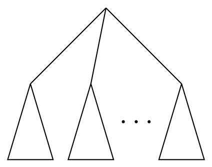
Lecture Notes, Fundamental Algorithms: Thinking Recursively, Part 2

Let's begin with a recursive definition of a tree.

The base case is a leaf, a 1 node tree: $\circ$.

In general, a tree consists of a root node and one or more subtrees, which are themselves trees.



Inconsistently, we will define binary trees using the empty tree as a base case. Then in the general case, a binary tree is a root node with two subtrees; each of the subtrees may or may not be empty.

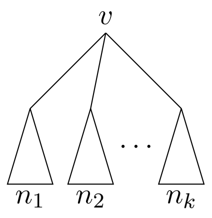
We will often use T to name a tree. The variable T is also used as a pointer to the root of this tree. In addition, when we use node v as a variable, we often use it as a pointer to this node and hence to the tree rooted at this node. To avoid clutter, we will suppress the dereferencing of pointer variables.

Let's turn to some problems on trees.

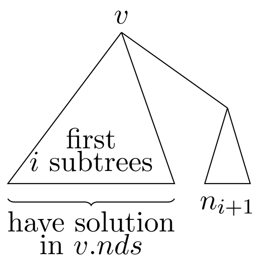
Problem 1 Report the number of nodes in a tree T.

It will be convenient to let each node v have a field $v.nds$, which will record the number of nodes in the subtree rooted at v.

For the base case, where v is a leaf, $v.nds \leftarrow 1$. For the general case, if node v has k children, with $n_1, n_2, \ldots, n_k$ nodes, respectively, $v.nds = 1 + \sum_{i=1}^{k} n_i$.



We compute this total by incorporating the count for each of v's subtrees one by one. The way to think about this is that we have the answer for the tree consisting of v plus its first i subtrees, and we are now expanding the solution to include the $(i+1)$-st subtree, and we do this for $i = 0, 1, \ldots, k-1$ in turn. Note that the case $i = 0$ is a tree comprising just node v, i.e. it is the leaf node case. For some problems we may need a separate base case for leaf nodes, but not here.



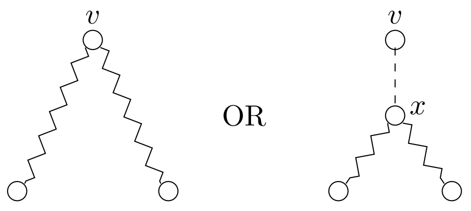
Below, we give a recursive program NodeCount, NC for short, to compute the values $v.$ nds for each node v in the tree.

```
NC($v$)
    $v.nds \leftarrow 1$;
    **for** each child $w$ of $v$ **do**
        NC($w$);
    $v.nds \leftarrow v.nds + w.nds$
    **end** (* for *)
**end** (* NC *)
```

If we want to compute this for a tree T, the initial call will be to NC(T). (Strictly speaking T is a pointer to (the root node of) the tree, and v is a pointer to a node.)

Problem 2 Report for every node v in tree T the length in edges of the longest path in v's subtree, storing the result for node v in $v.lp$.

This longest path could pass through node v or it might pass through a proper descendant x of v, as shown in the figure below.

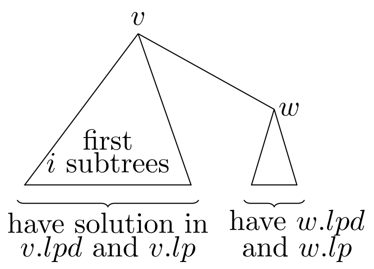


To aid in this computation, we start by finding the length of the longest path from v to a descendant leaf, which we will store in $v.lpd$ (the longest path to a descendant). Suppose v has $k \geq 1$ subtrees, with longest paths to descendants of lengths $\ell_1, \ell_2, \ldots, \ell_k$, resp. Then $v.lpd = 1 + \max_{1 \leq i \leq k} \ell_i$. While if v is a leaf, then $v.lpd = 0$. The following procedure computes these values.

```
LPD($v$)
    $v.lpd \leftarrow 0$;
    **for** each child $w$ of $v$ **do**
        LPD($w$);
        $v.lpd \leftarrow \max\{v.lpd, 1 + w.lpd\}$
    **end** (* for *)
**end** (* LPD *)
```

Again, the way to think about the computation in the general case is that we have the answer for the tree consisting of v plus its first i subtrees, and we are now expanding the solution to include the $(i+1)$-st subtree, and we do this for $i = 0, 1, \ldots, k-1$ in turn.

Now we are ready to compute $v.lp$. Suppose that we have the solution for v plus its first i subtrees, stored in $v.lp$, and the solution for its $(i+1)$-st subtree stored in $w.lp$, where w is the root of this subtree. In addition, we have the values $v.lpd$ and $w.lpd$ for the same pair of subtrees. What is the value of $w.lp$ for the tree that includes v's $(i+1)$-st subtree?



Please think about the answer to this question and what is the resulting recursive solution before going on to the next page.

The longest path in this tree could arise in one of three ways:

- It could be the current longest path.
- it could be built by combining the longest path descending from v in its first i subtrees, the edge (v, w), and the longest path descending from w. This has length $v.\mathrm{lpd} + 1 + w.\mathrm{lpd}$.
- It could be the longest path in the subtree rooted at w; this has length $w.\mathrm{lpd}$.

This yields the following program.

```
LP($v$)
    $v.lp \leftarrow 0$; $v.lpd \leftarrow 0$;
    **for** each child $w$ of $v$ **do**
        LP($w$);
        $v.lp \leftarrow \max\{v.lp, v.lpd + 1 + w.lpd, w.lp\}$;
        $v.lpd \leftarrow \max\{v.lpd, 1 + w.lpd\}$
    **end** (* for *)
**end** (* LP *)
```

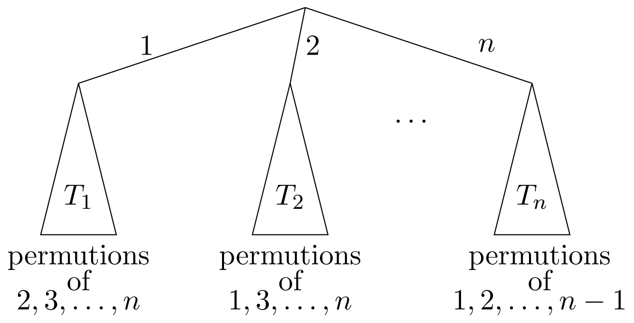
Note that we updated $v.\mathrm{lp}$ before $v.\mathrm{lpd}$; why?

Problem 3 Print all permutations of the numbers $1, 2, \ldots, n$.

The solution to this problem uses an implicit tree to help organize the collection of permutations. We say implicit to mean that there is no actual tree. It is just present in our imagination.

The idea is that each root to leaf path will correspond to a distinct permutation. Each edge will be labeled with a number, and the sequence of labels on a path will yield the permutation associated with that path.

More specifically, the root will have n subtrees. The n edges to these subtrees will have the labels $1, 2, \ldots, n$, and each label will be distinct. Let e_i be the edge labeled by i, and let T_i be the subtree reached by following e_i. Then T_i will correspond to the collection of all $(n-1)!$ permutations of $1, 2, \ldots, i-1, i+1, \ldots n$, i.e. the original set of numbers with i removed.


1
2
n
. . .
T_1
T_2
T_n
permutions of $2, 3, \ldots, n$
permutions of $1, 3, \ldots, n$
permutions of $1, 2, \ldots, n-1$

Next, we give a relatively inefficient procedure implementing this process. It takes 3 inputs:

- $k \le n$ a positive integer.
- S_{n-k}, a subset of $n-k$ integers from $\{1, 2, \ldots, n\}$.
- *crtperm*, the current permutation of the k numbers in $\{1, 2, \ldots, n\}/S_{n-k}$.

It outputs all $(n-k)!$ permutations of the integers $1 \ldots n$ that begin with *ctrperm*.

We need one piece of notation: $\circ$ denotes concatenation.

```
Perm($crtperm, S_{n-k}, k$)
    **if** $k = n$ **then** Print($crtperm$)
    **else**
        **for** each $i \in S_{n-k}$ **do**
            $newcrtperm \leftarrow crtperm \circ i$;
            $S_{n-(k+1)} \leftarrow S_{n-k}/\{i\}$;
            Perm($newcrtperm, S_{n-k-1}, k+1$);
        **end** (* for loop *)
**end** (* Perm *)
```

To reduce the runtime by a factor of $\Theta(n)$, we would like to avoid copying over the array S_{n-k} each time we make a recursive call. To this end, we use an array CurPerm$[1:n]$. If k edges on a root to leaf path have been traversed, CurPerm$[1:k]$ will store the labels for these k edges, in the order in which they were traversed. The remaining $n-k$ numbers are stored in CurPerm$[k+1:n]$; their order is not important. Initially, CurPerm$[i] = i$ for $1 \le i \le n$.

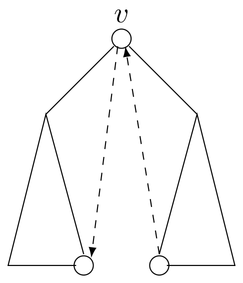
How might we update CurPerm as the tree is traversed so that we can output a new permutation each time a leaf is reached? We want to make just a few changes to CurPerm for each recursive call. Please think about this before going to the next page and see if you can complete the above recursive solution.

We need to place the edge being traversed in CurPerm[k+1]. Suppose this edge is in CurPerm[ℓ]. Then we simply swap these two entries, and proceed with the recursive call corresponding to this edge. After the recursive call, we perform the same swap so as to restore CurPerm to its previous state. This yields the following program.

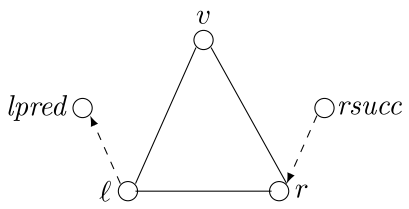
```
Perm($n, k$, CurPerm) (* for each of the $(n-k)!$ permutations $\pi$ of CurPerm[$k+1:n$], prints
                                        CurPerm[$1:k$] followed by $\pi$ *)
    **if** $k = n$ **then** Print(CurPerm[$1:n$])
    **else**
        **for** $\ell = k+1$ **to** $n$ **do**
            Swap(CurPerm[$k+1$], CurPerm[$\ell$]);
            Perm($n, k+1$, CurPerm);
            Swap(CurPerm[$k+1$], CurPerm[$\ell$]);
        **end** (* for loop *)
**end** (* Perm *)
```

The key observation is that each iteration of the loop leaves CurPerm unchanged. We can therefore deduce that each item in CurPerm[$k+1:n$] at the start of the loop will occupy location CurPerm[$k+1$] for one iteration of the loop, and consequently the algorithm considers all $n-k$ possible extensions of the current portion of the permutation, thereby creating $n-k$ distinct extensions of CurPerm[$1:k$] in CurPerm[$1:k+1$], one for each recursive call. We conclude that at each leaf (i.e. the recursive calls for which $k=n$) there is a distinct permutation of $1,\dots,n$ stored in CurPerm[$1:n$], and consequently this procedure prints all $n!$ permutations of $1,\dots,n$.

Problem 4 Add predecessor links to the nodes of a binary tree T. Specifically, for each node v, store its predecessor in $v.\text{pred}$. The leftmost node will have a predecessor of nil.



Let's consider a recursive solution. There are two pointers touching v's subtree, T_v that involve nodes outside T_v, namely the predecessor pointer for the leftmost node in T_v, and the pointer to the rightmost node in T_v; the latter pointer comes from this node's successor. Note that the leftmost node ℓ in T_v is characterized as the first node on the path descending from v to the left which does not have a left child. Let *lpred* be ℓ's predecessor, or **nil** if ℓ does not have a predecessor. The rightmost node r and its successor *rsucc* can be specified analogously.



There is one more issue with recursion on binary trees, namely the order in which the recursive calls to the left and right subtrees are made. As we can see from the algorithms for the previous problems, this does not really matter on trees where the degree can be arbitrarily large. For the moment we will choose to make the recursive call on the left subtree first. We can always revise this if it does not work well.

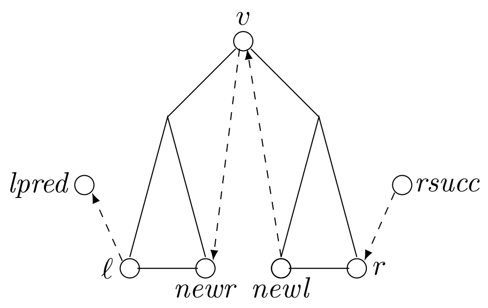
Now, let's consider the pair of nodes *lpred* and ℓ; which one is reached first in the recursion? It's node *lpred* (for it is in a left subtree of some proper ancestor of v, a node on the path from the root of the whole tree to v). Therefore, by the time the recursive call on v is made, the algorithm knows the value of *lpred*. So it makes sense to have *lpred* (strictly, a pointer to *lpred*) be one of the arguments for the recursive call on v.

Next, let's consider nodes r and *rsucc*. In this case, r is reached before *rsucc*. This suggests the recursive call on v should return a pointer to r to be used by *rsucc* when a recursive call on *rsucc* is made.

Consequently the recursive call to v, $\text{Pred}(v, \mathit{lpred}, r)$, needs three arguments.

- v, a pointer to the root of the tree,
- *lpred*, a pointer to the node that is the predecessor of the leftmost node in the current tree,
- and a return value, r, which at the end of the recursive call will store a pointer to the rightmost node in the subtree rooted at v.

Before obtaining the recursive solution let's also look at v's two subtrees.



What should the arguments be for the recursive call to v's left subtree and to v's right subtree? What is the update, if any, when there is no left subtree and no right subtree? Please think about your answer before going on to the next page.

```
Pred($v$, $lpred$, $r$)
    if $v.left$ = nil then $v.pred \leftarrow lpred$
        else do
            Pred($v.left$, $lpred$, $newr$);
            $v.pred \leftarrow newr$;
            end; (* else do *)
    if $v.right$ = nil then $r \leftarrow v$
        else  Pred($v.right$, $v$, $r$);
end (* Pred *)
```

The initial call, for a tree T, will be to Pred(T, **nil**, r).

Question: How should the procedure change if the first recursive call is always to the right subtree and not to the left subtree?