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

Introduction to Computer Science: Programming Methodology

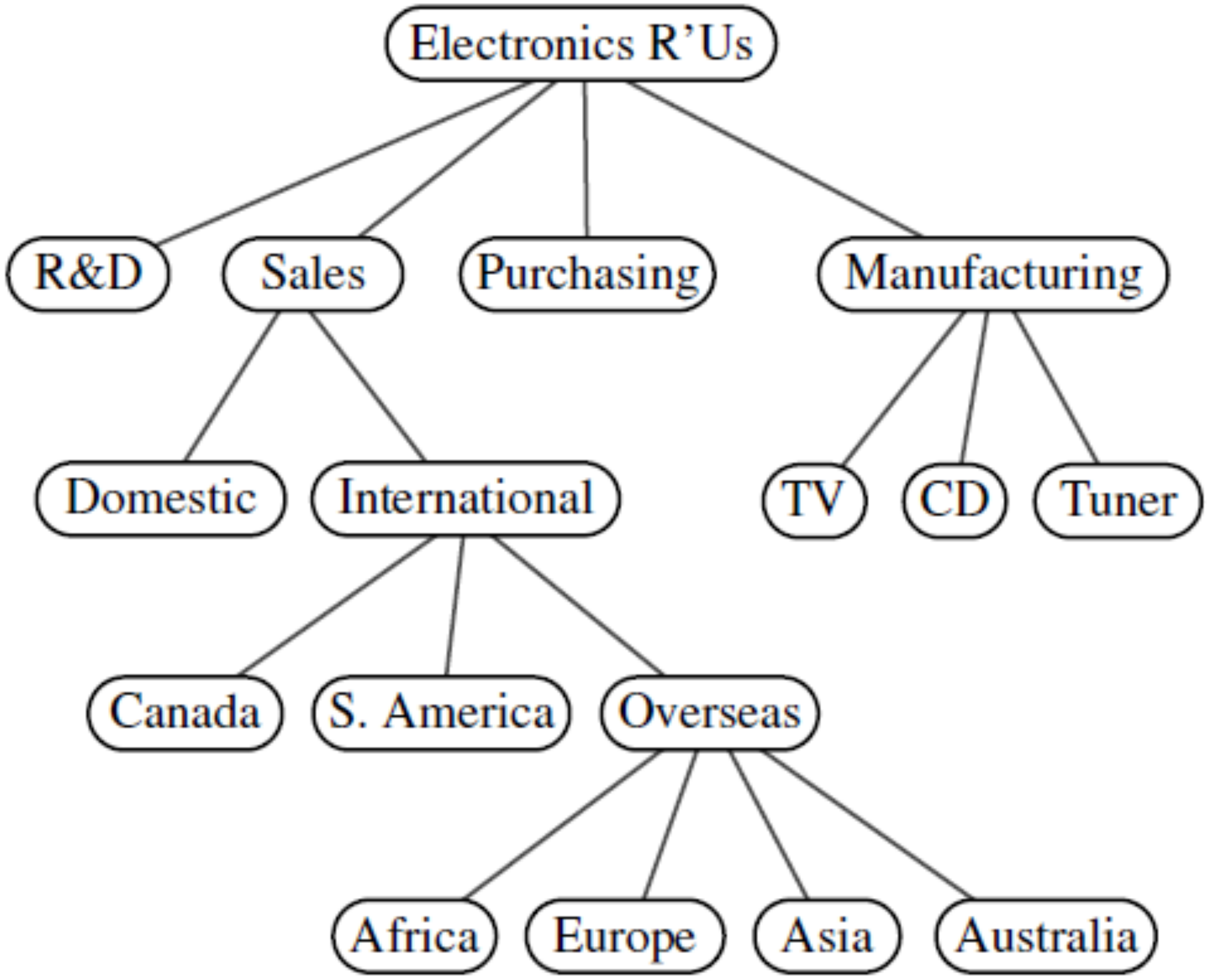
Lecture 11
Tree

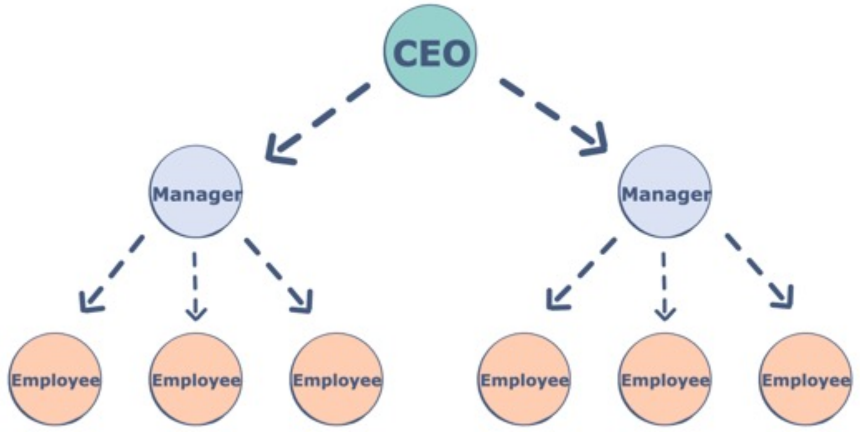
Tongxin Li
School of Data Science

Tree

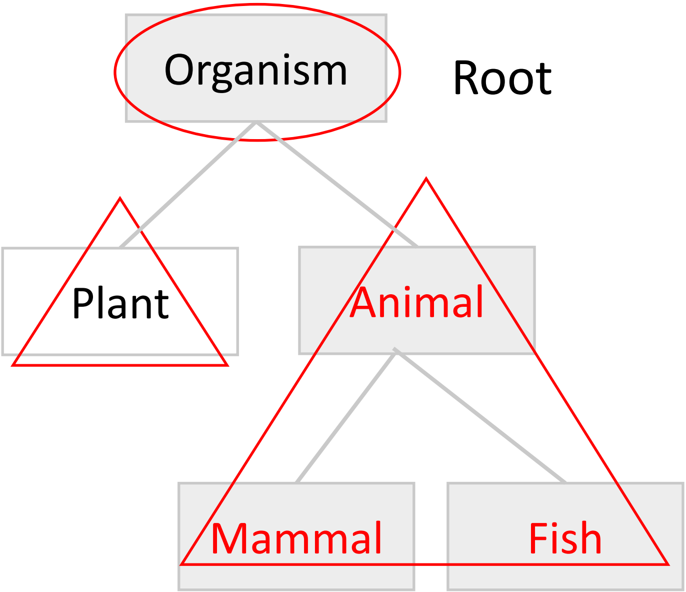
- A tree is a data structure that stores elements hierarchically
- With the exception of the top element, each element in a tree has a parent element and zero or more children elements
- We typically call the top element the root of the tree, but it is drawn as the highest element

Example: The organization of a company





Semantic concept



Formal definition of a tree

- Formally, we define a tree T as a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following properties:

  - ✓ If T is nonempty, it has a special node, called the root of T, that has no parent.
  - ✓ Each node v of T (different from the root) has a unique parent node w; every node with parent w is a child of w.

Edge and path

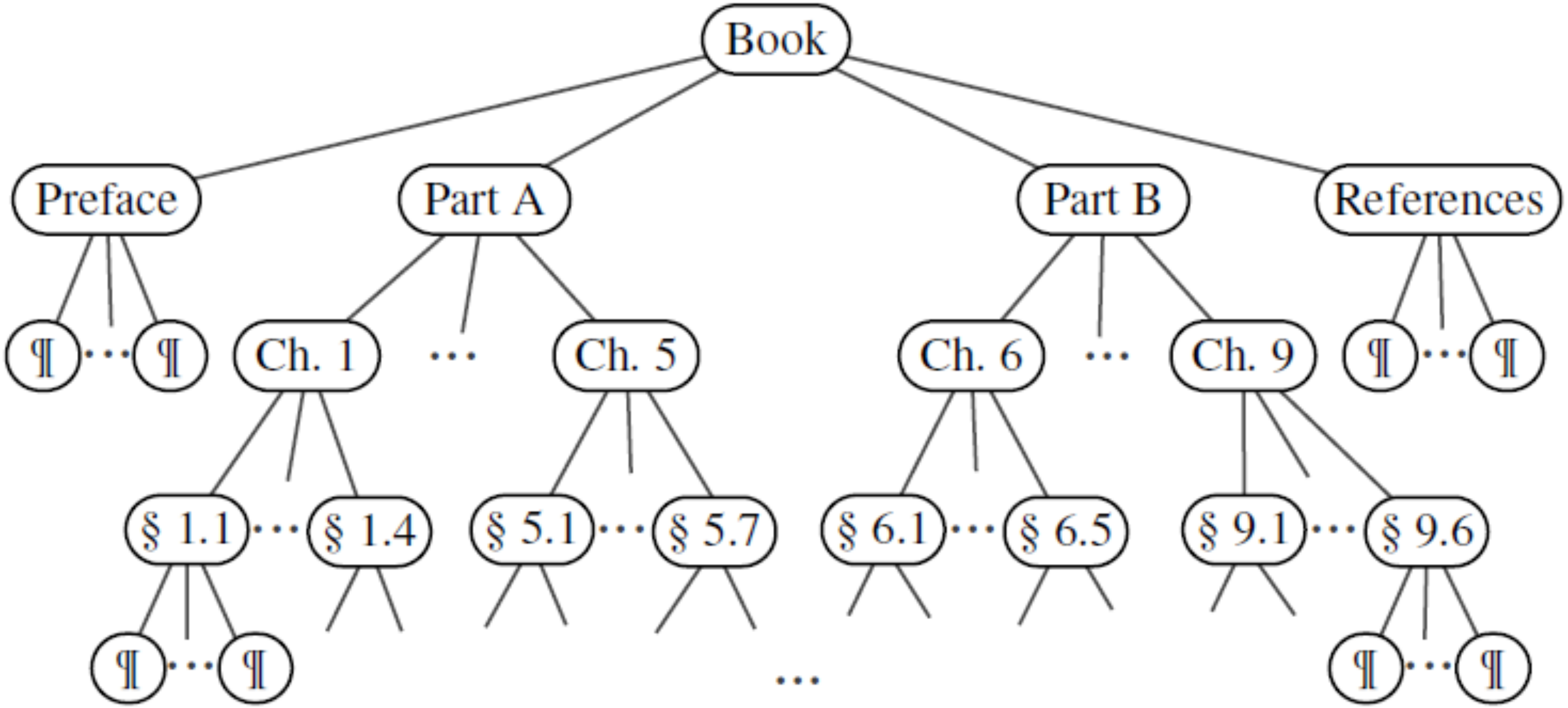
- An edge of tree T is a pair of nodes (u,v) such that u is the parent of v, or vice versa
- A path of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge
- The depth of a node v is the length of the path connecting root node and v

Internal and leaf nodes

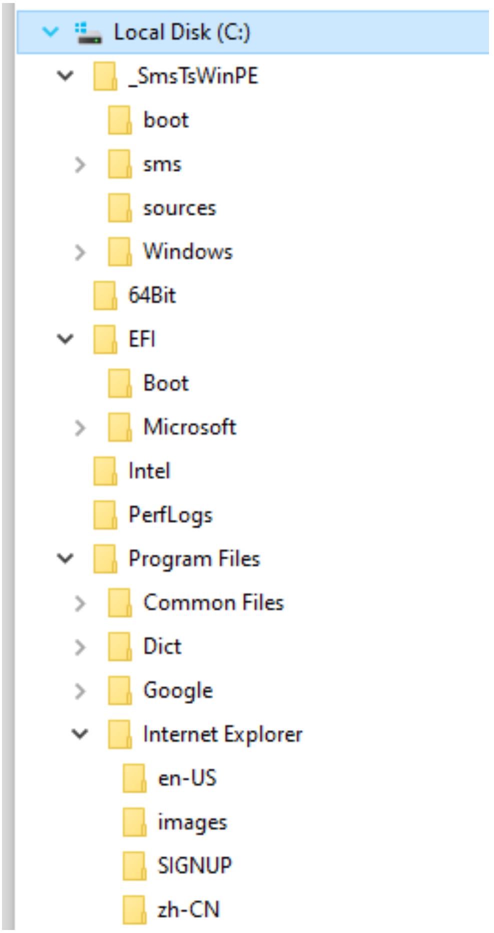
- A node is called a leaf node if it has no child
- If a node has at least one child, it is an internal node

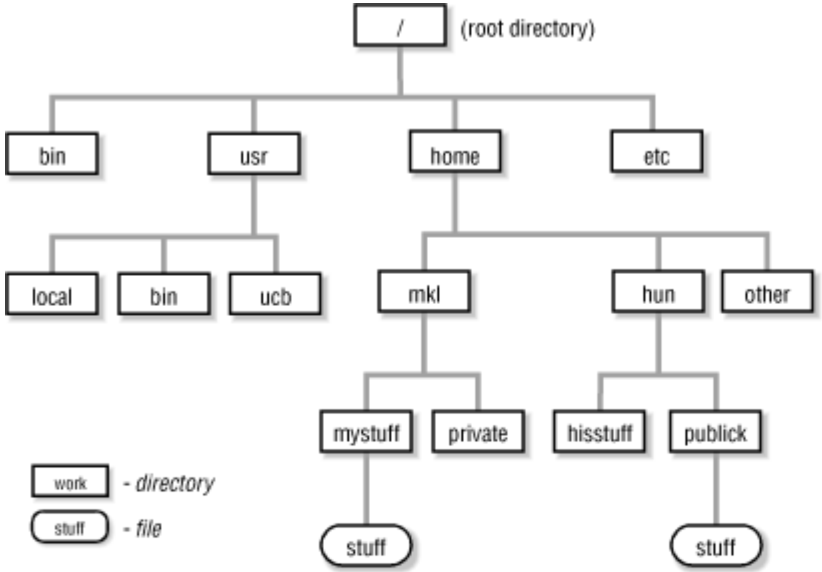
Ordered tree

- A tree is ordered if there is a meaningful linear order among the children of each node; such an order is usually visualized by arranging siblings from left to right, according to their order



Example: File system





A file is a leaf node, and a folder/directory is an internal node

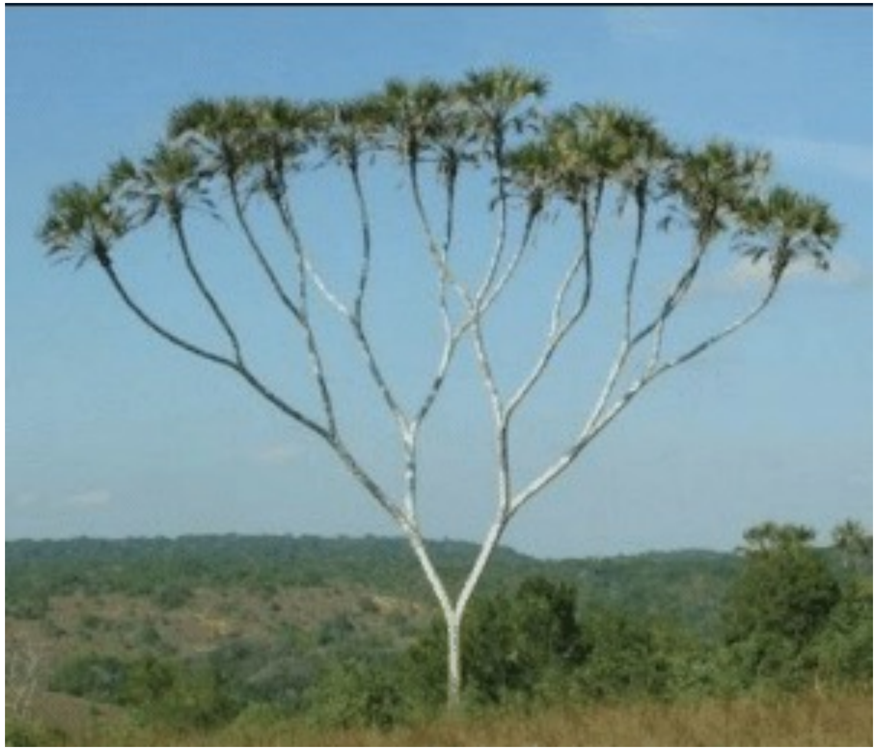
Binary tree

- A binary tree is an ordered tree with the following properties:
  1. Every node has at most two children
  2. Each child node is labelled as being either a left child or a right child
  3. A left child precedes a right child in the order of children of a node

The subtree rooted at a left or right child of an internal node v is called a left subtree or right subtree, respectively, of v

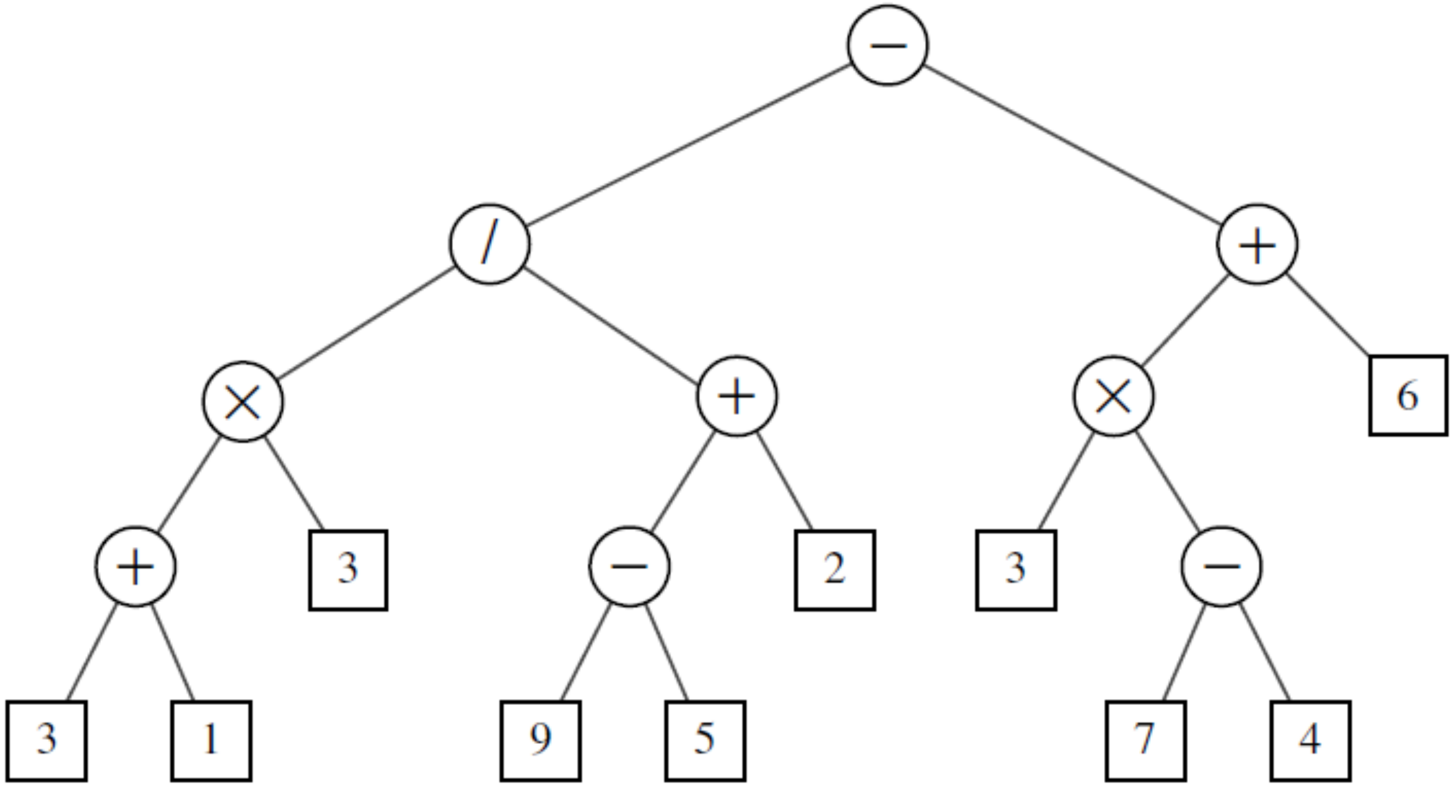
A binary tree is proper if each node has either zero or two children. Some people also refer to such trees as being full binary trees

A wild binary tree



Example: Represent an expression with binary tree

- An arithmetic expression can be represented by a binary tree whose leaves are associated with variables or constants, and whose internal nodes are associated with one of the operators +, −, ×, and /



Binary tree class

- We define a tree class based on a class called Node; an element is stored as a node

- Each node contains three references, one pointing to the parent node, two pointing to the child nodes

Implementing the binary tree

```
class Node:

    def __init__(self,element, parent = None,\
        left = None, right = None):
        self.element = element
        self.parent = parent
        self.left = left
        self.right = right

class LBTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def __len__(self):
        return self.size
```

```
def find_root(self):
    return self.root

def parent(self,p):
    return p.parent

def left(self,p):
    return p.left

def right(self,p):
    return p.right

def num_child(self,p):
    count = 0
    if p.left is not None:
        count+=1
    if p.right is not None:
        count+=1
    return count
```

Implementing the binary tree

```
def add_root(self,e):
    if self.root is not None:
        print('Root already exists.')
        return None
    self.size = 1
    self.root = Node(e)
    return self.root

def add_left(self,p,e):
    if p.left is not None:
        print('Left child already exists.')
        return None
    self.size+=1
    p.left = Node(e,p)
    return p.left
```

```
def add_right(self,p,e):
    if p.right is not None:
        print('Right child already exists.')
        return None
    self.size+=1
    p.right = Node(e,p)
    return p.right

def replace(self,p,e):
    old = p.element
    p.element = e
    return old

def delete(self,p):
    if p.parent.left is p:
        p.parent.left = None
    if p.parent.right is p:
        p.parent.right = None
    return p.element
```

Example: Use the binary tree class

```
def main():
    t = LBTree()
    t.add_root(10)
    t.add_left(t.root, 20)
    t.add_right(t.root, 30)
    t.add_left(t.root.left, 40)
    t.add_right(t.root.left, 50)
    t.add_left(t.root.right, 60)
    t.add_right(t.root.left.left, 70)

    print(t.root.element)
    print(t.root.left.element)
    print(t.root.right.element)
    print(t.root.left.right.element)
```

```
>>> main()
10
20
30
50
```

Traverse a linked list

```
p = head
while(p!=None):
   print(p.element)
   p = p.pointer
```

Traverse a binary tree

Different traversing strategies

- **Pre-order (depth-first)**
  - Visit the node
  - Traverse the left subtree in pre-order
  - Traverse the right subtree in pre-order

- **In-order**
  - Traverse the left subtree in in-order
  - Visit the node
  - Traverse the right subtree in in-order

- **Post-order**
  - Traverse the left subtree in post-order
  - Traverse the right subtree in post-order
  - Visit the node

Pre-order traversal

preorder traversal

Visit the root

↓

Traverse the left subtree

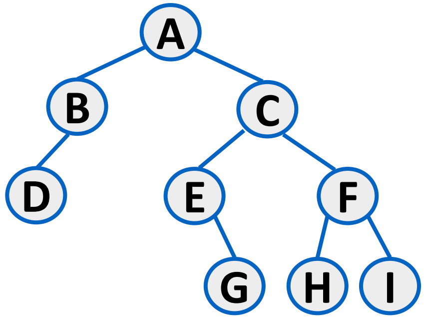
↓

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)

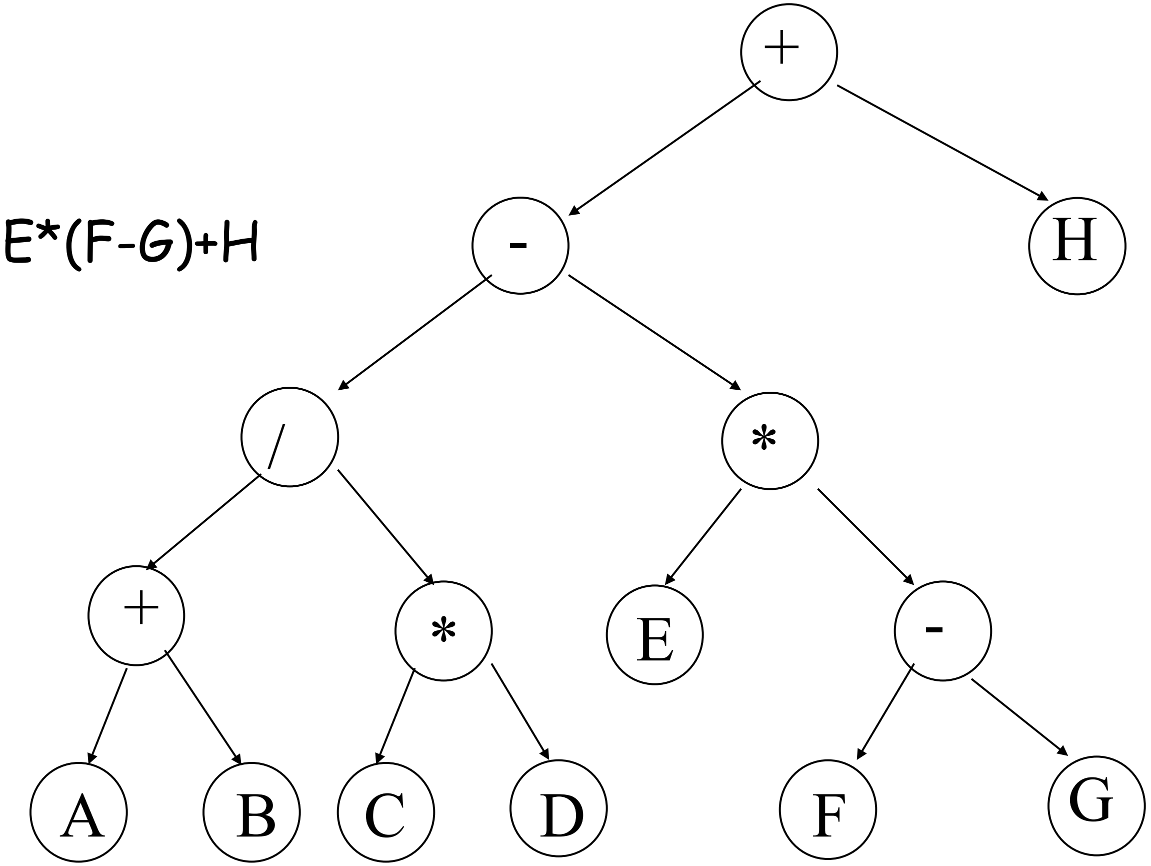
= A B D C **E G** F H (H's left=NULL) (H's right =NULL) (F's right)

= 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: Represent an expression

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



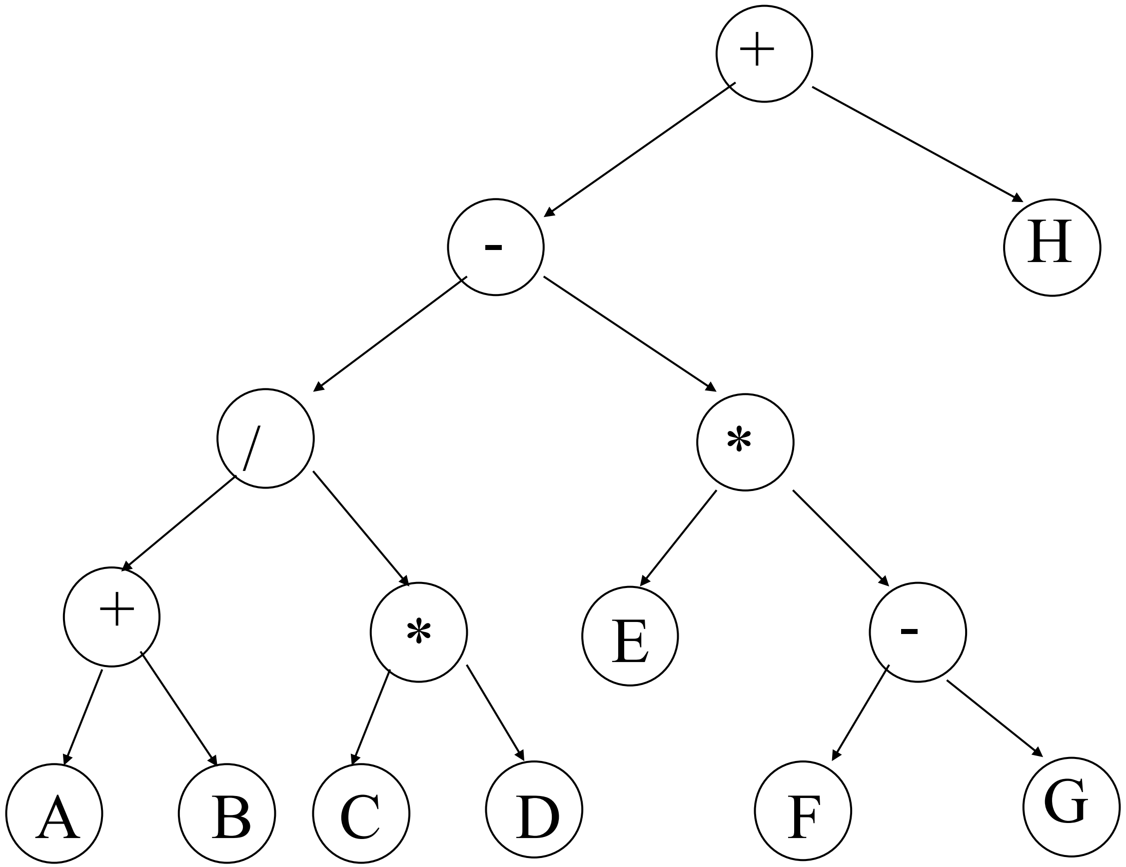
Example: Represent an expression

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

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

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

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



Example: Represent an expression

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

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

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

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

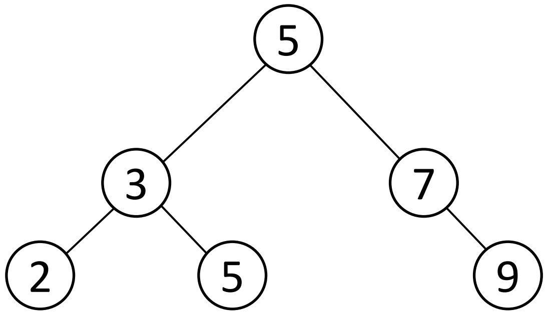
Postfix Expression	Infix Equivalent	Result
4 5 7 2 + − ×	4 × (5 − (7 + 2))	−16
3 4 + 2 × 7 /	((3 + 4) × 2)/7	2
5 7 + 6 2 − ×	(5 + 7) × (6 − 2)	48
4 2 3 5 1 − + × + ×	? × (4 + (2 × (3 + (5 − 1))))	not enough operands
4 2 + 3 5 1 − × +	(4 + 2) + (3 × (5 − 1))	18
5 3 7 9 + +	(3 + (7 + 9)) ... 5???	too many operands

Question: Given an expression, what is the relationship between its postfix and post-order?

Implementation (Pseudocode)

```
INORDER-TREE-WALK(x)
    if x is not None:
      then INORDER-TREE-WALK ( left [x] )
           print key [x]
           INORDER-TREE-WALK ( right [x] )
```

E.g.:

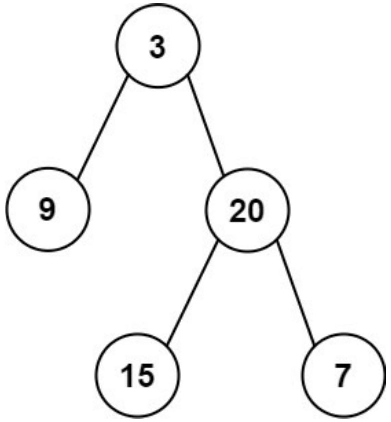


Output: 2 3 5 5 7 9

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

Exercise

- Given a binary tree, show its pre-order, in-order, and post-order



- Pre-order=[3, 9, 20, 15, 7]
- In-order=[9, 3, 15, 20, 7]
- Post-order=[9, 15, 7, 20, 3]

Example: Reconstruct a binary tree

Reconstruction of Binary Tree from its preorder and In-order sequences

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

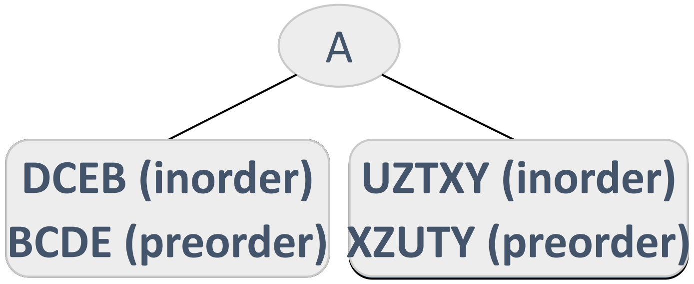
in-order : DCEBAUZTXY

pre-order : ABCDEXZUTY

Looking at the whole tree:

- “pre-order : **A**BCDEXZUTY”
  => A is the root
- Then, “in-order : DCEB**A**UZTXY”

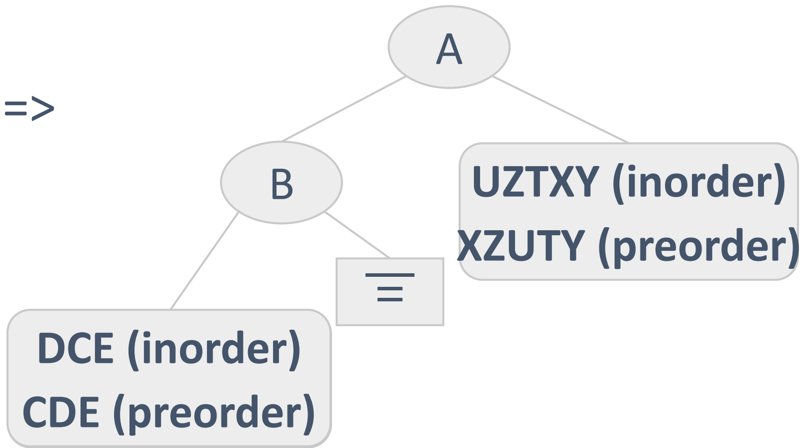
=>



Looking at the left subtree of A:

- “pre-order : BCDE”
  => B is the root
- Then, “in-order: DCE**B**”

=>

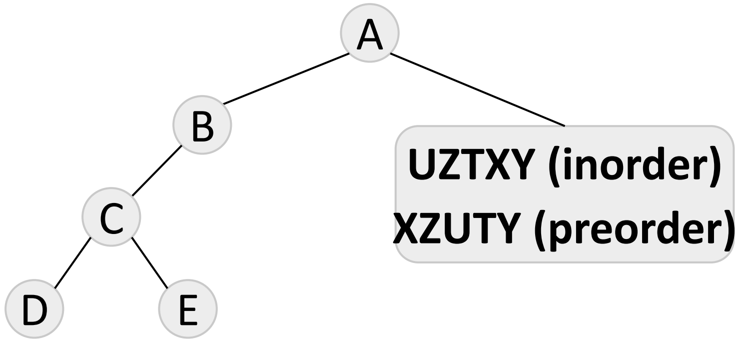


Reconstruct a binary tree

Looking at the left subtree of B:

- “preorder : CDE”
  => C is the root
- Then, “inorder: D**C**E”

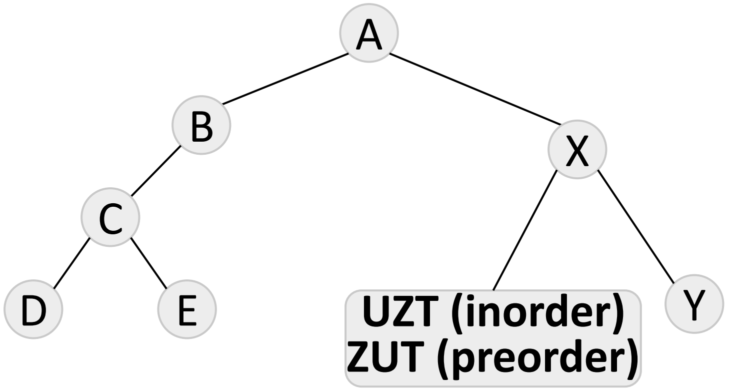
=>



Looking at the right subtree of A:

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

=>

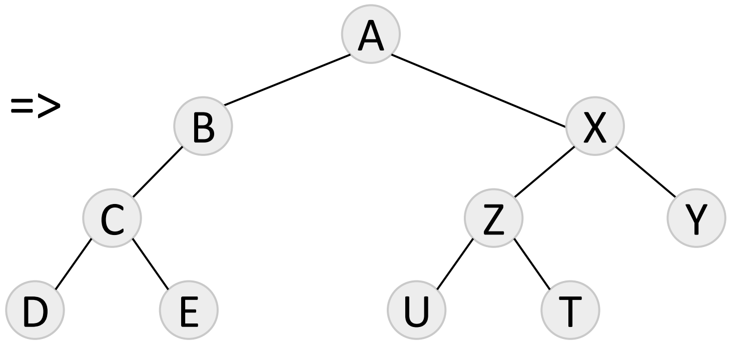


Reconstruct a binary tree

Looking at the left subtree of X:

- “pre-order : ZUT”
  => Z is the root
- Then, “in-order: U**<u>Z</u>**T”

=>



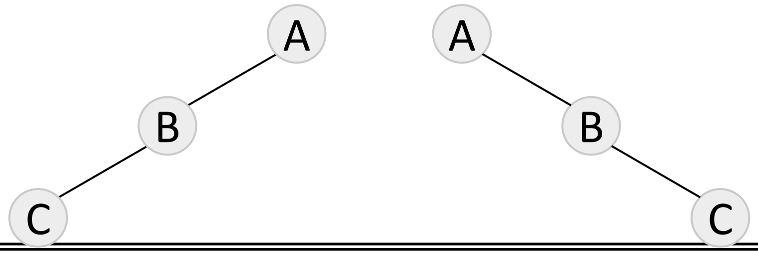
Reconstruct a binary tree

Warning: A binary tree may not be uniquely defined by its pre-order and post-order sequences.

Example: **Pre-order sequence:** **ABC**

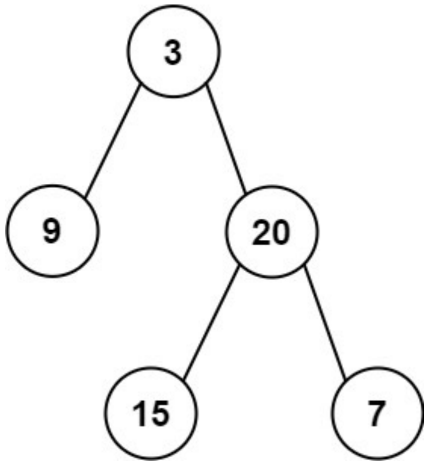
Post-order sequence: **CBA**

We can construct 2 different binary trees:



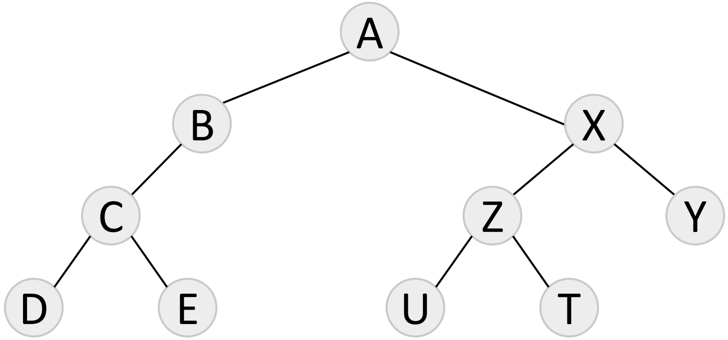
Exercise

- Construct a binary tree such that
  - Pre-order=[3,9,20,15,7]
  - In-order=[9,3,15,20,7]



Exercise

- Construct a binary tree such that
  - Pre-order=[A, B , C , D , E , X , Z , U , T , Y]
  - Post-order=[D , E , C , B , U , T , Z , Y , X , A]



Practice

- **Find the maximal element of a binary tree**

Example: Find the max number

```
class Node:
    def __init__(self, key=None, left=None, right=None):
        self.key = key
        self.left = left
        self.right = right

def findMax(root):
    if (root == None):
        return float('-inf’)
    res = root.data
    lres = findMax(root.left)
    rres = findMax(root.right)
    return max(res, lres, rres)
```

Practice

- **Check if two binary trees are identical or not**

```
         1                       1
       /   \                   /   \
      2     3                 2     3
    /  \   /  \             /  \   /  \
   4    5 6    7           4    5 6    7

Output: True
```

```
Input:
          1                    1
        /   \                /   \
       2     3              2     3
     /  \   /  \          /  \   /
    4    5 6    7        4    5 6

Output: False
```

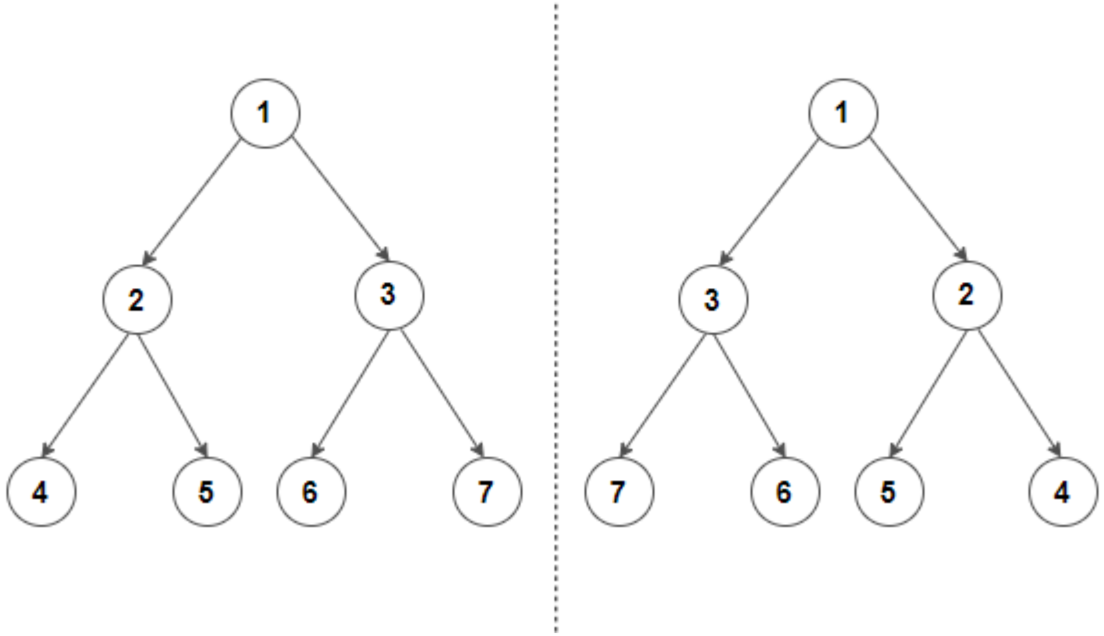
```
Input:
          1                    1
        /   \                /   \
       2     3              2     3
     /  \   /  \          /  \   /  \
    4    5 6    7        4    5 6    8

Output: False
```

Example: Check Identity

```
def isIdentical(x, y):
    if x is None and y is None:
        return True
    return (x is not None and y is not None) and (x.key == y.key) and \
        isIdentical(x.left, y.left) and isIdentical(x.right, y.right)
```

Practice

- Swap a tree (**Convert a binary tree to its mirror**)

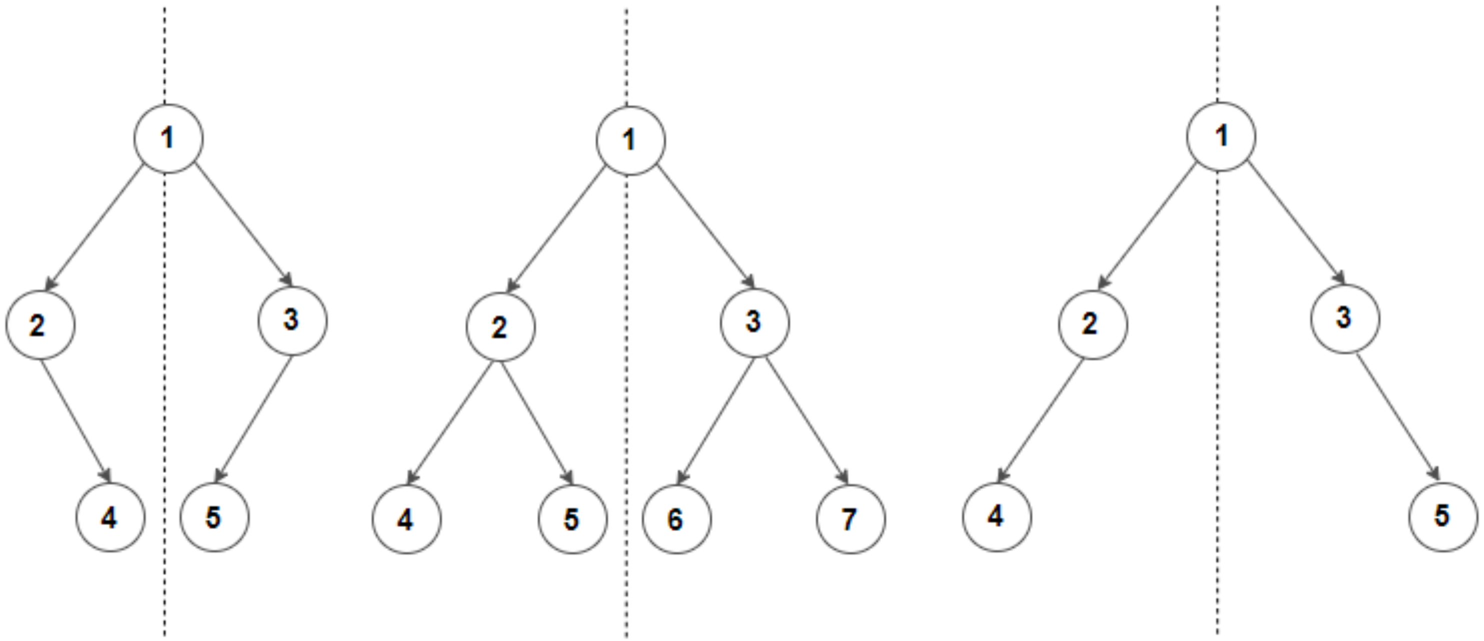


Example: Convert a binary tree to its mirror

```
def swap(root):
    if root is None:
        return
    temp = root.left
    root.left = root.right
    root.right = temp

def convertToMirror(root):
    if root is None:
        return
    convertToMirror(root.left)
    convertToMirror(root.right)
    swap(root)
```

Practice

- **Check if a binary tree is symmetric or not**



Example: Check if a binary tree is symmetric

```
def isSymmetric(X, Y):
    if X is None and Y is None:
        return True
    return (X is not None and Y is not None) and \
        isSymmetric(X.left, Y.right) and \
        isSymmetric(X.right, Y.left)
```

Summary: Tree Traversal

- Pre-order
- In-order
- Post-order

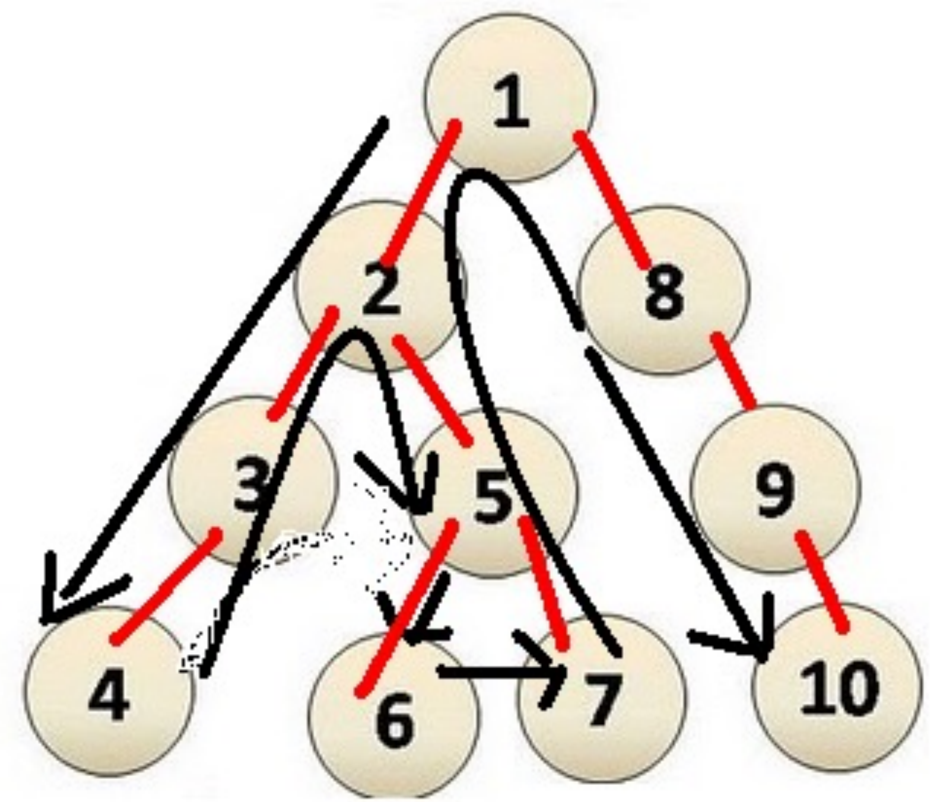
} **(Depth First)**

- Level-order **(Breadth First)**

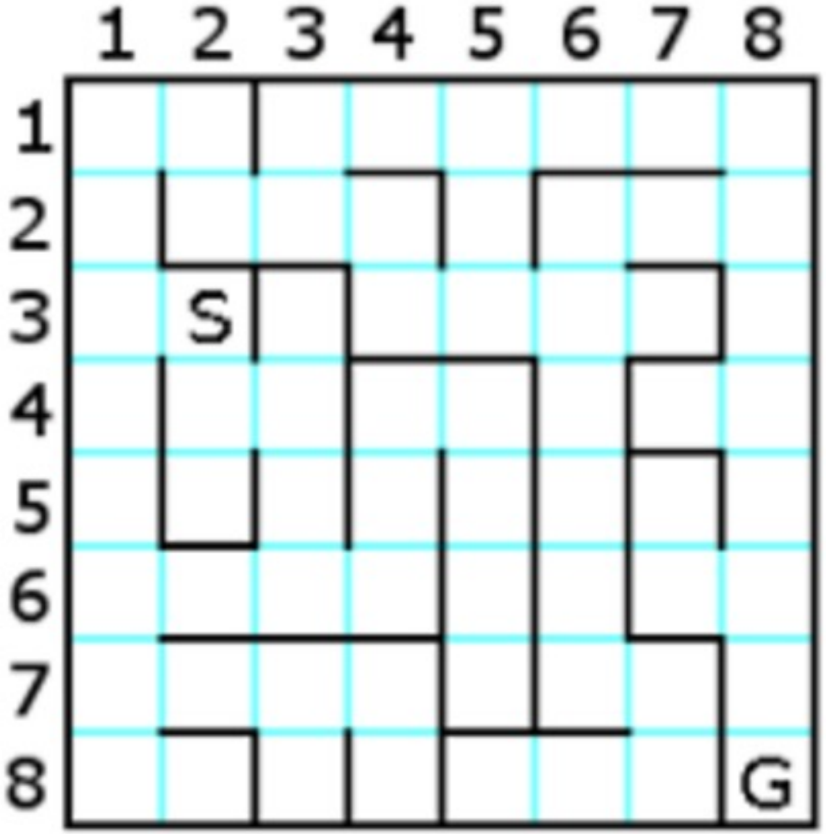
Depth first search over a tree

- Depth-first search (DFS) is a fundamental algorithm for traversing or searching tree data structures

- One starts at the root and explores as deep as possible along each branch before backtracking



Example: search a path in a maze



The code of DFS over a binary tree

```
def DFSearch(t):
    if t:
        print(t.element)
    if (t.left is None) and (t.right is None):
        return
    else:
        if t.left is not None:
            DFSearch(t.left)
        if t.right is not None:
            DFSearch(t.right)
```

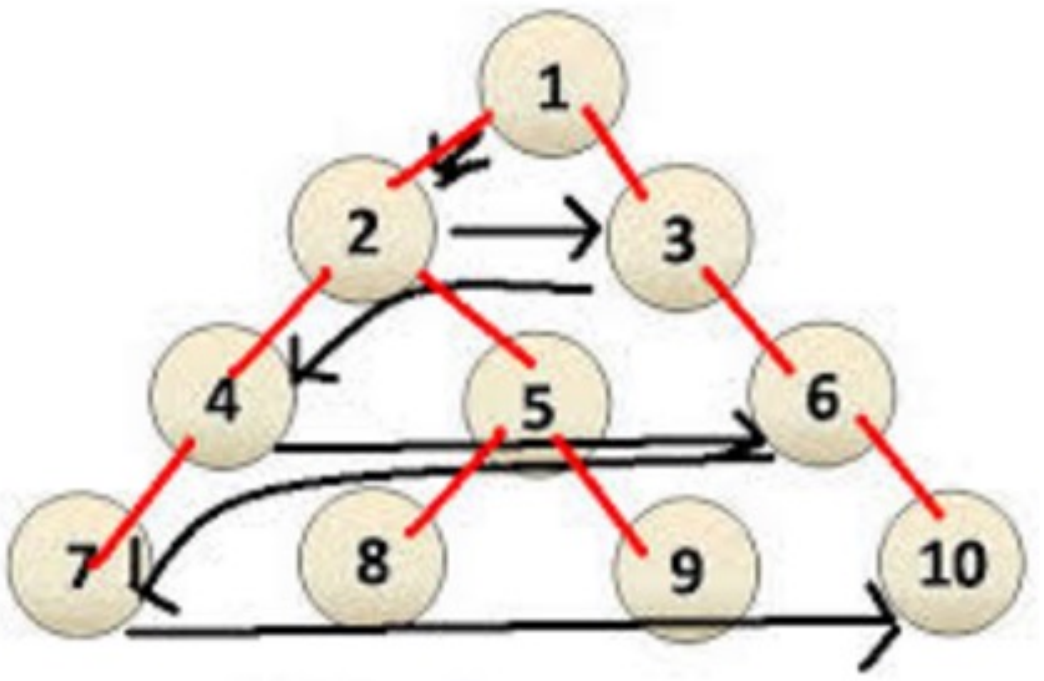
The code of DFS over a binary tree

Question: Is this pre-order, in-order, or post-order DFS?

```
def DFSearch(t):
    if t:
        print(t.element)
    if (t.left is None) and (t.right is None):
        return
    else:
        if t.left is not None:
            DFSearch(t.left)
        if t.right is not None:
            DFSearch(t.right)
```

Breadth first search over a tree

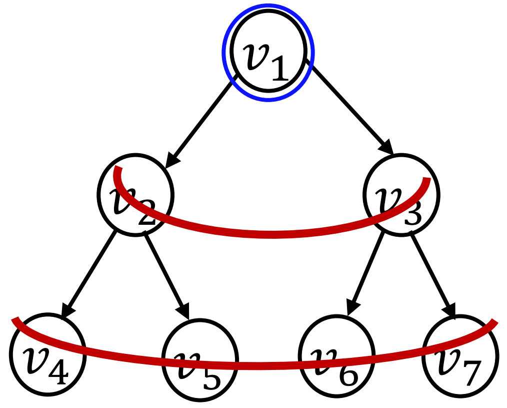
- Breadth-first search (BFS) is another very important algorithm for traversing or searching tree data structures
- Starts at the root and we visit all the positions at depth d before we visit the positions at depth d +1



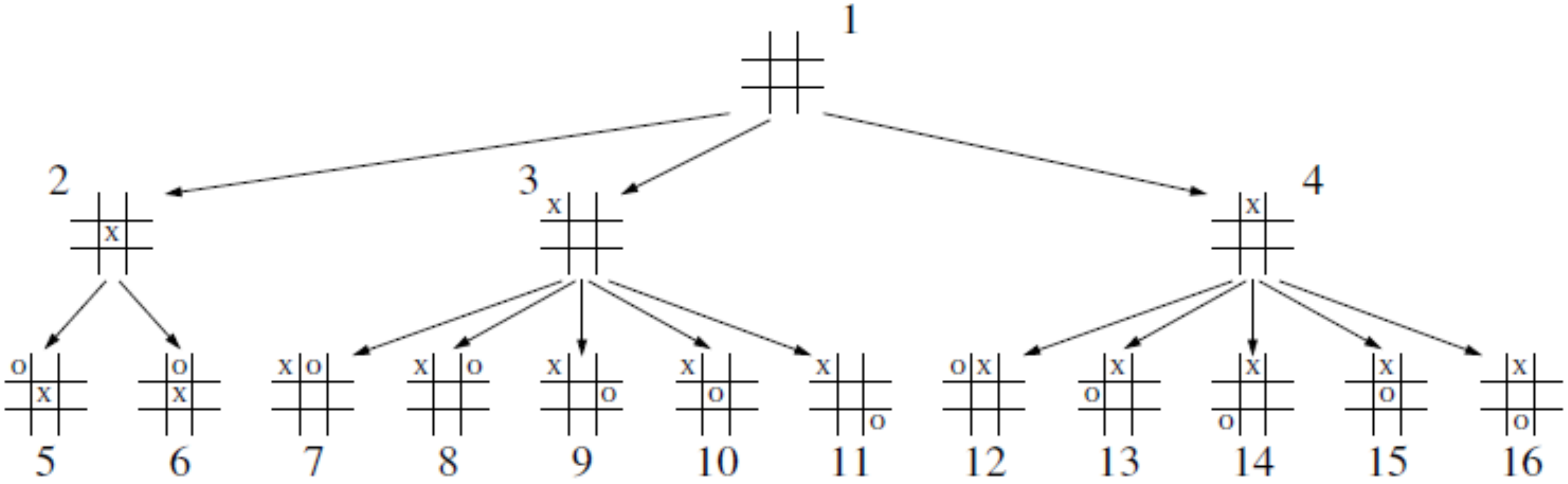
Breadth first search (BFS)

- **Intuition of BFS**
  - Given a source root s, always visit nodes that are closer to the source s first before visiting the others

- The result may not be unique, if we do not define an order among out-going edges from a node
  - Possible results
    - $v_1, v_2, v_3, v_4, v_5, v_6, v_7$
    - $v_1, v_3, v_2, v_7, v_6, v_5, v_4$
  - we could impose an order for children (from left to right)
    - $v_1, v_2, v_3, v_4, v_5, v_6, v_7$ (now become unique)



Example: finding the best move in a game



The code of BFS over a binary tree

```
def BFSearch(t):

    q = ListQueue()
    q.enqueue(t)

    while q.is_empty() is False:
        cNode = q.dequeue()
        if cNode.left is not None:
            q.enqueue(cNode.left)
        if cNode.right is not None:
            q.enqueue(cNode.right)
        print(cNode.element)
```

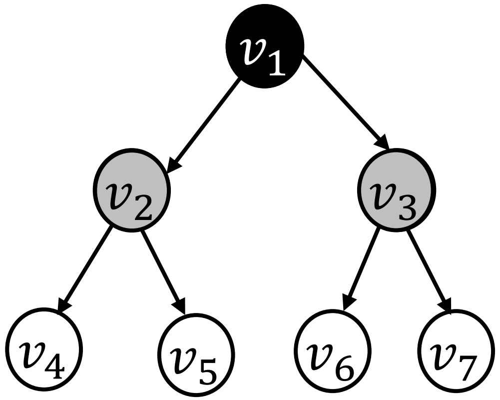
BFS procedure

- At the beginning, color all nodes to be white
- Create a queue Q, enqueue the root
- Repeat the following until queue Q is empty
  - Dequeue from Q, let the node be v
  - Enqueue children of v into Q
  - Visit v
- **Example**:
  - Assume the source is v_1

$Q = (v_1)$

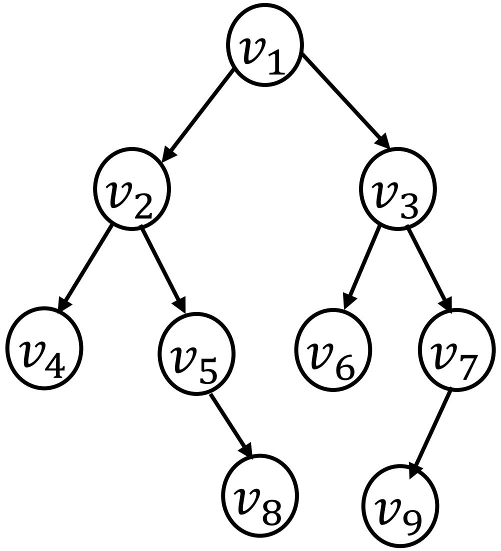
After dequeuing v_1

$Q = (v_2, v_3)$



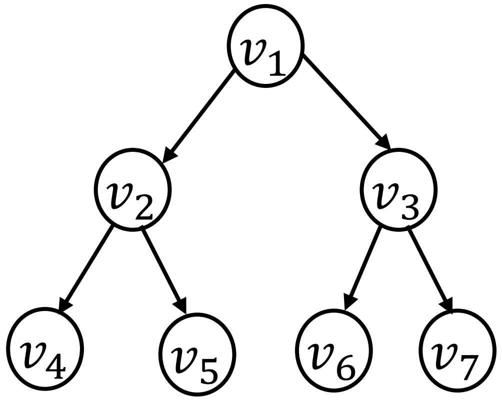
Practice

- Walk through BST for this given tree

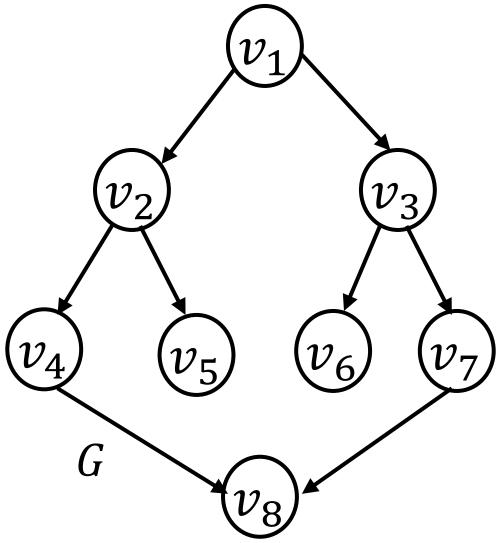


Think about a tree "with a circle"

DFS and BFS work for general graphs



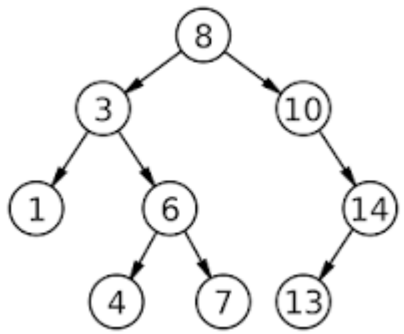
Tree

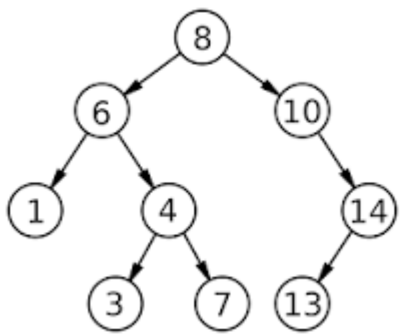


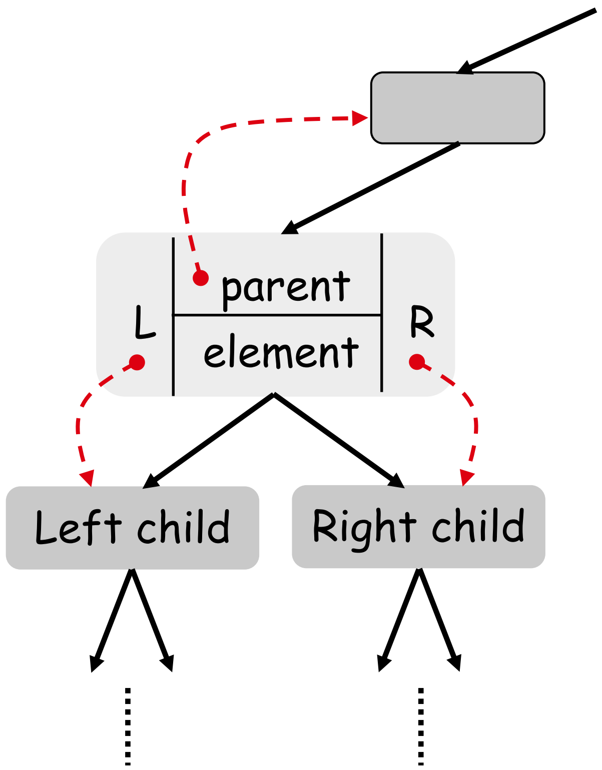
Graph

Binary search tree (optional)

- BST is a tree such that for each node T,
  - 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







BST (Optional)

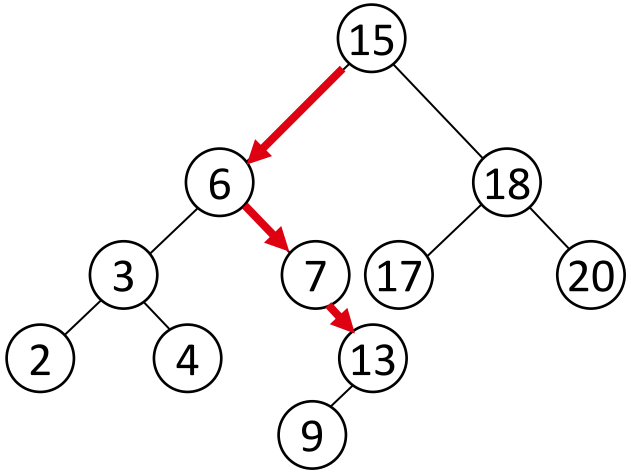
- Support many dynamic set operations
  - searchKey, findMin, findMax, successor, insert,

- Running time of basic operations on BST
  - On average: Θ(logn)
    - The expected height of the tree is log n
  - In the worst case: Θ(n)
    - The tree is a linear chain of n nodes

Example: Searching for a Key

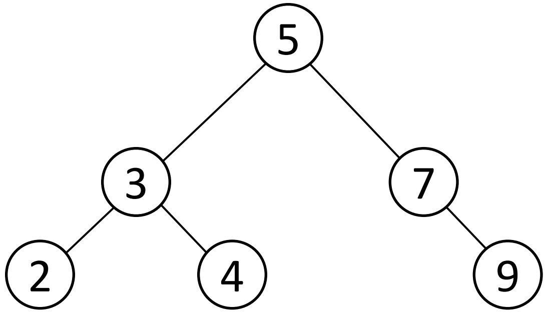
- Given a pointer to the root of a tree and a key k:
  - Return a pointer to a node with key k if one exists, otherwise return None

- Example



- Search for key 13:
  - 15 → 6 → 7 → 13

Example: Searching for a Key




```
find(x, k):
    if x is None or k is key [x]
        then return x
    if k < key [x]
        then return find(left [x], k )
        else return find(right [x], k )
```

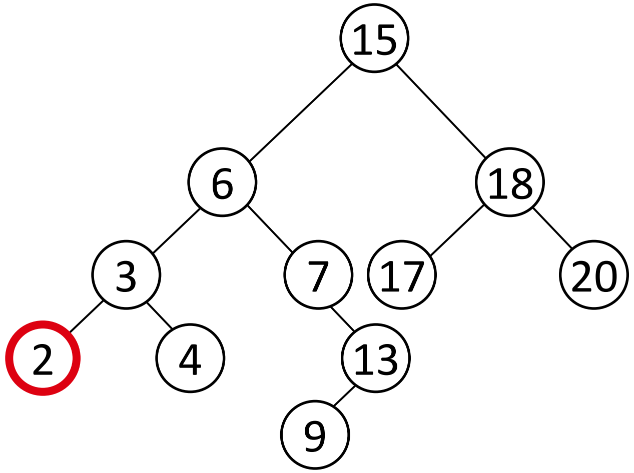
Running Time: O (h),
h is the height of the tree

Example: Finding the Minimum

- Goal: find the minimum value in a BST
  - Following left child pointers from the root, until a None is encountered

findMin(x)

1. **while** left [x] is not None
2. **do** x ← left [x]
3. **return** x



Minimum = 2

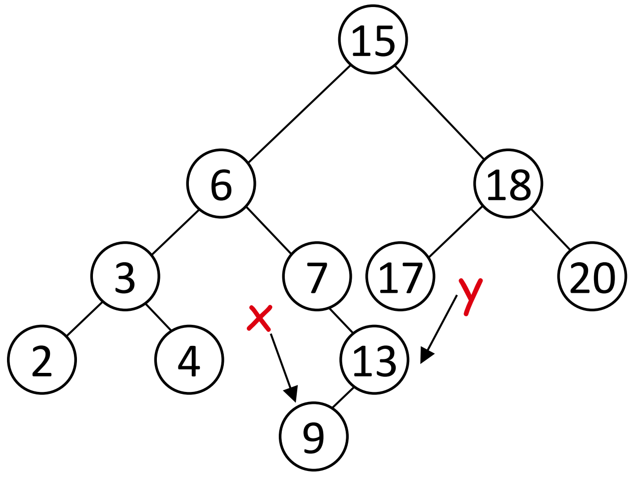
Running time: O(h)

h is the height of tree

Successor

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

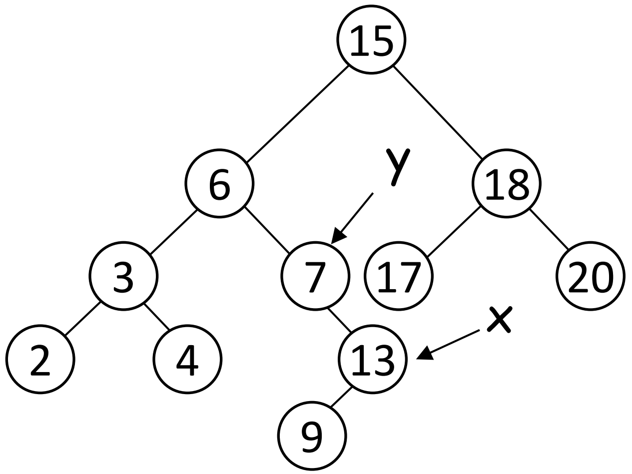
- E.g.: successor (15) = 17
  successor (13) = 15
  successor (9) = 13



- 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

Example: Finding the Successor

```
successor(x)
    if right [x] is not None
      then return findMin(right [x])
    y ← p[x]
    while y is not None and x = right [y]
      do x ← y
         y ← p[y]
    return y
```



Running time: O (h)

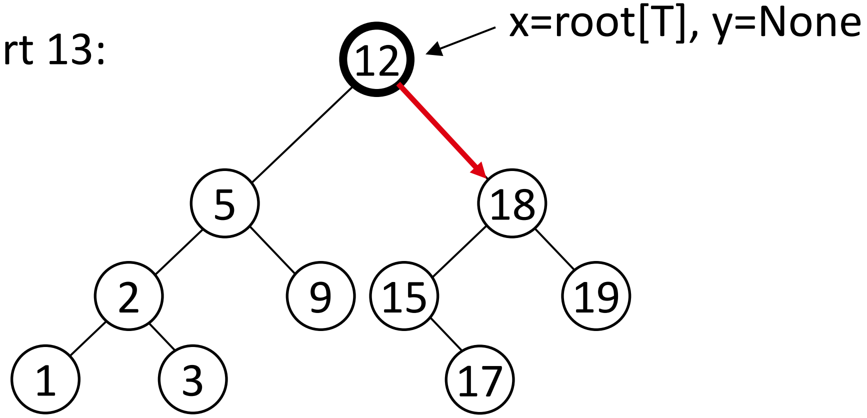
h is the height of the tree

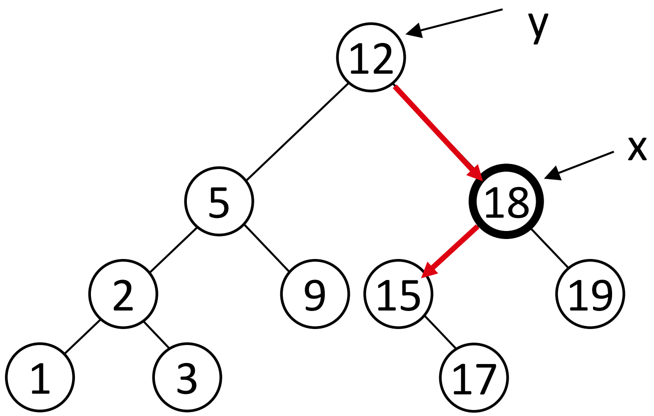
Example: Insertion

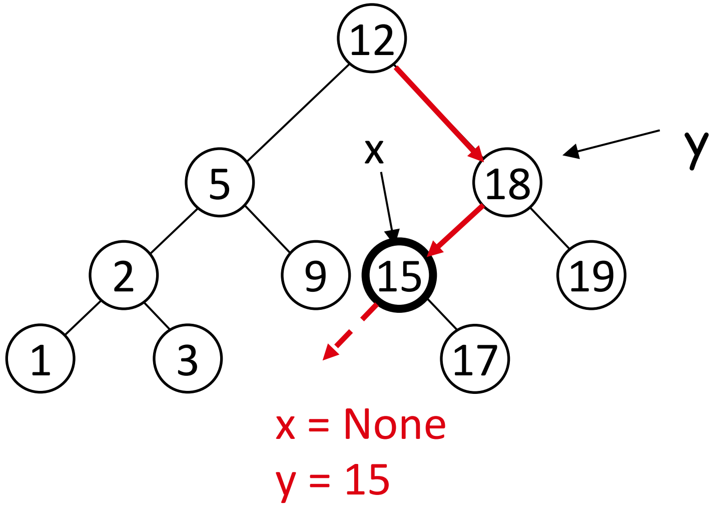
- Goal: Insert value v into a binary search tree
- Find the position and insert as a leaf:
  - If key [x] < v move to the right child of x,
    else move to the left child of x
  - When x is None, we found the correct position
  - If v < key [y] insert the new node as y's left child
    else insert it as y's right child
  - Begin at the root, go down the tree and maintain:
    - Pointer x : traces the downward path (current node)
    - Pointer y : parent of x ("trailing pointer")

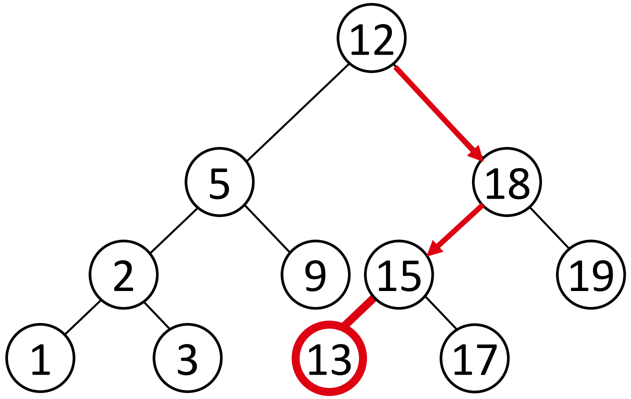
Example

Insert 13:









Exercise 1

- Build a binary search tree for the following sequence

  15, 6, 18, 3, 7, 17, 20, 2, 4