PREORDER TREE TRAVERSAL

Let us consider **the problem of numbering the vertices of a rooted tree in preorder (depth first search order).**

At first glance this problem looks sequential!

RECURSIVE PREORDER TRAVERSAL

```
PREORDER.TRAVERSAL(nodeptr):
Begin
   if nodeptr≠null then
      nodecount ← nodecount + 1
      nodeptr.label ← nodecount
      PREORDER.TRAVERSAL(nodeptr.left)
      PREORDER.TRAVERSAL(nodeptr.right)
   endif
End
```

Where is the parallelism? The fundamental operation assigns a label to a node.

We cannot assign labels to the vertices in the right subtree of the left subtree, until we know how many vertices are on the left subtree of the left subtree, and so on.

The algorithm seems inherently sequential!

Can we parallelize this?

PARALLELIZATION OF THE TRAVERSAL

Instead of focusing on the vertices, let us look into the edges.

When we perform a preorder traversal, we systematically work our way through the edges of the tree.

- We pass along every vertex twice: one heading down from the parent to the child, and one going from the child to the parent.

- ***If we divide each tree edge into two edges, one corresponding to the downward traversal, and one corresponding to the upward traversal, then the problem of traversing a tree turns into the problem of traversing a single linked list.***

4 steps:

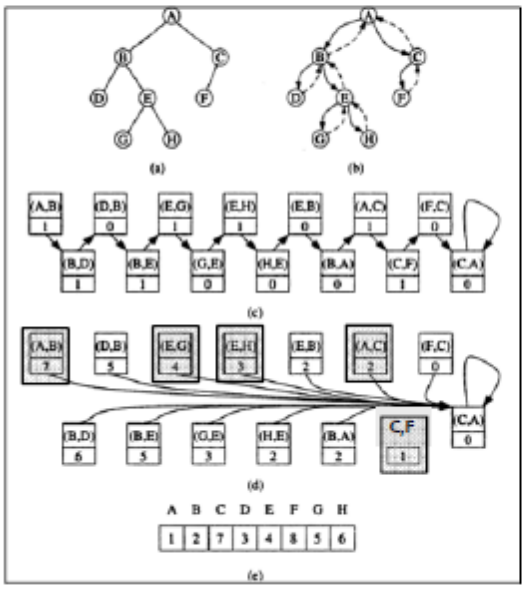
1. The algorithm constructs a singly linked list. Each vertex of the linked list corresponds to a downward or upward edge traversal.

2. Algorithm assigns weights to the vertices of the newly created single linked list.
   - For vertices corresponding to downward edges, the weight is 1 (it contributes to node count).
   - For vertices corresponding to upward edges, the weight is 0 (it does not contribute to node count).

3. For each element of the singly-linked list, the rank of each element is determined (by pointer jumping).

4. The processors associated with the downward edges use the ranks they have computed to assign a preorder traversal number to their associated tree nodes (the tree node at the end of the downward edge).

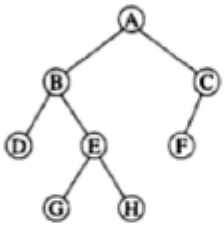
EXAMPLE



a) Tree
b) Double Tree Edges, distinguishing downward edges from upward edges.
c) Build linked list out of directed tree edges. Associate 1 with downward edges, and 0 with upward edges.
d) Use pointer jumping to compute total weight from each vertex to end of list.

The elements of the linked list which correspond to downward edges, have been shaded.
Processors managing these elements assign preorder values.
For example, (E,G) has a weight 4, meaning tree node G is 4th node from end of preorder traversal list.
The tree has 8 nodes, so it can compute that tree node G has label 5 in preorder traversal (=8-4+1)

DATA STRUCTURE FOR THE TREE



	A	B	C	D	E	F	G	H
parent	null	A	A	B	B	C	E	E
sibling	null	C	null	E	null	null	H	null
child	B	D	F	null	G	null	null	null

For every tree node, the data structure stores the node's parent, the node's immediate sibling to the right, and the node's leftmost child.

Representing the node this way keeps the amount of data stored a constant for each tree node and simplifies the tree traversal.

PROCESSOR ALLOCATION

The PRAM algorithm spawns 2(n-1) processors.

A tree with nodes have (n-1) edges.

We are dividing each edge into two edges, one for the downward traversal and one for the upward traversal.

So, the algorithm needs 2(n-1) processors to manipulate each of the 2(n-1) edges of the singly-linked list of elements corresponding to the edge traversals.

CONSTRUCTION OF THE LINKED LIST

Once all the processors have been activated they construct the linked list:

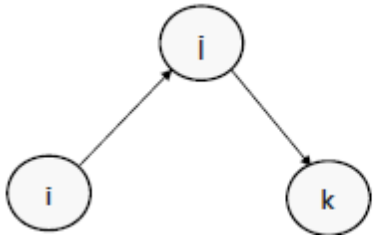
- P(i,j): The processor for the edge (i,j)
- Note (j,i) has a different processor P(j,i)

Given an edge (i,j), P(i,j) must compute the successor of (i,j) and store in a global array: succ[1...2(n-1)].

- If the successor of (i,j) is (j,k), then succ[(i,j)]←(j,k)

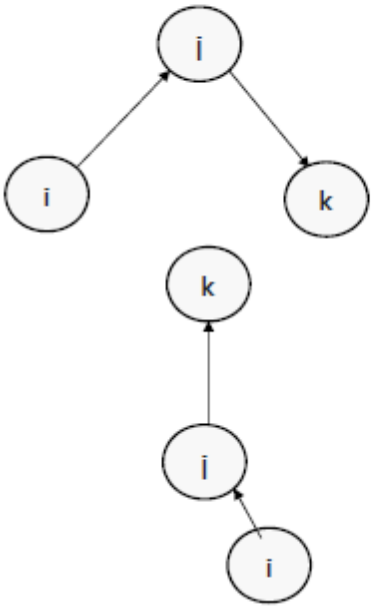
HANDLING UPWARD EDGES

Edge (i,j), such that parent(i)=j



If sibling[i]≠NULL
succ[(i,j)]←(j,sibling[i])

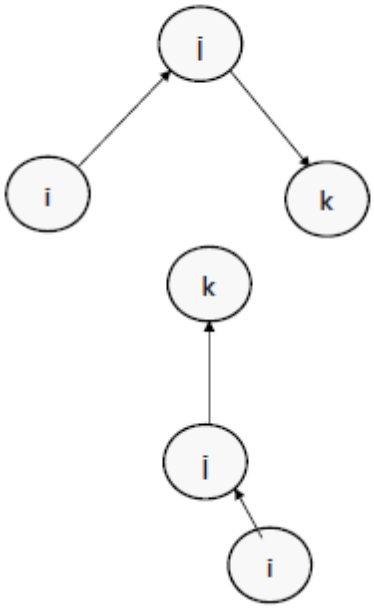
Edge (i,j), such that parent(i)=j



If sibling[i]≠NULL
succ[(i,j)]←(j,sibling[i])

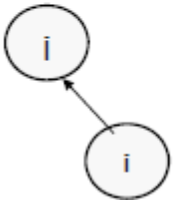
Else If parent[i]≠NULL
succ[(i,j)]←(j,parent[i])

Edge (i,j), such that parent(i)=j



If sibling[i]≠NULL
succ[(i,j)]←(j,sibling[i])

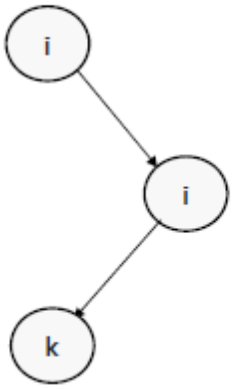
Else If parent[i]≠NULL
succ[(i,j)]←(j,parent[i])



Else
succ[(i,j)]←(i,j)
The edge is at the end of the tree traversal, so we put a loop at the end of the element list.

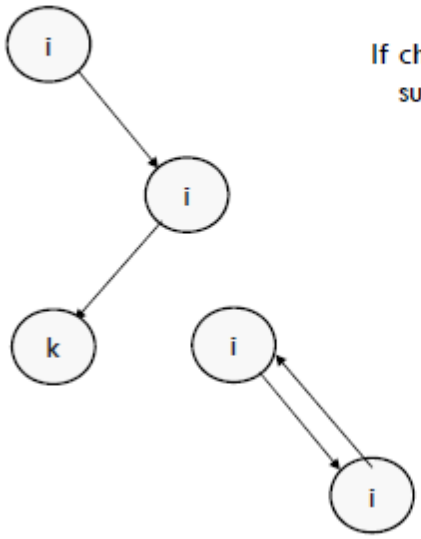
HANDLING DOWNWARD EDGES

Edge (i,j), such that parent[i]≠j.



If child[j]≠NULL
succ[(i,j)]←(j,child[j])

Edge (i,j), such that parent[i]≠j.



If child[j]≠NULL
succ[(i,j)]←(j,child[j])

else
succ[(i,j)]←(j,i)

ie. j is a leaf and the successor is the edge back from the child to the parent.

ASSIGNING EDGE RANKS

After the processors construct the list, they assign position values:

- 1 to those elements corresponding to downward edges
- 0 to those elements corresponding to upward edges.
- Note the root is already handled.

if parent[i]=j, position[(i,j)]←0
Else position[(i,j)]←1