1. Let \( T \) be a tree (not necessarily a binary tree) where all edges may be traversed in either direction. The diameter of a vertex \( v \) in \( T \) is the number of edges on a longest path between two vertices (not necessarily leaves) in the subtree rooted at \( v \) (this path does not necessarily pass through \( v \)). Assuming that vertices are already labeled with their height, give a linear time recursive algorithm \( \text{DIAMETER}(v) \) to label each vertex with its diameter. Justify why the time is linear and analyze the amount of space used in addition to that already used to store the tree.

procedure \( \text{DIAMETER}(p) \)

\[
\begin{align*}
\text{if } p \text{ is a leaf then return } 0 \\
\text{else if } p \text{ has only one child } q \text{ then return } \max(\text{HEIGHT}(q)+1, \text{DIAMETER}(q)) \\
\text{else begin} \\
\quad i &= \text{largest height of a child of } p \\
\quad j &= \text{second largest height of a child of } p \\
\quad k &= \text{largest diameter of a child of } p \\
\quad \text{return } \max(i+j+2,k) \\
\text{end}
\end{align*}
\]

Note: By defining \( \text{HEIGHT}(\text{nil}) = -1 \), and \( \text{DIAMETER}(\text{nil}) = 0 \), the if and if else statements can be eliminated.

The values \( i, j, \) and \( k \) can be computed by a single pass through the children of \( p \), with a recursive call to \( \text{DIAMETER} \) on each child; hence, \( \text{DIAMETER}(p) \) amounts to a traversal of the subtree rooted at \( p \) that works in time proportional to the number of vertices in that subtree.

\( O(1) \) space is used in addition to that used by the tree and used by the recursion stack. The recursion stack gets no deeper than the tree height, which is no more than the number of vertices.
2. Present an algorithm that uses at most \( n + \lceil \log_2(n) \rceil - 2 \) comparisons to find the largest and second largest element of a list of \( n \) distinct elements:

Let \( T \) be a full binary tree with \( n \) leaves (a complete binary tree with part of the bottom row missing if \( n \) is not exactly 1 less than a power of 2). Every leaf in this tree has depth at most \( \lceil \log_2(n) \rceil \) (if not all of the last row is filled, then some leaves have depth \( \lceil \log_2(n) \rceil - 1 \)).

Instead of a linear pass through the list to find the largest in a standard way with a for loop, place the items of the list in the leaves of \( T \). Place in each vertex in the row of height 1 the maximum of its two children. Place in each vertex in the row of height 2 the maximum of its two children. And so on.

A total of at \( n-1 \) comparisons are used to place in each vertex the maximum of the elements in its subtree, and the root ends up with the maximum of all the elements.

Now, at most an additional \( \lceil \log_2(n) \rceil - 1 \) comparisons can be used to find the maximum among the at most \( \lceil \log_2(n) \rceil \) items to which the item stored at the root was compared on its way up to the root.

So a total of \( (n-1) + (\lceil \log_2(n) \rceil - 1) = n + \lceil \log_2(n) \rceil - 2 \) comparisons were used to find both the largest and second largest element in the list.
3. Let \( S \) be a set of \( n \) items on which an ordering is defined. Describe how to combine the ideas of open hashing and 2-3 trees into a data structure to store \( S \) that supports INSERT and MEMBER operations in \( O(1) \) expected time and \( O(\log(n)) \) worst-case time.

Store each bucket in a 2-3 tree. The expected size of a hash bucket (the number of vertices in its 2-3 tree) remains unchanged, however, the worst-case time to access any element in a bucket is now \( O(\log(n)) \).
4. With closed hashing, whenever an item is hashed to a position that is already occupied, search for some other place in the table to put it (so items are stored directly in the table). Present pseudo-code for the MEMBER and INSERT operations with linear probing, where you simply scan from that position forward (wrapping around if you get to the end of the table) until an empty position is found.

Assumptions:

- The hash table array $A$ is indexed from 0 to $m-1$ for some integer $m$ greater than 0, and all positions are initialized to a value nil that is an illegal value for data item. For example, if items are positive integers, nil could be 0, if items are pointers to records, then $nil$ could be a nil pointer, etc.
- The variable count is initialized to 0 and will be used to keep track of the current number of data items in the table.
- The variable MAX is initialized to some value in the range 0 to $m$ (e.g., $m/2$) and represents the largest legal size for the table.
- MEMBER returns nil if the item is not in the table.
- INSERT does nothing if the element is already in the table.
- $h$ is a function that maps data items to the range 0 to $m-1$.

procedure INSERT($d$)

if (count=MAX) then ERROR (table is full)

$i = h(d)$

while $A[i] \neq d$ and $A[i] \neq nil$ do $i = i+1$ MOD $m$

if $A[i]=nil$ then begin

$A[i] = d$

count = count+1

end

end

function MEMBER($d$)

$i = h(d)$

while $A[i] \neq d$ and $A[i] \neq nil$ do $i = i+1$ MOD $m$

return $i$

end