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

Introduction to Computer Science: Programming Methodology

Lecture 11 Tree

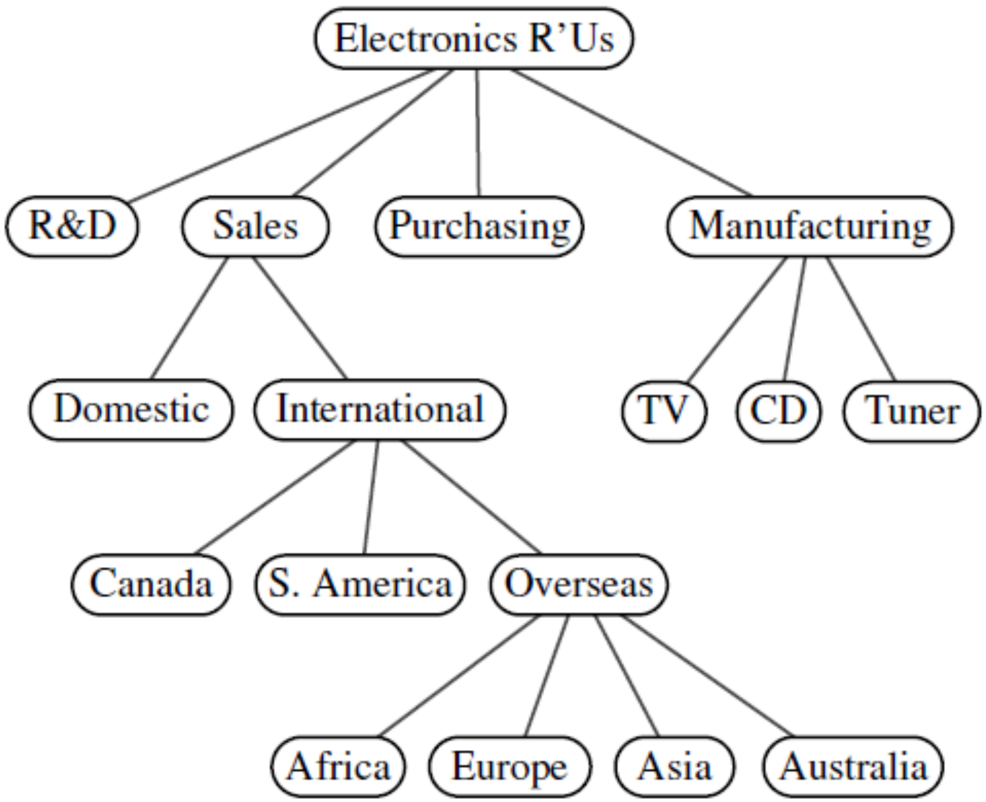
Prof. Yunming XIAO

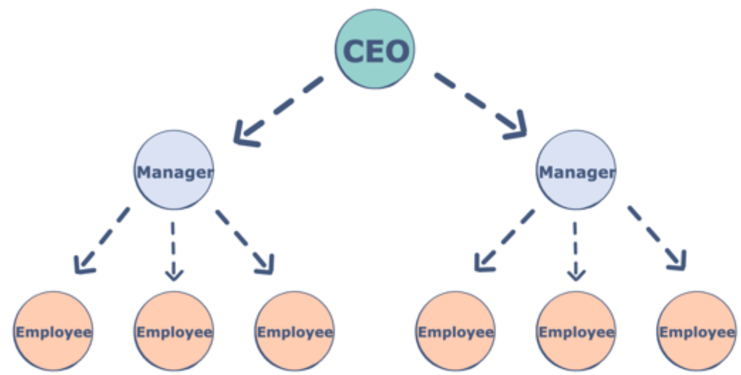
School of Data Science

Tree

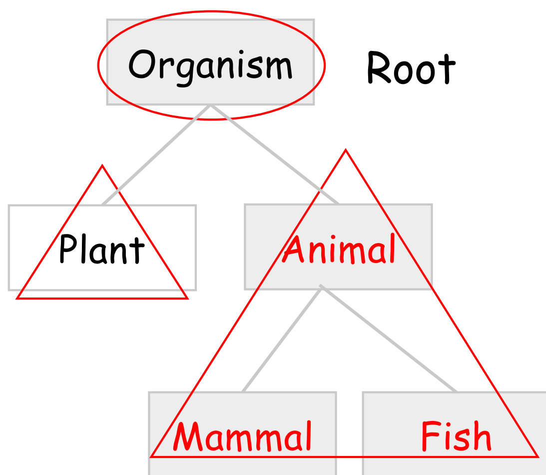
- A tree is a data structure that stores elements hierarchically
- With the exception of the top element, each element in a tree has a parent element and zero or more children elements
- We typically call the top element the root of the tree, but it is drawn as the highest element

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

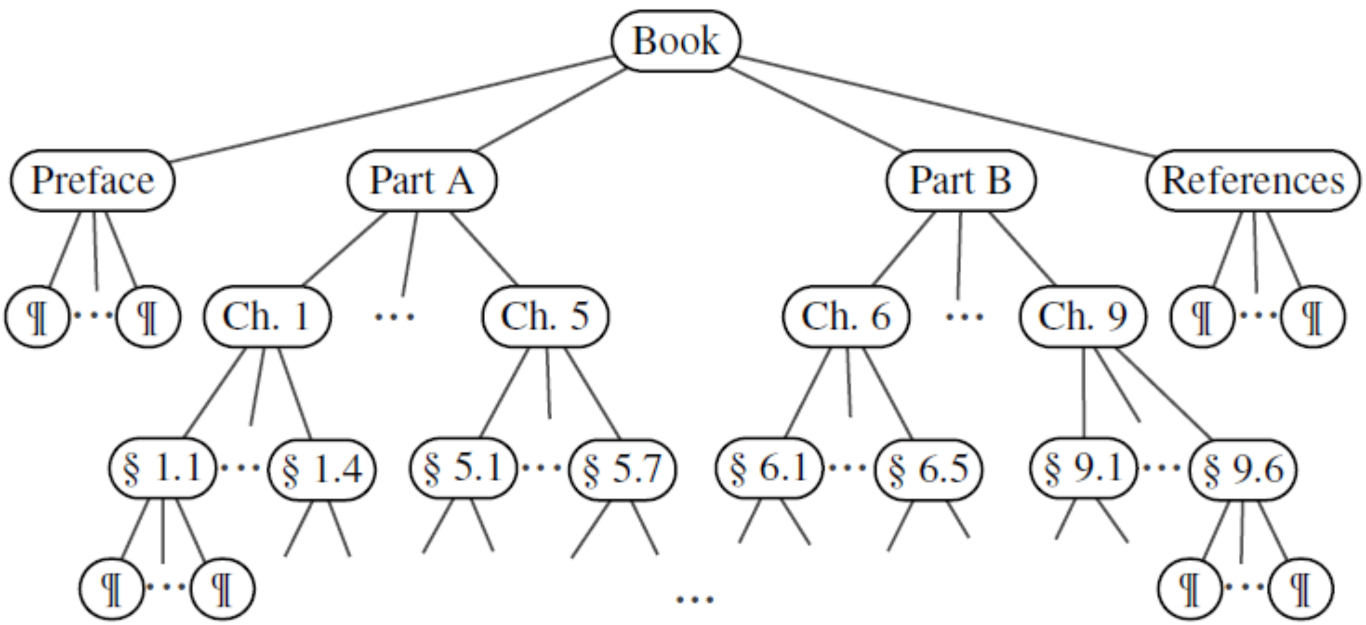
Internal and leaf nodes

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

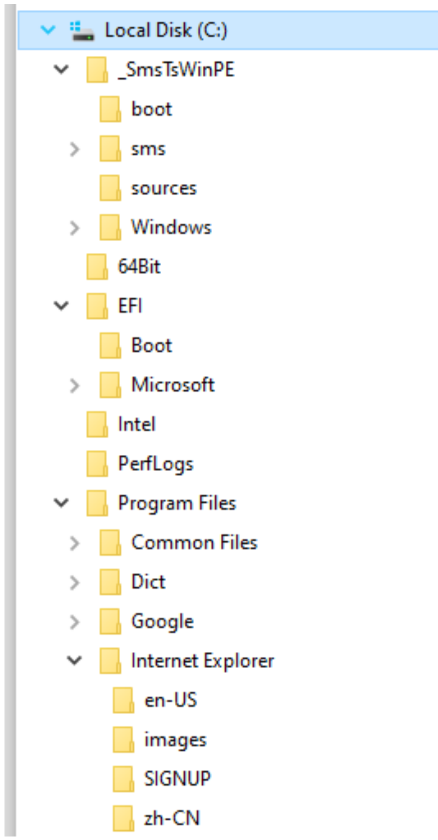
- If a node has at least one child, it is an internal node

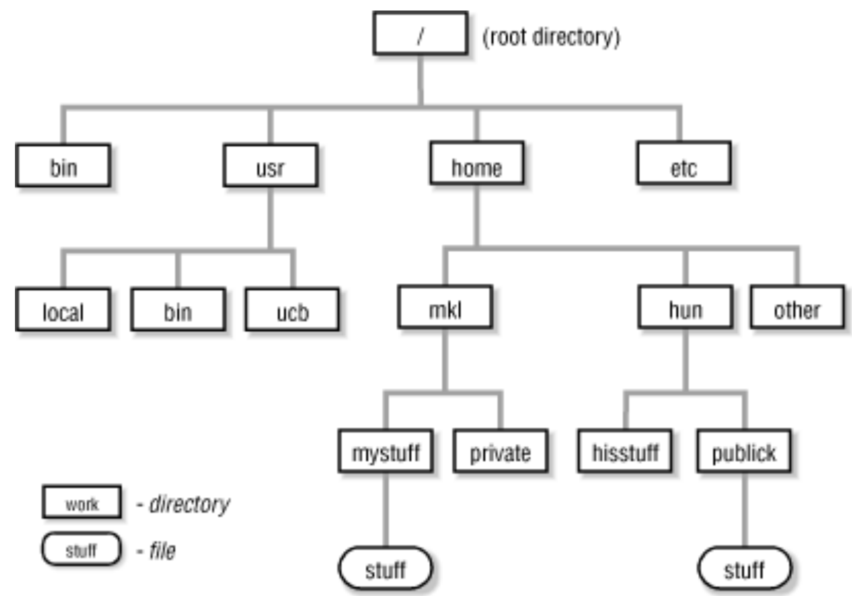
Ordered tree

- 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





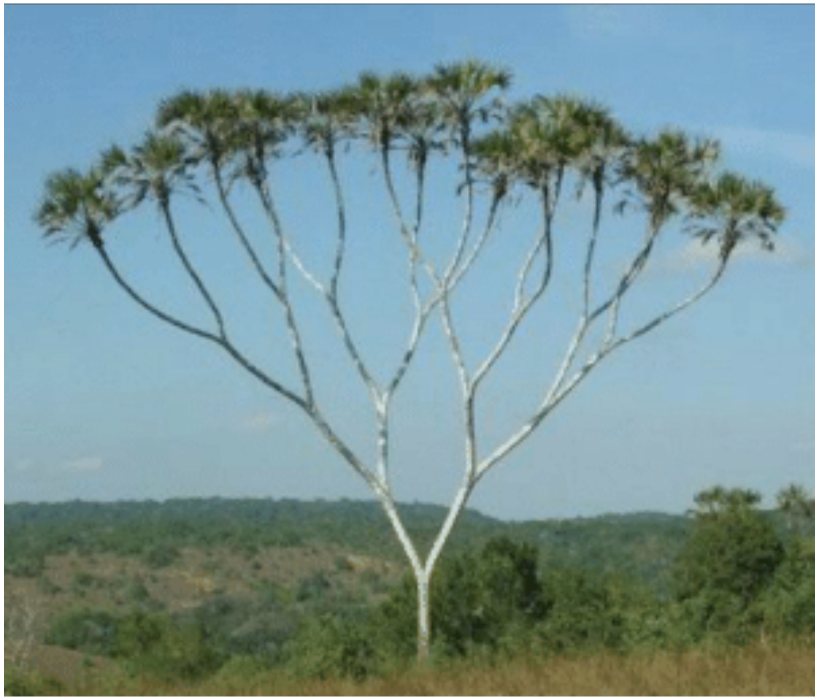
A file is a leaf node and a folder/directory is internal node

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



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 a 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 a 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
```

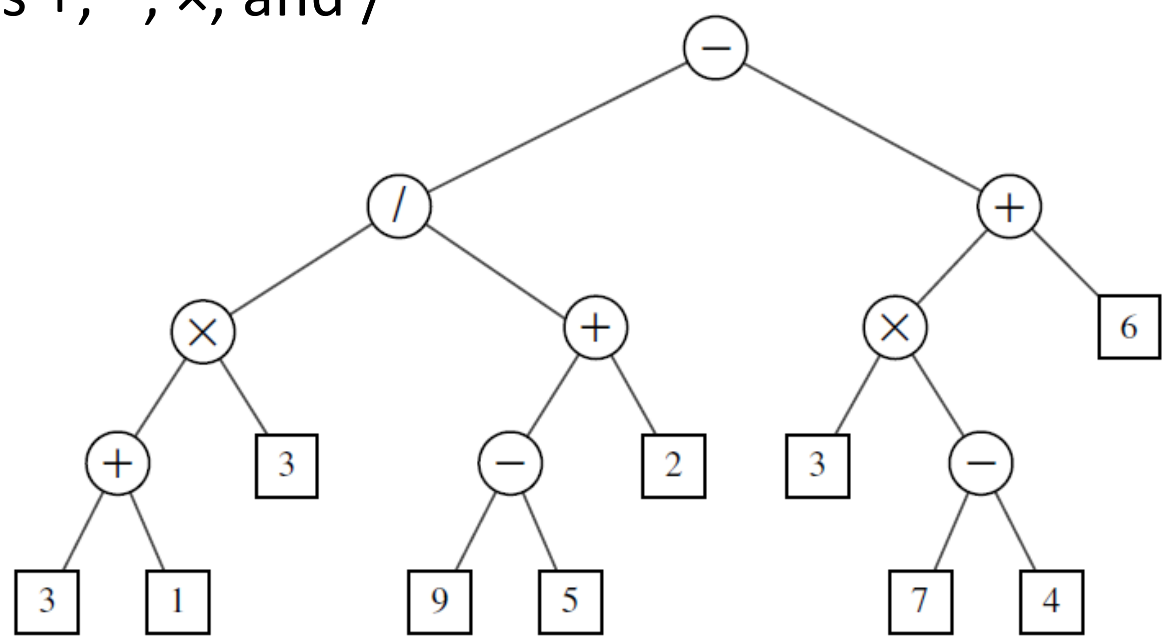
Example: use the binary tree class

```
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
```

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 /



Tree 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 a binary tree

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

preorder traversal

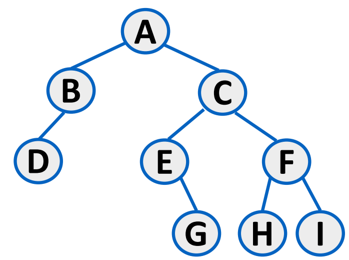
Visit the root

Traverse the left subtree

Traverse the right subtree

A B D C E G F H I

Example:

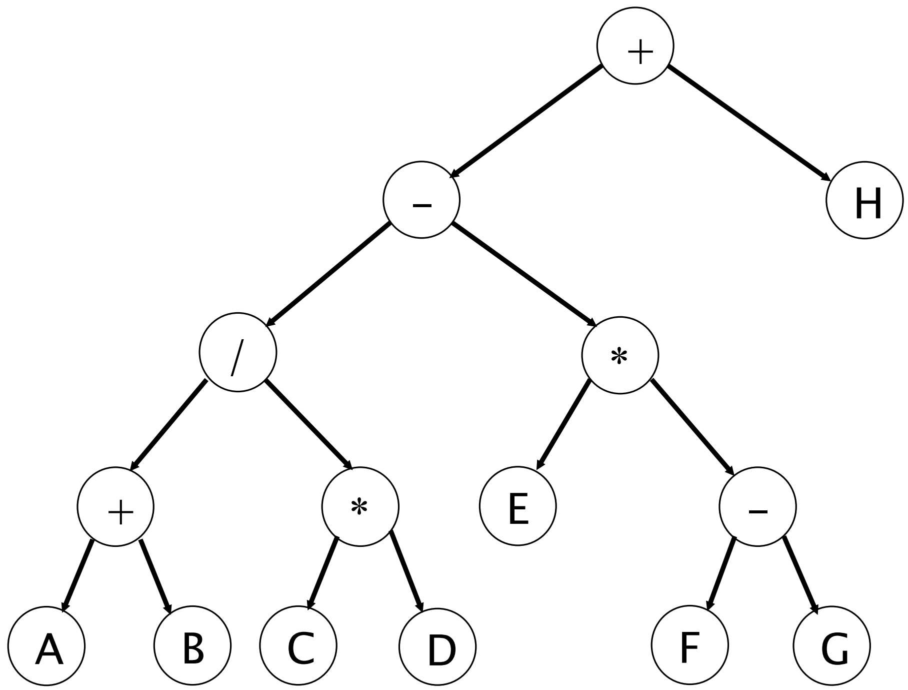


Result:

= 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



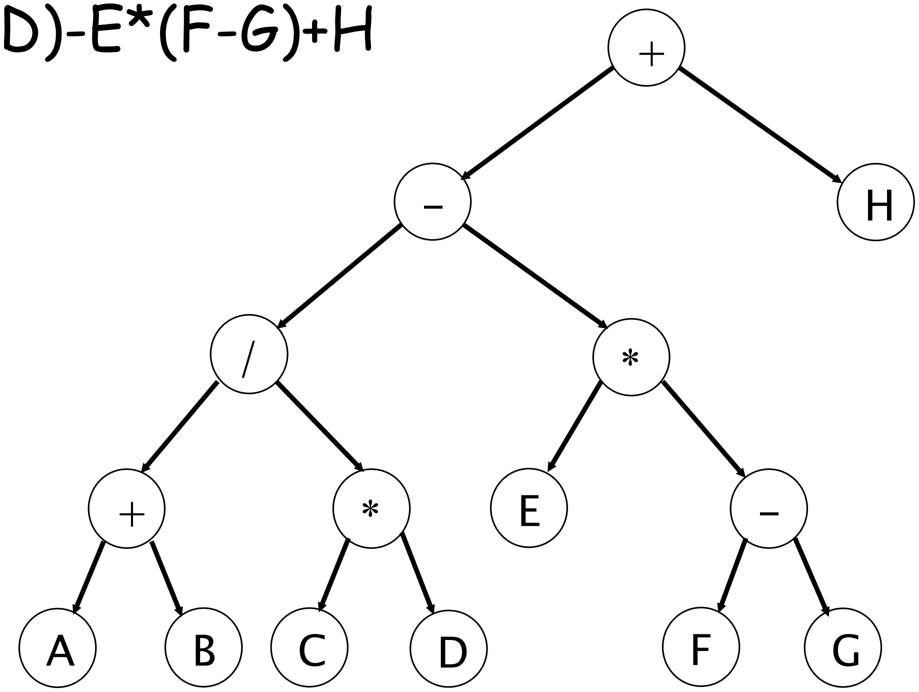
Example

(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+



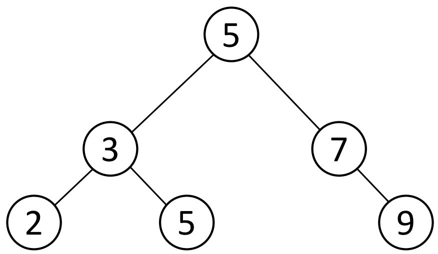
Given an expression, what is the relationship between its postfix and postorder?

Implementation

- INORDER-TREE-WALK(x)

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

- E.g.:

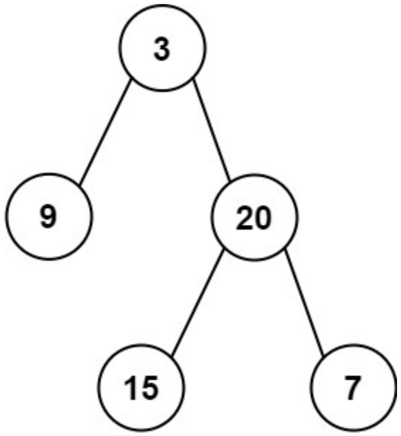


Output: 2 3 5 5 7 9

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

Practice

- 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 Binary Tree from its preorder and Inorder sequences

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

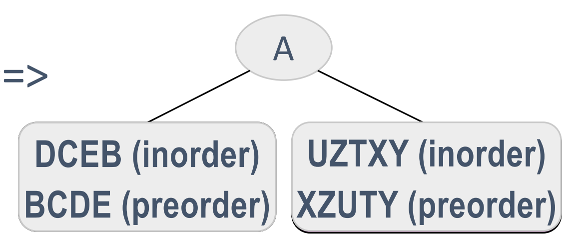
inorder : DCEBAUZTXY

preorder : ABCDEXZUTY

Looking at the whole tree:

- "preorder : **A**BCDEXZUTY"
  ==> A is the root
- Then, "inorder : DCEB**A**UZTXY"

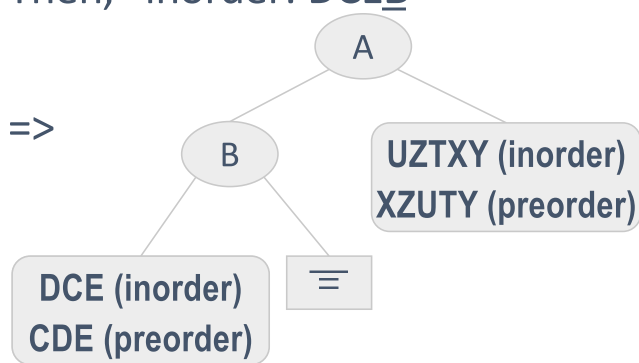
==>



Looking at the left subtree of A:

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

=>



Binary tree reconstruction

Reconstruction of Binary Tree from its preorder and Inorder sequences

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

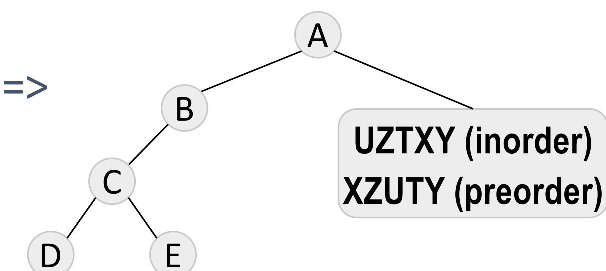
inorder : DCEBAUZTXY

preorder : ABCDEXZUTY

Looking at the left subtree of B:

- "preorder : **C**DE"
  ==> C is the root
- Then, "inorder : D**C**E"

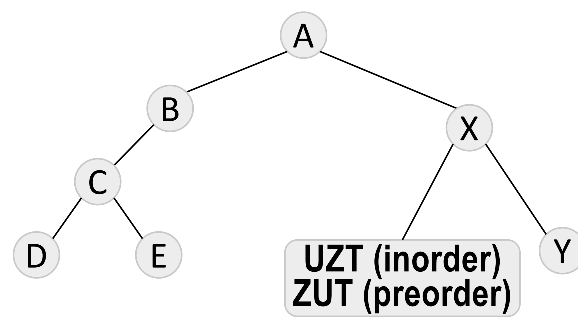
==>



Looking at the left subtree of A:

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

=>



Binary tree reconstruction

Reconstruction of Binary Tree from its preorder and Inorder sequences

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

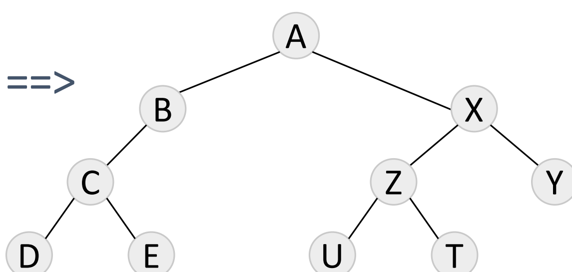
inorder : DCEBAUZTXY

preorder : ABCDEXZUTY

Looking at the left subtree of X:

- “preorder : **Z**UT”
  ==> Z is the root
- Then, “inorder : U<u>**Z**</u>T”

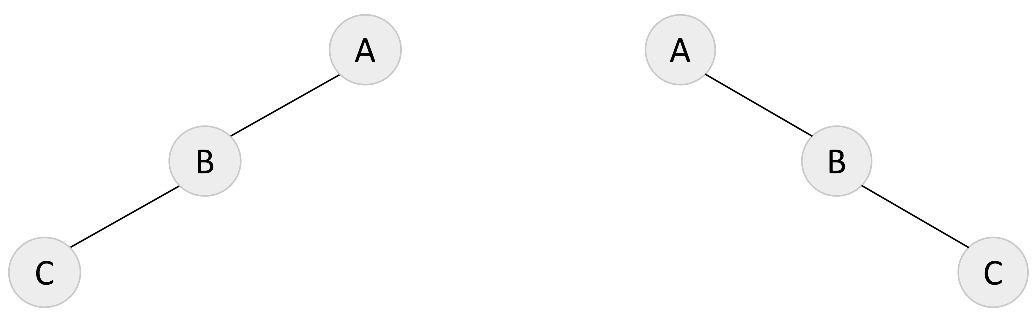
==>



Binary tree reconstruction

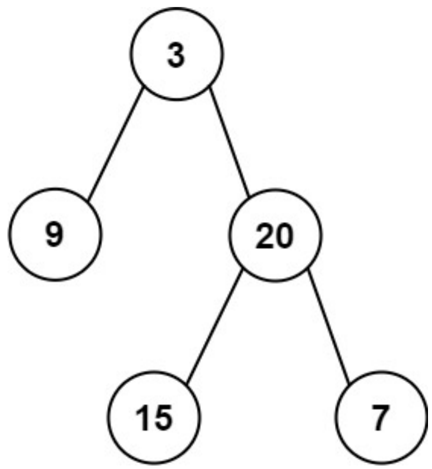
- But, a binary tree may not be uniquely defined by its preorder and postorder sequences

- Example: we can construct 2 different binary trees with
  - Preorder sequence: ABC
  - Postorder sequence: CBA



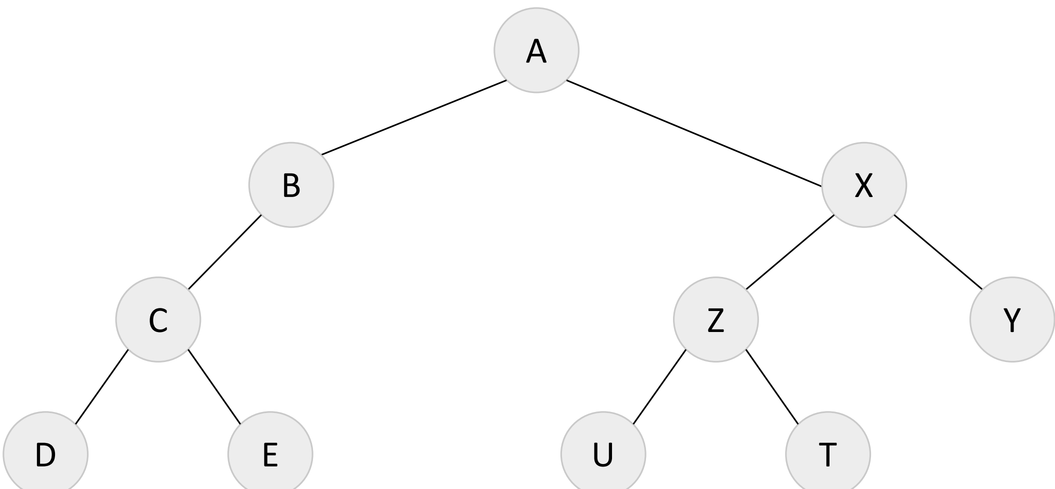
Practice

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



Practice

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



Practice

- Find the maximum number of a binary tree

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

Solution

```
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
```

Solution

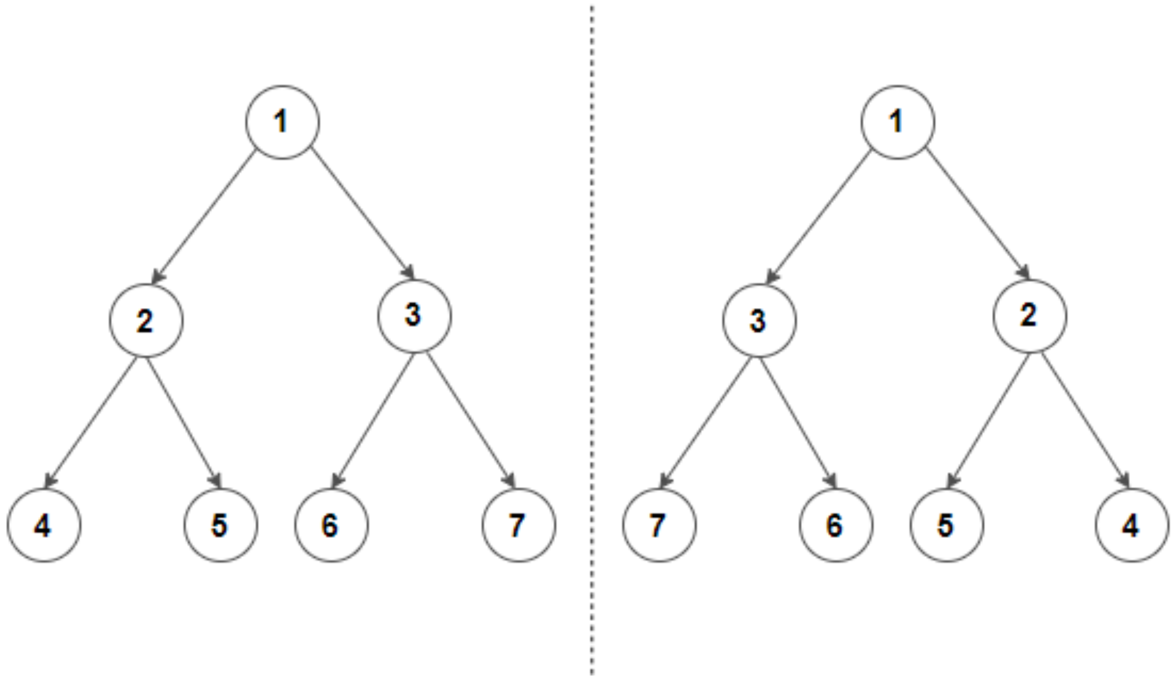
```
def isIdentical(x, y):
    if x is None and y is None:
        return True
    return (x is not None and y is not None) and (x.key == y.key) and \
        isIdentical(x.left, y.left) and isIdentical(x.right, y.right)
```

Solution – non-recursive solution

```
def is_identical(x, y):
    stack_treeA = ListStack()
    stack_treeB = ListStack()
    stack_treeA.push(x); stack_treeB.push(y)
    while not stack_treeA.empty() and not stack_treeB.empty():
        node_treeA = stack_treeA.pop(); node_treeB = stack_treeB.pop()
        if node_treeA.key != node_treeB.key:
            return False
        stack_treeA.push(node_treeA.left); stack_treeA.push(node_treeA.right)
        stack_treeB.push(node_treeB.left); stack_treeB.push(node_treeB.right)

    if stack_treeA.size() != stack_treeB.size():
        return False
    return True
```

Practice

- Swap a tree (convert a binary tree to its mirror)

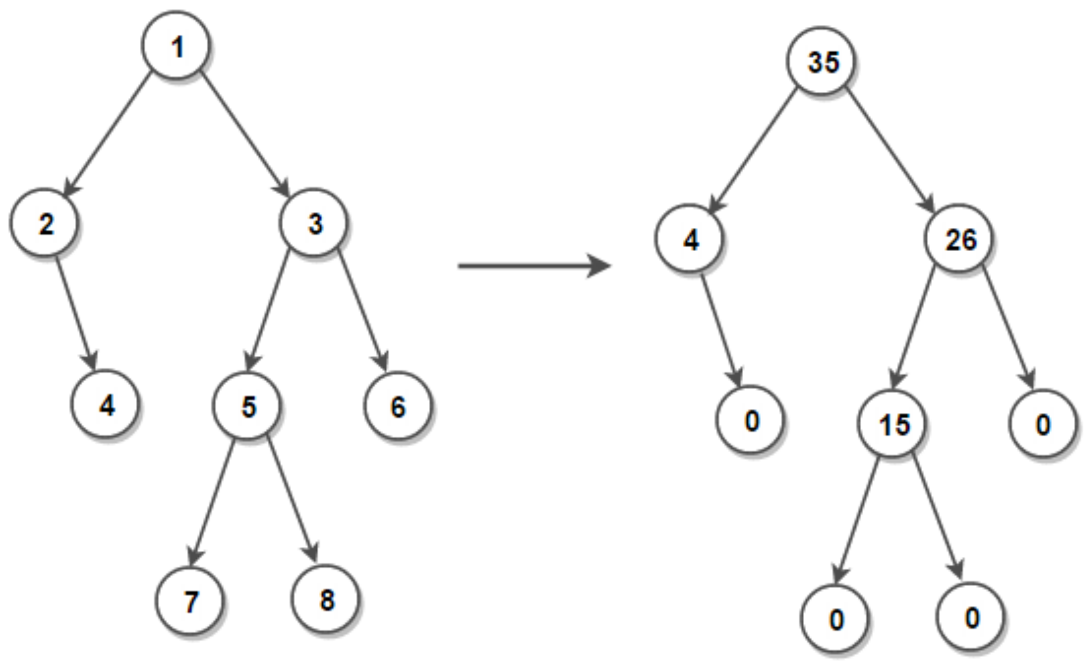


Solution

```
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

- Convert a binary tree to its sum tree

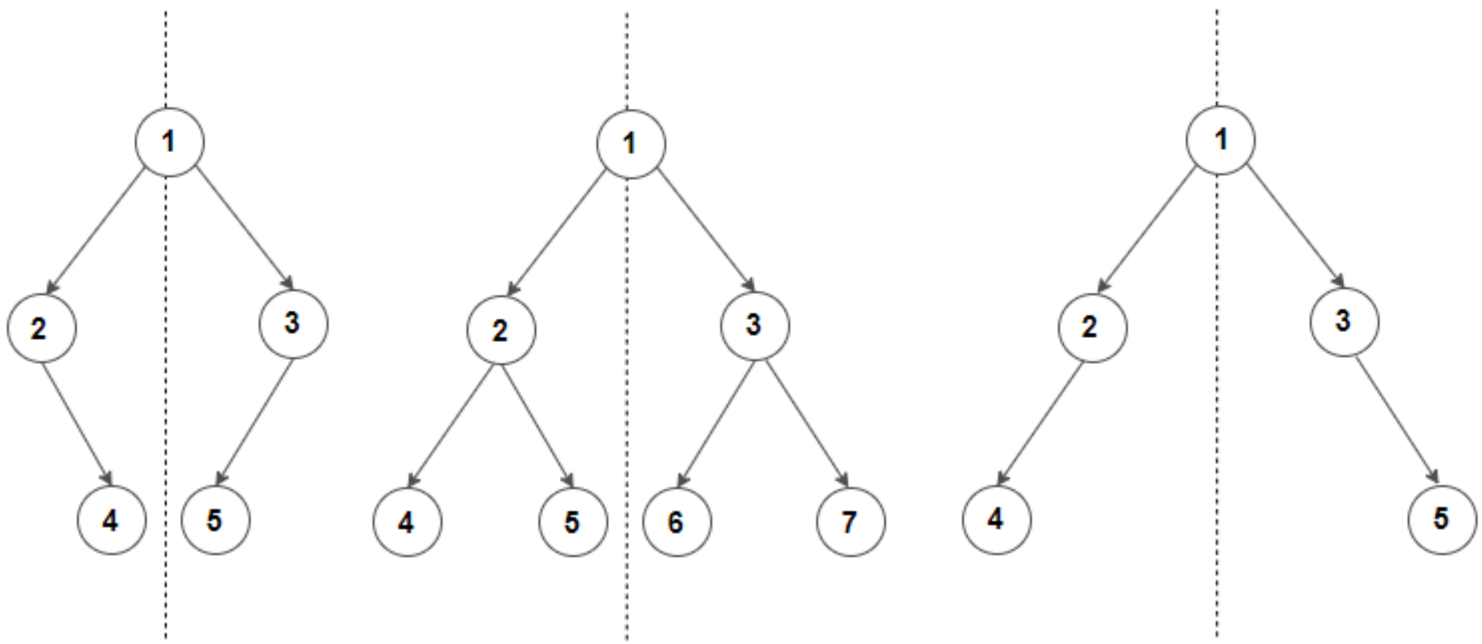


Solution

```
def transform(root):
     if root is None:
        return 0

    Left = transform(root.left)
    right = transform(root.right)

    old = root.data
    root.data = left + right

   return root.data + old
```

Practice

- Check if a binary tree is symmetric



Solution

```
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)
```

More practices

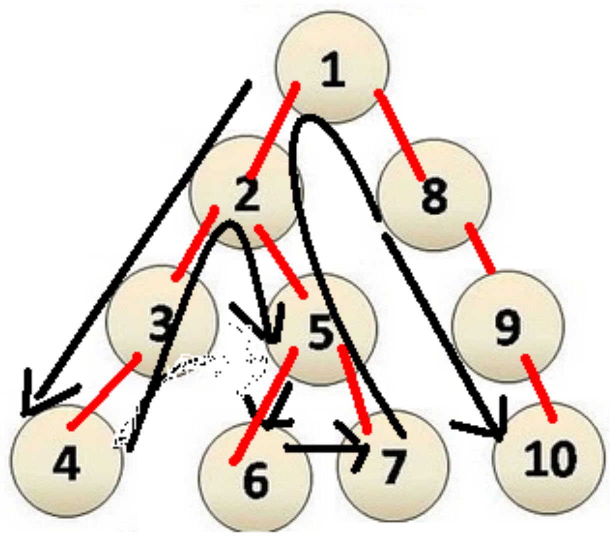
- https://medium.com/techie-delight/binary-tree-interview-questions-and-practice-problems-439df7e5ea1f

- https://practice.geeksforgeeks.org/explore?page=1&category[]=Tree&sortBy=submissions

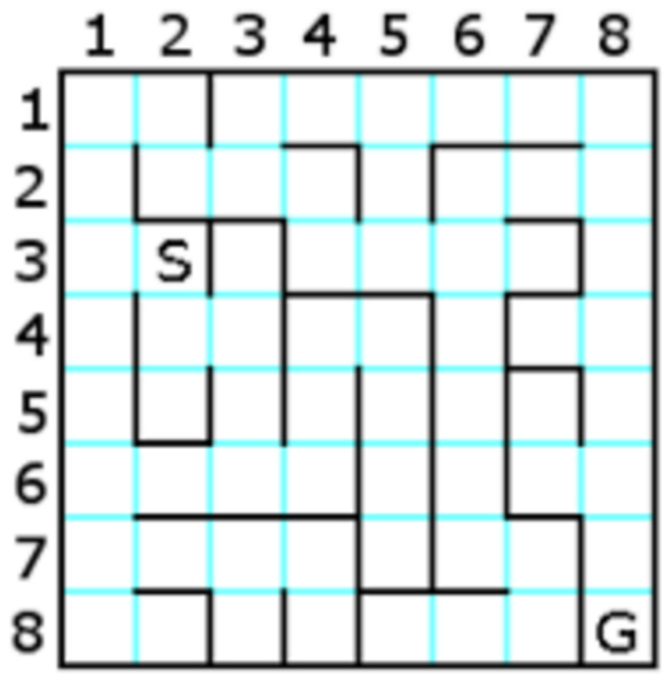
Depth-first search over a tree

- 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

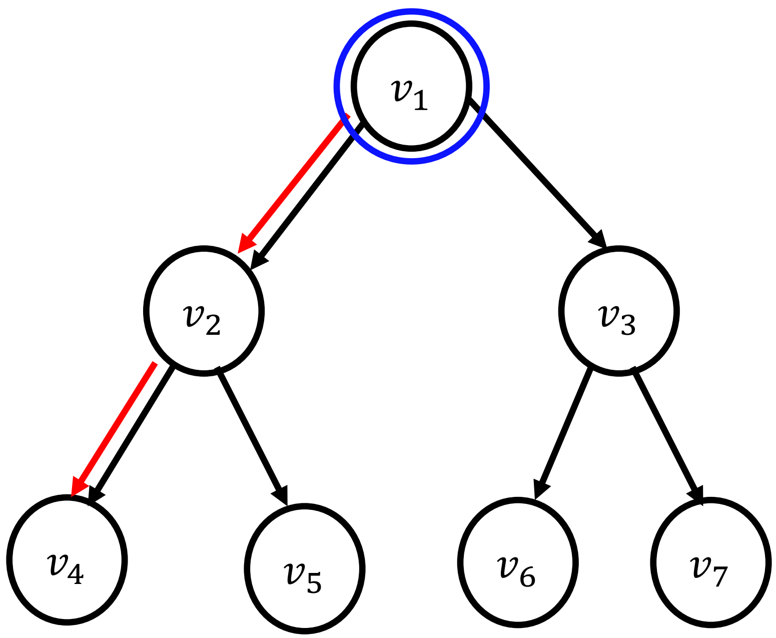


The code of DFS over a binary tree

```
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)
```

Depth-First Search (DFS)

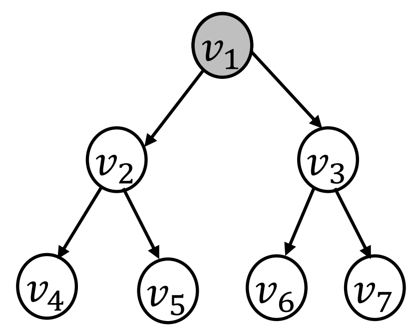
- Going along one path until we cannot go further



Depth-First Search (DFS)

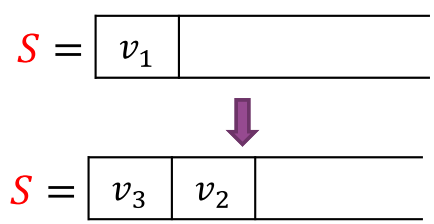
- Initialization:
  - At the beginning, create a stack S, push the root s to S
- Example:
- Assume that v_1 is the root

$S =$	v_1	

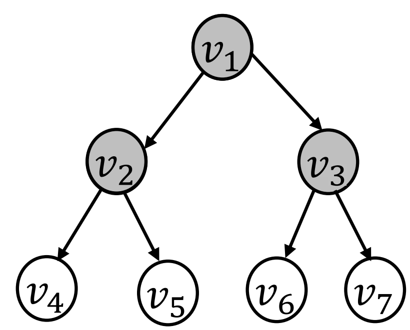


Depth-First Search (DFS)

- Repeat the following until S is empty
  - Pop the top node, denoted as v
  - Push its children to the stack (first push right child and then the left)
  - Visit v



Print v1



Depth-First Search (DFS)

- Repeat the following until S is empty
  - Pop the top node, denoted as v
  - Push its children to the stack (first push right child and then the left)
  - Visit v

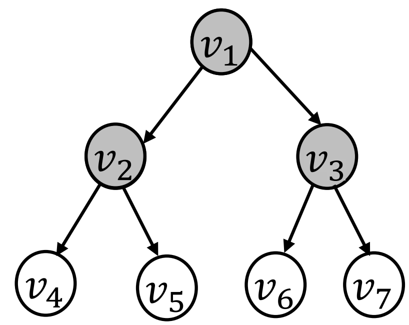
$S = [v_1]$

$S = [v_3, v_2]$ Print v1

$S = [v_3, v_5, v_4]$ Print v2

$S = [v_3, v_5]$ Print v4

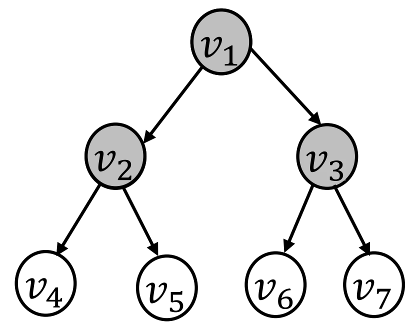
$S = [v_3]$ Print v5



Depth-First Search (DFS)

- Repeat the following until S is empty
  - Pop the top node, denoted as v
  - Push its children to the stack (first push right child and then the left)
  - Visit v

$S =$	v_3			Print v5
$S =$	v_7	v_6		Print v3
$S =$	v_7			Print v6
$S =$				Print v7

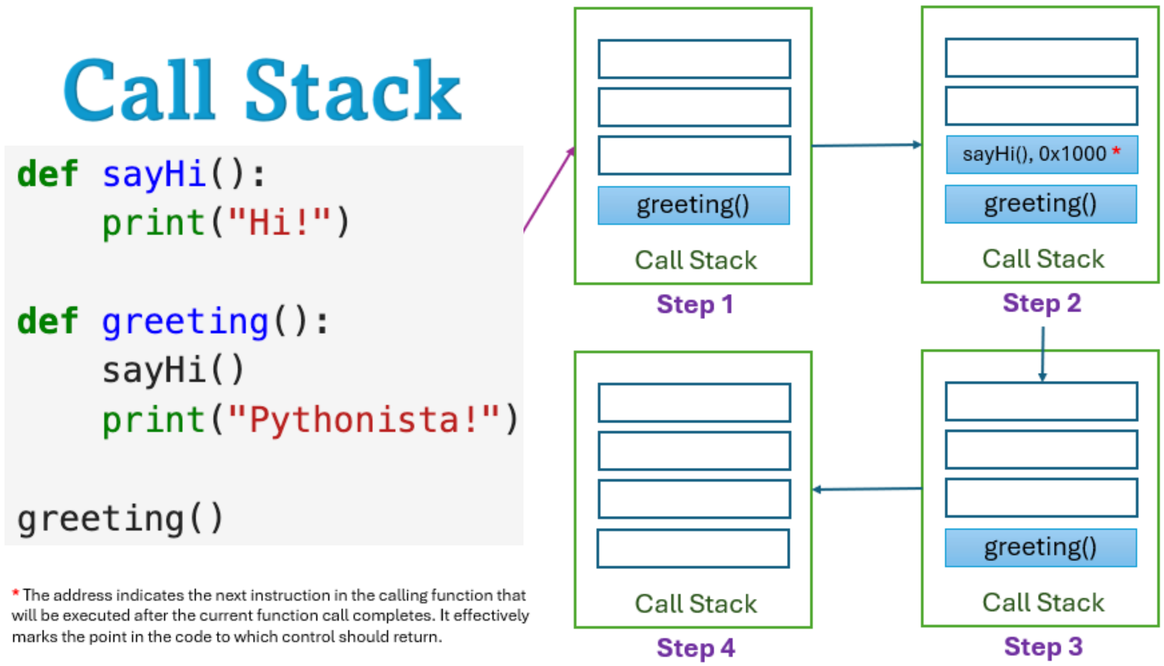


The code of DFS over a binary tree

```python
def dfs_preorder_recursive(node):
    if node is None:
        return

    print(node.element) # visit the node

    dfs_preorder_recursive(node.right)
    dfs_preorder_recursive(node.left)
```

Function calls are stacked (optional)

- Each function call pushes a new frame onto the call stack
- And returning from the function pops that frame



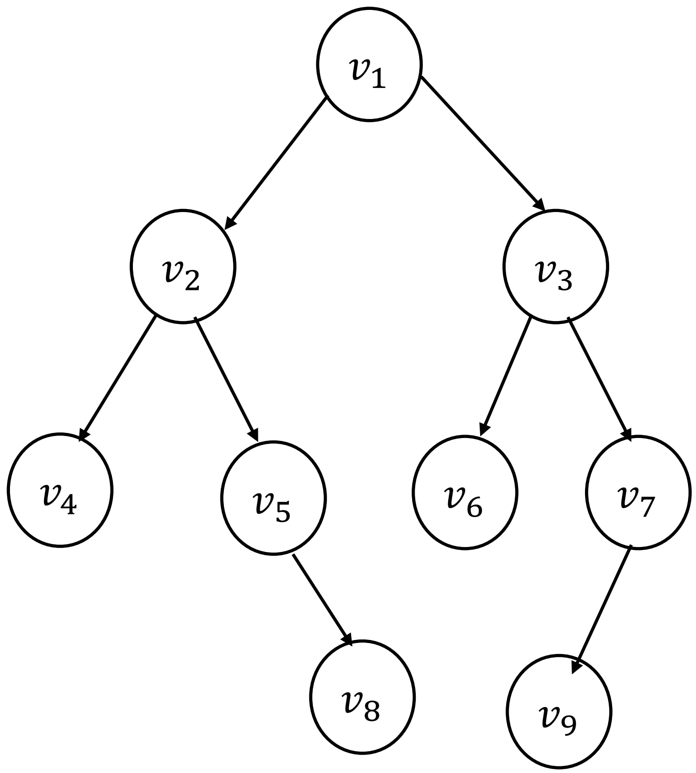
Recursive function vs. non-recursive with stack

```python
def dfs_preorder_recursive(node):
    if node is None:
        return

    print(node.element) # visit the node

    dfs_preorder_recursive(node.right)
    dfs_preorder_recursive(node.left)
```

```python
def dfs_preorder_iterative(root):
    if root is None:
        return

    stack = ListStack()
    stack.push(root)

    while not stack.empty():
        node = stack.pop()
        print(node.element) # visit

        if node.right is not None:
            stack.push(node.right)
        if node.left is not None:
            stack.push(node.left)
```

Practice

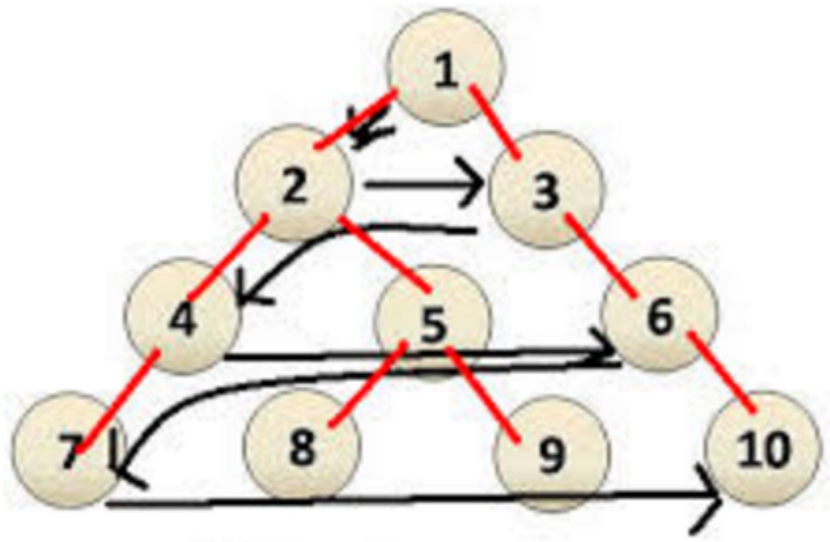
- DFS for the given tree



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

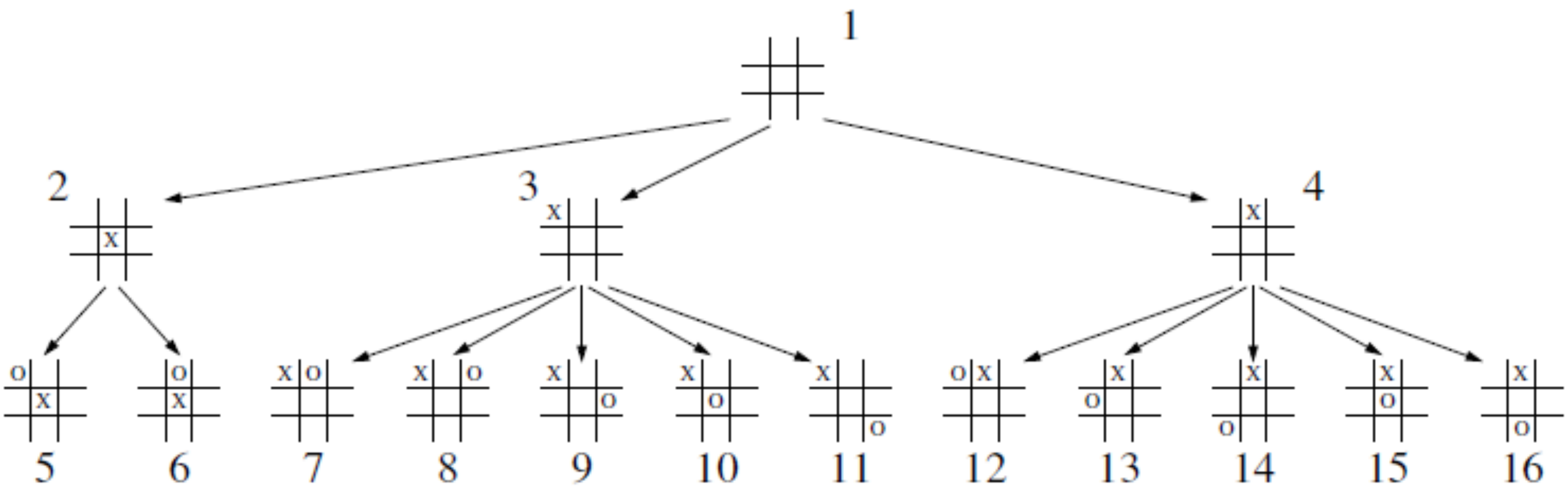


Breadth-first search over a tree

- 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 is not 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: find the best move in a game



The code of BFS over a binary tree

```
def bfs(root):
    if root is None:
        return

    q = ListQueue()
    q.enqueue(root)

    while not q.empty():
        node = q.dequeue()
        print(node.element) # visit

        if node.right is not None:
            q.enqueue(node.right)
        if node.left is not None:
            q.enqueue(node.left)
```

BFS steps

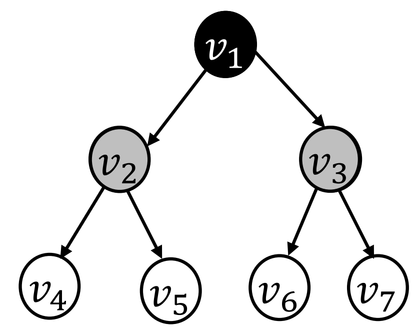
- At the beginning, color all nodes to be white
- Create a queue Q, enqueue the root
- Repeat the following until queue Q is empty
  - Dequeue from Q, let the node be v
  - Visit v
  - Enqueue children of v into Q

- Example:
  - Assume the source is v_1

$Q = (v_1)$

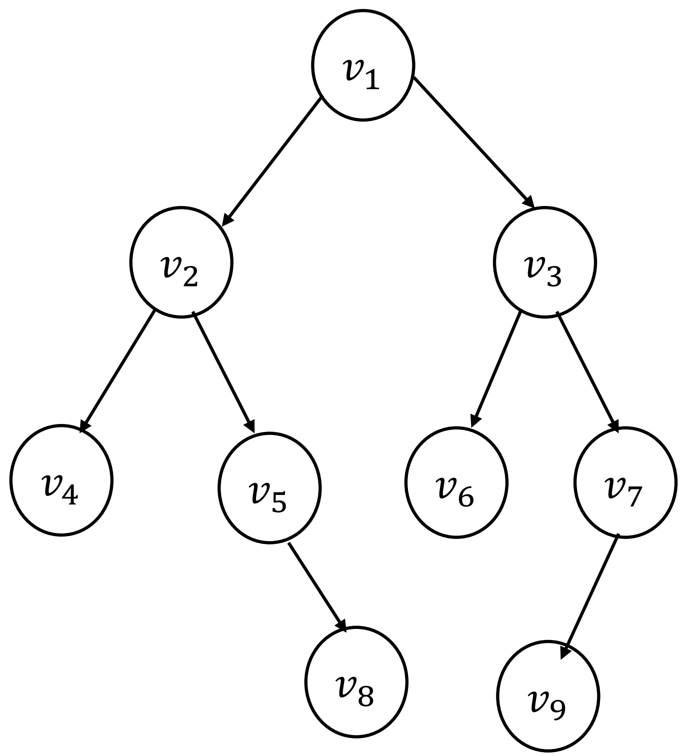
After dequeuing v_1

$Q = (v_2, v_3)$

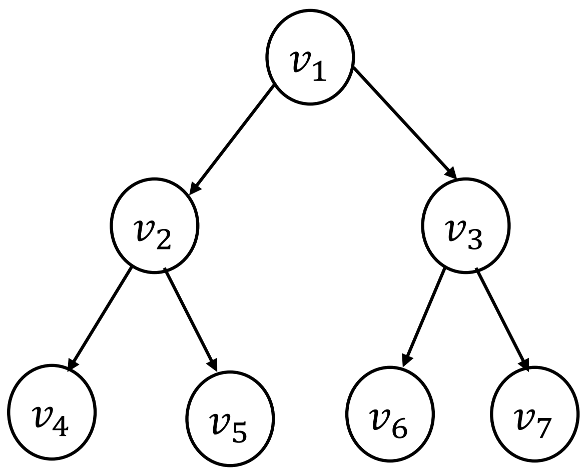


Practice

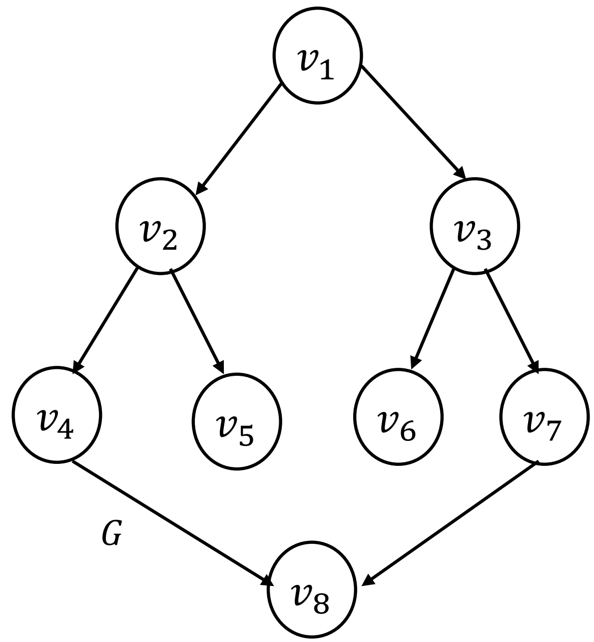
- BFS for the given tree



Think about a tree "with a circle" – graph

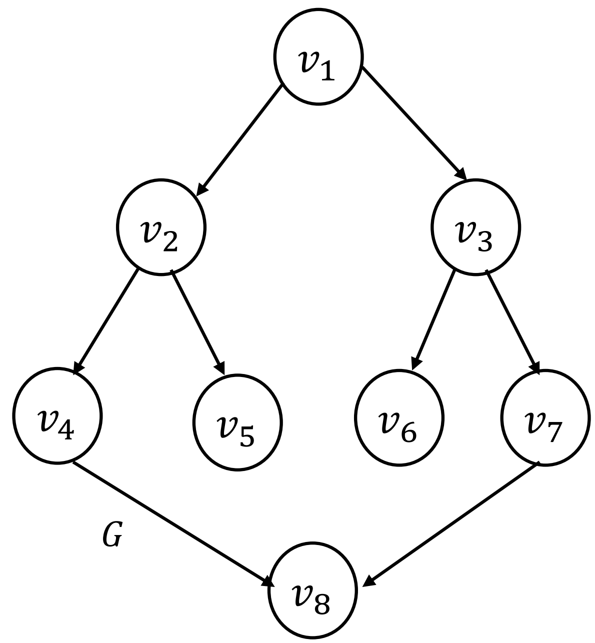


tree



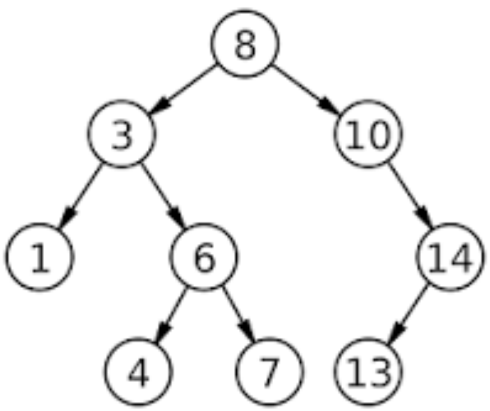
graph

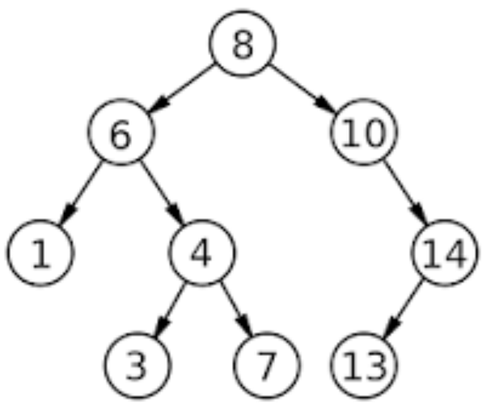
BFS/DFS for a graph

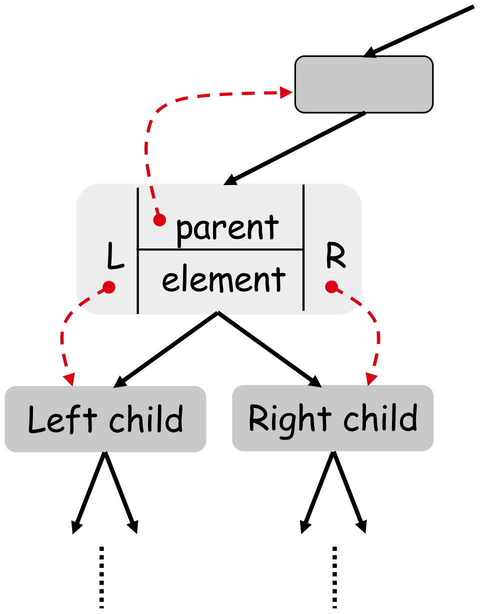


Binary search tree (BST)

- 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







Binary search tree (BST)

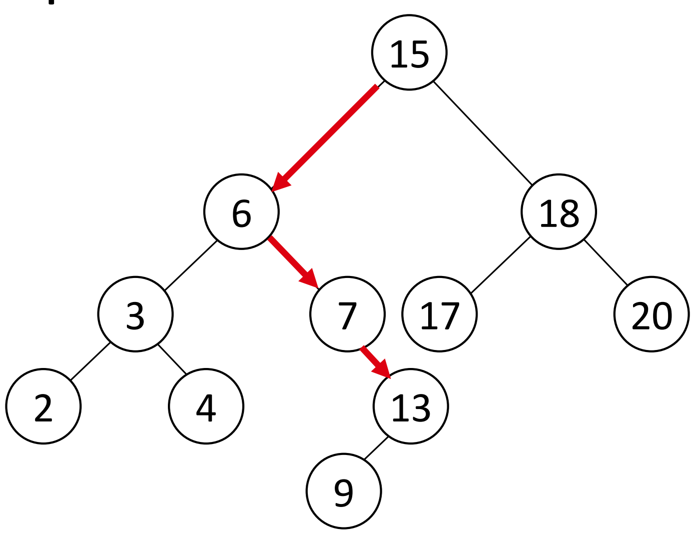
- Support many dynamic set operations
  - searchKey, findMin, findMax, successor, insert,

- Running time of basic operations on BST
  - On average: $\Theta(\log n)$
    - The expected height of the tree is $\log n$
  - 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

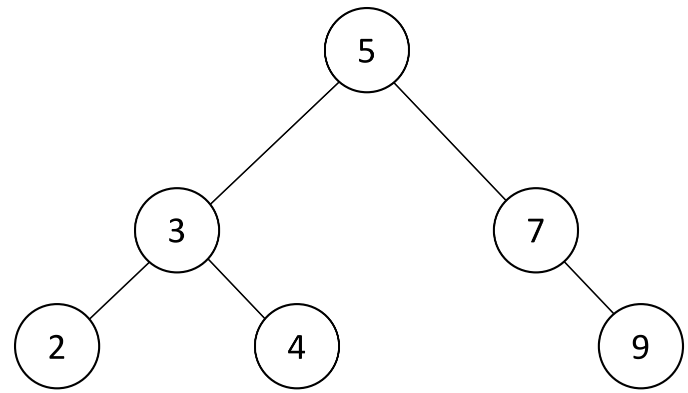
- Example



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

Searching for a key

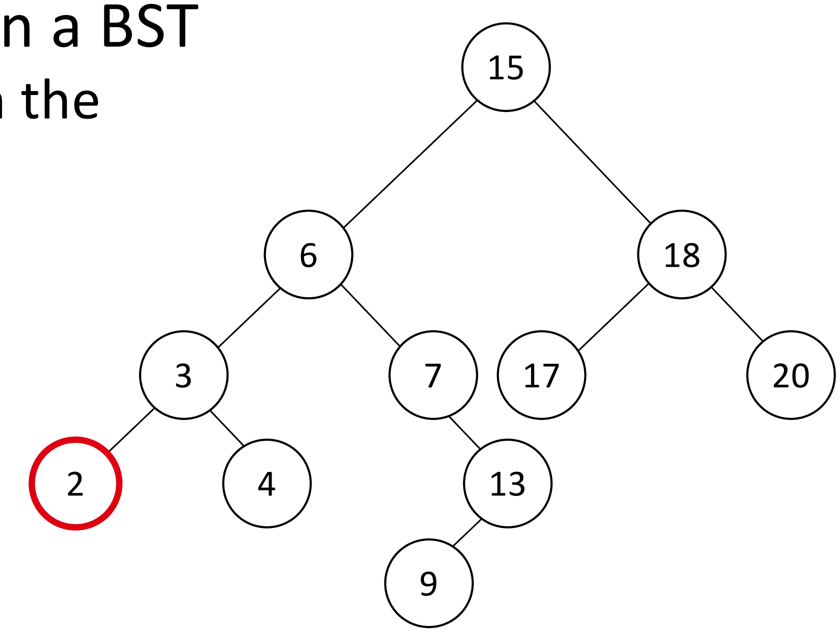
```
find(x, k)
    if x = NIL or k = x.key
        return x
    if k < x.key
        return find(x.left, k)
    else
        return find(x.right, k)
```



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



Minimum = 2

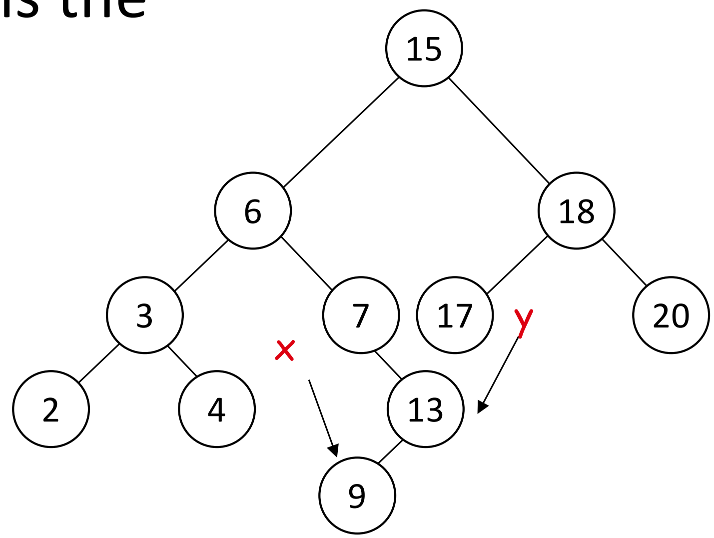
findMin(x)

1. **while x.**left ≠ NIL
2.   x ← x.left
3. **return** x

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

Successor

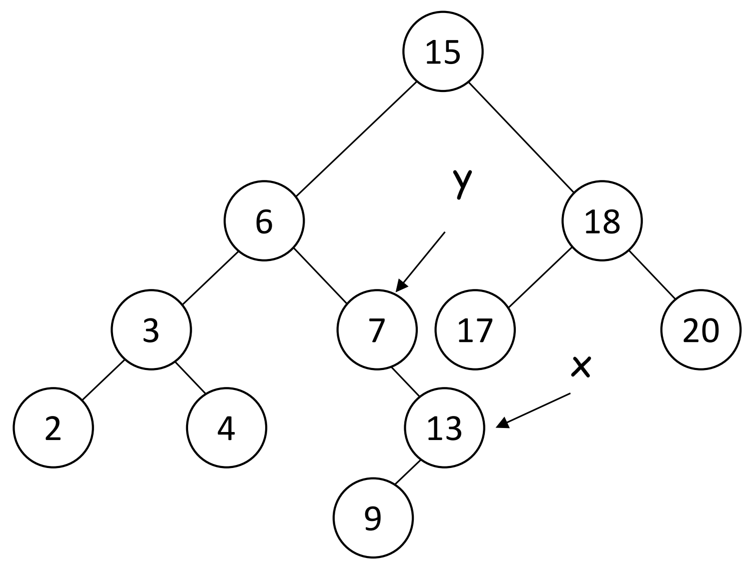
- Def: successor (x) = y, such that y.key is the
  smallest key > x.key
- E.g.: successor (15) = 17
  successor (13) = 15
  successor (9) = 13



- Case 1: x.right is non-empty
  - Successor (x) = the minimum in x.right
- Case 2: x.right 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

Finding the successor

```
successor(x)
    if x.right [x] ≠ NIL
        return findMin(x.right)
    y ← x.parent
    while y ≠ NIL and x = y.right
        x ← y
            y ← y.parent
    return y
```



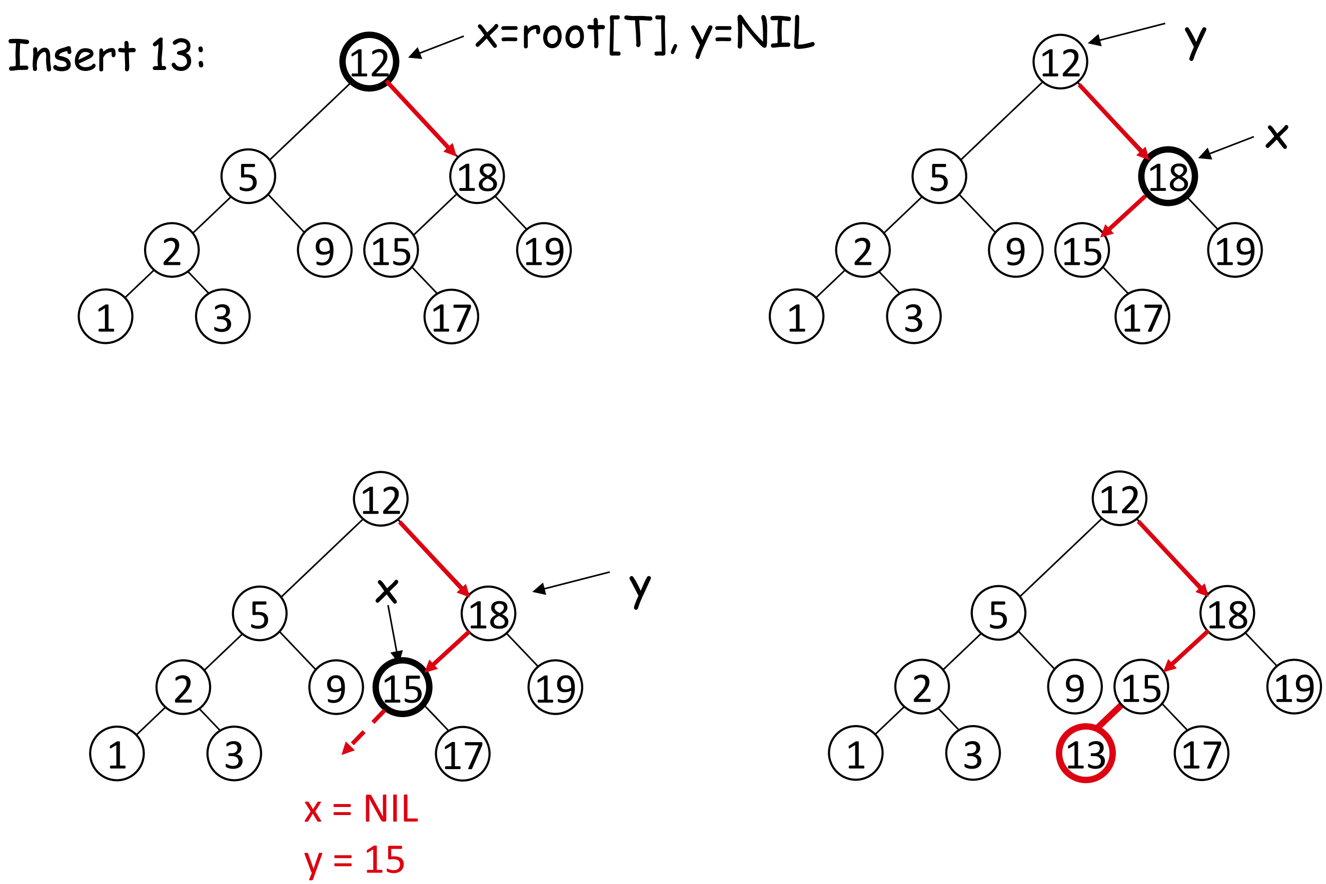
Running Time: O (h),
h is the height of the tree

Insertion

- Goal: Insert value v into a binary search tree

- Find the position and insert as a leaf:
  - If x.key < v move to the right child of x,
  - else move to the left child of x
  - When x is NIL, we found the correct position
  - If v < y.key insert the new node as y's left child
  - else insert it as y's right child
  - Beginning 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



Practice

- Build a binary search tree for the following sequence

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

Course and Teaching Evaluation (CTE)

- Please help to complete the evaluation!

Thanks