proper binary tree with n nodes, and describe a tree whose heightsum is such a value.

   (b) Design a divide and conquer algorithm `heightSum(T,v)` that computes the heightsum of T_v, where T_v denotes the subtree of T rooted at $v \in T$.

   (c) Analyze the complexity of `heightSum(T,T.root())`.

   Solution:

   (a) The m leaves have height 0, hence they can be ignored. Considering internal nodes in increasing order of height. Every such internal node can have height of at most a term $+1$ bigger then the previously considered node, hence since a proper binary tree has $n - m$ internal nodes, an upper bound is $1 + 2 + 3 + ... + (n - m)$. An example of tree with that heightsum is a tree where only the right child of a node has two children.

   (b) As already seen in the course, by calculating a richer information (sum of the heights and height) we can obtain an efficient algorithm.

```
heightSum(T,v)
input; Proper binary tree T, node v of T
output: Heightsum of T_v, height of v
```

```
if (T.isExternal(v)) then return (0,0)
(sL,hL) <- heightSum(T,T.left(v))
(sR,hR) <- heightSum(T,T.right(v))
h <- max{hL,hR} + 1
s <- sL + sR + h
return (s,h)
```

(c) The algorithm performs a postorder visit of T, where the visit of an internal node corresponds to the computation of quantities s and h, from the quantities obtained by the children, and hence it requires $O(1)$ time. Hence the complexity of `heightSum(T,T.root())` is $O(n)$, where n is the number of nodes of T.

Observation: simpler strategies such as performing any kind of visit of the tree, and at each visited node v calculating the height of v is not a recursive strategy. Moreover, its complexity would be much higher (quadratic in n).

4. You are told that $\pi = A, C, F, B, D, E, G$ is the sequence of nodes of a tree visited by some visit procedure.

(a) Exhibit one tree T whose node labels are A, B, C, D, E, F, G and whose preorder visit would visit the nodes of T in the order given by π.

(b) Exhibit one tree T whose node labels are A, B, C, D, E, F, G and whose postorder visit would visit the nodes of T in the order given by π.

(c) Exhibit one binary tree T whose node labels are A, B, C, D, E, F, G and whose inorder visit would visit the nodes of T in the order given by π.

Solution:

There are several valid solutions for each point. Here are some.

(a) root: A; children of A (left-to-right): C,D,E,G; children of C (left-to-right): F,B

(b) root: G; children of G (left-to-right): F,B,E; children of F (left-to-right): A,C; children of E: D

(c) root: B; children of B; (left-to-right): C,E; children of C (left-to-right): A,F; children of E (left-to-right): D,G