Pointer Jumping

The technique

- Simple operation. Each node i replaces its pointer P[i] with the pointer of the node that it points to, P[P[i]].
- Can be applied to both linked lists and trees.
- By repeating this operation, it is possible to compute, for each node in a list or tree, a pointer to the end of the list or root of the tree.
- We assume that the root points to itself or points to null.

Basic algorithm

Given a sequence P of pointers that represent a tree (i.e., pointers from children to parents), the following code will generate a pointer from each node to the root.

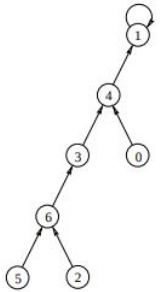
ALGORITHM: POINT_TO_ROOT(P)

```
1  for j from 1 to ⌈log |P|⌉
2      P := {P[P[i]] : i ∈ [0..|P|)}
```

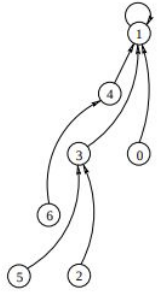
The idea behind this algorithm is that in each loop iteration the distance spanned by each pointer, with respect to the original tree, will double, until it points to the root.

Since a tree constructed from n = |P| pointers has depth at most n − 1, after ⌈log n⌉ iterations each pointer will point to the root.

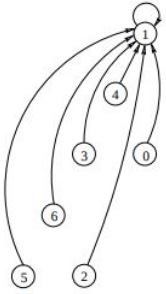
Example run



(a) The input tree $P = [4, 1, 6, 4, 1, 6, 3]$.



(b) The tree $P = [1, 1, 3, 1, 1, 3, 4]$ after one iteration of the algorithm.



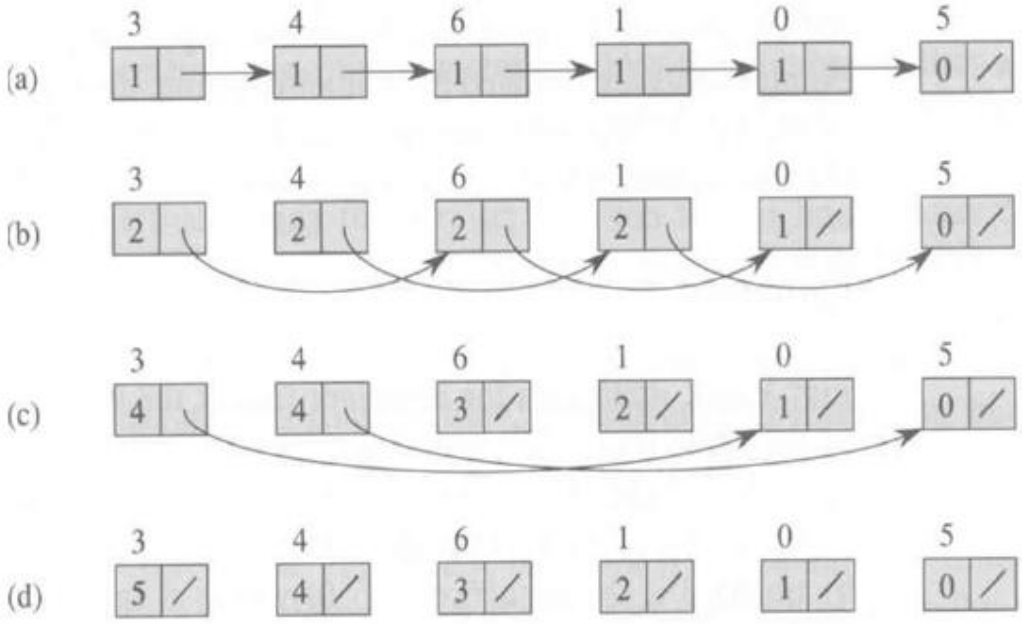
(c) The final tree $P = [1, 1, 1, 1, 1, 1, 1]$.

Application to problem 1: list ranking

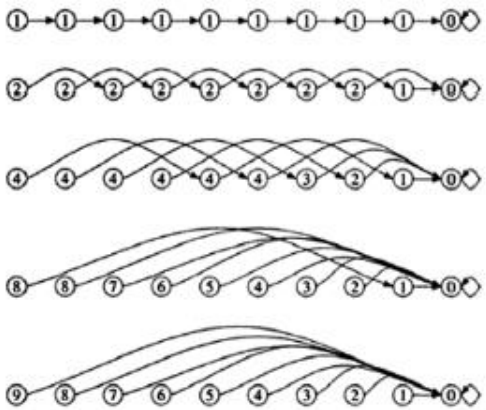
Computing distance from each node to the end of a linked list is called list ranking.

There are simple sequential algorithms that perform the same task using O(n) work. For example, by making two passes through the list. The goal of the first pass is simply to count the number of elements in the list. The elements can then be numbered with their positions from the end of the list in a second pass.

LIST RANKING — EXAMPLE 1



LIST RANKING – EXAMPLE 2



The position of each item on the n-element list can be determined in $\lceil \log n \rceil$ pointer jumping steps.

THE PRAM ALGORITHM

PRAM algorithm to compute, for each element of a singly-linked list, its distance from the end of the list.

```
LIST.RANKING (CREW PRAM):
  Initial condition: Values in array next represent a linked list
  Final condition: Values in array position contain original distance
                   of each element from end of list
  Global variables: n, position[0...(n - 1)],next[0...(n - 1)], j
  begin
    spawn (P0, P1, P2, ..., Pn-1)
    for all Pi where 0 ≤ i ≤ n - 1 do
      if next[i] = i then position[i] ← 0
      else position[i] ← 1
      endif
      for j ← 1 to ⌈log n⌉ do
          position[i] ← position[i] + position[next[i]]
          next[i] ← next[next[i]]
      endfor
    endfor
  end
```

Note this step does not depend on j.

There are $\lceil \log n \rceil$ steps.

There are n processors.

So total cost is:

$$\Theta(n \log n)$$

Not cost optimal!

THE SAME CODE USING POINTER NOTATIONS

```
List_ranking(L)
1.   for all P_i for each node i, do
2.       if i->next = null then i.d = 0
3.       else i.d = 1
4.       while(i->next != null) do
5.             i.d = i.d + i->next.d
6.             i->next = i->next->next
```

Synchronization is important

- In step 6 (i->next = i->next->next), all processors must read right hand side before any processor write left hand side

The list ranking algorithm is EREW

- If we assume in step 5 (i.d = i.d + i.next.d) all processors read i.d and then read i.next.d
- If j.next = i, i and j do not read i.d concurrently

Application to problem 2: preorder tree traversal

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

```
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.***

TARJAN AND VISHKIN (1984)

4 steps:

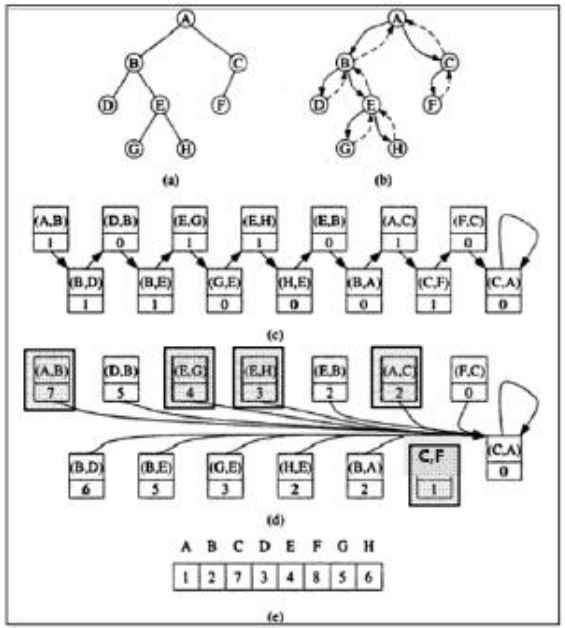
1. The algorithm constructs a singly linked list. Each vertex of the linked list corresponds to a downward or upward edge traversal.

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.

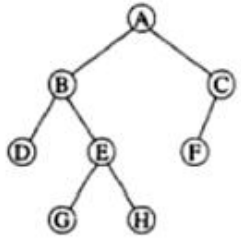
The elements of the linked list which correspond to downward edges, have been shaded.

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:

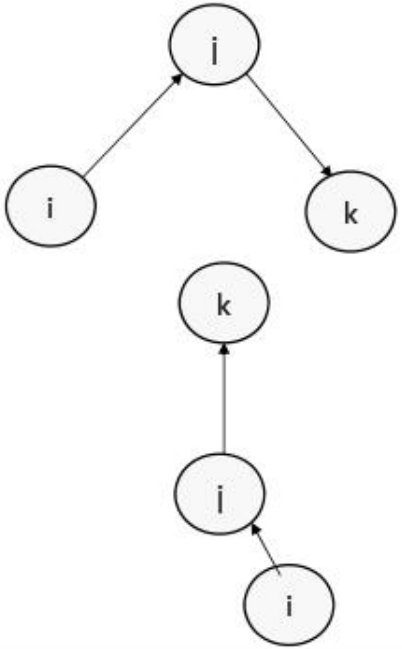
- P(i,j): The processor for the edge (i,j)
- Note (j,i) has a different processor P(j,i)

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



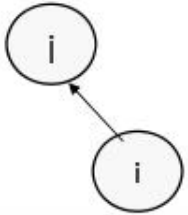
If sibling[i]≠NULL
succ[(i,j)]←(j,sibling[i])

j

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

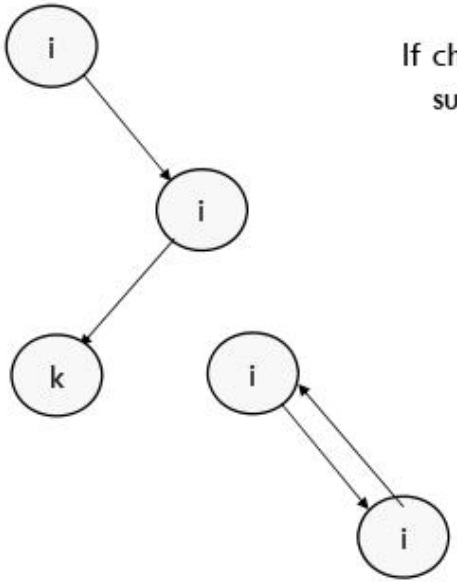
j

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])

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

POINTER JUMPING: SUFFIX SUM

The pointer jumping follows subsequently to compute the suffix sum.

The final position values indicate the number of preorder traversal nodes between the list element and the end of the list.

To compute each node's preorder traversal label compute (n-position+1).

PRAM PROGRAM

```
PREORDER.TREE.TRAVERSAL (CREW PRAM):
 Global  n                          {Number of vertices in tree}
         parent[1 ... n]            {Vertex number of parent node}
         child[1 ... n]             {Vertex number of first child}
         sibling[1 ... n]           {Vertex number of sibling}
         succ[1 ... (n - 1)]        {Index of successor edge}
         position[1 ... (n - 1)]    {Edge rank}
         preorder[1 ... n]          {Preorder traversal number}
 begin
  spawn (set of all P(i, j) where (i, j) is an edge)
  for all P(i, j) where (i, j) is an edge do
   {Put the edges into a linked list}
   if parent[i] = j then
     if sibling[i] ≠ null then
       succ[(i, j)] ← (j, sibling[i])
     else if parent[j] ≠ null then
       succ[(i, j)] ← (j, parent[j])
     else
       succ[(i, j)] ← (i, j)
       preorder[j] ← 1 {j is root of tree }
     endif
   else
     if child[j] ≠ null then succ[(i, j)] ← (j, child[j])
     else succ[(i, j)] ← (j, i)
     endif
   endif
```

PRAM ALGORITHM (CONTD.)

```
    if parent[i] = j then position[(i, j)] ← 0
    else position[(i, j)] ← 1
    endif
    {Perform suffix sum on successor list}
    for k ← 1 to ⌈log(2(n − 1))⌉ do
      position[(i, j)] ← position[(i, j)] + position[succ[(i, j)]]
      succ[(i, j)] ← succ[succ[(i, j)]]
    endfor
    {Assign preorder values}
    if i = parent[j] then preorder[j] ← n + 1 − position[(i, j)]
    endif
  endfor
end
```

Time Complexity:O($\lceil \log(n) \rceil$

Processors: O(n)

Cost: O(nlogn)