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

CSC3100 Data Structures Lecture 8: Tree, binary tree, binary search tree

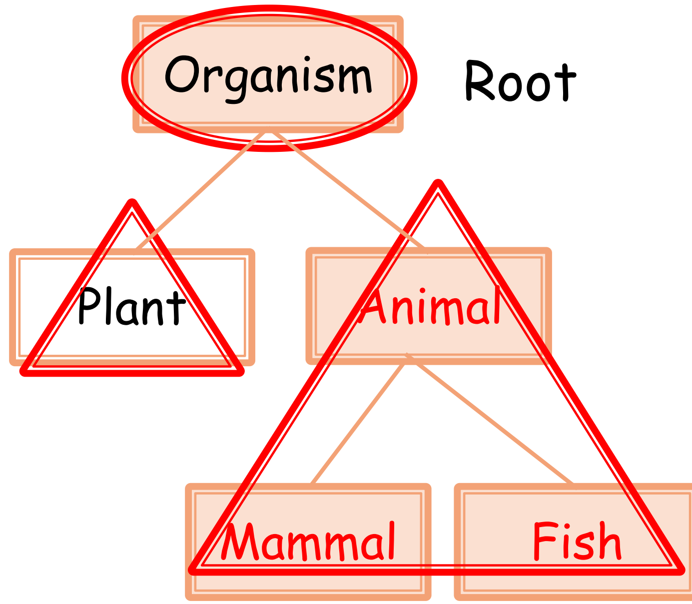
Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Outline

- In this lecture, we will learn
  - Basic concept of trees
  - Binary tree
  - Binary search tree - search in O(logN) average time

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



  - A tree with *N* nodes has one root, and *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

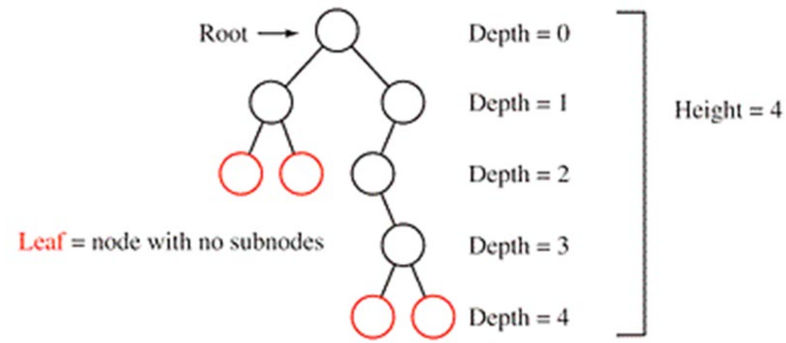
Definitions

- A path from node n_1 to n_k
  - A sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$
- 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 one path from the root to each node

Definitions

- 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

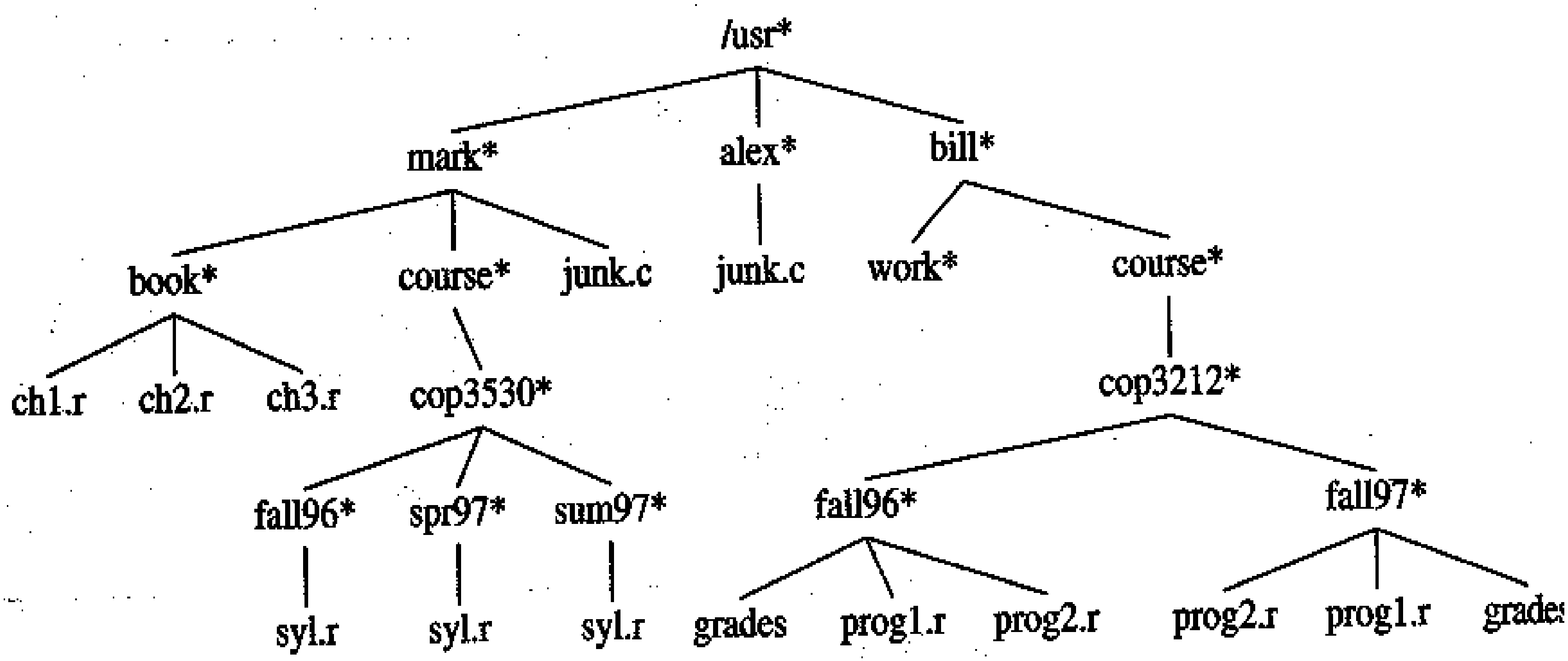
- Height of a node n_i
  - The height of n_i is the length of the longest path from n_i to a leaf
  - All leaves are at height 0



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

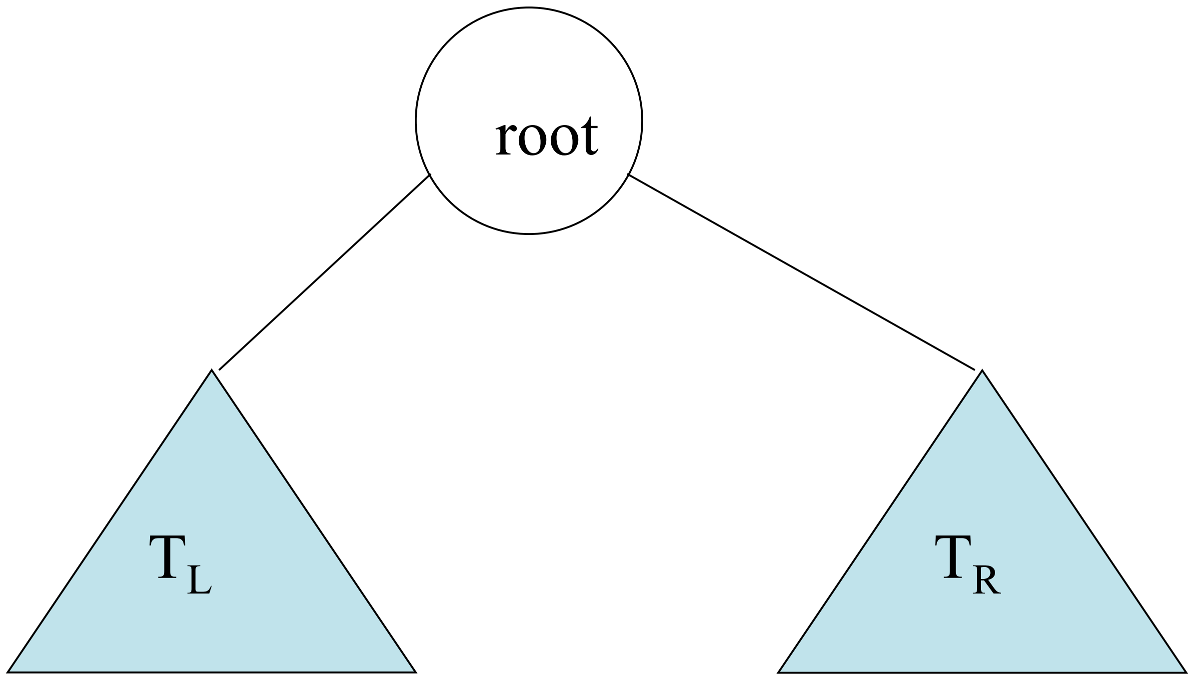
Note 2: The depth of a tree = the depth of the deepest leaf.

Applications – Unix file system



Binary tree

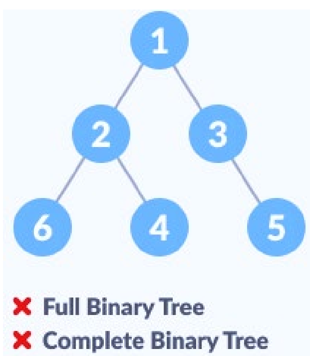
- A binary tree is a tree
  - in which no node can have more than two children (subtrees): T_L and T_R, both of which could possibly be empty

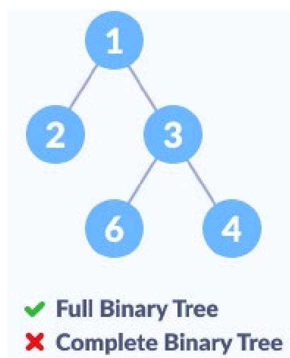


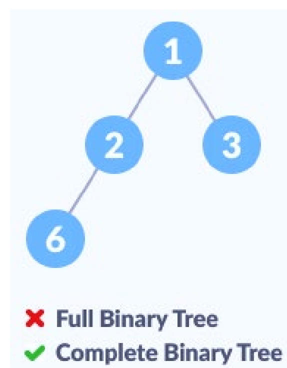
Binary tree

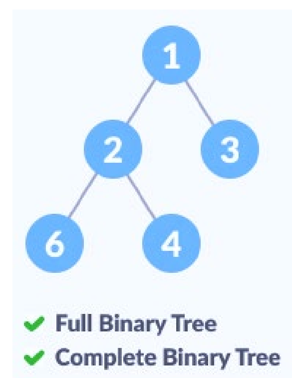
- Full binary tree
  - A binary tree where all the nodes have either two or no children

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









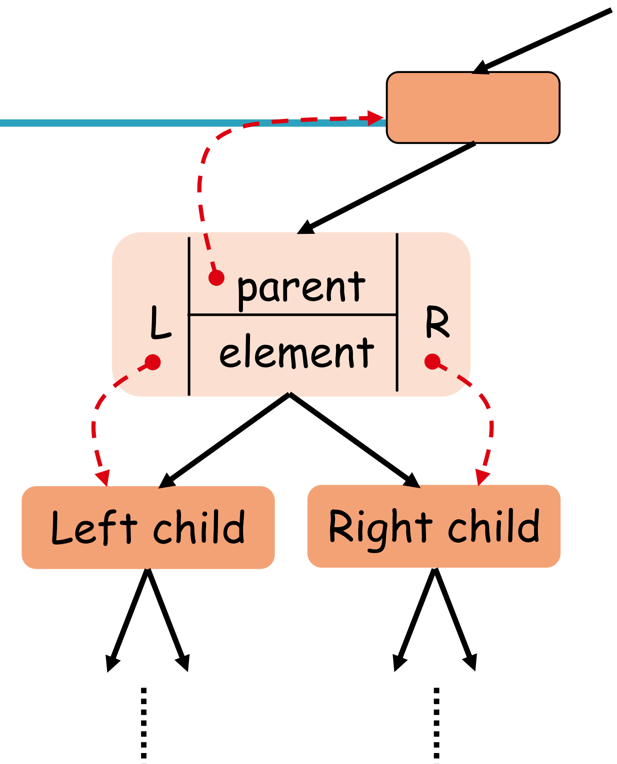
Binary Tree ADT

- 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

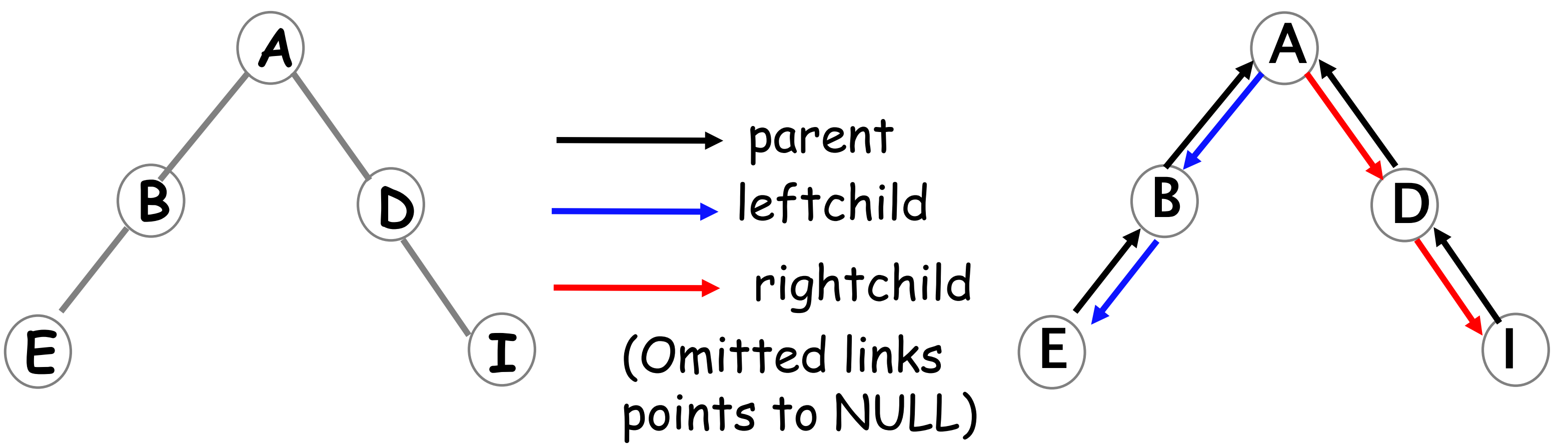
Binary Tree Design

- Two solutions
  - Using pointers
    - More intuitive solution
    - We will see the pseudo-code
  - 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



- Using pointers
  - 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



Binary Tree: Pointer Implementation

- Create(bintree)

Algorithm: ***create(bintree)***

```
bintree = NULL
```

- isEmpty(bintree)

Algorithm: ***isEmpty(bintree)***

```
return bintree == NULL
```

- MakeBT(bintree1,element,bintree2)

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

```
rootNode <- allocate new memory
rootNode.element = element
rootNode.parent = NULL
rootNode.leftchild = bintree1
rootNode.rightchild = bintree2
if bintree1 != NULL
  bintree1.parent = rootNode
if bintree 2 != NULL
  bintree2.parent = rootNode
return  rootNode
```

Binary Tree: Pointer Implementation

- Lchild(bintree)

Algorithm: *Lchild(bintree)*

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

- Rchild(bintree)

Algorithm: *Lchild(bintree)*

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

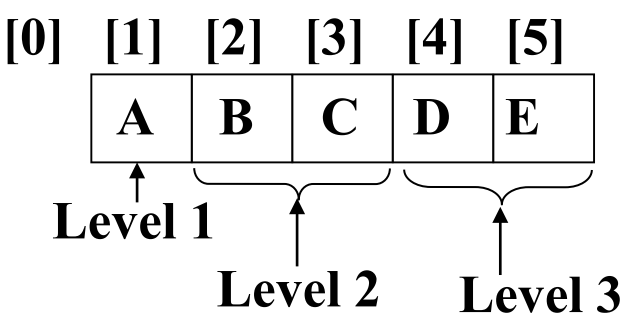
- Data(bintree)

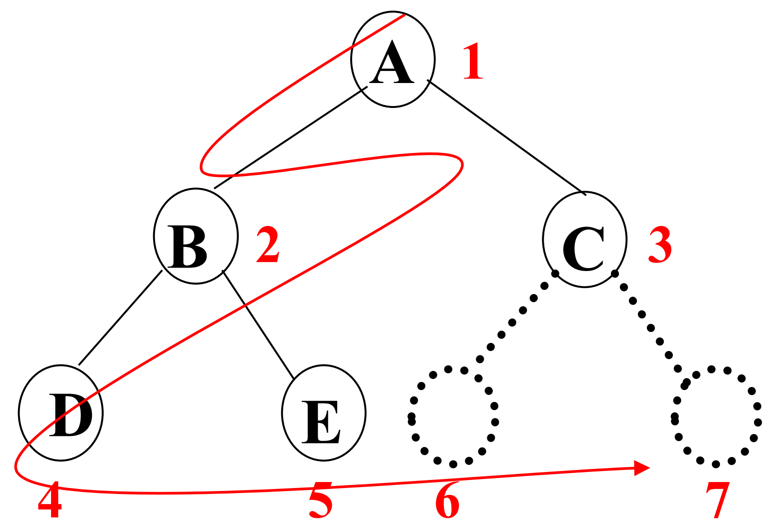
Algorithm: *Data(bintree)*

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

Binary Tree Design (ii)

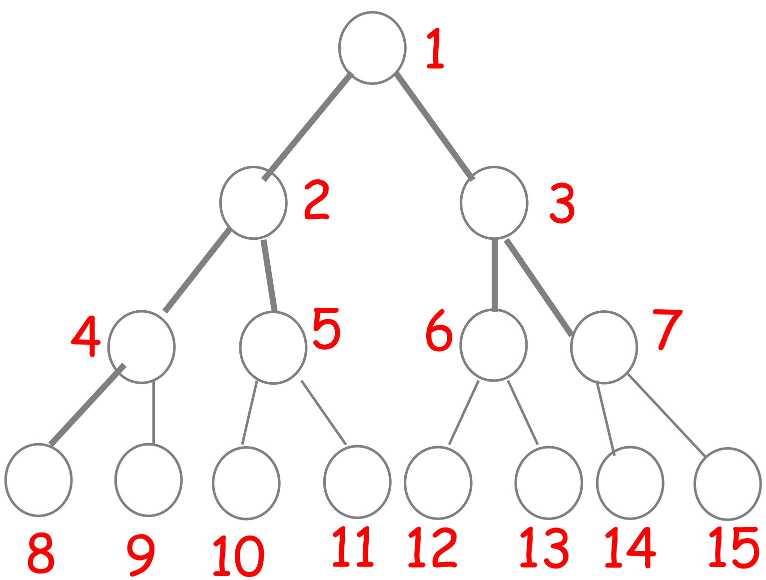
- An array representation
  - Given a complete binary tree with n nodes, for any i-th node, $1 \le i \le n$,
    - parent(i) is $\lfloor i/2 \rfloor$
    - leftChild(i) is at $2i$ if $2i \le n$. Otherwise, i has no left child
    - rightChild(i) is at $2i+1$ if $2i+1 \le n$; otherwise, i has no right child



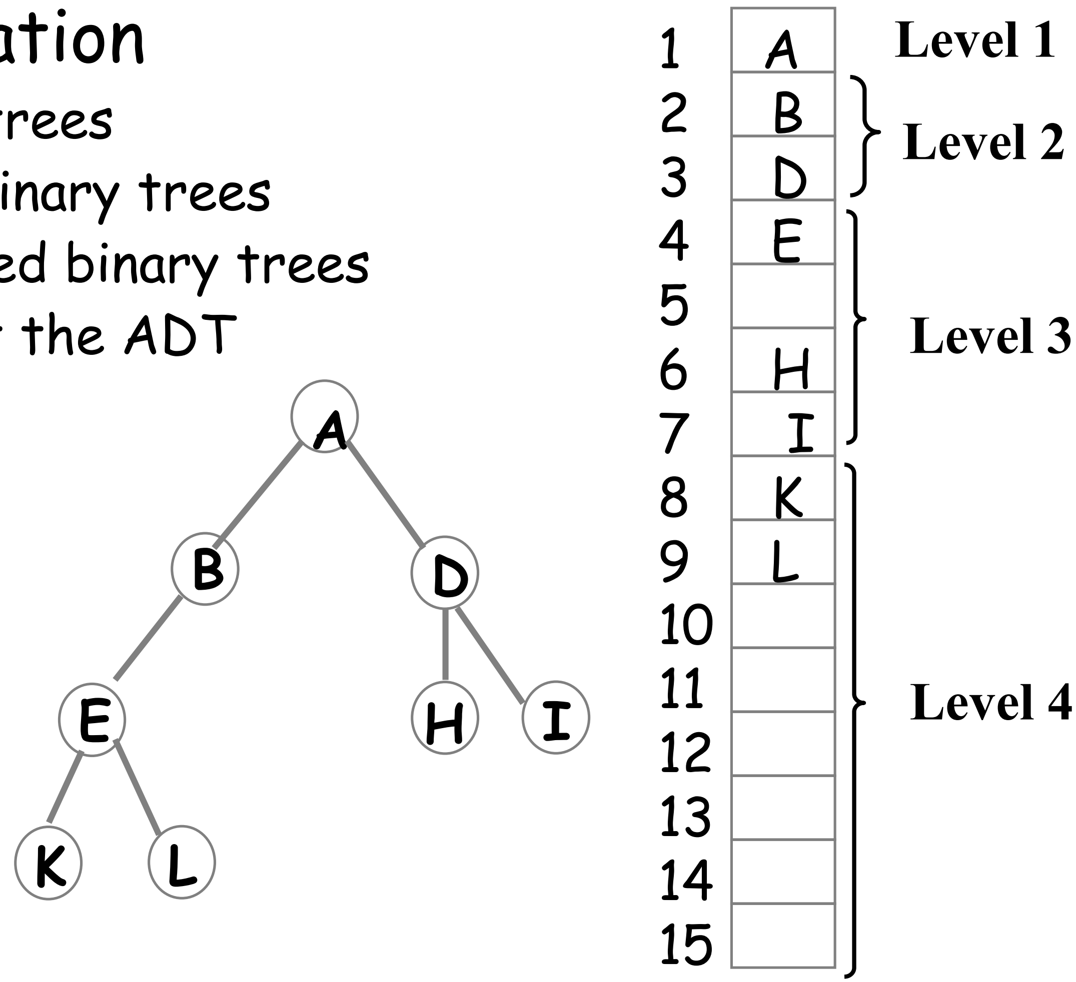


Binary Tree Design (ii)

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

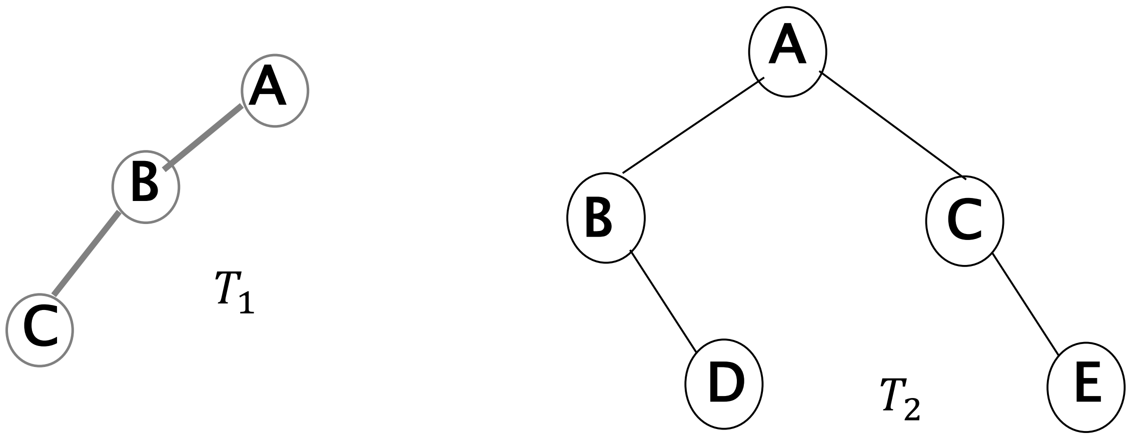


full binary tree



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



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

Traversing Strategy

- 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

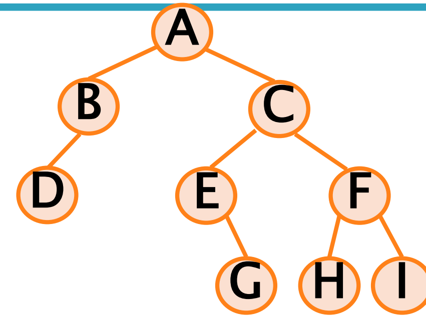
Visit the root

Traverse the left subtree

Traverse the right subtree

A B D C E G F H I

Example:

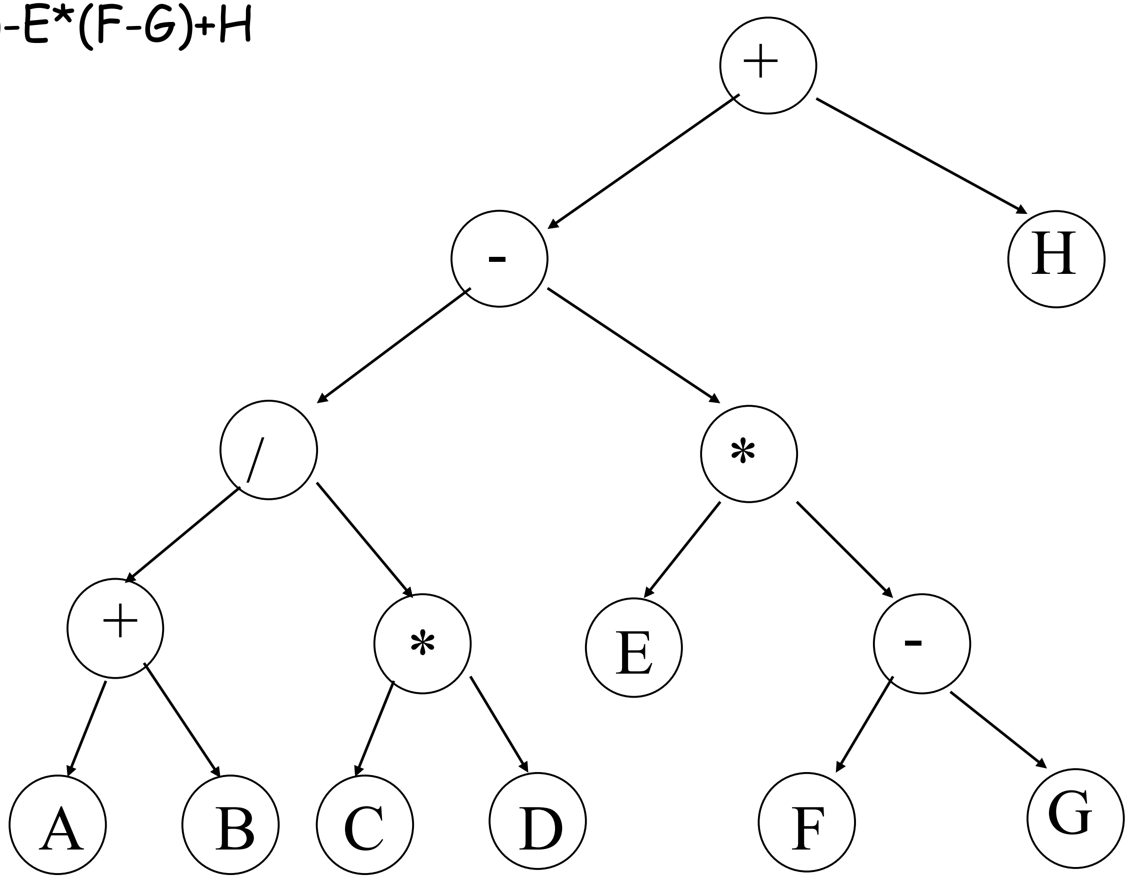


Result:

= 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

Example

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

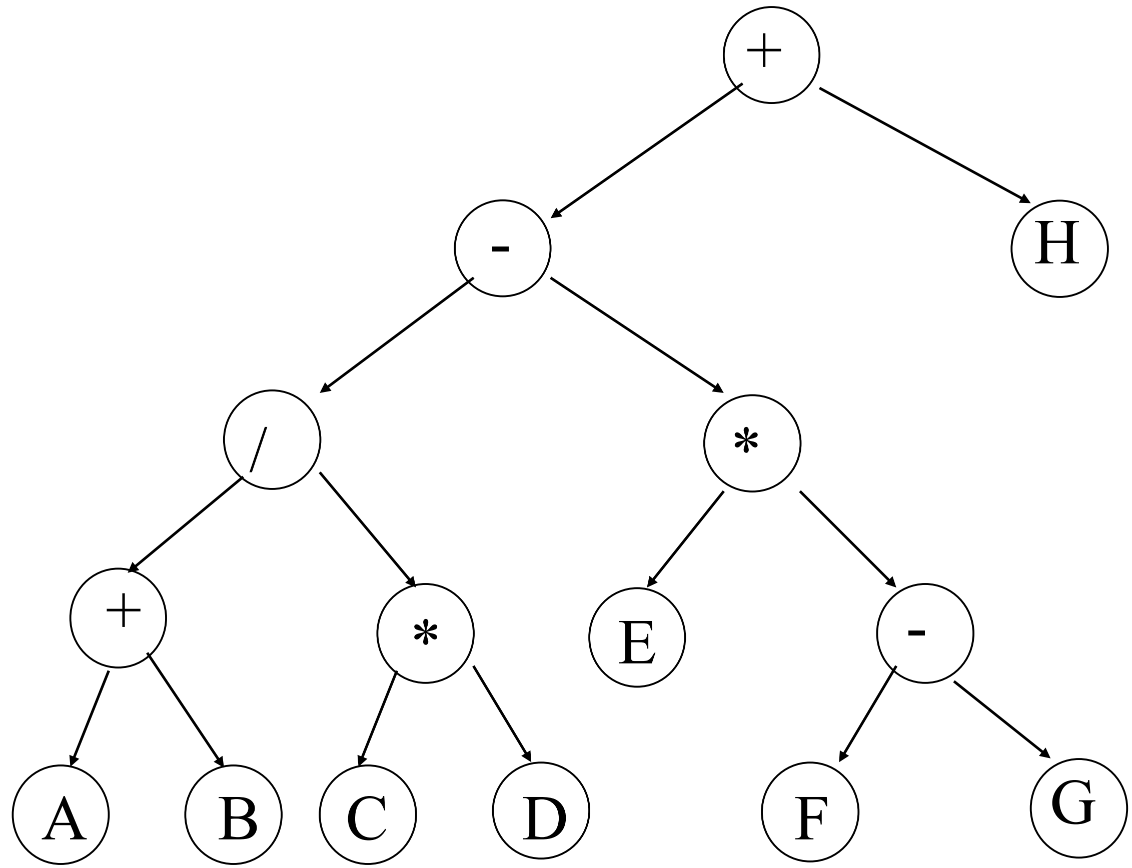


Example

Preorder:
+-/+AB*CD*E-FGH

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

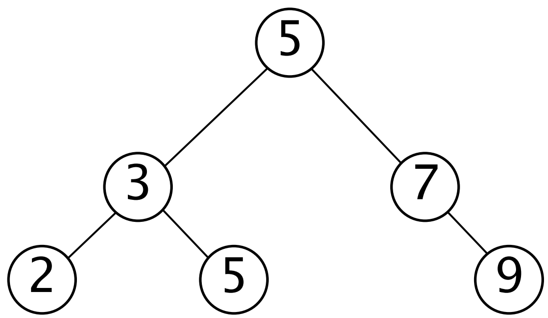
Postorder:
AB+CD*/EFG-*-H+



Implementation

```
INORDER-TREE-WALK(x)
    if x ≠ NIL
       then INORDER-TREE-WALK ( left [x] )
            print key [x]
            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

Traversing Binary Tree

Reconstruction of Binary Tree from its preorder and Inorder sequences

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

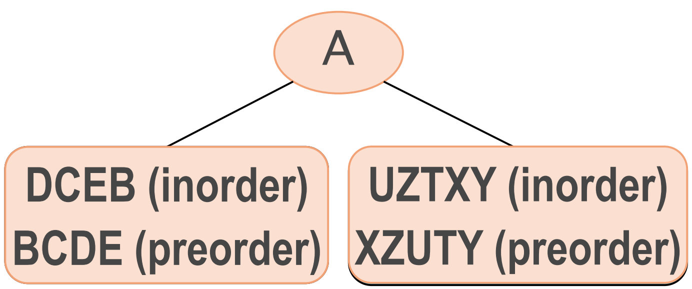
inorder : DCEBAUZTXY

preorder : ABCDEXZUTY

Looking at the whole tree:

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

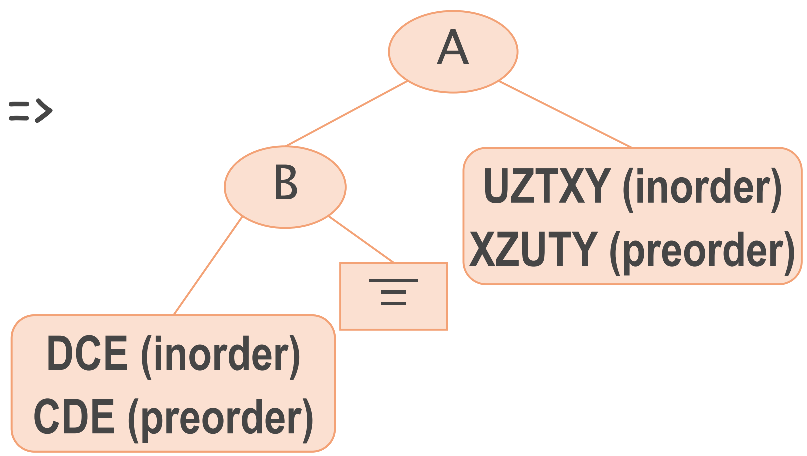
==>



Looking at the left subtree of A:

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

=>



Traversing Binary Tree

Reconstruction of Binary Tree from its preorder and Inorder sequences

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

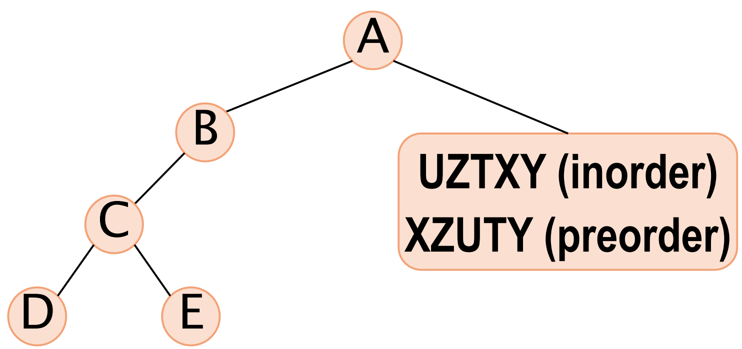
inorder : DCEBAUZTXY

preorder : ABCDEXZUTY

Looking at the left subtree of B:

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

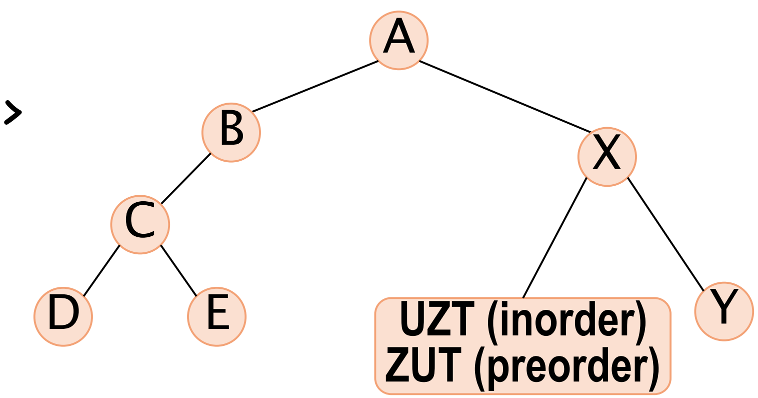
=>



Looking at the right subtree of A:

- "preorder : XZUTY"
  ==> X is the root
- Then, "inorder: UZT**X**Y"

=>



Traversing Binary Tree

Reconstruction of Binary Tree from its preorder and inorder sequences

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

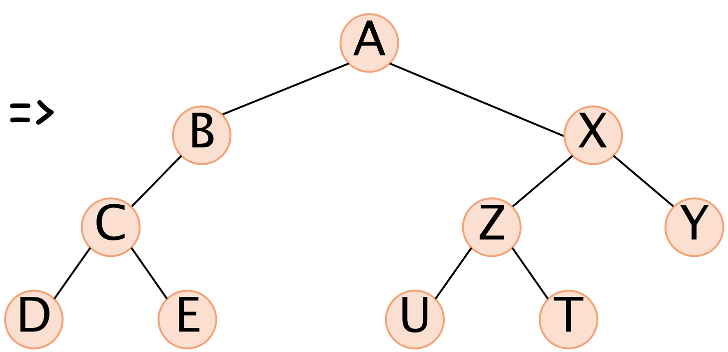
inorder : DCEBAUZTXY

preorder : ABCDEXZUTY

Looking at the left subtree of X:

- "preorder : ZUT"
  ==> Z is the root
- Then, "inorder: U<u>Z</u>T"

=>

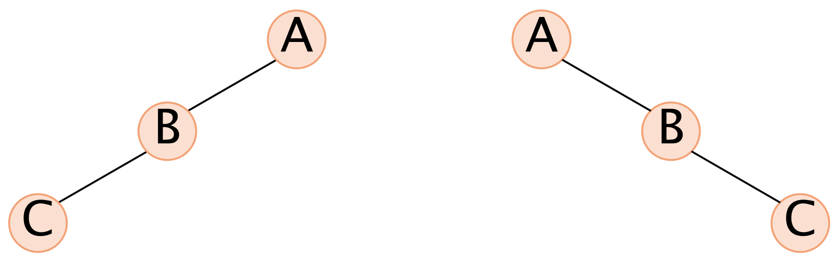


Traversing Binary Tree

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

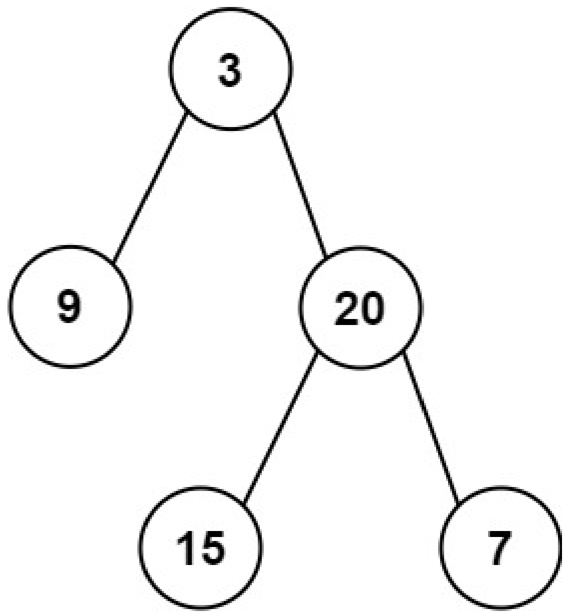
Example: **Preorder sequence: ABC**
Postorder sequence: CBA

We can construct 2 different binary trees:



Exercise

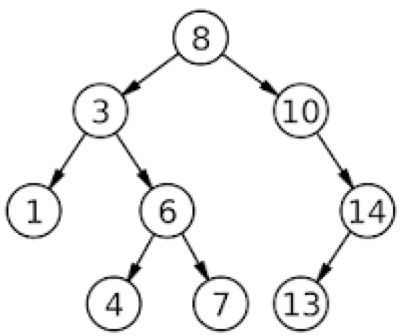
- Construct a binary tree such that
  - preorder=[3,9,20,15,7]
  - inorder=[9,3,15,20,7]

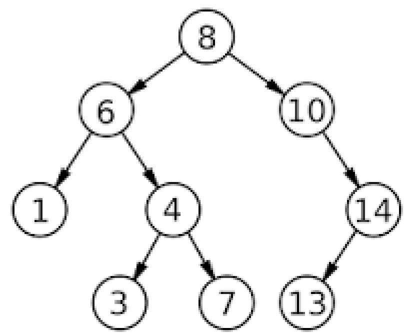


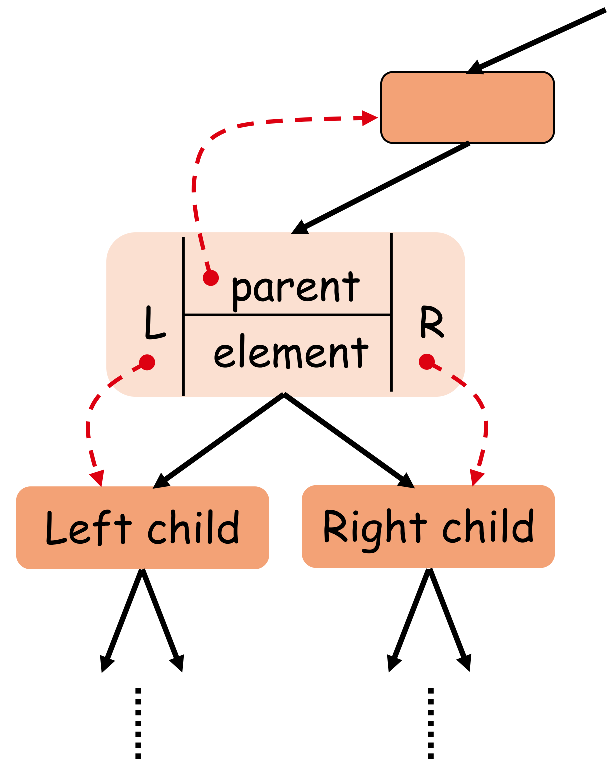
Binary search tree (BST)

Binary Search Tree (BST) Property

- For every node, T, in the tree
  - the key values in its left subtree are *smaller* than the key value of T
  - the key values in its right subtree are *larger* *than* the key value of T





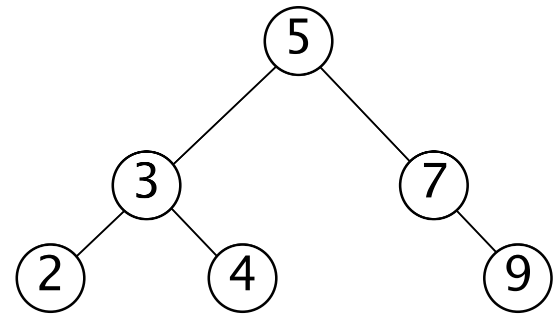


Binary Search Trees

- Support many dynamic set operations
  - find, findMin, findMax, predecessor, successor, insert, delete
- Running time of basic operations on binary search trees
  - On average: $\Theta(\log n)$
    - The expected height of the tree is logn
  - In the worst case: $\Theta(n)$
    - The tree is a linear chain of n nodes

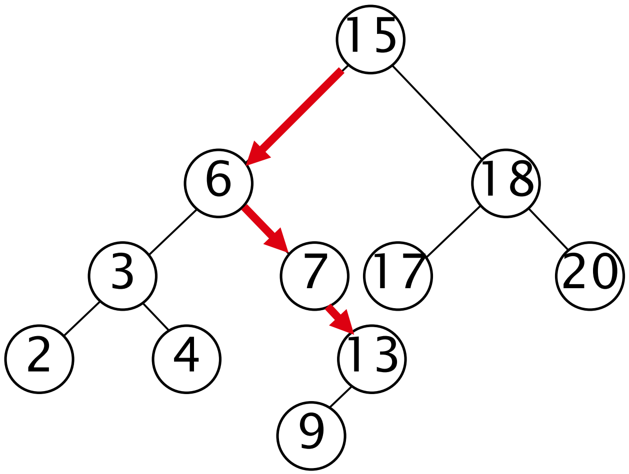
Searching for a Key

- Given a pointer to the root of a tree and a key k:
  - Return a pointer to a node with key k if one exists
  - Otherwise return NIL



- Idea
  - Starting at the root: trace down a path by comparing k with the key of the current node:
    - If the keys are equal: we have found the key
    - If k < key[x] search in the left subtree of x
    - If k > key[x] search in the right subtree of x

Example

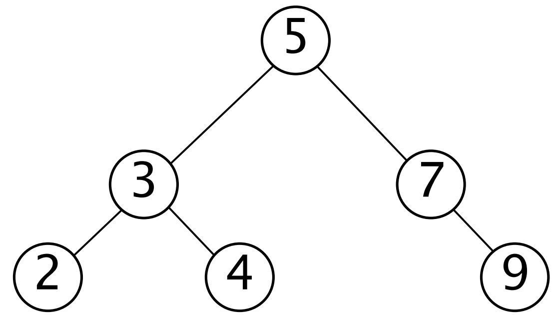


- Search for key 13:
  - $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$

Searching for a Key

find(x, k)

```
if x = NIL or k = key [x]
      then return x
if k < key [x]
      then return find(left [x], k )
      else return find(right [x], k )
```



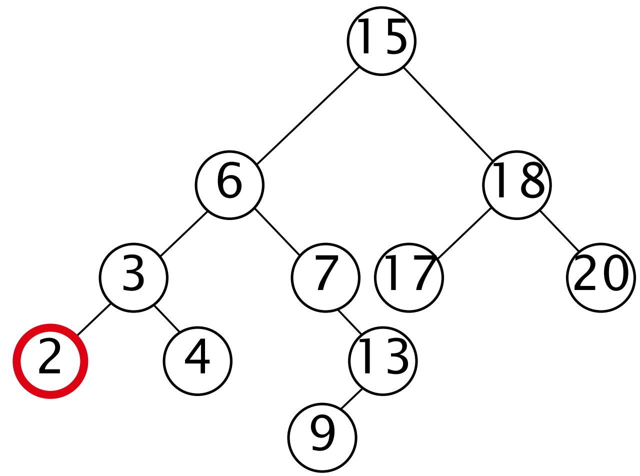
Running Time: O (h),
h is the height of the tree

Finding the Minimum

- Goal: find the minimum value in a BST
  - Following left child pointers from the root, until a NIL is encountered

findMin(x)

1. **while** left [x] $\neq$ NIL
2. **do** x $\leftarrow$ left [x]
3. **return** x



Minimum = 2

Running time: O(h)

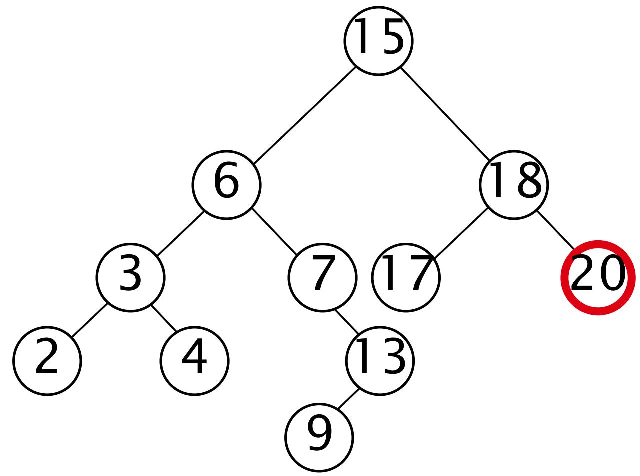
h is the height of tree

Finding the Maximum

- Goal: find the maximum value in a BST
  - Following right child pointers from the root, until a NIL is encountered

findMax(x)

```
while right [x] ≠ NIL
        do x ← right [x]
return x
```



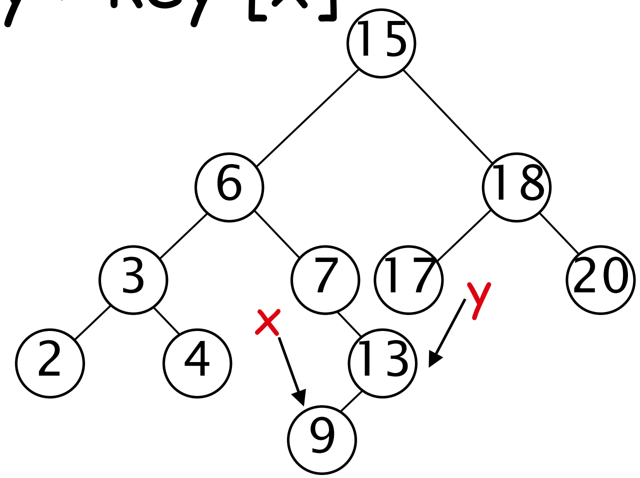
Maximum = 20

Running time: O(h)
h is the height of tree

Successor

Def: successor (x) = y, such that key [y] is the smallest key > key [x]

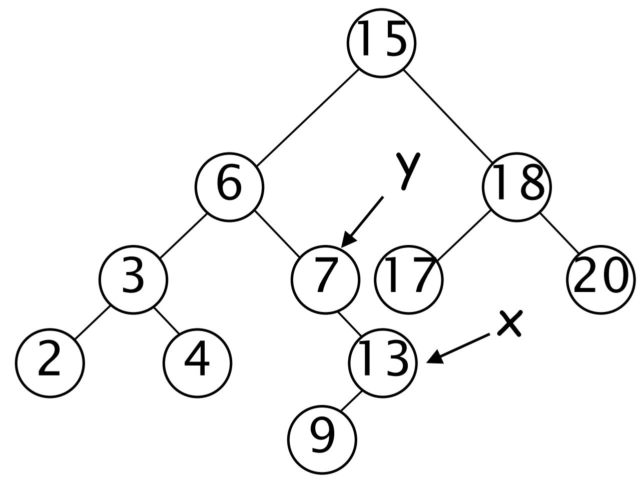
- E.g.: successor (15) = 17
  successor (13) = 15
  successor (9) = 13



- Case 1: right (x) is non empty
  - successor (x) = the minimum in right (x)
- Case 2: right (x) is empty
  - go up the tree until the current node is a left child: successor (x) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the largest element

Successor

```
successor(x)
    if right [x] ≠ NIL
        then return findMin(right [x])
    y ← p[x]
    while y ≠ NIL and x = right [y]
        do x ← y
            y ← p[y]
    return y
```



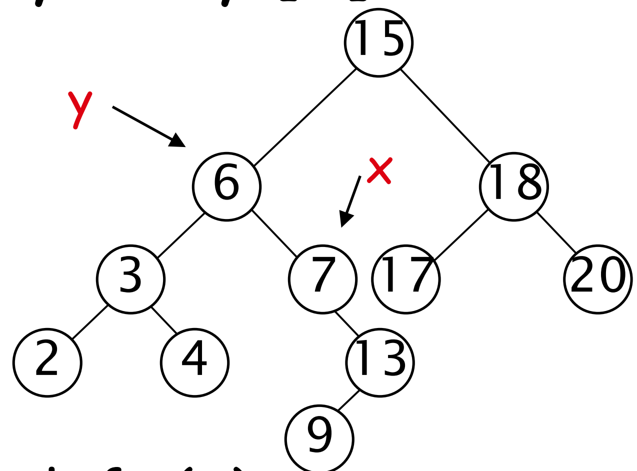
Running time: O (h)

h is the height of the tree

Predecessor

Def: predecessor (x) = y, such that key [y] is the biggest key < key [x]

- E.g.: predecessor (15) = 13
  predecessor (9) = 7
  predecessor (7) = 6



- Case 1: left (x) is non empty
  - predecessor (x) = the maximum in left (x)
- Case 2: left (x) is empty
  - go up the tree until the current node is a right child: predecessor (x) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the smallest element