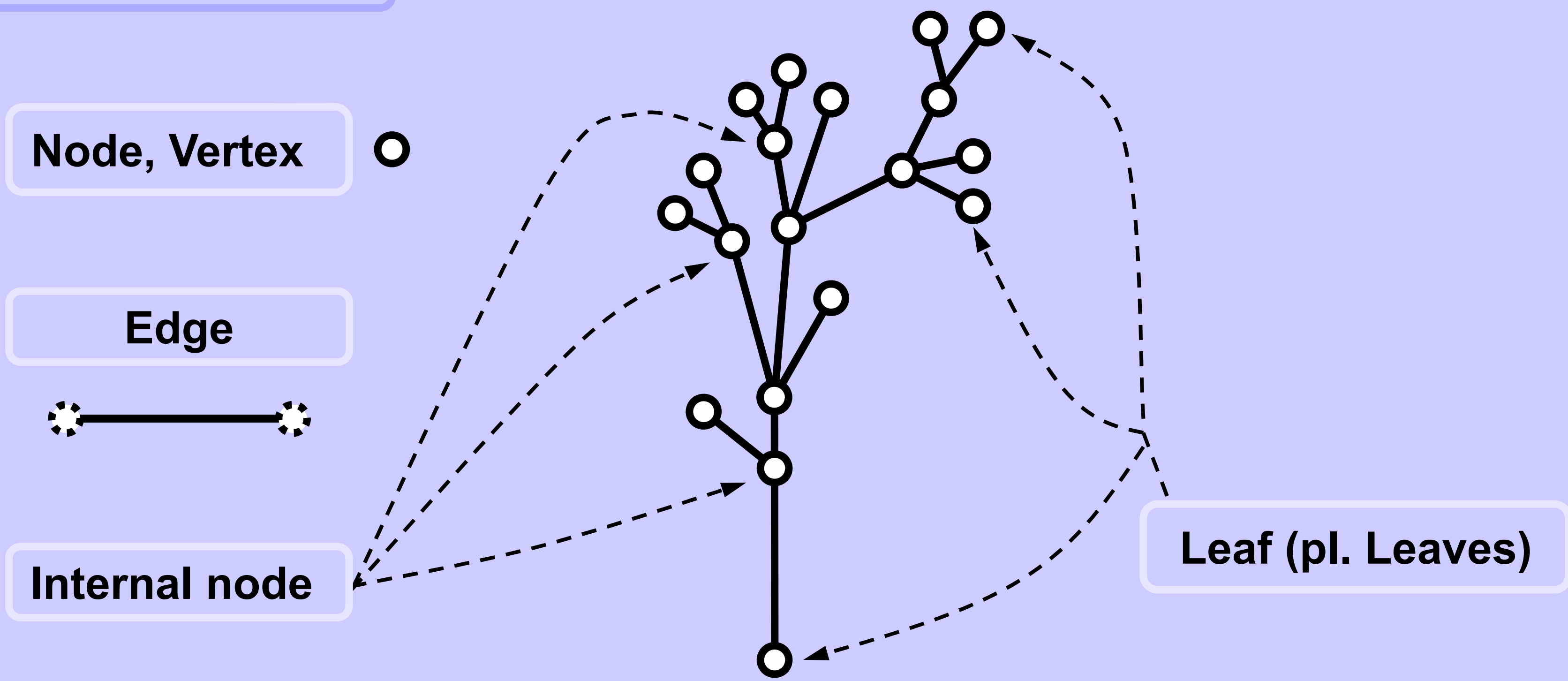
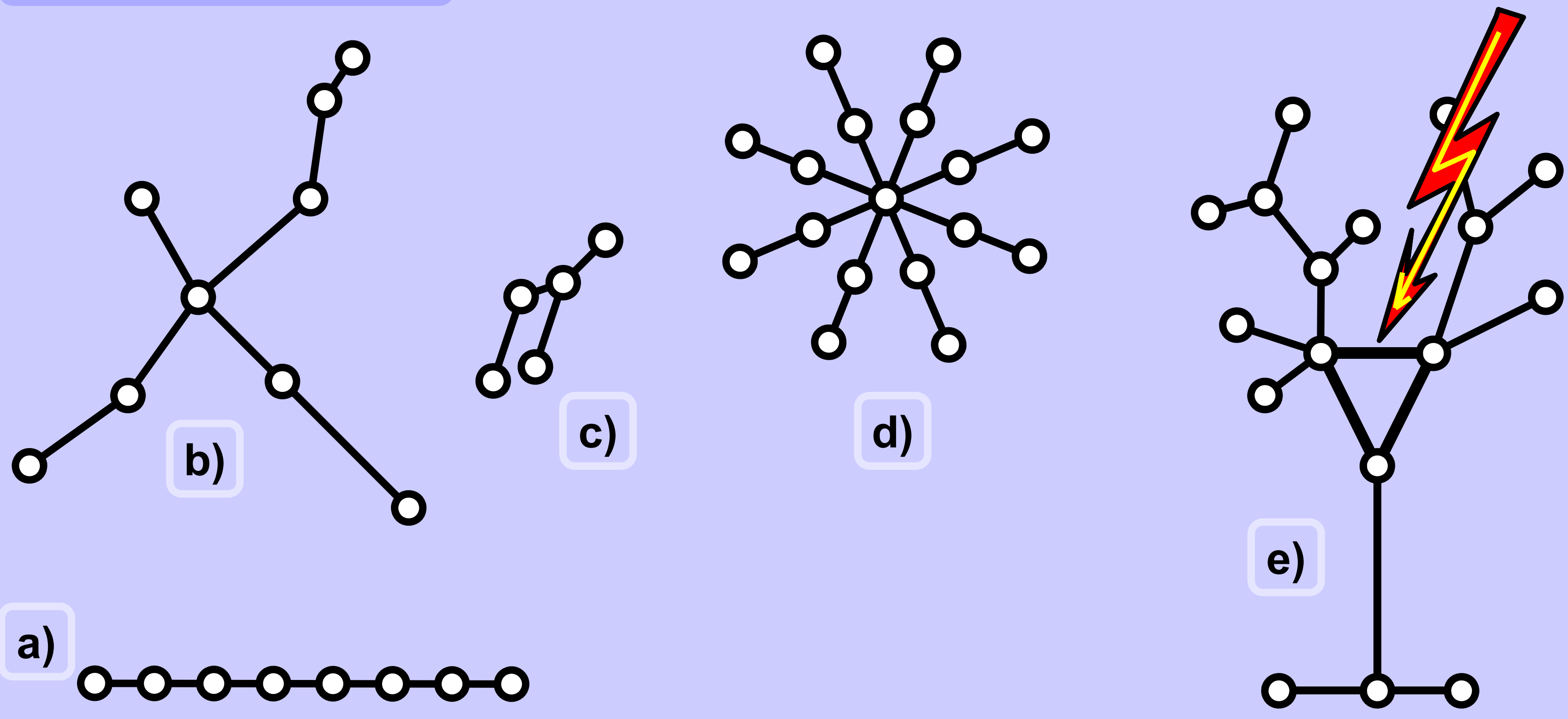
TREES, BINARY TREES
REALTION BETWEEN TREES AND RECURSION
USING STACK TO IMPLEMENT RECURSION
BACKTRACKING

Tree



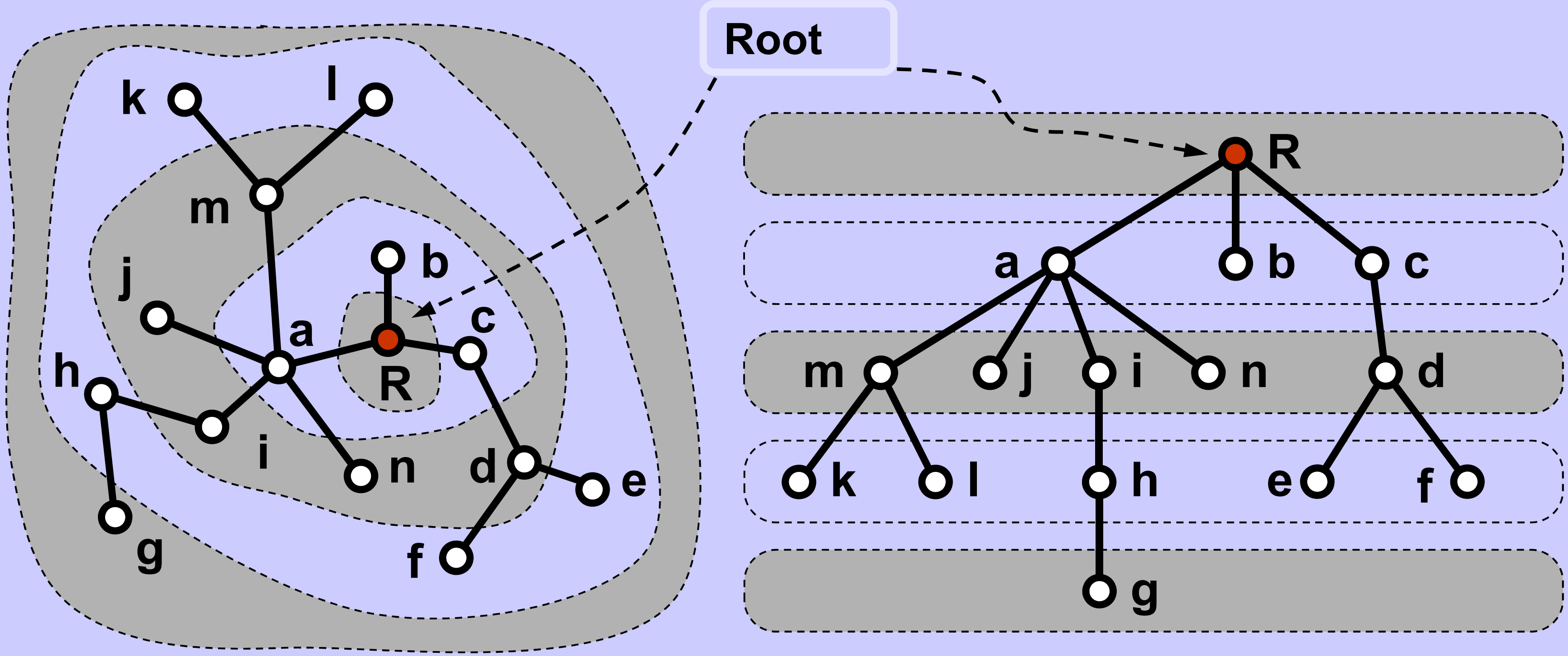
Tree examples



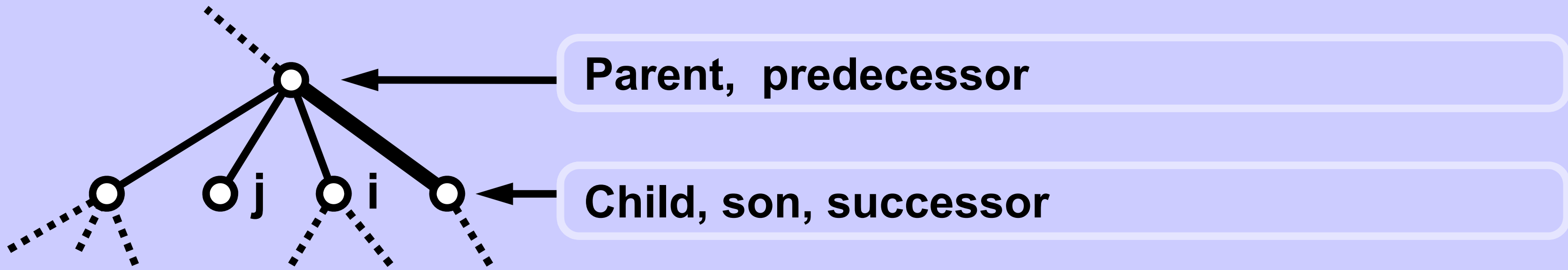
Tree properties

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.

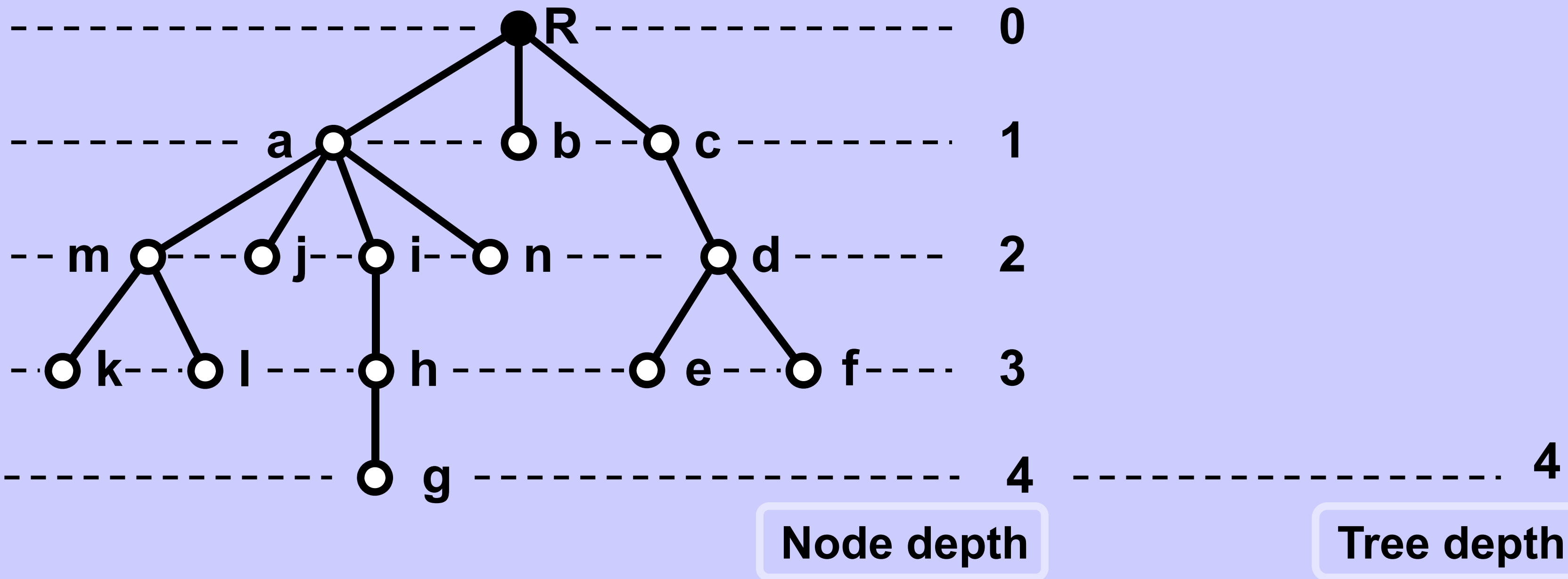
Rooted tree



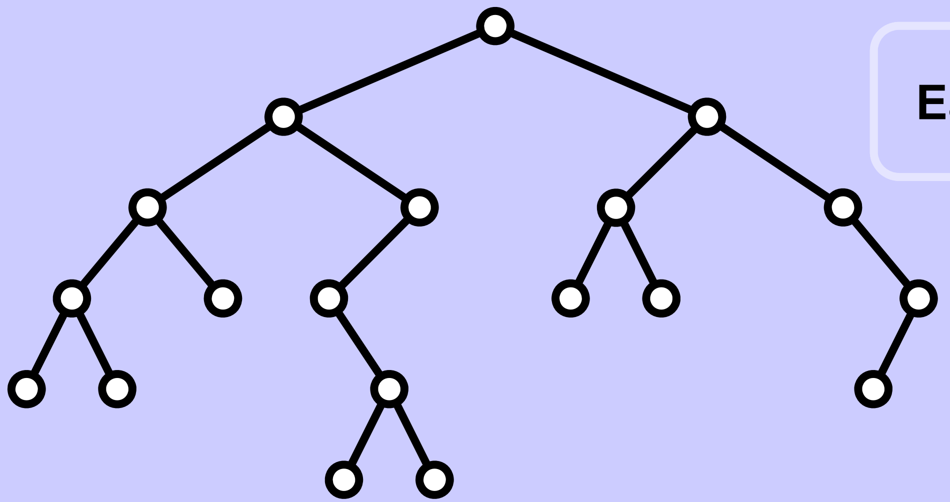
Terminology



Tree depth

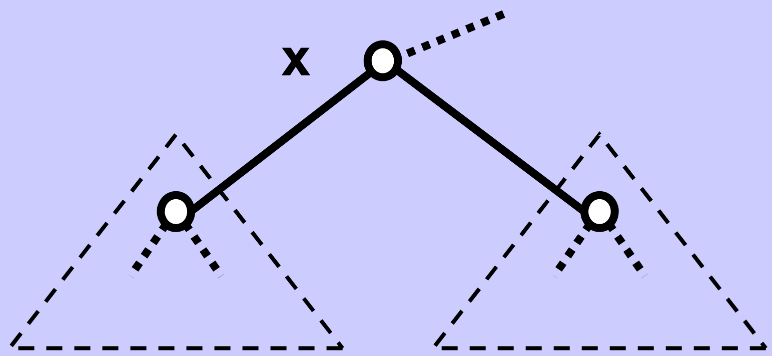


Binary (rooted!!) tree



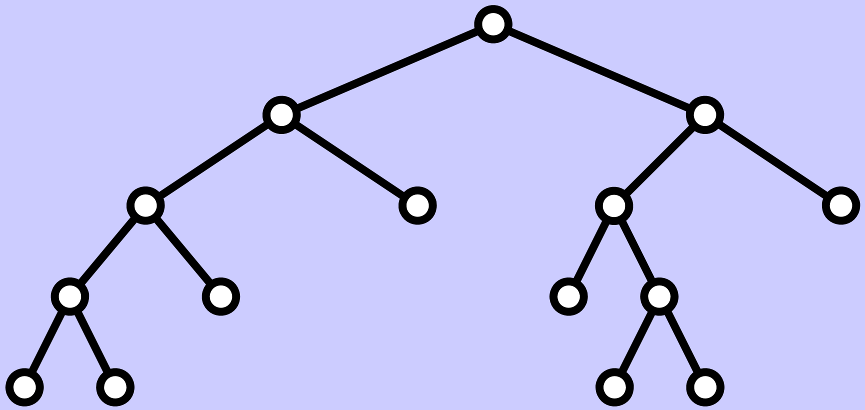
Each node has 0 or 1 or 2 children.

Left and right subtree



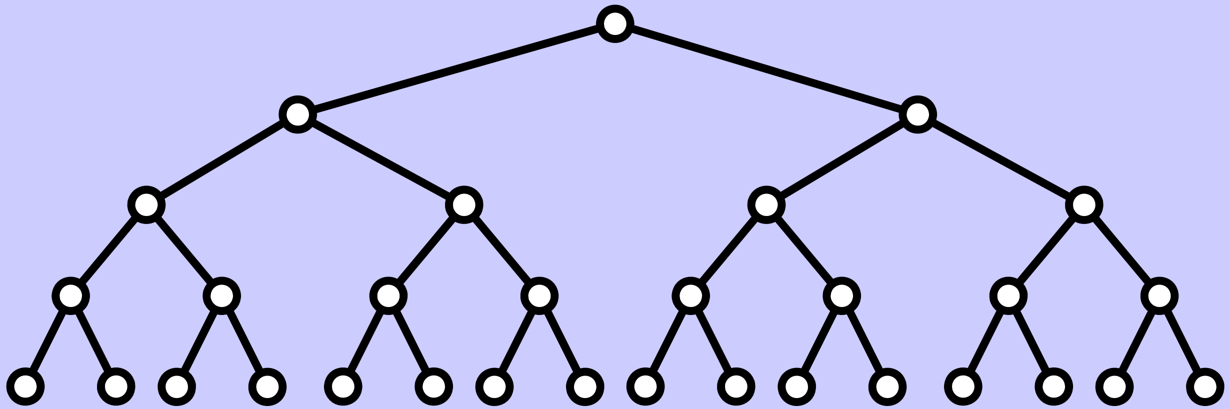
Subtree of node x .... left .... right

Regular binary tree



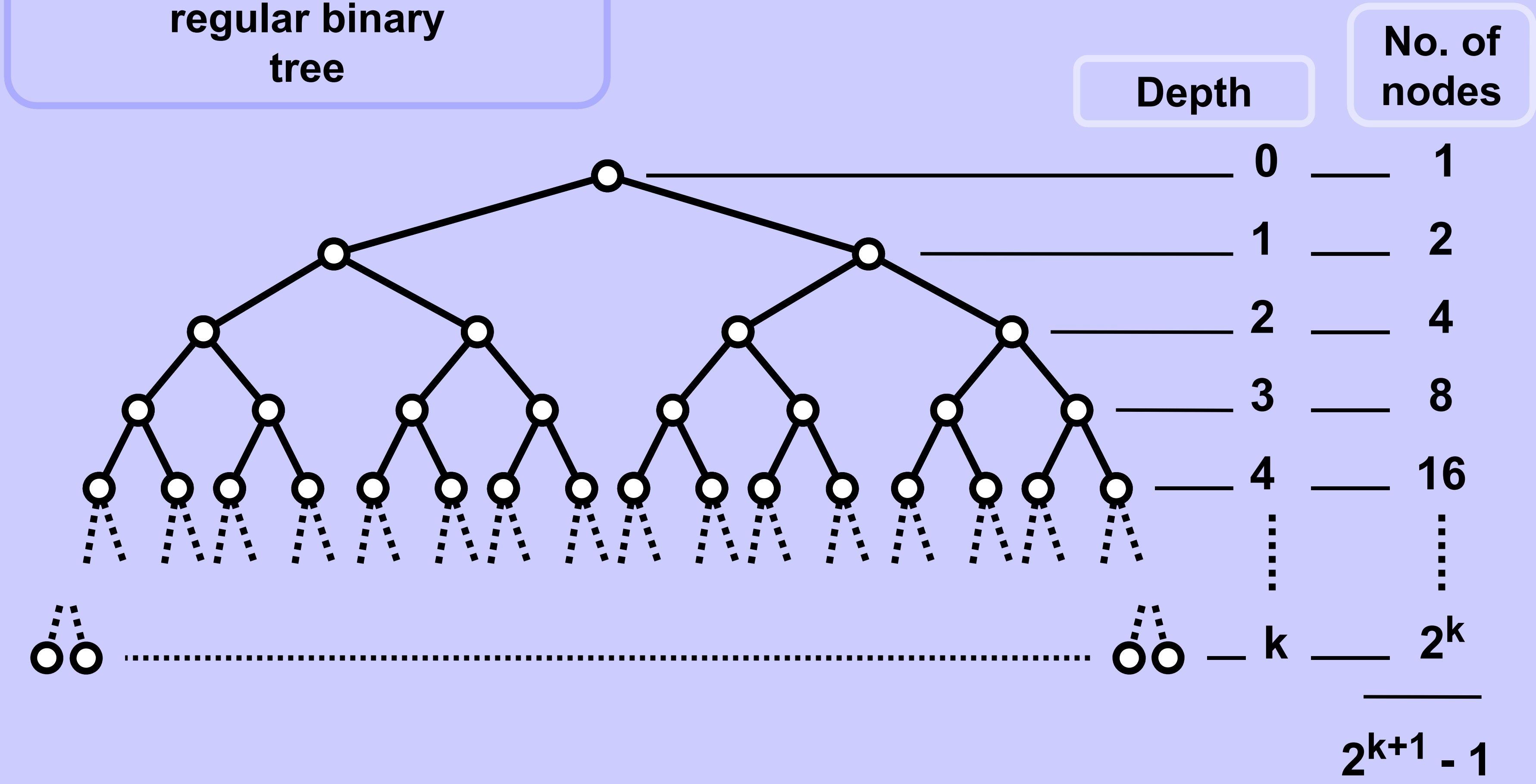
Each node has 0 or 2 children.
Not 1 child

Balanced tree



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

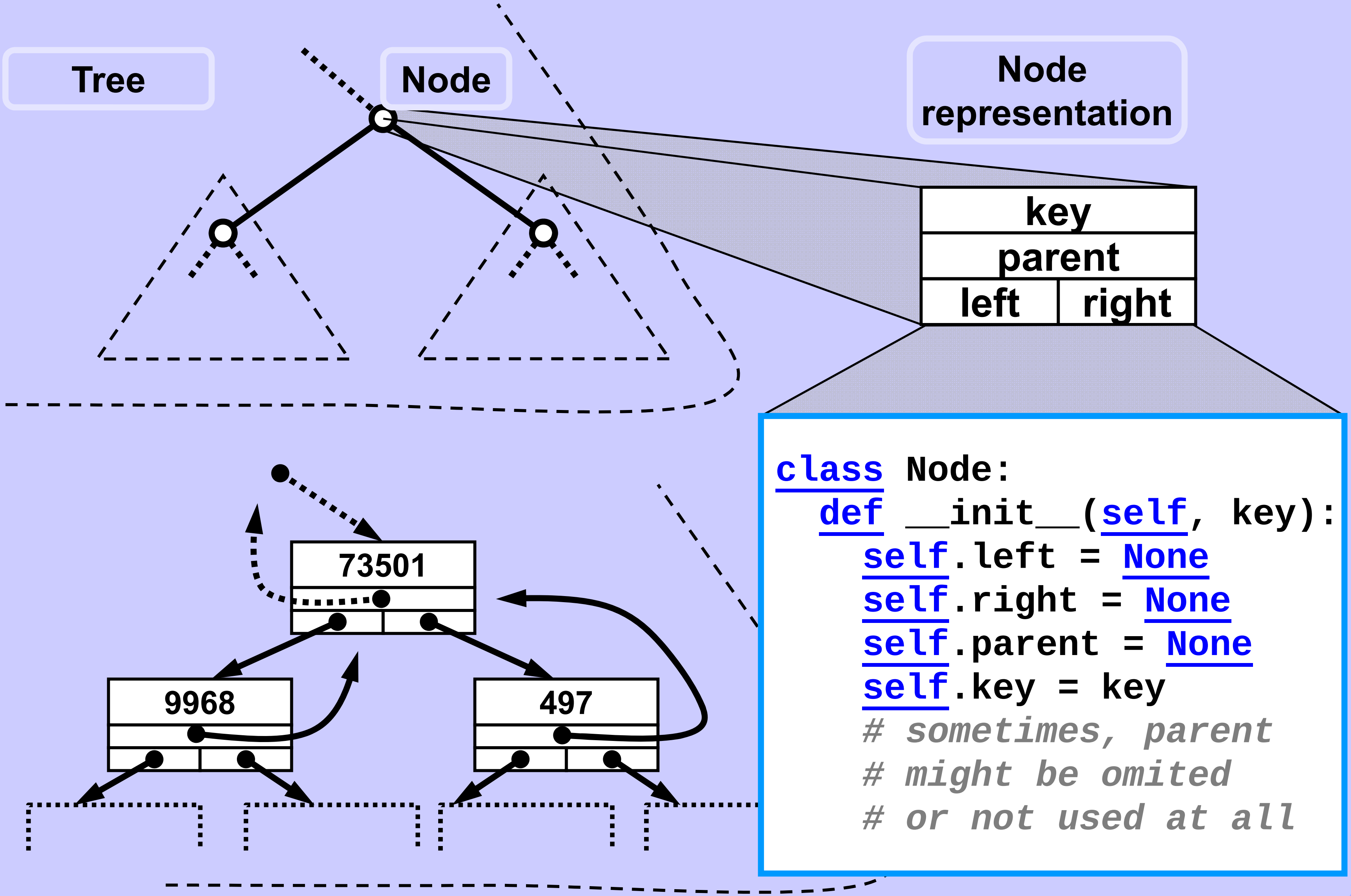
Depth of a balanced regular binary tree



$(2^{depth+1} – 1)$ ~ no. of nodes

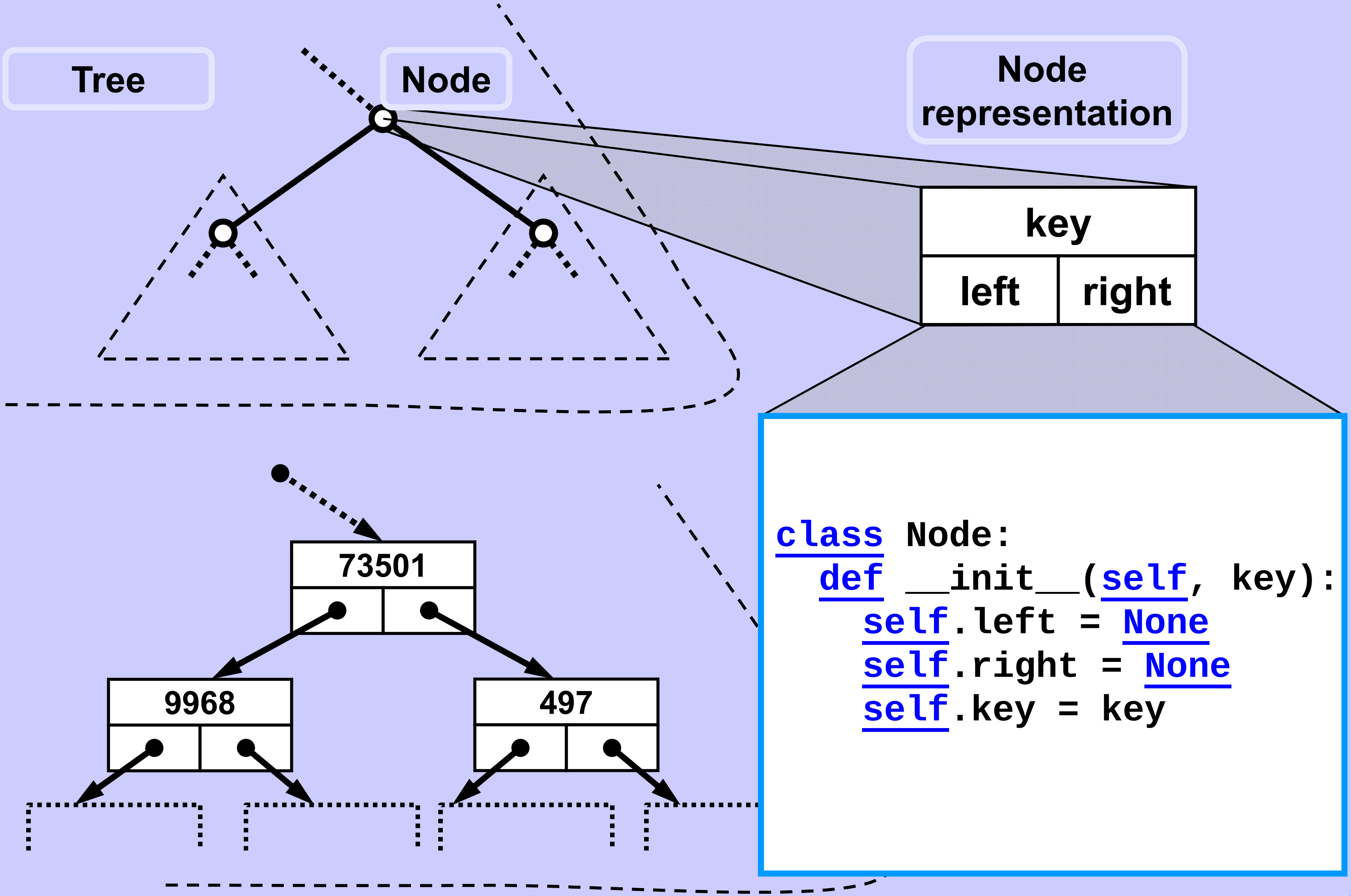
Depth ~ $\log_2(|nodes|+1) – 1$ ~ $\log_2(|nodes|)$

Binary tree implementation -- Python




```python
class Node:
  def __init__(self, key):
    self.left = None
    self.right = None
    self.parent = None
    self.key = key
    # sometimes, parent
    # might be omited
    # or not used at all
```

Binary tree implementation -- Python




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

Build a random binary tree -- Python

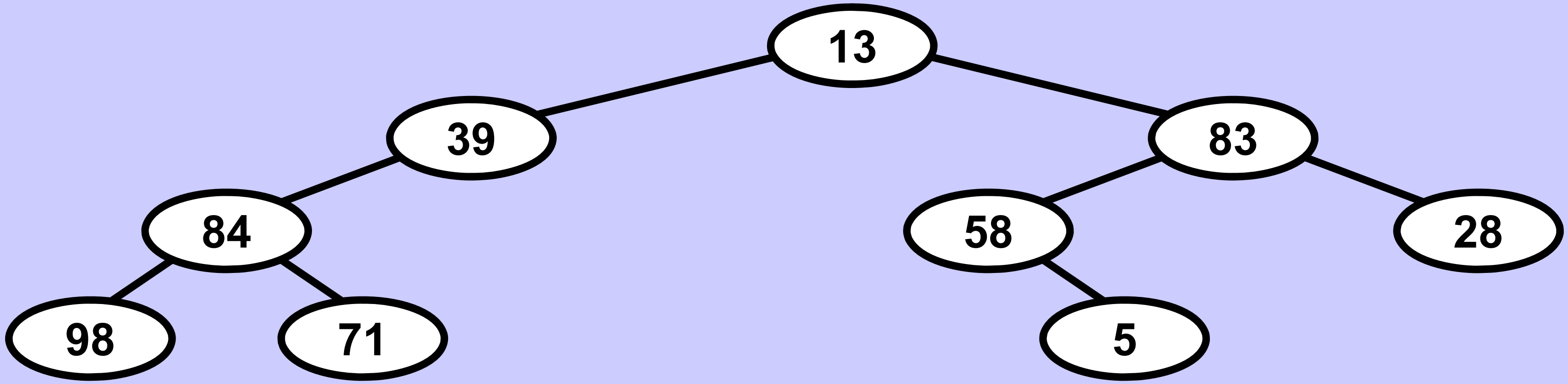
```
@staticmethod # Binary tree calls this method
def rndTree( depth ):
    if depth <= 0 or random.randrange(10) > 7 :
        return None
    newnode = Node( 10+random.randrange(90) )
    newnode.left = Node.rndTree( depth-1 )
    newnode.right = Node.rndTree( depth-1 )
    return newnode
```

Example of function call

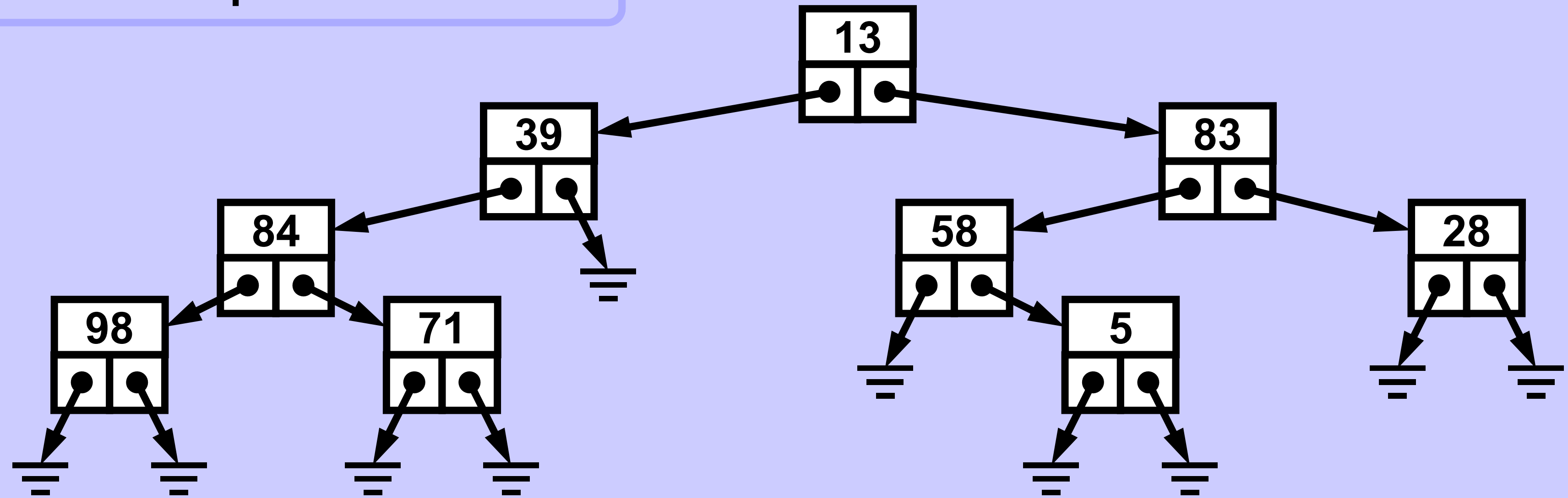
```
tree1 = BinaryTree()
tree1.randomTree(4)
```

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

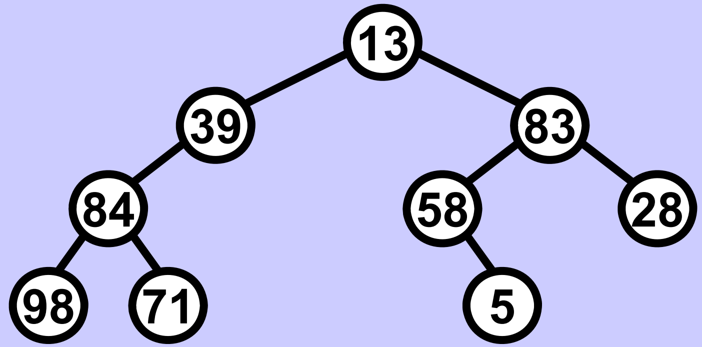


Tree representation



Inorder traversal of a binary tree

Tree



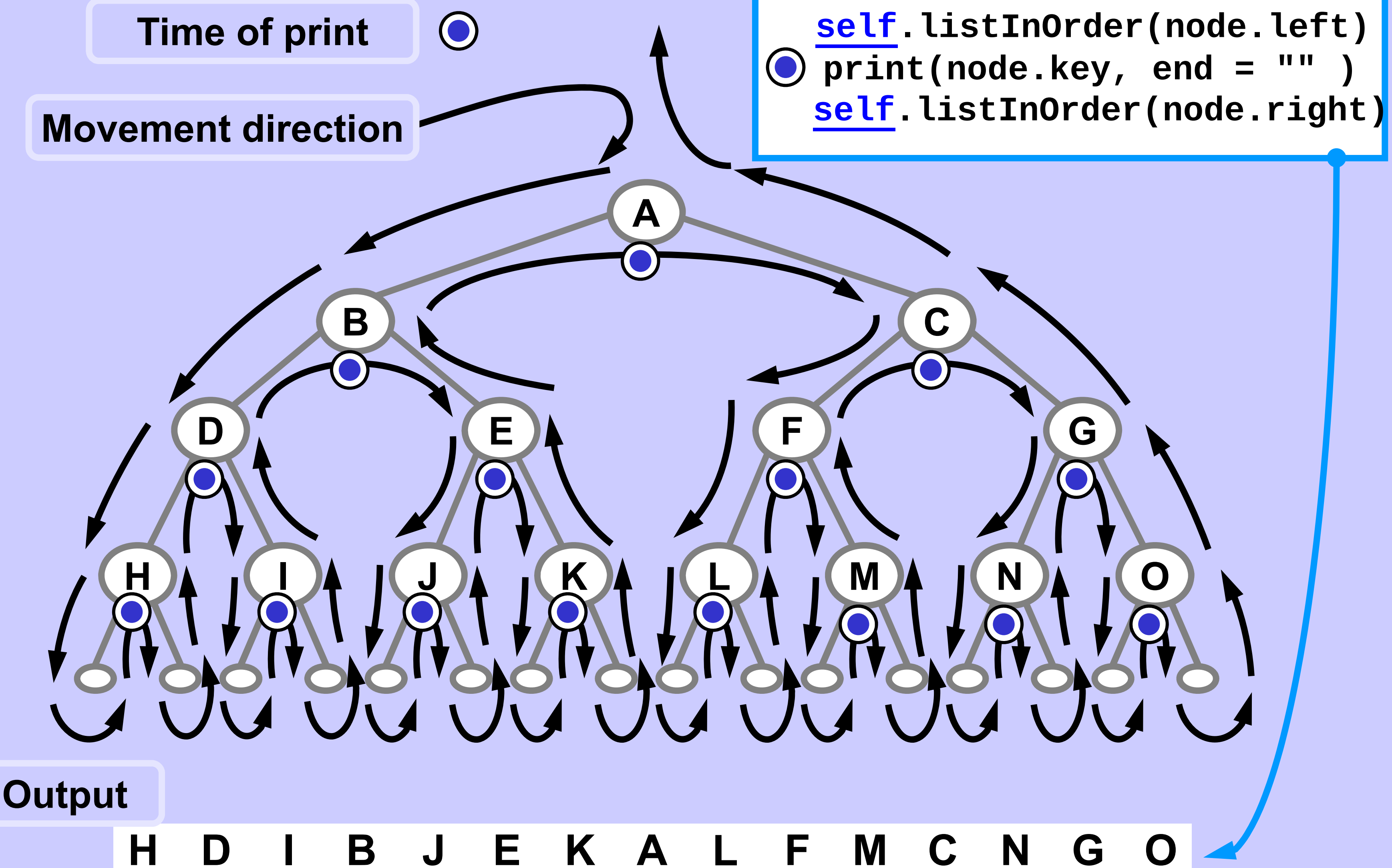
INORDER traversal

```
def listInOrder( self, node ):
   if node == None: return
   self.listInOrder( node.left )
   print( node.key, end = " " )
   self.listInOrder( node.right )
```

Output

98 84 71 39 13 58 5 83 28

Movement in the tree during inorder traversal



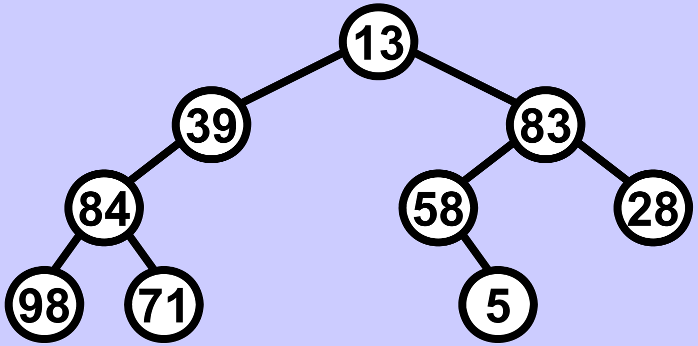

```
self.listInOrder(node.left)
print(node.key, end = "" )
self.listInOrder(node.right)
```

Output

H D I B J E K A L F M C N G O

Preorder traversal of a binary tree

Tree



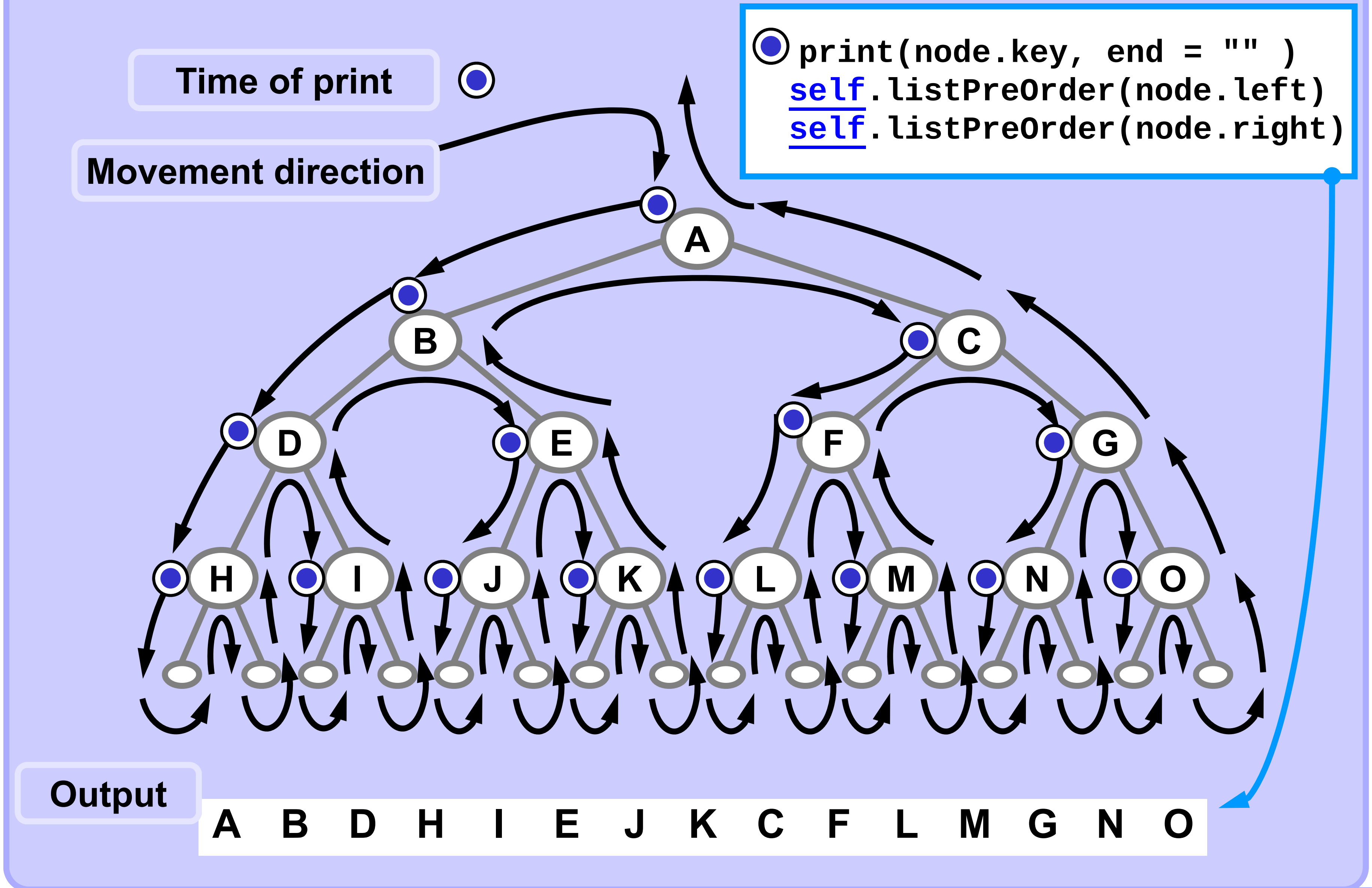
PREORDER traversal

```
def listPreOrder( self, node ):
    if node == None: return
    print( node.key, end = " " )
    self.listPreOrder( node.left )
    self.listPreOrder( node.right )
```

Output

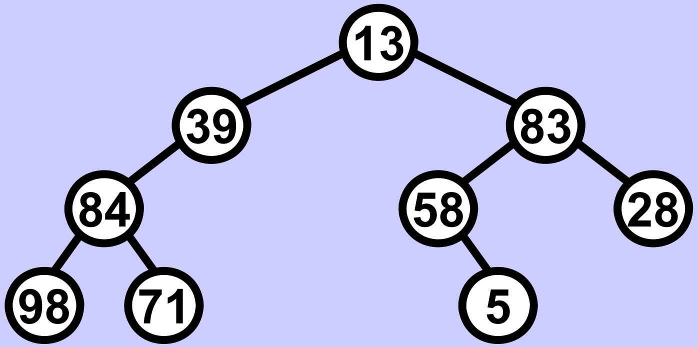
13 39 84 98 71 83 58 5 28

Movement in the tree during preorder traversal



Postorder traversal of a binary tree

Tree



POSTORDER traversal

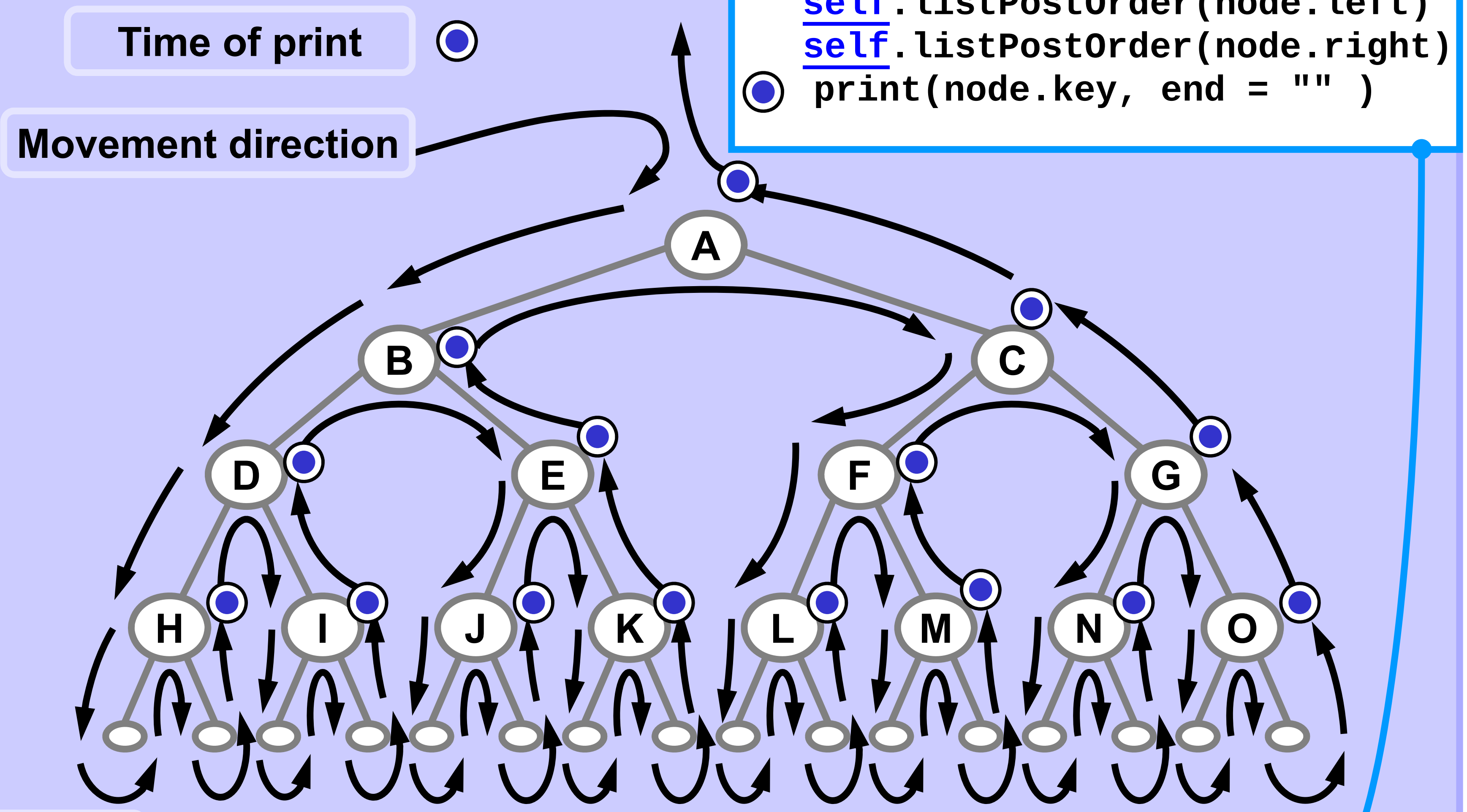
```
def listPostOrder( self, node ):
  if node == None: return
  self.listPostOrder( node.left )
  self.listPostOrder( node.right )
  print( node.key, end = " " )
```

Output

98 71 84 39 5 58 28 83 13

Movement in the tree during postorder traversal

```
self.listPostOrder(node.left)
self.listPostOrder(node.right)
print(node.key, end = "" )
```

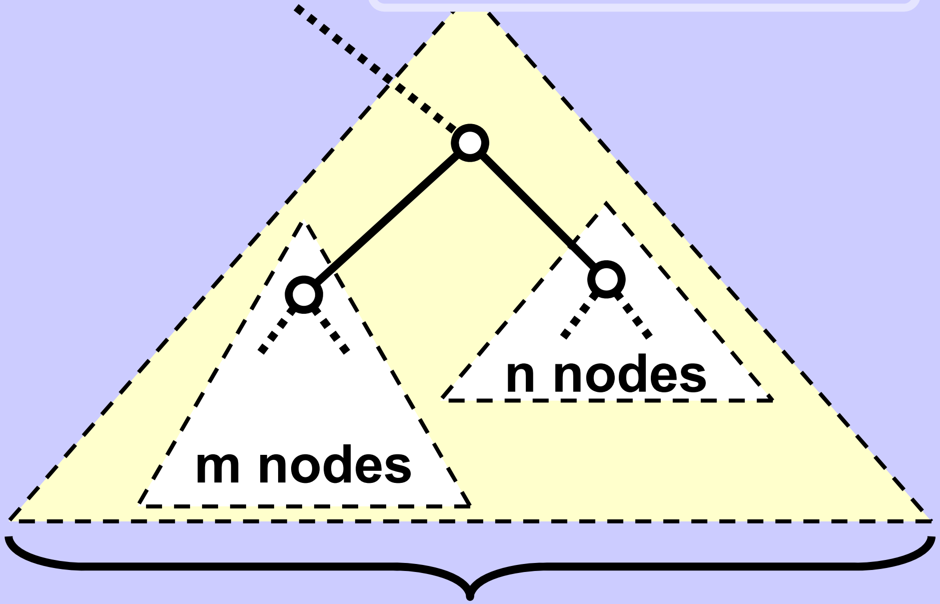


Output

H I D J K E B L M F N O G C A

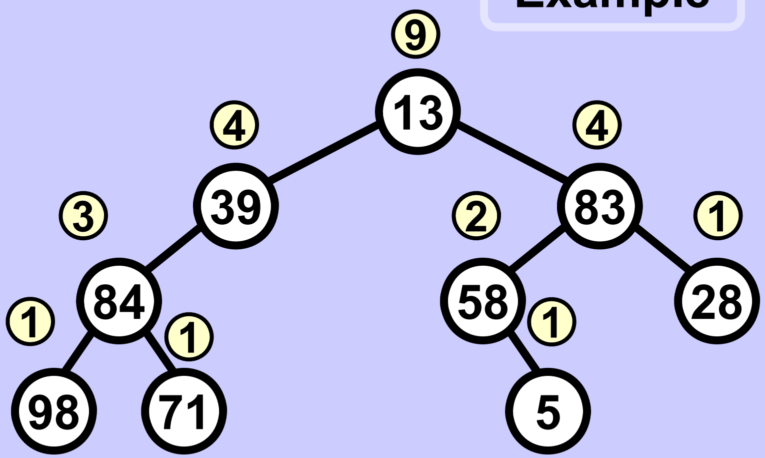
Tree size (= number of nodes) recursively

A tree or a subtree



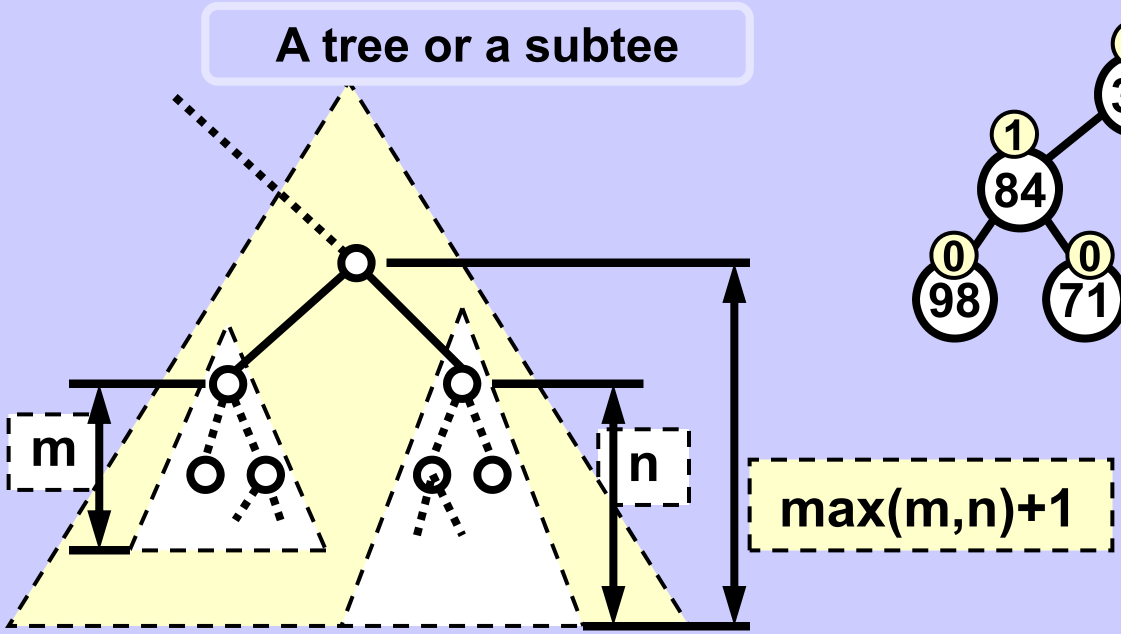
total ... m+n+1 nodes

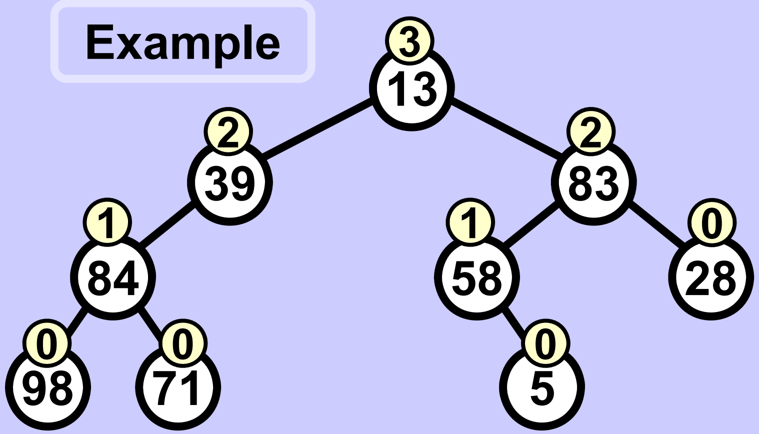
Example




```
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



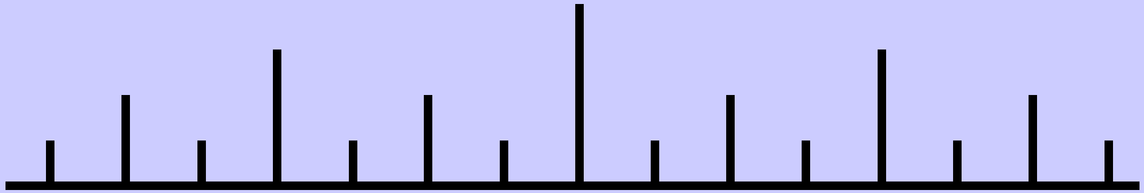



```python
def depth( self, node):
   if node == None: return -1
   return 1 + max(self.depth(node.left),self.depth(node.right))
```

Simple recursive example

Binary ruler

Ruler notches



Notch lengths

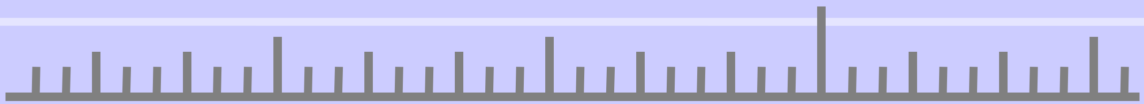
1	2	1	3	1	2	1	4	1	2	1	3	1	2	1

Print the lengths of all notches

```
def ruler( val ):
    if val < 1: return
    ruler( val-1 )
    print( val, end = '' )
    ruler( val-1 )

Call: ruler(4)
```

Exercise: Ternary ruler:



Simple recursive example

Binary ruler vs. Inorder traversal

Ruler

```
def ruler( val ):
   if val < 1: return
   ruler( val-1 )
   print( val, end='' )
   ruler( val-1 )
```

Inorder

```
def listInOrder( self, node ):
   if node == None: return
   self.listInOrder( node.left )
   print( node.key, end = " " )
   self.listInOrder( node.right )
```

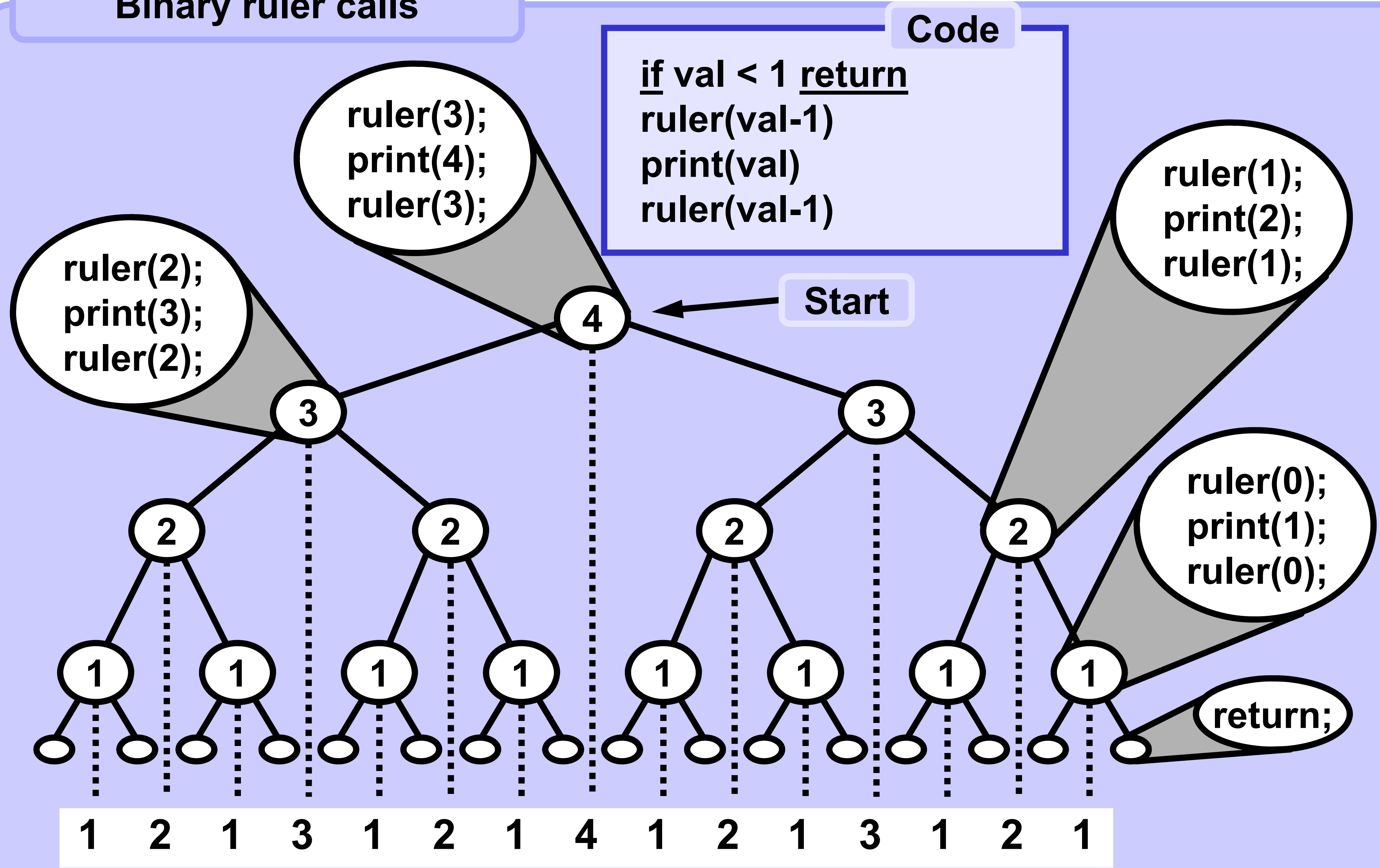
Structurally identical!

Ruler output

1	2	1	3	1	2	1	4	1	2	1	3	1	2	1

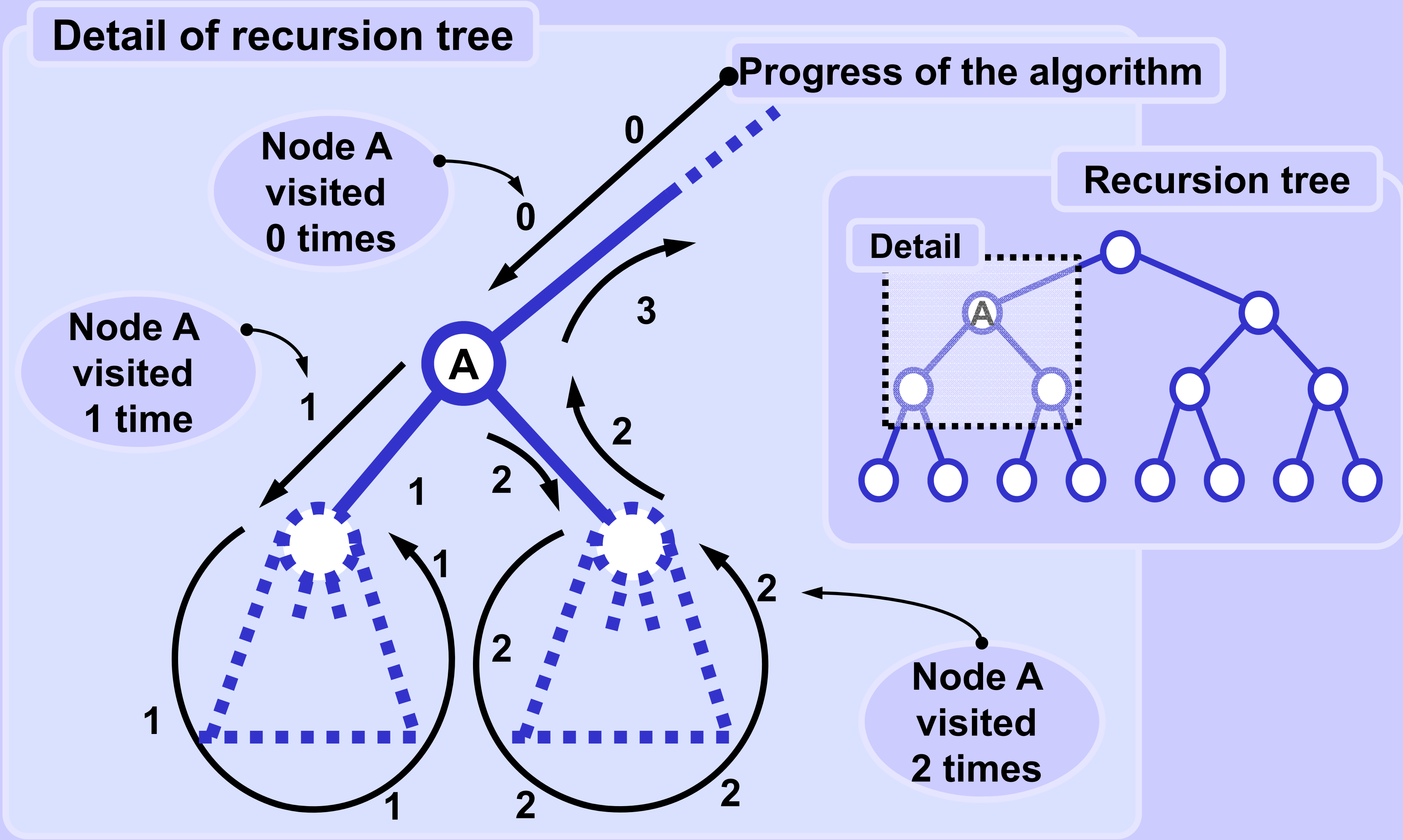
Simple recursive example

Binary ruler calls



Stack implements recursion

Binary ruler



Stack implements recursion

Standard strategy

Using the stack:

Whenever possible process only the data which are on the stack.

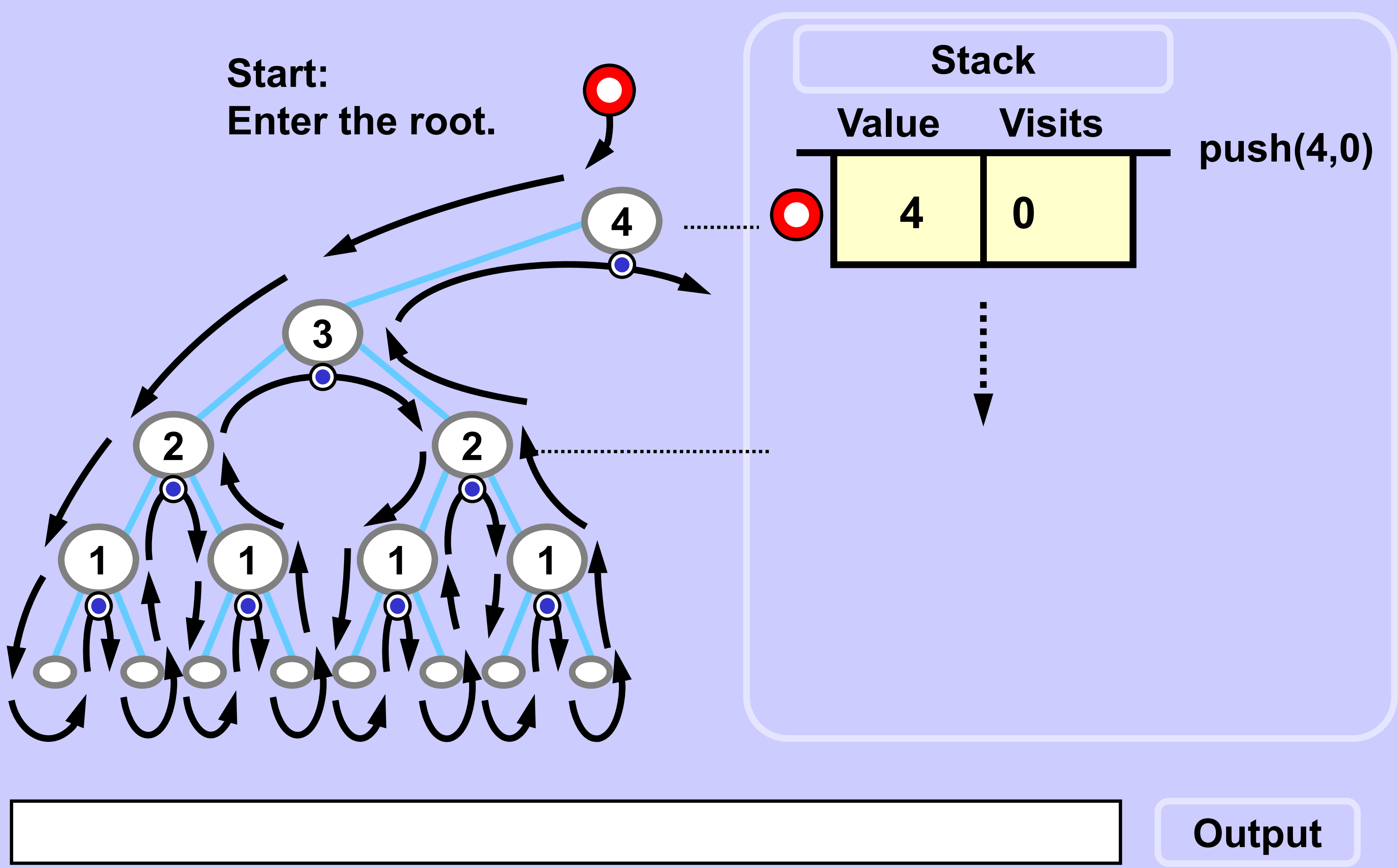
Standard approach

Push the first node (first element to be processed) to the stack.
Push each next node (next element to be processed) to the stack too.
Process only the node (element) at the top of the stack.
Pop the processed element from the stack.
Stop when the stack is empty.

Stack implements recursion

Each frame in the following sequence shows the situation right BEFORE processing a node.

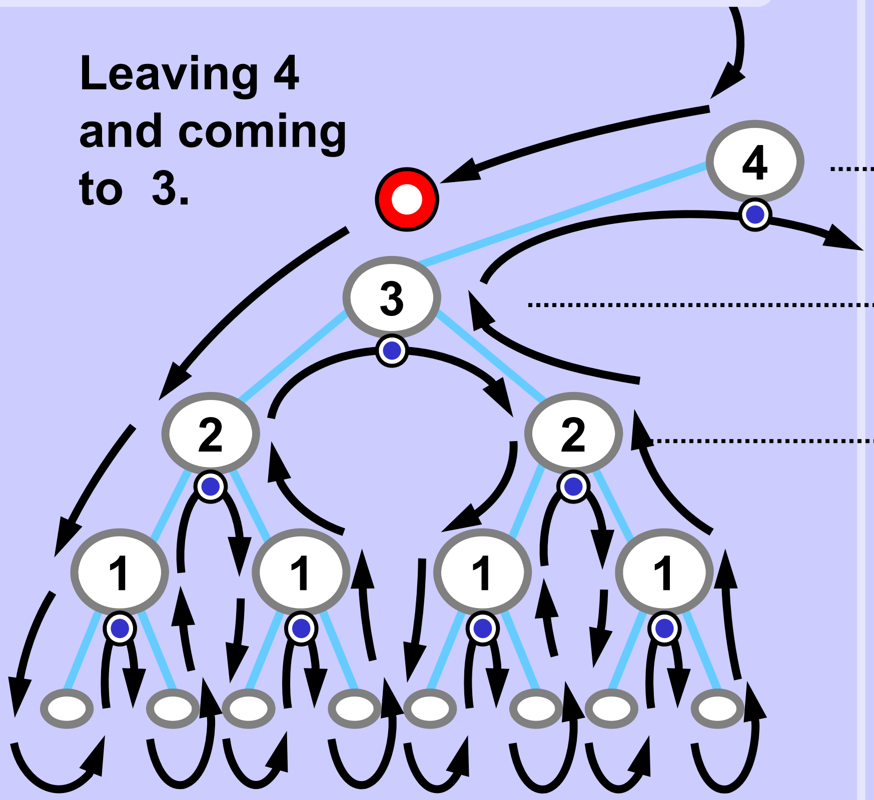
Current position



Stack implements recursion

Recursion tree traversal

Leaving 4
and coming
to 3.



Stack

Value	Visits
4	1
3	0

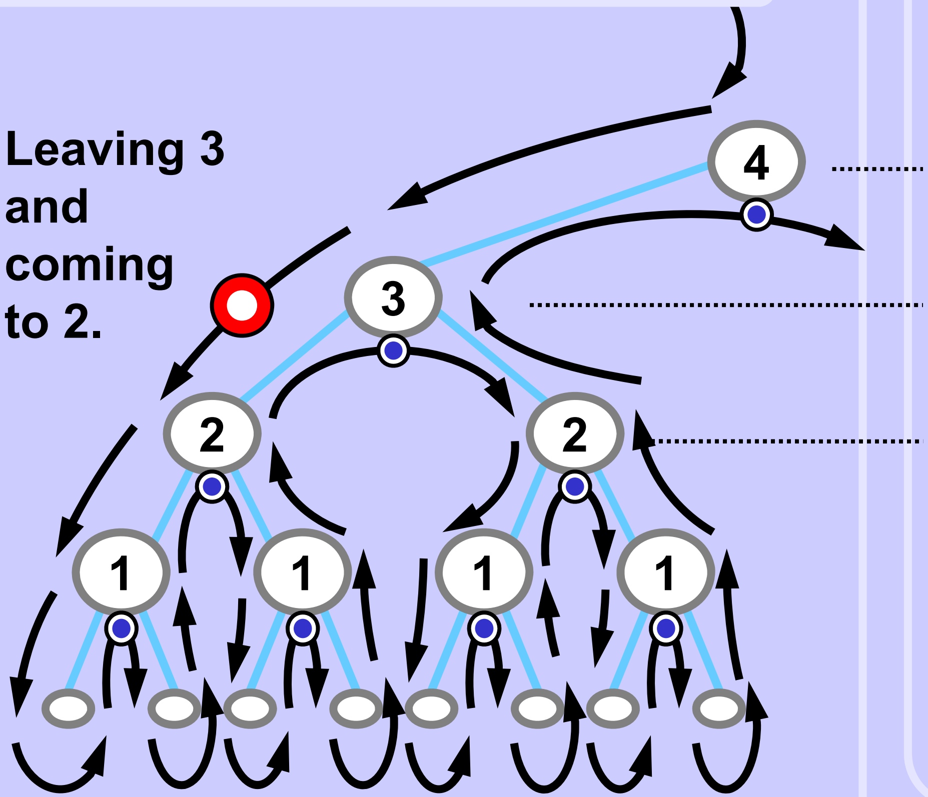
push(3,0)

Output

Stack implements recursion

Recursion tree traversal

Leaving 3
and
coming
to 2.



Stack

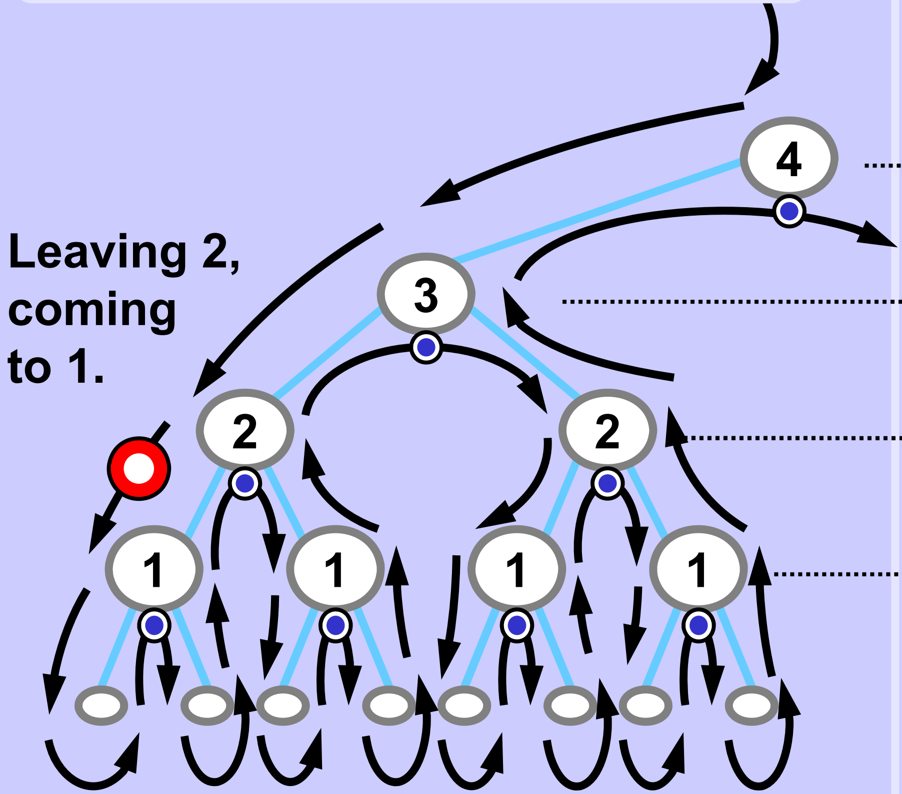
Value	Visits
4	1
3	1
2	0

push(2,0)

Output

Stack implements recursion

Recursion tree traversal



Stack

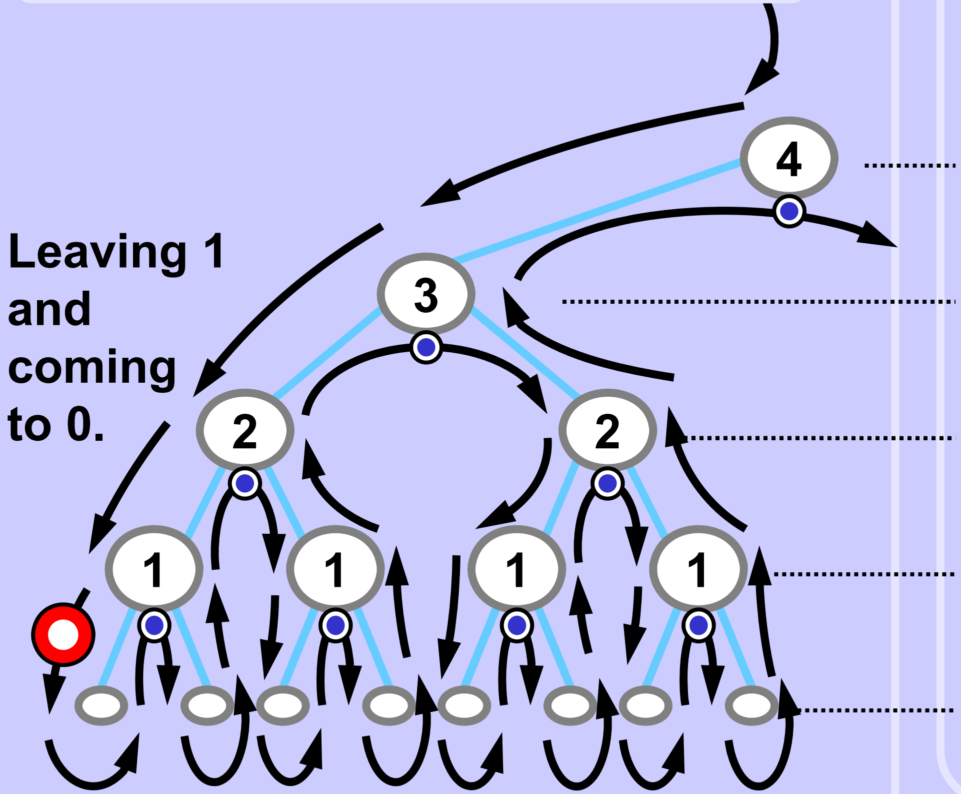
Value	Visits
4	1
3	1
2	1
1	0

push(1,0)

Output

Stack implements recursion

Recursion tree traversal



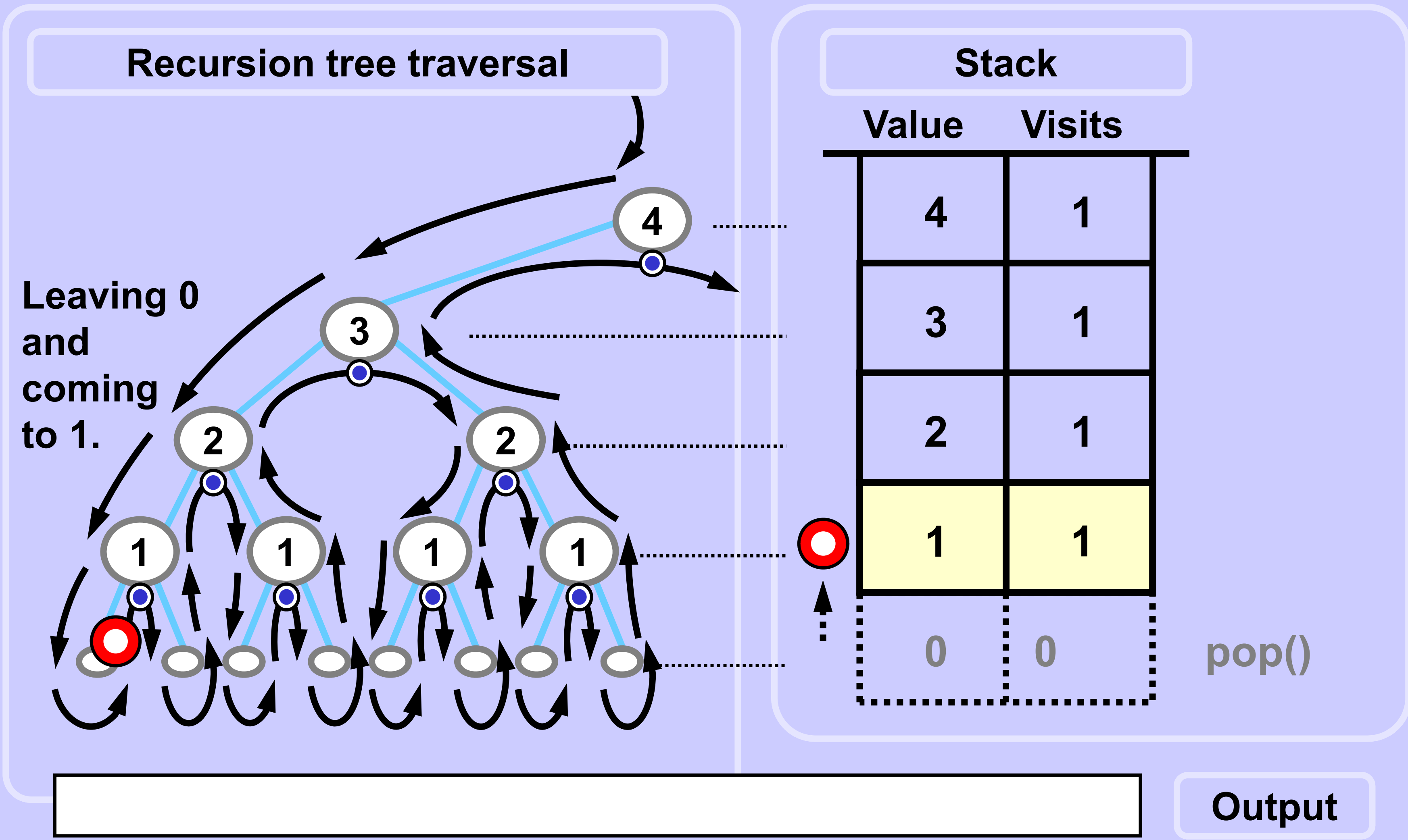
Stack

Value	Visits
4	1
3	1
2	1
1	1
0	0

push(0,0)

Output

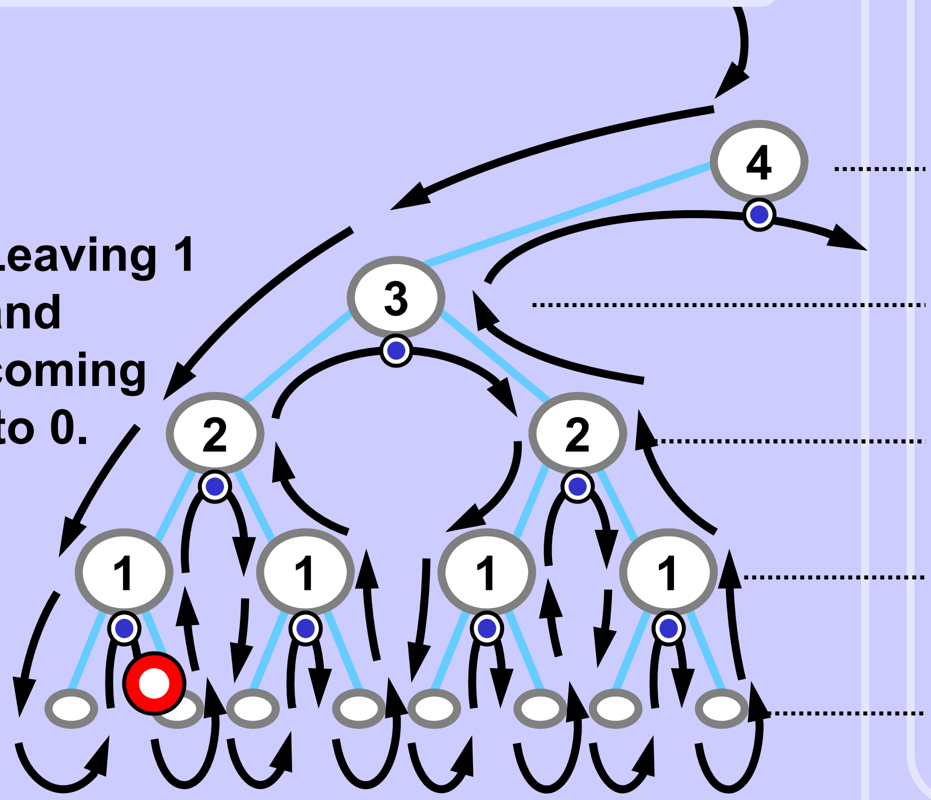
Stack implements recursion



Stack implements recursion

Recursion tree traversal

Leaving 1 and coming to 0.



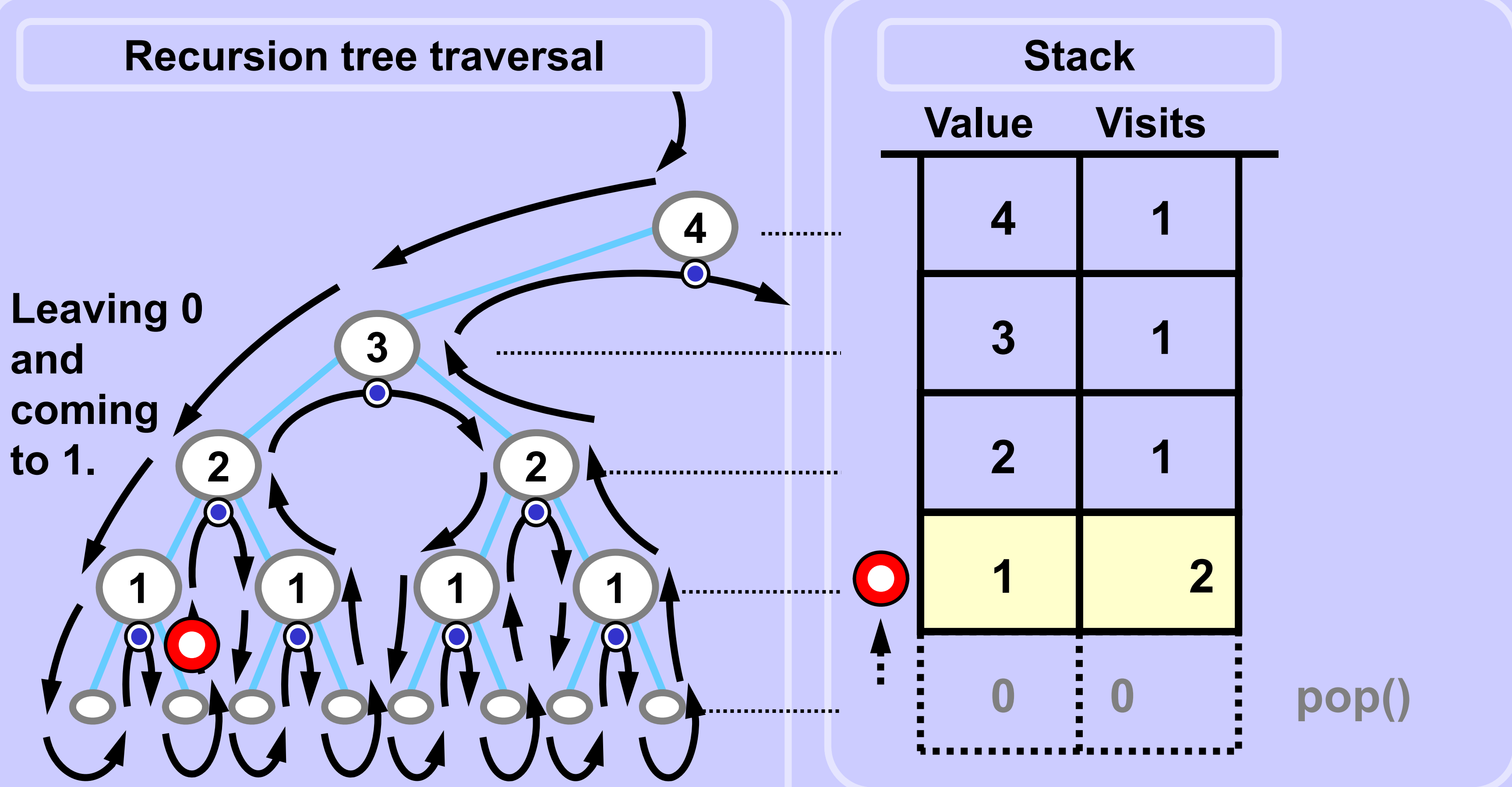
Stack

Value	Visits
4	1
3	1
2	1
1	2
0	0

push(0,0)

Output: 1

Stack implements recursion



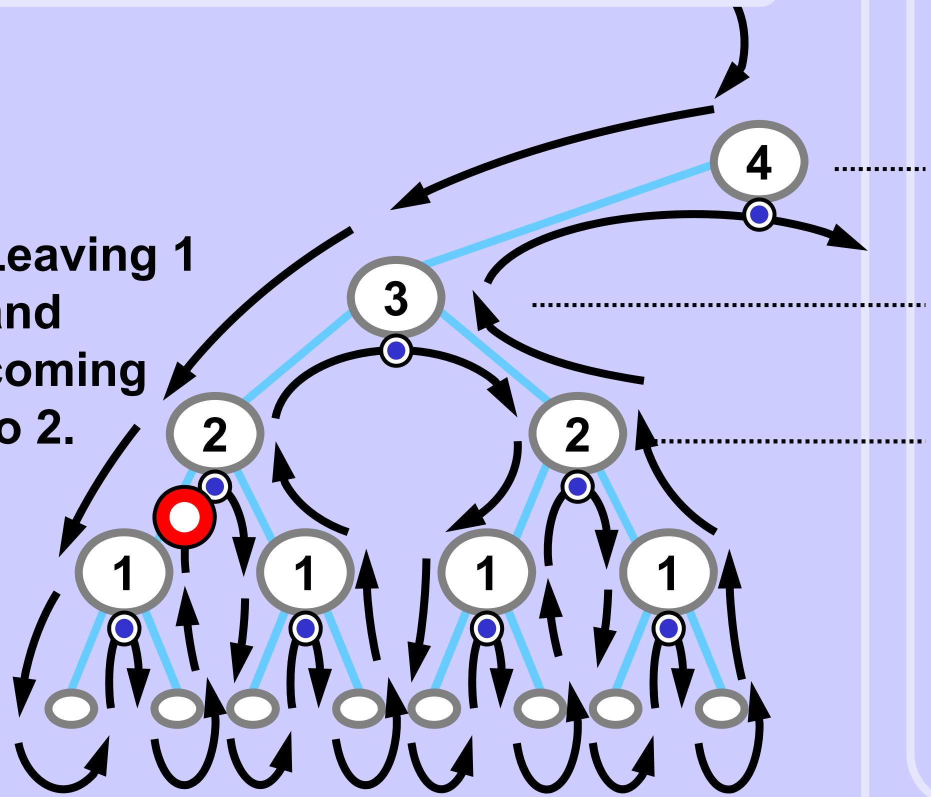
1

Output

Stack implements recursion

Recursion tree traversal

Leaving 1 and coming to 2.



Stack

Value	Visits
4	1
3	1
2	1
1	2

pop()

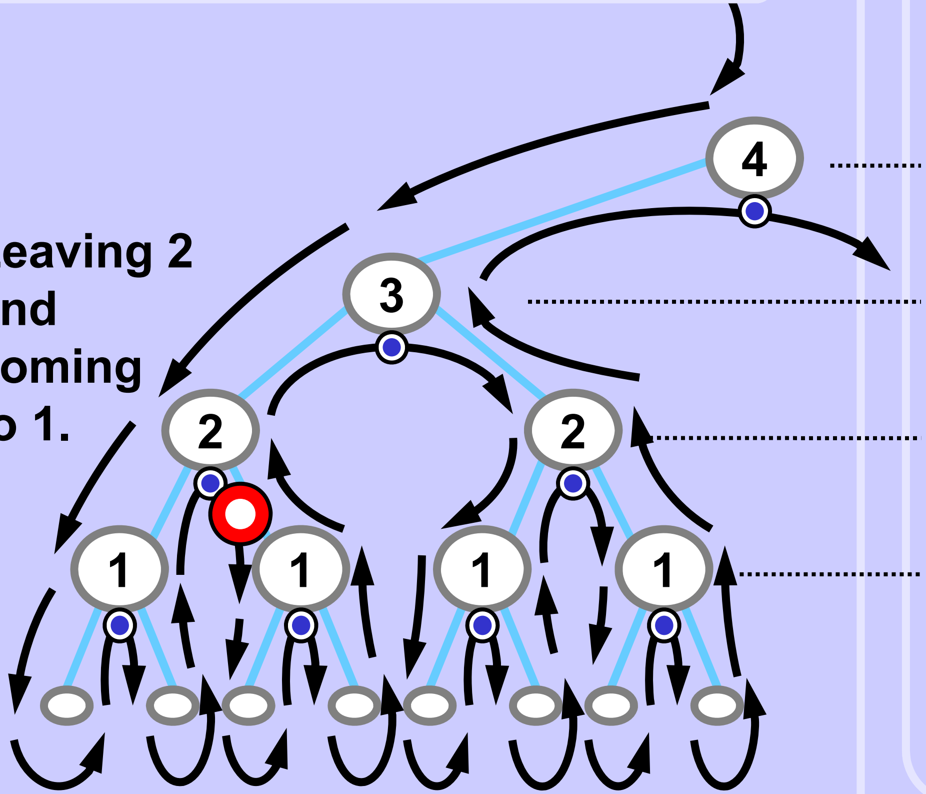
1

Output

Stack implements recursion

Recursion tree traversal

Leaving 2
and
coming
to 1.



Stack

Value	Visits
4	1
3	1
2	2
1	0

push(1,0)

Output

1 2

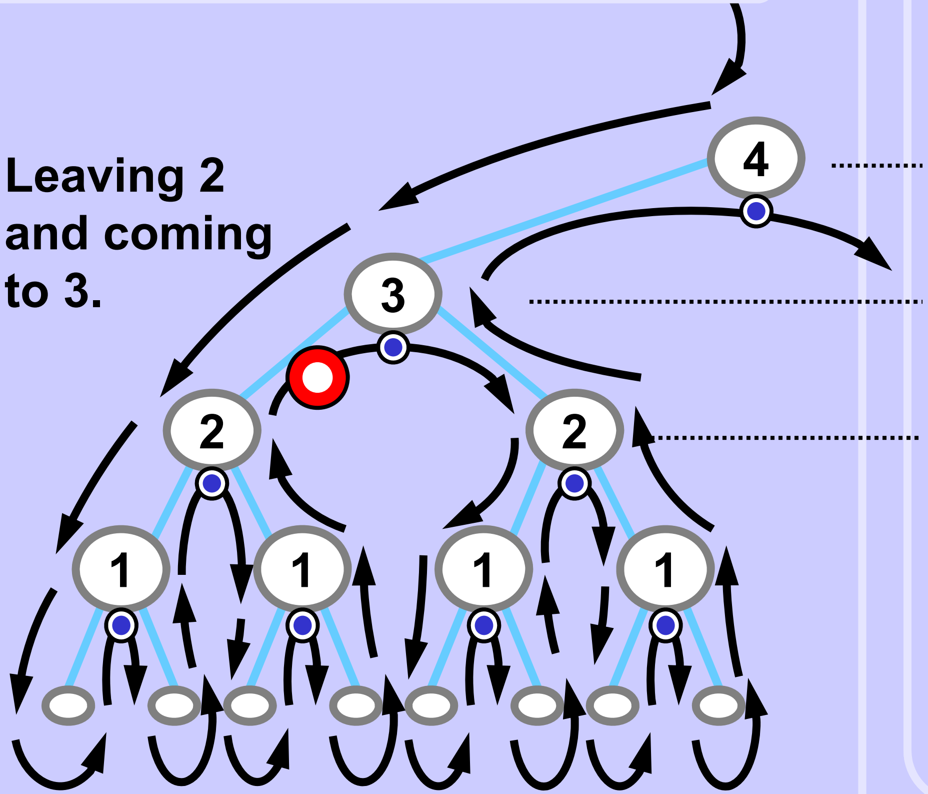
etc…

Stack implements recursion

… after a while …

Recursion tree traversal

Leaving 2 and coming to 3.



Stack

Value	Visits
4	1
3	1
2	2

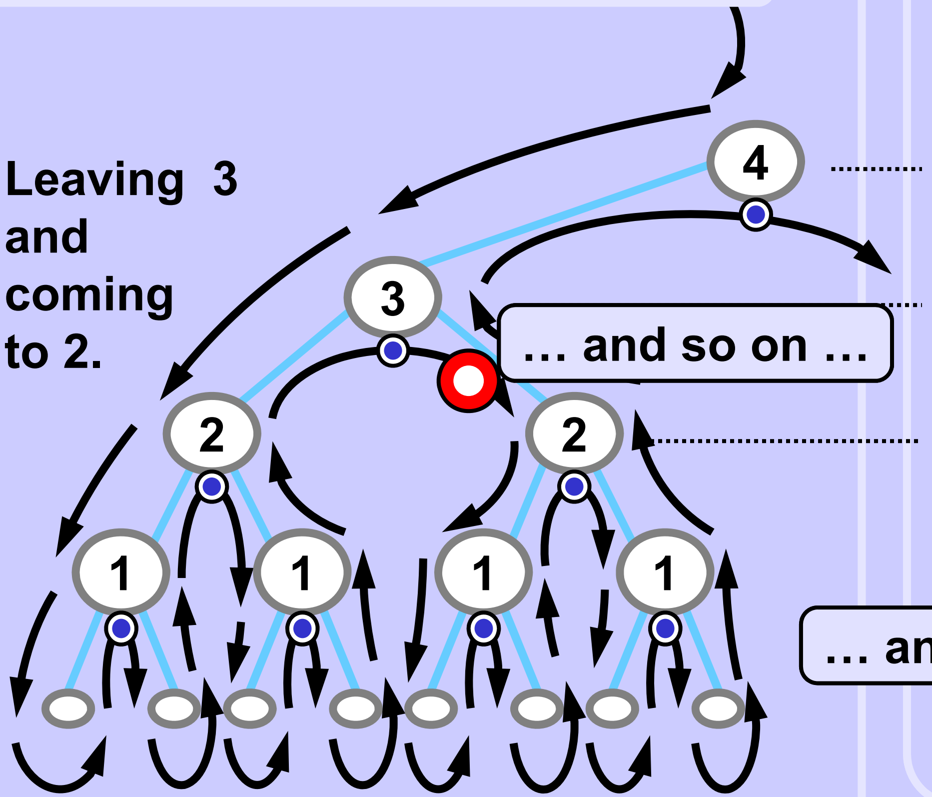
pop()

Output: 1 2 1

Stack implements recursion

Recursion tree traversal

Leaving 3 and coming to 2.



Stack

Value	Visits
4	1
3	2
2	0

push(2,0)

… and so on …

… and so on …

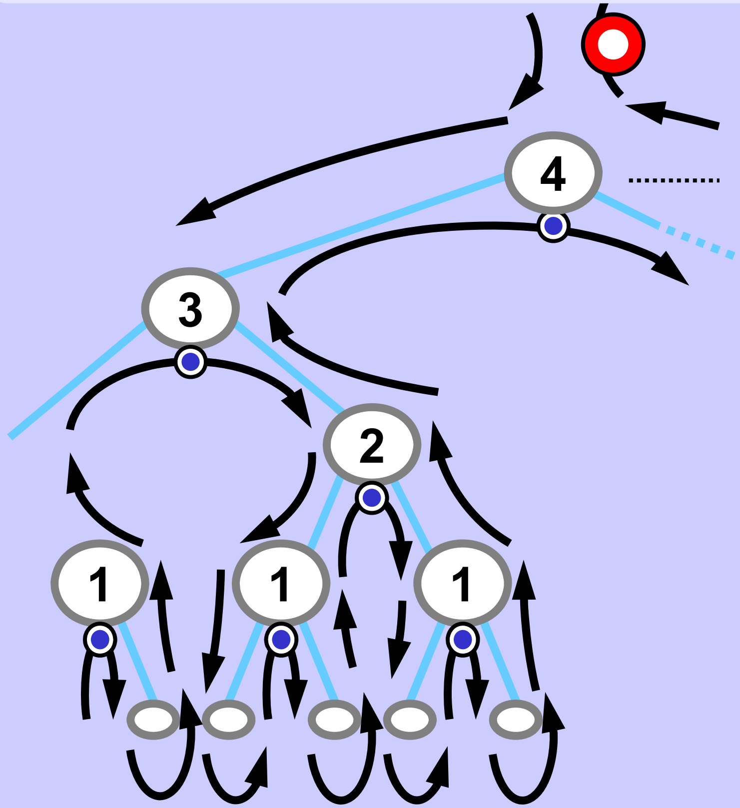
Output

1 2 1 3

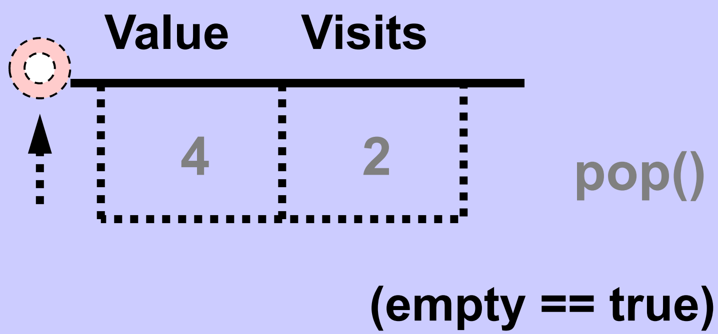
Stack implements recursion

… after another while … completed.

Recursion tree traversal



Stack



1	2	1	3	1	2	1	4	1	2	1	3	1	2	1

Output

Stack implements recursion

Recursive ruler without recursive calls
Pseudocode, nearly a code

```
def rulerNoRec( N ):
  stack = Stack()
  stack.push( N, 0) # 0 == no. of visits to the root
  while not stack.isEmpty():
    if stack.top().value == 0: stack.pop()
    if stack.top().visits == 0:
      stack.top().visits += 1
      stack.push( stack.top().value-1, 0)
    elif stack.top().visits == 1:
      print(stack.top().value, end = ' ')
      stack.top().visits += 1
      stack.push(stack.top().value-1, 0)
    elif stack.top().visits == 2:
       stack.pop()
```

Stack implements recursion

Recursive ruler without recursive calls
Easy implementation with arrays

```
def rulerWithArrays( N ):
  max = 100                        # fixed, for simplicity
  stackVal = [0] * max                   # stack Value field
  stackVis = [0] * max                   # stack Visits field
  SP = 0                                 # stack pointer
  stackVis[SP] = 0; stackVal[SP] = N
  while SP >= 0:                         # while unempty
    if stackVal[SP] == 0: SP -= 1        # pop: in leaf
    if stackVis[SP] == 0:                # first visit
      stackVis[SP] += 1; SP += 1
      stackVal[SP] = stackVal[SP-1]-1    # go left
      stackVis[SP] = 0;
    elif stackVis[SP] == 1:              # second visit
      print(stackVal[SP], end = ' ')     # process the node
      stackVis[SP] += 1;  SP += 1;
      stackVal[SP] = stackVal[SP-1]-1    # go right
      stackVis[SP] = 0;
    elif stackVis[SP] == 2: SP -= 1;     # pop: node done
```

Stack implements recursion

Recursive ruler without recursive calls
Easy implementation with arrays

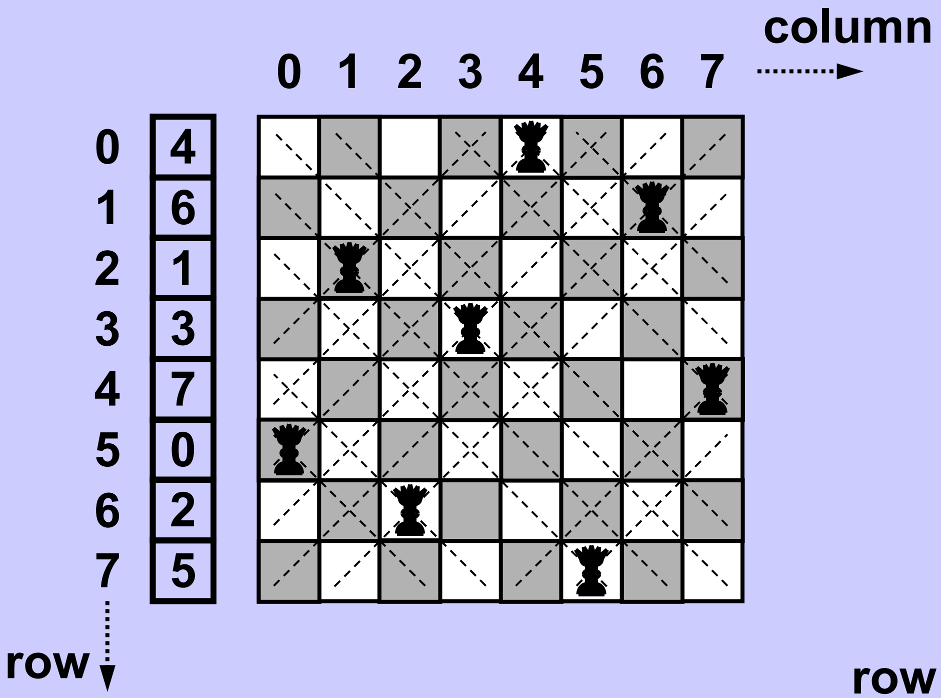
A little more compact code, identical functionality

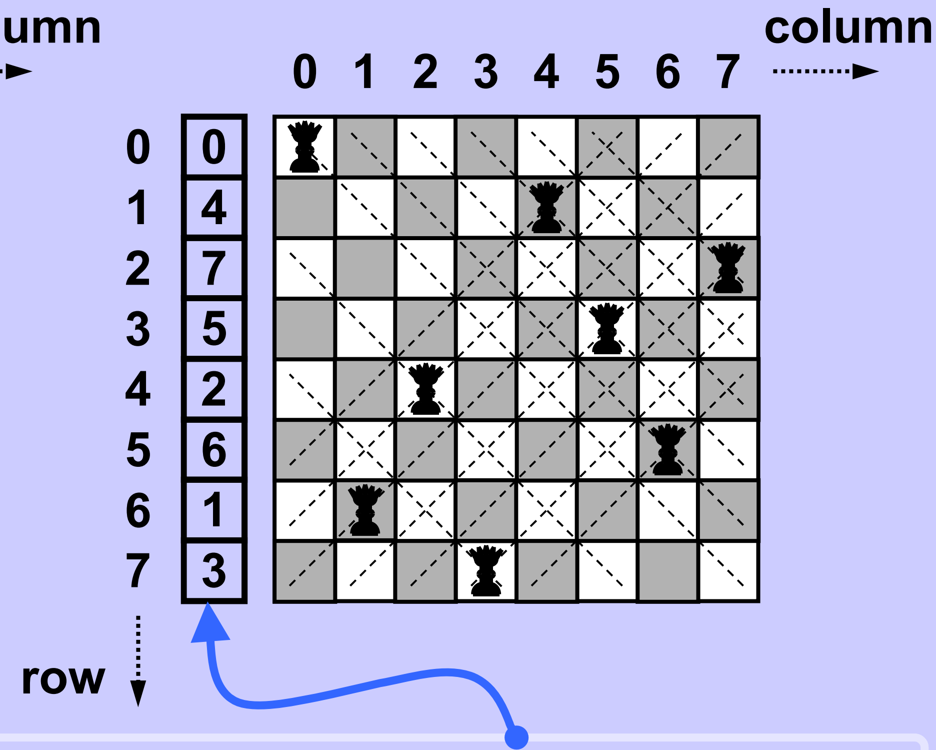
```
def rulerWithArrays2(N):
  stackVal = [0] * 100;  stackVis = [0] * 100
  SP = 0; stackVis[SP] = 0; stackVal[SP] = N
  while (SP >= 0):                             # while unempty
    if stackVal[SP] == 0: SP -= 1              # pop: in leaf
    if stackVis[SP] == 2: SP -= 1              # pop: node done
    else:
      if stackVis[SP] == 1:                    # if second visit
        print(stackVal[SP], end = '' )         # process the node
      stackVis[SP] += 1; SP += 1               # and
      stackVal[SP] = stackVal[SP-1] - 1        # go deeper
      stackVis[SP] = 0
```

Easy backtrack problem 8 queens puzzle

Put 8 chess queens on a standard 8x8 chessboard
so that no two queens threaten each other.

Some solutions

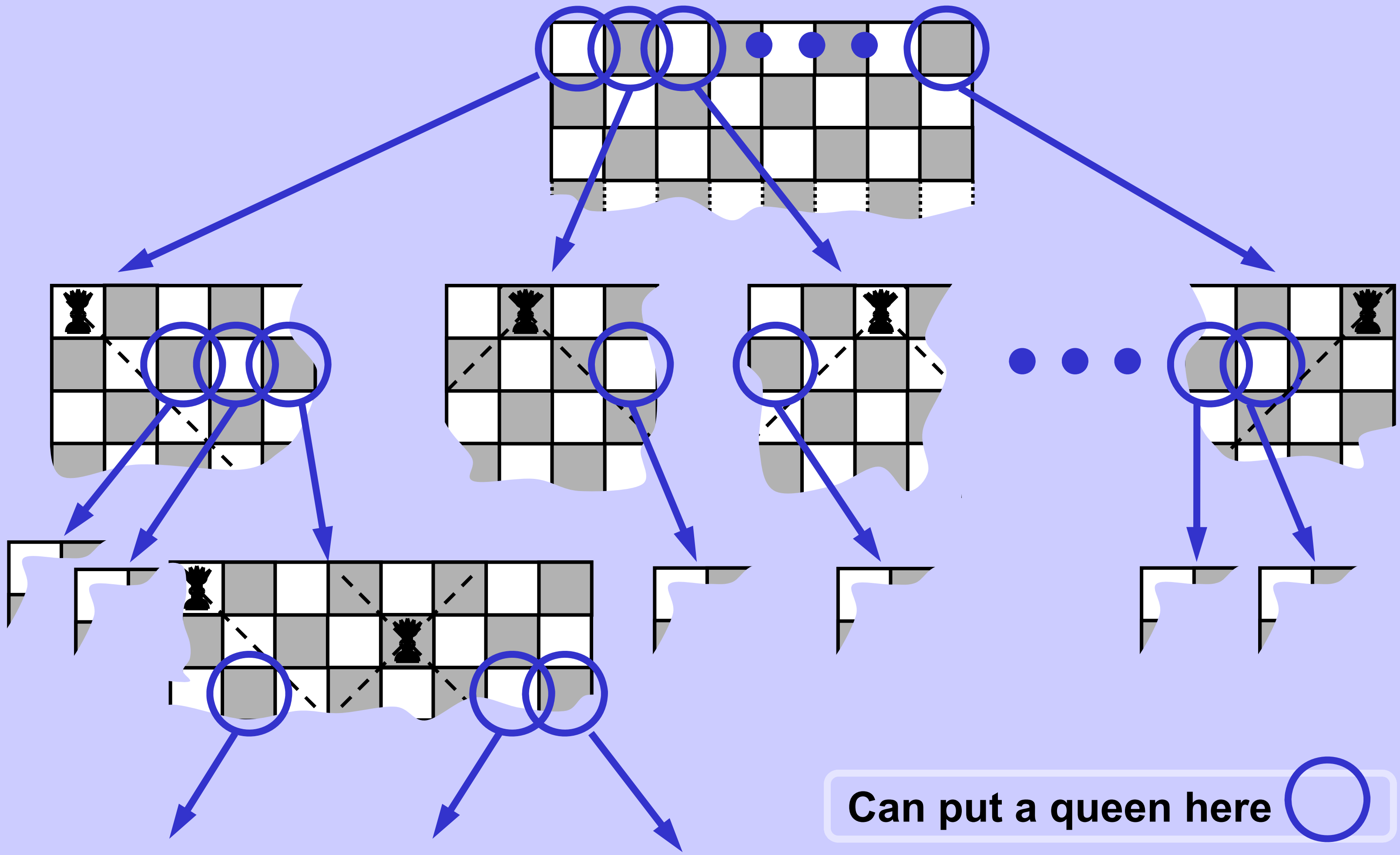




Single data structure: array queenCol[ ] (see the code)

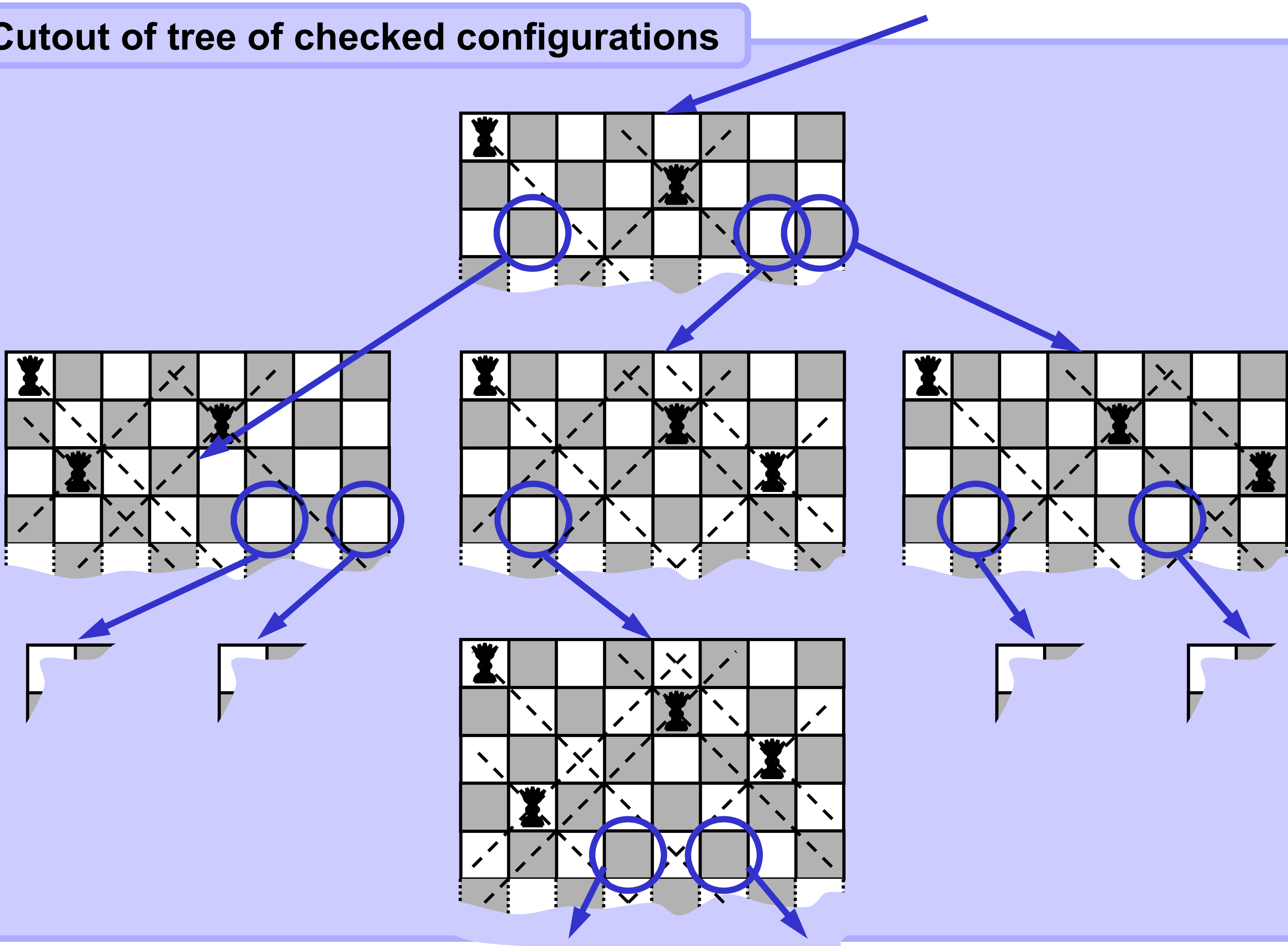
Easy backtrack problem 8 queens puzzle

Tree of checked configurations (a root and a few successors)



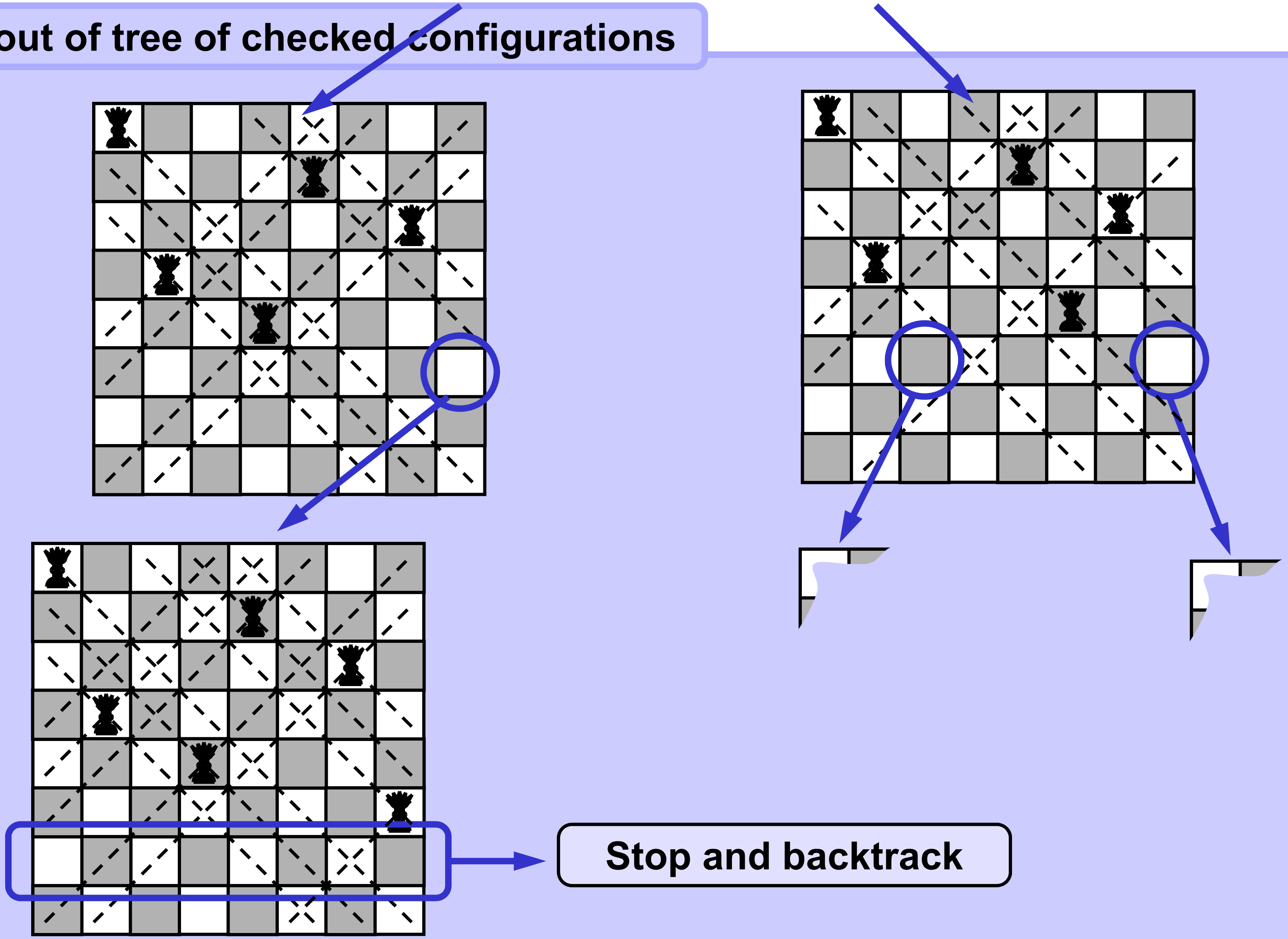
Easy backtrack problem 8 queens puzzle

Cutout of tree of checked configurations



Easy backtrack problem 8 queens puzzle

Cutout of tree of checked configurations



Easy backtrack problem 8 queens puzzle

N queens puzzle (N x N chessboard)

N queens	No. of solutions	No. of tested queen positions		Speedup
		Brute force (N^N)	Backtrack	
4	2	256	240	1.07
5	10	3 125	1 100	2.84
6	4	46 656	5 364	8.70
7	40	823 543	25 088	32.83
8	92	16 777 216	125 760	133.41
9	352	387 420 489	651 402	594.75
10	724	10 000 000 000	3 481 500	2 872.33
11	2 680	285 311 670 611	19 873 766	14 356.20
12	14 200	8 916 100 448 256	121 246 416	73 537.00

Tab 3.1 Speed of N queens puzzle solutions

Easy backtrack problem   8 queens puzzle

```
NQ = 8                                        # number of queens
queenCol = [0 for x in range(NQ)]             # 1D array is enough

def positionOK( r, c ):                       # r: row, c: column
  for i in range( 0, r ):
    if queenCol[i] == c  or \                 #same column or
       abs(r-i) == abs(queenCol[i]-c)):       # same diagonal
      return False
  return True

def putQueen( row, col ):
  queenCol[row] = col;                    # put a queen there
  if row == NQ-1:                         # if solved
    print( queenCol )                # output solution
  else:
    for c in range( 0, NQ ):              # test all columns
      if positionOK( row+1, c ):          # if free
        putQueen( row+1, c )              # next row recursion

Call:    for col in range( NQ ): putQueen( 0, col )
```

8 queens puzzle - More intuitive output

```
def printQ():
  for row in range(0, NQ):
    for col in range( 0, NQ ):
      if col == queenCol[row]: print( " Q", end = '')
      else:                    print( " .", end = '')
     print() # end of row
  print() # extra empty line
```

All 10 cases for 5 queens (NQ = 5)

```
 Q . . . .
 . . Q . .
 . . . . Q
 . Q . . .
 . . . Q .
```

```
 . Q . . .
 . . . Q .
 Q . . . .
 . . Q . .
 . . . . Q
```

```
 . . Q . .
 Q . . . .
 . . . Q .
 . Q . . .
 . . . . Q
```

```
 . . . Q .
 Q . . . .
 . . Q . .
 . . . . Q
 . Q . . .
```

```
 . . . . Q
 . Q . . .
 . . . Q .
 Q . . . .
 . . Q . .
```

```
 Q . . . .
 . . . Q .
 . Q . . .
 . . . . Q
 . . Q . .
```

```
 . Q . . .
 . . . . Q
 . . Q . .
 Q . . . .
 . . . Q .
```

```
 . . Q . .
 . . . . Q
 . Q . . .
 . . . Q .
 Q . . . .
```

```
 . . . Q .
 . Q . . .
 . . . . Q
 . . Q . .
 Q . . . .
```

```
 . . . . Q
 . . Q . .
 Q . . . .
 . . . Q .
 . Q . . .
```