



香港中文大學 (深圳)
The Chinese University of Hong Kong

CSC3100 Data Structures

Lecture 13: Tree and binary tree

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen



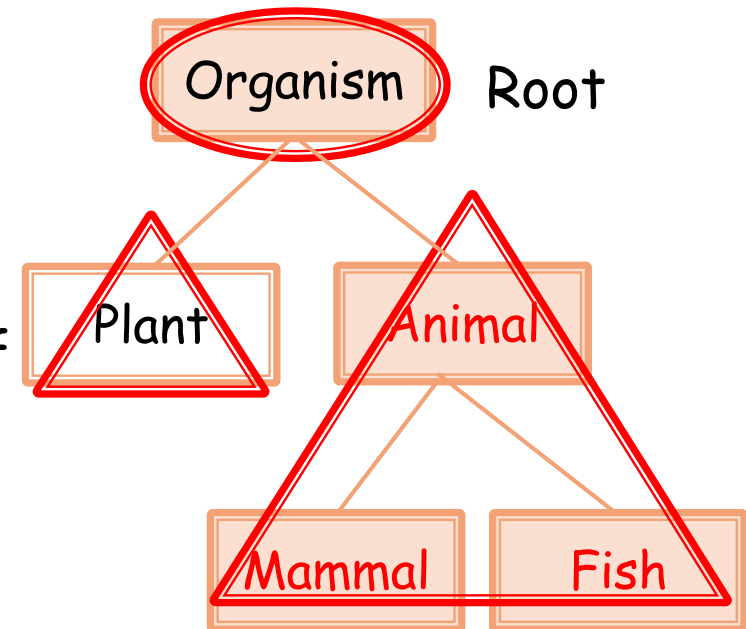
Outline

- ▶ In this lecture, we will learn
 - Basic concept of trees
 - Binary tree ADT
 - Traversal of binary trees
 - Reconstruction of binary trees



Tree definition

- ▶ A tree is a finite set of one or more nodes such that
 - Each node stores an element
 - There is a specially node called the **root**
 - The remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, \dots, T_n where each of these sets is a tree
 - We call T_1, \dots, T_n the subtrees of the root
- ▶ Note
 - A tree with N nodes has $N-1$ edges
 - Every node in the tree is the root of some subtree (**recursive definition**)





Definitions

▶ Parent

- Node *A* is the parent of node *B* if *B* is the root of the left or right sub-tree of *A*

▶ Left (right) child

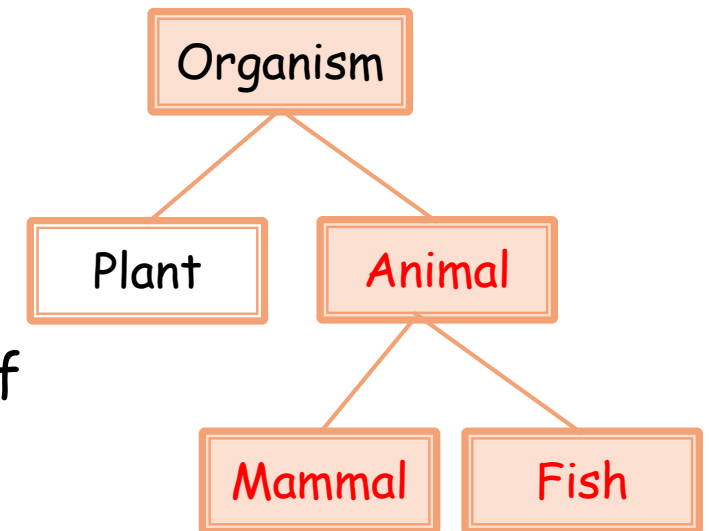
- Node *B* is the left (right) child of node *A* if *A* is the parent of *B*

▶ Sibling

- Node *B* and node *C* are siblings if they have the same parent

▶ Leaf

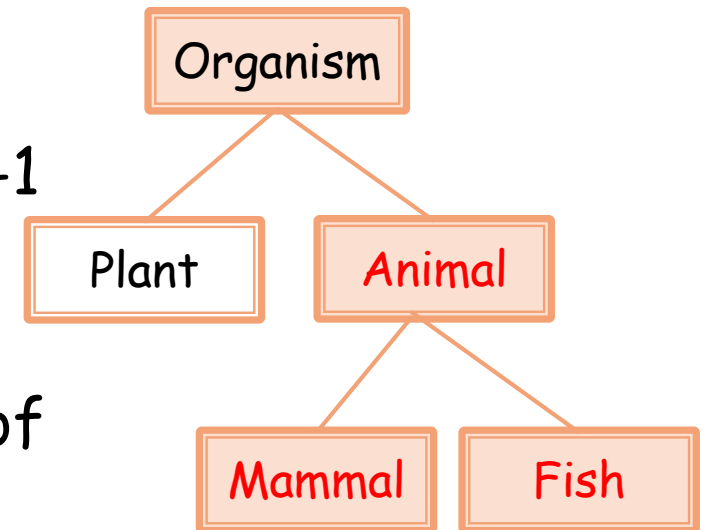
- A node is called a leaf if it has no children





Definitions

- ▶ A path from node n_1 to n_k
 - A sequence of nodes n_1, n_2, \dots, n_k such that n_i is the parent of n_{i+1} for $1 \leq i < k-1$
- ▶ Length of a path
 - The length of this path is the number of edges on the path, namely $k-1$
 - Notice that in a tree, there is exactly only one path from the root to each node





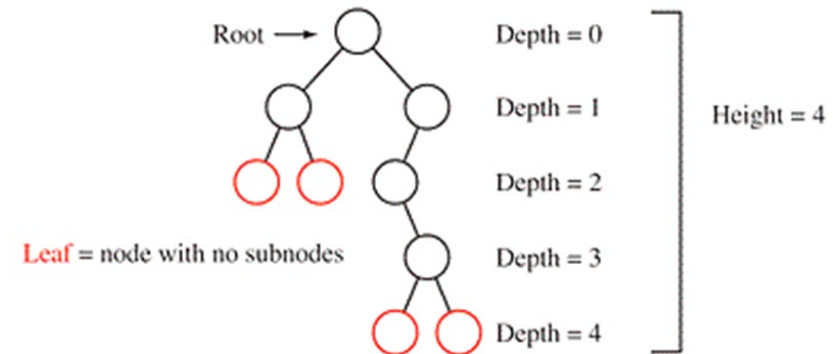
Definitions

▶ Depth of a node

- The depth of a node n_i is the length of the unique path from the root to n_i
- The root is at depth 0

▶ Height of a node

- The height of a node n_i is the length of the longest path from n_i to a leaf
- All leaves are at height 0



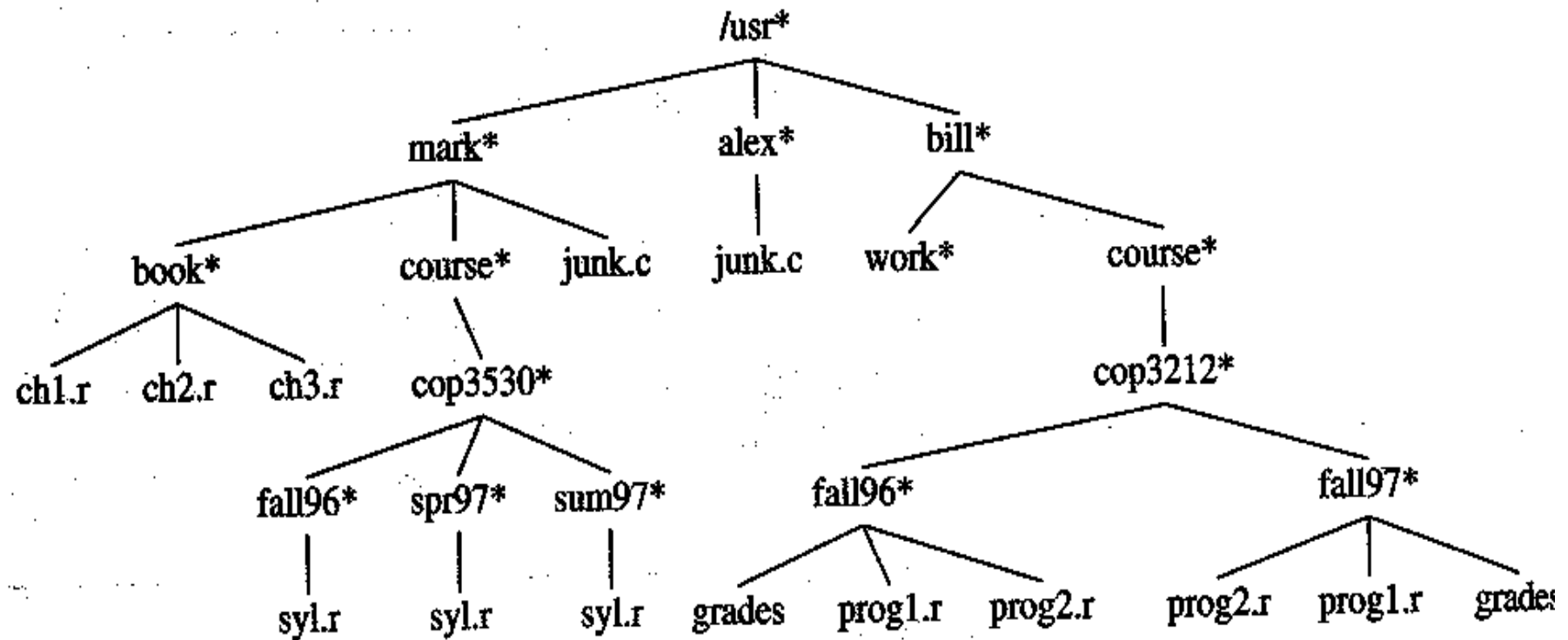
Note 1: The height of a tree is equal to the height of the root

Note 2: The depth of a tree is equal to the depth of the deepest leaf

Note 3: The maximum height is equal to the maximum depth



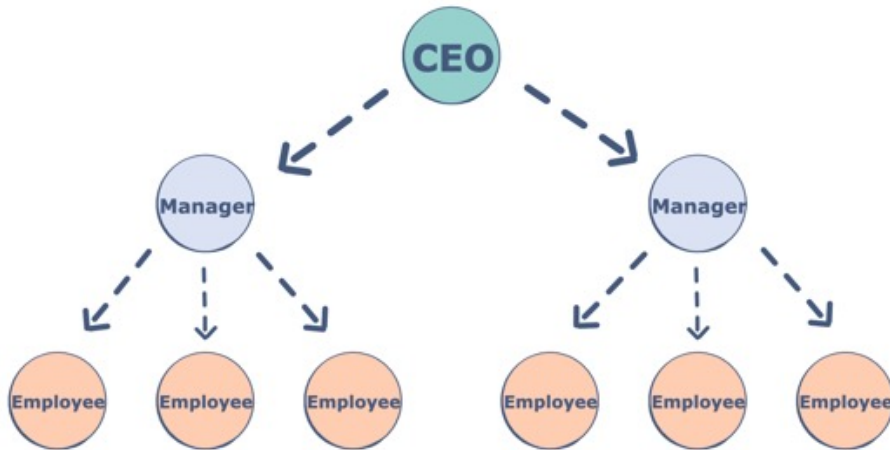
Applications: Unix file system



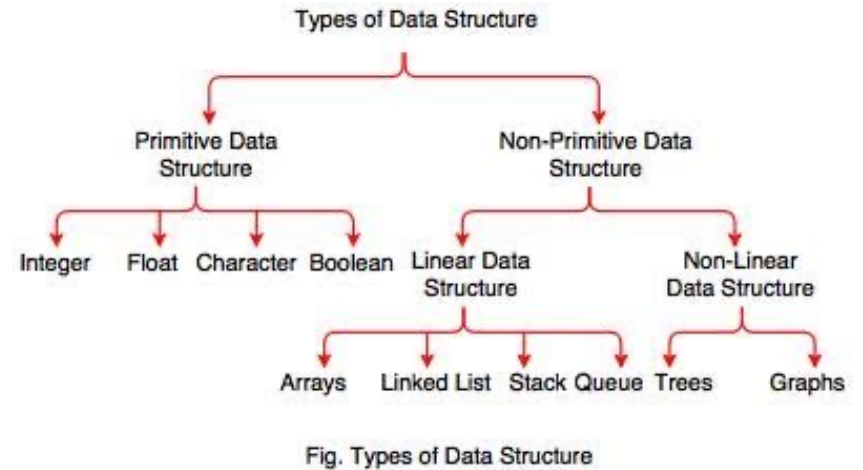


More applications

► HR system



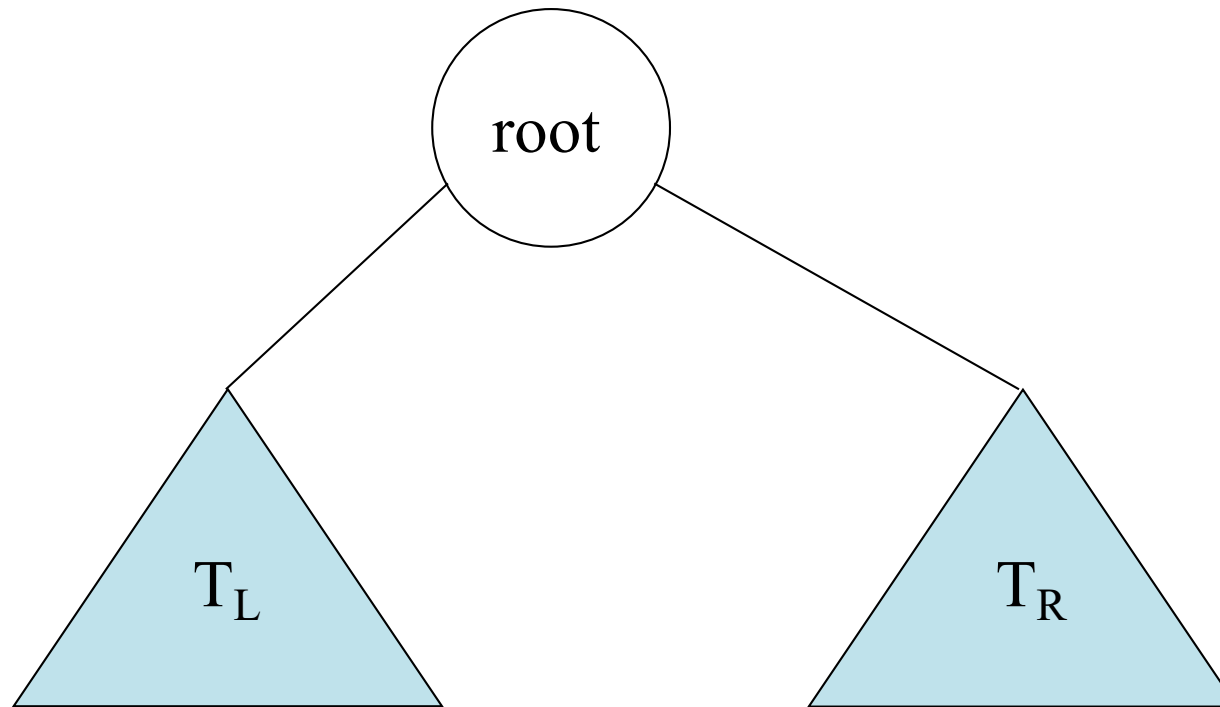
► Java data types





Binary tree

- ▶ A binary tree is a tree, in which
 - No node can have more than two children (subtrees): T_L and T_R , both of which could possibly be empty





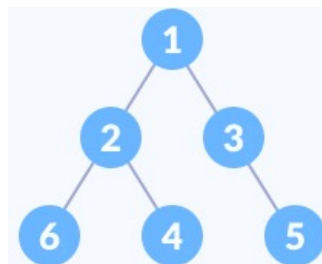
Binary tree

► Full binary tree

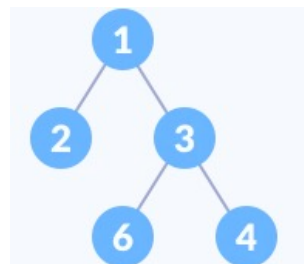
- A binary tree where all the nodes have either **two or no** children

► Complete binary tree

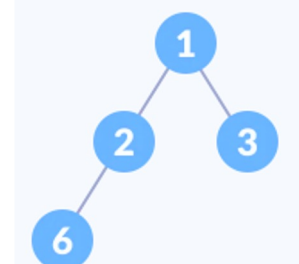
- A binary tree where **all the levels are completely filled** except possibly the lowest one, which is filled from **the left**



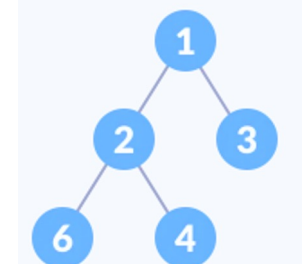
✗ Full Binary Tree
✗ Complete Binary Tree



✓ Full Binary Tree
✓ Complete Binary Tree



✗ Full Binary Tree
✓ Complete Binary Tree



✓ Full Binary Tree
✓ Complete Binary Tree



Binary tree ADT

► Operations:

- **Create(bintree):** creates an empty binary tree
- **Boolean IsEmpty(bintree):** if **bintree** is empty return TRUE else FALSE
- **MakeBT(bintree1,element,bintree2):** return a binary tree whose left subtree is **bintree1** and right subtree is **bintree2**, and whose root node contains the data **element**
- **Lchild(bintree):** if **bintree** is empty return error else return the left subtree of **bintree**
- **Rchild(bintree):** if **bintree** is empty return error else return the right subtree of **bintree**
- **Data(bintree):** if **bintree** is empty return error else return the **element** data stored in the root node of **bintree**



Binary tree design

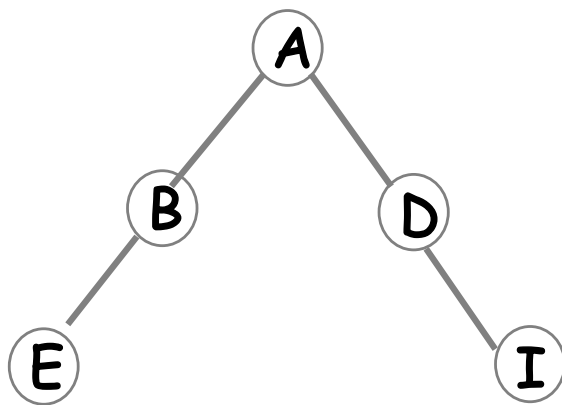
- ▶ Two solutions
 - Using pointers
 - More intuitive solution
 - We will see the pseudo-codes
 - Using array
 - Need more complicated design, and cannot efficiently handle all operations (thus will omit its implementations for each operation)
 - Will be used for heap, a special type of complete binary tree



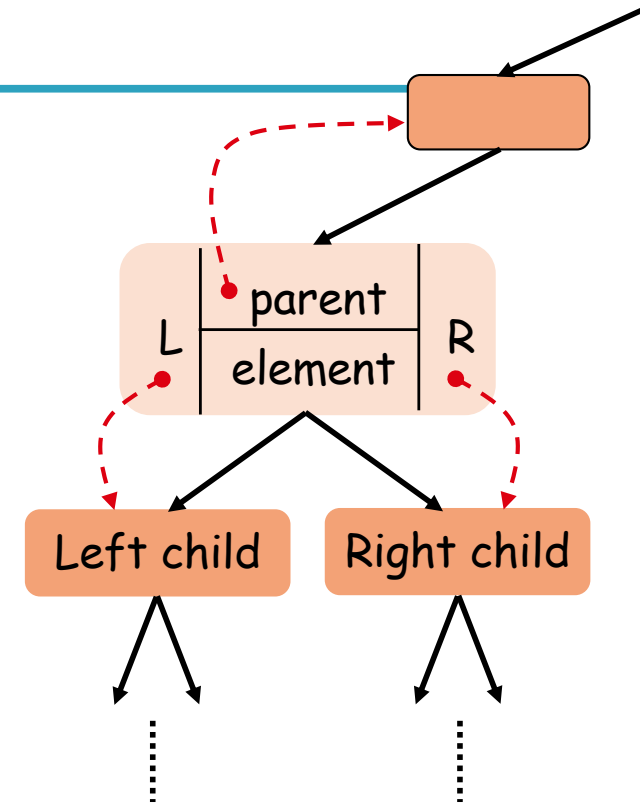
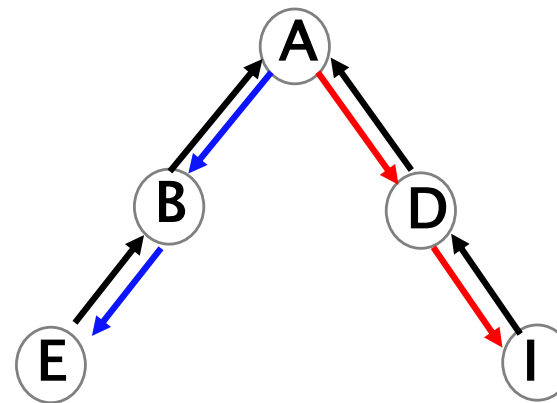
Binary tree design (i)

► A pointer representation

- For each node **node**, we maintain
 - node.parent**: store the address its parent,
 - node.leftchild**: store the address of its left child,
 - node.rightchild**: store the address of its right child
 - node.element**: store the values



→ parent
→ leftchild
→ rightchild
(Omitted links points to NULL)





Binary tree: pointer implementation

- ▶ Create(bintree)

Algorithm: *create(bintree)*

```
1 bintree = NULL
```

- ▶ isEmpty(bintree)

Algorithm: *isEmpty(bintree)*

```
1 return bintree == NULL
```

- ▶ MakeBT(bintree1, element, bintree2)

Algorithm: *MakeBT(bintree1, element, bintree2)*

```
1 rootNode <- allocate new memory
2 rootNode.element = element
3 rootNode.parent = NULL
4 rootNode.leftchild = bintree1
5 rootNode.rightchild = bintree2
6 if bintree1 != NULL
7     bintree1.parent = rootNode
8 if bintree2 != NULL
9     bintree2.parent = rootNode
10 return rootNode
```



Binary tree: pointer implementation

▶ Lchild(bintree)

Algorithm: *Lchild(bintree)*

```
1 if bintree == NULL
2   error "empty tree"
3 return bintree.leftchild
```

▶ Rchild(bintree)

Algorithm: *Rchild(bintree)*

```
1 if bintree == NULL
2   error "empty tree"
3 return bintree.rightchild
```

▶ Data(bintree)

Algorithm: *Data(bintree)*

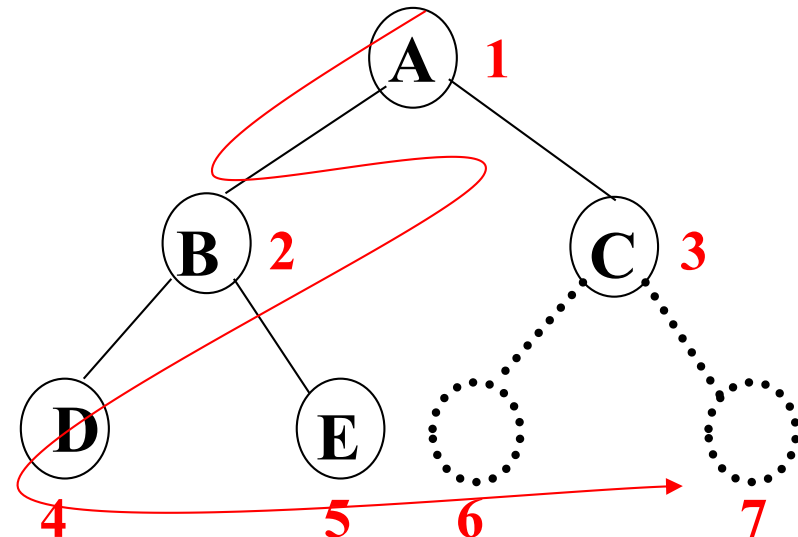
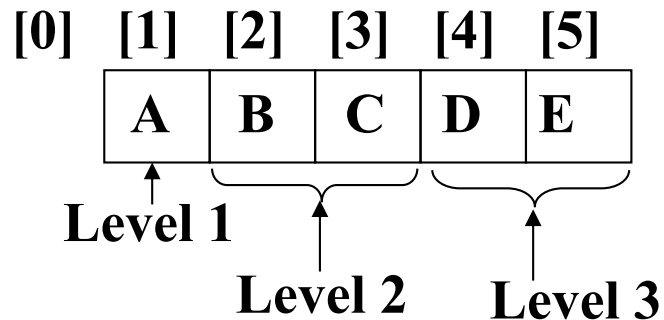
```
1 if bintree == NULL
2   error "empty tree"
3 return bintree.element
```



Binary tree design (ii)

► An array representation

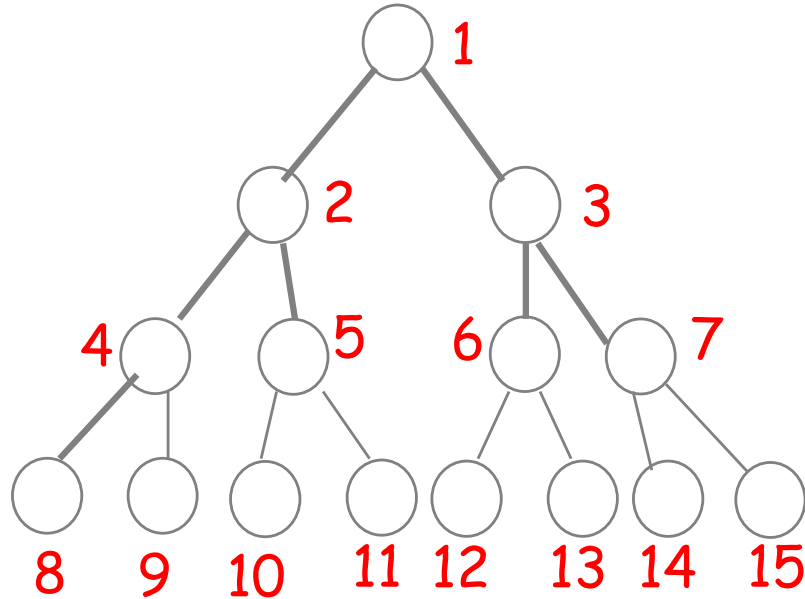
- Given a complete binary tree with n nodes, for any i -th node, $1 \leq i \leq n$,
 - $\text{parent}(i)$ is $\lfloor i/2 \rfloor$
 - $\text{leftChild}(i)$ is at $2i$ if $2i \leq n$; otherwise, i has no left child
 - $\text{rightChild}(i)$ is at $2i + 1$ if $2i + 1 \leq n$; otherwise, i has no right child



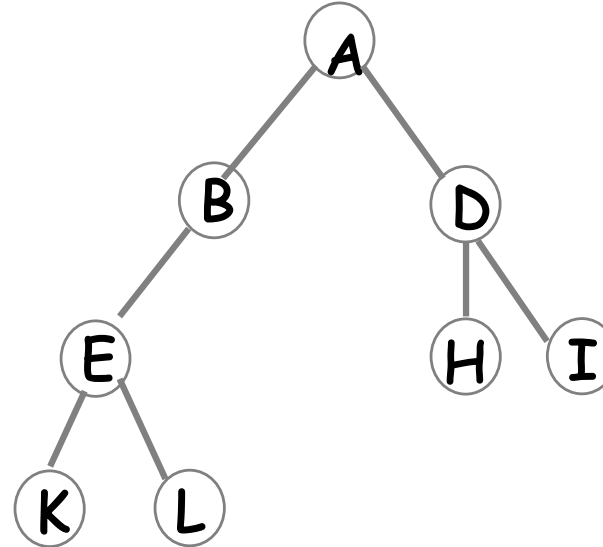


Binary tree design (ii)

- ▶ An array representation
 - Generalize to all binary trees
 - Efficient for complete binary trees
 - But inefficient for skewed binary trees
 - Inefficient to implement the ADT



full binary tree

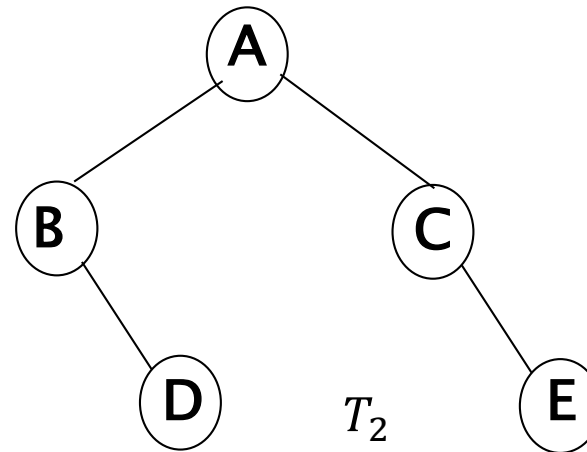
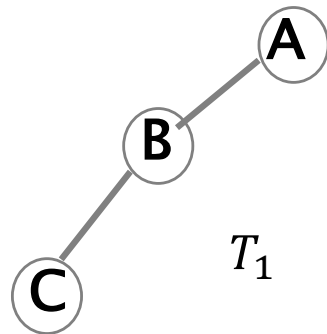


1	A	}	Level 1
2	B		
3	D	}	Level 2
4	E		
5		}	Level 3
6	H		
7	I		
8	K		
9	L	}	Level 4
10			
11			
12			
13			
14			
15			



Practice

- ▶ What are the array representation of the following binary trees?
 - Show the content in the array
 - Hint: first obtain the ID for each node



arr

[1]	[2]	[3]	[4]	[5]	[6]	[7]



Traversing strategies

- ▶ Preorder (depth-first)
 - Visit the node
 - Traverse the left subtree in preorder
 - Traverse the right subtree in preorder

- ▶ Inorder
 - Traverse the left subtree in inorder
 - Visit the node
 - Traverse the right subtree in inorder

- ▶ Postorder
 - Traverse the left subtree in postorder
 - Traverse the right subtree in postorder
 - Visit the node



Traversing binary tree

When the binary tree is empty, it is "traversed" by doing nothing, otherwise:

preorder traversal

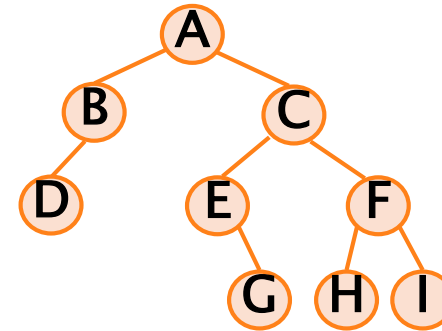
Visit the root

Traverse the left subtree

Traverse the right subtree

A B D C E G F H I

Example:



Result:

= A (A's left) (A's right)
= A B (B's left) (B's right = NULL) (A's right)
= A B (B's left) (A's right)
= A B D (D's left=NULL) (D's right = NULL) (A's right)
= A B D (A's right)
= A B D C (C's left) (C's right)
= A B D C E (E's left=NULL) (E's right) (C's right)
= A B D C E (E's right) (C's right)
= A B D C E G (G's left=NULL) (G's right = NULL) (C's right)
= A B D C E G (C's right)
= A B D C E G F (F's left) (F's right)
= A B D C E G F H (H's left=NULL) (H's right = NULL) (F's right)
= A B D C E G F H I (I's left=NULL) (I's right = NULL)
= A B D C E G F H I



Exercise

$$(A+B)/(C*D)-E*(F-G)+H$$

Preorder:

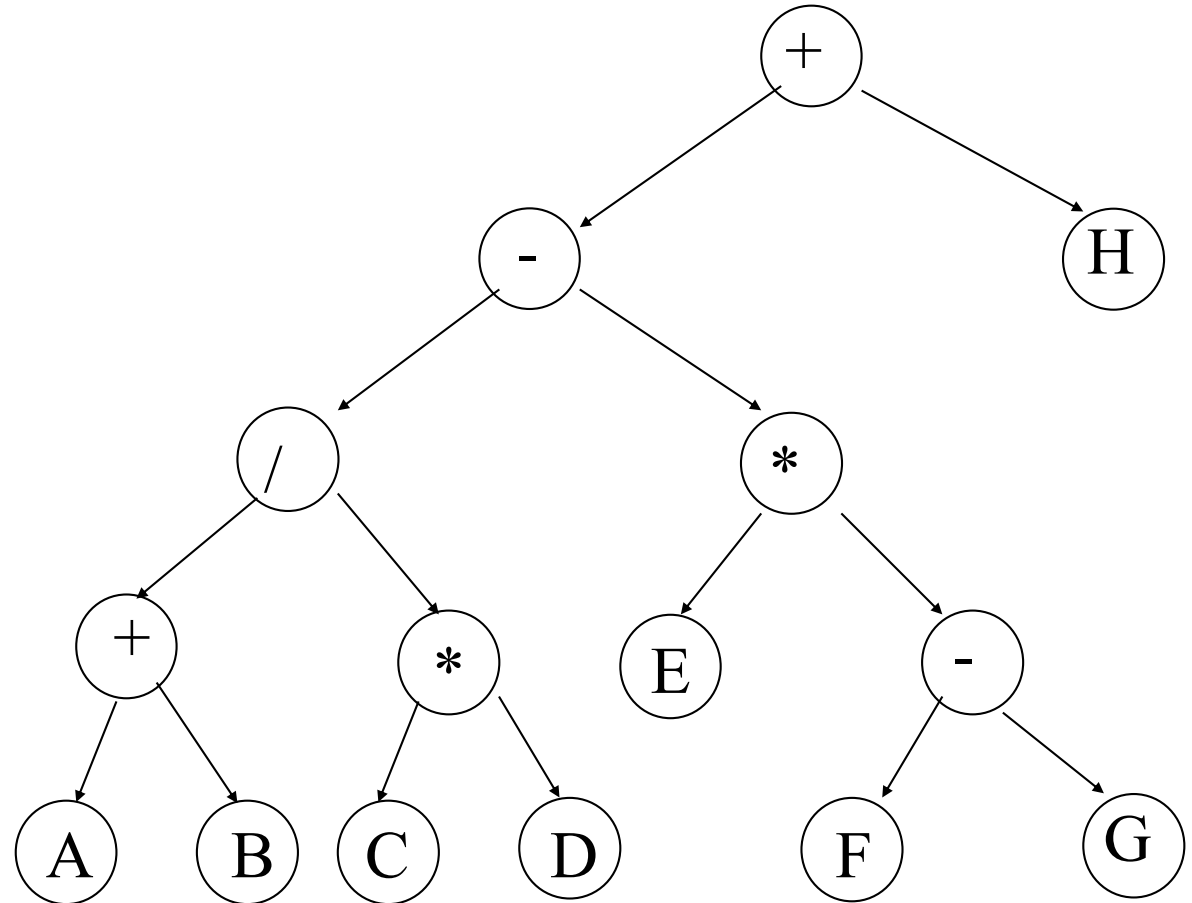
$+ - / + A B * C D * E - F G H$

Inorder :

$A + B / C * D - E * F - G + H$

Postorder:

$A B + C D * / E F G - * - H +$



Given an expression, what is the relationship between its postfix and postorder?

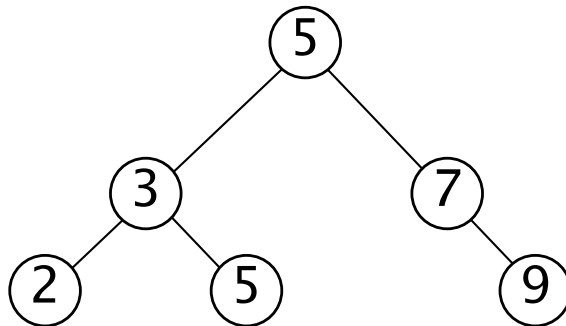


Implementation

INORDER-TREE-WALK(x)

1. if $x \neq \text{NIL}$
2. then INORDER-TREE-WALK (left [x])
3. print key [x]
4. INORDER-TREE-WALK (right [x])

E.g.:



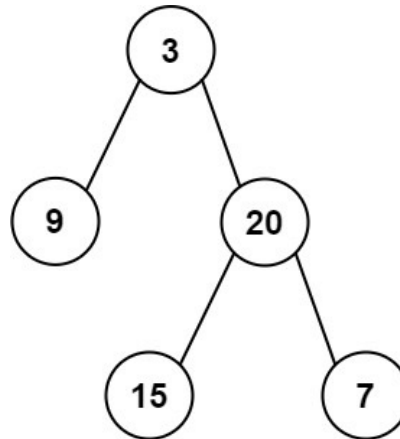
Output: 2 3 5 5 7 9

- ▶ Running time:
 - $\Theta(n)$, where n is the size of the tree rooted at x



Exercise

- ▶ Given a binary tree, show its preorder, inorder, and postorder



- preorder=[3, 9, 20, 15, 7]
- inorder=[9, 3, 15, 20, 7]
- postorder=[9, 15, 7, 20, 3]



Binary tree reconstruction

Reconstruction of a binary tree from its preorder and inorder sequences

Example: Given the following sequences, find the corresponding binary tree:

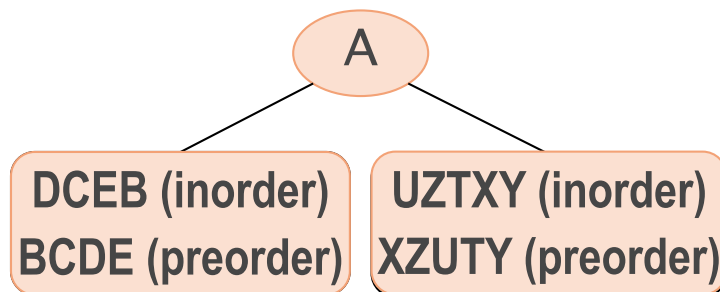
inorder : DCEBAUZTXY

preorder : ABCDEXZUTY

Looking at the whole tree:

- ▶ "preorder : ABCDEXZUTY"
==> A is the root
- ▶ Then, "inorder : DCEBAUZTXY"

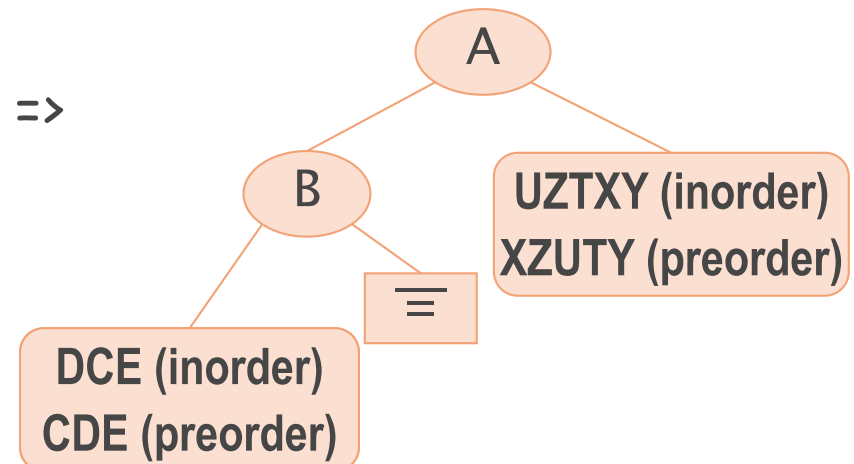
==>



Looking at the left subtree of A:

- "preorder : BCDE"
==> B is the root
- Then, "inorder: DCEBU"

=>





Binary tree reconstruction

Reconstruction of a binary tree from its preorder and inorder sequences

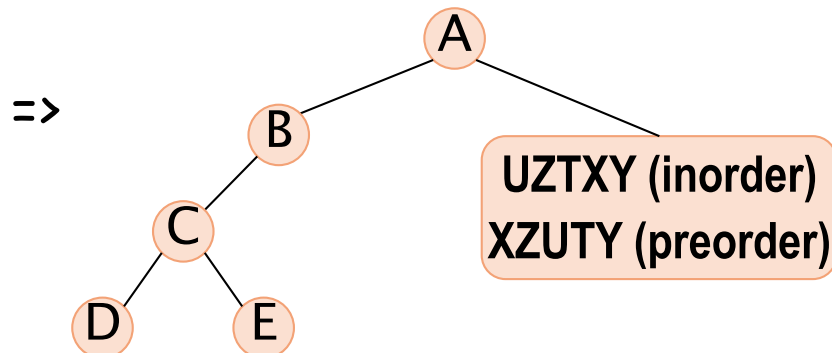
Example: Given the following sequences, find the corresponding binary tree:

inorder : DCEBAUZTX~~Y~~

preorder : ABCDEXZUT~~Y~~

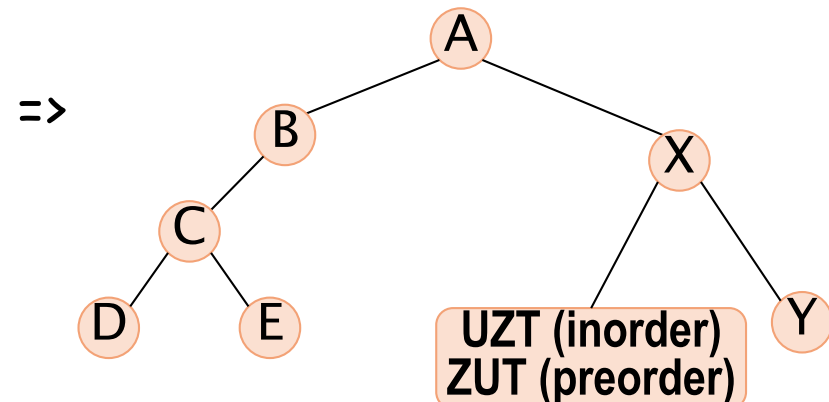
Looking at the left subtree of B:

- "preorder : CDE"
==> C is the root
- Then, "inorder: DCE"



Looking at the right subtree of A:

- "preorder : XZUTY"
==> X is the root
- Then, "inorder: UZTXY"





Binary tree reconstruction

Reconstruction of
a binary tree from
its preorder and
inorder sequences

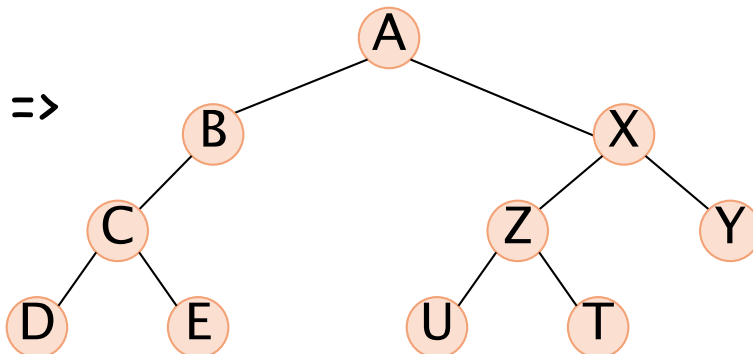
Example: Given the following sequences, find
the corresponding binary tree:

inorder : DCEBAUZTX~~Y~~

preorder : ABCDEXZUT~~Y~~

Looking at the left subtree of X:

- "preorder : ZUT"
==> Z is the root
- Then, "inorder: UZT"





Binary tree reconstruction

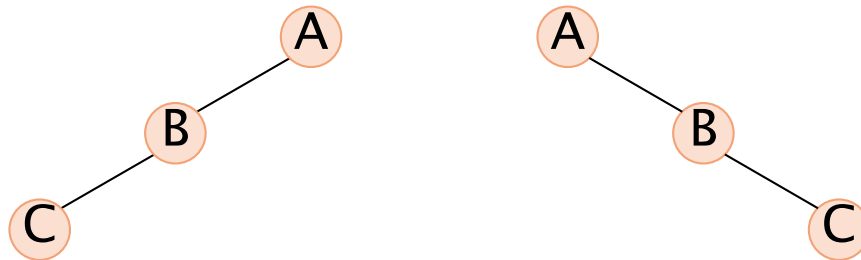
But: A binary tree may not be uniquely defined by its preorder and postorder sequences

Example:

Preorder sequence: ABC

Postorder sequence: CBA

We can construct 2 different binary trees:





Java implementation codes

```
1. class BTreeBuilder {
2.     private static int find(char[] array, int v){
3.         for (int i = 0; i < array.length; i++){
4.             if (array[i] == v) return i;
5.         }
6.         return -1;
7.     }
8.
9.     public TreeNode build (char[] preorder, char[] inorder) {
10.         if (preorder.length == 0) return null;
11.
12.         //step 1: find the root key and create a root node
13.         char rootValue = preorder[0];
14.         TreeNode root = new TreeNode(rootValue);
15.
16.         //step 2: find the index of root key in in-order
17.         int leftSize = find(inorder, rootValue);
18.
19.         //step 3: build left and right sub-tree's
20.         char[] leftPreorder = Arrays.copyOfRange(preorder, 1, 1+leftSize);
21.         char[] leftInorder = Arrays.copyOfRange(inorder, 0, leftSize);
22.         root.left = build (leftPreorder, leftInorder);
23.
24.         char[] rightPreorder = Arrays.copyOfRange(preorder, 1+leftSize, preorder.length);
25.         char[] rightInorder = Arrays.copyOfRange(inorder, leftSize+1, preorder.length);
26.         root.right = build(rightPreorder, rightInorder);
27.
28.         return root;
29.     }
30. }
```

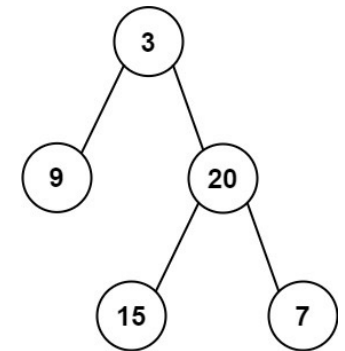
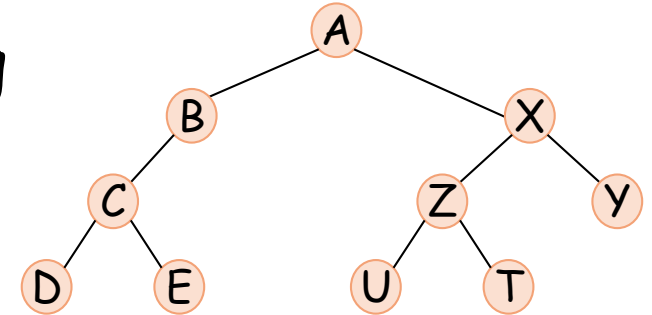
```
1. class TreeNode {
2.     int val;
3.     TreeNode left, right;
4.
5.     TreeNode(int x) {
6.         val = x;
7.     }
8.
9.     public class Test{
10.         public static void main(String[] args) {
11.             char[] preOrder =
12.                 new char[]{'A','B','C','D','E'};
13.             char[] inOrder =
14.                 new char[]{'C','D','B','A','E'};
15.
16.             BTreeBuilder b = new BTreeBuilder();
17.             TreeNode root = b.build(preOrder, inOrder);
18.             System.out.println("build successfully");
19.         }
20.     }
21. }
```

What's the time complexity?



Exercises

- ▶ Show the results of traversing using preorder, inorder, and postorder respectively
- ▶ Construct a binary tree such that
 - preorder=[3,9,20,15,7]
 - inorder=[9,3,15,20,7]
- ▶ Is it possible to reconstruct a binary tree from its inorder and postorder?
 - If yes, how to do it? if no, why?





Recommended reading

- ▶ Reading this week
 - Chapter 12, textbook
- ▶ Next lecture
 - Binary search trees: chapter 12