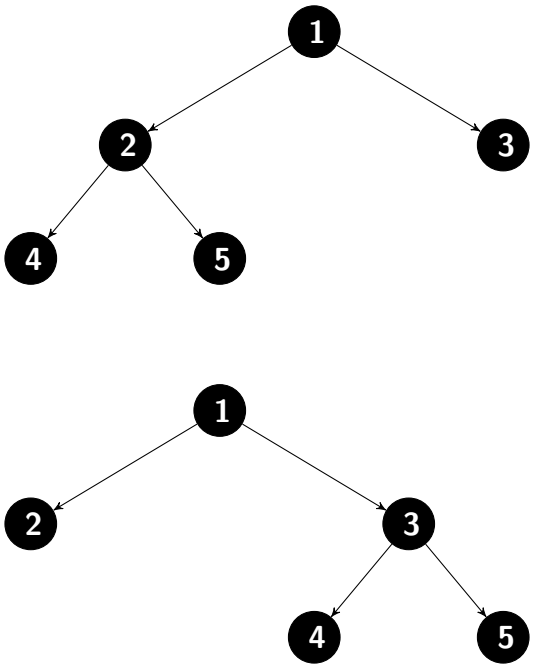
CPS 430/590.06 **Prof. John Reif**
Design and Analysis of Algorithms **Fall 2013**
Homework 2 **Matt Dickenson**

Question 1

Part 1 The following figures show all two (2) full binary trees with 5 vertices (F_5):



Part 2 Having seen the case of $F_5 = 2$, we now find the recurrence relation for F_n. First, we can see that any binary tree must have an odd number of vertices, because there will be one root vertex and an even number of non-root vertices. For the base cases F_1 and F_3, there is exactly one full binary tree: $F_1 = F_3 = 1$. Once we add two more nodes (F_5) we can place them in either of two positions on the F_3 tree, so $F_5 = 2 \times F_3 = 2$.

More generally, for each $n = 2k + 1$, $F_n = m \times F_{n-2} = m \times F_{2k-1}$, where m is the number of nodes with zero children on the F_{n-2} tree. Because each increase in size of two vertices leads to a one-unit increase in m, $m = k = \frac{n-1}{2}$. Thus, the recurrence relation is $F_n = \frac{n-1}{2} \times F_{n-2}$.

We can see that F_n is $\Omega(2^n)$ by induction:

$$
\begin{aligned}
F_1 &= 1 \\
F_3 &= 1 \\
&= 2^0 \\
F_5 &= 2 \\
&= 2^1 \\
F_n &= \frac{n-1}{2} \times F_n \\
&= \frac{n-1}{2} \times \frac{n-3}{2} \times F_{n-4} \\
&= k \times (k-1) \times \ldots \times 1 \\
&= k! \\
&= (2n+1)! \\
&\geq 2^n \\
&= \Omega(2^n)
\end{aligned}
$$

Question 2

To create a new list with duplicates removed, I would store the n elements in a binary search tree. I would store the first element of the list at the root, with all elements with lesser sort values in its left subtree and all elements with greater sort values in its right subtree (and so on for each subtree). If a value already exists in the tree, its duplicate will not be stored. The running time for this algorithm is $O(n)$ for going through the n-element list and $O(\log n)$ for each search to store an element or determine that it is a duplicate, for a total running time of $(n \log n)$. The tree could then be converted back into a list without duplicates.

Question 3

Since the array is sorted and without duplicates, we can begin checking whether there exists an index j such that $A[j] = j$ at $j = \frac{n}{2}$. If $A[j] = j$ we are done, if $A[j] > j$ we check to the left ($A[\frac{n}{4}]$), and if $A[j] < j$ we check to the right ($A[\frac{3n}{4}]$).

More generally, on the i^{th} iteration we visit j_i and assess whether $A[j_i] = j_i$. If If $A[j_i] = j_i$ we are done, otherwise we go halfway between $A[j_i]$ and $A[j_{i-1}]$ so that $j_{i+1} = \frac{j_i + j_{i-1}}{2}$. Because each iteration bisects the list, our search takes at most $\log n$ steps, for an $O(\log n)$ algorithm. Once we have performed $\log n$ iterations we have either found the desired index j or determined that there is no j such that $A[j] = j$.

Question 4

To sort an array A of integers of size n in O(n+(MAX-MIN)) time, we can use an adaptation of counting sort that leverages the additional information about the minimum and maximum values, with an auxiliary array C and output array B (as in CLRS):

1. Let C[0..MAX] be a new array
2. for $i = 0$ to MAX:
3. $\quad C[i] = 0$
4. for $j = 1$ to n:
5. $\quad C[A[j]] = C[A[j]] + 1$
6. for $i =$ MIN to MAX:
7. $\quad C[i] = C[i] + C[i-1]$
8. for $j = n$ to i:
9. $\quad B[C[A[j]]] = A[j]$
10. $\quad C[A[j]] = C[A[j]] - 1$

By line 5, C contains counts of how many times the element i occurred in A. On line 7, it contains counts of all elements less than or equal to i (we can ignore all indices less than MIN, assuming MIN$\geq$0). The correctness of this algorithm is apparent as it is an adaptation of counting sort: each value is inserted $C[A[j]] - C[A[j-1]]$ times (the number of times it appears in A) in the final output array B.

The running time is apparent from the number of iterations in the last two for-loops: the penultimate loop iterates (MAX-MIN) times and the final loop runs n times, for a total running time of $O(n + (MAX - MIN))$. (We can drop the first two loops because they are less than or equal to n, and multiples of n can be dropped in big-O notation.)

Question 5

To find the i^{th} smallest element of a union list of $n_1 + n_2$ elements, we can perform the following algorithm. First, combine lists L_1 and L_2 into a binary search tree, keeping track of each node's value, its right and left children, and the number of children in its left subtree (`leftChildren`). The root of T is now the median ($n/2$ smallest element) of the union list. All elements lesser than the median are in the root's left subtree and all greater values are in its right subtree, and so on. Thus, we can find the i^{th} smallest element using the following recursive algorithm:

1. find(`root`, i):
2. $\quad$ if `root.leftChildren` $== i$:
3. $\quad\quad$ return `root`
4. $\quad$ else if `root.leftChildren` $< i$:
5. $\quad\quad$ $j = i -$ `root.leftChildren`
6. $\quad\quad$ return find(`root.rightChild`, j)

7. else:

8. return find(`root.leftChild`, i)

The algorithm stops when it finds the node with i elements in its left subtree, i.e. the i^{th} smallest element in the union list. Because the search relies on a binary search tree, it takes at most $log(n_1 + n_2)$ operations.