[CSED233-01] Data Structure

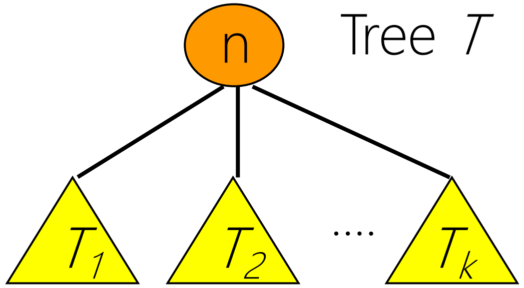
Tree (2)

Jaesik Park

POSTECH

Recap: Tree Traversal

- Types of traversals

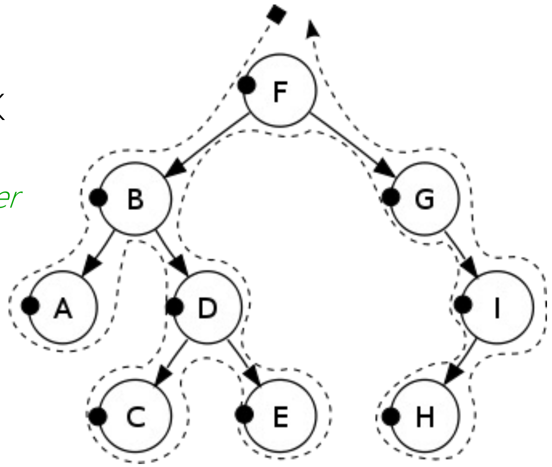


- $Preorder(T) = \langle n,\ Preorder(T_1), \ldots, Preorder(T_k) \rangle$
- $Postorder(T) = \langle Postorder(T_1), \ldots, Postorder(T_k),\ n \rangle$
- $Inorder(T) = \langle Inorder(T_1),\ n,\ Inorder(T_2), \ldots, Inorder(T_k) \rangle$
  - No natural definition of *Inorder*

Binary Tree Traversals

- Passing through the tree & visit each of its nodes
- Depth-first search
  - Go down first
  - Recursive way – via (call) STACK
  - 3 variations
    - *Pre-order, Post-order, In-order*

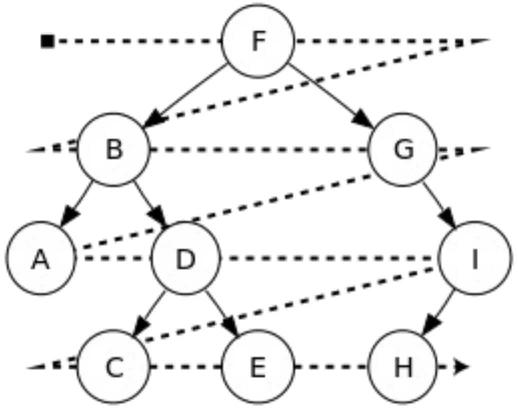
  While True {
  current node = stack.pop()
  For the current node's child
      stack.push(child)
  }



$O(n)$, where n = # of nodes in a tree

- Breadth-first search
  - Go across first (same level)
  - Implemented via QUEUE
  - No variation
    - *Level-order*

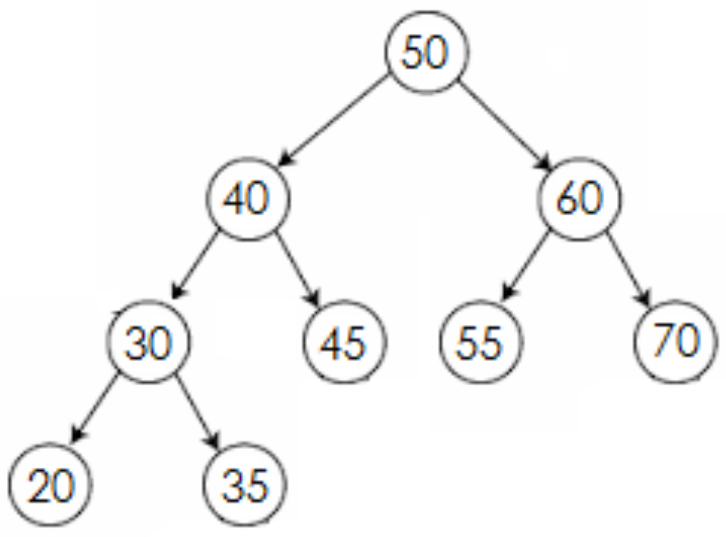
  While True {
  current node = queue.dequeue()
  For the current node's child
      queue.enqueue(child)
  }



Question

- We don’t know the exact structure of a binary tree,
- If we know the traversals of the binary tree, can we reconstruct the true structure of the tree?

Binary Tree



Traversals
- Inorder
- Postorder
- Preorder

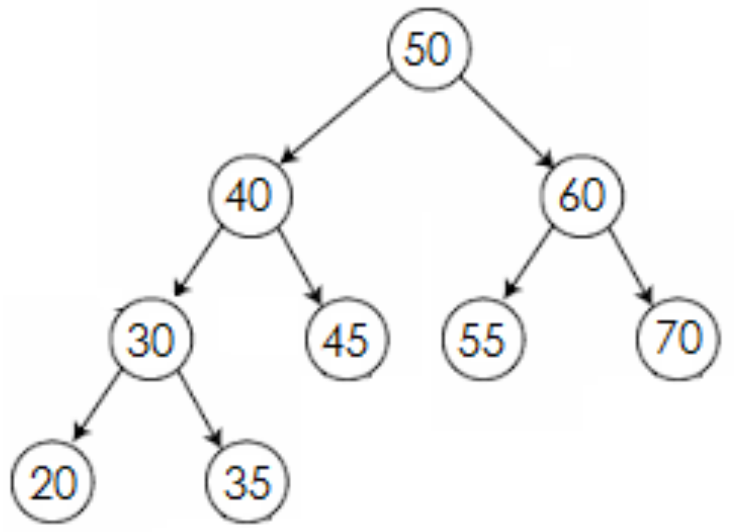
Single Traversal

- Any single traversal sequence (*pre-*, *in-* or *post-order*) cannot describes the underlying tree uniquely

- Inorder = <20,30,35,40,45,50,55,60,70>

- Postorder = <20,35,30,45,40,55,70,60,50>

Binary Tree



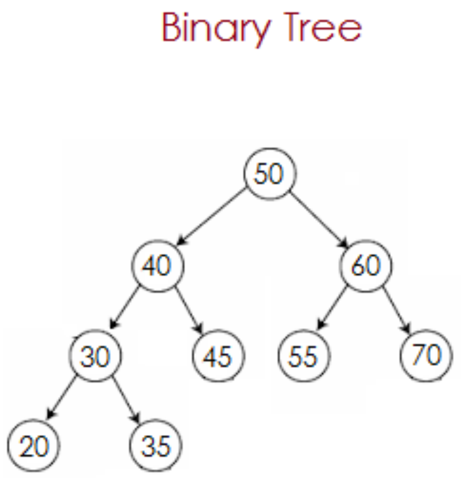
Unique Binary Tree (by Two Traversals)

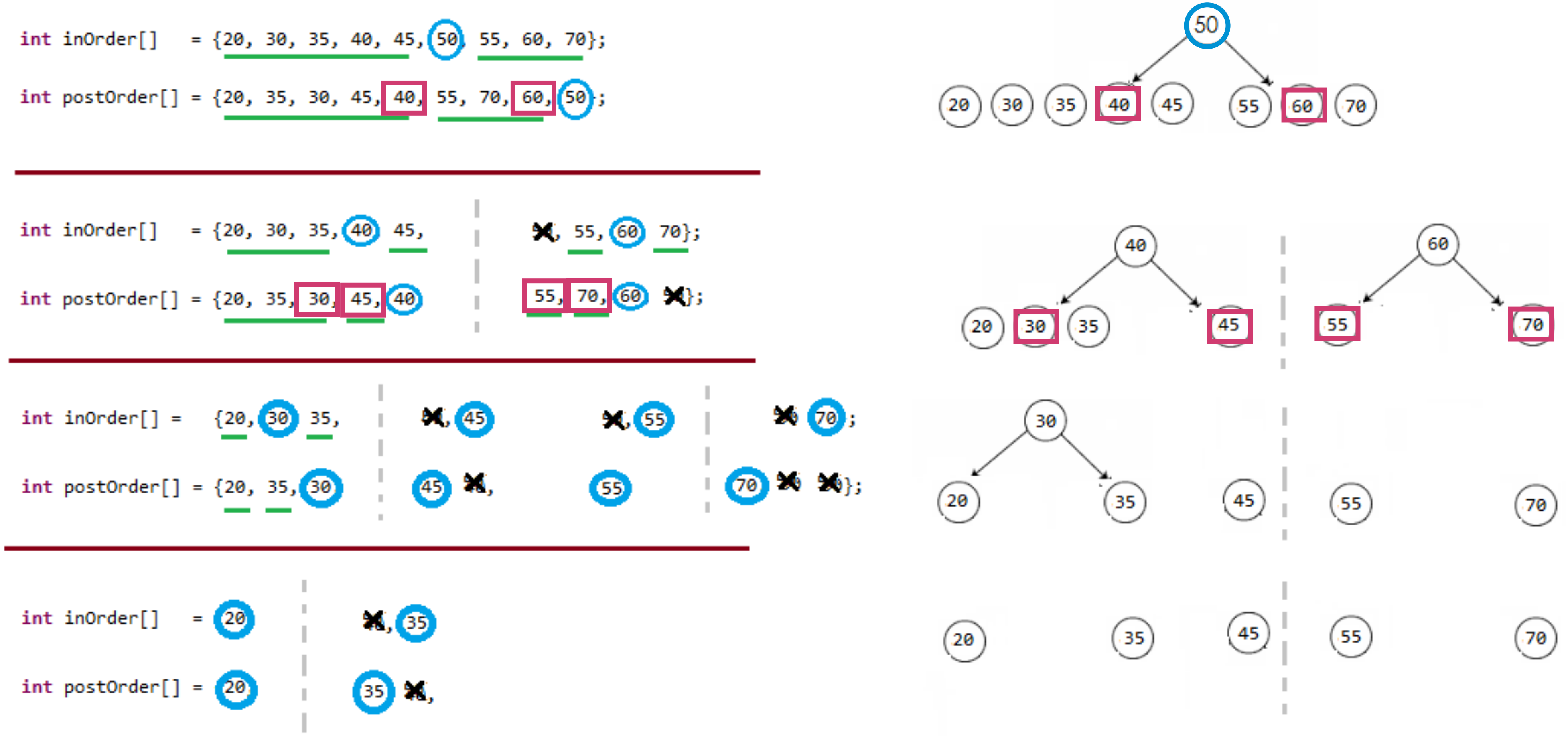
- We can identify the binary tree uniquely by two traversal sequences like:
  - (*postorder* & *inorder*), (*preorder* & *inorder*), (*level-order* & *inorder*)
    - *inorder*: to find Left & Right child/subtrees
    - *postorder*: to find the Root (the last in *postorder*)
    - *preorder*: to find the Root (the first/ in *preorder*)
    - *level-order*: to find the Root

- However, the other combinations leaves some ambiguity in the tree structure

Binary Tree (by Two Traversals)

- Example:



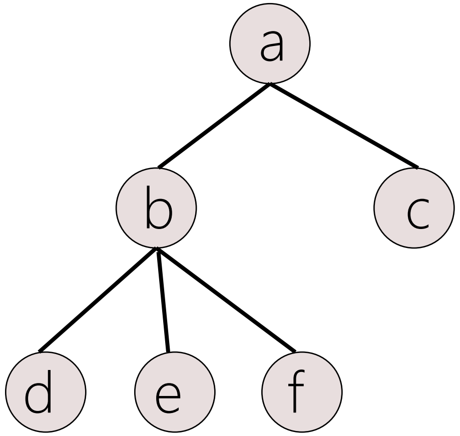


Algorithm

- Algorithm (when Preorder & Inorder is given)
  - (1) *Root* = the first node in the preorder
  - (2) Find the left child:
    - *Left_subtree* = all nodes that are left to the *Root* in the inorder
    - *Left_child* = the first node in the preorder of the *Left_subtree*
  - (3) In the same way, find the right child:
    - *Right subtree* = all nodes that are right to the Root in the inorder
    - *Right_child* = the first node in the preorder of the *Right_subtree*
  - (4) Repeat the steps 2 & 3 with each new node (until every node is not visited in preorder)

General Tree Implementations

- General k-ary tree is a tree in which each node has no more than k children
- Implementations
  - Simple array implementation
  - *List-of-Children* implementation
  - *Left-Child/Right-Sibling* implementation
  - …



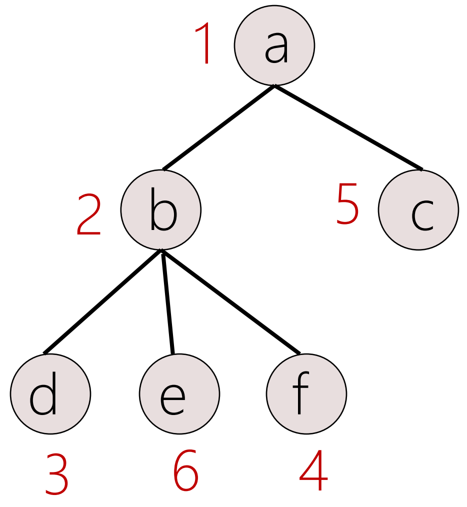
- Again, there would be other implementations as well
- No rules here, they have their own pros/cons
- Let's analyze them now

Simple Approach

- Parent point representation
- Two linear arrays – parallel to each other
  - Tree[k] = m ⇔ node m is the parent of node k
  - Label[k] : the label (value, info) of node k
- Of course, you can also use structure

	1	*2*	*3*	*4*	*5*	*6*	*.....*	*n*
Tree[]	*0*	*1*	*2*	*2*	*1*	*2*		
Label[]	*a*	*b*	*d*	*f*	*c*	*e*		

Not specify the order of children

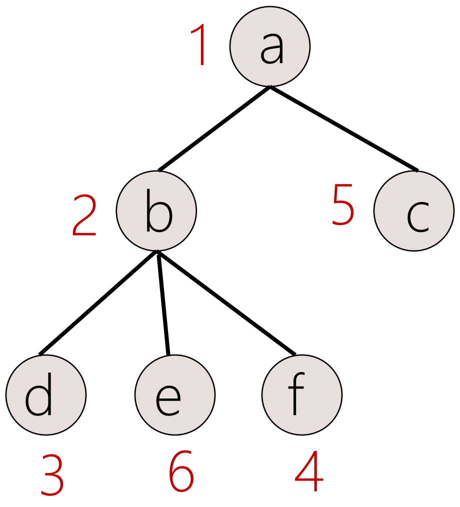


Simple Approach: Time Complexity

- *Parent*(*k*, *T*)
  - What is the parent node of the node index k in the tree T?
  - Easy to compute: O(1)
- *Leftmost_Child*(*k*, *T*)
  - What is the leftmost child of the node index k in the tree T?
  - Not well defined (no order of children)
  - O(*n*)

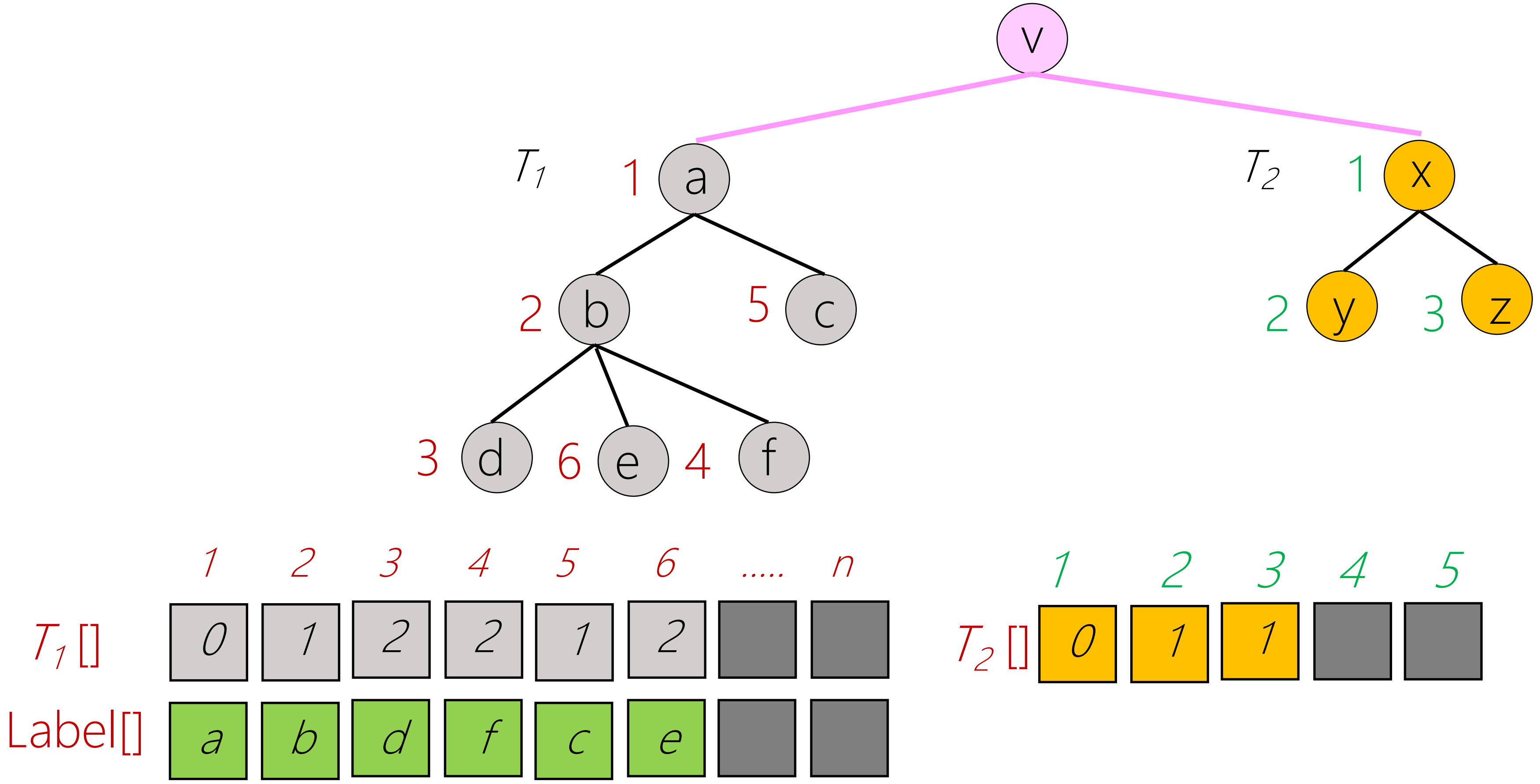
	1	*2*	*3*	*4*	*5*	*6*	.....	*n*
Tree[]	*0*	*1*	*2*	*2*	*1*	*2*		
Label[]	*a*	*b*	*d*	*f*	*c*	*e*		

Tree T



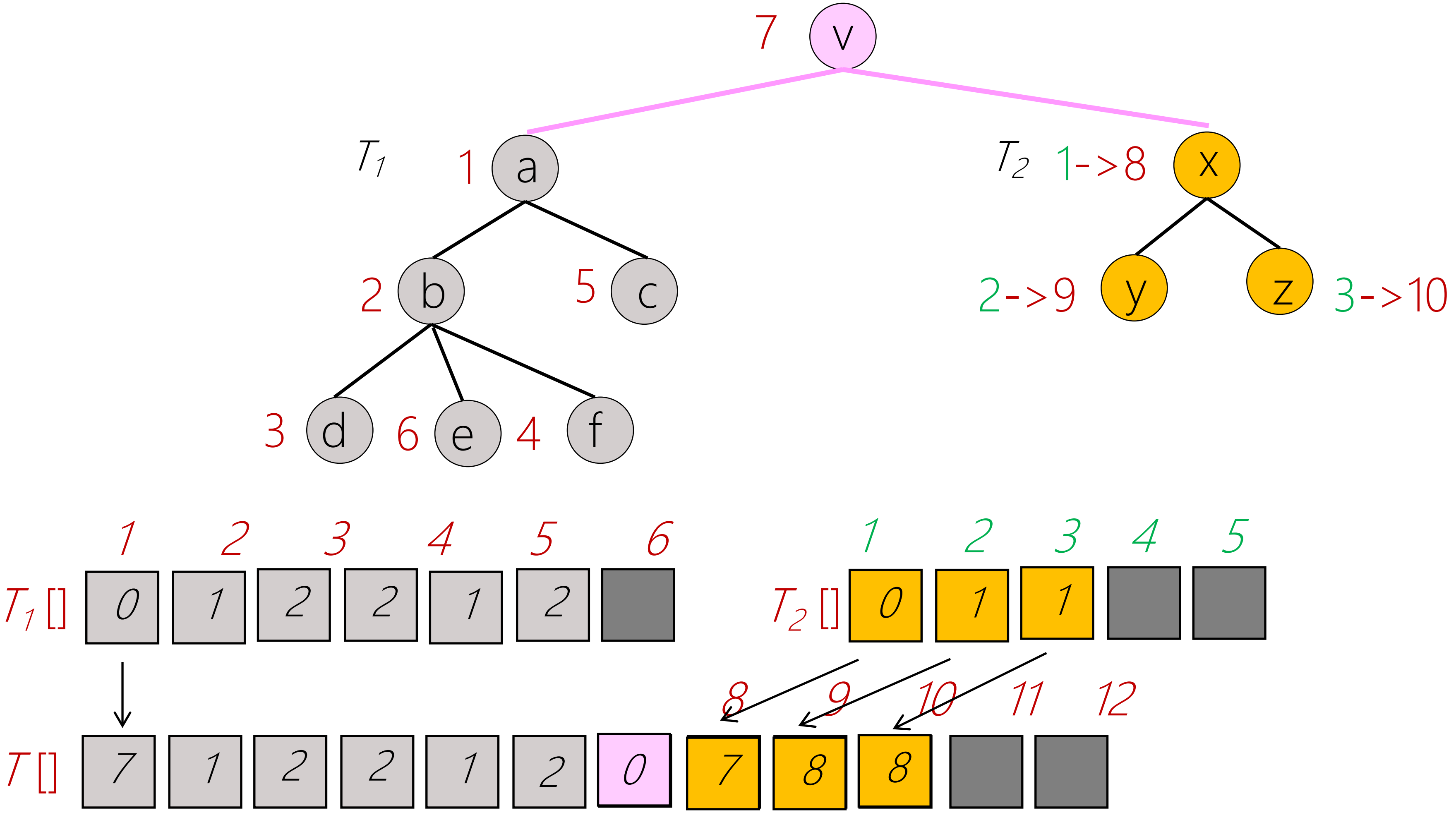
Simple Approach: Time Complexity

- $Create_m(v, T_1, T_2, \ldots, T_m)$
  - Make a tree with a root v and subtrees $T_1, T_2, \ldots, T_m$



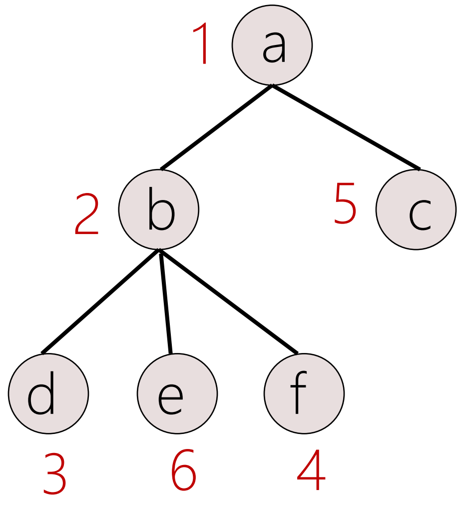
Simple Approach: Time Complexity

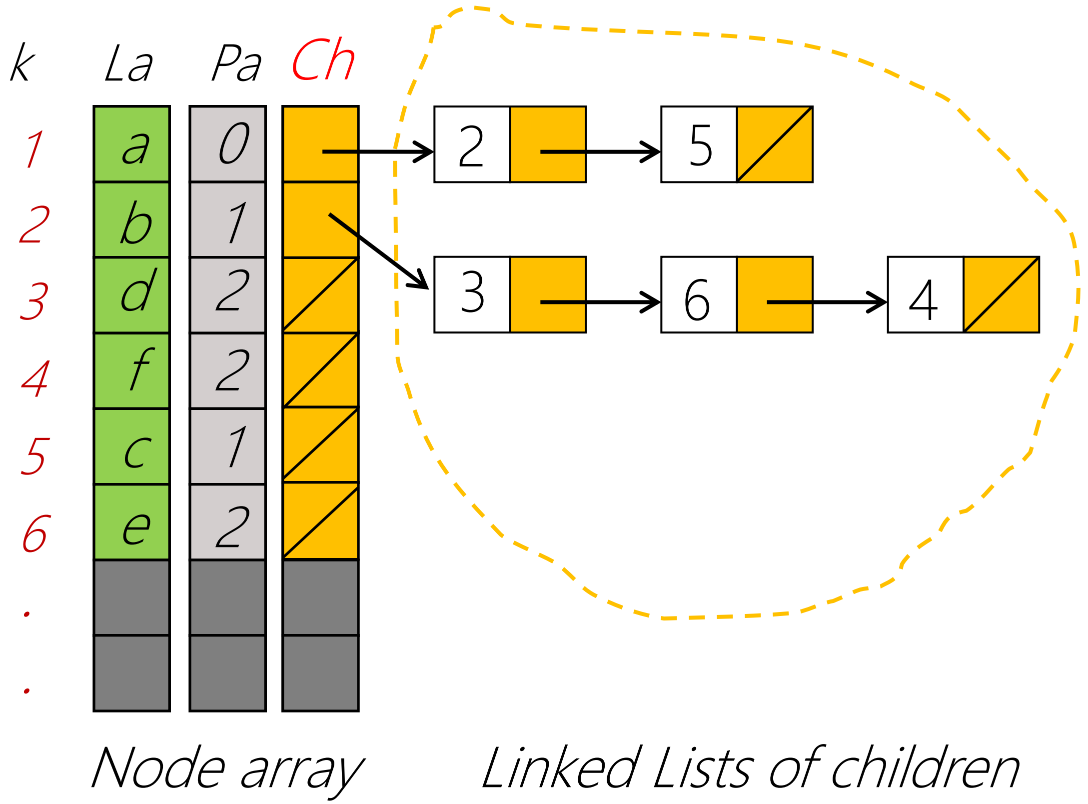
- $Create_m(v, T_1, T_2, \ldots, T_m)$
  - Quite hard to handle as we should change the tree indices for $T_2, \ldots, T_m$



List-of-Children Approach

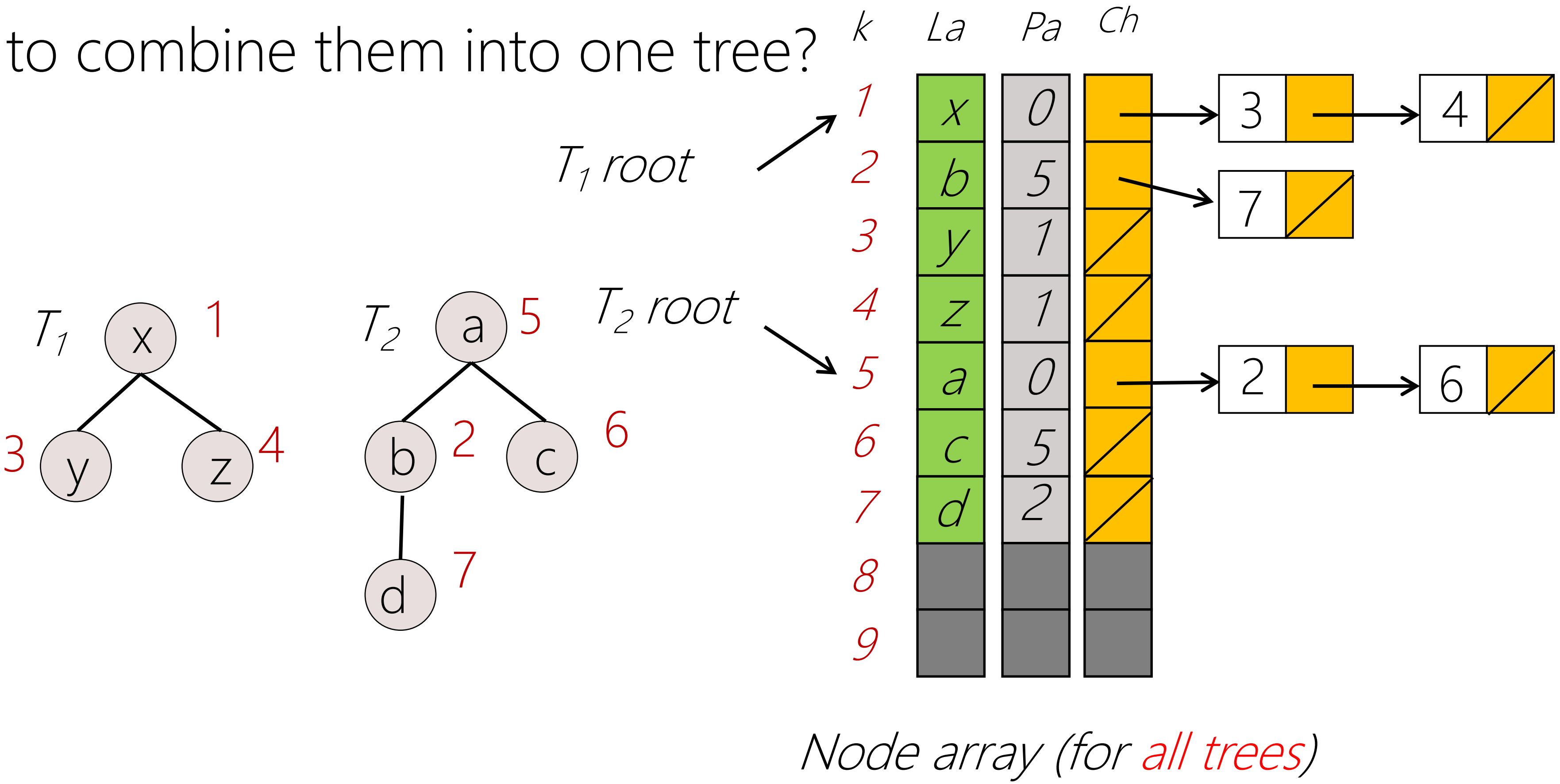
- Similar to the simple approach, we have the tree and the value arrays
- But, we have an additional array
  - Linked list of children (for each non-terminal node: (a,b) in this example)
  - Idea: let's keep both parent and children information!
- Node array (for all tree nodes)





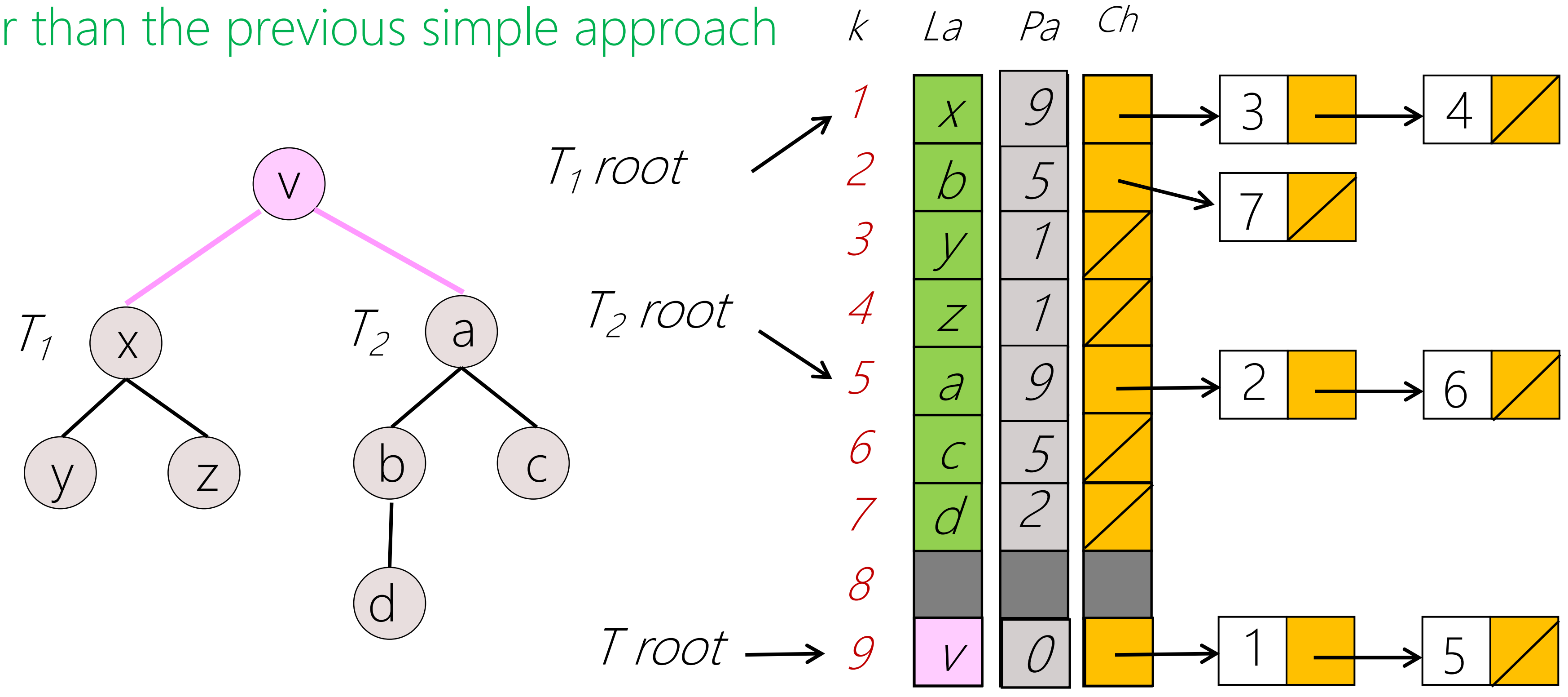
List-of-Children Approach: Combining Trees

- *Create*$_2$*(v, T*$_1$*, T*$_2$*)*
- If we saved two trees T_1 and T_2 *in a single array*
- How to combine them into one tree?



List-of-Children Approach: Combining Trees

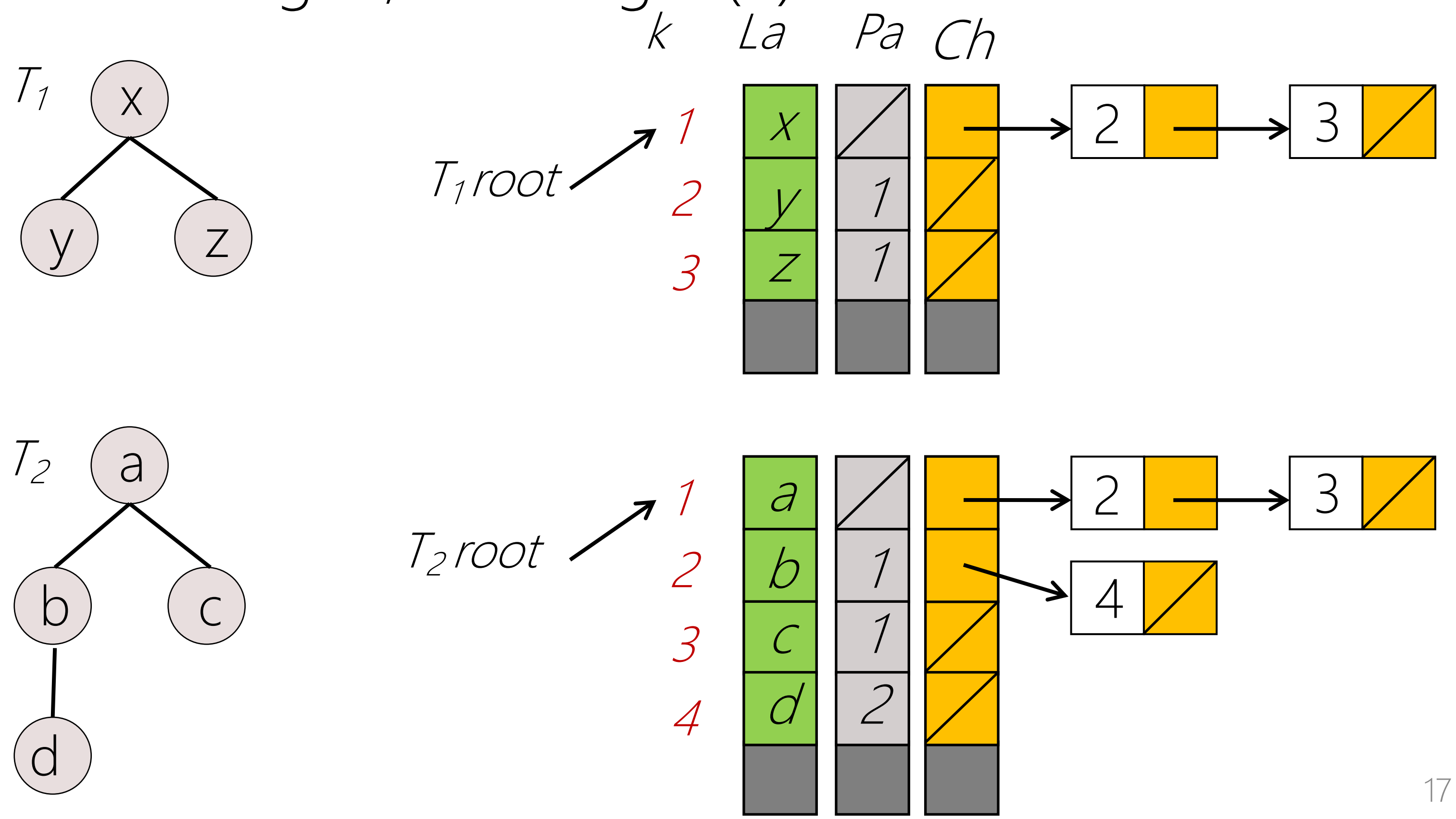
- *Create$_2$(v, T_1, T_2)*
  - Combining trees takes time complexity O(1) (in case of binary tree)
    - Change the parents of the subtrees' roots
    - Easier than the previous simple approach



Node array (for *all trees*)

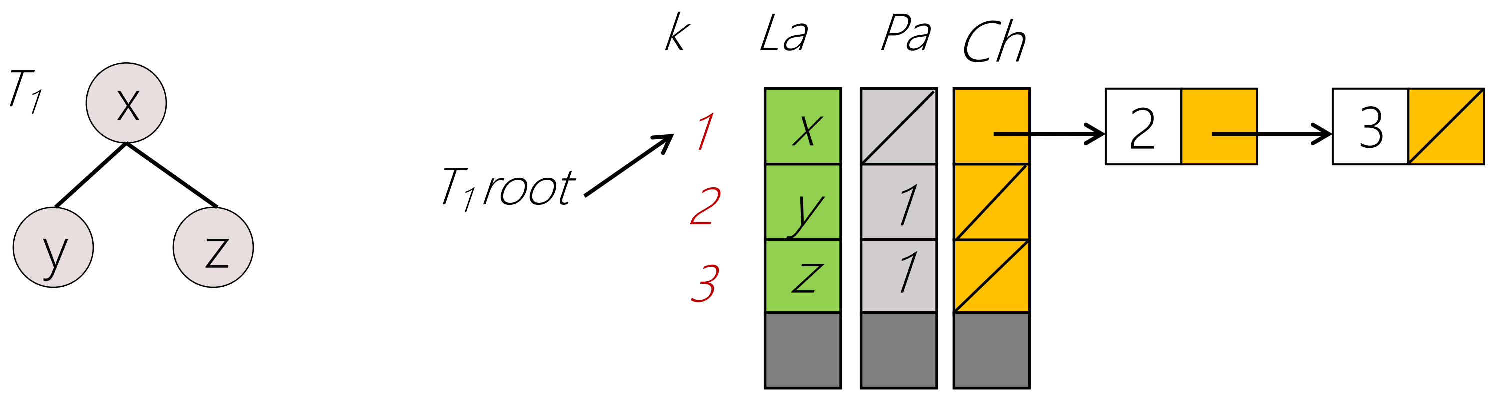
List-of-Children Approach: Combining Trees

- However... If each tree is stored in a separate array
  - Combining trees is OK?: *more cumbersome*
  - If we want to make it as a single array after the operation, again the second subtree's indices should be changed, resulting O(n)...



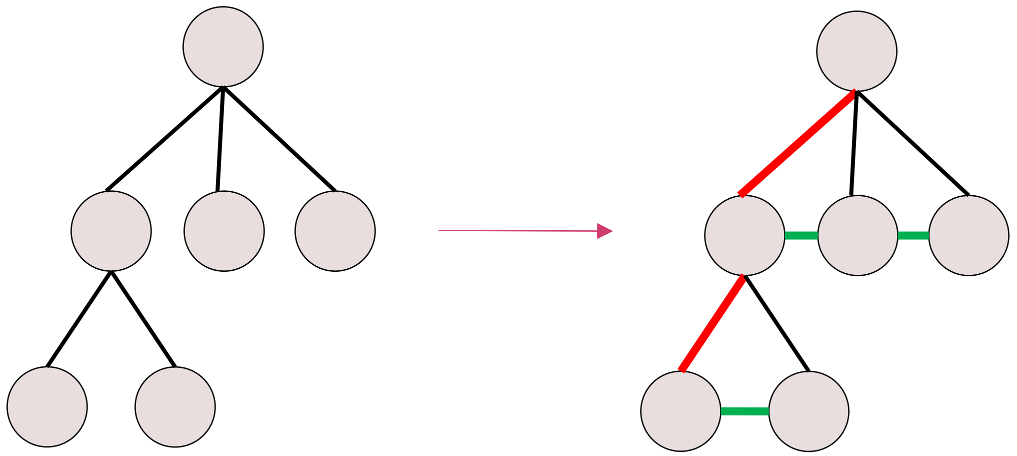
List-of-Children Approach: Problems

- Additional problems
  - Difficult to access a node's right sibling
    - How to access y's sibling?
      - Access parent's children list
  - Duplication of nodes
    - Each node appears both in the *node array* & in the *linked list* of children
      - Node 2's information is in the original list as well as the children list of its parent



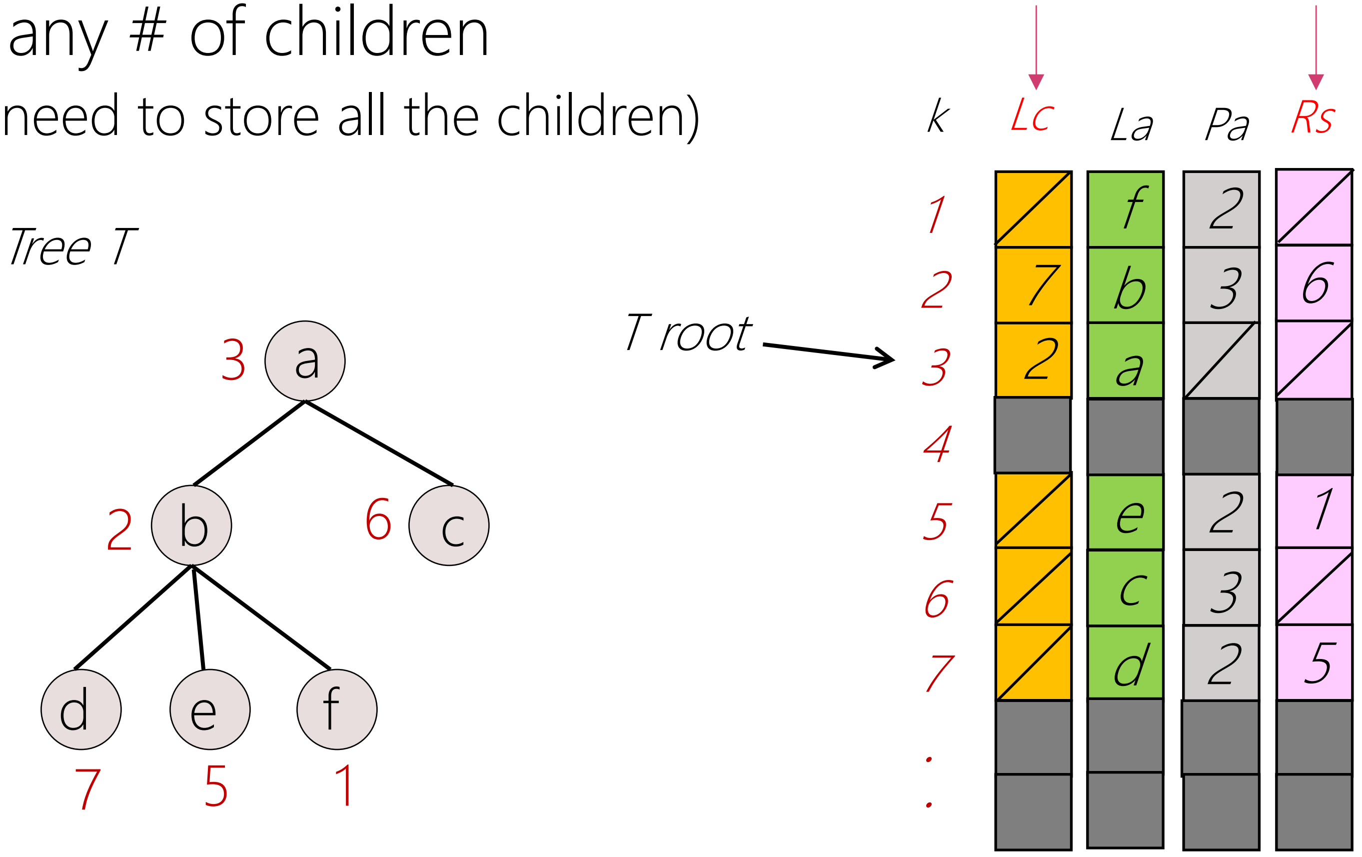
Left-Child/Right-Sibling Approach

- Save left child and right sibling for each node



Left-Child/Right-Sibling Approach: Static

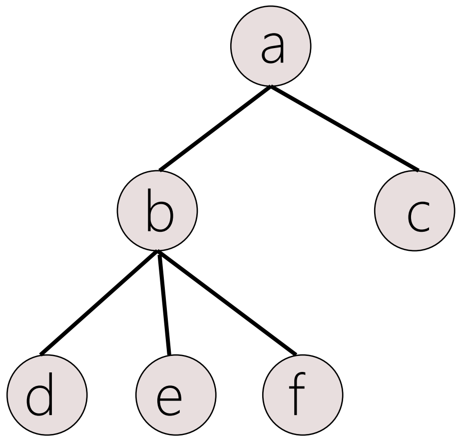
- Static implementation (using array)
  - Each node has pointers to its *parent*, *leftmost child*, & *right sibling*
  - Efficient for any # of children
    - (we don't need to store all the children)

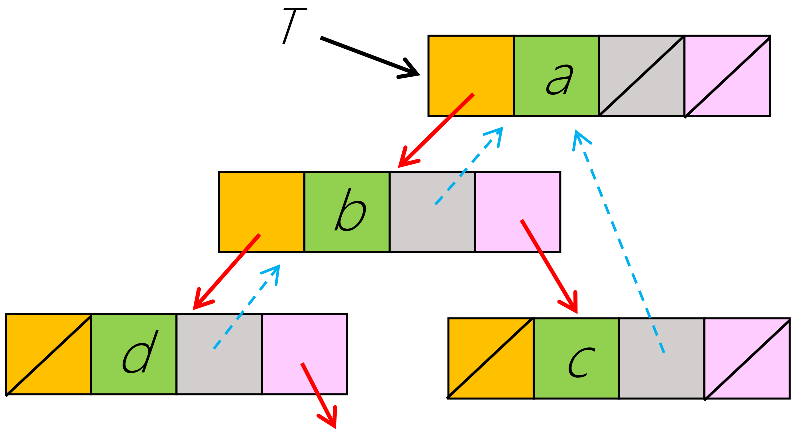


Left-Child/Right-Sibling Approach: Dynamic

- Problem of static *left-child/right-sibling* approach
  - Using an array to store the collection of nodes
  - Fixed space for nodes (no dynamic space allocation)

- Can we use linked implementation – dynamic one?

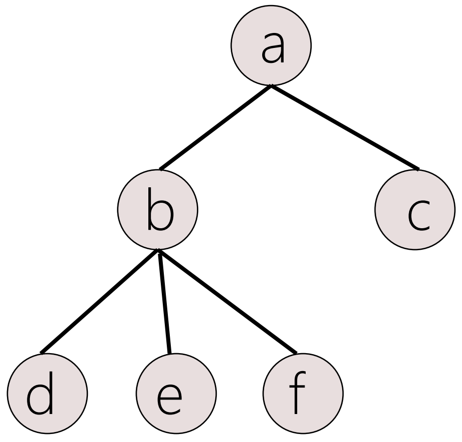


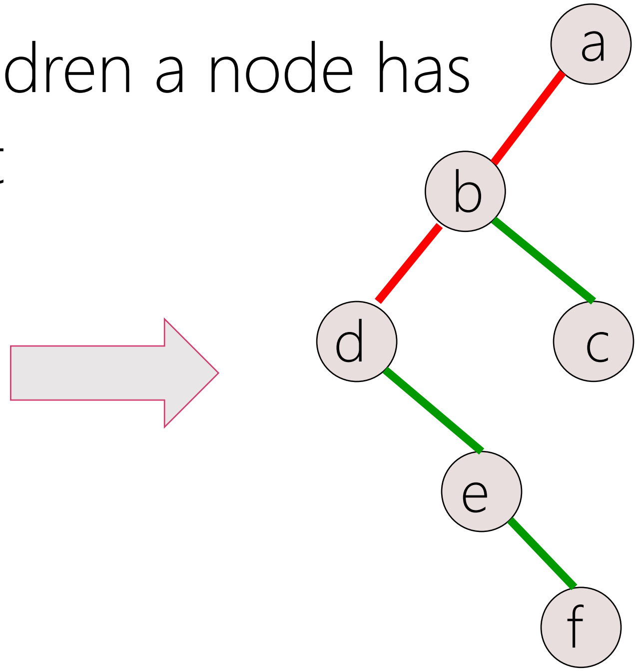


Left-Child/Right-Sibling Approach

- k-ary tree is a tree in which each node has no more than k children

- Any general k-ary tree can be implemented with left-child/right-sibling approach and this allows us to represent a k-ary tree with a binary tree ☺
  - Don't need to know about the number of children a node has
  - Easy to code as we have two children at most





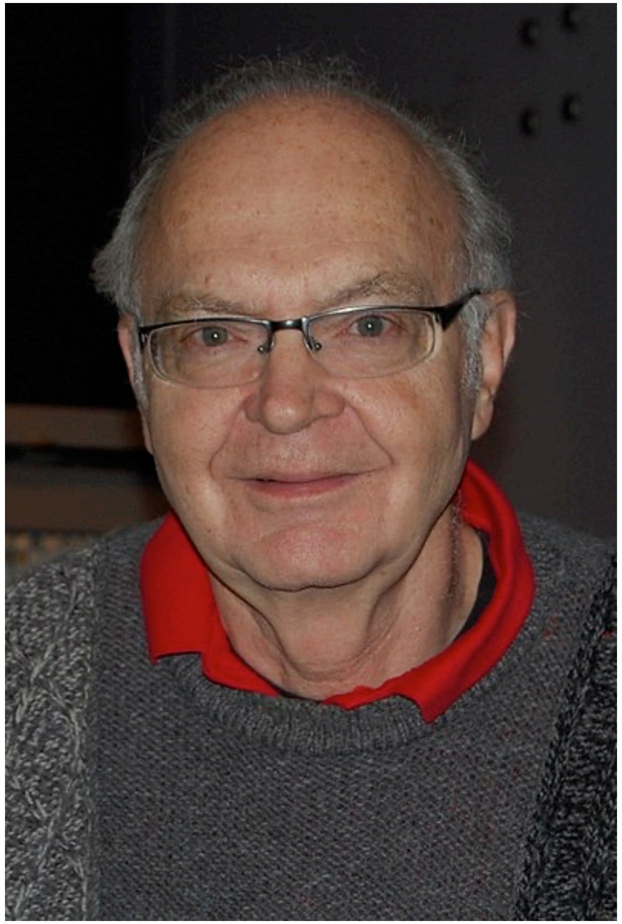
Left-Child/Right-Sibling Approach

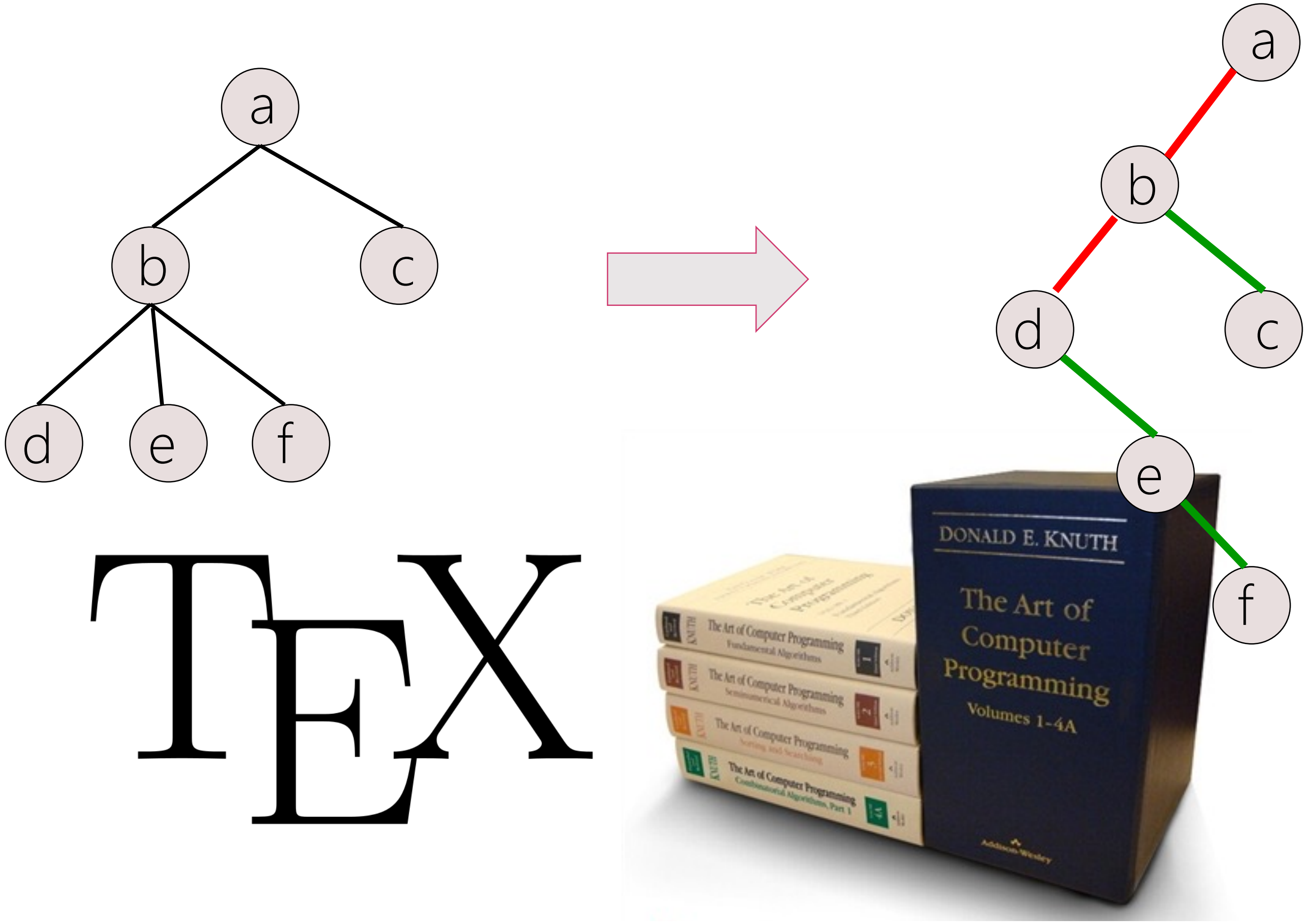
- Step-1: Convert a (forest of) general tree into a binary tree
  - *Leftmost child* → Left child
  - *Right sibling* → Right child (tree roots are assumed to be siblings)
  - Rotate the *left-child/right-sibling* tree clockwise by 45 degrees
- So-called, Knuth transform

- Step-2: Linked implementation of the converted binary tree

Donald Knuth

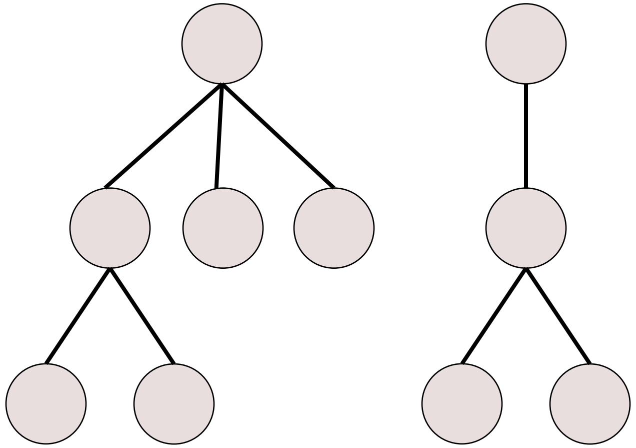
- Author of the multi-volume work *The Art of Computer Programming*
- Analysis of the computational complexity of algorithms & systematized formal mathematical techniques
- creator of the TeX computer typesetting system



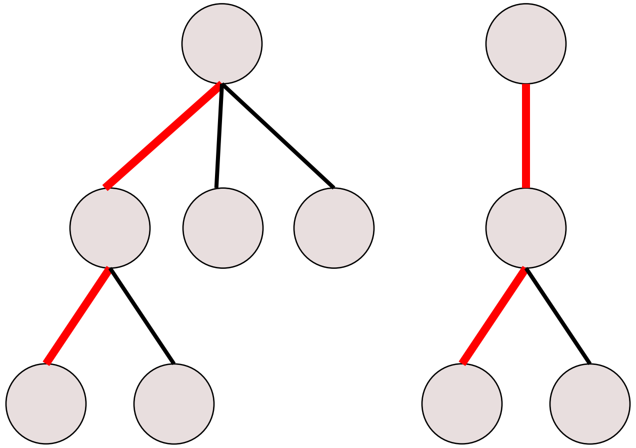


Converting into a Binary Tree (1)

- Input: general trees

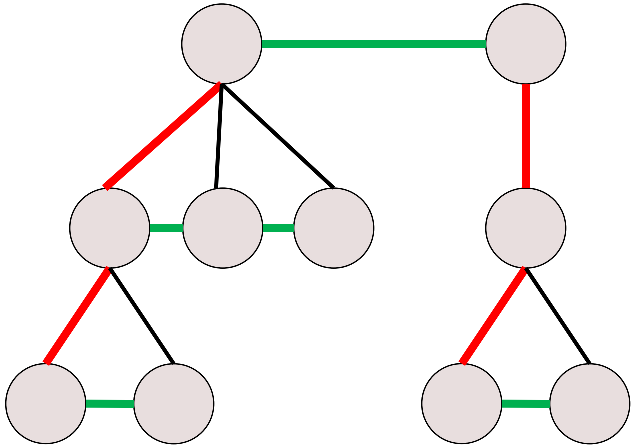


- *Leftmost child* → Left child

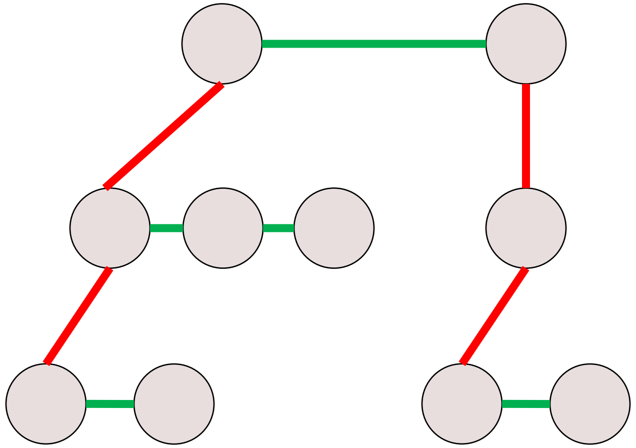


Converting into a Binary Tree (2)

- *Right sibling* → Right child
  - Tree roots are assumed to be siblings

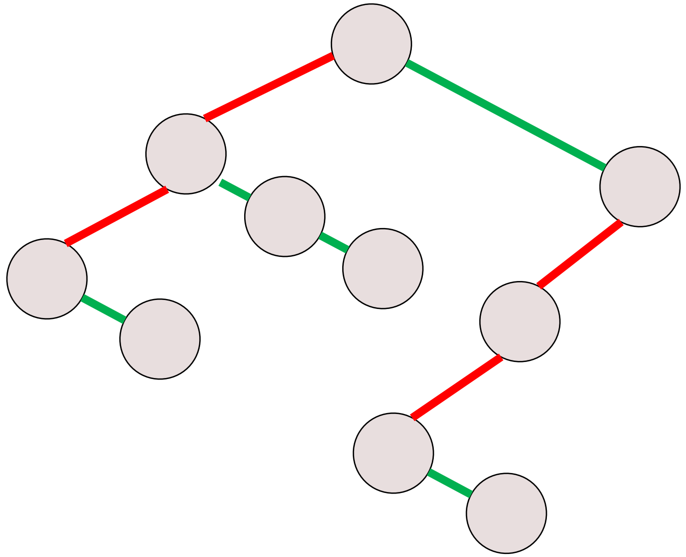


- Remove the other links



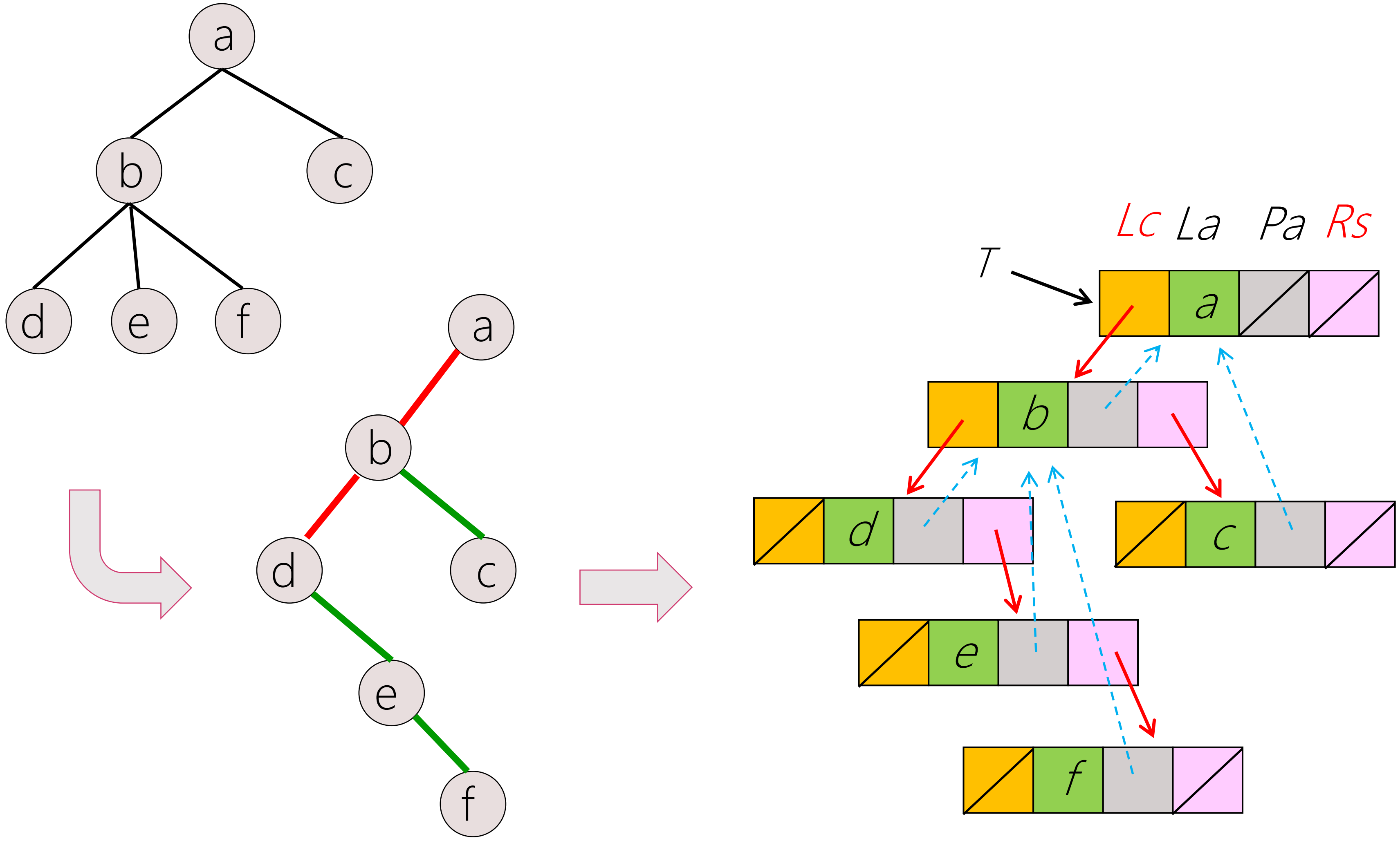
Converting into a Binary Tree (3)

- Rotate clockwise by 45°



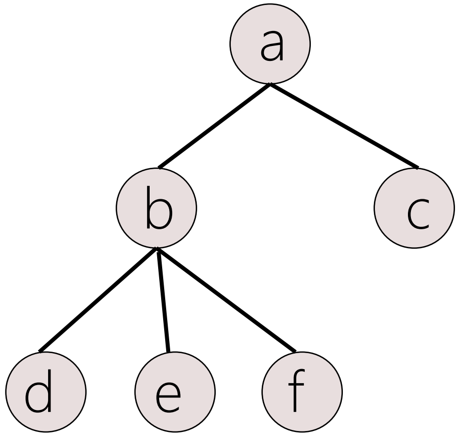
- Now ready for its linked implementation

Left-Child/Right-Sibling Tree: Linked Rep.

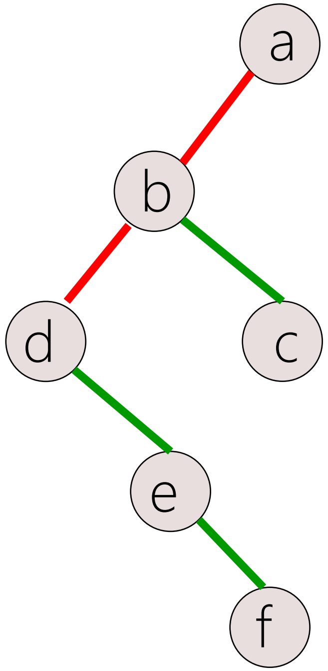


Traversal: Original vs. Converted Binary

- Original general tree T



- Converted binary tree T'

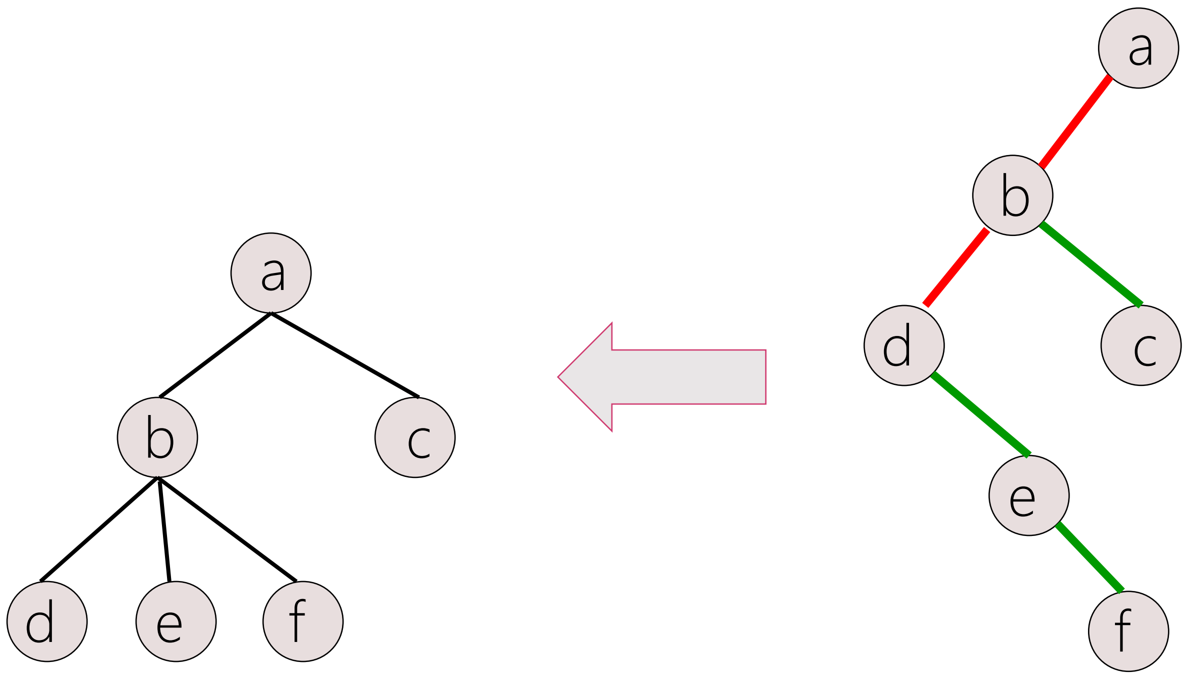


- Preorder = a b d e f c
- Inorder = d b e f a c
  d e b f a c
- Postorder = d e f b c a

- Preorder = a b d e f c
- Inorder = d e f b c a
- Postorder = f e d c b a

Question

- How to convert a converted binary tree to a general tree?



References

- Further reading list and references
  - https://en.wikipedia.org/wiki/Donald_Knuth
  - https://en.wikipedia.org/wiki/Tree_(data_structure)
  - https://en.wikipedia.org/wiki/TeX
  - https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming
  - https://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree
  - https://www.geeksforgeeks.org/left-child-right-sibling-representation-tree/

- Slide credit
  - Jaesik Park
  - Seung-Hwan Baek
  - Jong-Hyeok Lee