How do I determine the order of visiting all leaves of a rooted tree, so that in each step I visit a leaf whose path from root contains the most unvisited nodes?

http://forums.d2jsp.org/topic.php?t=70139590&f=120

This question previously had details. They are now in a comment.

1 Answer

John Kurlak, works at Facebook
Updated Nov 7, 2014

Edit: this is a long answer with lots of places to get confused, so please ask questions if you don't understand! (that also means lots of places where I could have messed up, so let me know if you see anything amiss)

When I first looked at this problem, it was pretty clear to me that we would have to do preprocessing of some sort in order to solve it in $O(n)$ time. I considered the following options for preprocessing:

- Retrieving the depth of each node
- Retrieving the list of nodes, ordered by depth
- Storing parent pointers for each node
- Storing the number of edges down to the deepest descendant

(and a few others)

I tried a few approaches for mapping the problem into a different one (like taking the nodes in level order and realizing that each path from the root to a node was a subsequence of level order).

Ultimately, these explorations led to me realize one thing: we could determine the first node in our answer in $\Theta(n)$ time, but it would probably take $O(n)$ time per node to compute each successive node, unless we went about things differently. To get an $O(n)$ algorithm, we would need some way of traversing the tree that would allow us to compute the rank of each node as we went without backtracking.

Each node's rank would have to be a linear function of the number of unvisited

Related Questions

You are given a binary tree in which each node contains a value. Design an algorithm to print all paths which sum to a given value. The path d...

How do I find the greatest sum on a path from root to leaf (that can only touch one node per level—no going up and down) in this tree like str...

What is the root of Java?

Consider a tree where every node holds an integer. What is a function that takes one such tree and returns true if the number in every leaf is...

Is rooting worthwhile?

How do tree-based regression algorithms determine root node and the "cut-point" (threshold/limt) for a given node?

How can I visit Flatiron School?

Given a tree and leaf node, how do you write the code to make the leaf node as the root of the tree?

Who provides the path from leaf sibling to root when testing membership of an element in the set stored as a Merkle tree? How do you derive th...

How do I find the sum of all the vertices of an ordered tree if I consider a path from Node U to Node V?

+ Ask New Question

More Related Questions

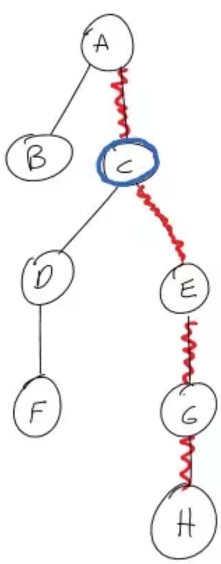
Thus, if we could compute the rank for each node in $\Theta(n)$ time, we could sort the nodes by their rank in $\Theta(n)$ time (we make an array of size $\Theta(n)$, where the key is the rank, and the value is a list of nodes with that rank, ordered by some comparator; then, we iterate over the array in reverse order of rank, outputting the values).

Since I had already investigated various preprocessing steps that I might use, I focused on the second part: iterating over the tree in a meaningful way to compute the ranks as I went.

My initial thought was to start at the deepest leaf and iterate up, possibly deleting nodes as I went. I thought about maybe compacting the tree to represent multiple edges as a single weighted edge (similar to how a radix tree compacts a trie). Ultimately, my investigations from the bottom proved fruitless, and I considered iterating from the top of the tree down, which seemed kind of counter-intuitive to me.

I figured a depth first traversal would be hard to work with, so I focused on traversing the tree level-by-level using a queue.

When we get to a node in a level by level traversal, is there a way to know the rank of that node? Well, we need to know about any edges that are already visited before we get to a node. If we're at a given node, can we easily determine how many edges leading to it will be visited by the time we get to that node? Not really. However, I did come up with one helpful insight: if we get to a node, and that node has an ancestor that has children that are deeper than the node we're at, the edges above that ancestor will be visited before we get to that node. For example, consider:



Suppose we are at node F. Observe that F has an ancestor C, which has children whose depth reaches lower than F. That means when counting the number of edges to F, we can't count the edges from the root A to C.

backtrack up the tree until we get to an ancestor that has deeper children, and then discount the number of edges from the root to that ancestor. What if, however, we could propagate this data down the tree as we iterated through it?

Well, if we're doing a level-by-level traversal of the tree, then we can pass additional information to each node when we add it to our queue of nodes to visit. So what if every time we enqueued a node, x, we passed along the number of unvisited edges above it? Then, by the time we get to each leaf node, we could calculate a rank based on how many unvisited nodes are above it.

The tricky part is determining how many unvisited nodes will be above the next node we're enqueueing. Let's look at the graph above. Suppose we're currently visiting node C and we want to enqueue it's children, D and E. When we enqueue D, we have to pass along that there is 1 unvisited edge above it, and when we enqueue E, we have to pass along that there are 2 unvisited edges above it. The number of unvisited edges from the root node down through a node u at level i to a node, v at level $i + 1$ will be:

$$F(v) = 1 + \begin{cases} F(u), & \text{if v is visited before its siblings} \\ 0, & \text{otherwise} \end{cases}$$

Let's break this down. First, the minimum number of unvisited edges to any node is 1 because there is always one unique edge to every node, so when we get to a node, it will be the first time we visit that edge. Then, if our node is along the path to the deepest node (when compared to paths down through its siblings), then that node will be the first node that goes through the ancestor nodes, so it will inherit the full number of unvisited edges above it. Otherwise, the edges above it will all be visited, so it will inherit nothing.

Since we're iterating through the tree from the top to the bottom, we will always compute $F(u)$ before we need to compute $F(v)$, so we have a dynamic programming solution. This makes sense since trees tend to exhibit optimal substructure.

The base case is that $F(rootnode) = 0$ since there are no unvisited edges above the root node.

The last part in solving this problem is finding a way to determine if a node v is visited before its sibling nodes. A node is visited before its sibling nodes if the subtree below it goes deeper than the subtrees below its siblings. This is where preprocessing will be helpful. In each node, we can store a pointer to the child whose descendants reach the deepest into the tree.

So how exactly do we do that? Well, we can do a depth first traversal of the tree, passing the depth of each node down as we visit children (just add one to the current node's depth). As we recurse back up we compare the maximum depths that each child and its subtree (if any) reached, and we store a pointer to that up, we'll we be able to know which child of a node has a subtree that goes deepest in constant time (note: if two children of a node have the same depth, we consider the child with the lower integer value as the child that reaches deepest; we must do this to satisfy the requirement: "In case of a tie, visit the leaf with the smallest integer").

What this means is that we can compute $F(v)$ in constant time, and $F(v)$ for all v in $\Theta(n)$ time.

As described earlier, we use an array of size $\Theta(n)$ to store the ranks of each leaf node (call this array $ranks$). Any time we compute the rank of a leaf node (call this rank r), we add the leaf node to the list stored at $ranks[r]$.

Next, we need to extract the values in our lists in descending order of rank, and ascending order of value within the same rank. Typically, this would require an $\Omega(n \lg n)$ time sorting algorithm. However, we have some guarantees that allow us to get around this bound.

Namely, we're guaranteed that we don't have any duplicate numbers, and we have a guarantee that the range of values lies in $\Theta(n)$.

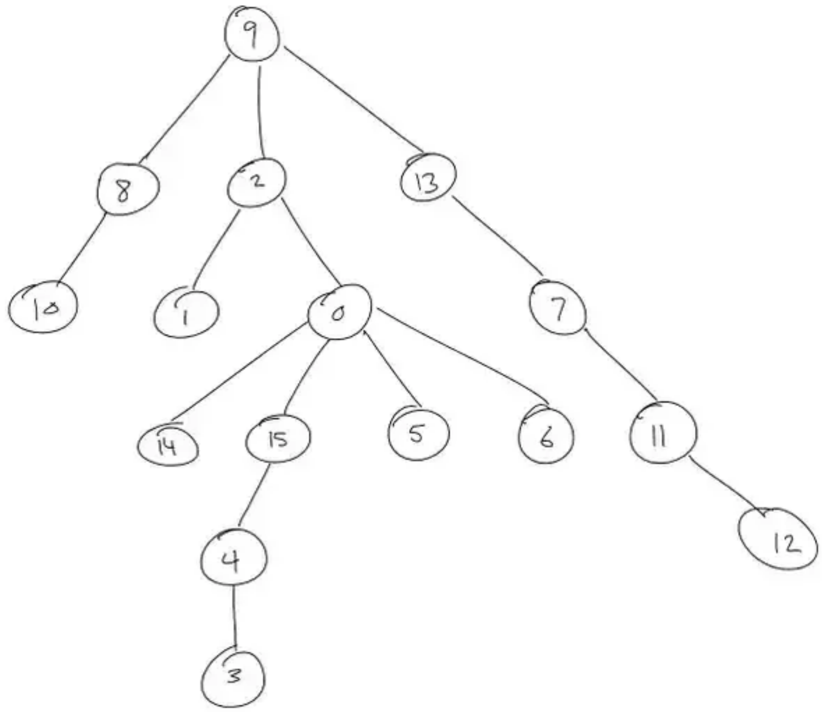
So first, we iterate over our $rank$ array and iterate over each value in the list for each rank. We build an array with value being the key and rank being the value. This takes $\Theta(n)$ time.

While we are iterating, we also build another array, with rank being the key, and starting index in the sorted results being the value (the highest rank has the lowest starting index). (This array is similar to the histogram of key frequencies in counting sort.)

Finally, we iterate over our range of values in ascending order. For each number, we look up its rank using our first array. Then we look up the starting index for that rank using our second array. We place the current number at that index in our answer array, and then we increase the value of the starting index for the current rank.

By the time we've iterated over all values, we have our answer, sorted as required in the question details.

Let's go through an example run, where n=16.



First, we preprocess:

node 9: deepest = (node 8, depth 2) -> (**node 2**, depth 5)
--->**node 8:** deepest = (**node 10**, depth 2)
------->**node 10:** deepest = **undefined**
--->**node 2:** deepest = (node 1, depth 2) -> (**node 0**, depth 5)
------->**node 1:** deepest = **undefined**
------->**node 0:** deepest = (node 14, depth 3) -> (**node 15**, depth 5)
----------->**node 14:** deepest = **undefined**
----------->**node 15:** deepest = (**node 4**, depth 5)
--------------->**node 4:** deepest = (**node 3**, depth 5)
-------------------->**node 3:** deepest = **undefined**
------------>**node 5:** deepest = **undefined**
------------>**node 6:** deepest = **undefined**
--->**node 13:** deepest = (**node 7**, depth 4)
------->**node 7:** deepest = (**node 11**, depth 4)
----------->**node 11**: deepest = (**node 12**, depth 4)
--------------->**node 12**: deepest = **undefined**

So I'll assume we have that stored in an array somewhere like:

deepestChild[0] = 15
deepestChild[1] = undefined
deepestChild[2] = 0
deepestChild[3] = undefined
deepestChild[4] = 3
deepestChild[5] = undefined
deepestChild[6] = undefined
deepestChild[7] = 11
deepestChild[8] = 10

Upvote 8 Share

deepestChild[10] = undefined

deepestChild[11] = 12

deepestChild[12] = undefined

deepestChild[13] = 7

deepestChild[14] = undefined

deepestChild[15] = 4

Now, let's go through the level-by-level traversal.

Let X = node, and E = number of unvisited edges above that node

Queue

(X = 9, E = 0) (start with root node, use 0 unvisited ancestor edges for base case)

Dequeue(X = 9) -> get (E = 0)

Enqueue(X = 8, E = 1+0 = 1)

Enqueue(X = 2, E = 1+0 = 1)

Enqueue(X = 13, E = 1+0 = 1)

Queue

(X = 8, E = 1), (X = 2, E = 1), (X = 13, E = 1)

Dequeue(X = 8) -> get (E = 1)

Enqueue(X = 10, E = 1+1 = 2)

Queue

(X = 2, E = 1), (X = 13, E = 1), (X = 10, E = 2)

Dequeue(X = 2) -> get (E = 1)

Enqueue(X = 1, E = 1+0 = 1)

Enqueue(X = 0, E = 1+1 = 2)

Queue

(X = 13, E = 1), (X = 10, E = 2), (X = 1, E = 1), (X = 0, E = 2)

Dequeue(X = 13) -> get (E = 1)

Enqueue(X = 7, E = 1+1 = 2)

Queue

(X = 10, E = 2), (X = 1, E = 1), (X = 0, E = 2), (X = 7, E = 2)

Dequeue(X = 10) -> get (E = 2)

No children, so leaf node. Store rank[2] = [10].

Queue

(X = 1, E = 1), (X = 0, E = 2), (X = 7, E = 2)

Dequeue (X = 1) -> get (E = 1)

No children, so leaf node. Store rank[1] = [1]

Queue

(X = 0, E = 2), (X = 7, E = 2)

Dequeue(X = 0) -> get (E = 2)

Enqueue(X = 14, E = 1+0 = 1)

Enqueue(X = 15, E = 1+2 = 3)

Enqueue(X = 5, E = 1+0 = 1)

Enqueue(X = 6, E = 1+0 = 1)

Queue

(X = 7, E = 2), (X = 14, E = 1), (X = 15, E = 3), (X = 5, E = 1), (X = 6, E = 1)

Dequeue(X = 7) -> get (E = 2)

Enqueue(X = 11, E = 1 + 2 = 3)

Queue

(X = 14, E = 1), (X = 15, E = 3), (X = 5, E = 1), (X = 6, E = 1), (X = 11, E = 3)

Dequeue(X = 14) -> get (E = 1)

No children, so leaf node. Update rank[1] = [1, 14]

Queue

(X = 15, E = 3), (X = 5, E = 1), (X = 6, E = 1), (X = 11, E = 3)

Dequeue(X = 15) -> get (E = 3)

Enqueue(X = 4, E = 1+3 = 4)

Queue

(X = 5, E = 1), (X = 6, E = 1), (X = 11, E = 3), (X = 4, E = 4)

Dequeue(X = 5) -> get (E = 1)

No children, so leaf node. Update rank[1] = [1, 14, 5]

Queue

(X = 6, E = 1), (X = 11, E = 3), (X = 4, E = 4)

Dequeue(X = 6) -> get (E = 1)

No children, so leaf node. Update rank[1] = [1, 14, 5, 6]

Queue

(X = 11, E = 3), (X = 4, E = 4)

Dequeue(X = 11) -> get (E = 3)

Enqueue(X = 12, E = 1+3 = 4)

Queue

(X = 4, E = 4), (X = 12, E = 4)

Dequeue(X = 4) -> get (E = 4)

Queue
(X = 12, E = 4), (X = 3, E = 5)

Dequeue(X = 12) -> get (E = 4)
No children, so leaf node. Update rank[4] = [12]

Queue
(X = 3, E = 5)

Dequeue(X = 3) -> get (E = 5)
No children, so leaf node. Update rank[5] = [3]

Queue is now empty.

By now, we have:

rank[0] = undefined
rank[1] = [1, 14, 5, 6]
rank[2] = [10]
rank[3] = undefined
rank[4] = [12]
rank[5] = [3]
rank[6] = undefined
rank[7] = undefined
rank[8] = undefined
rank[9] = undefined
rank[10] = undefined
rank[11] = undefined
rank[12] = undefined
rank[13] = undefined
rank[14] = undefined
rank[15] = undefined

Now, we build our array mapping values to ranks:

valueToRank[0] = undefined
valueToRank[1] = 1
valueToRank[2] = undefined
valueToRank[3] = 5
valueToRank[4] = undefined
valueToRank[5] = 1
valueToRank[6] = 1
valueToRank[7] = undefined
valueToRank[8] = undefined
valueToRank[9] = undefined
valueToRank[10] = 2
valueToRank[11] = undefined
valueToRank[12] = 4

valueToRank[14] = 1
valueToRank[15] = undefined

Then we build our array mapping rank to starting index for that rank:

rankToIndex[0] = undefined
rankToIndex[1] = 3
rankToIndex[2] = 2
rankToIndex[3] = undefined
rankToIndex[4] = 1
rankToIndex[5] = 0
rankToIndex[6] = undefined
rankToIndex[7] = undefined
rankToIndex[8] = undefined
rankToIndex[9] = undefined
rankToIndex[10] = undefined
rankToIndex[11] = undefined
rankToIndex[12] = undefined
rankToIndex[13] = undefined
rankToIndex[14] = undefined
rankToIndex[15] = undefined

Then we iterate over the values from 0 to $n-1$.

0
valueToRank[0] = undefined
Skip

1
valueToRank[1] = 1
rankToIndex[1] = 3, update rankToIndex[1] to 4
answer[3] = 1

2
valueToRank[2] = undefined
Skip

3
valueToRank[3] = 5
rankToIndex[5] = 0, update rankToIndex[5] to 1
answer[0] = 3

4
valueToRank[4] = undefined
Skip

5
valueToRank[5] = 1
rankToIndex[1] = 4, update rankToIndex[1] to 5

6

valueToRank[6] = 1

rankToIndex[1] = 5, update rankToIndex[1] to 6

answer[5] = 6

7

valueToRank[7] = undefined

Skip

8

valueToRank[8] = undefned

Skip

9

valueToRank[9] = undefined

Skip

10

valueToRank[10] = 2

rankToIndex[2] = 2, update rankToIndex[2] to 3

answer[2] = 10

11

valueToRank[11] = undefined

Skip

12

valueToRank[12] = 4

rankToIndex[4] = 1, update rankToIndex[4] to 2

answer[1] = 12

13

valueToRank[13] = undefined

Skip

14

valueToRank[14] = 1

rankToIndex[1] = 6, update rankToIndex[1] to 7

answer[6] = 14

15

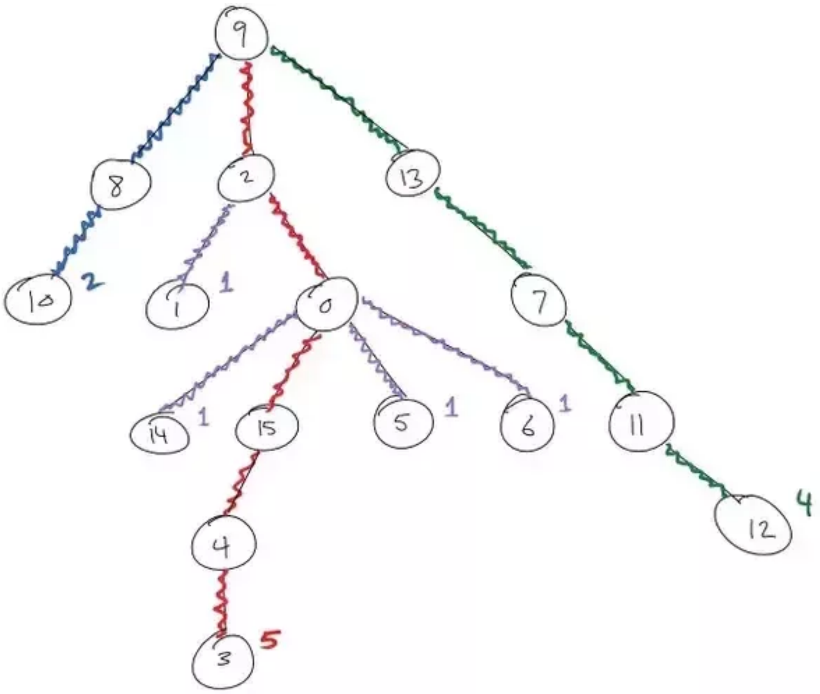
valueToRank[15] = undefind

Skip

Answer

[3, 12, 10, 1, 5, 6, 14]

Now let's check that answer:



It looks like it worked. It also works for the graph in the original question.

This algorithm runs in $\Theta(n)$ time and space.

Let's just hope it works for all cases... I probably missed something :)

Java code (really crappy, but here in case anyone needs clarification)

```
import java.util.*;
public class LeafOrder {
    public static void main(String[] args) {
        /***************************************************************
         * Build tree
        \***************************************************************
        final int n = 16;
        TreeNode[] nodes = new TreeNode[n];
        for (int i = 0; i < n; i++) {
            nodes[i] = new TreeNode(i);
        }
        // Level 1
        TreeNode root = nodes[9];
        // Level 2
        root.addChild(nodes[8]);
        root.addChild(nodes[2]);
        root.addChild(nodes[13]);
        // Level 3
        nodes[8].addChild(nodes[10]);
        nodes[2].addChild(nodes[1]);
        nodes[2].addChild(nodes[0]);
        nodes[13].addChild(nodes[7]);
        // Level 4
        nodes[0].addChild(nodes[14]);
        nodes[0].addChild(nodes[15]);
        nodes[0].addChild(nodes[5]);
        nodes[0].addChild(nodes[6]);
        nodes[7].addChild(nodes[11]);
```

```
        nodes[15].addChild(nodes[4]);
        nodes[11].addChild(nodes[12]);
        // Level 6
        nodes[4].addChild(nodes[3]);
        /*****************************************************************
         * Preprocessing
        \*****************************************************************
        preprocessDeepestReachingChildren(root);
        /*****************************************************************
         * Level-by-level traversal
        \*****************************************************************
        Queue<NodeRank> queue = new LinkedList<NodeRank>();
        queue.offer(new NodeRank(root, 0));
        List<List<Integer>> rank = new ArrayList<List<Integer>>(n);
        boolean[] valueIsLeaf = new boolean[n];
        int[] valueToRank = new int[n];
        int[] rankToIndex = new int[n];
        for (int i = 0; i < n; i++) {
            rank.add(new ArrayList<Integer>());
        }
        while (!queue.isEmpty()) {
            NodeRank dequeued = queue.poll();
            TreeNode currentNode = dequeued.getNode();
            int currentValue = currentNode.getValue();
            int currentRank = dequeued.getRank();
            ArrayList<TreeNode> children = currentNode.getChildren();
            TreeNode deepestReachingChild = currentNode.getDeepestReachi
            int numChildren = children.size();
            if (numChildren == 0) {
                List<Integer> rankList = rank.get(currentRank);
                rankList.add(currentValue);
                valueToRank[currentValue] = currentRank;
                valueIsLeaf[currentValue] = true;
            }
            for (int i = 0; i < numChildren; i++) {
                TreeNode currentChild = children.get(i);
                int currentChildRank = 1 + ((currentChild == deepestReac
                queue.offer(new NodeRank(currentChild, currentChildRank)
            }
        }
        /*****************************************************************
         * Building answer in sorted order
        \*****************************************************************
        for (int i = n - 1, currentIndex = 0; i >= 0; i--) {
            List<Integer> rankList = rank.get(i);
            int rankListSize = rankList.size();
            if (rankListSize == 0) {
                continue;
            }
            rankToIndex[i] = currentIndex;
            currentIndex += rankListSize;
        }
        int[] answer = new int[n];
        int highestIndexSet = 0;
        for (int i = 0; i < n; i++) {
            if (!valueIsLeaf[i]) {
                continue;
            }
            int rankForValue = valueToRank[i];
```

```java
            answer[indexForRank] = i;
            highestIndexSet = Math.max(highestIndexSet, indexForRank);
        }
        /****************************************************************
         * Outputting answer
        \****************************************************************
        for (int i = 0; i <= highestIndexSet; i++) {
            System.out.print(answer[i] + ((i != highestIndexSet) ? ", "
        }
        System.out.println();
    }
    public static int preprocessDeepestReachingChildren(TreeNode root) {
        return preprocessDeepestReachingChildren(root, 0);
    }
    public static int preprocessDeepestReachingChildren(TreeNode root, i
        int maxDepth = 0;
        TreeNode maxChild = null;
        ArrayList<TreeNode> children = root.getChildren();
        int numChildren = children.size();
        if (numChildren == 0) {
            return depth;
        }
        for (int i = 0; i < numChildren; i++) {
            TreeNode currentChild = children.get(i);
            int currentChildDepth = preprocessDeepestReachingChildren(cu
            if (maxChild == null || currentChildDepth > maxDepth || (cur
                maxDepth = currentChildDepth;
                maxChild = currentChild;
            }
        }
        root.setDeepestReachingChild(maxChild);
        return maxDepth;
    }
}
class TreeNode {
    private int value;
    private ArrayList<TreeNode> children;
    private TreeNode deepestReachingChild;
    public TreeNode(int nodeValue) {
        this.value = nodeValue;
        this.children = new ArrayList<TreeNode>();
    }
    public int getValue() {
        return this.value;
    }
    public void addChild(TreeNode child) {
        this.children.add(child);
    }
    public ArrayList<TreeNode> getChildren() {
        return this.children;
    }
    public TreeNode getDeepestReachingChild() {
        return this.deepestReachingChild;
    }
    public void setDeepestReachingChild(TreeNode child) {
        this.deepestReachingChild = child;
    }
}
class NodeRank {
```

```
    public NodeRank(TreeNode node, int rank) {
        this.node = node;
        this.rank = rank;
    }
    public TreeNode getNode() {
        return this.node;
    }
    public int getRank() {
        return this.rank;
    }
}
```

Edit: Another way to solve this if you don't want to use a queue is to do a depth-first traversal after preprocessing. Every time you get to a node, you look at its children and find the one that reaches the deepest into the tree and is smallest (we know this from preprocessing). Recurse down through that child, passing along the current depth + 1. When you come back from the recursion, visit the other children, passing a depth of 0. When you reach a leave, store its depth as its rank.

That approach would look like:

```
import java.util.*;
public class LeafOrder {
    public static void main(String[] args) {
        /*****************************************************************
         * Build tree
        \*****************************************************************
        final int n = 16;
        TreeNode[] nodes = new TreeNode[n];
        for (int i = 0; i < n; i++) {
            nodes[i] = new TreeNode(i);
        }
        // Level 1
        TreeNode root = nodes[9];
        // Level 2
        root.addChild(nodes[8]);
        root.addChild(nodes[2]);
        root.addChild(nodes[13]);
        // Level 3
        nodes[8].addChild(nodes[10]);
        nodes[2].addChild(nodes[1]);
        nodes[2].addChild(nodes[0]);
        nodes[13].addChild(nodes[7]);
        // Level 4
        nodes[0].addChild(nodes[14]);
        nodes[0].addChild(nodes[15]);
        nodes[0].addChild(nodes[5]);
        nodes[0].addChild(nodes[6]);
        nodes[7].addChild(nodes[11]);
        // Level 5
        nodes[15].addChild(nodes[4]);
        nodes[11].addChild(nodes[12]);
        // Level 6
        nodes[4].addChild(nodes[3]);
        /*****************************************************************
```

```
        \*******************************************************************
        List<List<Integer>> rank = new ArrayList<List<Integer>>(n);
        boolean[] valueIsLeaf = new boolean[n];
        int[] valueToRank = new int[n];
        for (int i = 0; i < n; i++) {
            rank.add(new ArrayList<Integer>());
        }
        preprocessDeepestReachingChildren(root);
        preprocessRanks(root, rank, valueToRank, valueIsLeaf);
        /*******************************************************************
         * Building answer in sorted order
        \*******************************************************************
        int[] rankToIndex = new int[n];
        for (int i = n - 1, currentIndex = 0; i >= 0; i--) {
            List<Integer> rankList = rank.get(i);
            int rankListSize = rankList.size();
            if (rankListSize == 0) {
                continue;
            }
            rankToIndex[i] = currentIndex;
            currentIndex += rankListSize;
        }
        int[] answer = new int[n];
        int highestIndexSet = 0;
        for (int i = 0; i < n; i++) {
            if (!valueIsLeaf[i]) {
                continue;
            }
            int rankForValue = valueToRank[i];
            int indexForRank = rankToIndex[rankForValue];
            rankToIndex[rankForValue]++;
            answer[indexForRank] = i;
            highestIndexSet = Math.max(highestIndexSet, indexForRank);
        }
        /*******************************************************************
         * Outputting answer
        \*******************************************************************
        for (int i = 0; i <= highestIndexSet; i++) {
            System.out.print(answer[i] + ((i != highestIndexSet) ? ", "
        }
        System.out.println();
    }
    public static int preprocessDeepestReachingChildren(TreeNode root) {
        return preprocessDeepestReachingChildren(root, 0);
    }
    public static int preprocessDeepestReachingChildren(TreeNode root, i
        int maxDepth = 0;
        TreeNode maxChild = null;
        ArrayList<TreeNode> children = root.getChildren();
        int numChildren = children.size();
        if (numChildren == 0) {
            return depth;
        }
        for (int i = 0; i < numChildren; i++) {
            TreeNode currentChild = children.get(i);
            int currentChildDepth = preprocessDeepestReachingChildren(cu
            if (maxChild == null || currentChildDepth > maxDepth || (cur
                maxDepth = currentChildDepth;
                maxChild = currentChild;
```

Upvote 8 Share ooo

```java
        root.setDeepestReachingChild(maxChild);
        return maxDepth;
    }
    public static void preprocessRanks(TreeNode root, List<List<Integer>:
        preprocessRanks(root, 0, rank, valueToRank, valueIsLeaf);
    }
    public static void preprocessRanks(TreeNode root, int currentRank, L
        TreeNode deepestReachingChild = root.getDeepestReachingChild();
        ArrayList<TreeNode> children = root.getChildren();
        int numChildren = children.size();
        if (numChildren == 0) {
            int currentValue = root.getValue();
            List<Integer> rankList = rank.get(currentRank);
            rankList.add(currentValue);
            valueToRank[currentValue] = currentRank;
            valueIsLeaf[currentValue] = true;
        }
        for (int i = 0; i < numChildren; i++) {
            TreeNode currentChild = children.get(i);
            preprocessRanks(currentChild, 1 + ((currentChild == deepestR
        }
    }
}
class TreeNode {
    private int value;
    private ArrayList<TreeNode> children;
    private TreeNode deepestReachingChild;
    public TreeNode(int nodeValue) {
        this.value = nodeValue;
        this.children = new ArrayList<TreeNode>();
    }
    public int getValue() {
        return this.value;
    }
    public void addChild(TreeNode child) {
        this.children.add(child);
    }
    public ArrayList<TreeNode> getChildren() {
        return this.children;
    }
    public TreeNode getDeepestReachingChild() {
        return this.deepestReachingChild;
    }
    public void setDeepestReachingChild(TreeNode child) {
        this.deepestReachingChild = child;
    }
}
class NodeRank {
    private TreeNode node;
    private int rank;
    public NodeRank(TreeNode node, int rank) {
        this.node = node;
        this.rank = rank;
    }
    public TreeNode getNode() {
        return this.node;
    }
    public int getRank() {
        return this.rank;
```

3.6k Views · View Upvoters

Your feedback is private.

Is this answer still relevant and up to date?

Yes No

Add a comment...

Recommended All

Miguel Oliveira
Mar 9, 2014 · 2 upvotes including John Kurlak

The F() part is brilliant. I believe the algorithm is correct. Thank you!

Just a minor bug:
"Dequeue(X = 5) -> get (E = 1)
No children, so leaf node. Update rank[1] = [1, 14, 15]"

After this, you're using 15 instead of 5, including the solution. Should be [3, 12, 10, 1, 5, 6, 14].

Reply · Upvote · Downvote · Report

John Kurlak
Mar 9, 2014 · 2 upvotes including Miguel Oliveira

Thanks for catching that. Answer updated!

Reply · Upvote · Downvote · Report

Top Stories from Your Feed

Answer written · **Philosophy of Everyday Life** · Topic you might like · 11h

What are some undeniable facts about life?

Takudzwa Razemba
Answered 11h ago

1. Gucci doesn't go on sale. Because it holds its name and it knows its value. When you know who you are and what name has been placed on you, nobody can reduce you. Because seduction is to put your v...

Read In Feed

Answer written · **Philosophy of Everyday Life** · Topic you might like · 8h

What do you need right now?

Iva Izabela Miholic, The Duchess of Destruction at Consiglio Devastations
Answered 8h ago

This is Habib Fanny. I am guest writing this for Iva. We have been waiting for her to hit 33,333 followers for months. But you guys have been following her way too fast for the past

Read In Feed

Answer written · **Philosophy of Everyday Life** · Topic you might like · Thu

What doesn't make sense to you?

Takudzwa Razemba
Updated 44m ago

1. The fact that we all just walk around pretending it's not weird that one of our hands is better at stuff than the other.

2. You're only afraid of being alone in the dark because you're afraid you're no...

(more)

Read In Feed

Upvote 8 Share