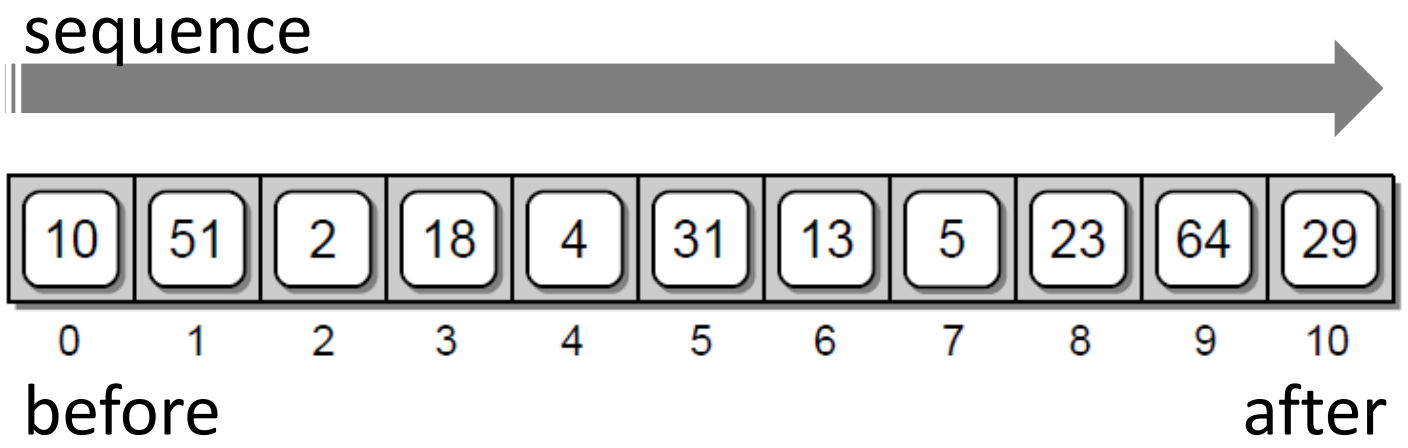
Tree

CPE111: Programming with Data Structures

Reference

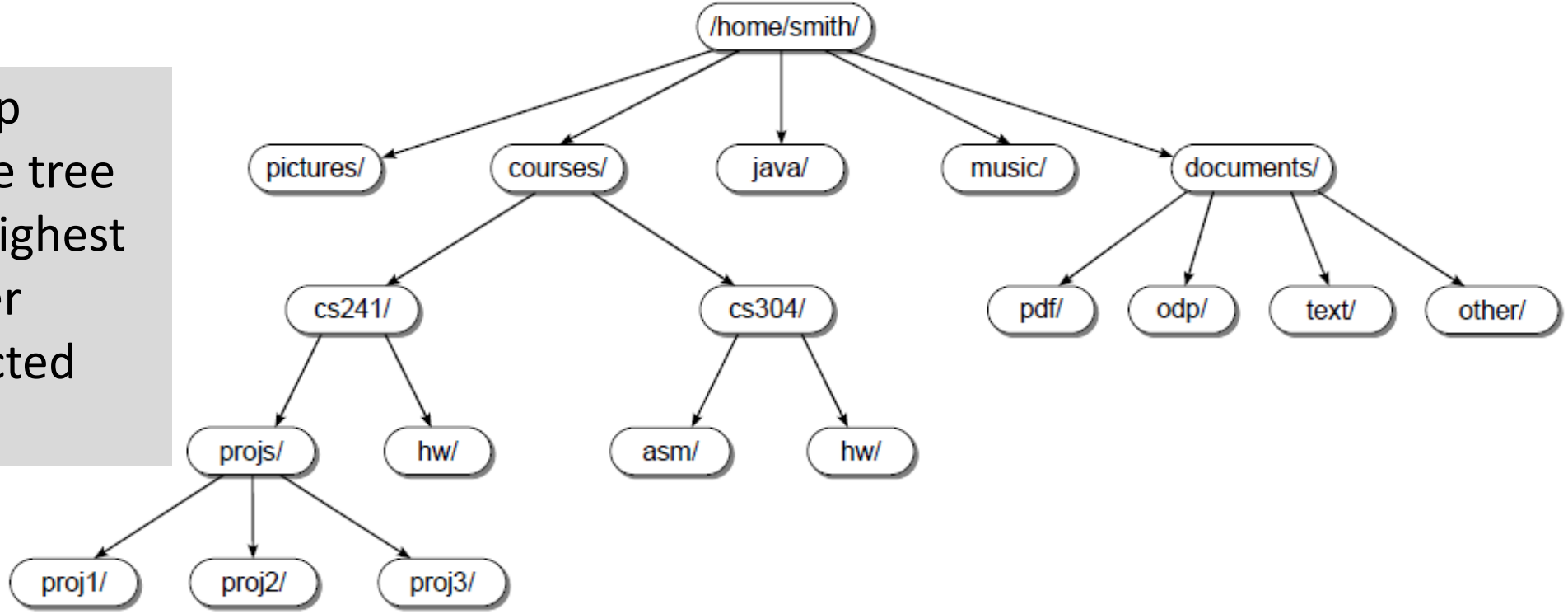
- Michael T.Goodrich, Roberto Tamassia, Michael H. Goodwasser. *Data Structures and Algorithms in python*. *Chapter8, Chapter 11*. John Wiley&Sons,Inc. 2013
- Rance D. Necaise. *Data Structures and algorithms using python*. *Chapter13-14*. John Wiley&Sons,Inc., 2011
- http://interactivepython.org/runestone/static/pythonds/Trees/toctree.html

“Sequential structures” such as the array, Python list, linked list, stacks, and queues. These structures organize data in a linear fashion in which the data elements have a “before” and “after” relationship.

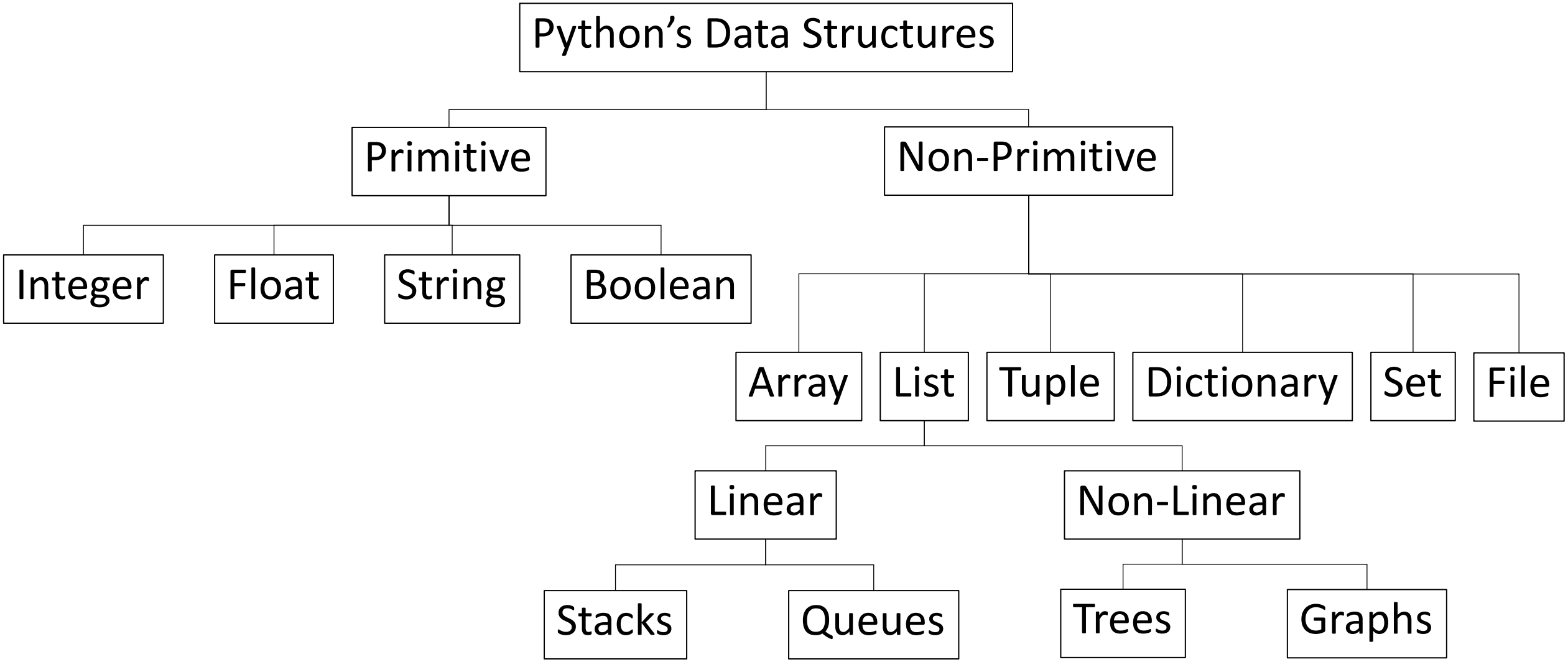


The relationships in a tree are hierarchical, with some objects being "above" and some "below" others. Actually, the main terminology for tree data structures comes from family trees, with the terms "parent," "child," "ancestor," and "descendant" being the most common words used to describe relationships.

We typically call the top element the ***root*** of the tree but it is drawn as the highest element, with the other elements being connected below

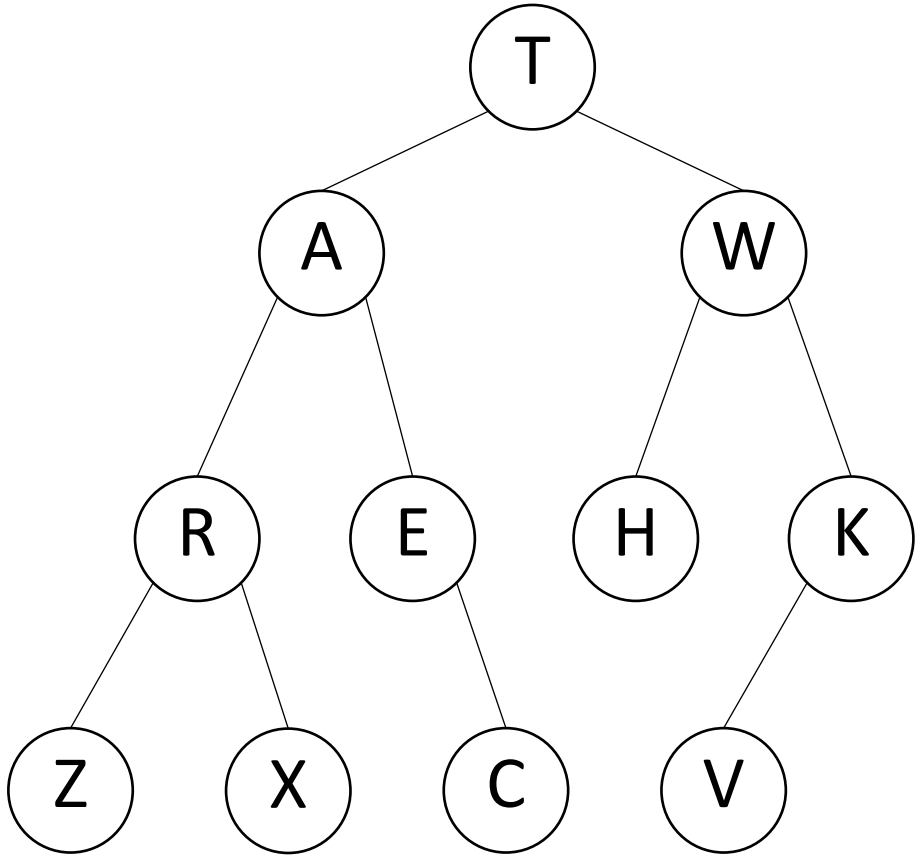


The data elements are stored in nodes and pairs of nodes are connected by edges.



https://www.datacamp.com/community/tutorials/data-structures-python

A tree is usually visualized by placing elements inside ovals or rectangles, and by drawing the connections between parents and children with straight lines.



node edge path

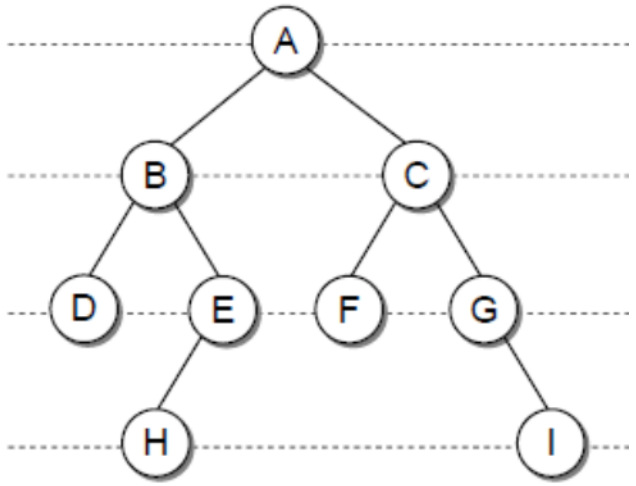
root node subtree

parent child relative

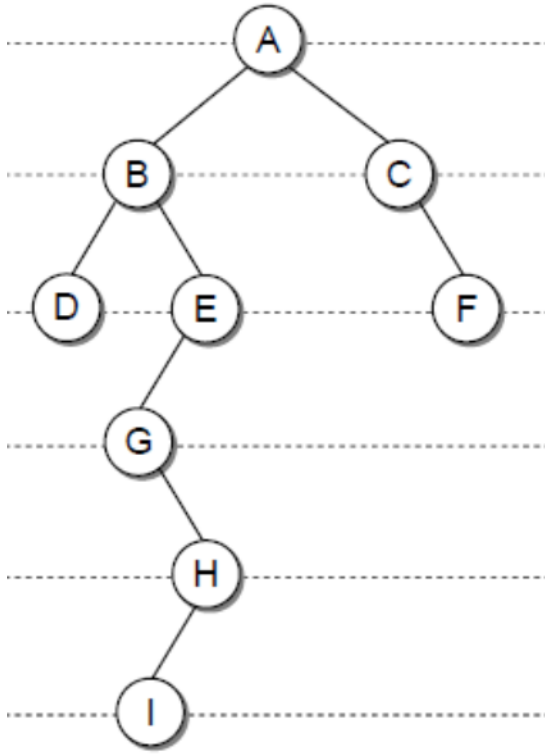
siblings leaf

external node internal node

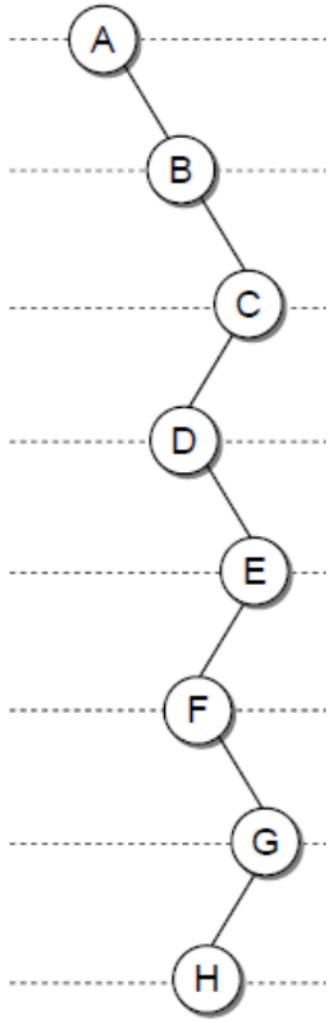
Size of Trees



(a)_____



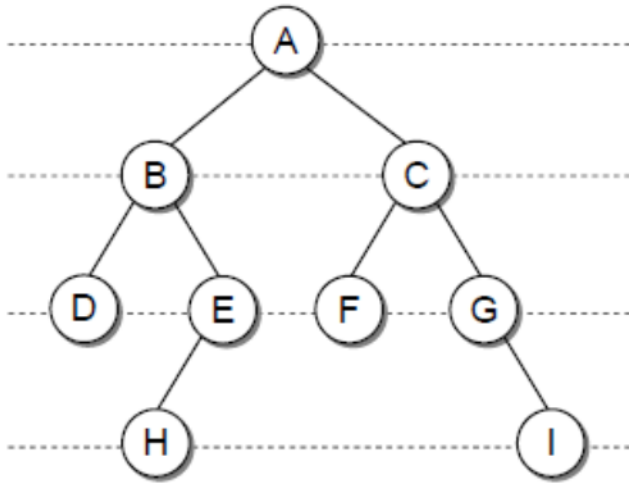
(b) _____



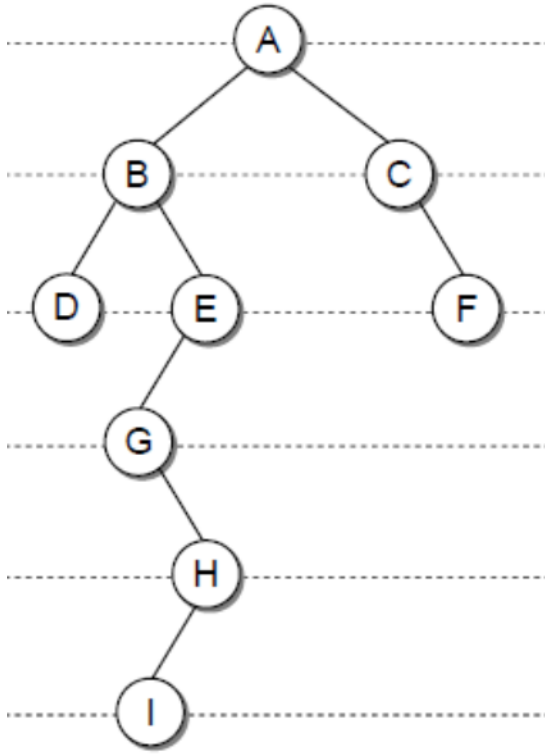
(c)_____

number of nodes

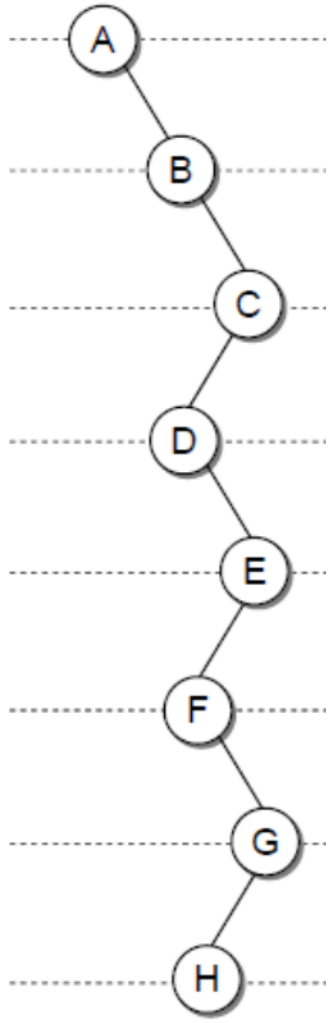
Size of Trees



(a)_____



(b) _____



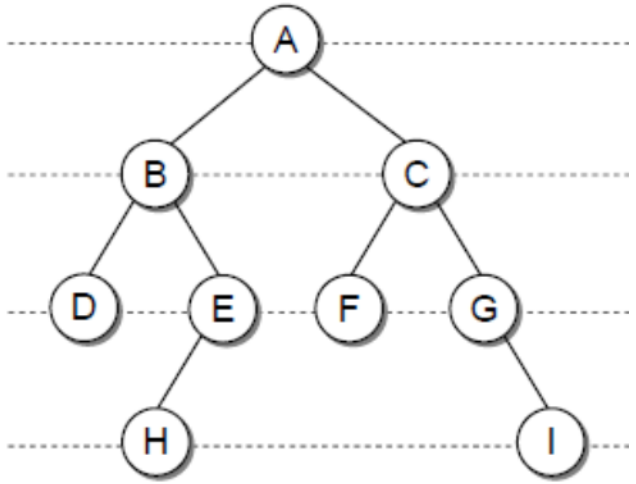
(c)_____

hint: the root node at level 0, its children at level 1 and so on.

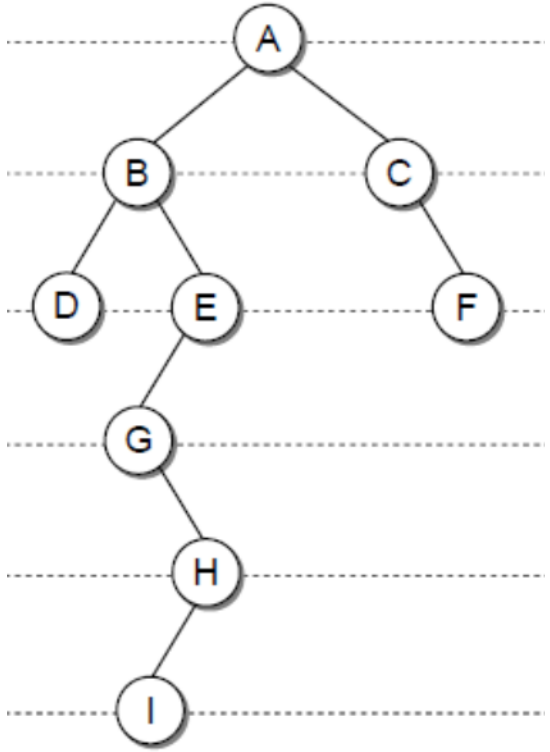
number of levels

Size of Trees

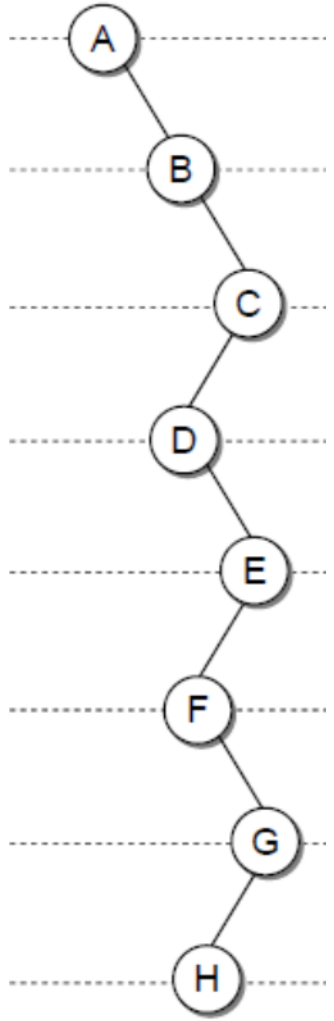
The height of a tree is the number of levels in a tree



(a)_____



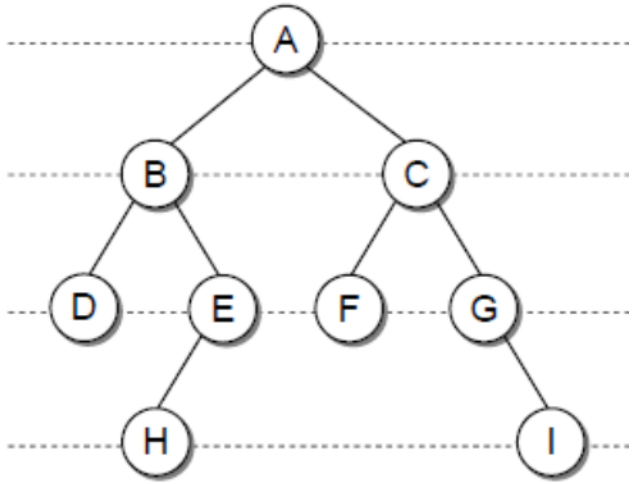
(b) _____



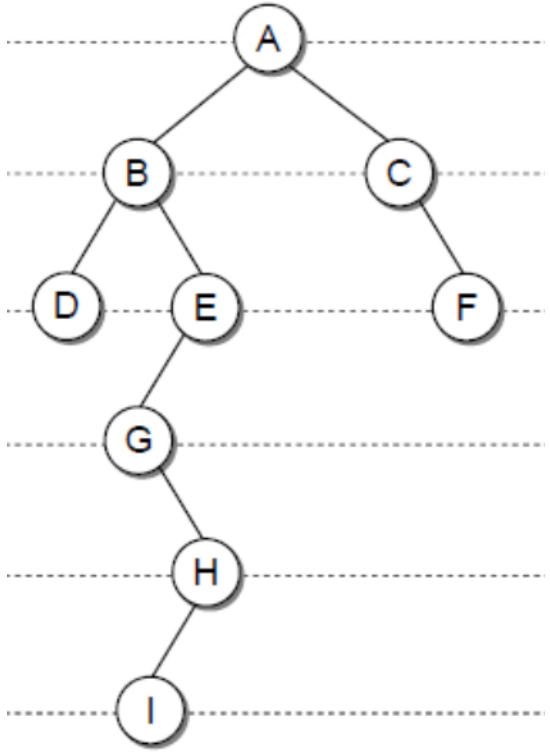
(c)_____

Size of Trees

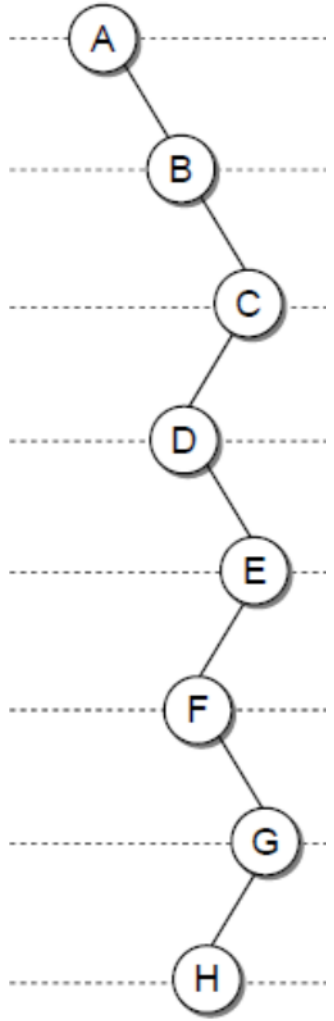
The width of a tree is number of nodes on the level containing the most nodes.



(a)_____



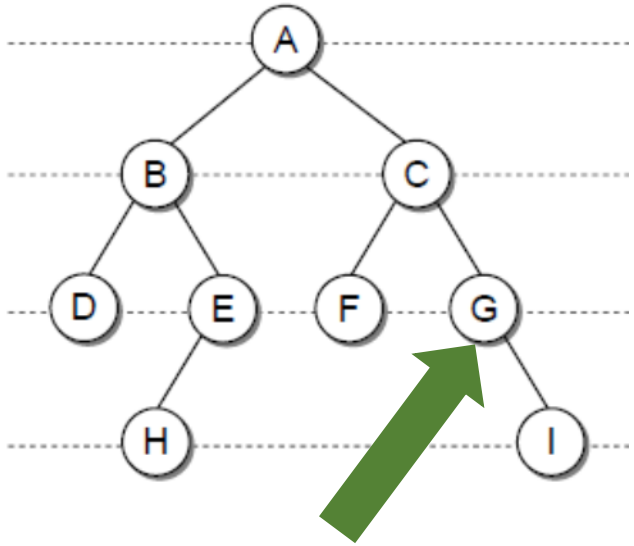
(b) _____



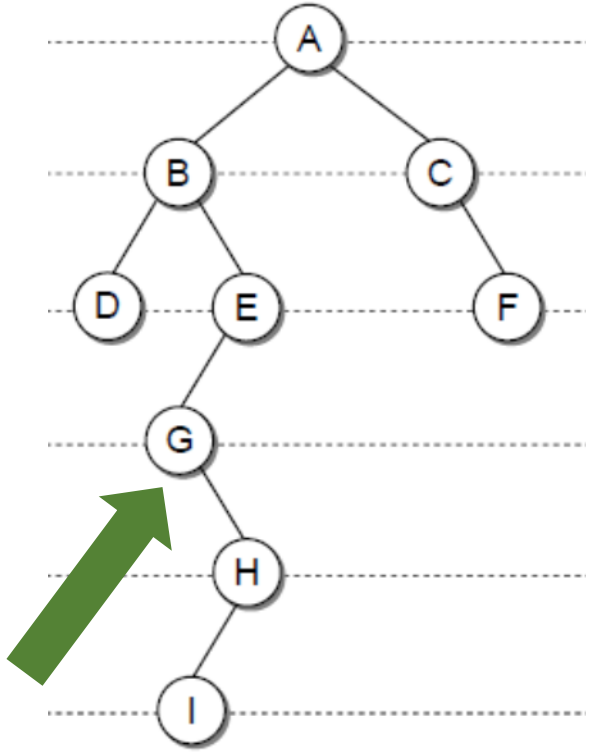
(c)_____

Size of Trees

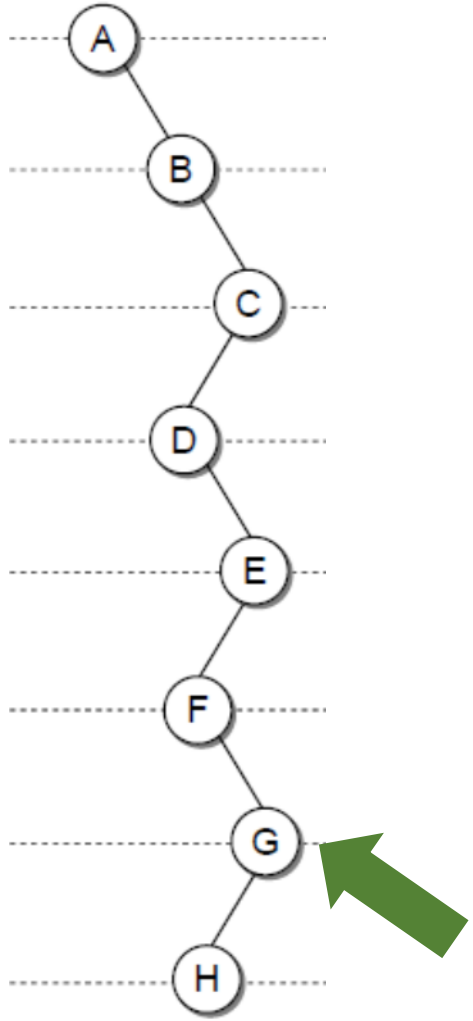
The depth of a node is its distance from the root, with distance being the number of levels that separate the two.



(a)_____



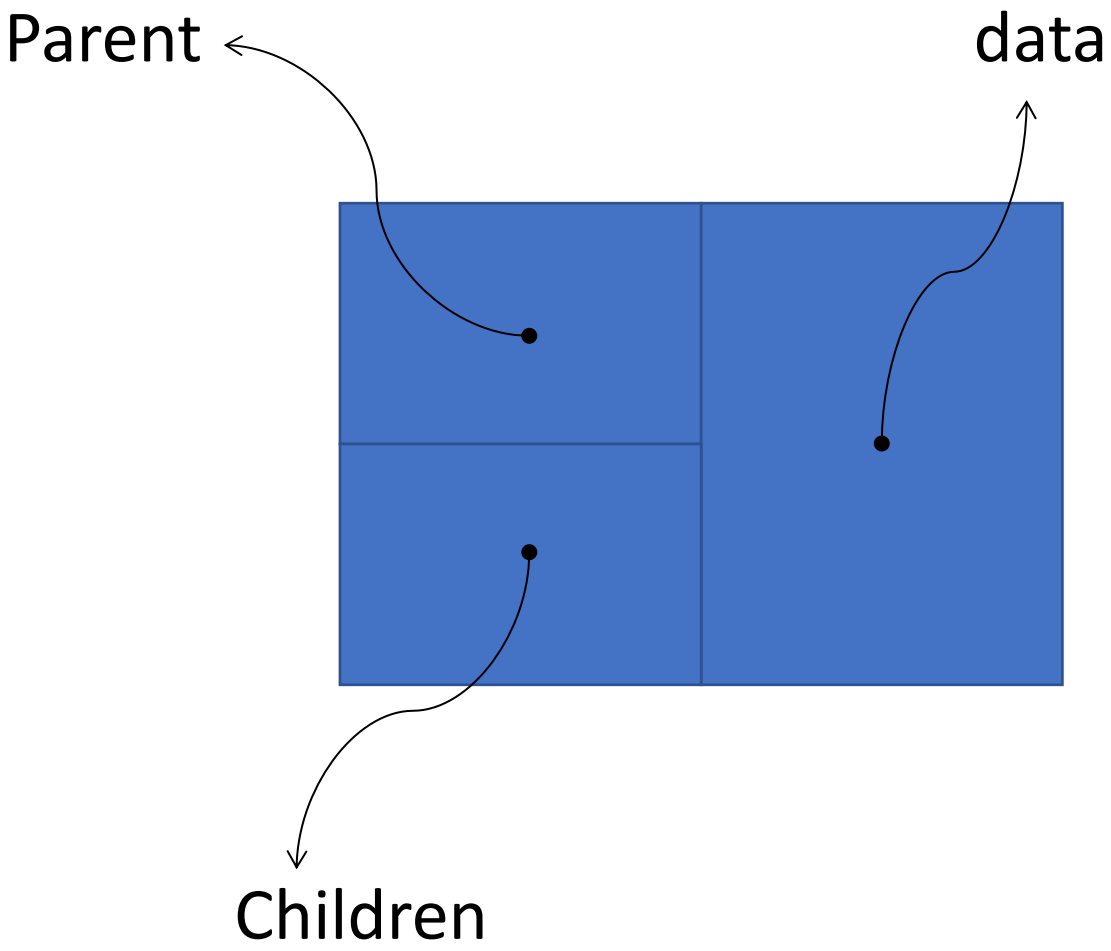
(b) _____



(c)_____

the depth of node G is ?

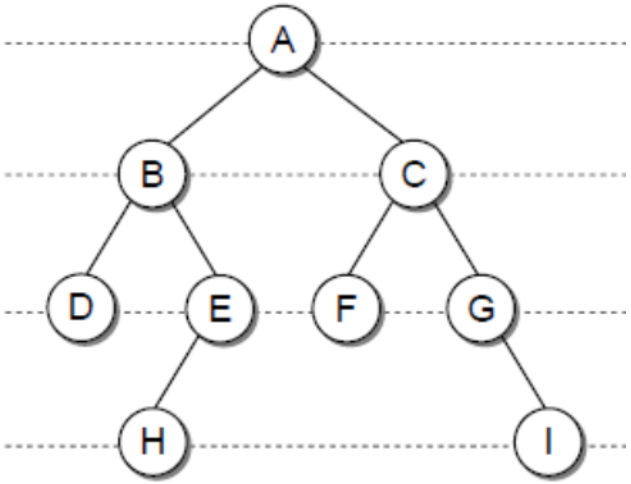
Implementation of General Tree node



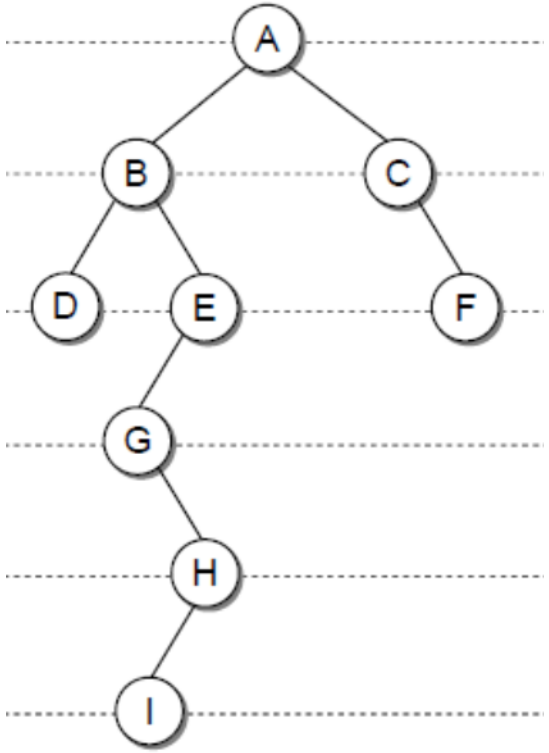

```
class gen_Tree_Node(self):
    def __init__(self , data):
        self.data = data
        self.Parent = None
        self.Children = list()
```

Binary Tree

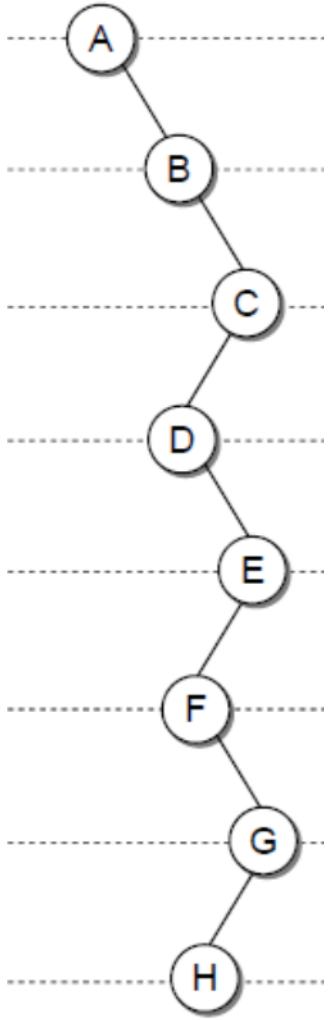
A binary tree is a tree in which <u>each node can have at most two children</u>.
One child is identified as the **left child** and the other as the **right child**.



(a)_____

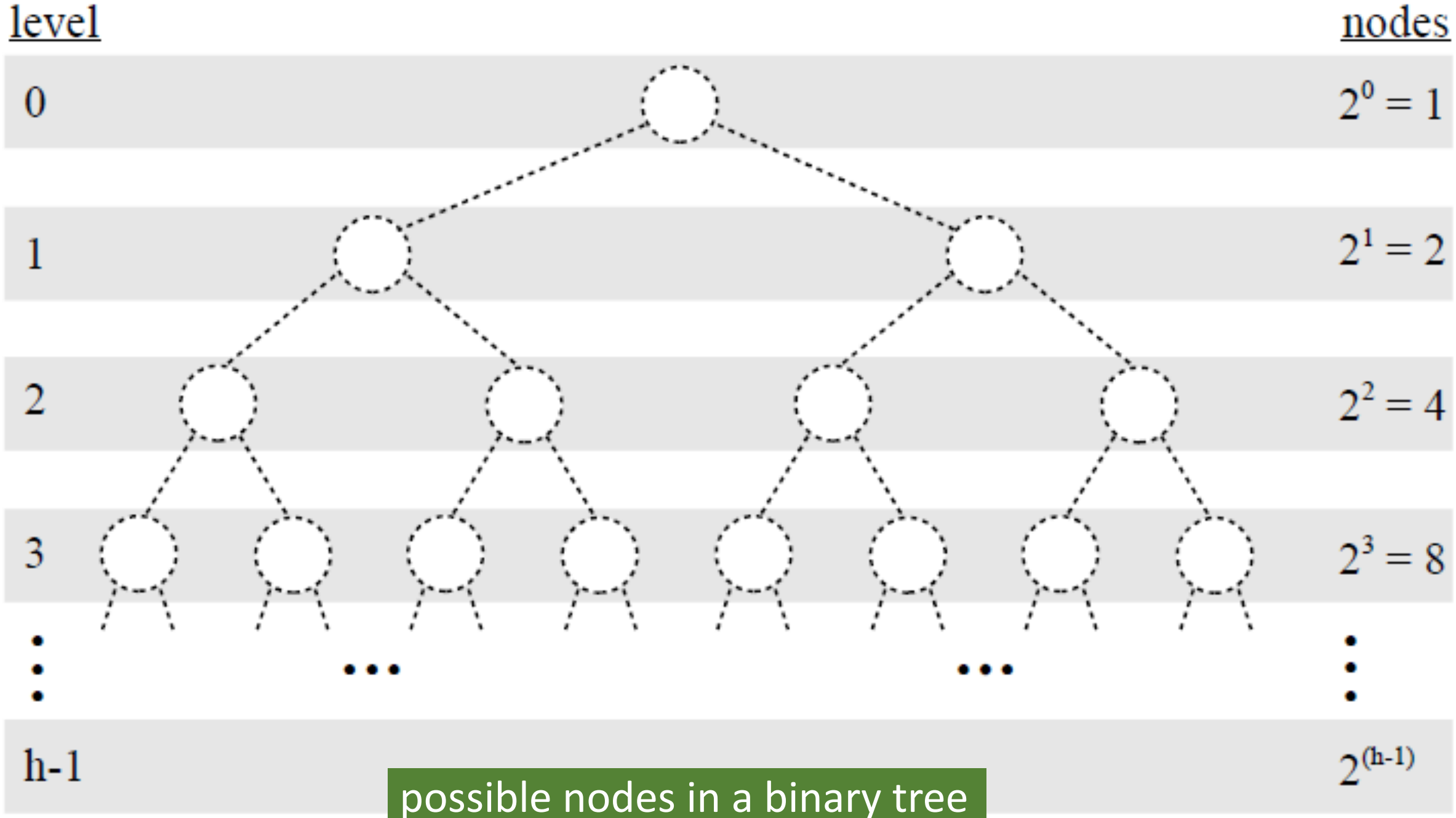


(b) _____



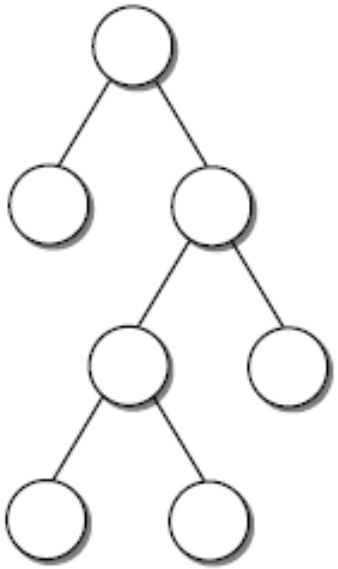
(c)_____

Which is a binary tree?

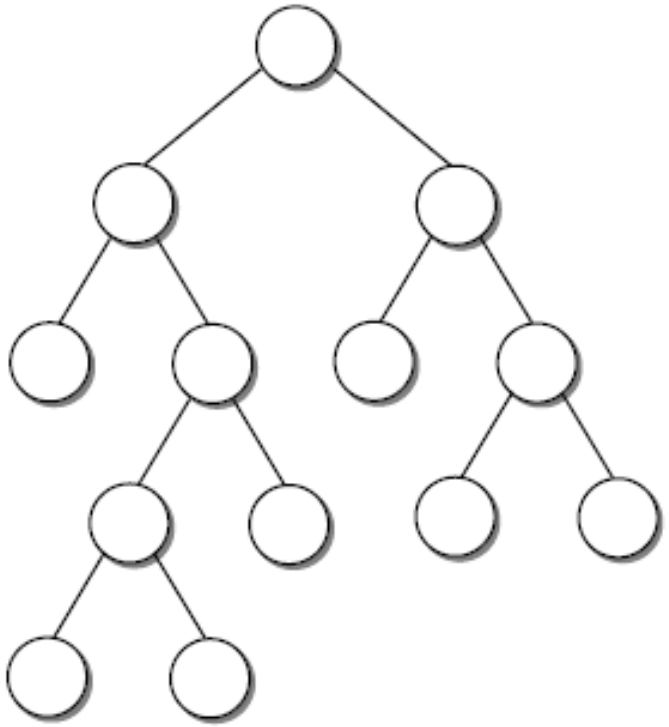


possible nodes in a binary tree

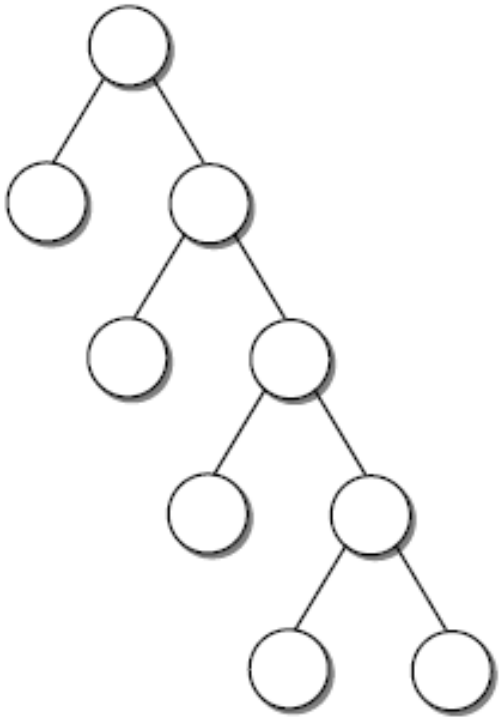
A full binary tree is a binary tree in which each **internal node** contains **two children**.



(a)_____

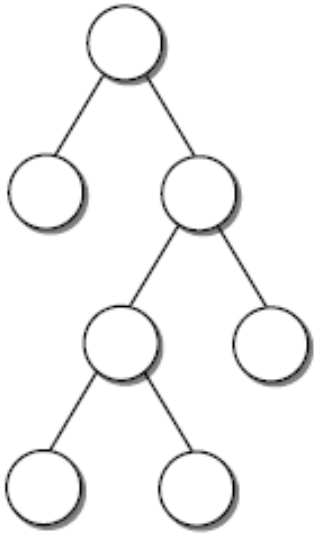


(b) _____

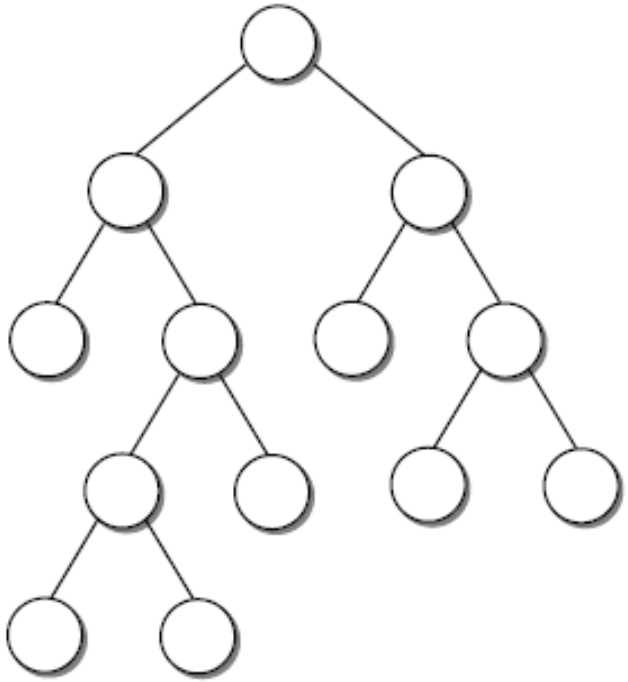


(c)_____

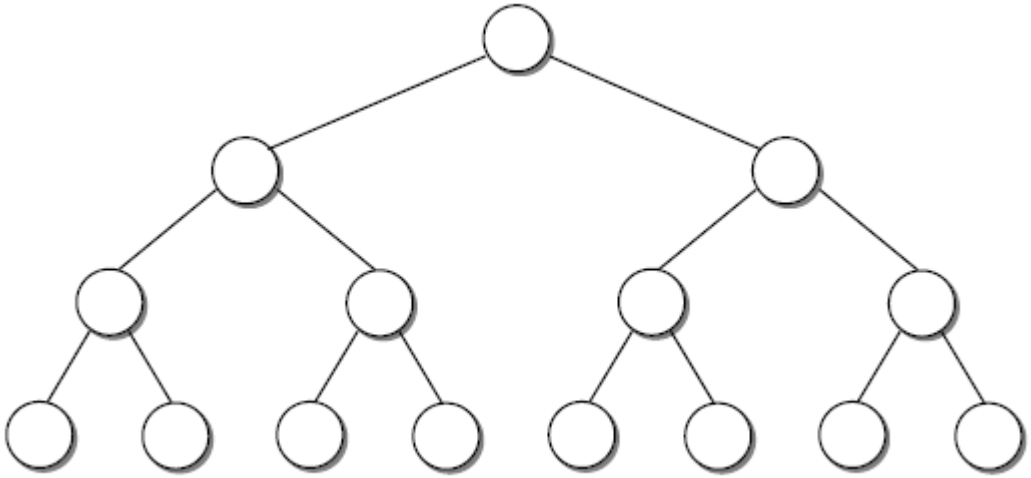
A perfect binary tree is a full binary tree in which all leaf nodes are at the same level. The perfect tree has all possible node slots filled from top to bottom with no gaps.



(a)_____

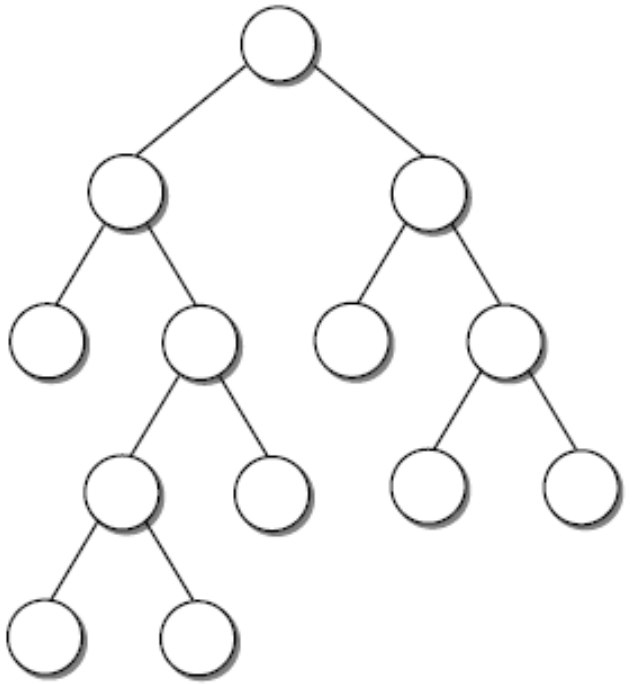


(b) _____

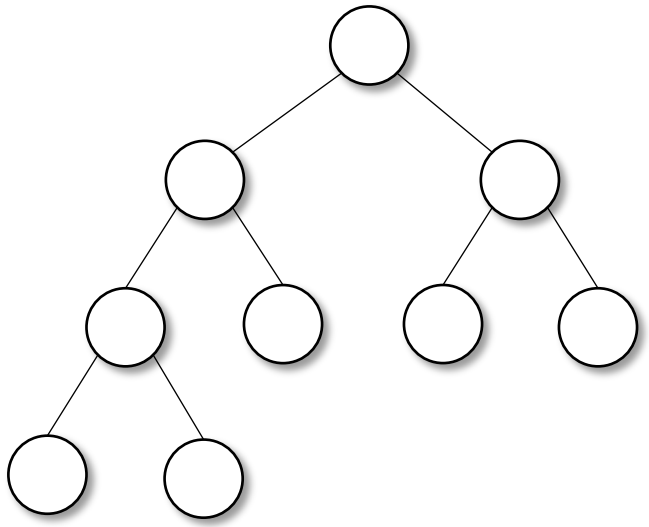


(c)_____

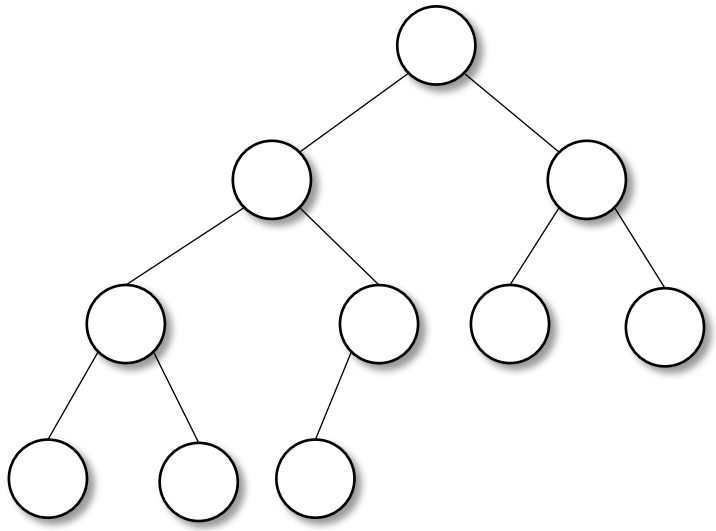
A complete binary tree is a binary tree that it is **a perfect binary tree down to height *h - 1*** and the nodes on the lowest level fill the available slots from **left to right** leaving no gaps.



(a)_____



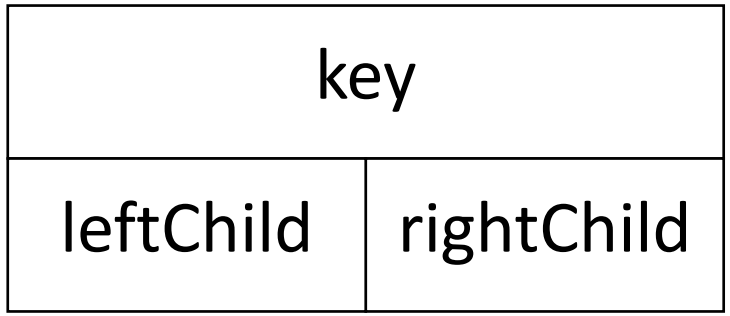
(b) _____



(c)_____

```
class BinaryTree:
    def __init__(self,Rootobj):
        self.key = Rootobj
        self.leftChild = None
        self.rightChild = None
```

BinaryTree

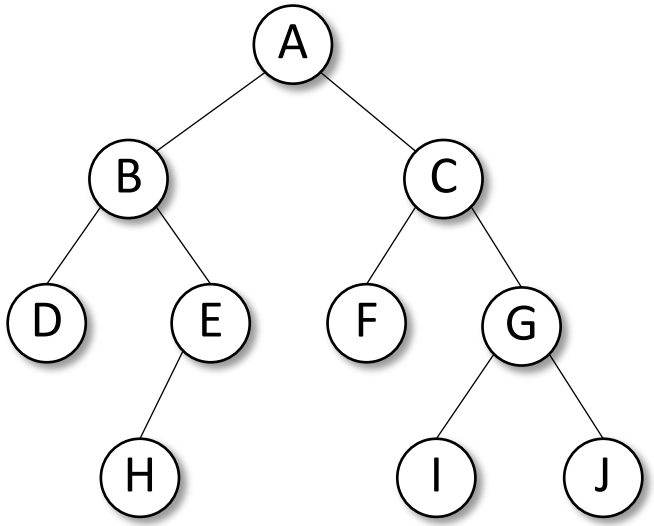



```
def insertLeft(self,newNode):
    if self.leftChild == None:
        self.leftChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.leftChild = self.leftChild
        self.leftChild = t
```

```
def insertRight(self,newNode):
    if self.rightChild == None:
        self.rightChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.rightChild = self.rightChild
        self.rightChild = t
```

Tree Traversals

Refer to **a linked list**, the traversal is simple because we can start with the first node and follow the links through the next nodes, that there is only one next node. But for **Tree**, there is **no single path** from the root to every other node.



Preorder Traversal

- **root → left → right**

Postorder Traversal

- **left → right → root**

Inorder Traversal

- **left → root → right**

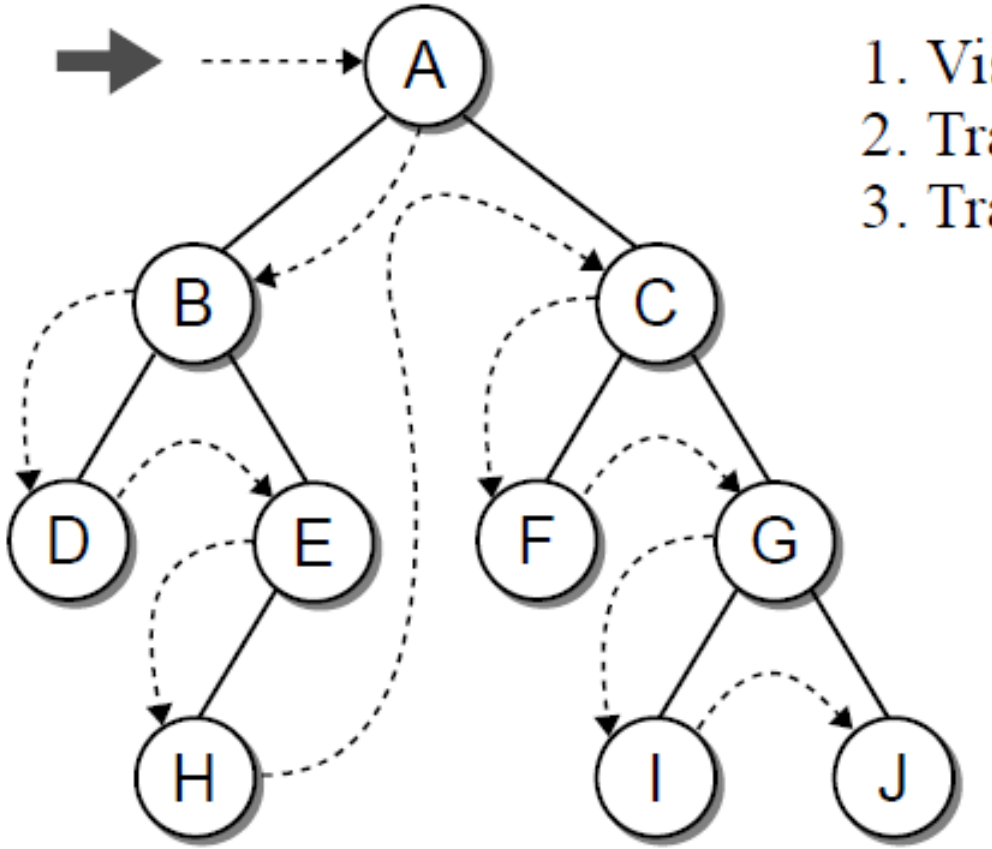
Depth-First Traversal

Breadth-First Traversal

- **root → left → right (by level)**

Preorder Traversal

- root → left → right



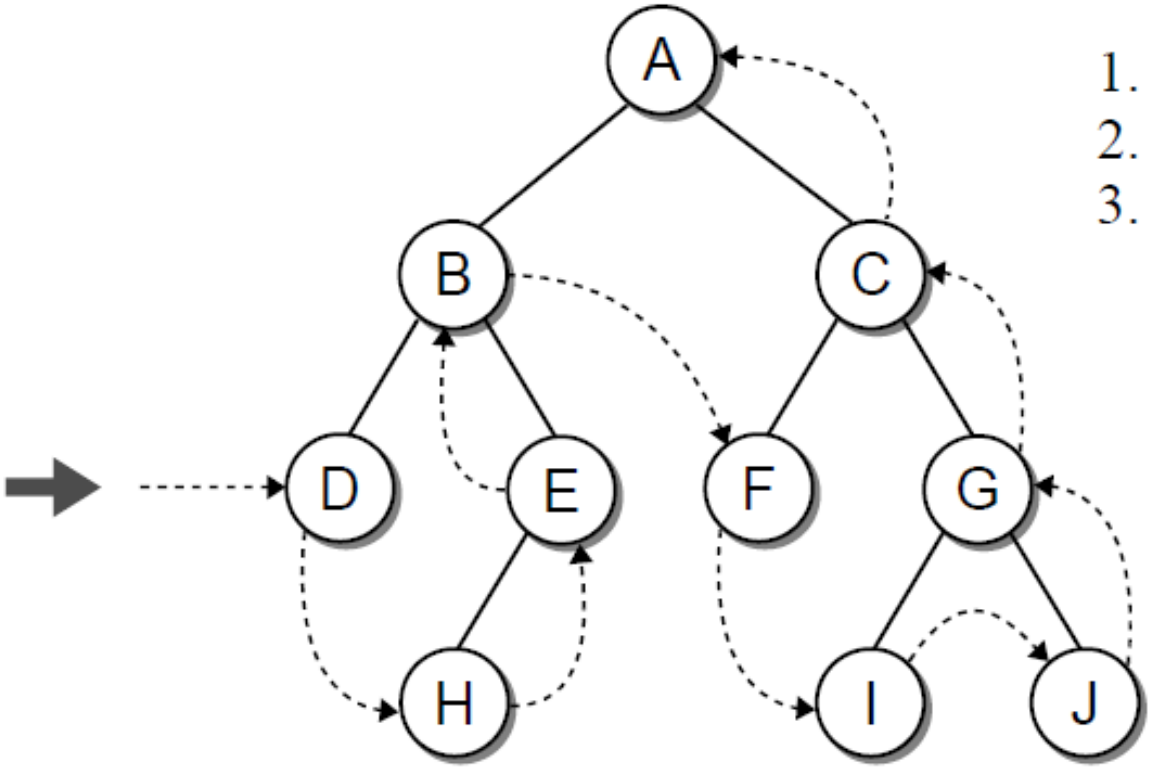
1. Visit the node.
2. Traverse the left subtree.
3. Traverse the right subtree.

```
def preorder(self):
    print(self.key)
    if self.leftChild:
        self.leftChild.preorder()
    if self.rightChild:
        self.rightChild.preorder()
```

[A B D E H C F G I J]

Postorder Traversal

- left → right → root

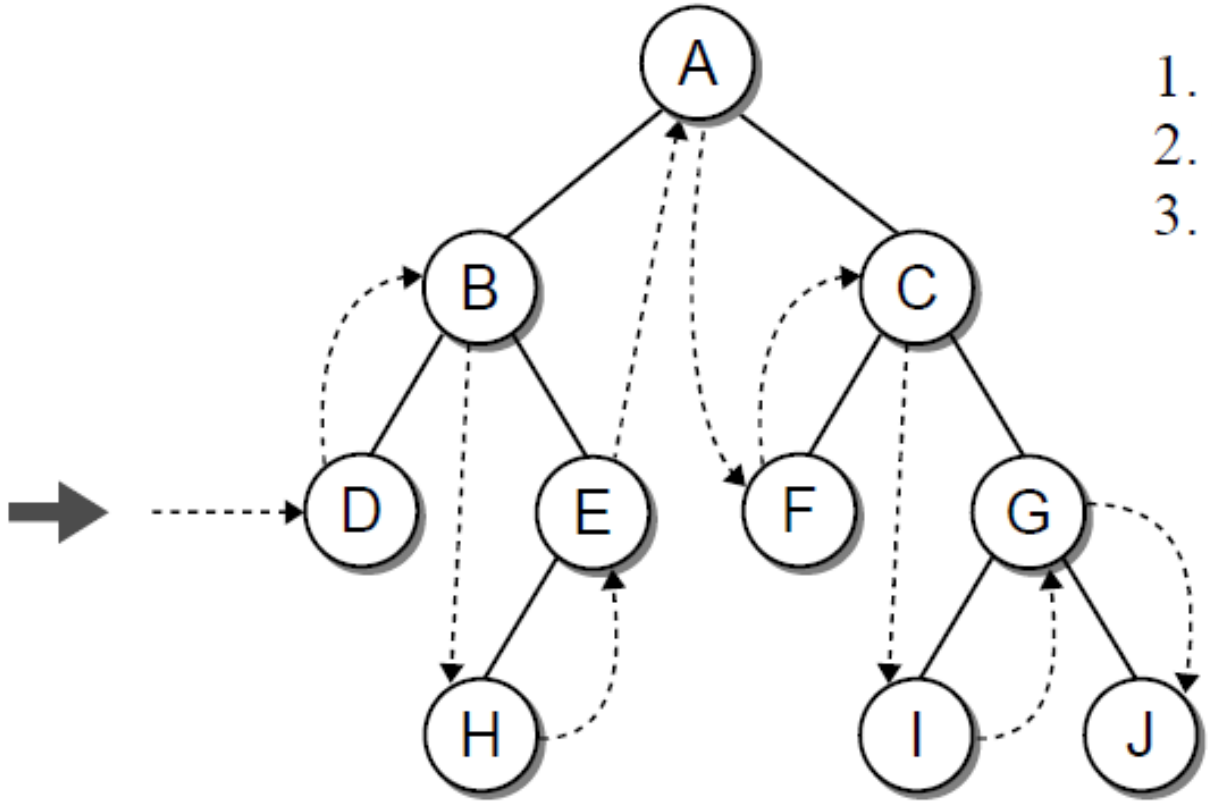


1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the node.

[D H E B F I J G C A]

Inorder Traversal

- left → root → right

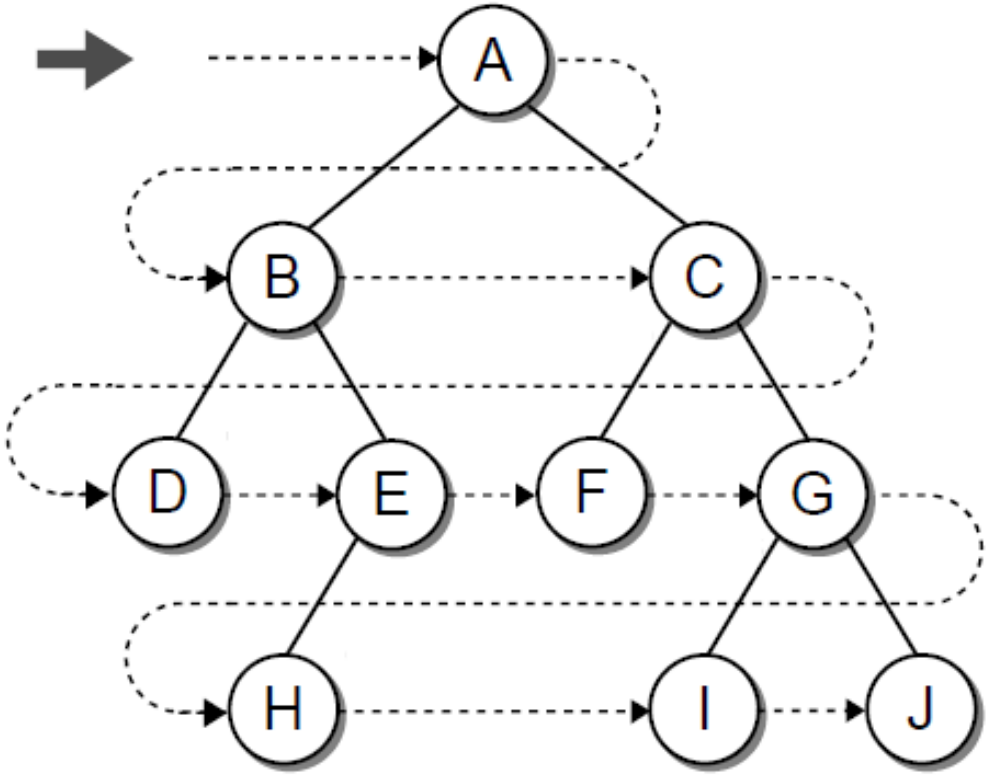


1. Traverse the left subtree.
2. Visit the node.
3. Traverse the right subtree.

[D B H E A F C I G J]

Breadth-First Traversal

- root → left → right (by level)



[A B C D E F G H I J]

```
def breadthorder(self):
        q = list()
        q.append(self)
      while q:
            node = q.pop(0)
            print(node.key)
           if node.leftChild:
                q.append(node.leftChild)
           if node.rightChild:
                q.append(node.rightChild)
```

Heap

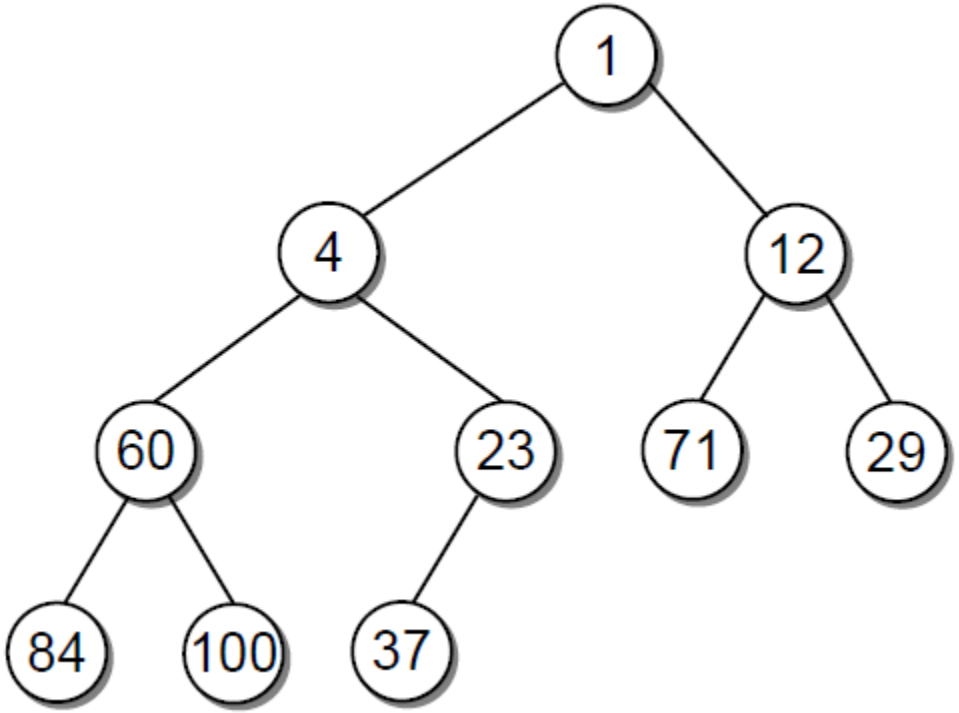
A Heap is a complete binary tree in which the nodes are organized based on their data entry values.

- **A max-heap** has the property, known as the heap order property, that for each non-leaf node ***V*** , the value in ***V*** is greater than the value of its two children.

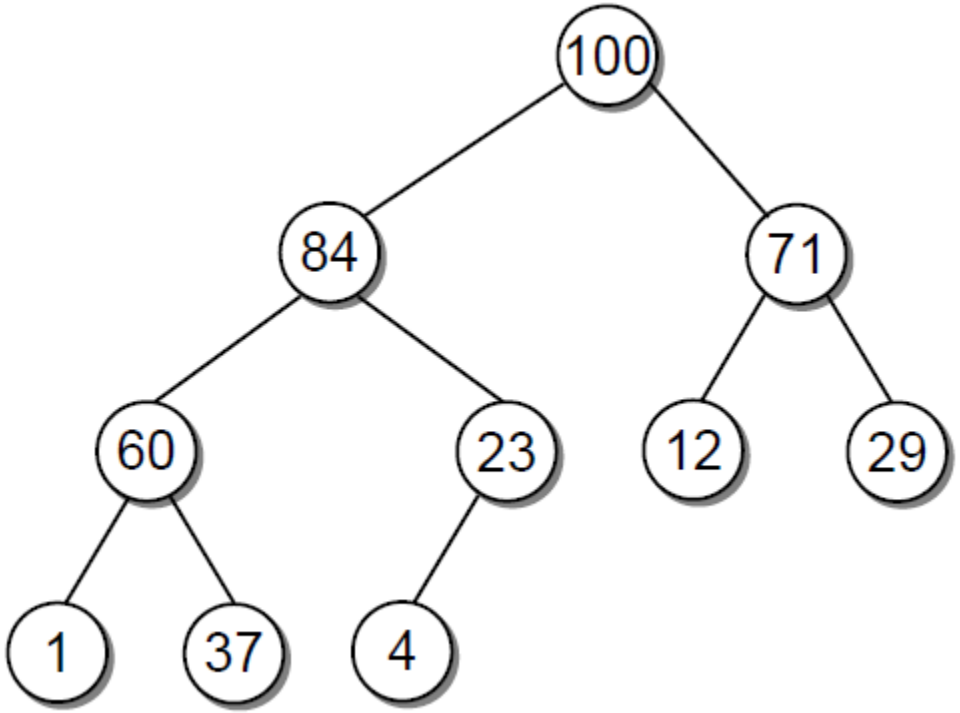
  The largest value in a max-heap will always be stored in the root while the smallest values will be stored in the leaf nodes.

- **The min-heap** has the opposite property. For each non-leaf node ***V*** , the value in ***V*** is smaller than the value of its two children.

Which one is min-heap ?



(a)_____

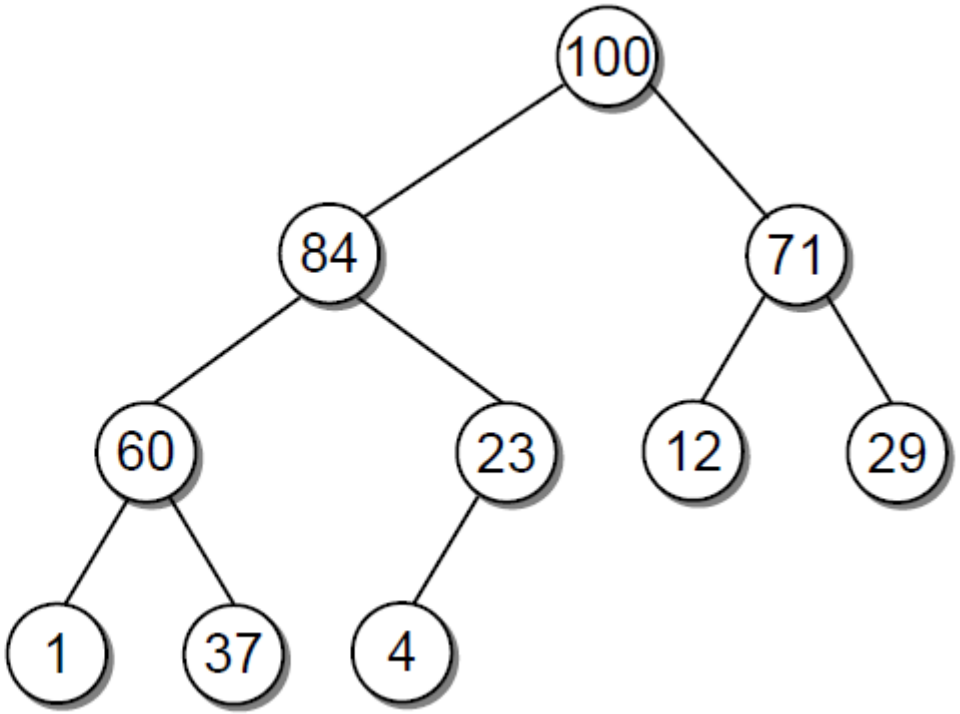


(b) _____

The heap is a specialized structure with limited operations. We can **insert** a new value or **extract** and **remove** the root node's value from the heap.

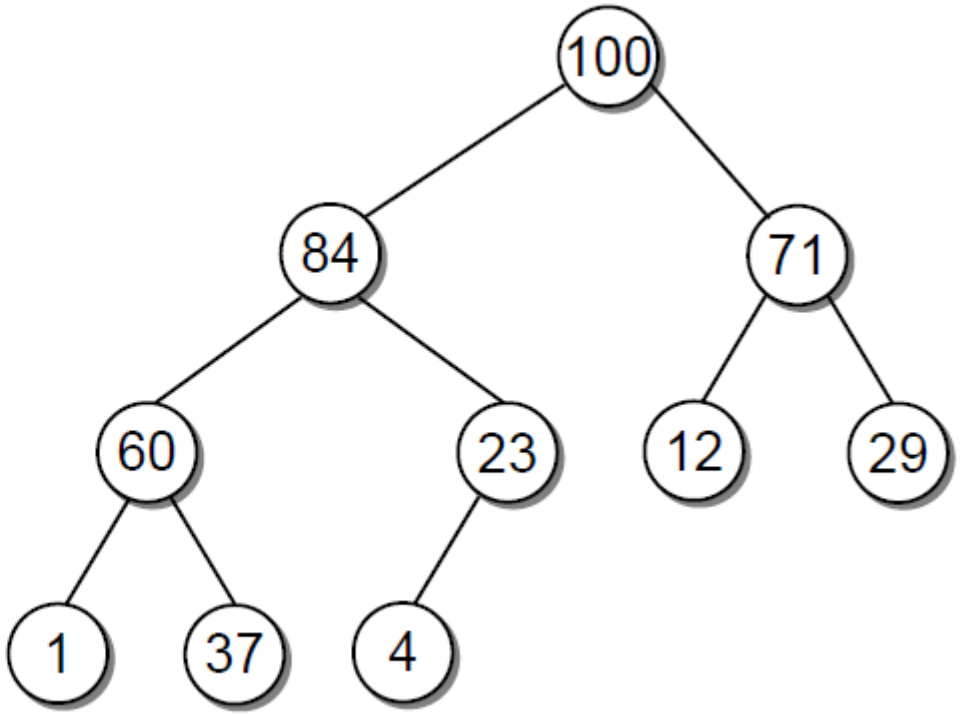
Insertion

When a new value is inserted into a heap, the heap order property and the heap shape property (a complete binary tree) must be maintained.



answer: _____

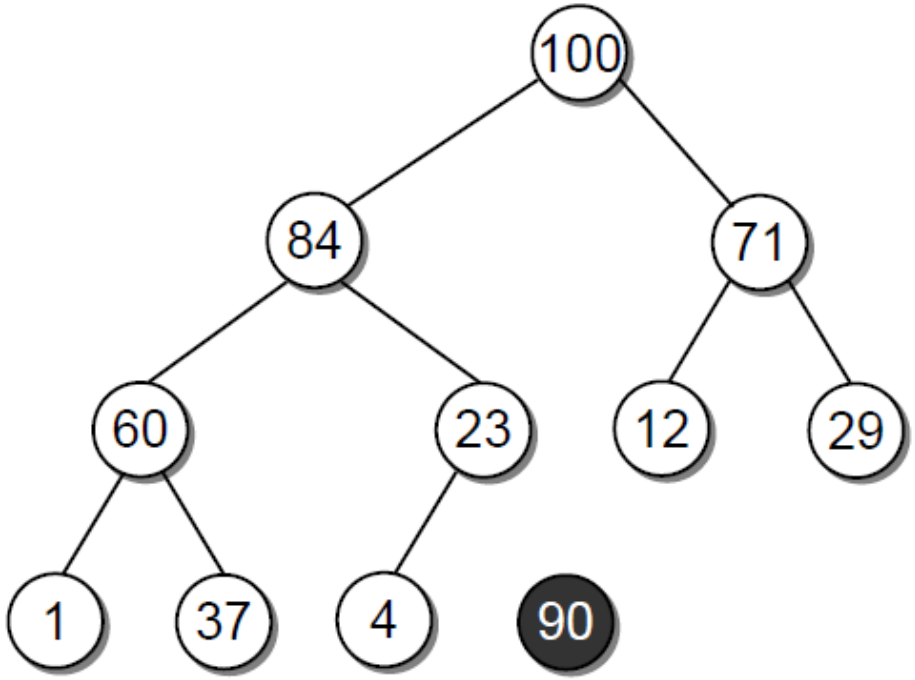
if we insert the value **90** into this max-heap tree, where should be inserted ?



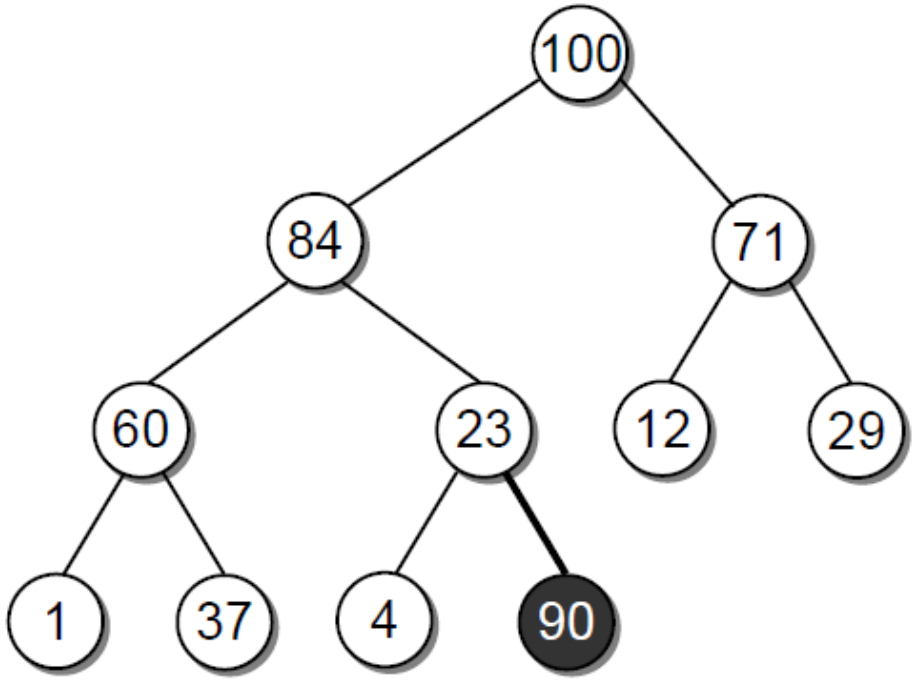
answer: _____

if we insert the value **41** into this max-heap tree, where should be inserted ?

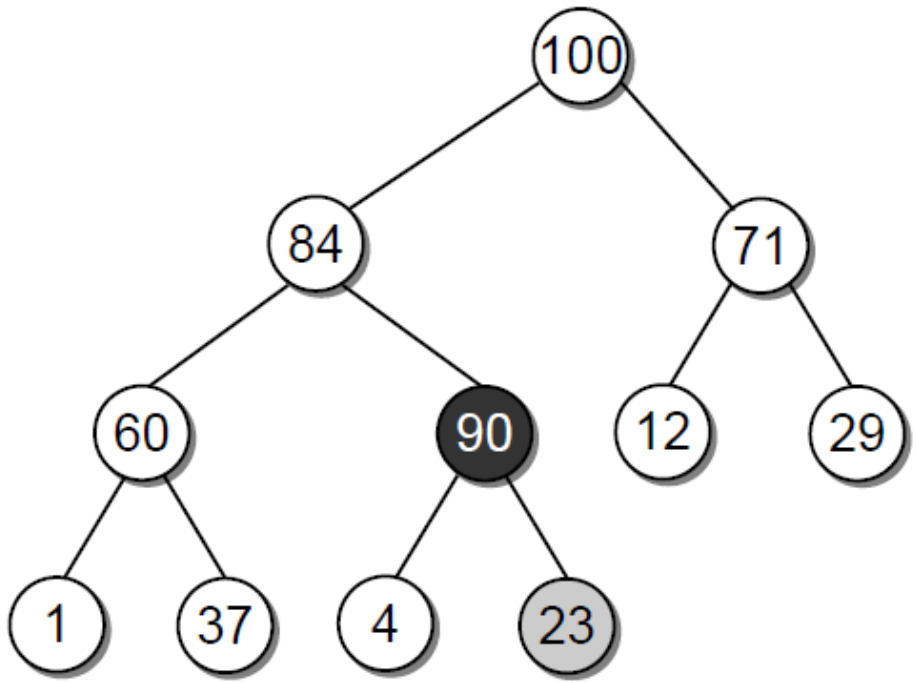
Implementation of Heap's Insertion



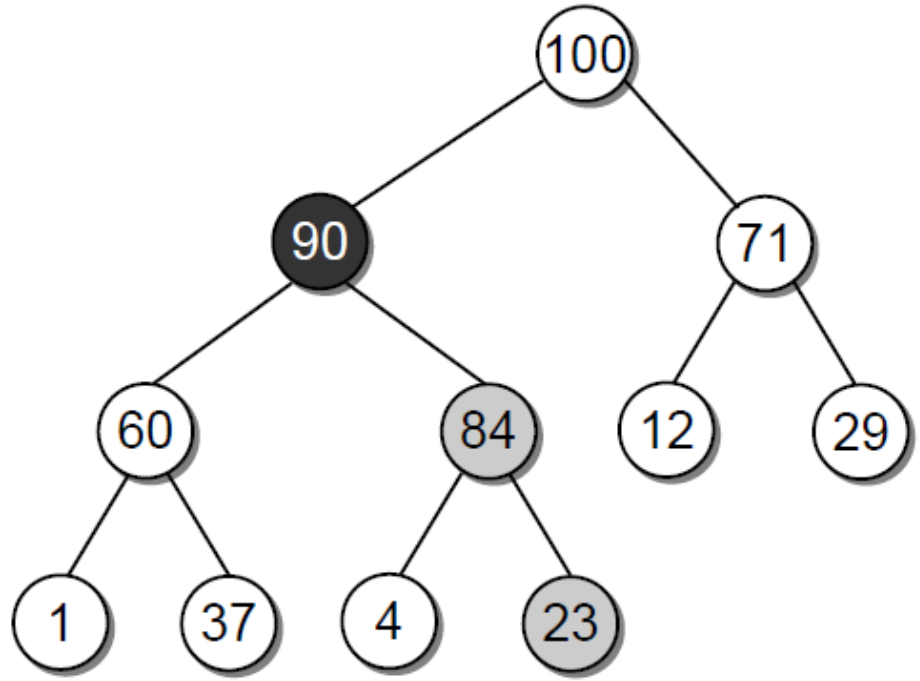
(a) create a new node for 90.



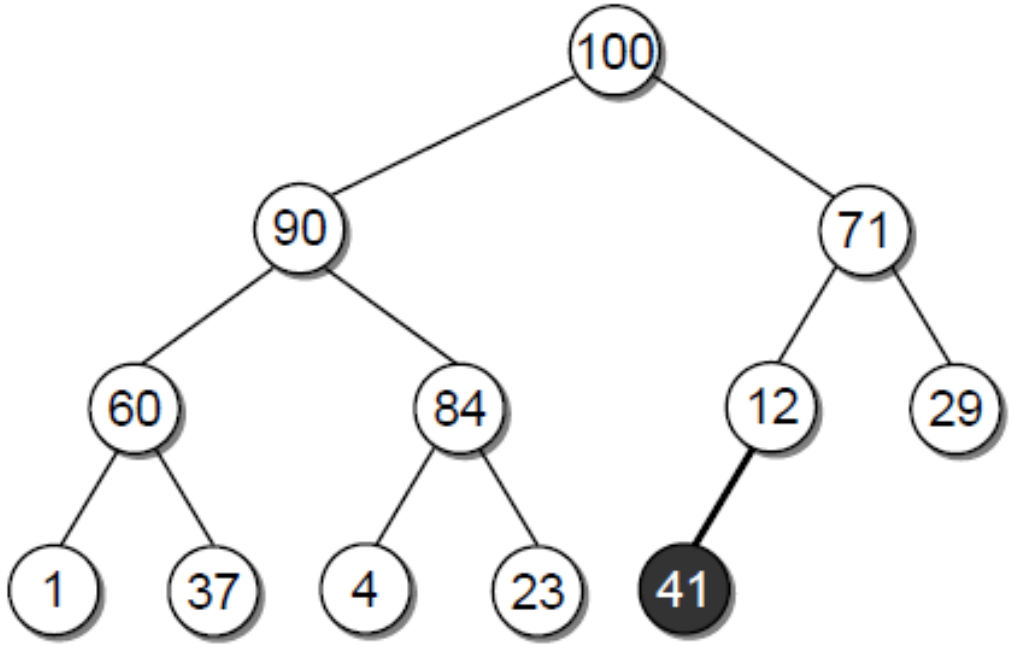
(b) link the node as the last child.



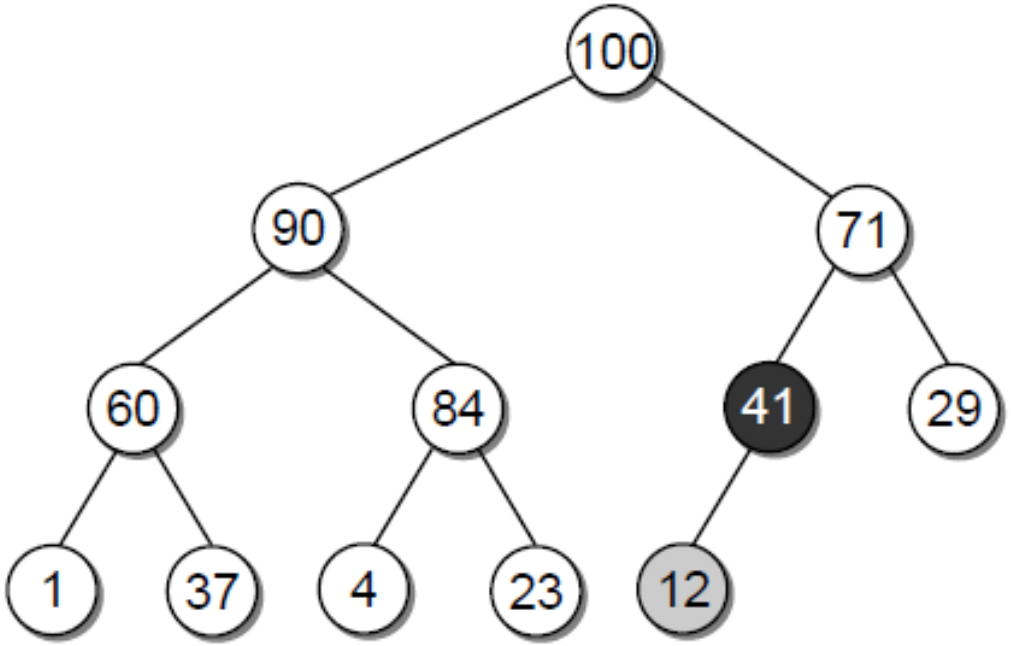
(c) sift-up: swap 23 and 90.



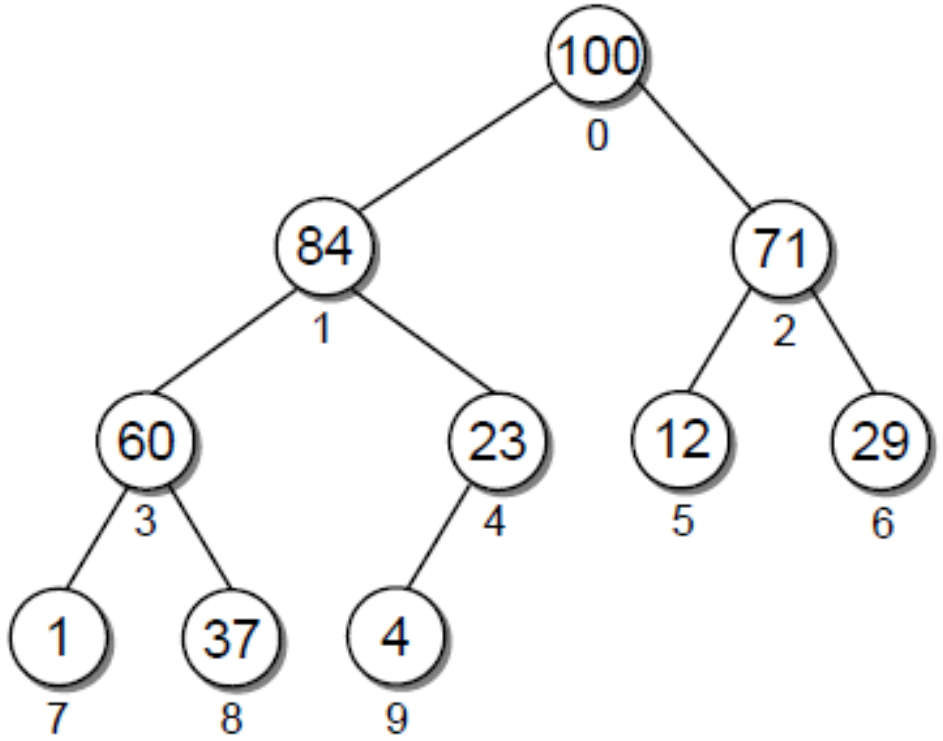
(d) sift-up: swap 84 and 90.



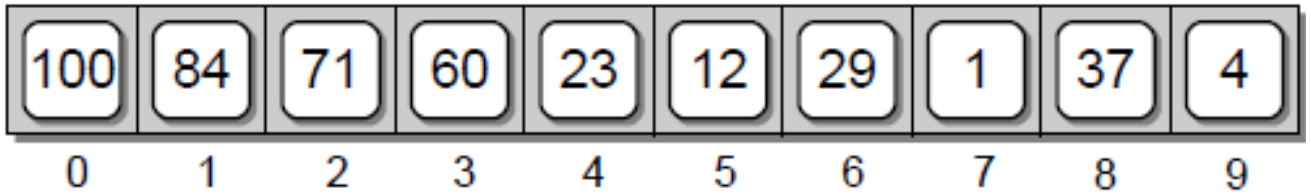
(a) create a new node and link for 41



(b) sift-up: swap 41 and 12



(a)



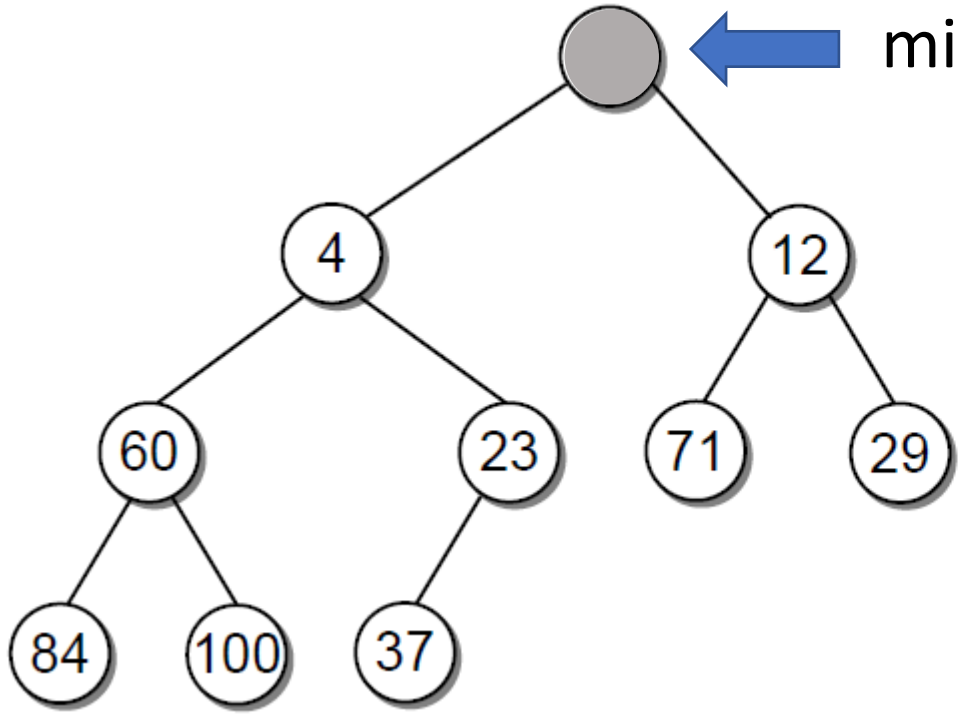
(b)

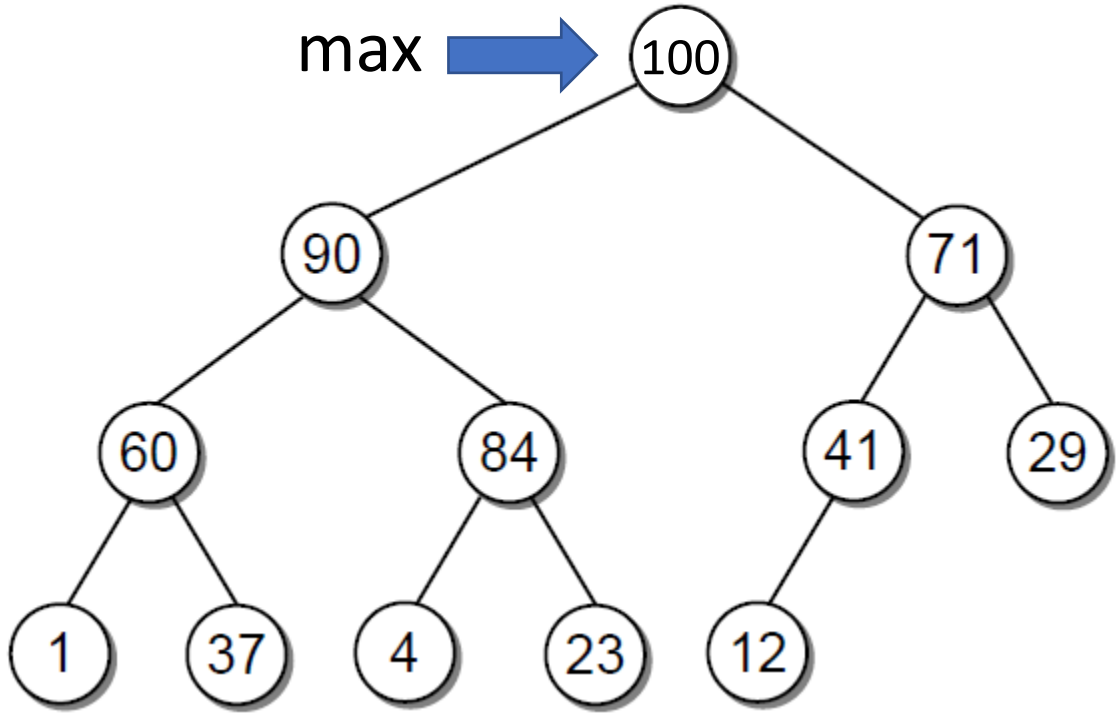
i = index number in array
parent = (i -1) // 2
leftChild = 2 * i + 1
rightChild = 2 * i + 2

```
def _siftUp( self, ndx ):
    if ndx > 0 :
        parent = ndx // 2
        if self._elements[ndx] > self._elements[parent] :
            # swap elements
            tmp = self._elements[ndx]
            self._elements[ndx] = self._elements[parent]
            self._elements[parent] = tmp
            self._siftUp( parent )
```

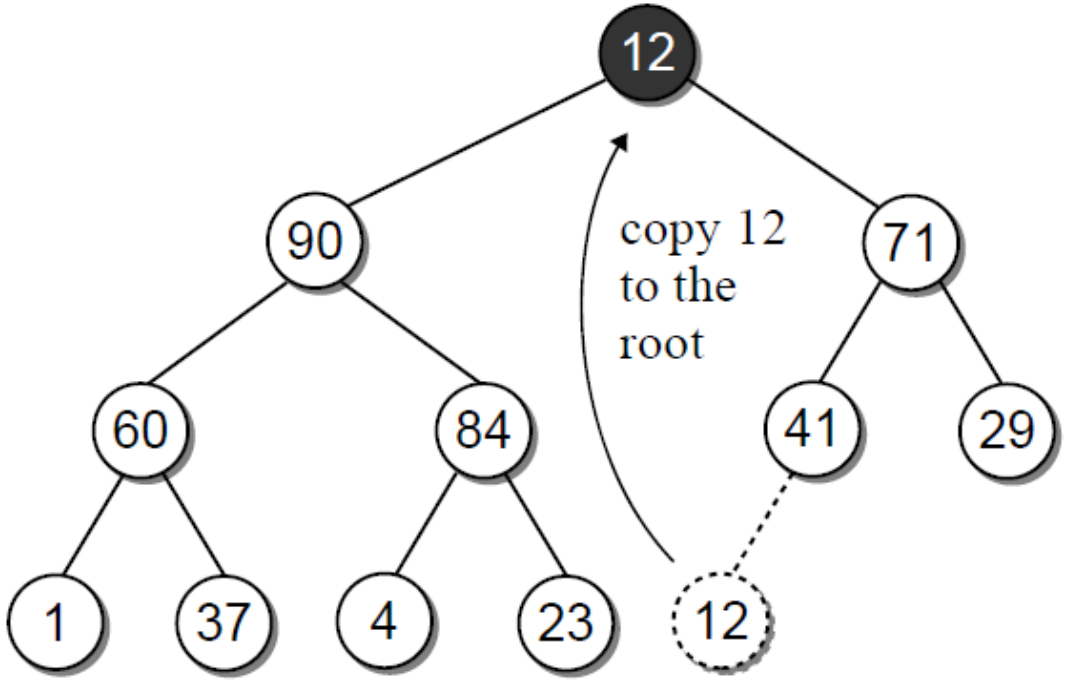
Extractions

When a value is extracted and removed from the heap, it can only come from the root node. Thus, in a max-heap, we always extract the largest value and in a min-heap, we always extract the smallest value.

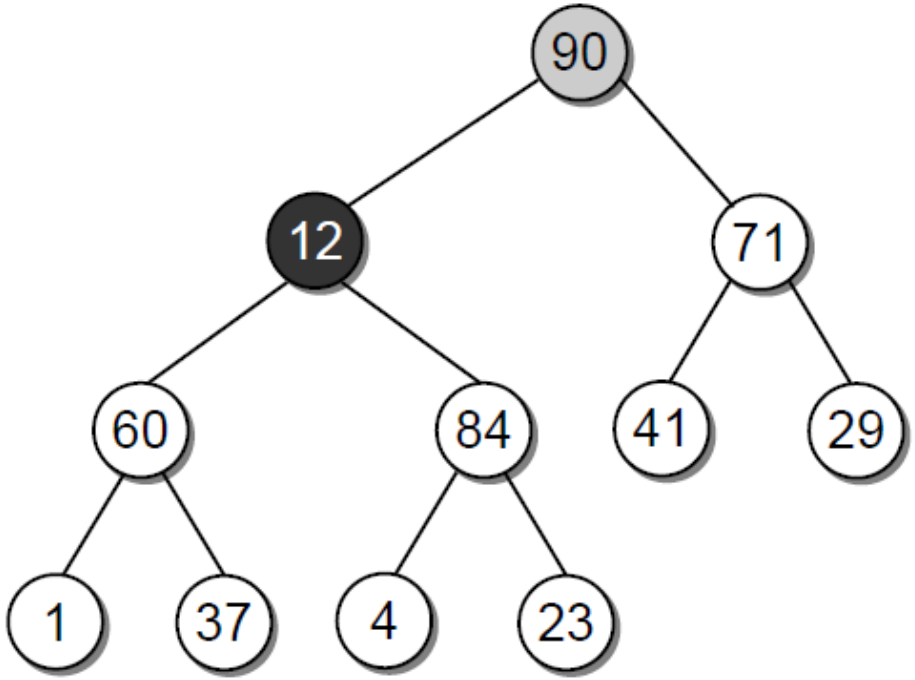




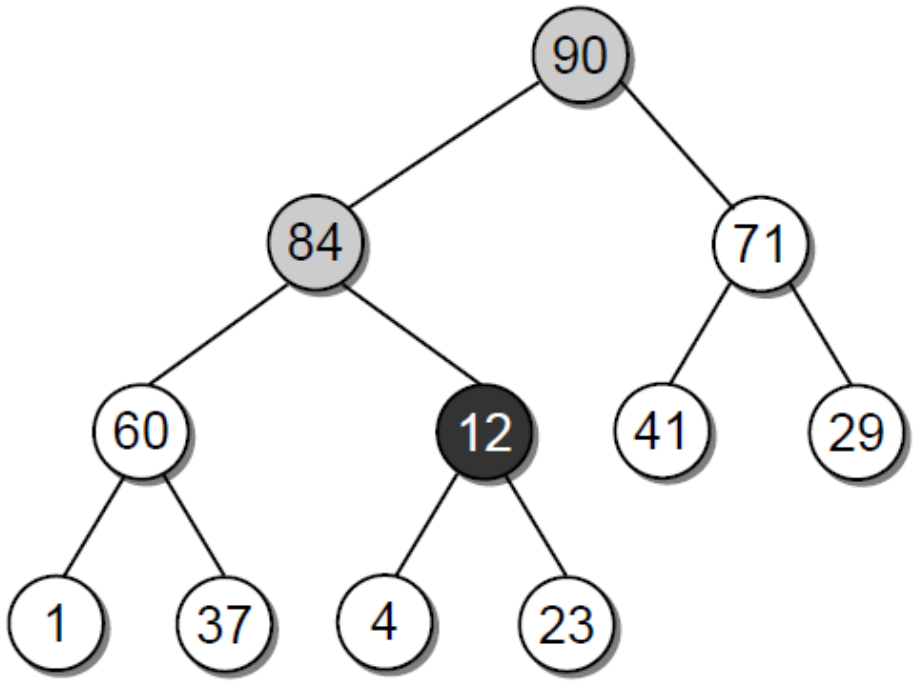
Implementation of Heap's Extraction



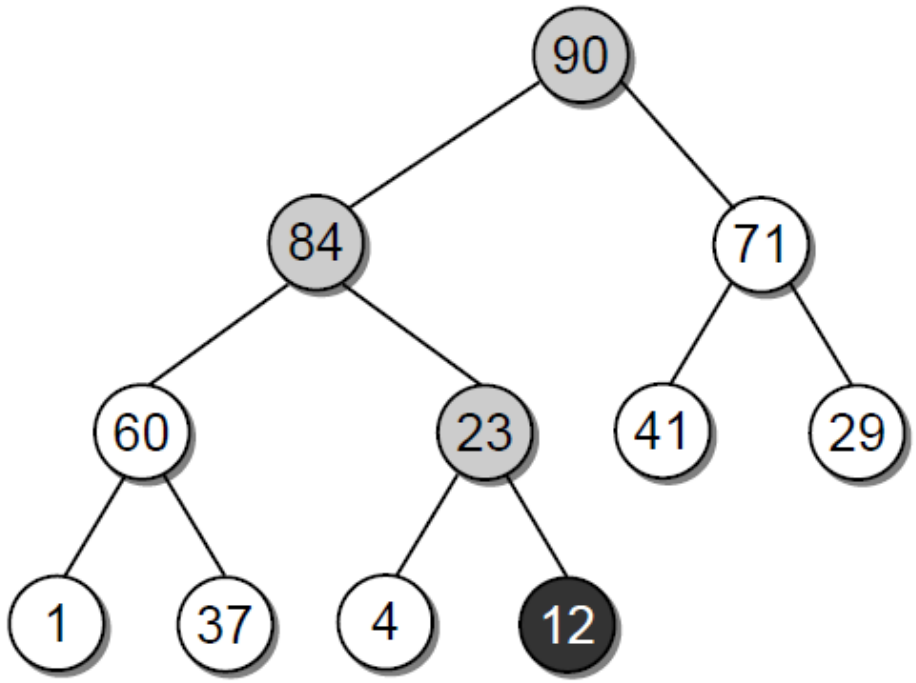
(a) copy the last item to the root.



(b) sift-down: swap 12 and 90.



(c) sift-down: swap 12 and 84.

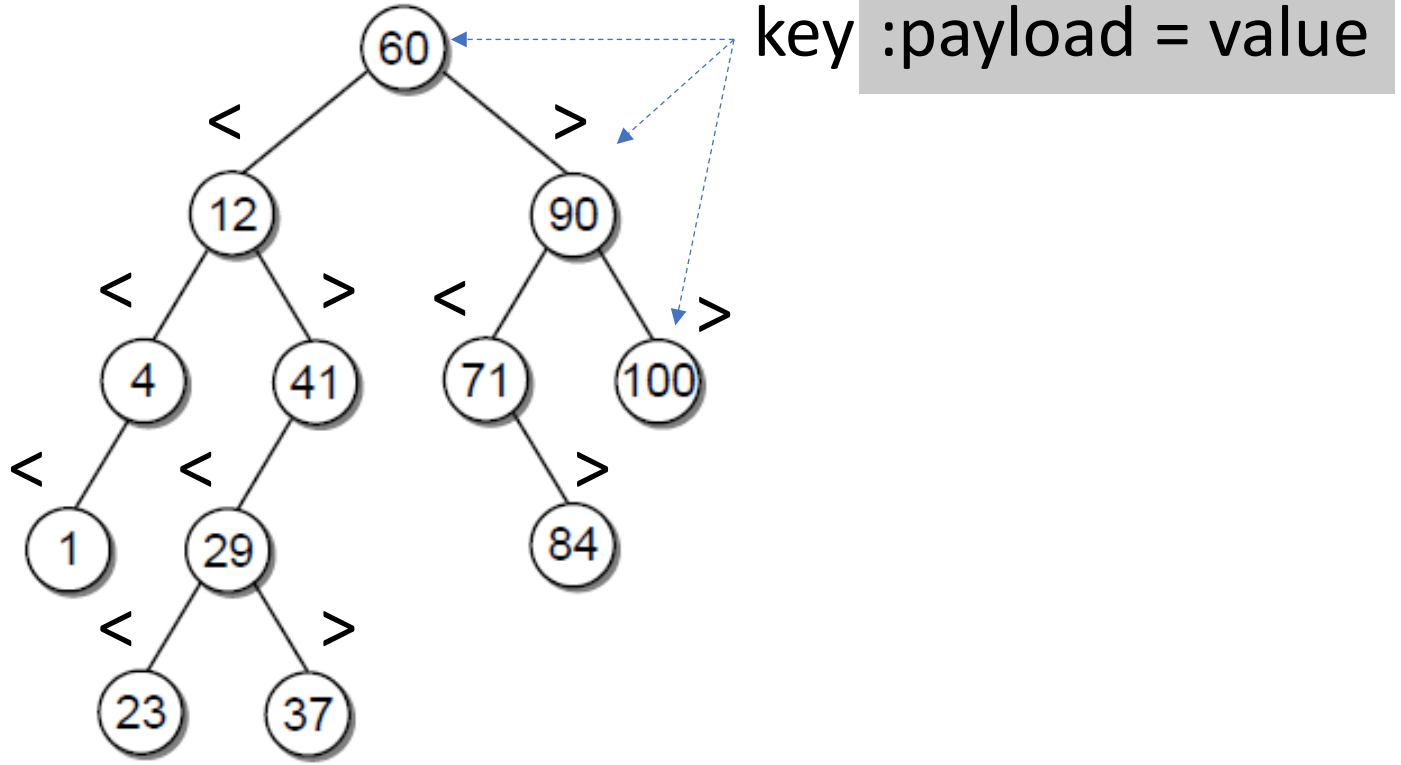


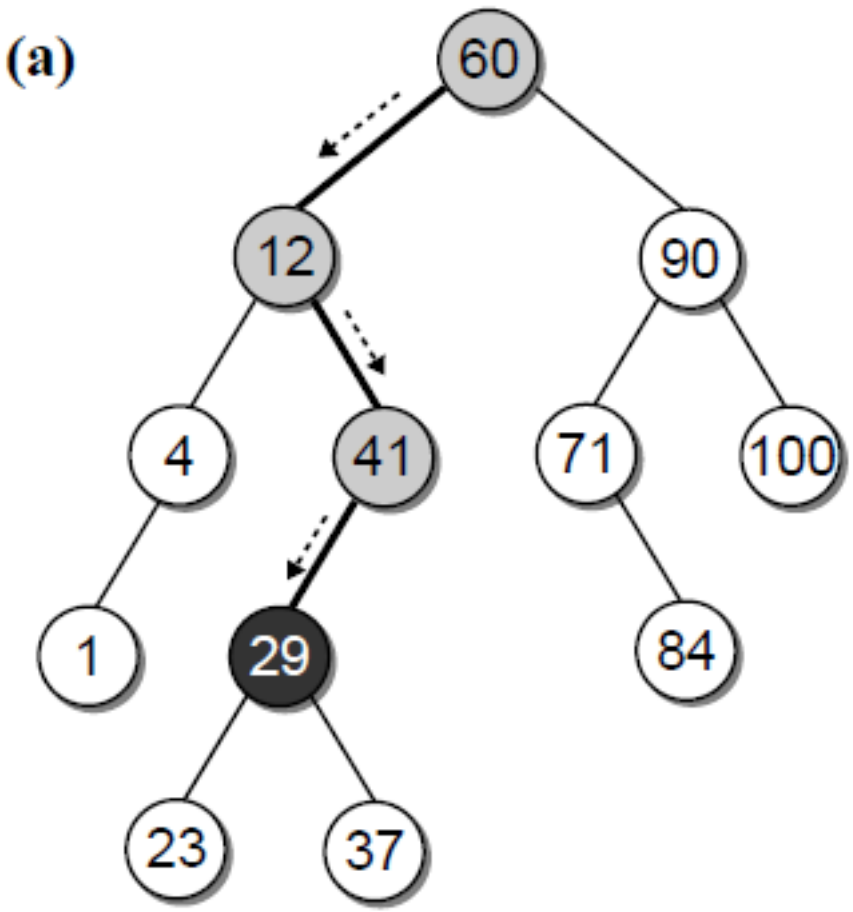
(d) sift-down: swap 12 and 23.

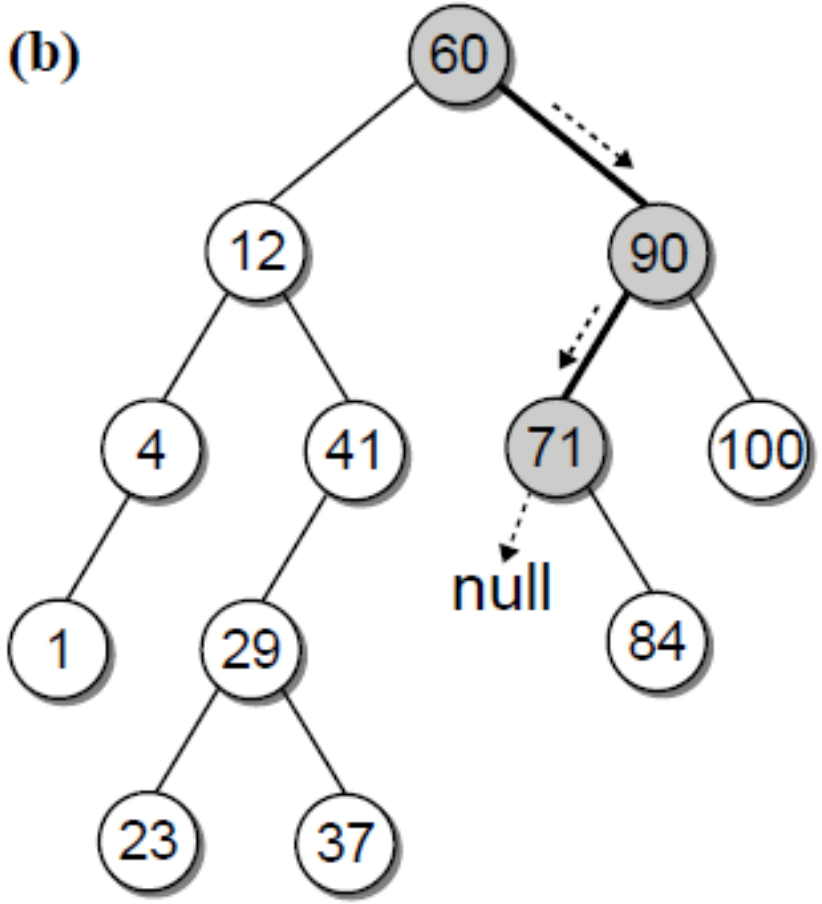
```
def _siftDown( self, ndx ):
    left = 2 * ndx + 1
    right = 2 * ndx + 2
    # Determine which node contains the larger value.
    largest = ndx
    if left < count and self._elements[left] >= self._elements[largest] :
        largest = left
    elif right < count and self._elements[right] >= self._elements[largest]:
        largest = right
    # If the largest value is not in the current node (ndx), swap it with
    # the largest value and repeat the process.
    if largest != ndx :
        swap( self._elements[ndx], self._elements[largest] )
        _siftDown( largest )
```

A binary search tree (BST) is a binary tree in which each node contains a search key within its payload and the tree is structured such that for each interior node ***V*** :

- All keys **less than the key in node *V*** are stored in the **left subtree of *V***
- All keys **greater than the key in node *V*** are stored in the **right subtree of *V***







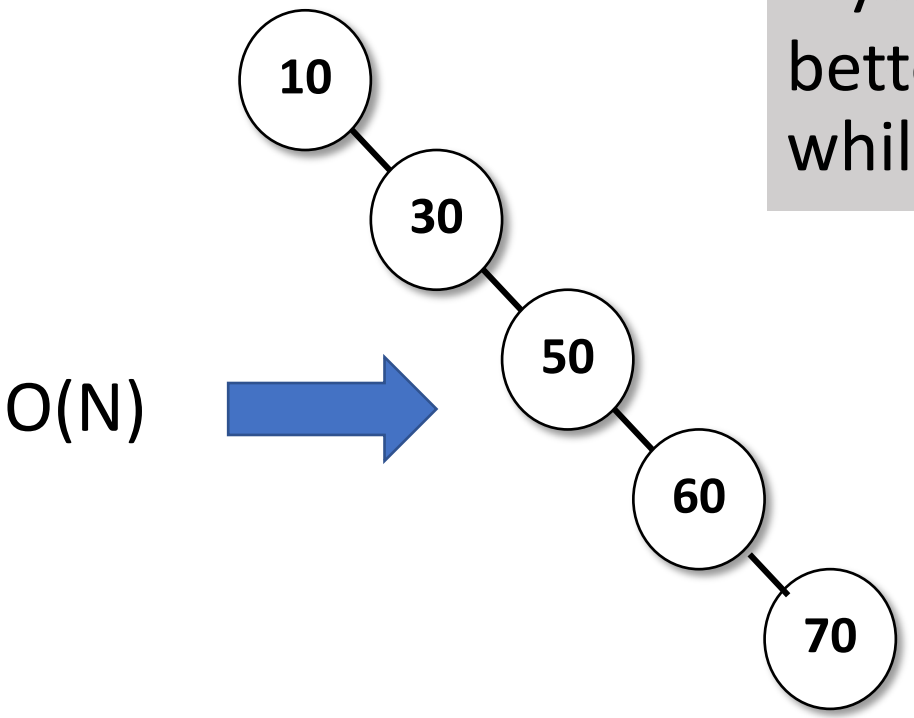
O(log(N))

Searching Path to “29”

Searching Path to “68” (not found)

Binary Search Tree V.S. Heap

Heap guarantees that elements on higher levels are greater (for max-heap) or smaller (for min-heap) than elements on lower levels, whereas BST guarantees order (from "left" to "right").



If you want sorted elements, go with BST. Heap is better at findMin/findMax (O(1)),
while BST is expected to good at all finds (O(logN))

Balanced Binary Search Trees

This tree is called an **AVL tree** and is named for its inventors: G.M. Adelson-Velskii and E.M. Landis.

AVL tree is a special kind of binary search tree that automatically makes sure that the tree remains balanced at all times.

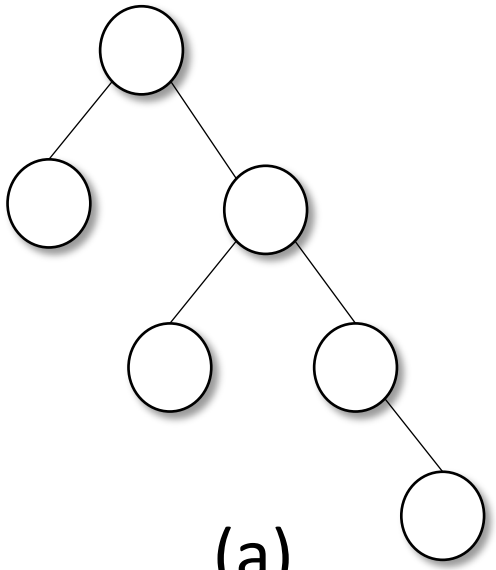
AVL tree need to keep track of a balance factor for each node in the tree. By define the balance factor for a node as the difference between the height of the **left subtree** and the height of the **right subtree**.

BalanceFactor = **height**(leftSubTree) – **height**(rightSubTree)

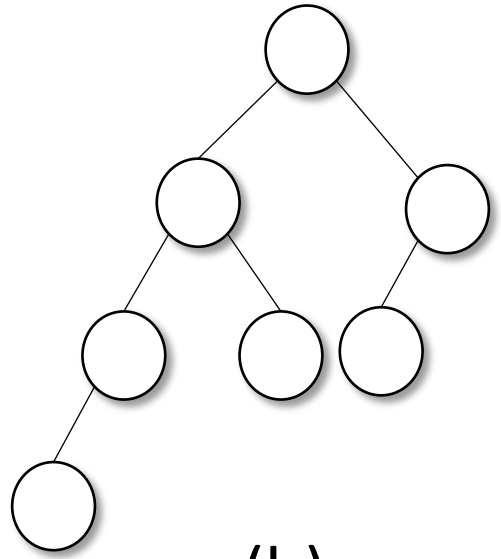
Using the definition for balance factor, said that

- **the subtree is left-heavy if the balance factor is greater than zero.**
- **the subtree is right-heavy if the balance factor is less than zero.**
- **If the balance factor is zero then the tree is perfectly in balance.**

** Purposes: **Balance Factor is -1, 0, or 1** **



(a)

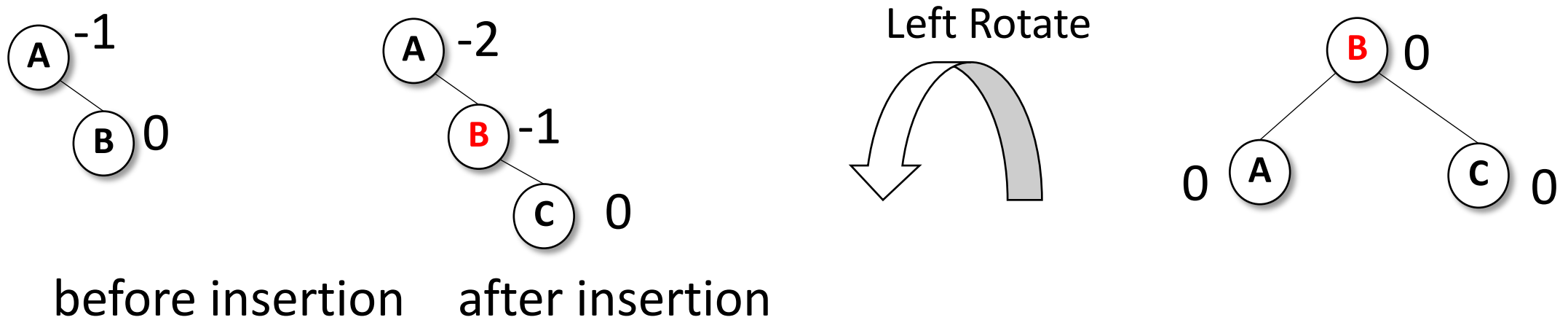


(b)

BalanceFactor = **height**(leftSubTree) – **height**(rightSubTree)

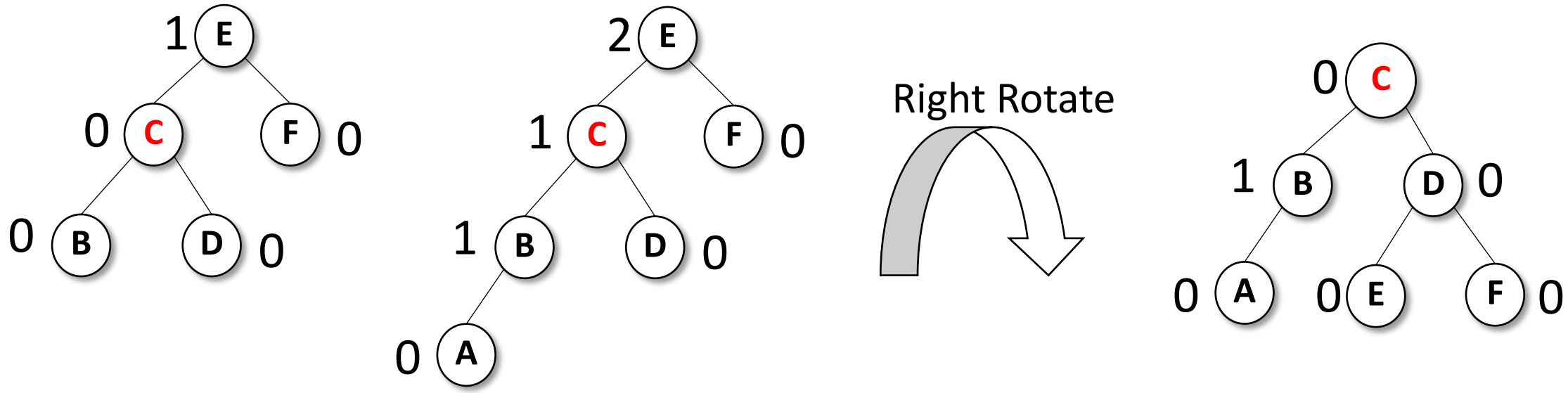
Rebalancing method

If an insertion of a new node make an AVL unbalanced, rebalancing method is necessary to making the AVL Tree work with efficiency performance. One or more **rotation**s are performed on the tree.



before insertion after insertion

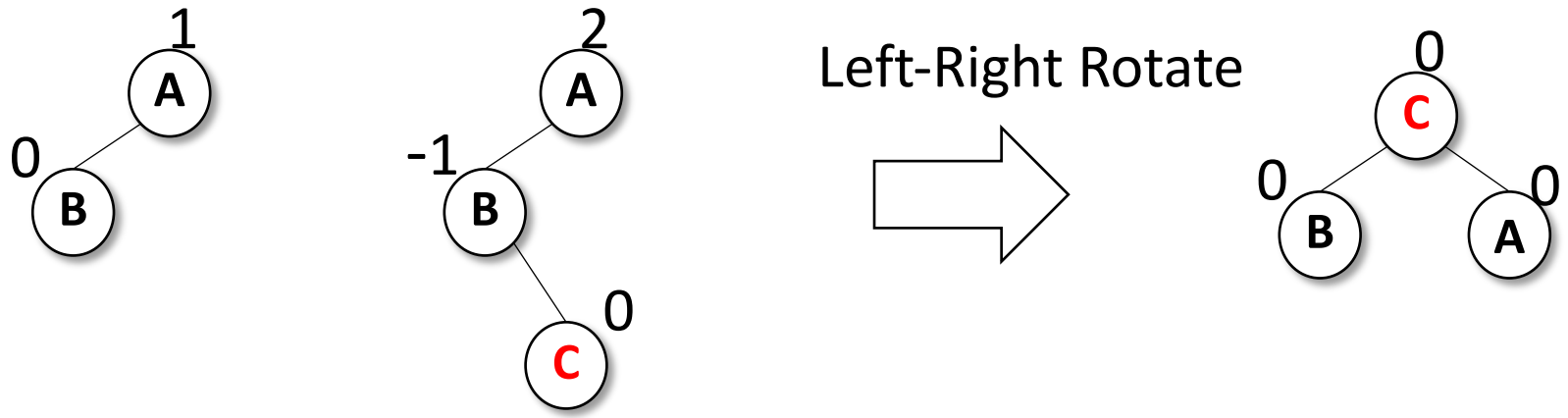
- Promote the right child (B) to be the root of the subtree.
- Move the old root (A) to be the left child of the new root.
- If new root (B) already had a left child then make it the right child of the new left child (A).
  Note: Since the new root (B) was the right child of A the right child of A is guaranteed to be empty at this point. This allows us to add a new node as the right child without any further consideration.



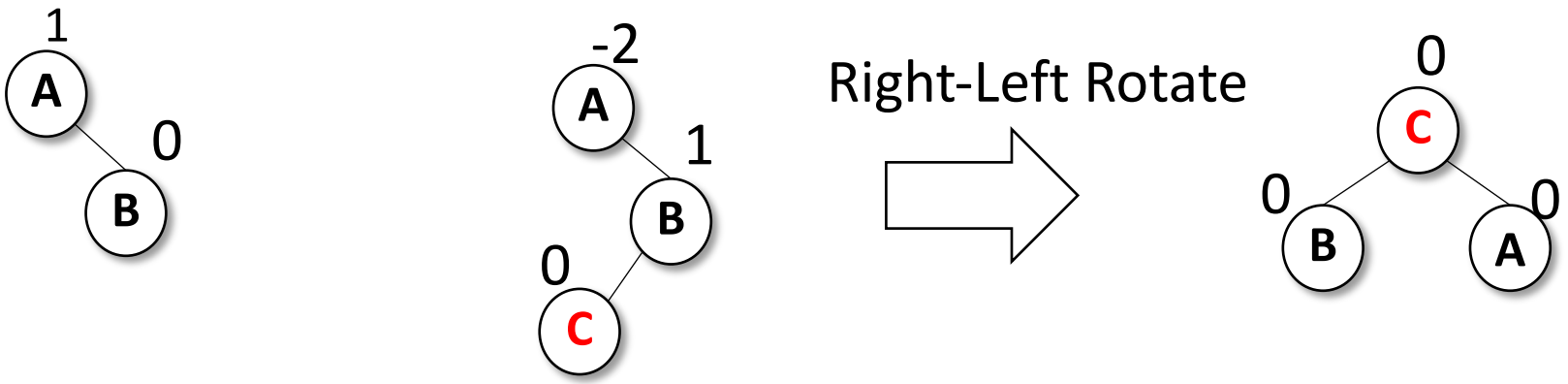
before insertion after insertion

- Promote the left child (C) to be the root of the subtree.
- Move the old root (E) to be the right child of the new root.
- If the new root(C) already had a right child (D) then make it the left child of the new right child (E).
  Note: Since the new root (C) was the left child of E, the left child of E is guaranteed to be empty at this point. This allows us to add a new node as the left child without any further consideration

Double Rotation



before insertion after insertion



before insertion after insertion