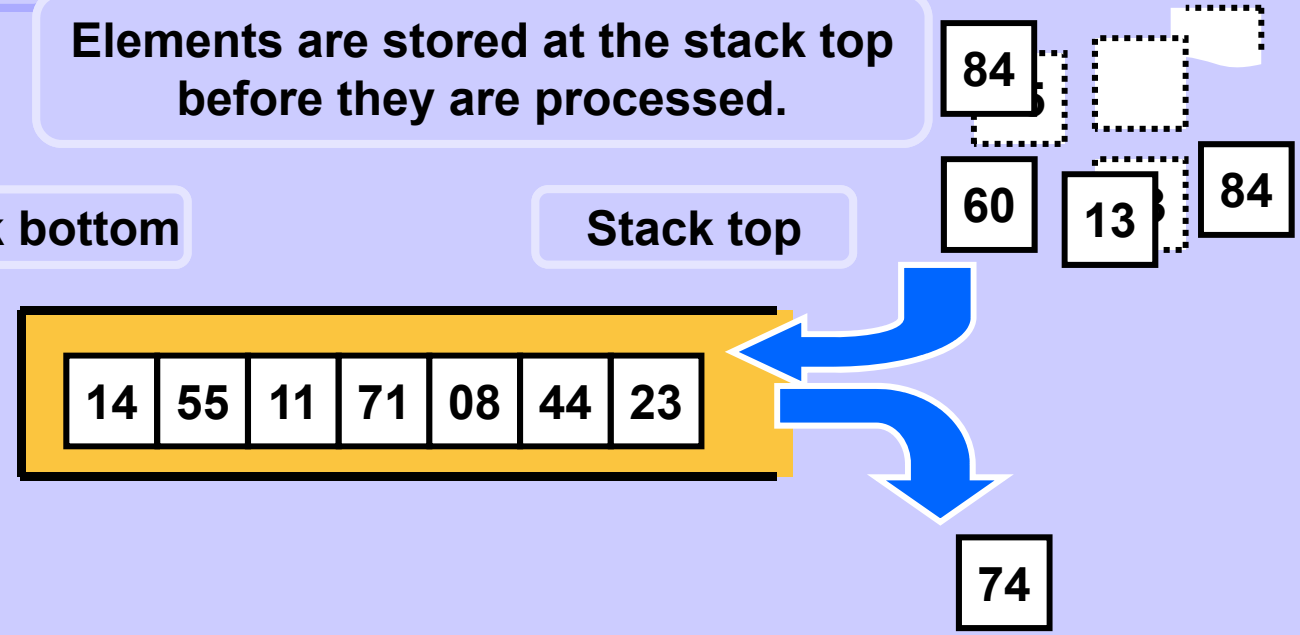
Stack
Operations pop, push, Empty...

Queue
Operations Enqueue, Dequeue, Front, Empty....
Cyclic queue implementation

Breadth-first search (BFS) in a binary tree

Stack

Elements are stored at the stack top before they are processed.



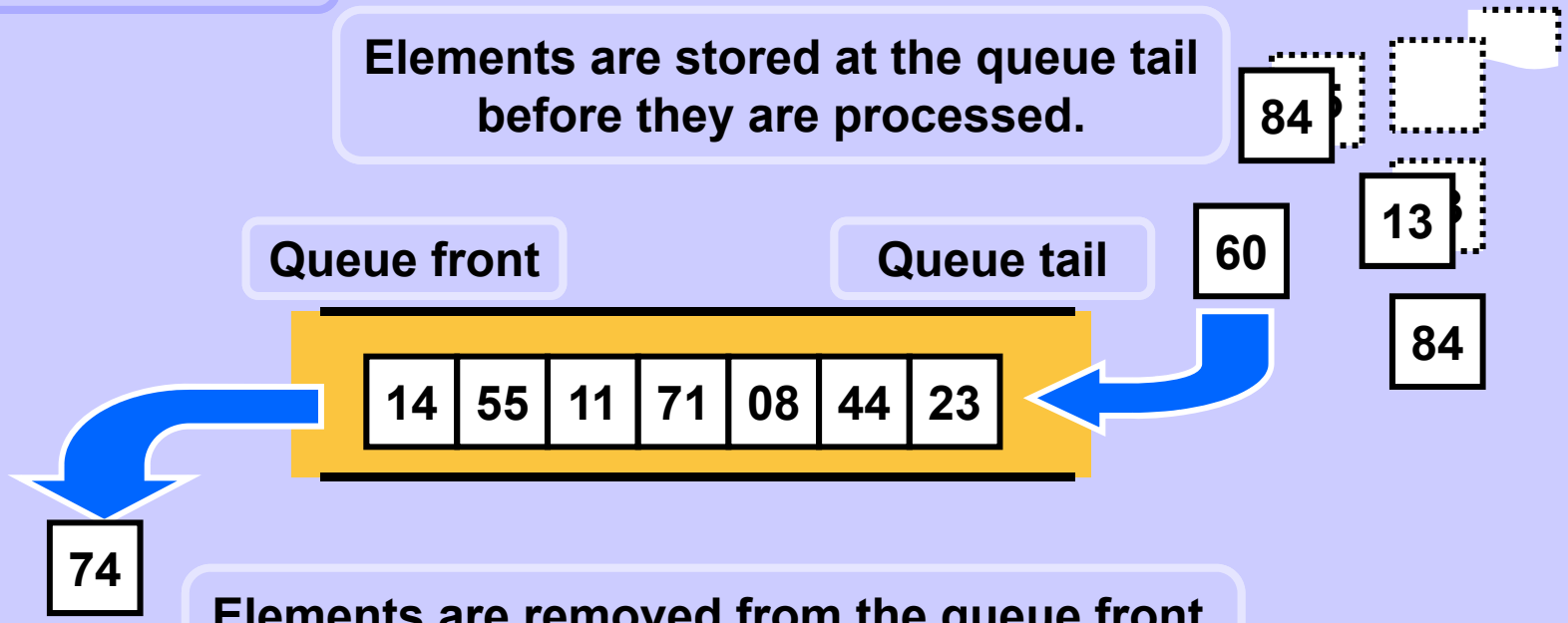
Elements are removed from the stack top and then they are processed.

Operation names

Put at the top	Push
Remove from the top	Pop
Read the top	Top
Is the stack empty?	Empty

Queue

Elements are stored at the queue tail before they are processed.



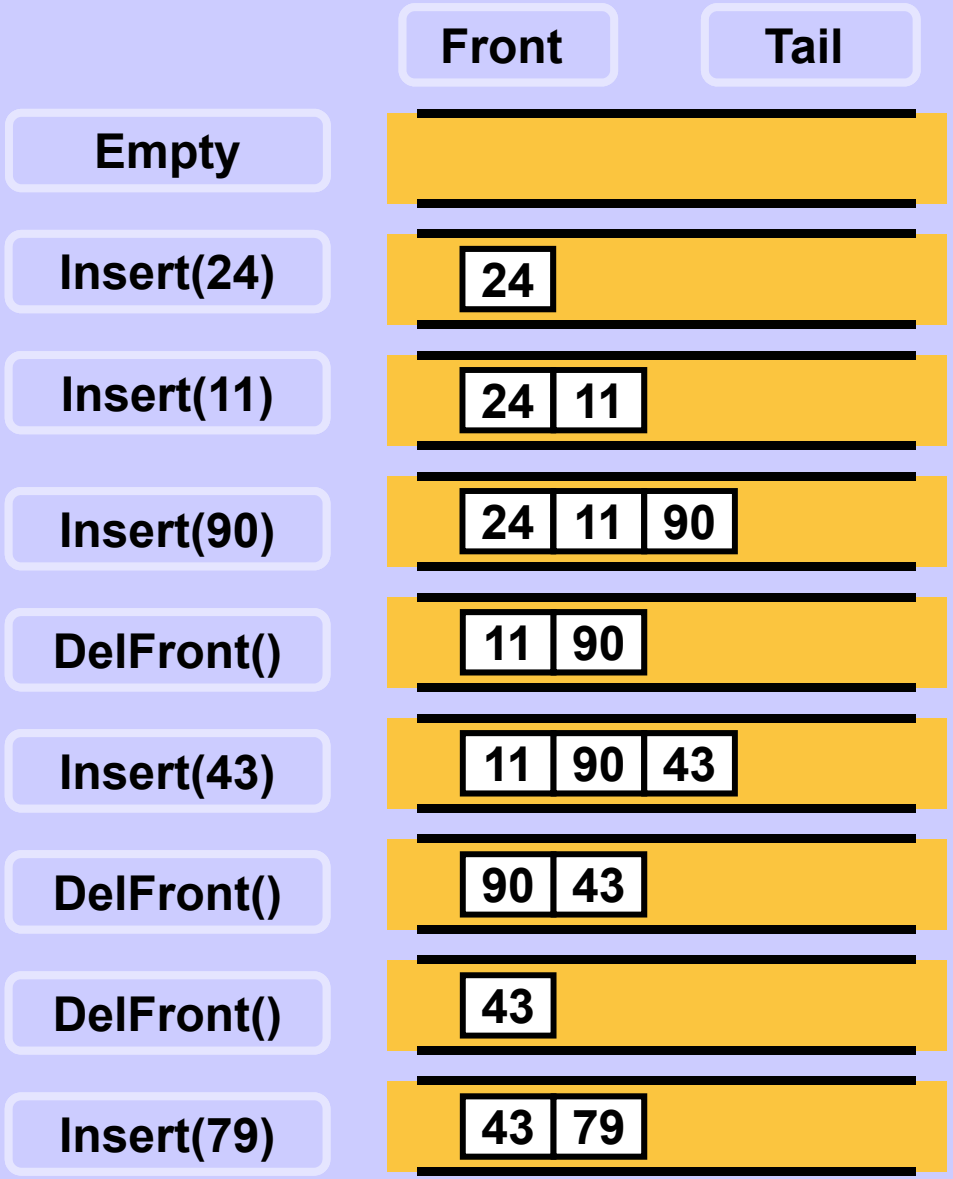
Elements are removed from the queue front and then they are processed.

Operation names

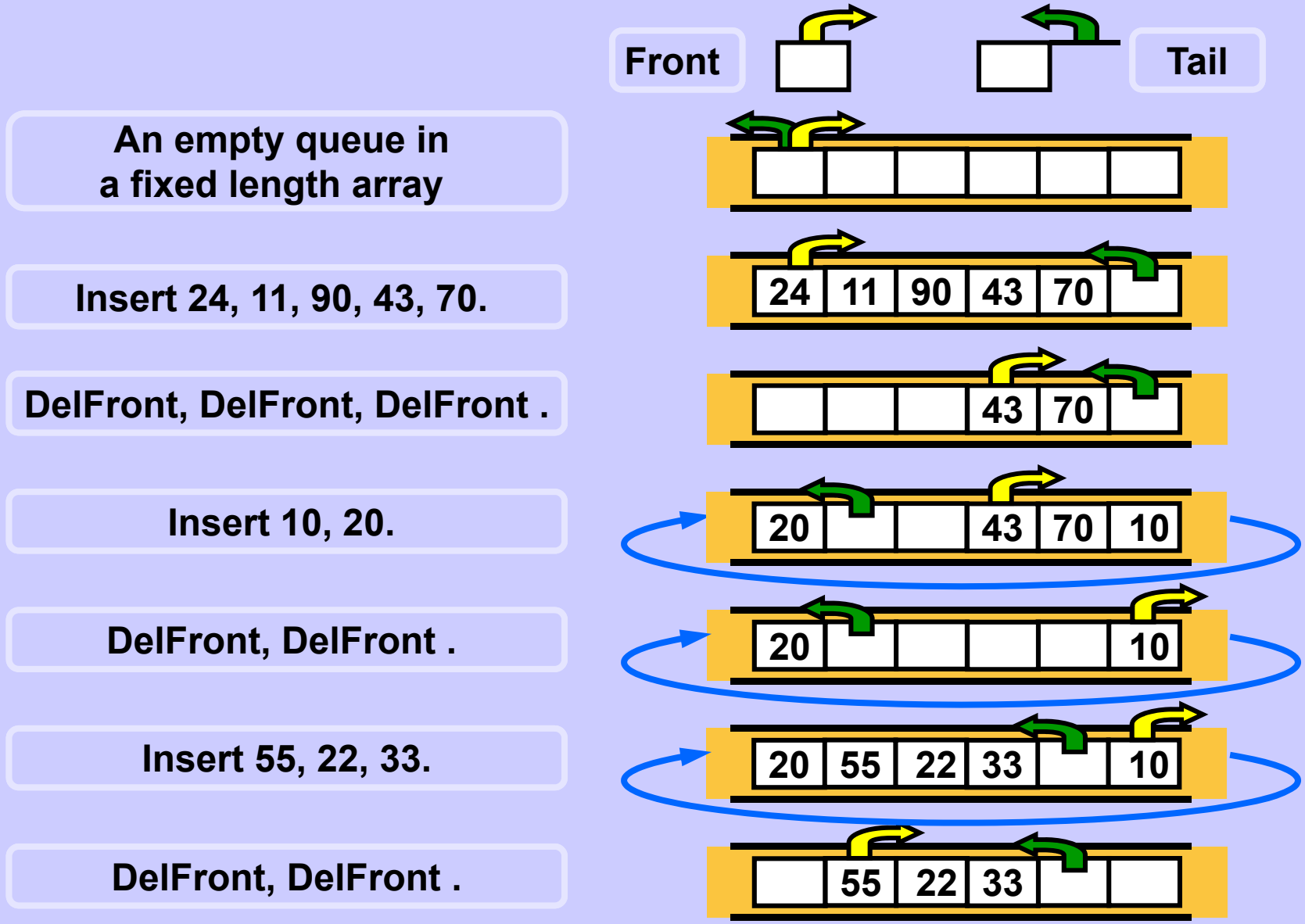
Insert at the tail	Enqueue / InsertLast / Push ...
Remove from the front	Dequeue / delFront / Pop ...
Read the front elem	Front / Peek ...
Is the queue empty?	Empty

Queue

Easy example of a queue life cycle.



Cyclic queue implementation in an array



Cyclic queue implementation in an array

Tail index points to the first free position behind the last queue element.
Front index points to the first position occupied by a queue element.
When both indices point to the same position the queue is empty.

```
class Queue:
    def __init__(self, sizeOfQ):
        self.size = sizeOfQ
        self.q = [None] * sizeOfQ
        self.front = 0
        self.tail = 0

def isEmpty(self):
    return (self.tail == self.front)

def Enqueue(self, node):
    if self.tail+1 == self.front or \
        self.tail - self.front == self.size-1:
            pass                    # implement overflow fix here
    self.q[self.tail] = node
    self.tail = (self.tail + 1) % self.size
```

Continue...

Cyclic queue implementation in an array

Tail index points to the first free position behind the last queue element.
Front index points to the first position occupied by a queue element.
When both indices point to the same position the queue is empty.

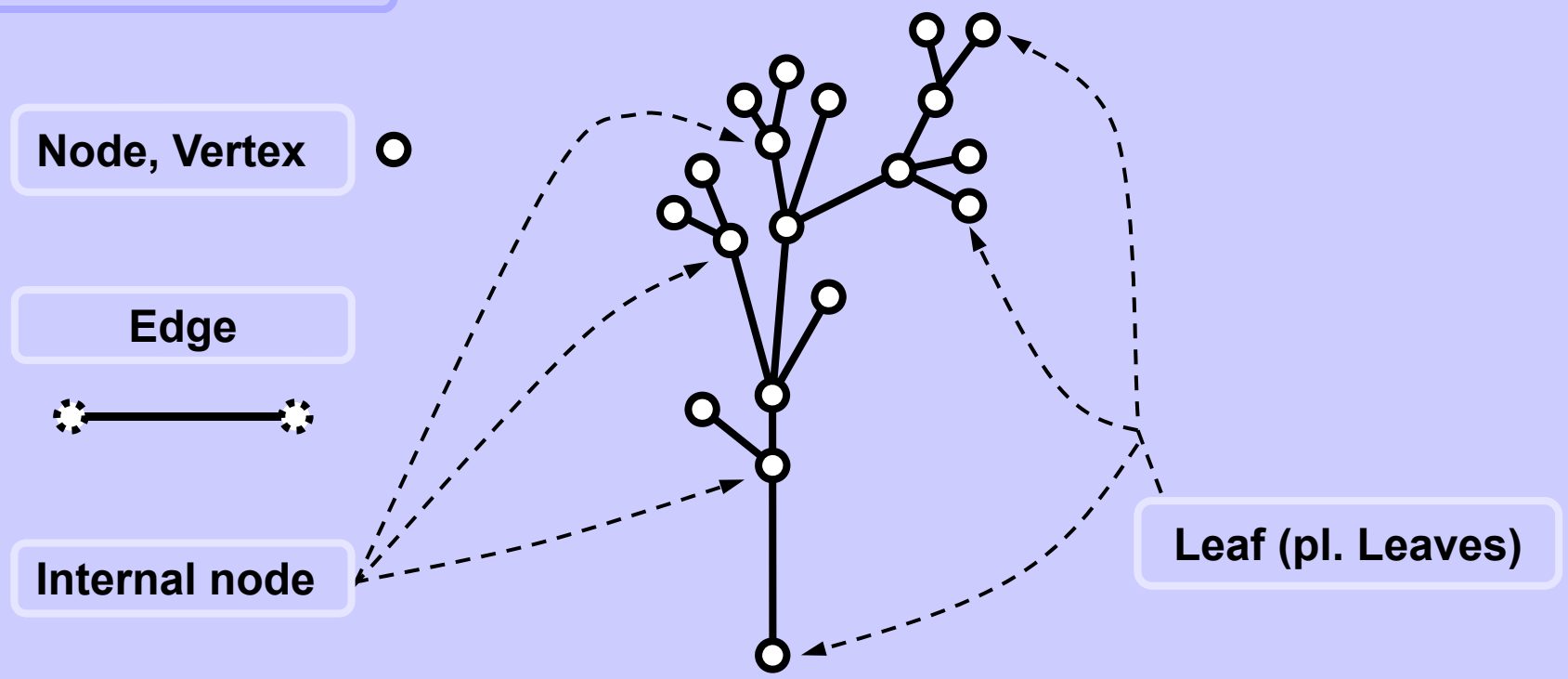
... continued

```
def Dequeue(self):
    node = self.q[self.front]
    self.front = (self.front + 1) % self.size
    return node

def pop(self):
    return self.Dequeue()

def push(self, node):
    self.Enqueue(node)
```

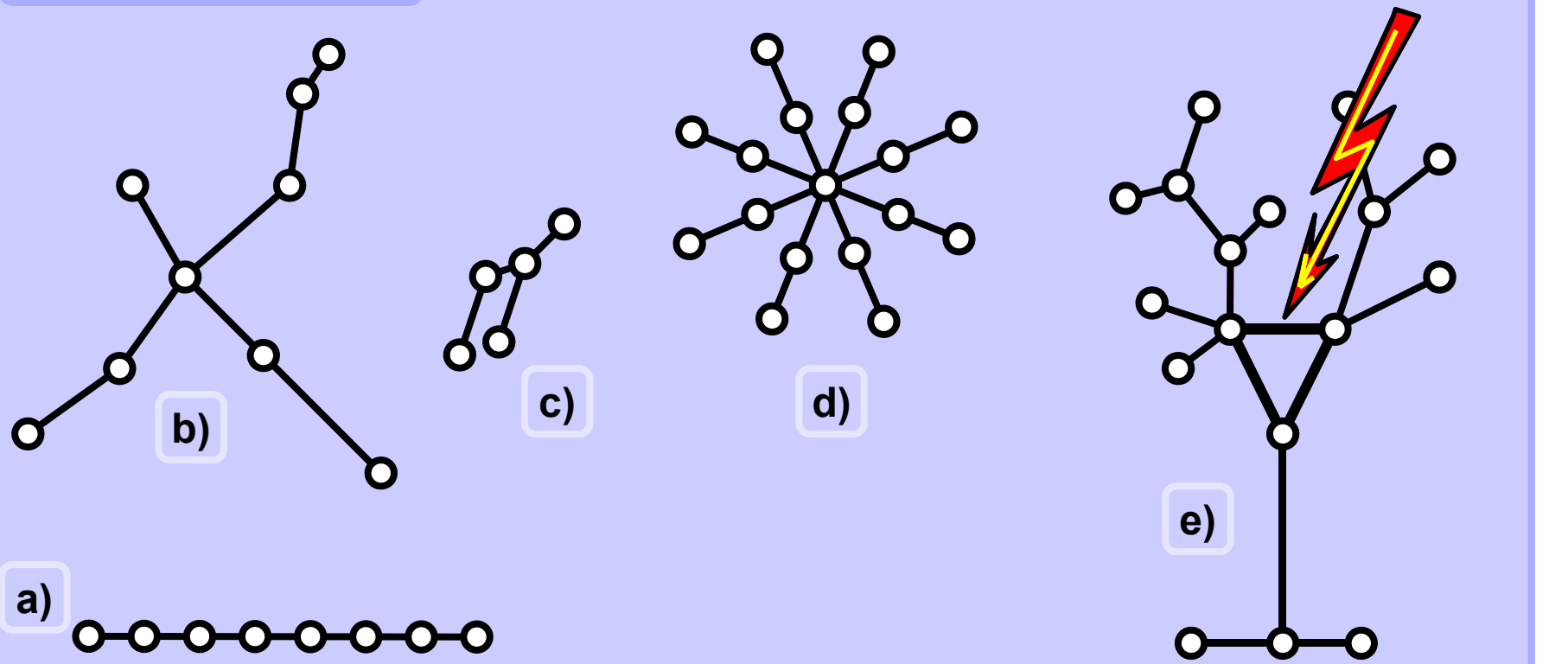
TREES, BINARY TREES

Traversing a tree with Breadth-First Search (BFS)

Tree



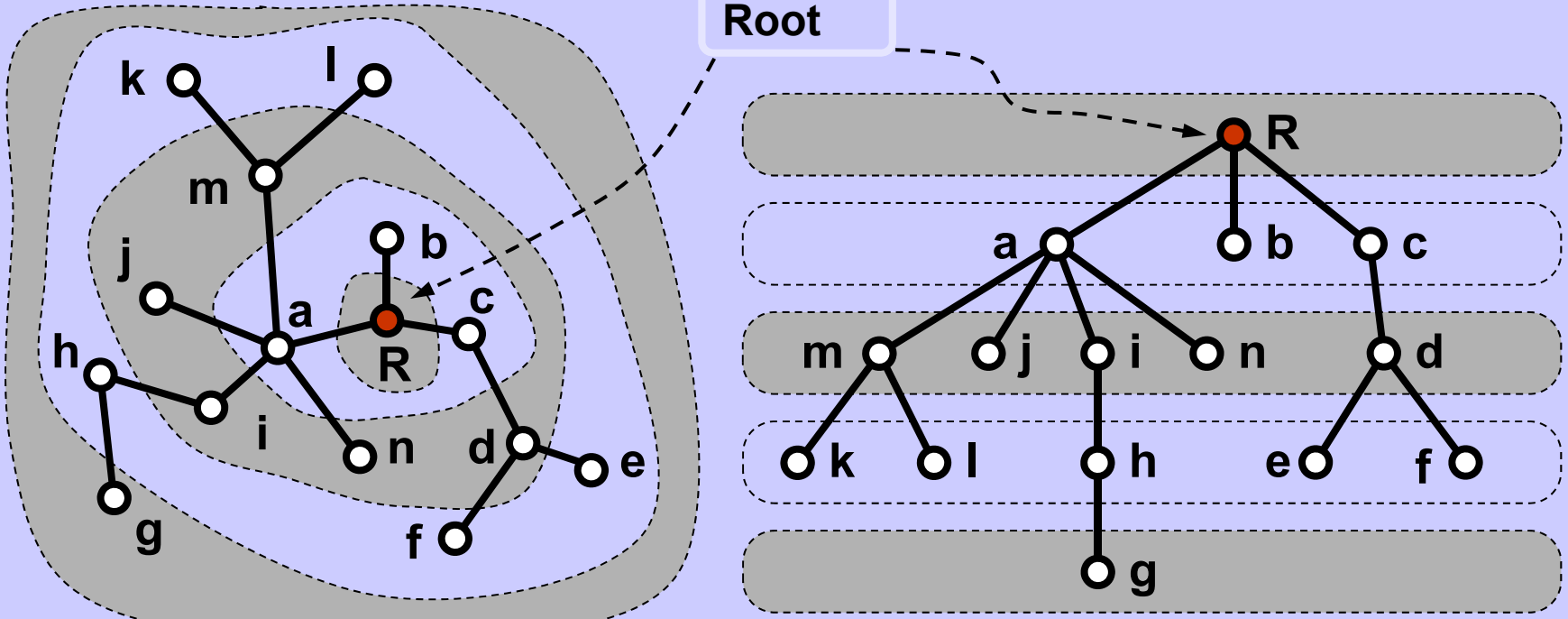
Tree examples



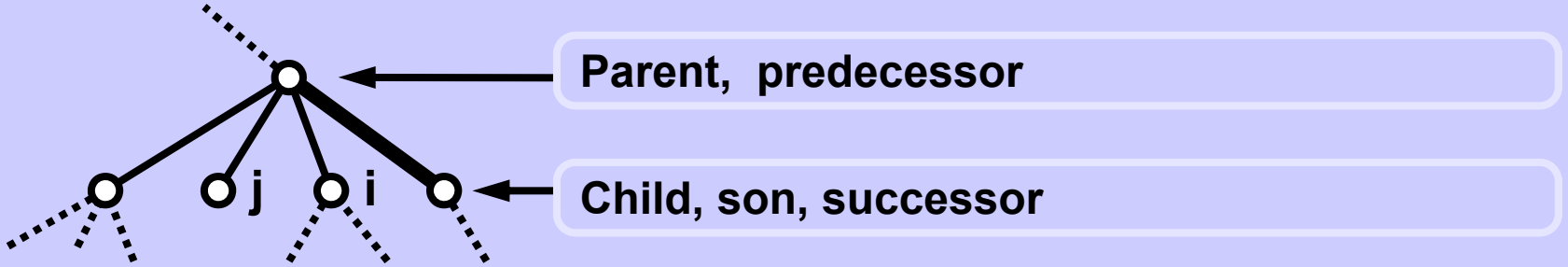
Tree properties

1. A tree is connected, there is a path between each its two nodes.
2. There is exactly one path path between any of its two nodes.
3. Removing any edge results in tree divided into two separate parts.
4. Number of edges is always less by one than the number of nodes.

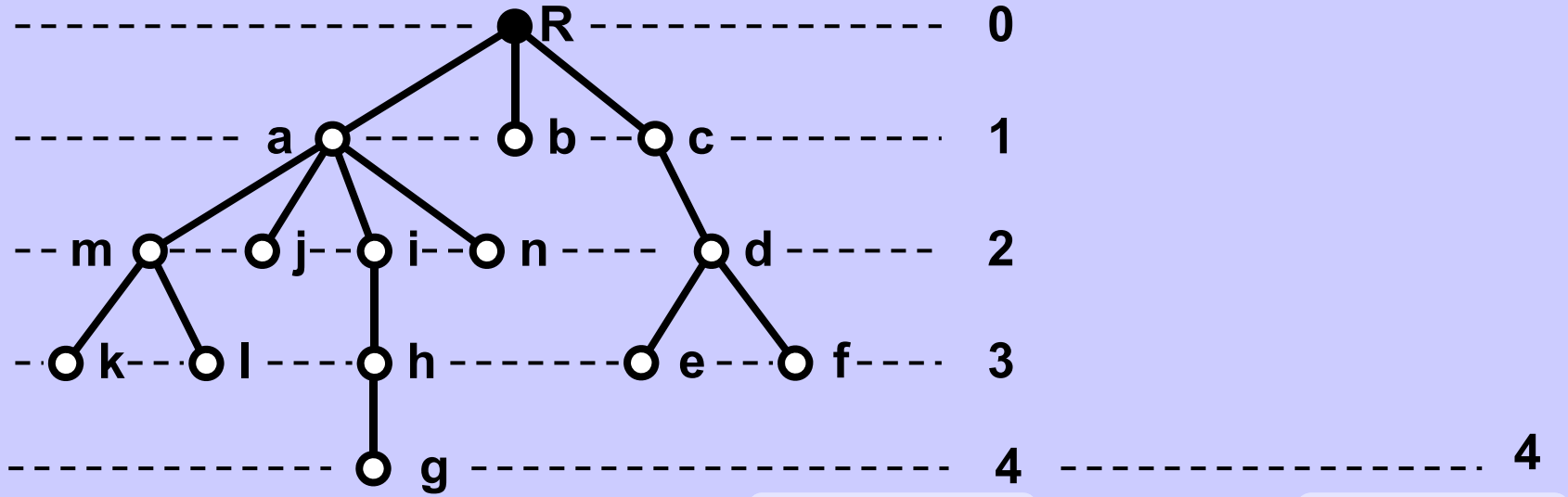
Rooted tree



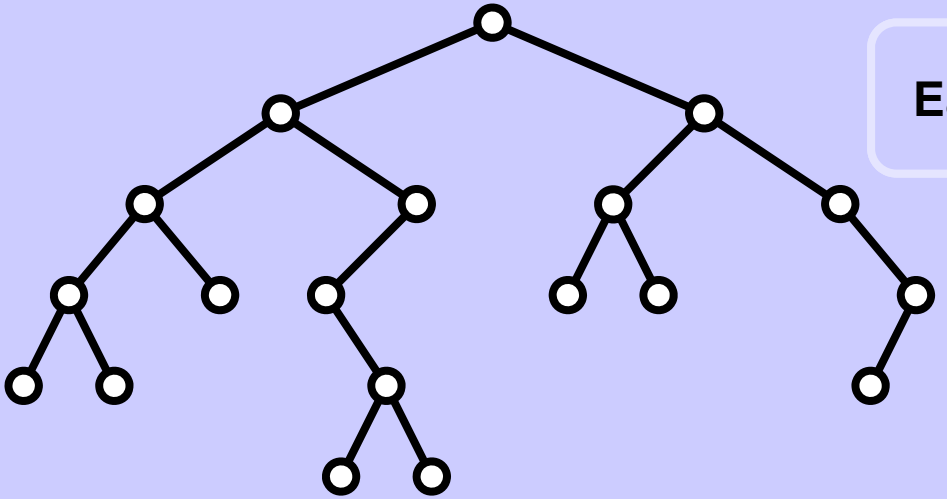
Terminology



Tree depth

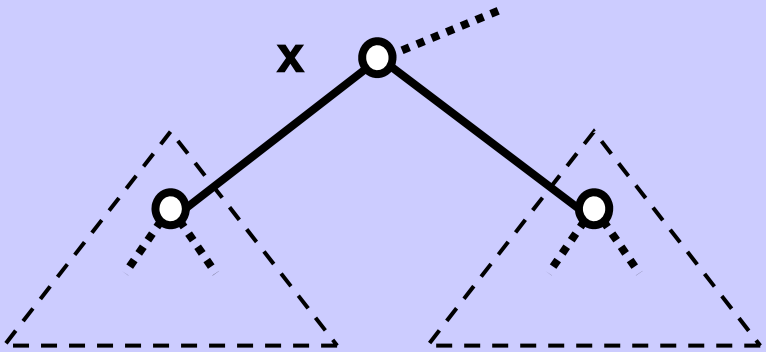


Binary (rooted!!) tree



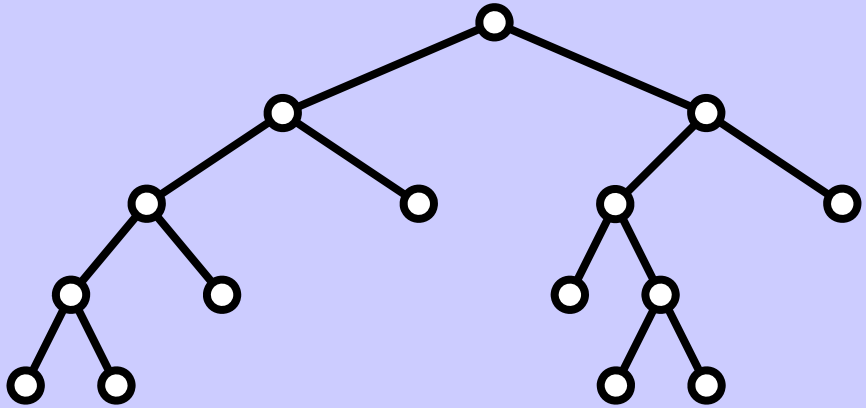
Each node has 0 or 1 or 2 children.

Left and right subtree



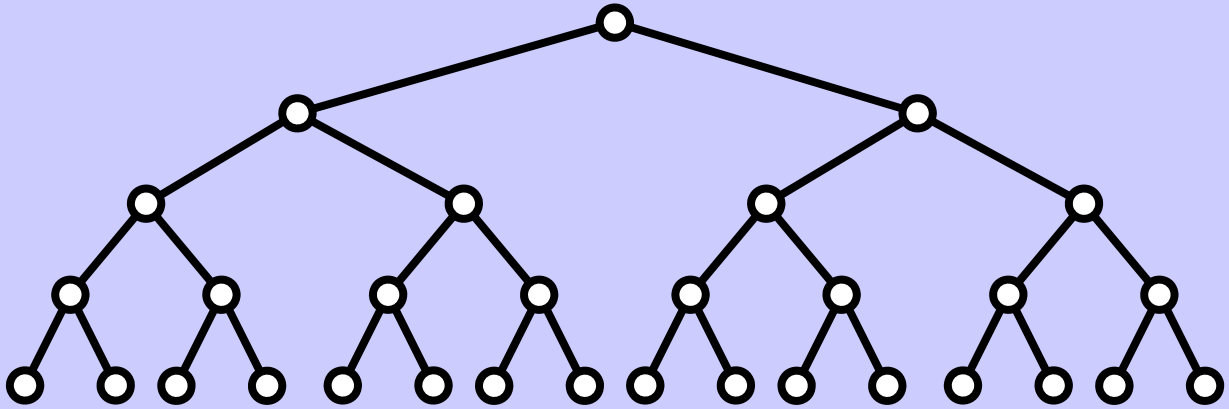
Subtree of node x ..... left ..... right

Regular binary tree



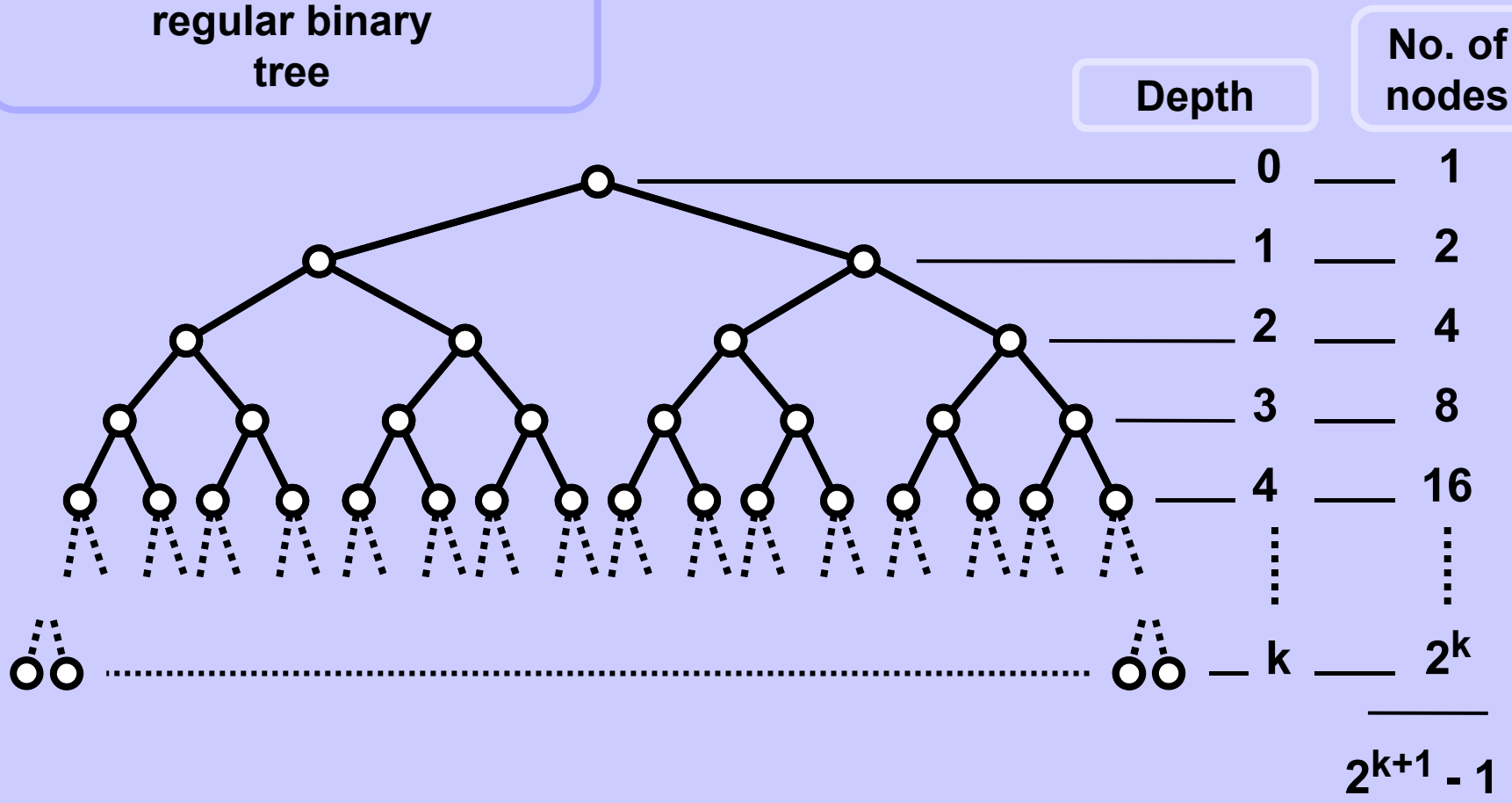
Each node has 0 or 2 children.
Not 1 child

Balanced tree



The depths of all leaves are (approximately) the same.

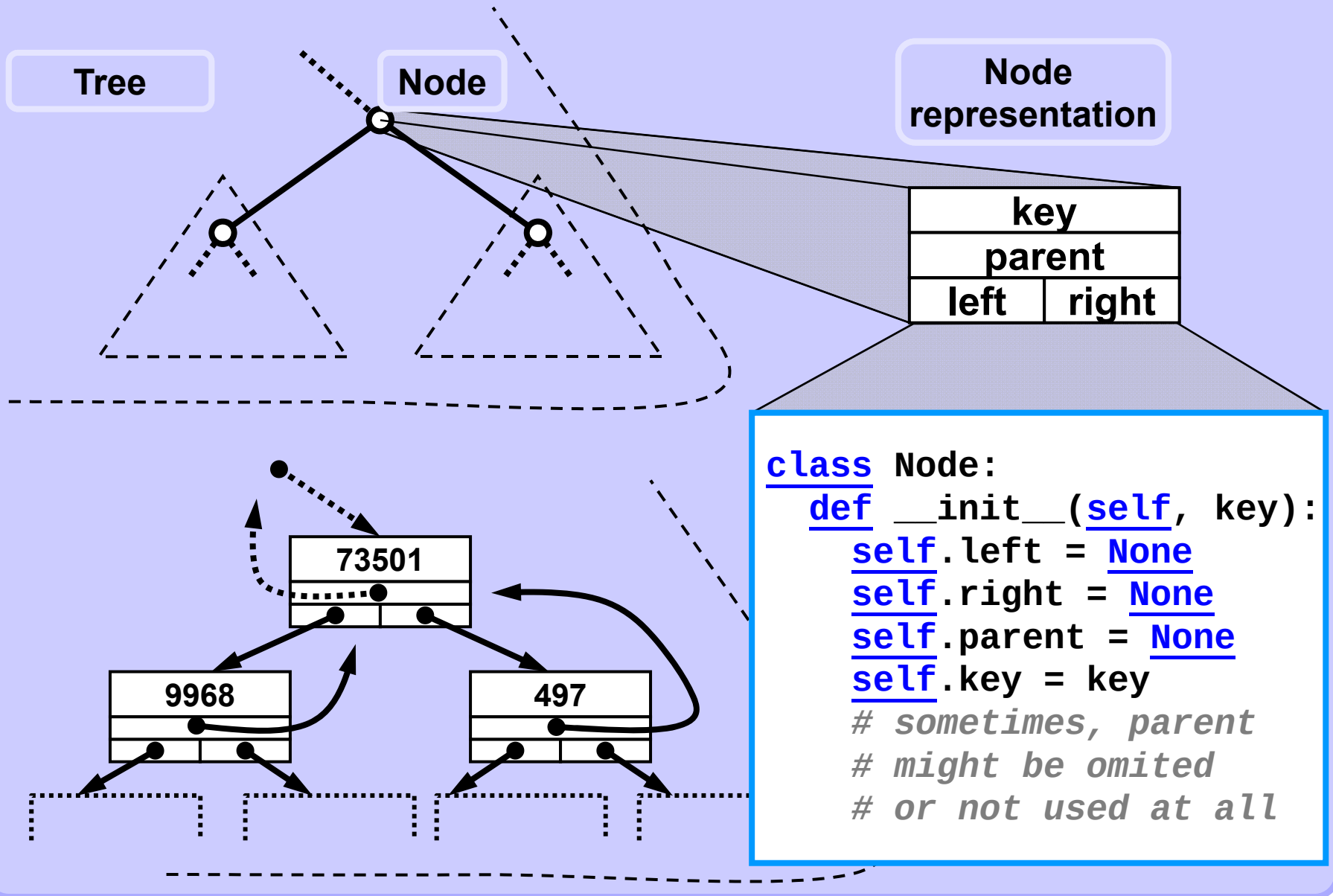
Depth of a balanced regular binary tree



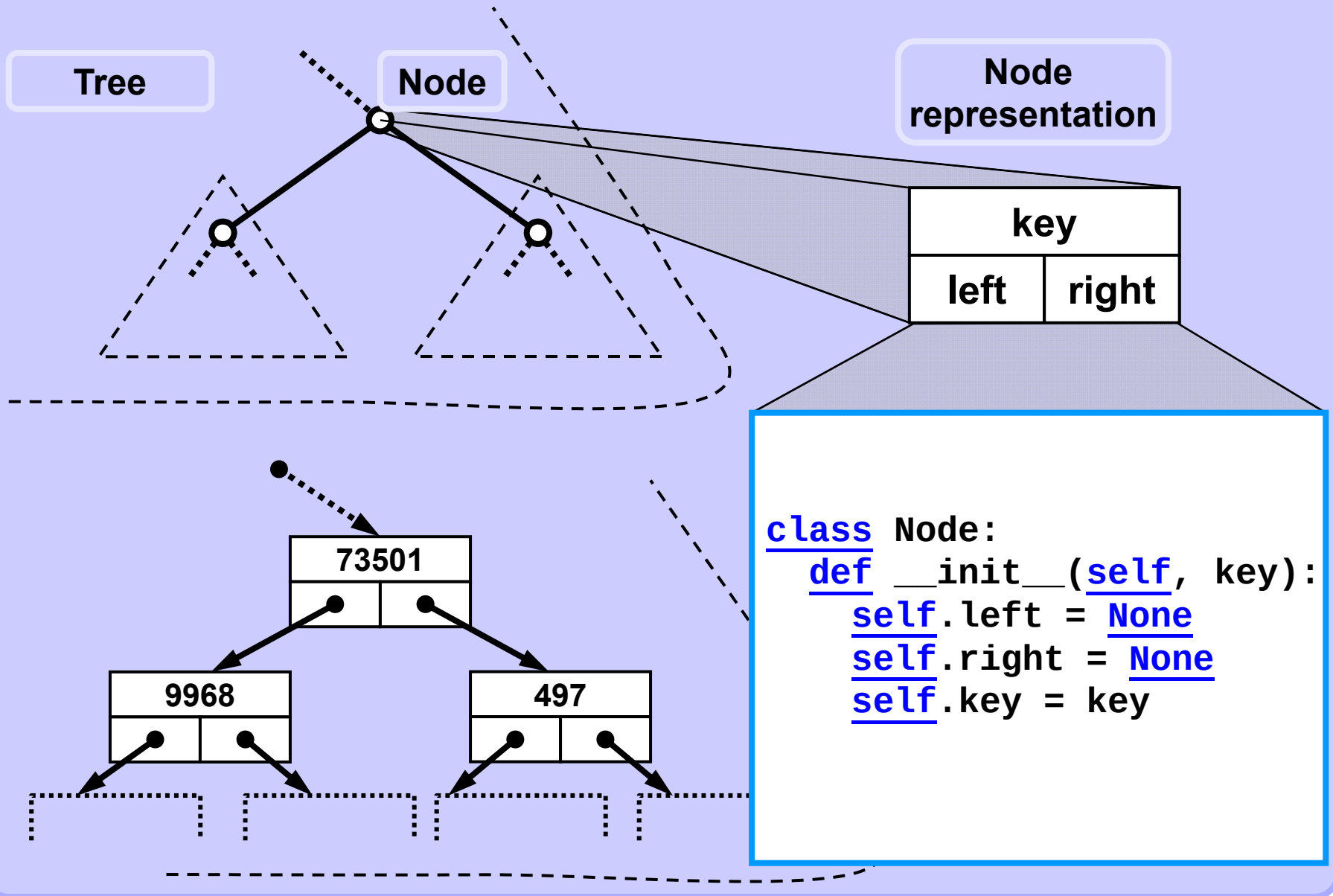
$(2^{depth+1} - 1)$ ~ no. of nodes

Depth ~ $\log_2(|nodes|+1) - 1$ ~ $\log_2(|nodes|)$

Binary tree implementation -- Python



Binary tree implementation -- Python



Build a random binary tree -- Python

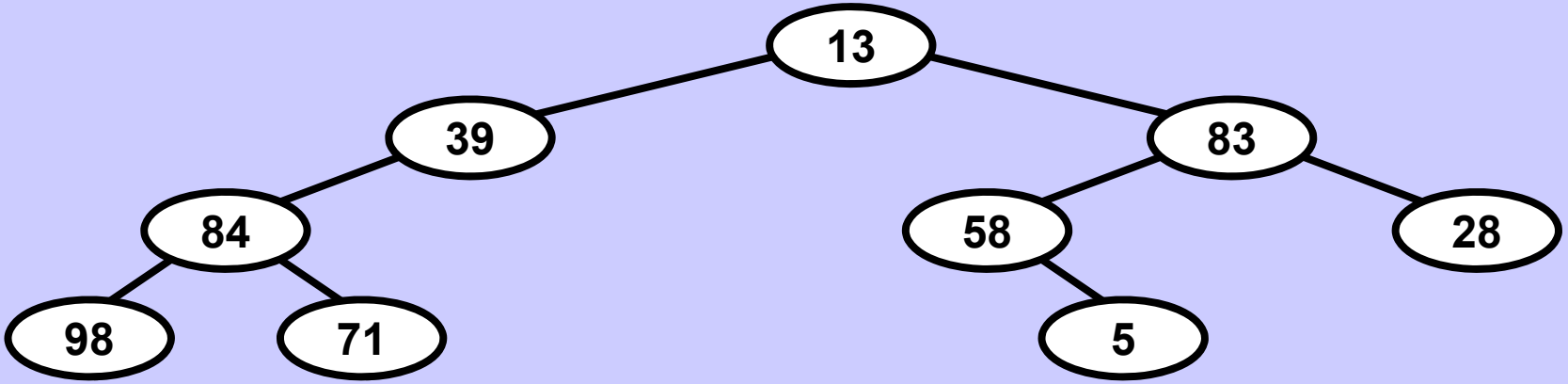
```
@staticmethod # Binary tree calls this method
def rndTree( depth ):
    if depth <= 0 or random.randrange(10) > 7 :
        return None
    newnode = Node( 10+random.randrange(90) )
    newnode.left = Node.rndTree( depth-1 )
    newnode.right = Node.rndTree( depth-1 )
    return newnode
```

Example of function call

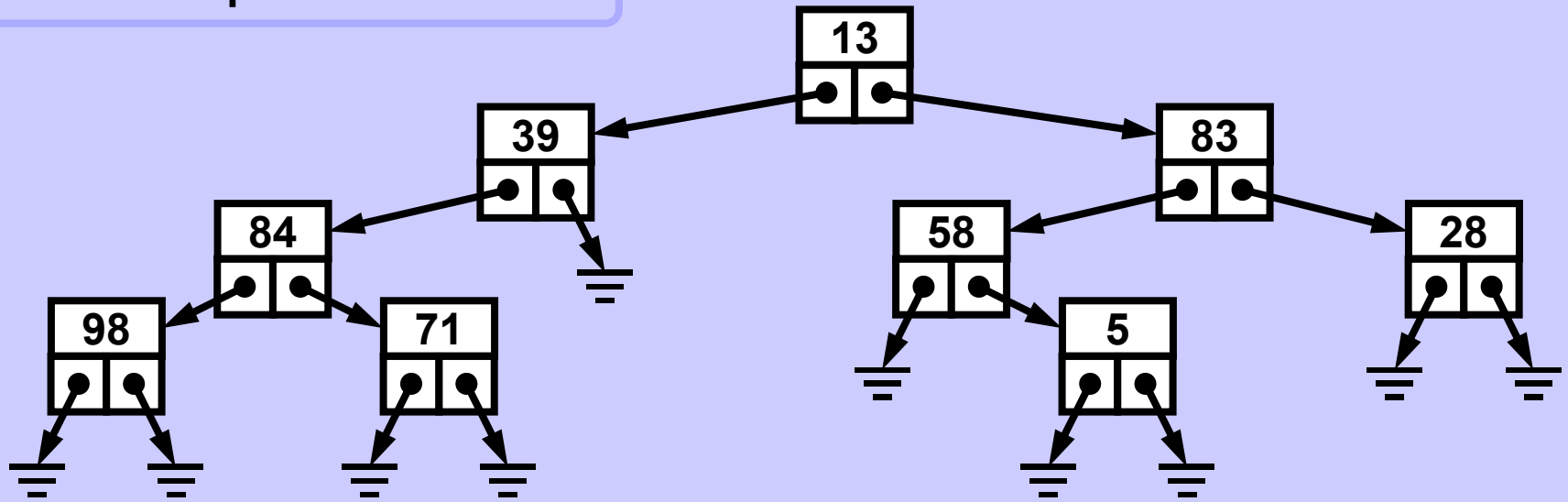
```
tree1 = BinaryTree()
tree1.randomTree(4)
```

Note. A call random.randrange(n) returns a pseudorandom integer in the range from 0 to n-1. Function random() is not implemented here.

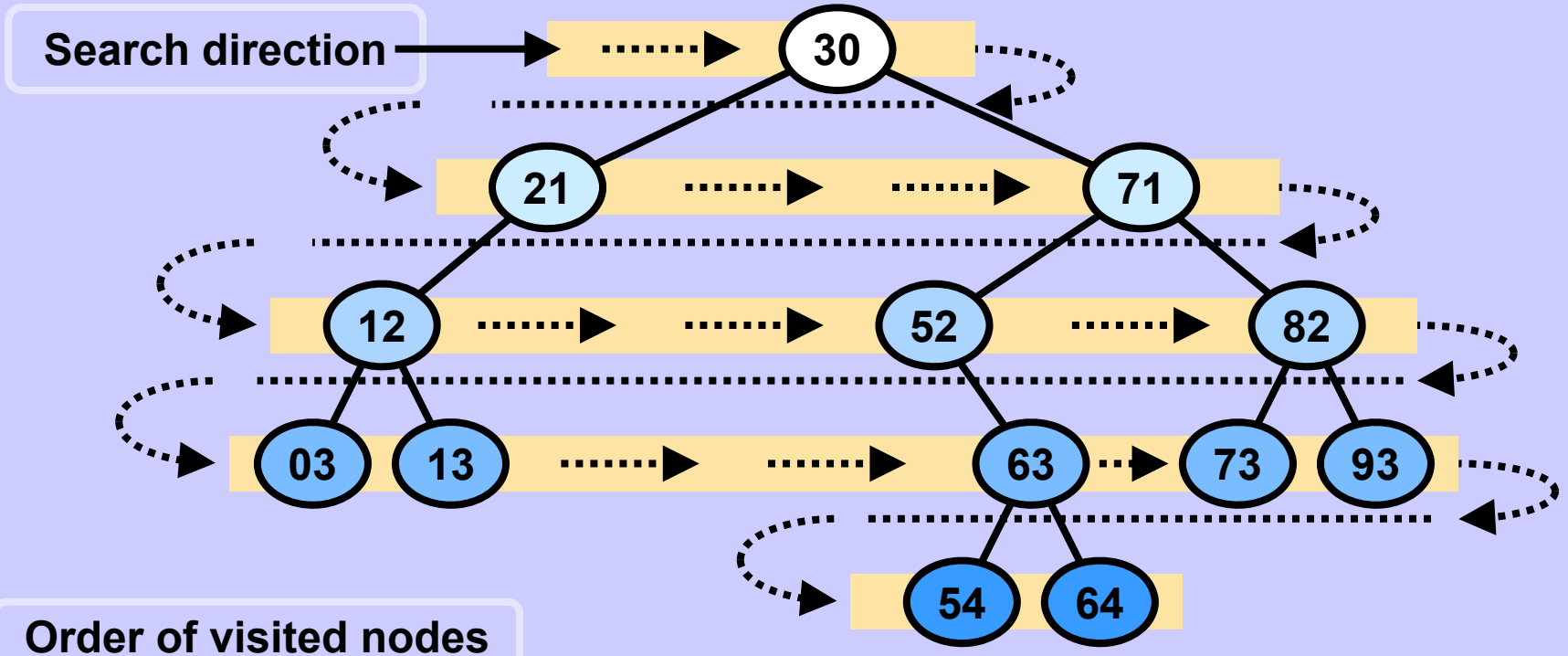
Random binary tree



Tree representation



Breadth-first search (BFS) of a tree



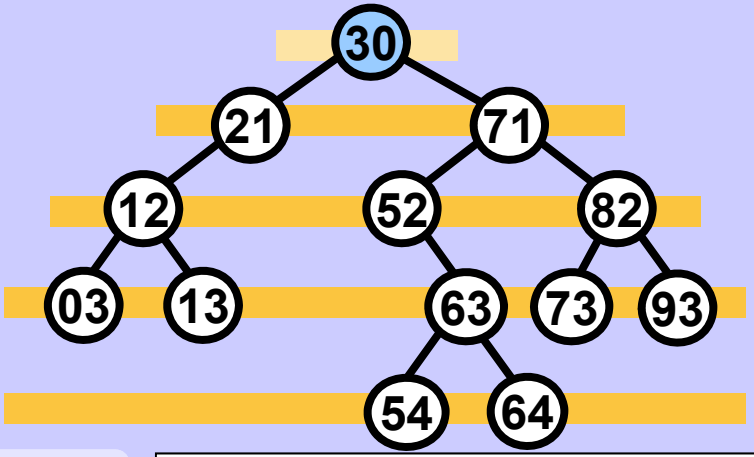
Order of visited nodes

30	21	71	12	52	82	03	13	63	73	93	54	64

Nor the tree structure nor the recursion support this approach directly.

Breadth-first search (BFS) of a tree

Initialization



Output

Create an empty queue.

Enqueue the tree root.

Front

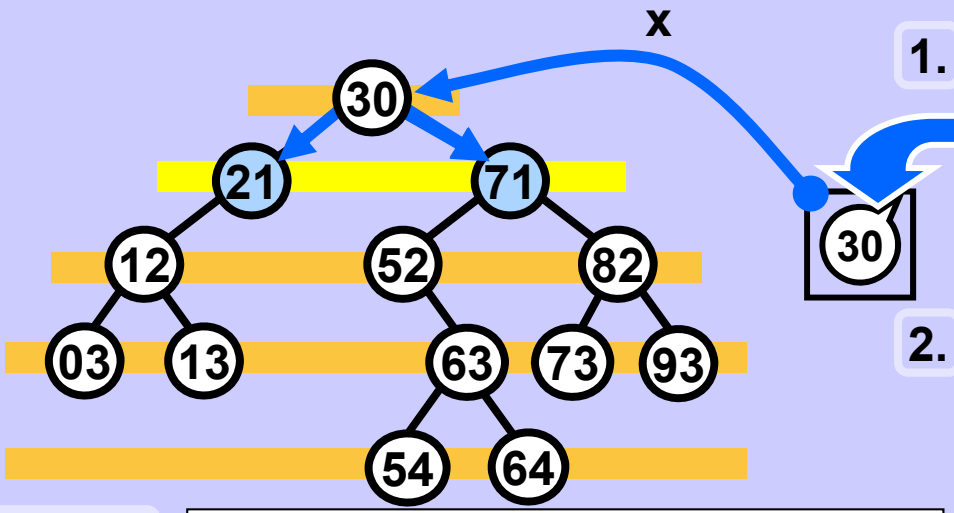
Tail

Main loop

While the queue is not empty do:

1. Remove the first element from the queue and process it.
2. Enqueue the children of removed element.

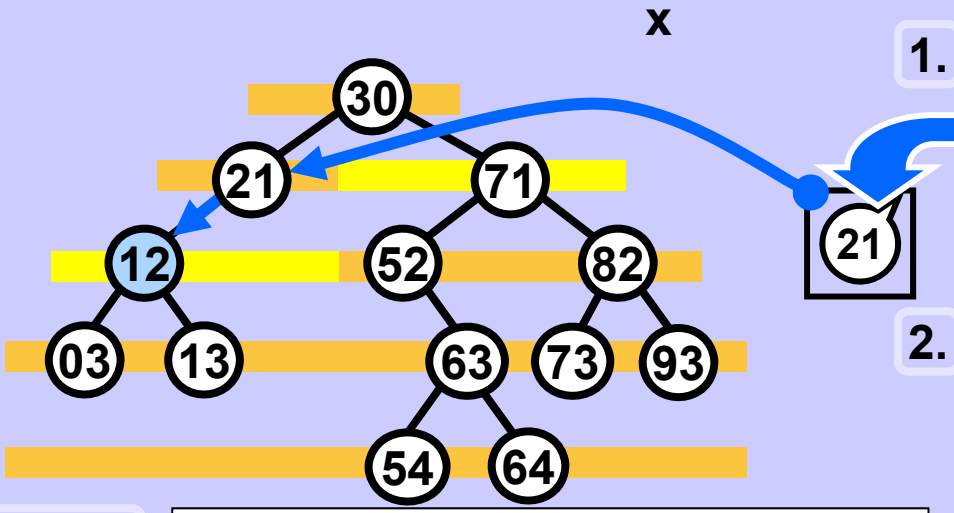
Breadth-first search (BFS) of a tree



1. x = Dequeue(), print (x.key).

2. Enqeue(x.left), Enqueue(x.right). *)

Output: 30



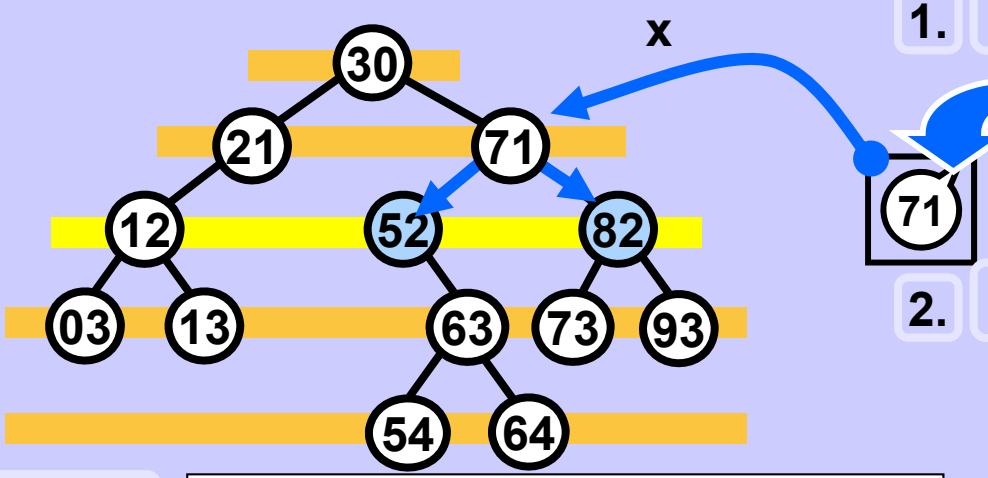
1. x = Dequeue(), print (x.key).

2. Enqeue(x.left), Enqueue(x.right). *)

Output: 30 21

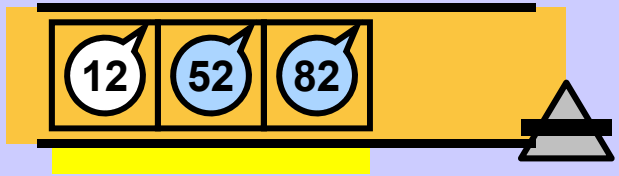
*) if exists

Breadth-first search (BFS) of a tree

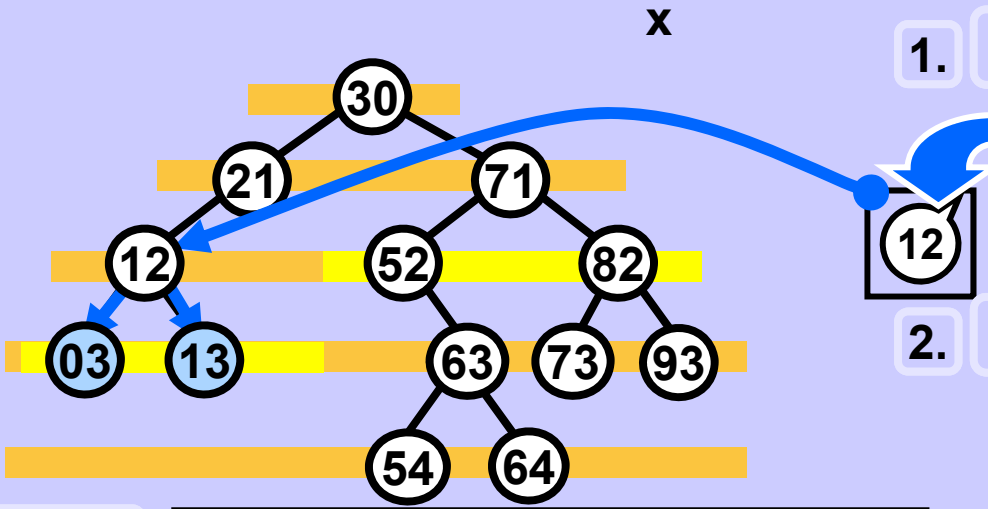


1. x = Dequeue(), print (x.key).

2. Enqeue(x.left), Enqueue(x.right). *)

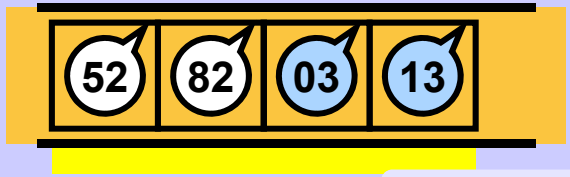


Output: 30 21 71



1. x = Dequeue(), print (x.key).

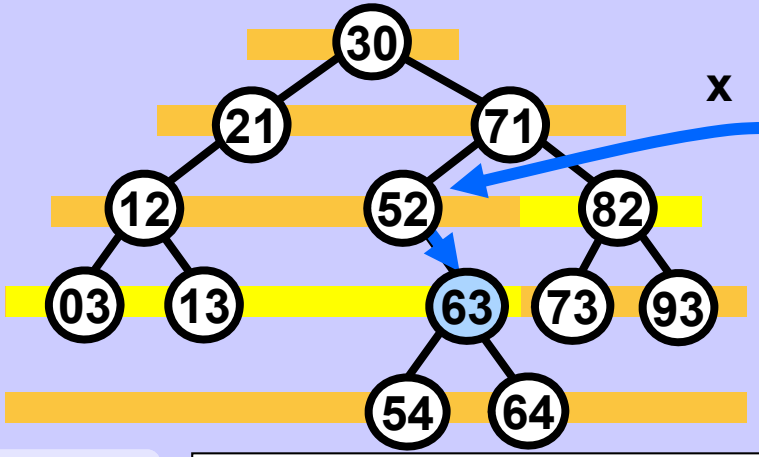
2. Enqeue(x.left), Enqueue(x.right). *)



Output: 30 21 71 12

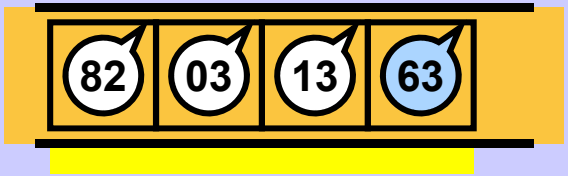
*) if exists

Breadth-first search (BFS) of a tree

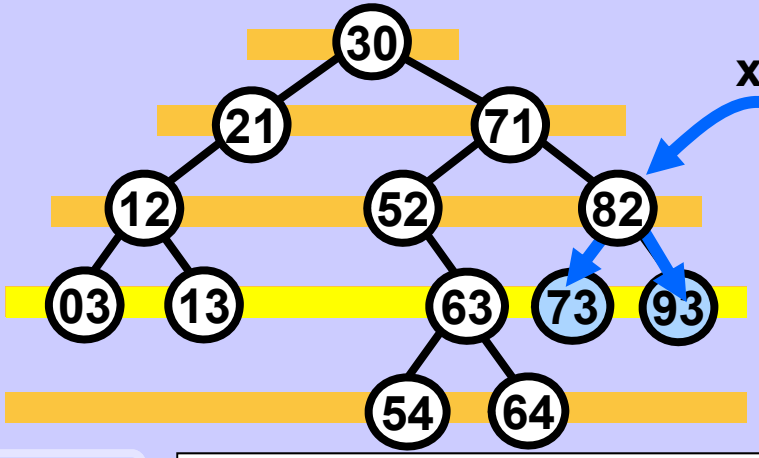


1. x = Dequeue(), print (x.key).

2. Enqeue(x.left), Enqueue(x.right). *)

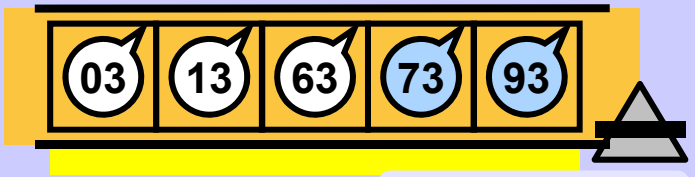


Output: 30 21 71 12 52



1. x = Dequeue(), print (x.key).

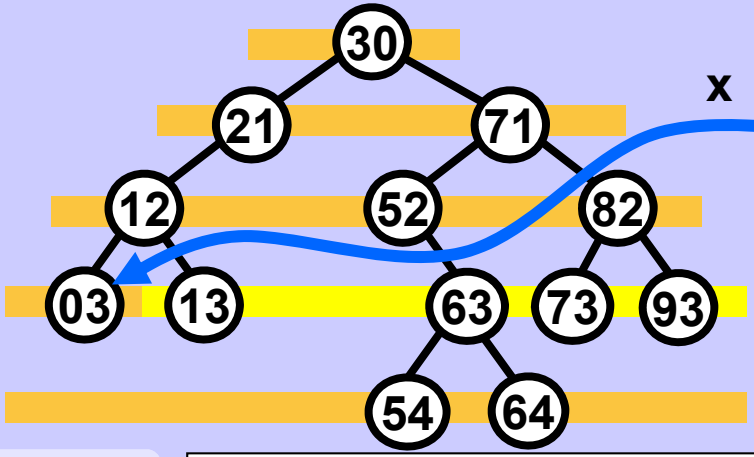
2. Enqeue(x.left), Enqueue(x.right). *)



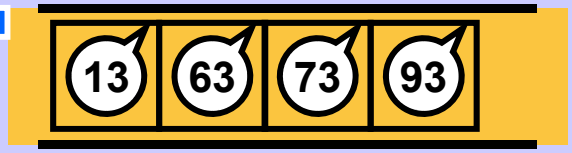
Output: 30 21 71 12 52 82

*) if exists

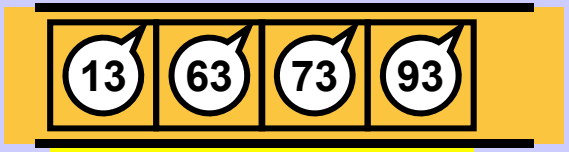
Breadth-first search (BFS) of a tree



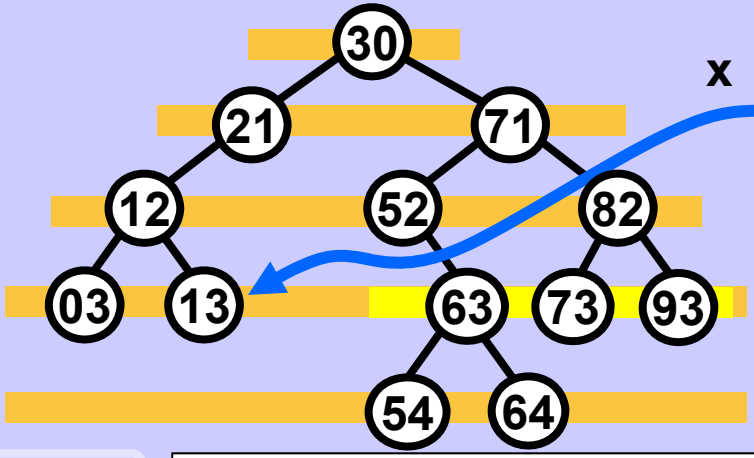
1. x = Dequeue(), print (x.key).



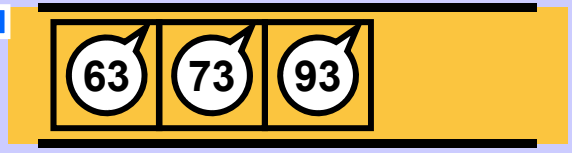
2. Enqeue(x.left), Enqueue(x.right). *)



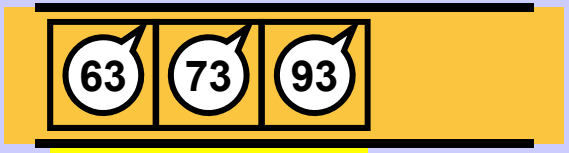
Output: 30 21 71 12 52 82 03



1. x = Dequeue(), print (x.key).



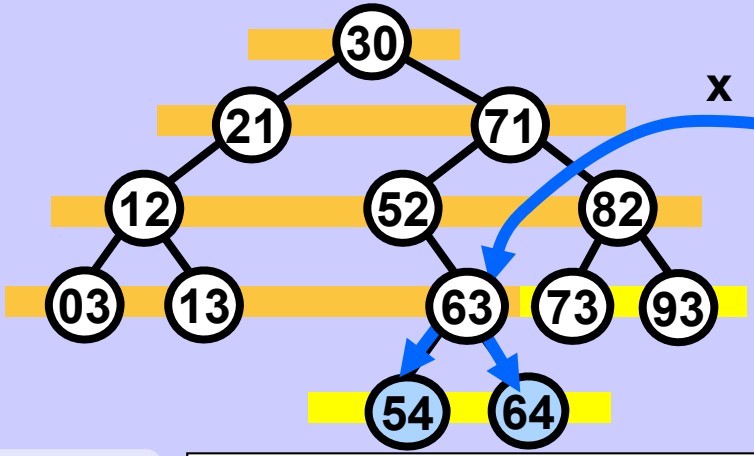
2. Enqeue(x.left), Enqueue(x.right). *)



Output: 30 21 71 12 52 82 03 13

*) if exists

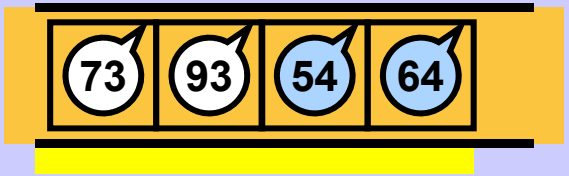
Breadth-first search (BFS) of a tree



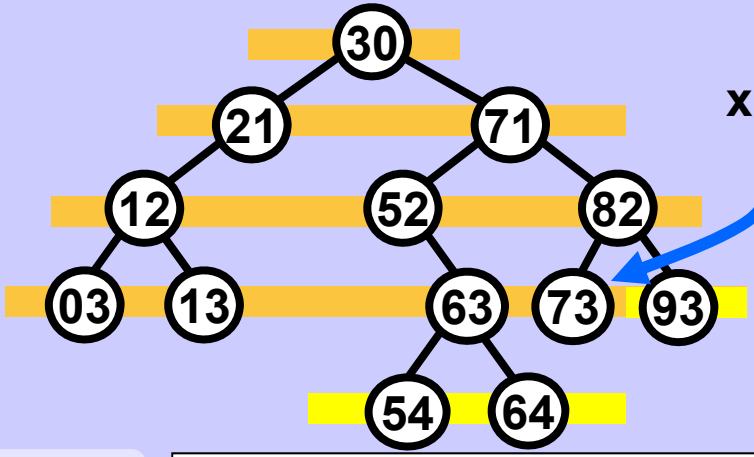
x

1. x = Dequeue(), print (x.key).

2. Enqeue(x.left), Enqueue(x.right). *)



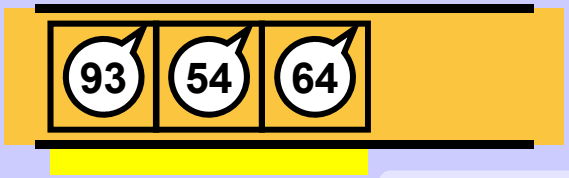
Output: 30 21 71 12 52 82 03 13 63



x

1. x = Dequeue(), print (x.key).

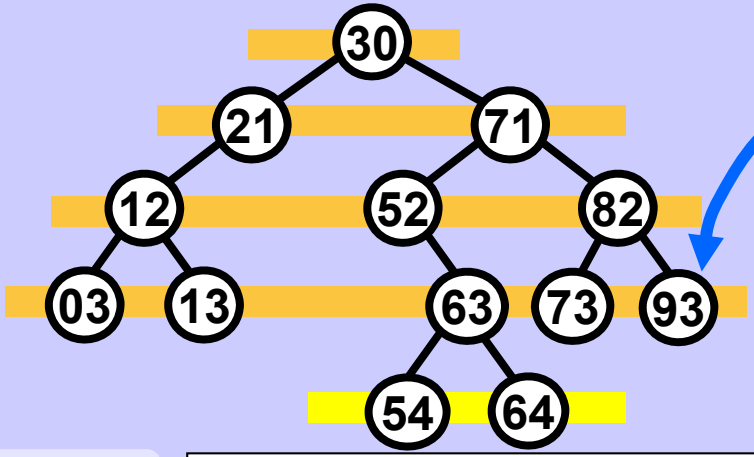
2. Enqeue(x.left), Enqueue(x.right). *)



Output: 30 21 71 12 52 82 03 13 63 73

*) if exists

Breadth-first search (BFS) of a tree

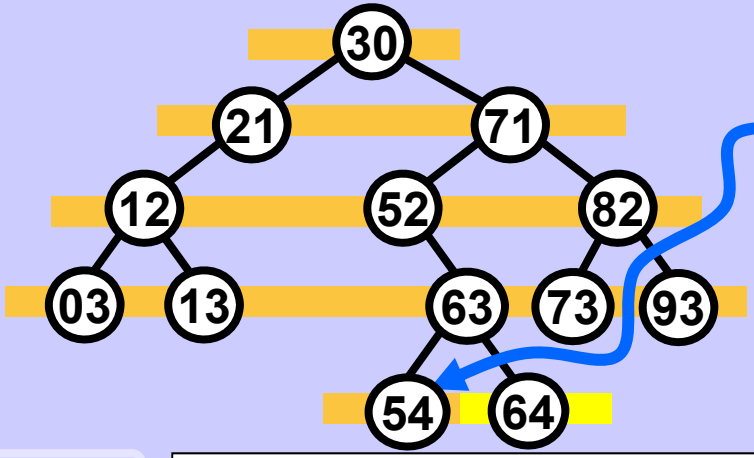


1. x = Dequeue(), print (x.key).

x

2. Enqeue(x.left), Enqueue(x.right). *)

Output: 30 21 71 12 52 82 03 13 63 73 93



1. x = Dequeue(), print (x.key).

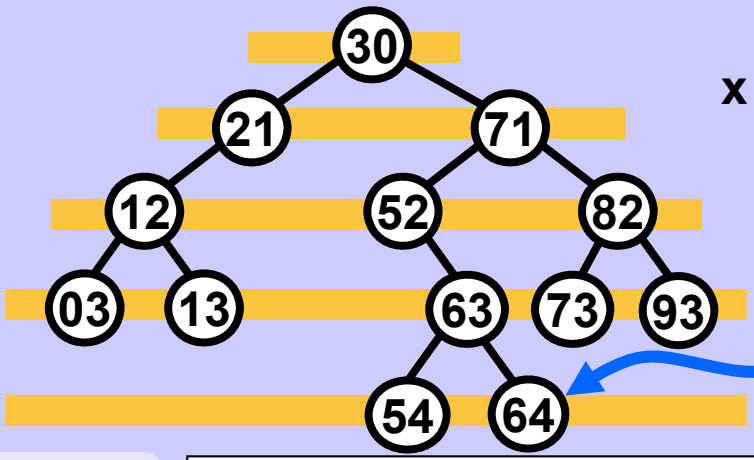
x

2. Enqeue(x.left), Enqueue(x.right). *)

Output: 30 21 71 12 52 82 03 13 63 73 93 54

*) if exists

Breadth-first search (BFS) of a tree



x

1. x = Dequeue(), print (x.key).

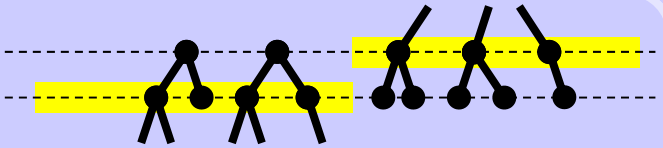
2. Enqeue(x.left), Enqueue(x.right). *)

Output: 30 21 71 12 52 82 03 13 63 73 93 54 64

*) if exists.

The queue is empty,
BFS is complete.

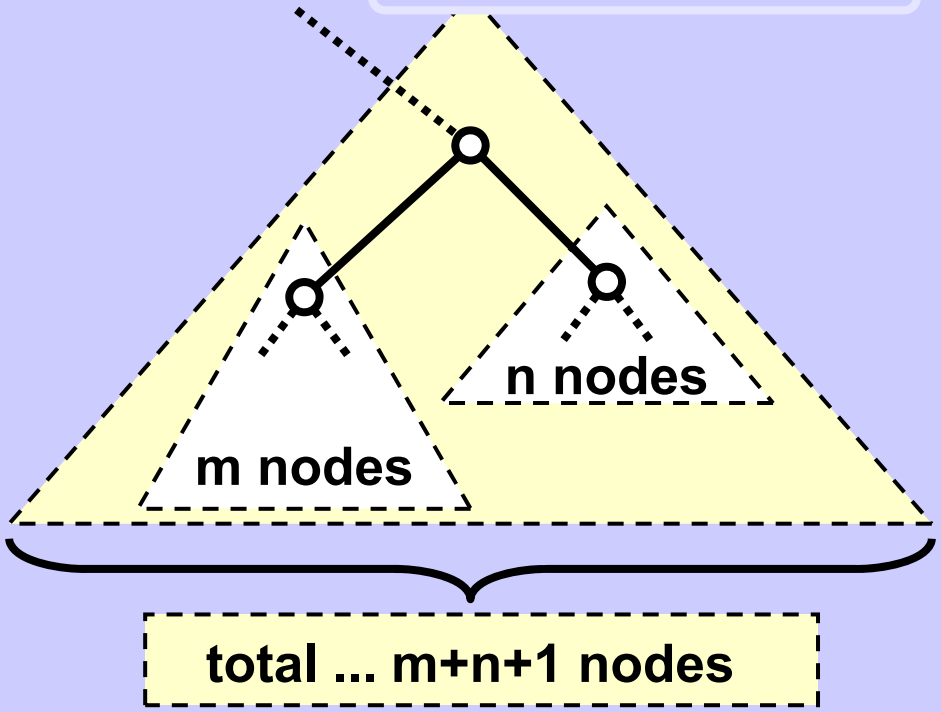
An unempty queue always contains exactly
-- some (or all) nodes of one level and
-- all children of those nodes of this level which have already left the queue.



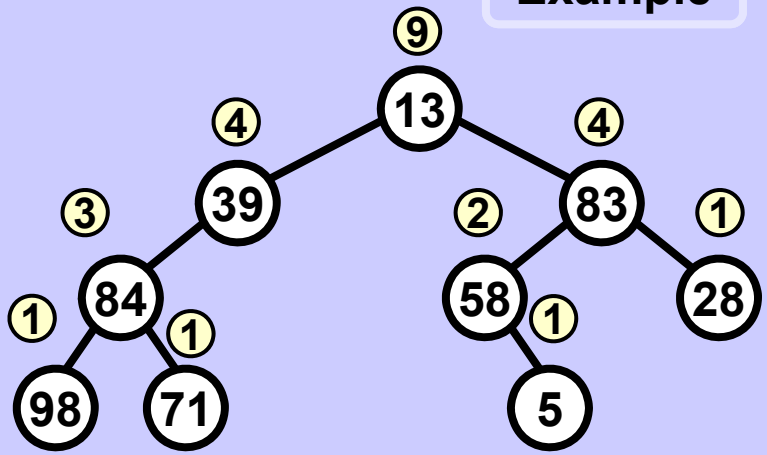
Sometimes the queue contains just nodes of one level. See above:

Tree size (= number of nodes) recursively

A tree or a subtree



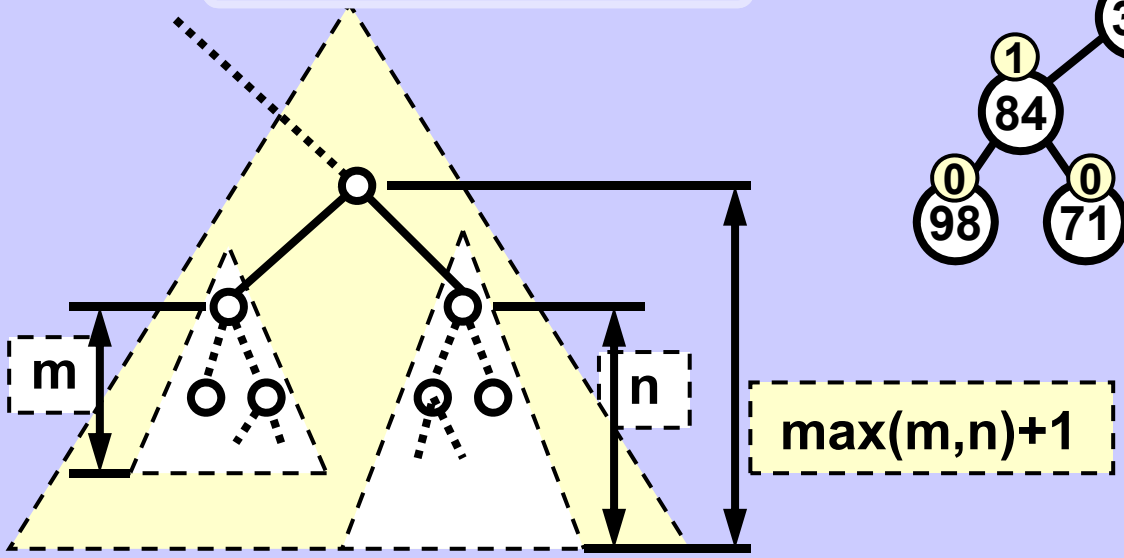
Example



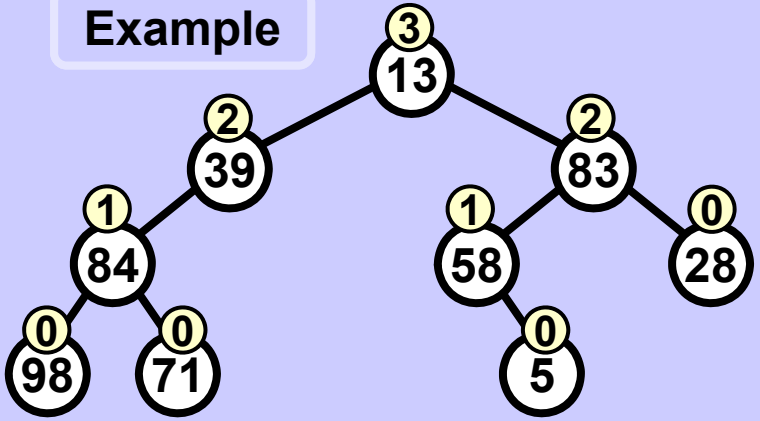

```
def count( self, node ):
  if node == None: return 0
  return 1 + self.count(node.left) + self.count(node.right)
```

Tree depth (= max depth of a node) recursively

A tree or a subtee



Example




```
def depth( self, node):
  if node == None: return -1
  return 1 + max(self.depth(node.left),self.depth(node.right))
```