DATA STRUCTURES AND ALGORITHMS

Extra reading 9

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

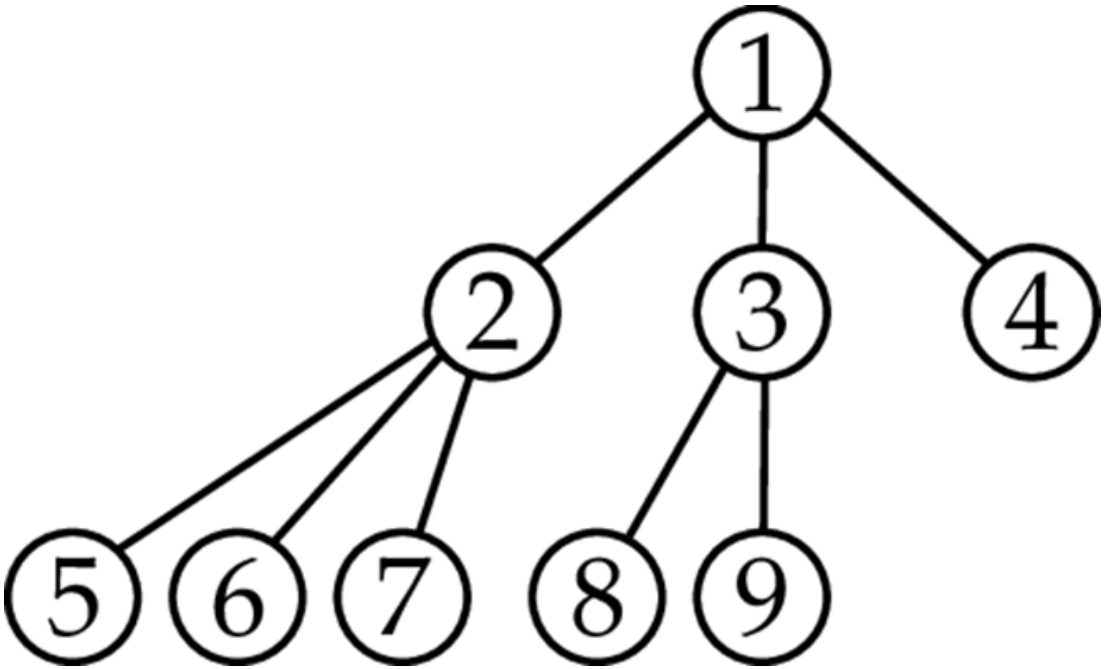
2023 - 2024

Content

- This extra reading gives the explanation and solution for the problem statement from Extra reading 8.

Problem statement - Recap I

- You are given a tree of out-degree K with N nodes or, in other words, each node can have at most K children. The tree is constructed so it is of the "lowest energy": the nodes are placed in a new depth of the tree only when all the places (from left to right) in the previous depth have been filled. This is also the order of enumerating the nodes, starting with 1. Figure 1 depicts an example of a tree of order 3 with 9 nodes.


1
2
3
4
5
6
7
8
9

- Given two nodes, x and y ($x \neq y$, both valid numbers in the tree) determine the minimum number of steps to get from node x to y. For the above example:
  - minDist(5, 9) = 4
  - minDist(5, 3) = 3
  - minDist(5, 7) = 2
  - minDist(7, 2) = 1

Solution

- If we look at how the tree is built, we can see that it is the same principle that we use for a binary heap (filling nodes from left to right and going to the next level only when the current one is full), but here, instead of having 2 children, every node will have K children (so it is like a K-ary heap, instead of binary heap).
- All those rules that we use to navigate in a binary heap (going from a node to its children and going from a child to its parent) can be adapted to this case as well. For this problem it is actually enough to go from a node to its parent. Having the two nodes, we can go up in the tree (and count how many steps we make) until we find the first common parent (technically called Least Common Ancestor).

- So how can we go from a node to its parent?
- For a binary heap, parent of node p was on position p/2.
- If we look at the heap from the figure, we can see that parent of node p is on position (p+1)/3.
- If we draw a heap with K = 4, and look at the nodes, we will see that the parent of node p is on position (p+2)/4
- Let's see how we can find a general expression. Intuitively we know that we will have to divide by K.

- Let's see the form of the children of a given node:
  - Root has the number 1
  - Children of node 1 will go from 2 to $K + 1$ (this is how we get K children)
  - Then children of node 2 will go from $K + 2$ to $2K + 1$
  - Then children of node 3 will go from $2K + 2$ to $3K + 1$
  - etc.
- We can see that for a node X, its children will be from $(X - 1) * K + 2$ to $X * K + 1$ (Obs. You can easily check this rule for $K = 2$ – the regular binary heap – and $K = 3$ - the example from Figure 1.)

- We know how to go from a parent to its children, but we want the opposite, we want to go from a child to the parent.
- And we want a formula so that when we divide the child's number by K, the result should be X. So we should make the "smallest" node $(X-1)*K+2$ to be $X*K \rightarrow$ we can do this by adding to it K–2.
- So, for the "smallest" (leftmost) child, its parent will be:

$$\frac{((X-1)*K+2+K\text{–}2)}{K} \Rightarrow$$

$$\frac{X*K\text{–}K+2+K\text{–}2}{K} = X$$

- If we use the same rule, for the "greatest" (rightmost) child, its parent will be:

$$\frac{X * K + 1 + K\text{–}2}{K} \Rightarrow$$

$$\frac{K * (X + 1)\text{–}1}{K}$$

- This is less than $K * (X + 1)/K$ (which would be $X + 1$), so the result (since we are doing integer division) is X.

- So now we know (and we have proof that this is correct) how to go from a node Y to its parent:

$$\frac{(Y + K\text{–}2)}{K}$$

- So now we need to figure out how to find the common parent. From the example, we can see that it might happen that one of the nodes is already the common parent (ex: 7 and 2, 2 is the common parent). We might have to go up the same number of steps for both nodes (ex: 5, 9), but we might need to make more steps with one node than with the other (ex: 5, 3).

- So we cannot just do something like:

```
steps ← 0
while x ≠ y execute:
    x ← parent of x
    y ← parent of y
    steps ← steps + 2
end-while
```

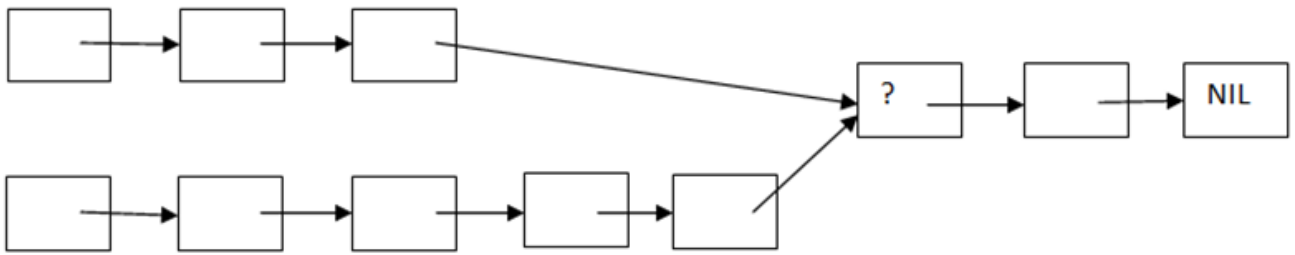
- It is a lot better to change just one node at a time, but which one?

- We can see that if $x > y$, it means that either x is at a lower level than y or they are on the same level, but x is to the right of y. In any case, x is definitely not the common parent. So, if we just change one node, we should change the one with the bigger number.

```
function minDist(N, K, x, y) is:
    distance ← 0
    while x≠ y execute:
        if x > y then
            x ← (x + K - 2) / K
            distance ← distance + 1
        else
            y ← (y + K - 2) / K
            distance ← distance + 1
        end-if
    end-while
    minDist ← distance
end-function
```

- The fact that the nodes are numbered allows us to know which node to change. But what if we did not know this? What if we had a function to return the parent of a node, but the nodes were not numbered ascendingly?

- This is the same as the following problem: we have two singly linked lists, which at a given point merge into the same list (see the figure). You only have the first node of the two lists, nothing else. How would you find the first common node?



- If you rotate the above figure with 90 degrees counterclockwise, you will have the same situation as we have in case of the heap, considering that we only have the paths from nodes x and y to the root, ignoring the remaining nodes

- One solution is to go through one list, and for every node of that list, check if that node appears in the second. The moment you find the first such node, you have the common node. But this is inefficient from a time complexity perspective.

- A more efficient solution (from the time complexity perspective, because it will require some extra memory) is to take two stacks: in one we will add the nodes from the first list (from the first to the last) and in the other the nodes from the second list. Both stacks will have on top the common node(s). So we just have to remove in parallel from them until we get the last common node.

- The above idea can be translated to our heap problem as well:
    - In a stack we will put the "path" from node x to the root. For example: 5, 2, 1.
    - In another stack we will put the "path" from node y to the root: For example: 3, 1
    - Top of the stack contains the common nodes, we have to eliminate those and then count how many nodes we have left in the two stacks. This is the minimum distances between nodes x and y.