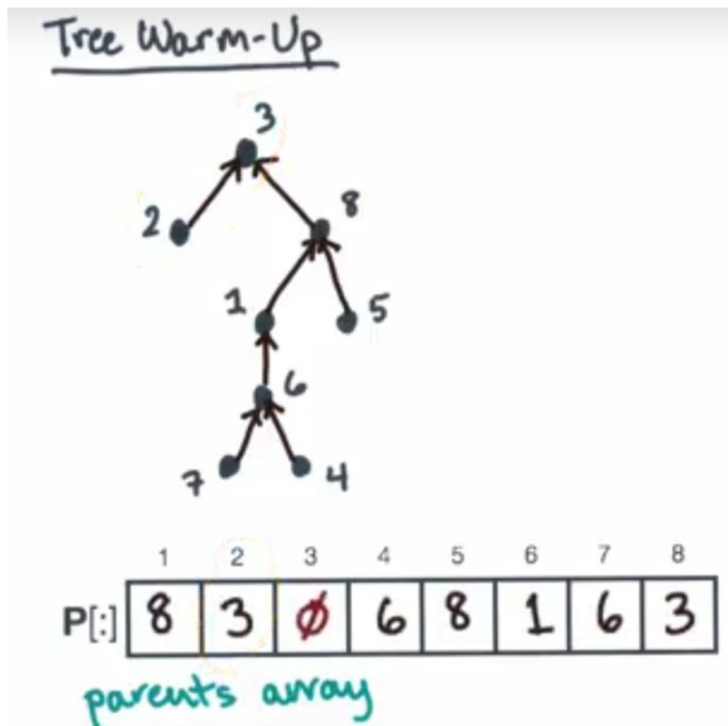
Lesson 1-5 Tree Computations

Tree Warm Up

A tree can be stored in an array.
Step 1: number the nodes
Step 2: store the parent of each node in the array. Undirected edges are converted to directed edges.
Step 3: The array is called P and has one entry per node. For example, Node 2 points to parent 3. So a ‘3’ is placed in array location ‘2’. Fill in the array. In the tree example below you should get the following array:



To find the root of the tree:

- Pick any node
- Follow the parent pointers until you reach the root. The root is a node with no parent.

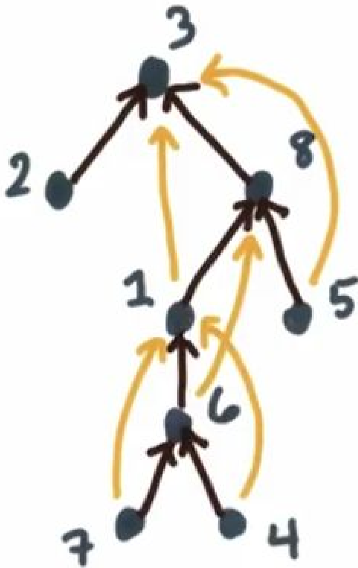
```
root ( P[1:n])
  if n < 1 then return ∅
  node ← (any node, 1..n)
  while P[node] ≠ ∅ do
     node ← P[node]
  return node
```

Running time = O(n)

The running time is linear in the number of nodes because in the worst case, the tree is a single branch. You might then have to traverse the entire tree to get to the root.

How to do this in parallel?

Explore from all nodes simultaneously. At each node, change parent to grandparent. If a node has no grandparent, it must be pointing to the root. For example, nodes 2 and 8 do not have grandparents.



Nodes 1,2,5,8 all point to the root. Continue with the rest of the nodes. Eventually all nodes will point to the root.

This pseudo code determines if a node has a grandparent:

```
hasGrandparent (k, P[1:n])
    return (k > Ø) and (P[k] > 0)
                   and (P[P[k]] > 0)
```

This pseudo code records either the grandparent (if it has one) or the parent's id.

```
adopt ( P[1:n], G[1:n])
    par-for i ← 1 to n do
        if hasGrandparent (i, P[:])
        then G[i] ← P[P[i]]
        else G[i] ← P[i]
```

The findRoots algorithm:
This algorithm double buffers the parents array, the outer loop is sequential and taken over all the possible levels, the max. num of which is log(n).

findRoots uses: pointer-jumping, has polylogarithmic span, and works on a forest, not just a tree. The algorithm would make every node point to the root of its own tree. It is not work optimal.

Work-Optimal List Scan/Prefix-Sum Part 1

Given a link list, compute its rank.
Do this with using scan or prefix-sum n parallel using pointer-jumping, known as Wyllie's algo.

[Wyllie's Algorithm]

The cost of the scheme is not work optimal:
W(n) = O(n log(n))
D(n) = O(log(n))

There is at least one trick to make Wyllie's more work optimal.
1. Shrink the list to size m < n.
2. Run Wyllie on the smaller list. O(m log(m))
3. Restore full list and ranks

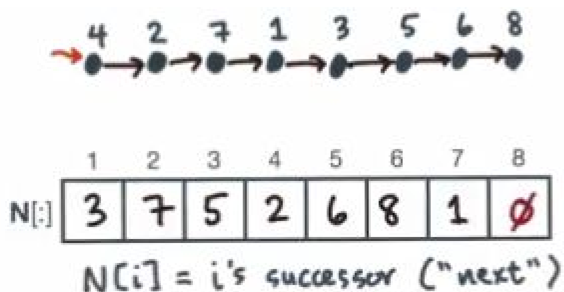
The question now is …. how should m be chosen to lead to work-optimality? n/log(n)

Parallel Independent Sets, Part 1: A Randomized Algorithm

To shrink a list in parallel, use an independent set.

To do this… N[i] = i's successor ("next")

Given this list, here is the successor array.



An independent set is: A set I of vertices such that i $\in I \Rightarrow N[i] \notin I$

(i is an element of the set does not have its successor in the set)

An independent set of the above is I = {3,7,8}

This set is **not** an independent set {3,4,6,8} because 8 is a success of 6.

To generate an independent set sequentially, traverse the list and put every other node in the list. For example the independent set doing this algorithm would be: I = {4,7,3,6} (follow the linked list)

Doing the same thing in parallel is trickier.
Problem of symmetry: when looking at an individual node, all nodes look alike.

A scheme is needed to break the symmetry.

- For each node flip a coin. Heads Pr[heads] = Pr[tails] = ½
- Heads will be included in the independent set and tails will be left out.
- There is a possibility that a node and its neighbor are both heads.
- In this case change the head into a tail, so any head is adjacent to a tail

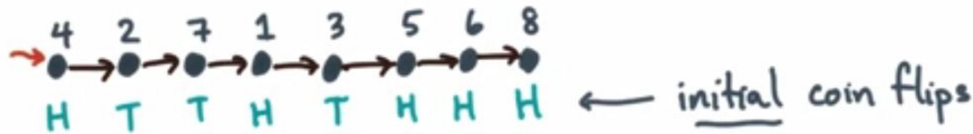
Creating the parallel independent set:

```
ParIndSet (N[1:n], I[:])
    let C[1:n], Ĉ[1:n] ≡ space for coins
    par for i ← 1 to n do
        C[i] ← flip coin (H or T)
        Ĉ[i] ← C[i]    // make a copy
    par-for i ← 1 to n do
        if (Ĉ[i] = H) and (N[i] > ∅) and (Ĉ[N[i]] = H) then
            C[i] ← T
    I[:] ← gather If(1:n, C[1:n])        "symmetry breaking"
```

Parallel Independent Set

Which of these nodes will end up in the independent set. Remember the test for heads must be run simultaneously. So nodes 5 and 6 will both see that they are part of a double head pair. Both will change to tails according to the algorithm.

Parallel Independent Sets, Part 2: Quiz!



	1	2	3	4	5	6	7	8
N[:]	3	7	5	2	6	8	1	Ø

N[i] = i's successor ("next")

Your task: List the nodes of the independent set after the randomized ParIndSet completes.

1 4 8

The Work and Span of the Parallel Independent Sets

```
ParIndSet (N[1:n], I[:])
    let C[1:n], Ĉ[1:n] ≡ space for coins
    par for i ← 1 to n do
        C[i] ← flip coin (H or T)
        Ĉ[i] ← C[i]   // make a copy
    par-for i ← 1 to n do
        if (Ĉ[i] = H) and (N[i] > Ø) and (Ĉ[N[i]] = H) then
            C[i] ← T
    I[:] ← gather If(1:n, C[1:n])
```

Q: What are the work & span of this algorithm?

$W(n) = O($ n $)$

$D(n) = O($ log n or 1 $)$

What is the average number of vertices that end up in the independent set? ¼ n
From the coin flip, there are four possibilities (hh,ht,th,tt) after the initial flip. Then a correct is applied for the case of double heads. So the possibilities are now (th,ht,th,tt). So now there is one possibility the first node of the pair is a heads.

Work-Optimal List Scan/Prefix-sum, Part 2: Details
Recall the trick for a work optimal list ranking algorithm.

1. Shrink list to size m
2. Run Wyllie
3. Restore full list and ranks.

How to Shrink the List
Use the independent set, jumping over the independent set elements.

1. begin with the list of ranks:

	1	2	3	4	5	6	7	8
N[:]	3	7	5	2	6	8	1	Ø
R[:]								

2. Assign a 0 to the head and a 1 to all the other nodes. (4 is the head of the linked list)

	1	2	3	4	5	6	7	8
N[:]	3	7	5	2	6	8	1	Ø
R[:]	1	1	1	0	1	1	1	1

3. Now find the independent set.
   Flip the coin for each node, and remove the double heads.

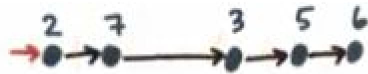
	1	2	3	4	5	6	7	8
N[:]	3	7	5	2	6	8	1	Ø
R[:]	1	1	1	0	1	1	1	1

4. The independent is {4,1,8}

5. Remove the independent set. This means removing the vertices 4,1,8 and rewiring 7 to point to 3.
6. The independent

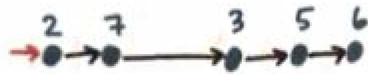
	1	2	3	4	5	6	7	8
N[:]		7	5		6	Ø	3	
R[:]		1	2		1	1	1	

7. We need to get to a list of size n/log(n).
8. Flip the coins and do again.



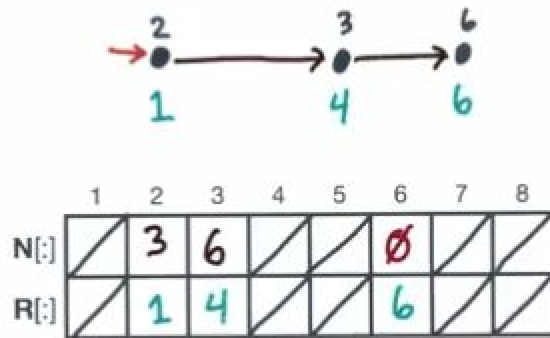
	1	2	3	4	5	6	7	8
N[:]		7	5		6	Ø	3	
R[:]		1	2		1	1	1	

9. In this instance, 5 and 7 are in the independent set. So they need to be removed.

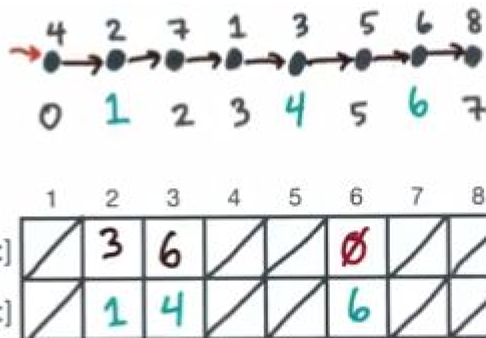


	1	2	3	4	5	6	7	8
N[:]		7	5		6	Ø	3	
R[:]		1	2		1	1	1	

10. Now, suppose the list is now the right size.
11. Run List Scan on it. You get the following result:



12. Compare the found result with the what they should be based on the original list. They are the same.



The downloadables has a detailed discussion on how to do the rest of the list.

Work-Optimal List Scan/Prefix-Sum Part 3

In addition to the bookkeeping, how many times do you need to run the independent set to shrink the list to O(n/log(n)) in length? O(log(log(n)))

You expect to pick n/4 nodes to start, so length after the first Par-IS is ..

E[list length after 1 Par-IS] = 3/4 n

Now run Par-IS 'k' times:

E[list length after k calls] = $(¾)^k n$

The target list length is n/log(n) so ….

E[list length after k calls] = $(¾)^k n < n/log(n)$

k < $\theta(log\ log\ n)$

W_1 = O(n log log (n))

D_1 = O(log(n)

You will also want to know what is the length when not around the average.
You will also want to know -- how much book keeping is necessary to implement steps 1 and 3.

A Seemingly Sequential Tree Algorithm

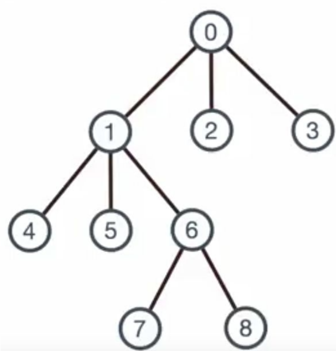
To perform postorder numbering on a tree:

Postorder numbering

holds numbering

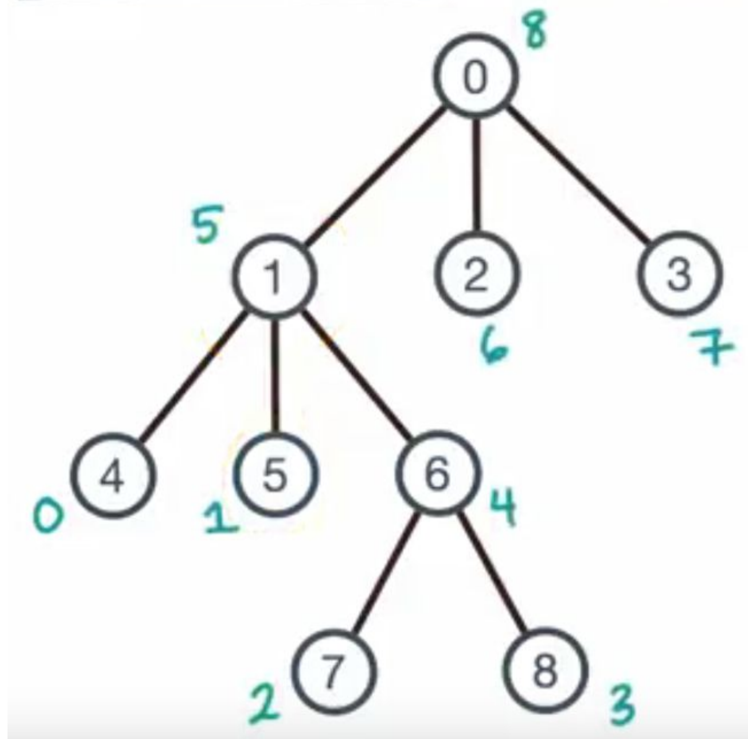
```
postorder (root, V[1:n], v0)
  v ← v0                     initial val
  for each c ∈ children(root) do
     v ← postorder(c, V[:], v) + 1
  V[root] ← v
  return v
```

Use the algorithm on a tree:



Start with the root (in this case #0).
Trace it to leaf number 4. Since node 4 has no children, it gets the initial value of 0.

The next child without a child (node 5) will get the next value (0 + 1). Continuing for the entire tree you should get:



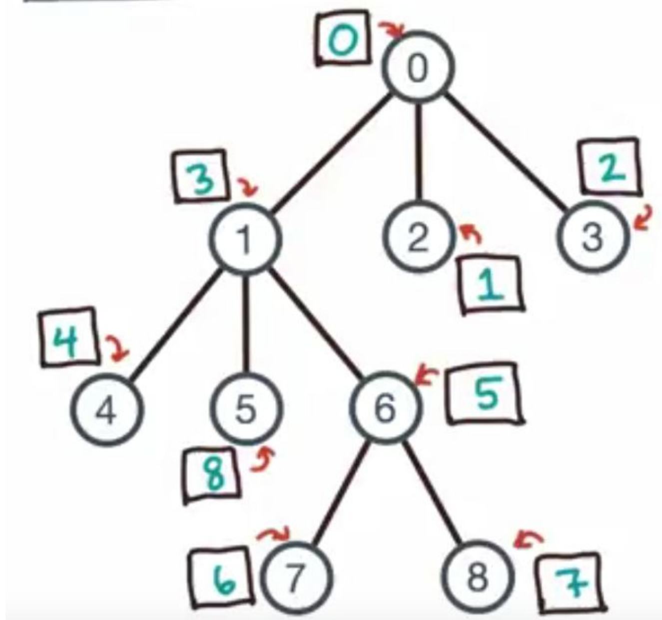
PreOrder Numbering

The tree is preOrdered:

```
preorder(root, V[1:n], v0)
    v ← v0
    V[root] ← v
    foreach c ∈ children(root) do
        v ← preorder(c, V[:], v+1)
    return v
```

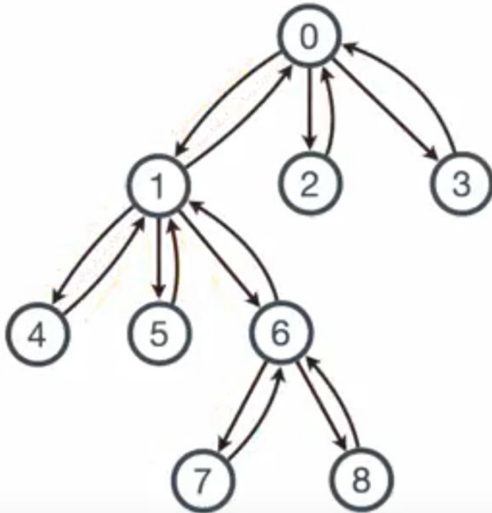
There is a slight ambiguity - it does not say in what order to visit the children.

So this answer is valid also:



Euler Tour Technique

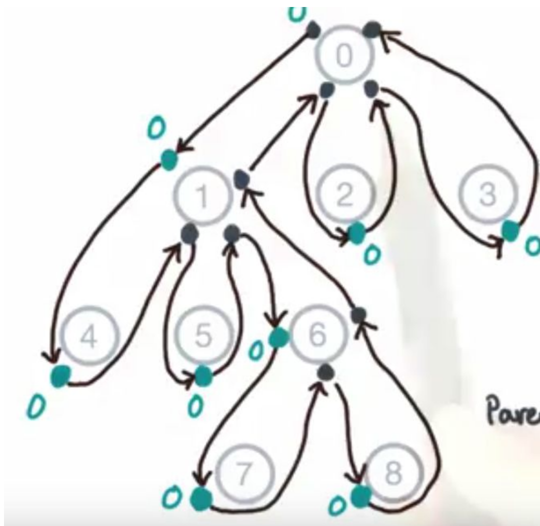
To view a tree as a list:



- take every undirected edge and represent it as a pair of directed edges.
- At every node the number of incoming edges equals the number of outgoing edges.
- This makes this graph Eulerian.
- For every Eulerian directed graph there is a directed circuit.
  Euler circuit: a closed path that uses every edge once.
- This circuit gives you a linked list.

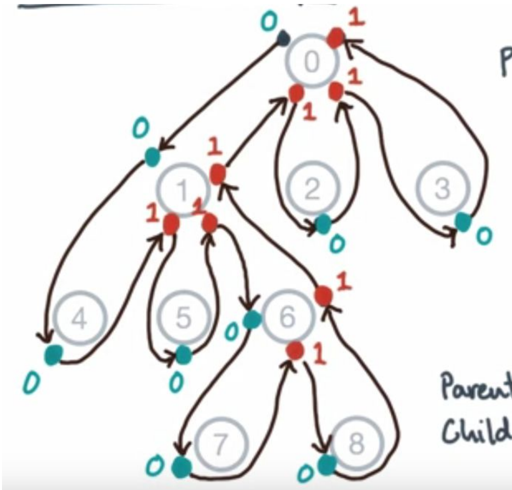
To do postOrder numbering:

- Assign a 0 to the head of the list.
- Then mark all parent-to-child sinks with a 0

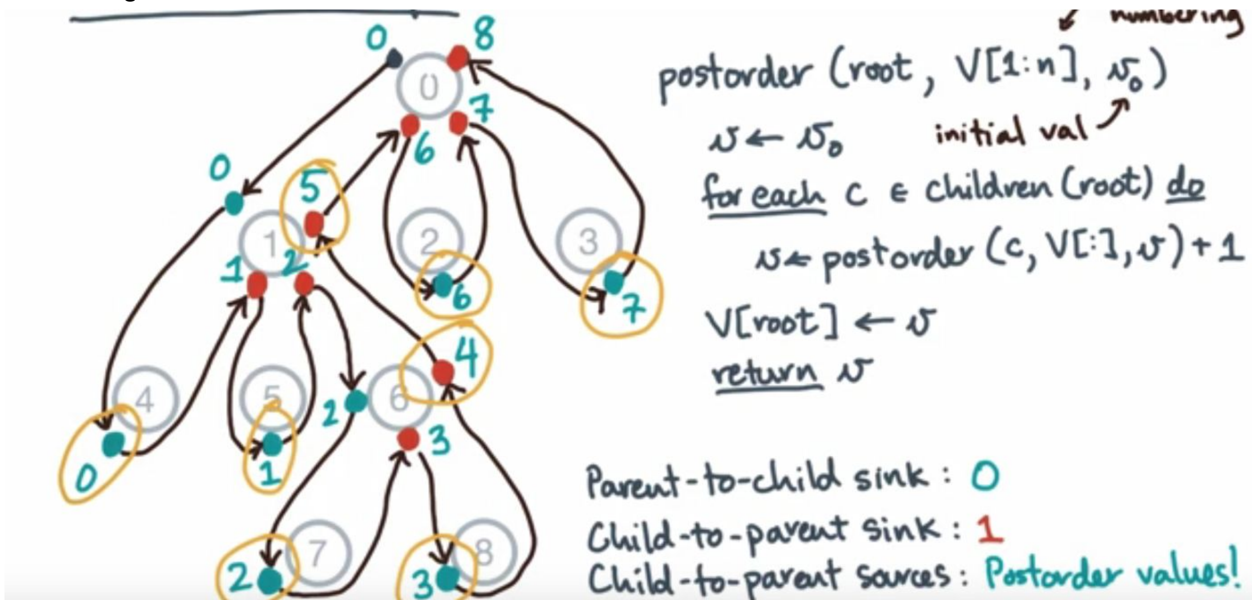


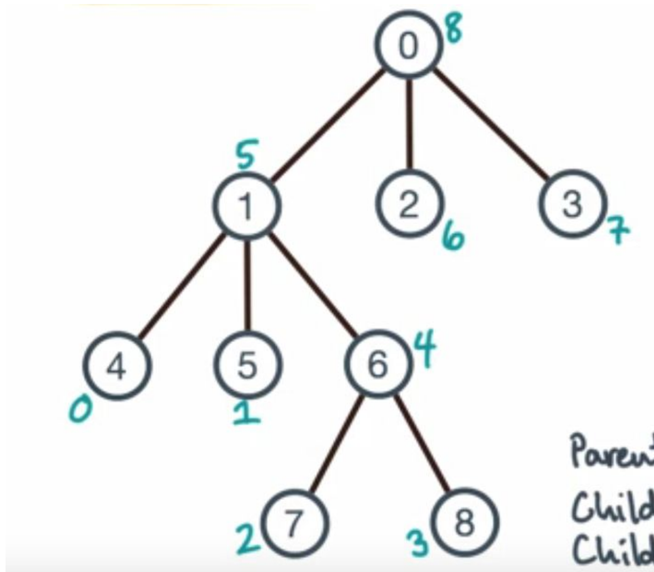
.

- For the remaining nodes, notice they are all sinks that go from children to parents. This corresponds to the return value, so put a ‘1’ on these nodes.



Now when you do a scan of this circuit, starting at the root, the values are the postOrder numberings.





A summary of the Euler Tour Technique

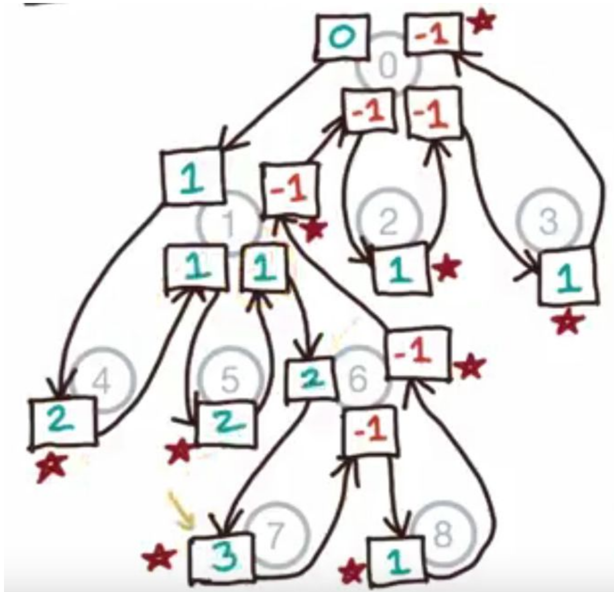
1. Tree → List
2. Parent-to-child sink = 0
3. Child-to-Parent sink = 1
4. Child-to-Parent Source = postOrder values
5. List prefix scan

The span of an Euler Tour

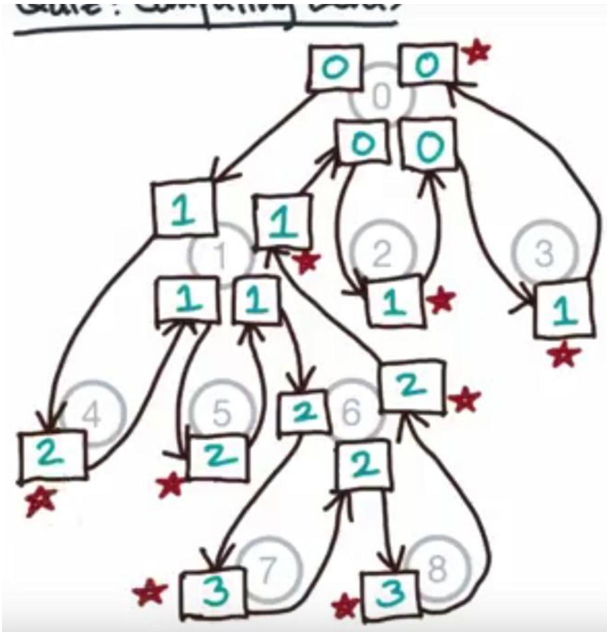
The overall span of the Euler tour is O(log(n))
The whole point of the Euler tour is to turn the tree into a list and then the tree's shape will not matter.
It is not always possible to convert a computation into an equivalent Euler computation.

Computing Levels
The level (depth)of a tree node is the minimum number of edges from the root of the node.

1. Computer an Euler circuit
2. Then consider each sink. Put a +1 for each parent to child path.
3. Then put a -1 for each child to parent path.



4. Then run a listScan



Implementing Euler Tours

How should the tree be stored and how should the tour be computed?

Make each undirected edge a pair of directed edges.

For each node, its adjacency list will be the set of its outgoing neighbors.

The full list of the adjacency lists:

v	u_0	u_1	u_2	u_3
0	1	2	3	
1	4	5	6	0
2	0			
3	0			
4	1			
5	1			
6	7	8	1	
7	6			
8	6			

$adj(v) \equiv \{u_0, u_1, \ldots, u_{d_v} - 1\}$

To complete the Euler tour: define a Successor Function

Given an edge that goes from u_i to v, the function returns the NEXT neighbor in v adjacency list. The mod function makes the list circular.

Successor function

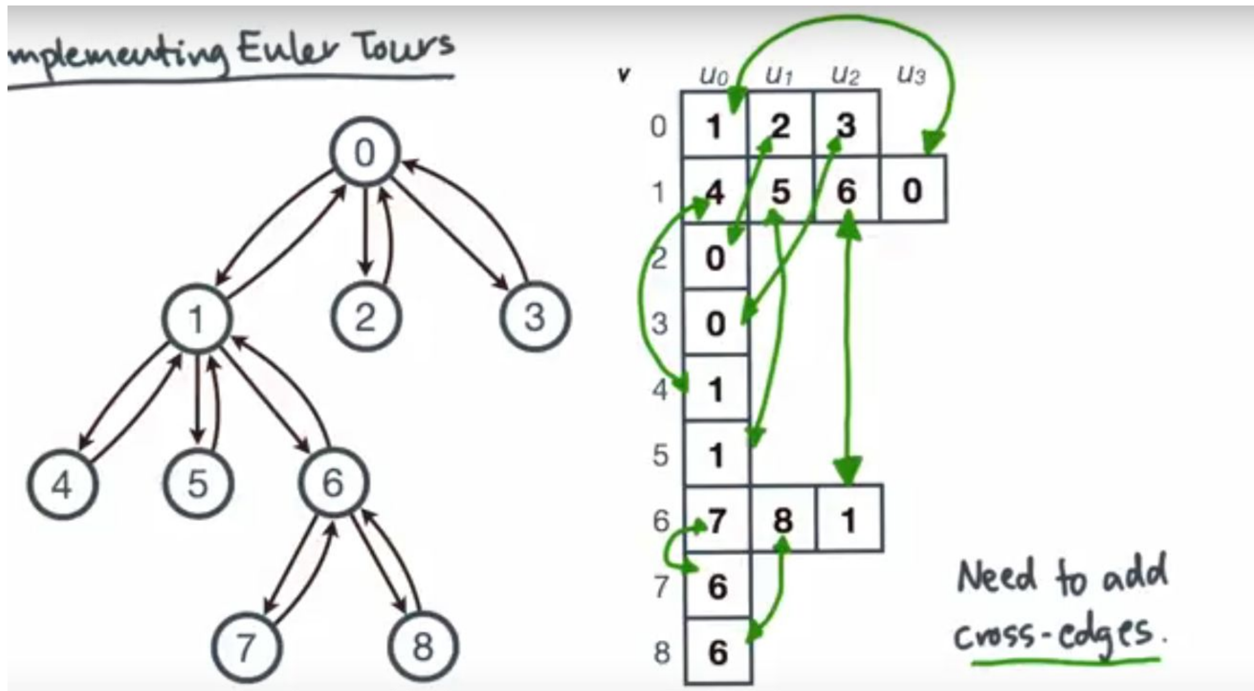
$s(u_i, v)$

$\equiv (v, u_{(i+1) \bmod d_v})$

For example the successor of the edge 0→ 1 is 1 → 4

What is the cost of the successor function?

To be able to quickly get to the successor function … add cross edges.



What is S(S(S(6,8)))?

S(6,8) returns (8,6)

apply the definition again … get (6,1)

apply again … get (1,0)