Queue

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

Breadth-first search (BFS) in a binary tree with the help of a queue

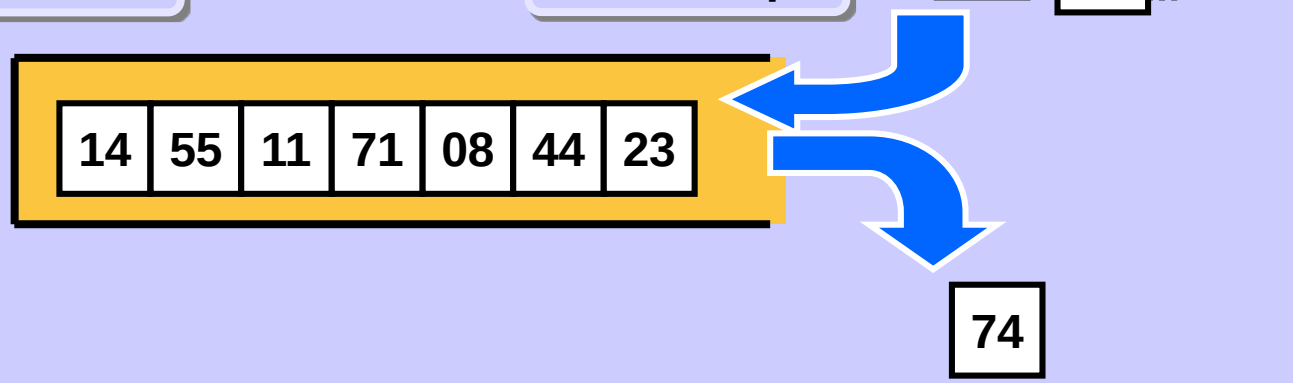
Stack

Operations pop, push, Empty...

Processing a tree or a recursive task with the help of a stack

Stack

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



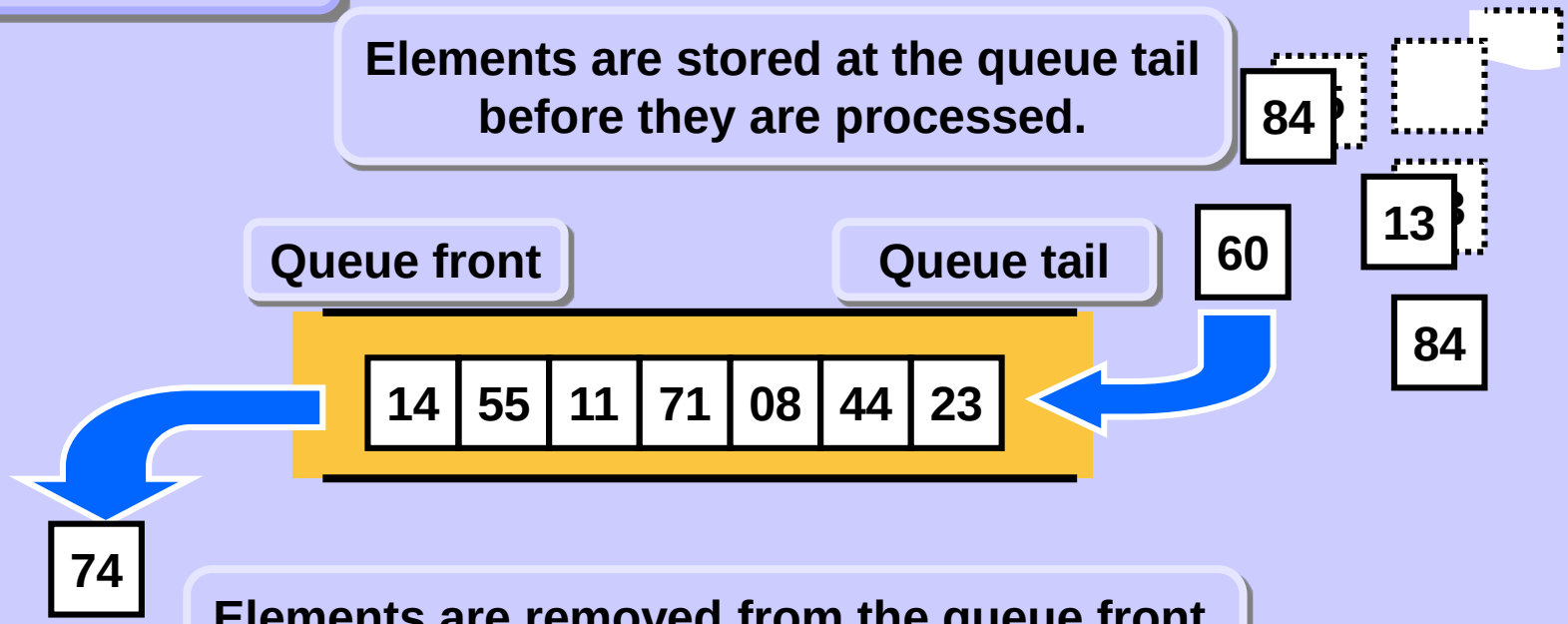
Elements are removed from the stack top and then they are processed.

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.



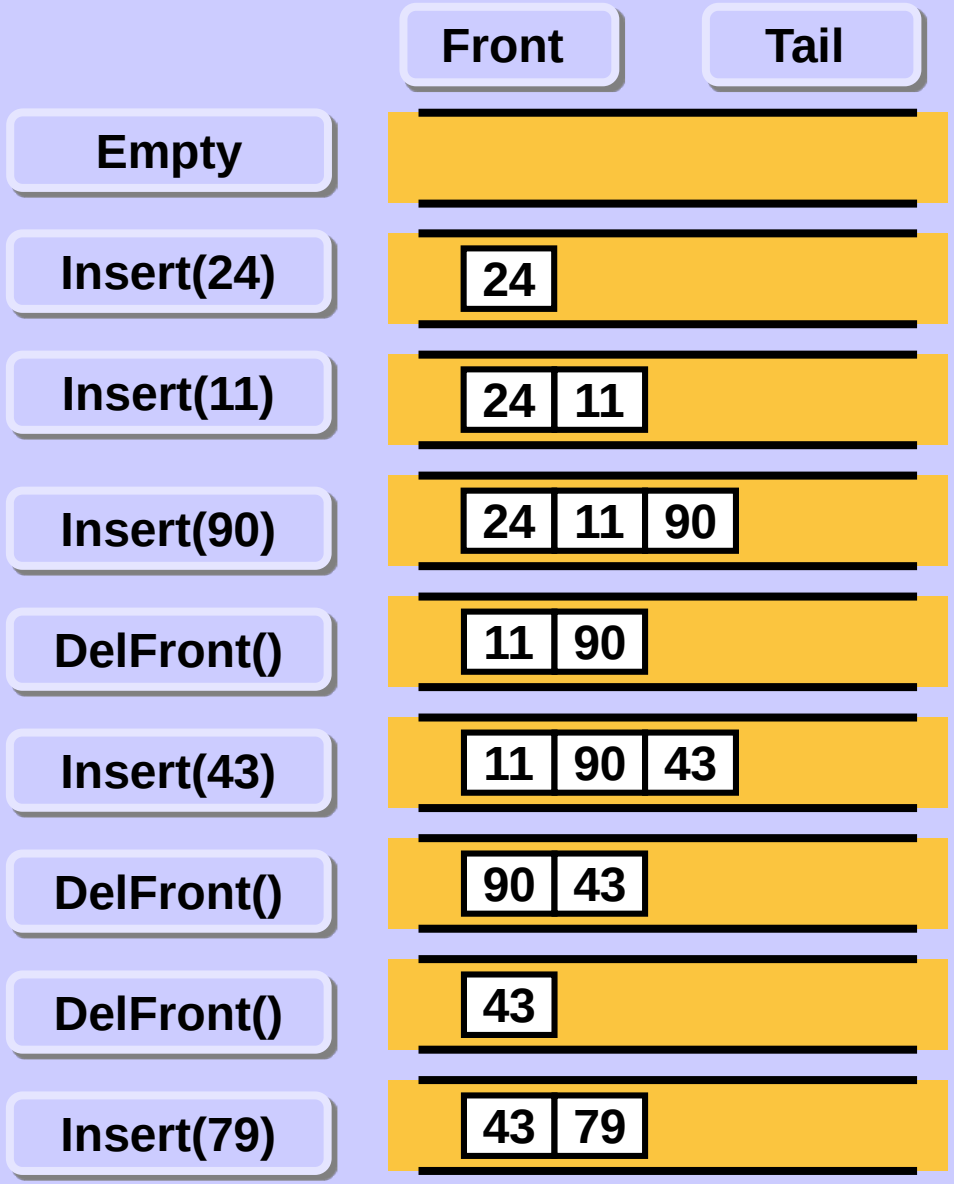
Elements are removed from the queue front and then they are processed.

Operation names

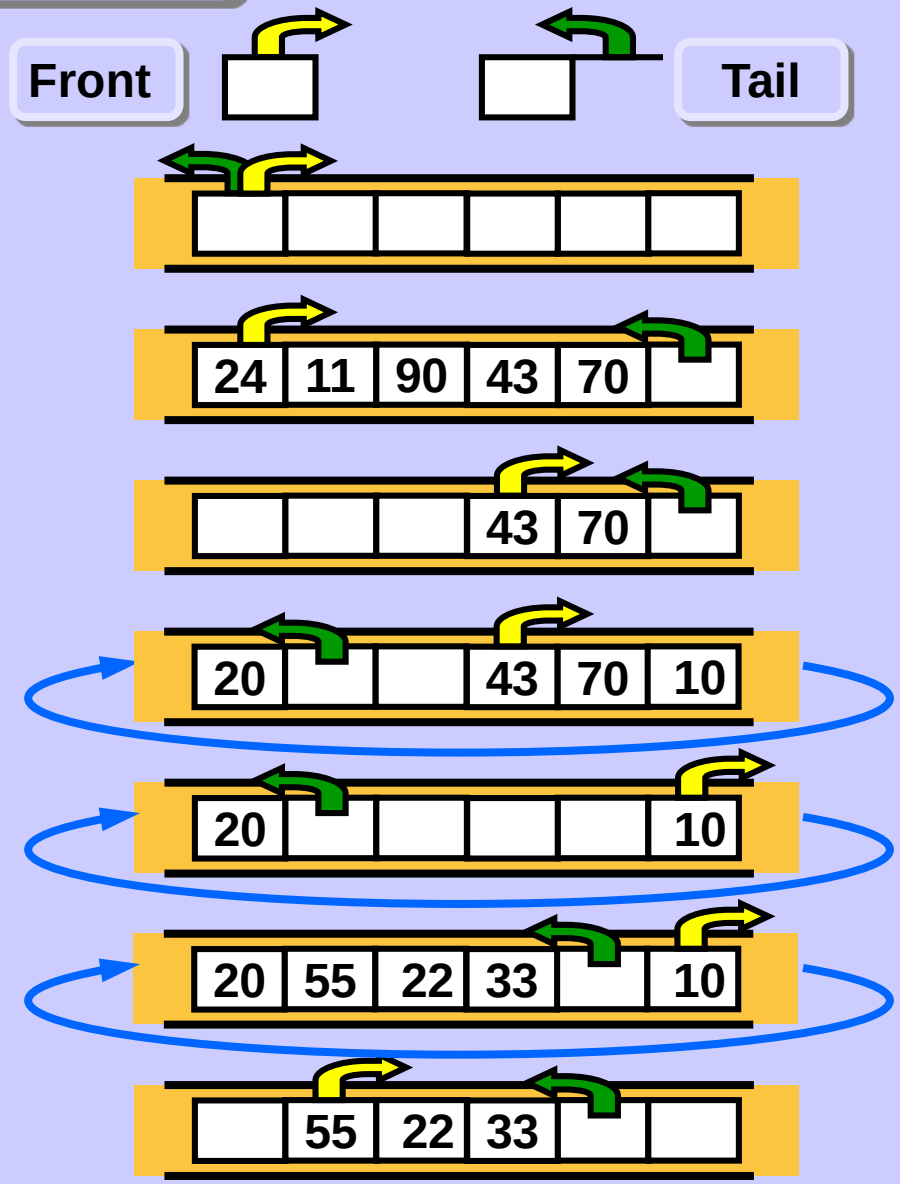
Insert at the tail	Enqueue / InsertLast / Push ...
Remove from the front	Dequeue / delFront / Pop ...
Read the front elem	Front / Peek ...
Is the queue empty?	Empty

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.

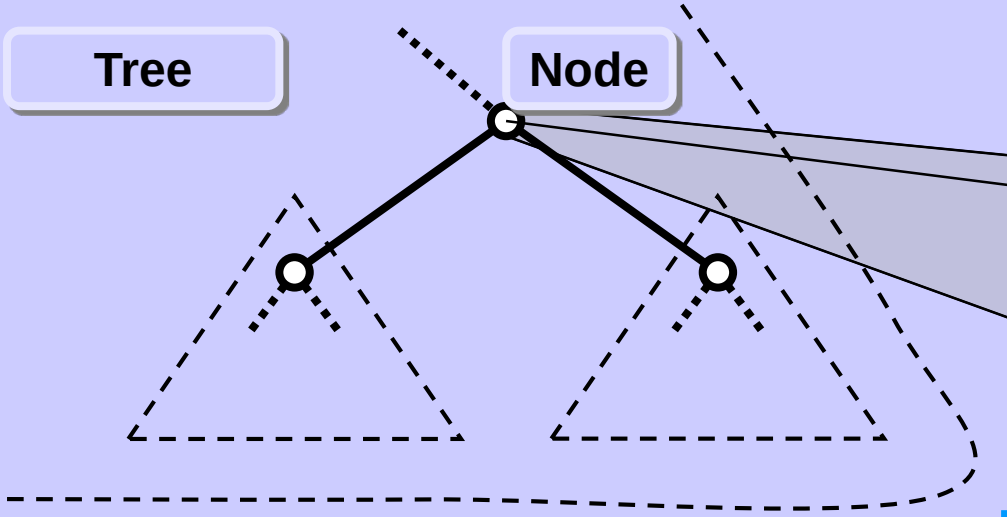
... continued

```
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)
```

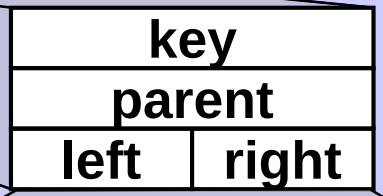
TREES, BINARY TREES

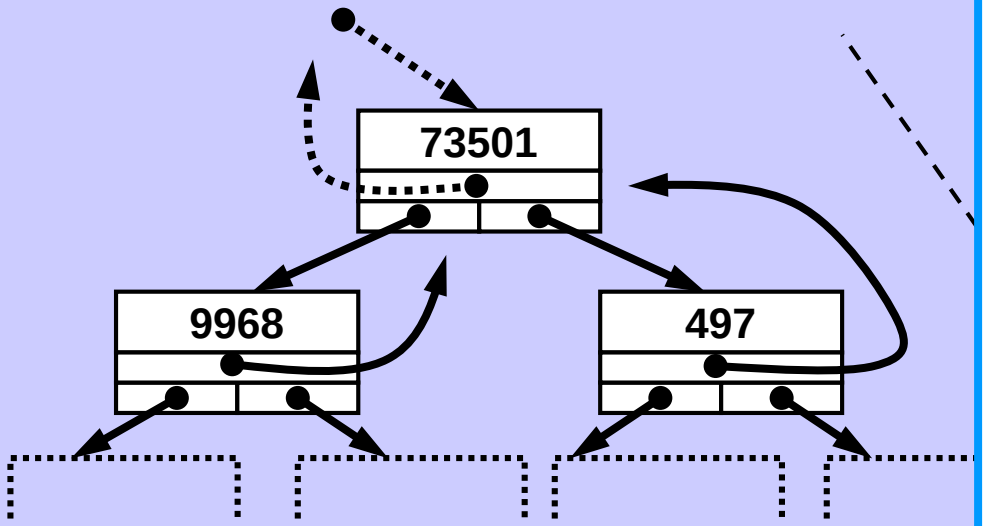
Traversing a tree with Breadth-First Search (BFS)

Binary tree implementation -- Python



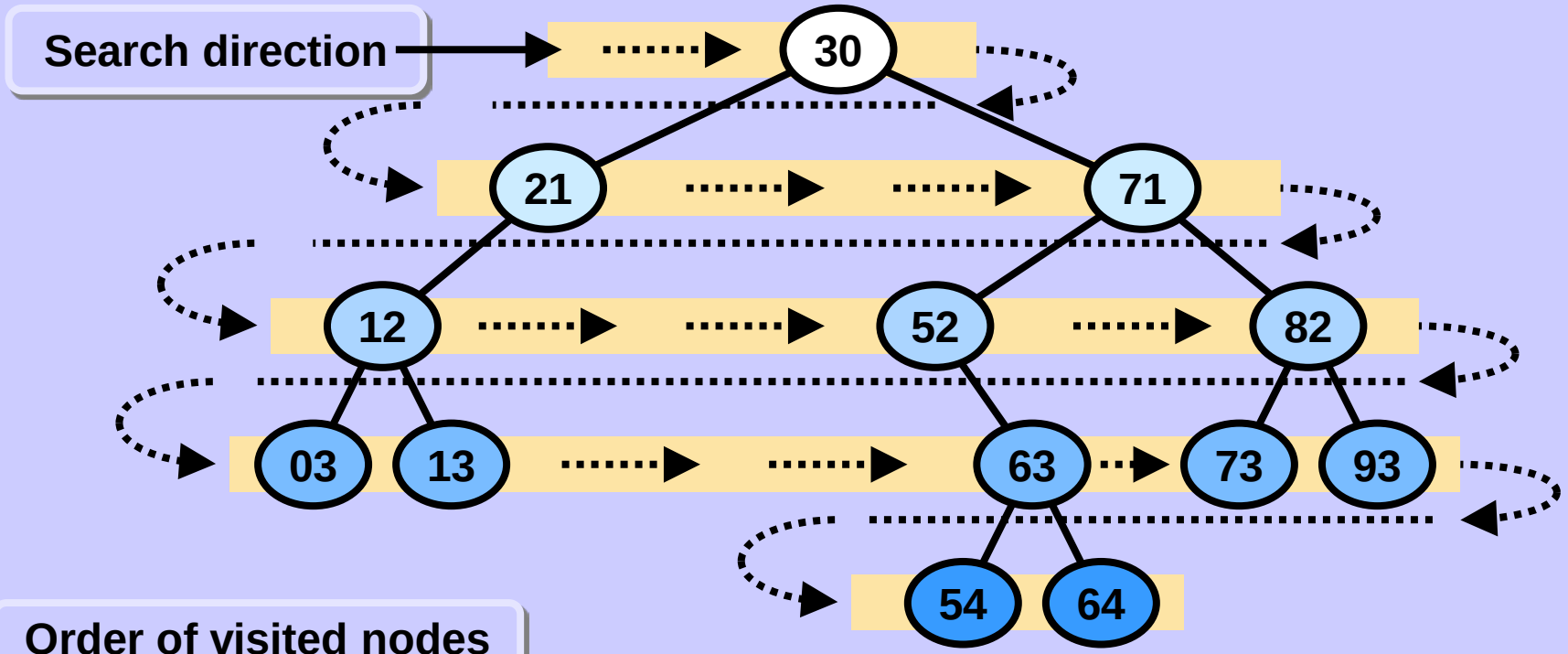
Node representation






```
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
```

Breadth-first search (BFS) of a tree



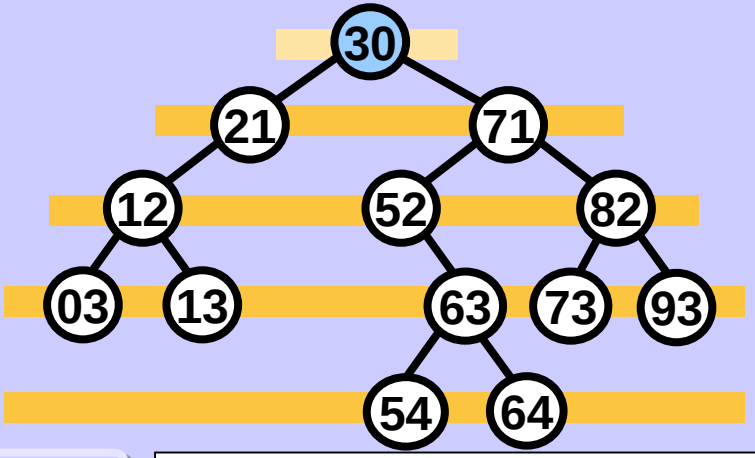
Order of visited nodes

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

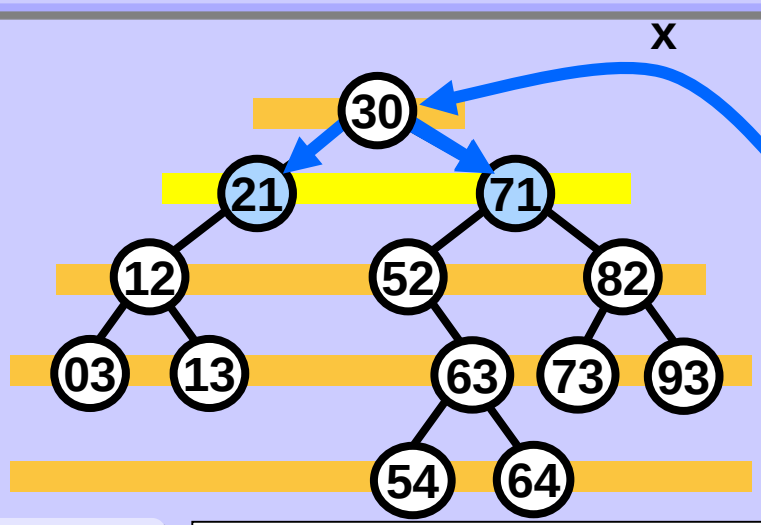
Tail

Main loop

While the queue is not empty do:

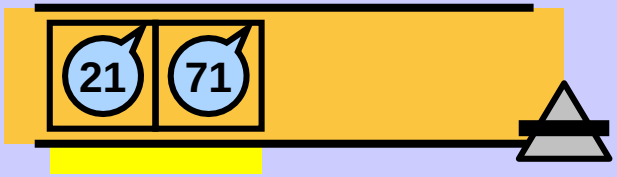
1. Remove the first element from the queue and process it.
2. Enqueue the children of removed element.

Breadth-first search (BFS) of a tree

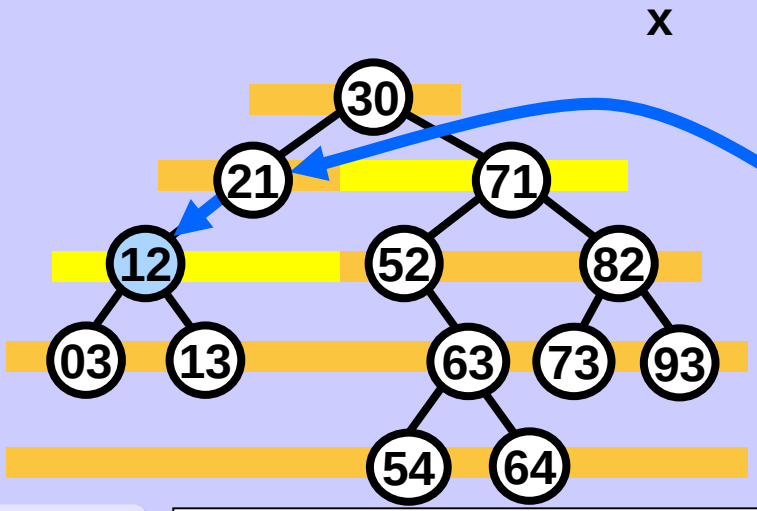


1. x = Dequeue(), print (x.key).

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



Output: 30



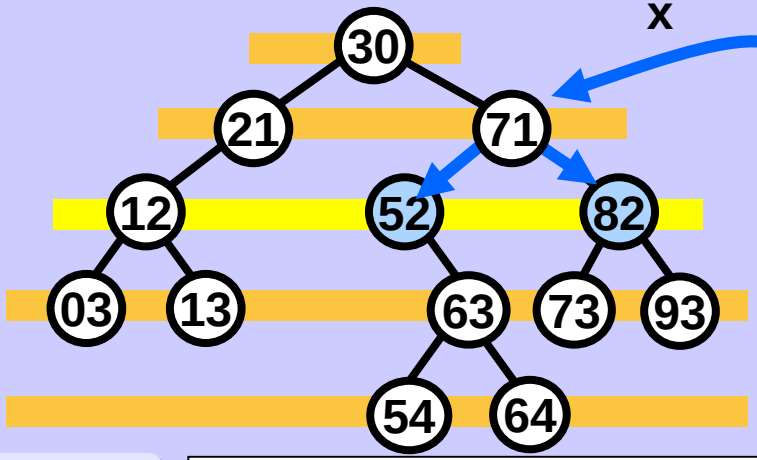
1. x = Dequeue(), print (x.key).

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

Output: 30 21

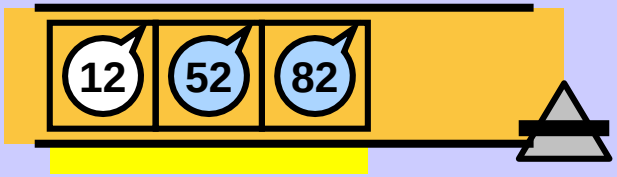
*) if exists

Breadth-first search (BFS) of a tree

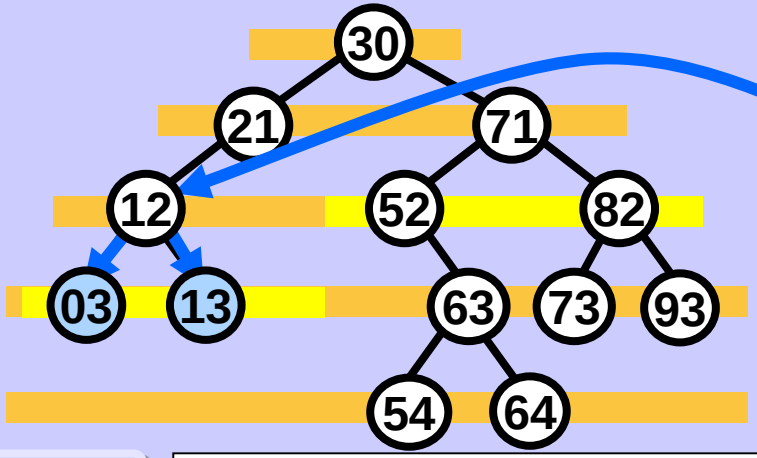


1. x = Dequeue(), print (x.key).

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

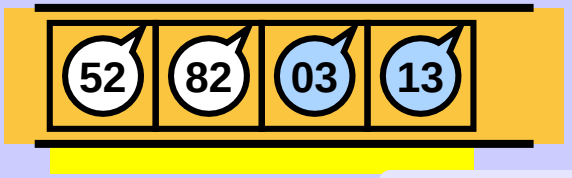


Output: 30 21 71



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

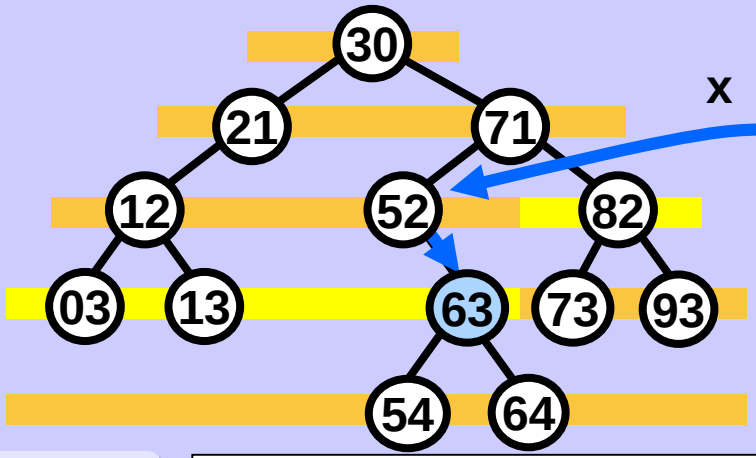
2. Enqeue(x.left), Enqueue(x.right). *)



Output: 30 21 71 12

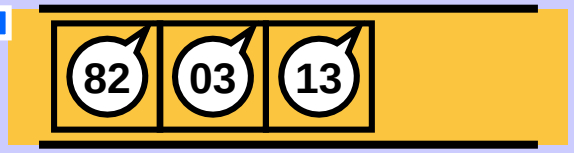
*) if exists

Breadth-first search (BFS) of a tree

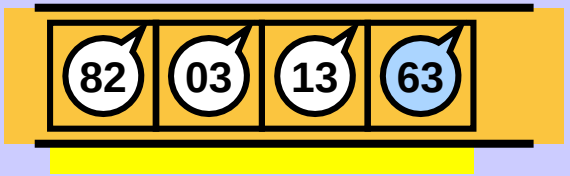


x

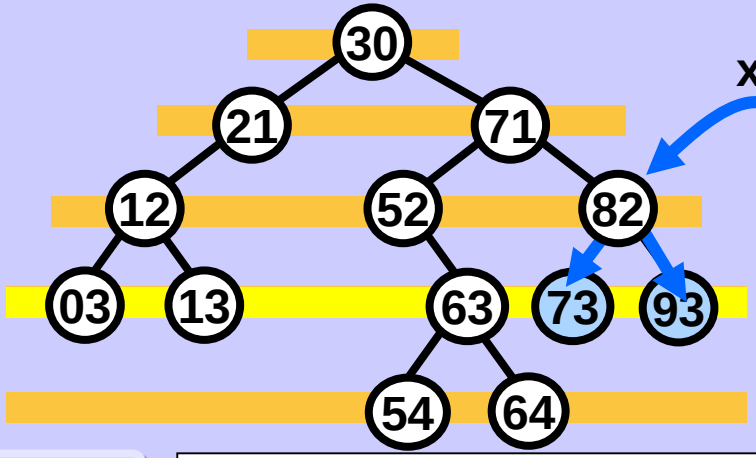
1. x = Dequeue(), print (x.key).



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

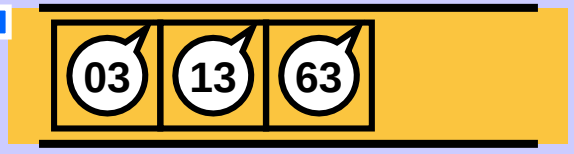


Output: 30 21 71 12 52

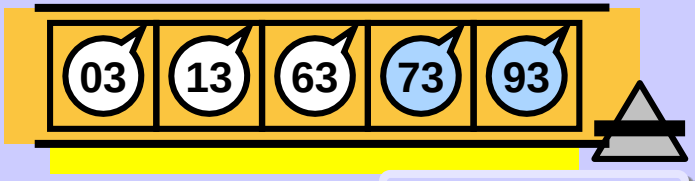


x

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



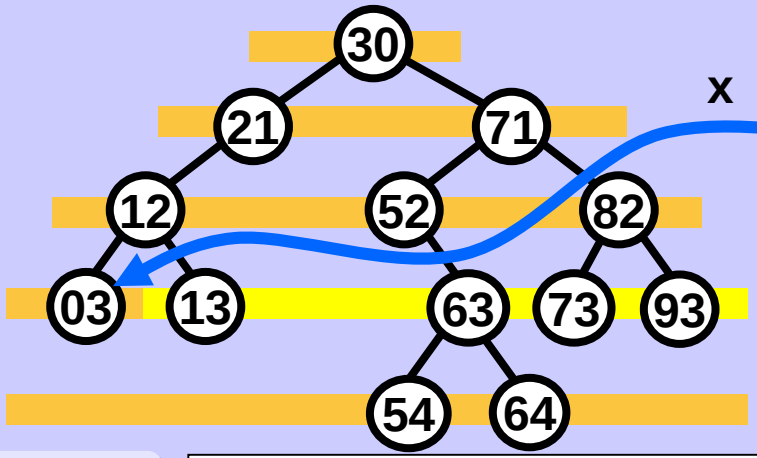
2. Enqeue(x.left), Enqueue(x.right). *)



Output: 30 21 71 12 52 82

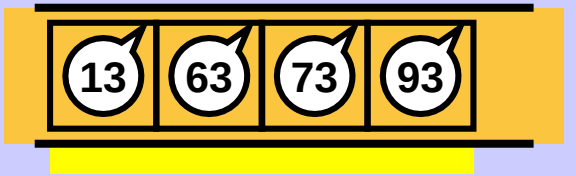
*) if exists

Breadth-first search (BFS) of a tree

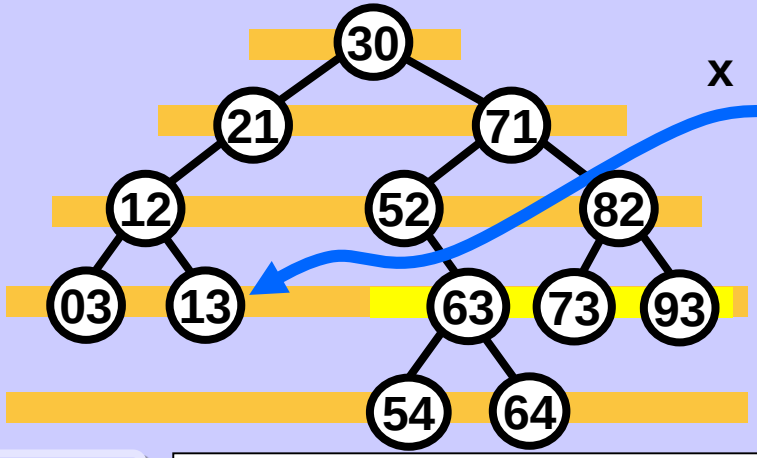


1. x = Dequeue(), print (x.key).

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

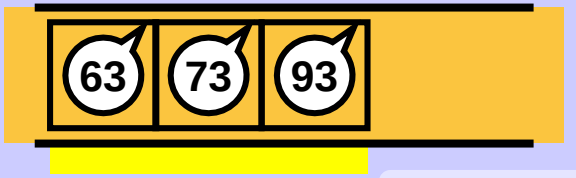


Output: 30 21 71 12 52 82 03



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

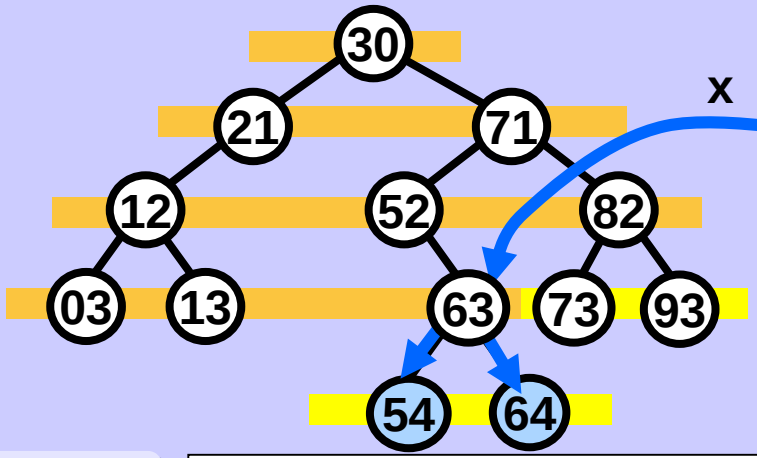
2. Enqeue(x.left), Enqueue(x.right). *)



Output: 30 21 71 12 52 82 03 13

*) if exists

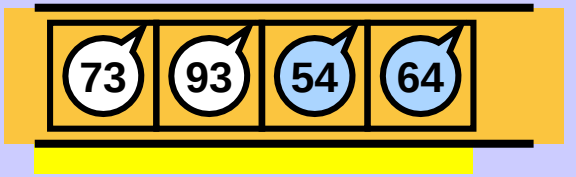
Breadth-first search (BFS) of a tree



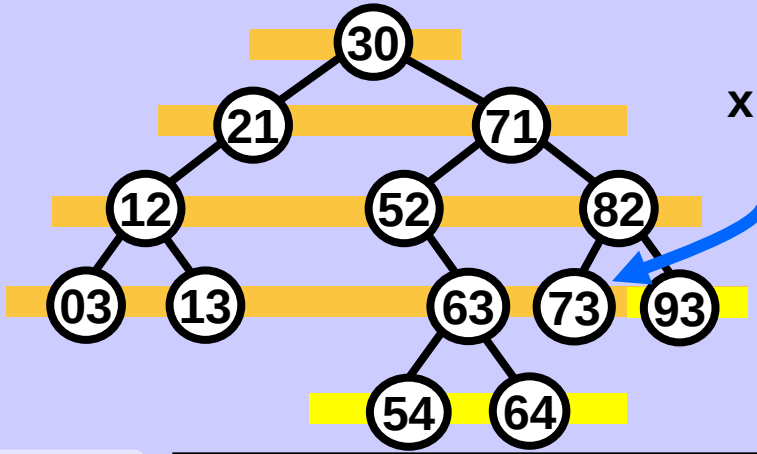
x

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

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



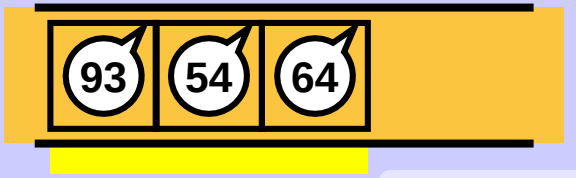
Output: 30 21 71 12 52 82 03 13 63



x

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

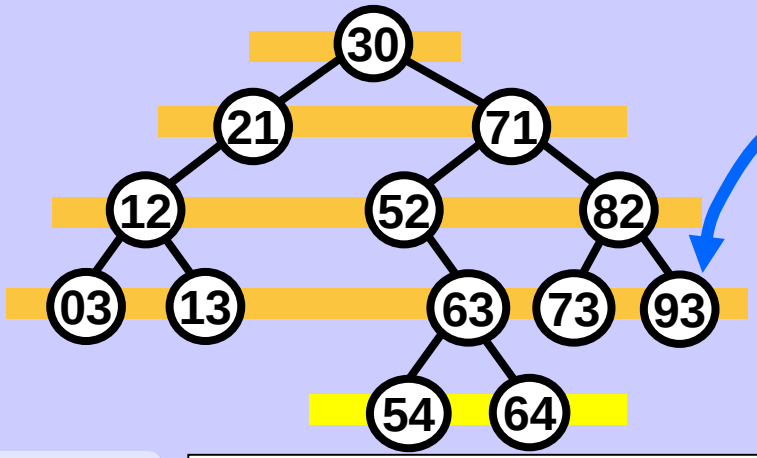
2. Enqeue(x.left), Enqueue(x.right). *)



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

*) if exists

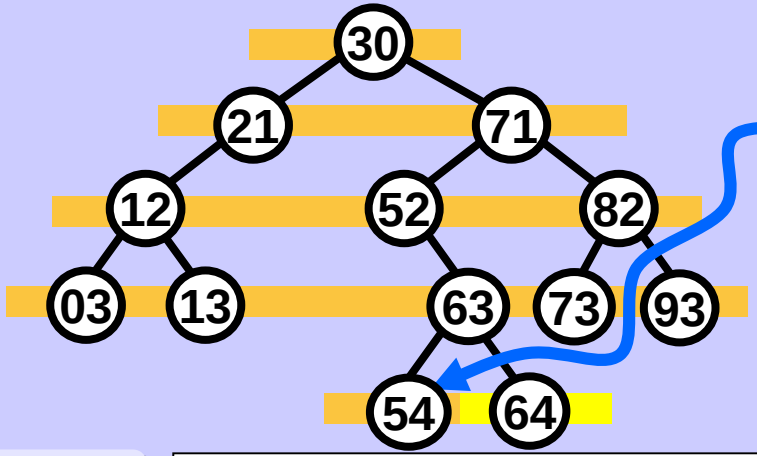
Breadth-first search (BFS) of a tree



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

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

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



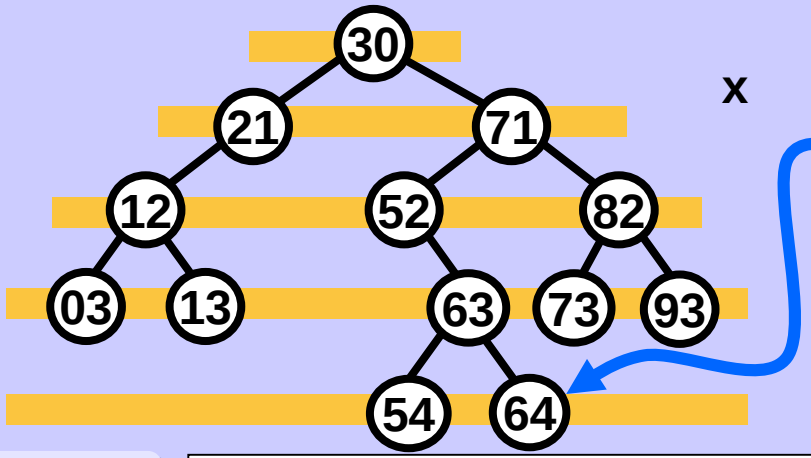
1. x = Dequeue(), print (x.key).

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

64

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

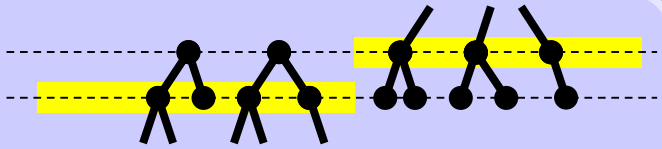
2. Enqeue(x.left), Enqueue(x.right). *)

*) if exists.

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

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.



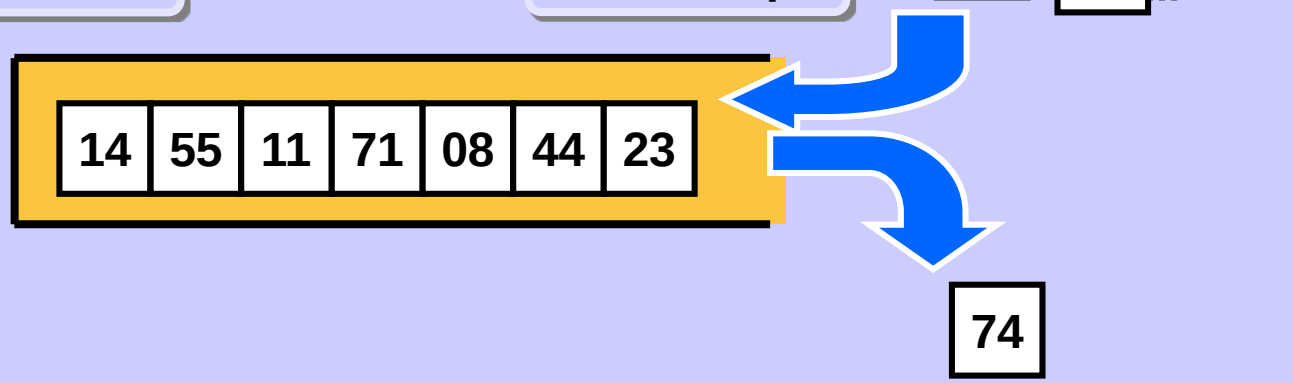
Sometimes the queue contains just nodes of one level. See above:

TREES, STACK

Processing a tree with the help of a stack

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

Stack implements recursion

Standard strategy

Using the stack:

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

Standard approach

1. Before processing an item
   * Push each node (item to be processed) to the stack.

   While processing an item
2. * Process only the node (item) at the top of the stack.

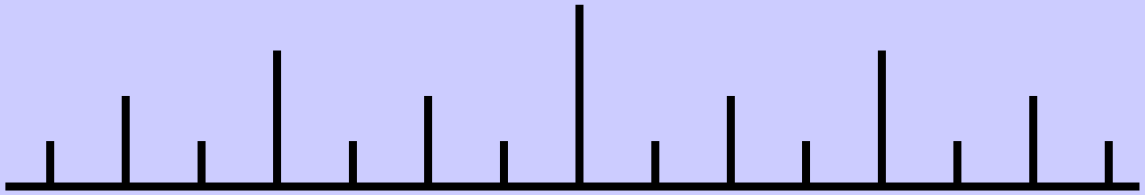
   After processing an item
3. * Pop the processed element from the stack.

Stop when the stack is empty.

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

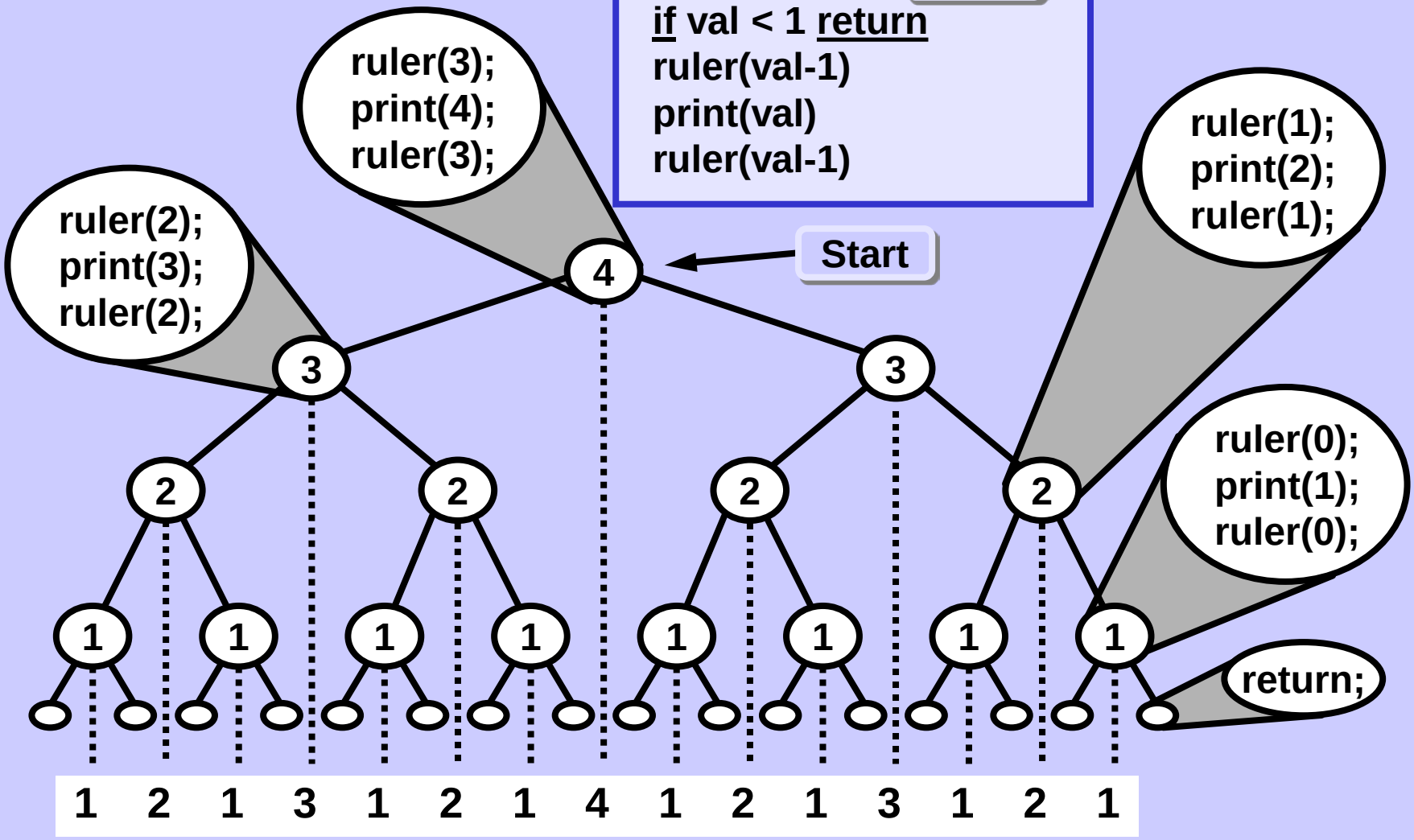
```
def listInOrder( self, node ):
  if node == None: return
  self.listInOrder( node.left )
  print( node.key, end = " " )
  self.listInOrder( node.right )
```

Structurally identical!

Ruler output 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

Simple recursive example

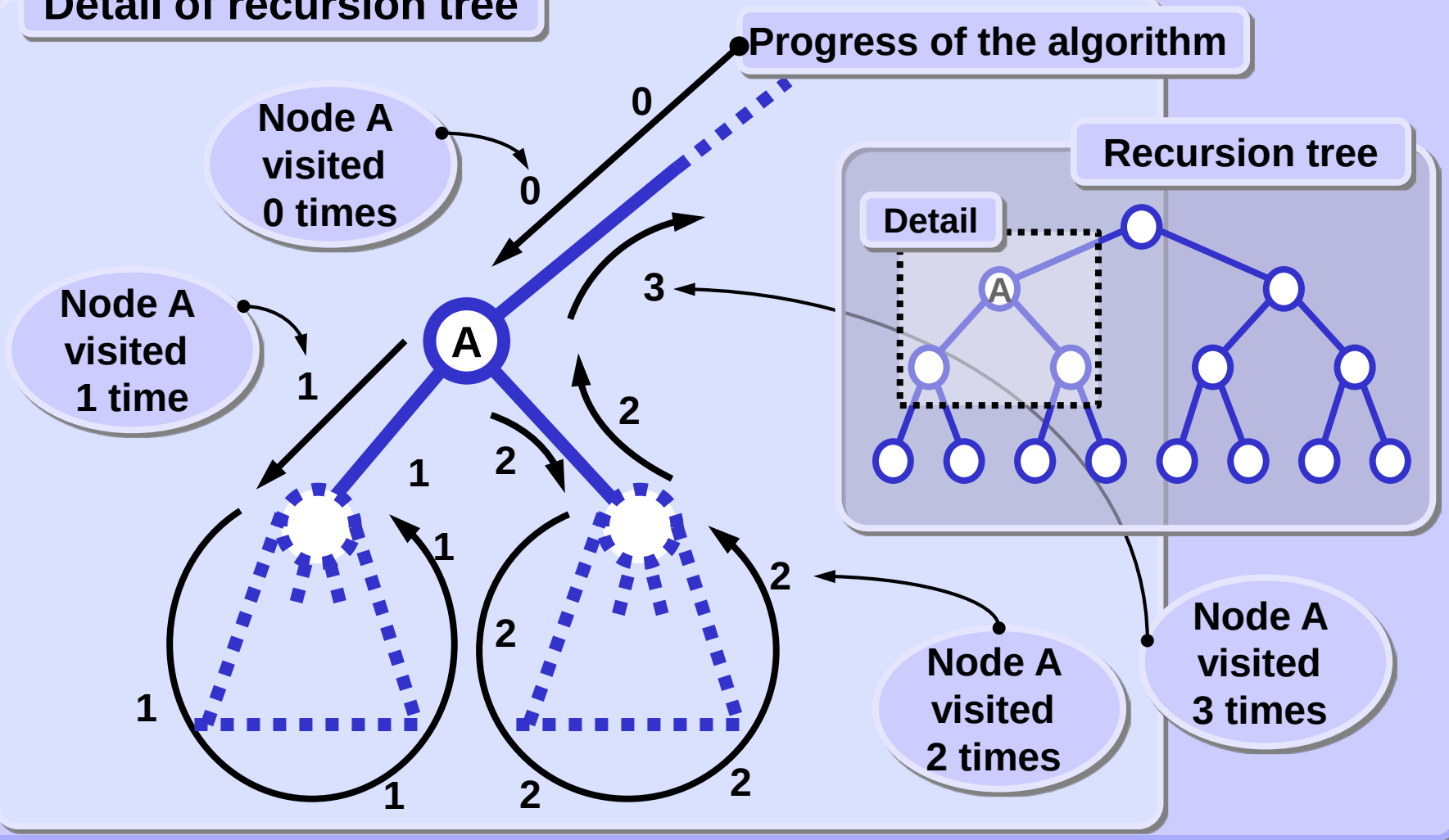
Binary ruler calls



Stack implements recursion

Binary ruler

Detail of recursion tree



Stack implements recursion

Standard strategy

Using the stack:

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

Standard approach

1. Before processing an item
   * Push each node (item to be processed) to the stack.

   While processing an item
2. * Process only the node (item) at the top of the stack.

   After processing an item
3. * Pop the processed element from the stack.

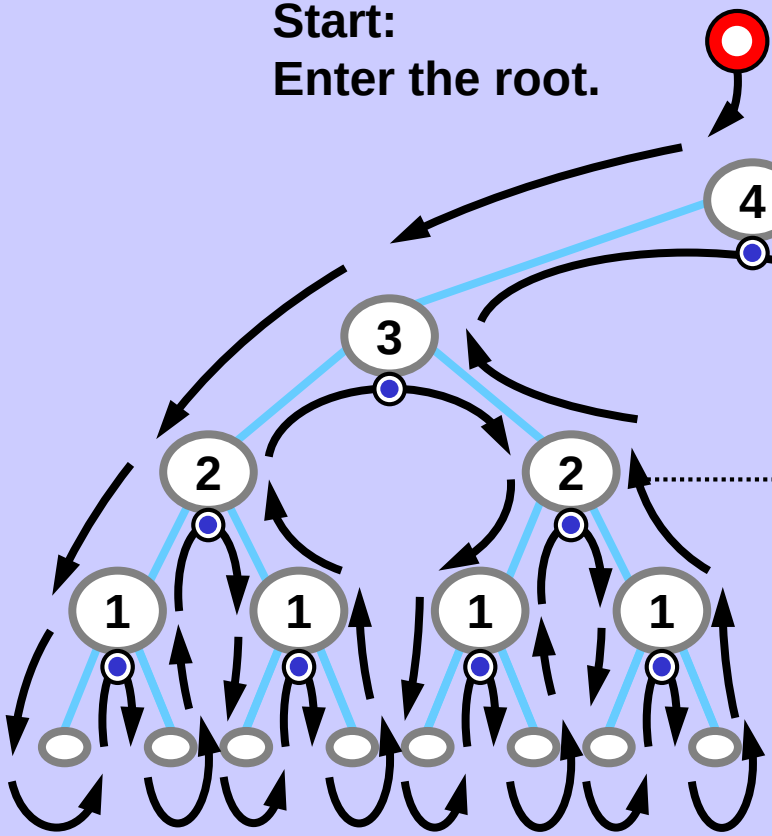
Stop when the stack is empty.

Stack implements recursion

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

Current position

Start:
Enter the root.



Stack

Value	Visits
4	0

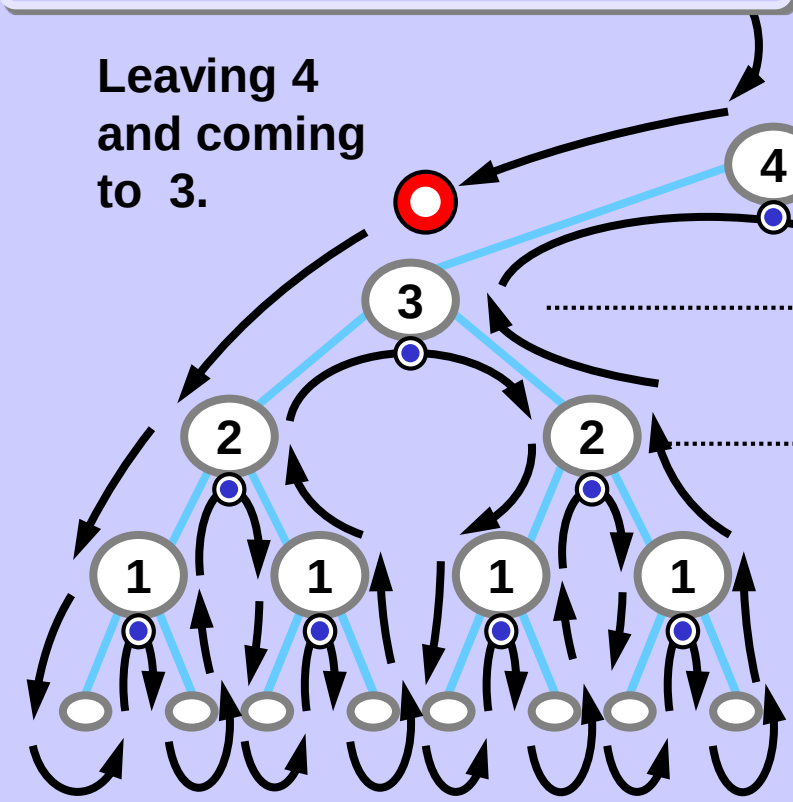
push(4,0)

Output

Stack implements recursion

Recursion tree traversal

Leaving 4
and coming
to 3.



Stack

Value	Visits
4	1
3	0

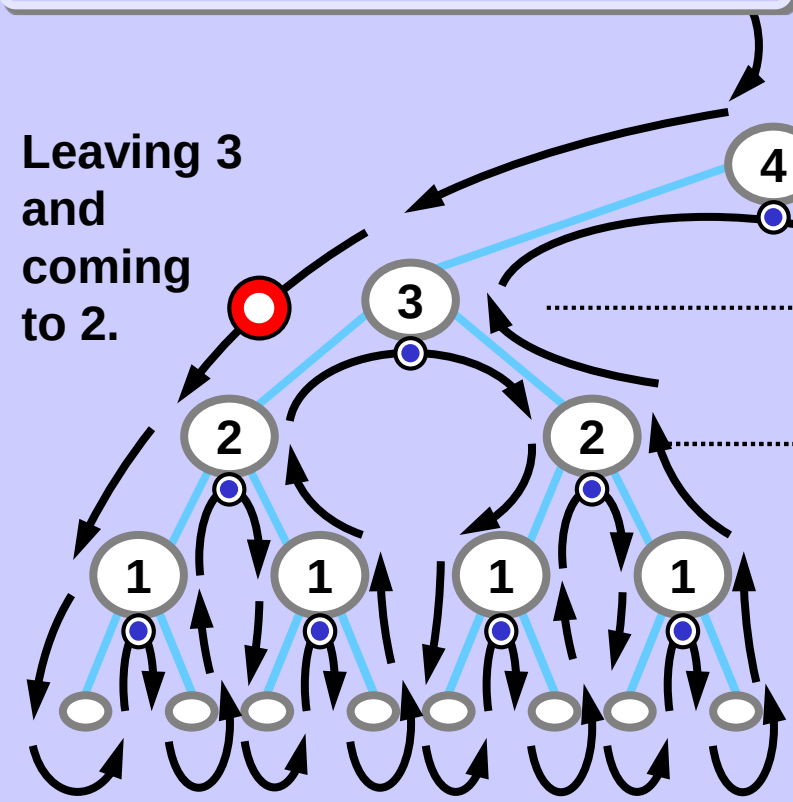
push(3,0)

Output

Stack implements recursion

Recursion tree traversal

Leaving 3 and coming to 2.



Stack

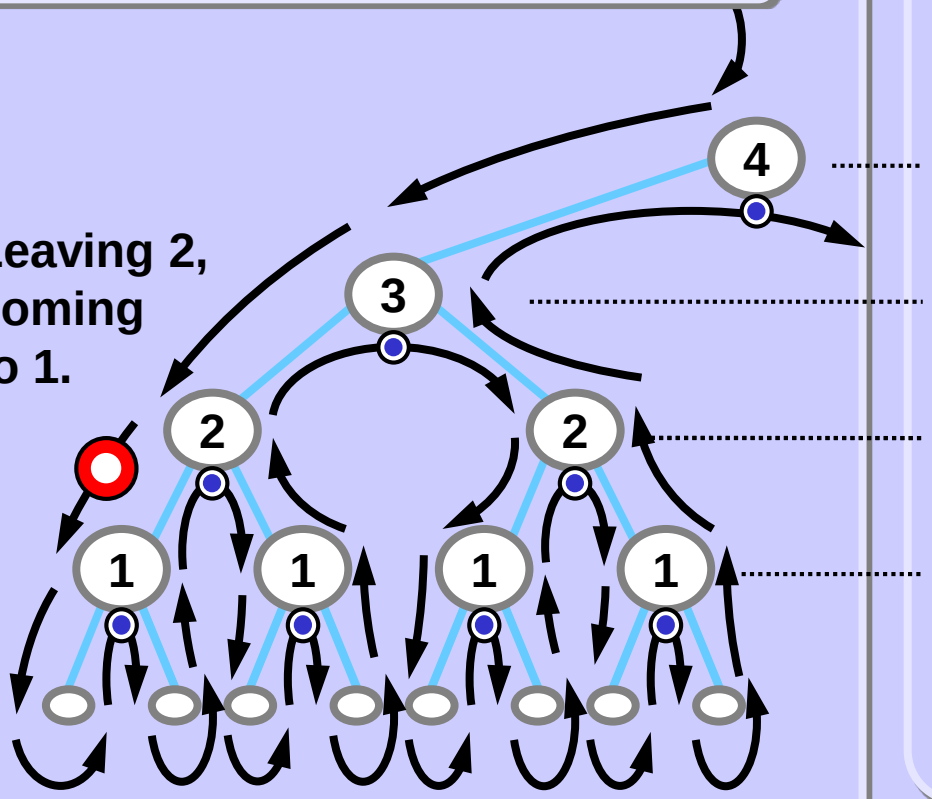
Value	Visits
4	1
3	1
2	0

push(2,0)

Output

Stack implements recursion

Recursion tree traversal



Leaving 2, coming to 1.

Stack

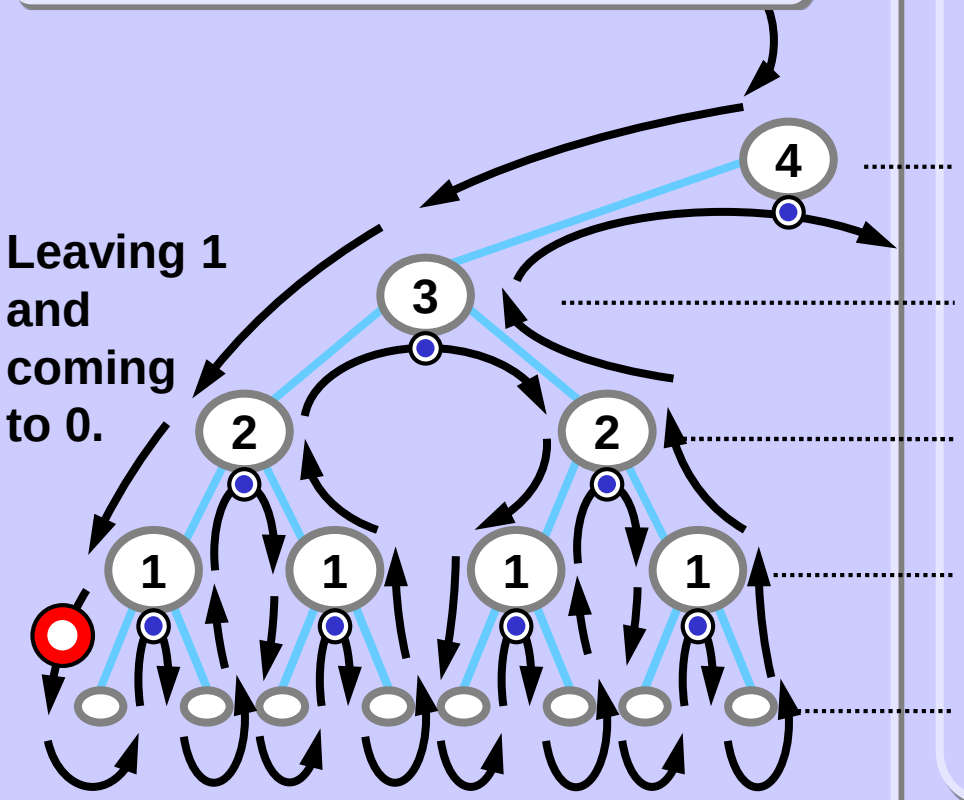
Value	Visits
4	1
3	1
2	1
1	0

push(1,0)

Output

Stack implements recursion

Recursion tree traversal



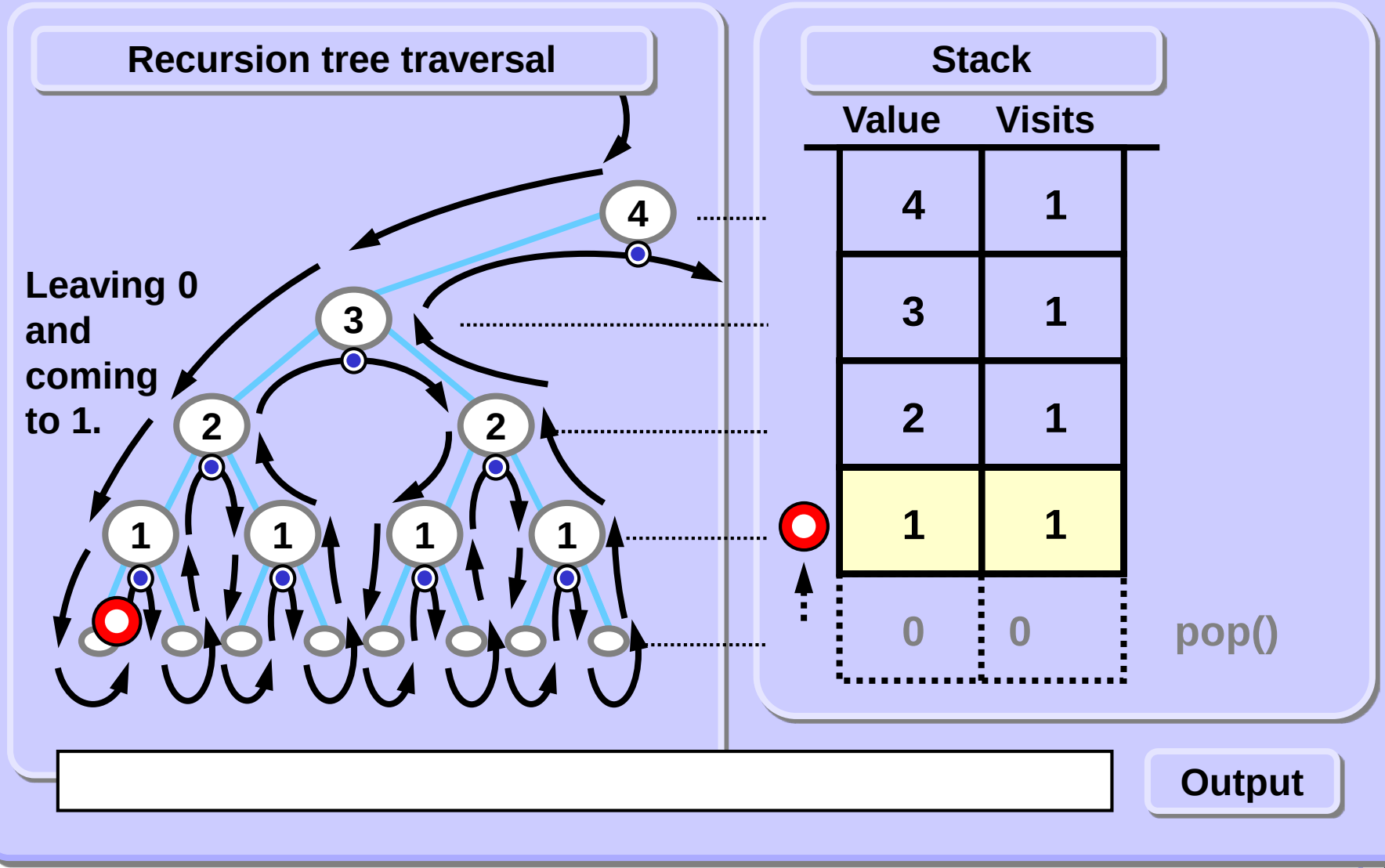
Stack

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

push(0,0)

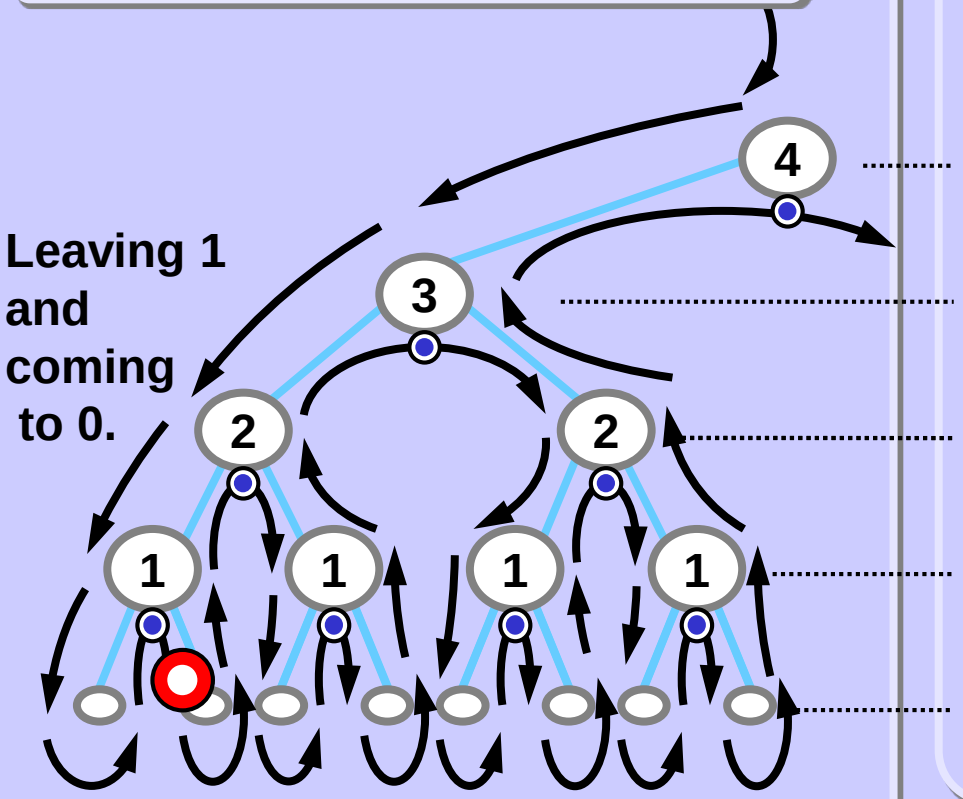
Output

Stack implements recursion



Stack implements recursion

Recursion tree traversal



Stack

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

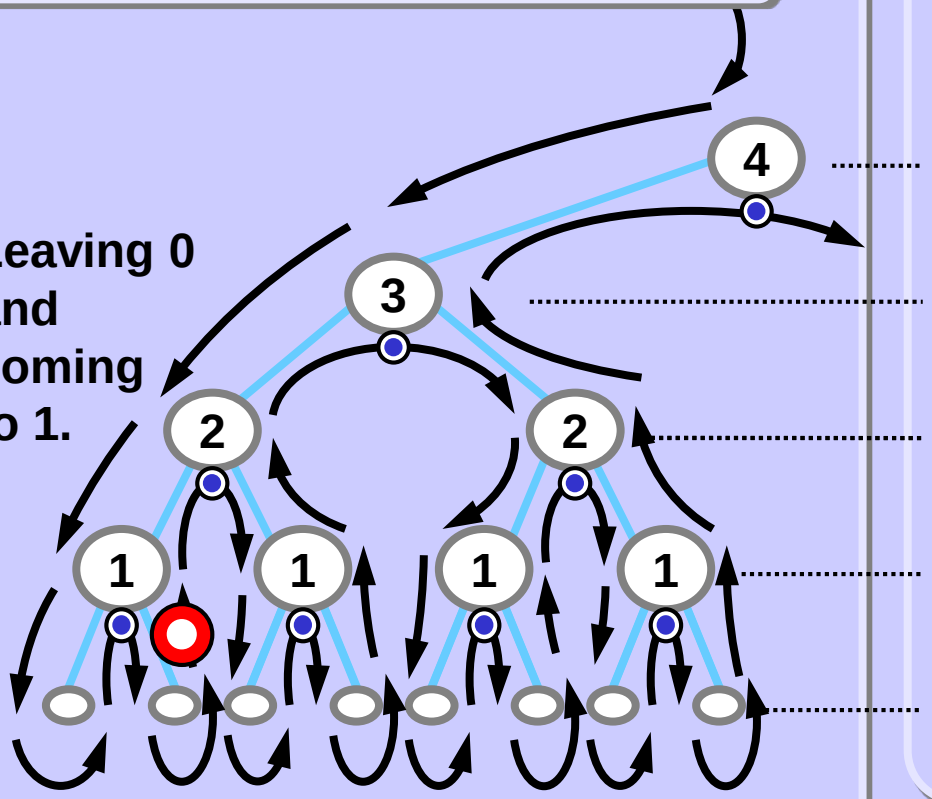
push(0,0)

1

Output

Stack implements recursion

Recursion tree traversal



Stack

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

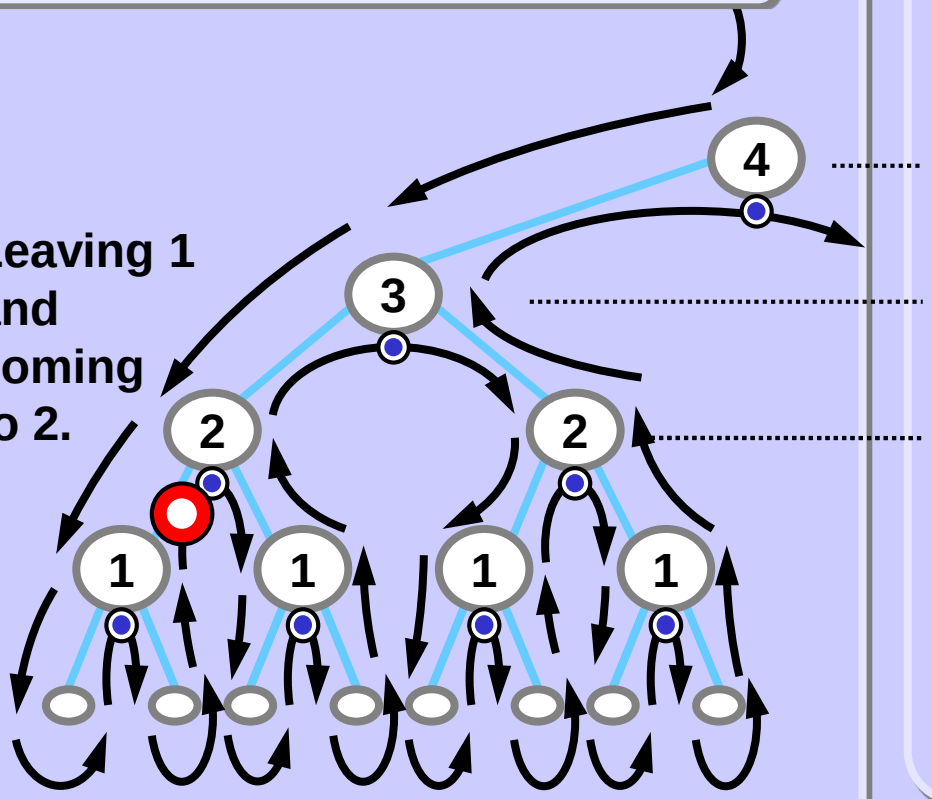
pop()

1

Output

Stack implements recursion

Recursion tree traversal



Stack

Value	Visits
4	1
3	1
2	1
1	2

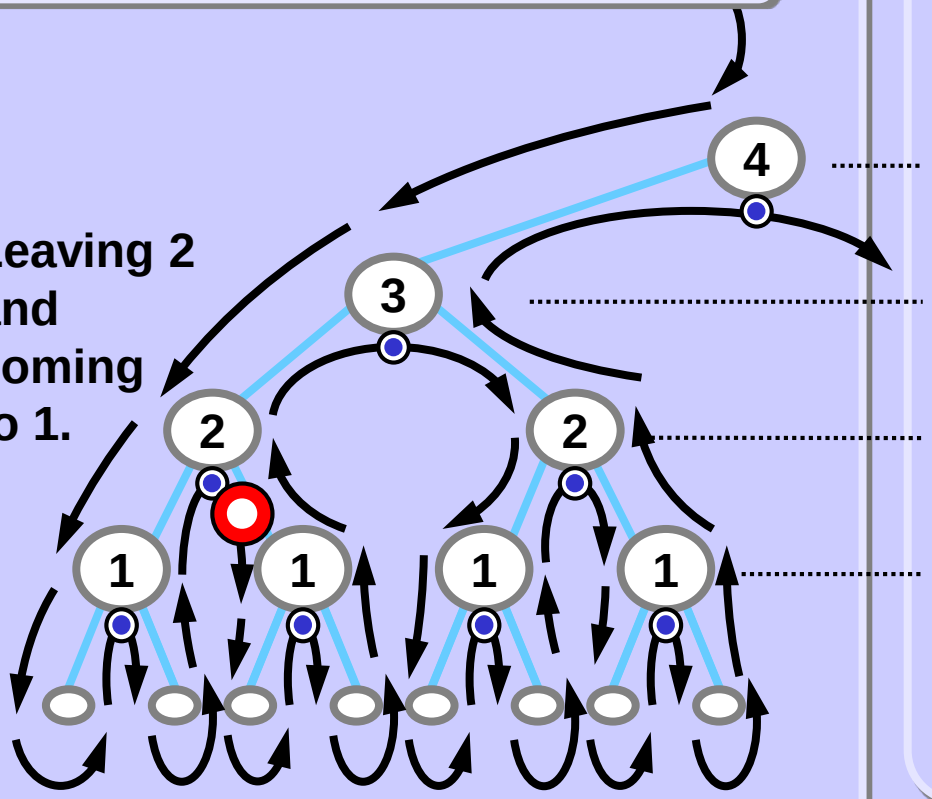
pop()

1

Output

Stack implements recursion

Recursion tree traversal



Stack

Value	Visits
4	1
3	1
2	2
1	0

push(1,0)

1 2

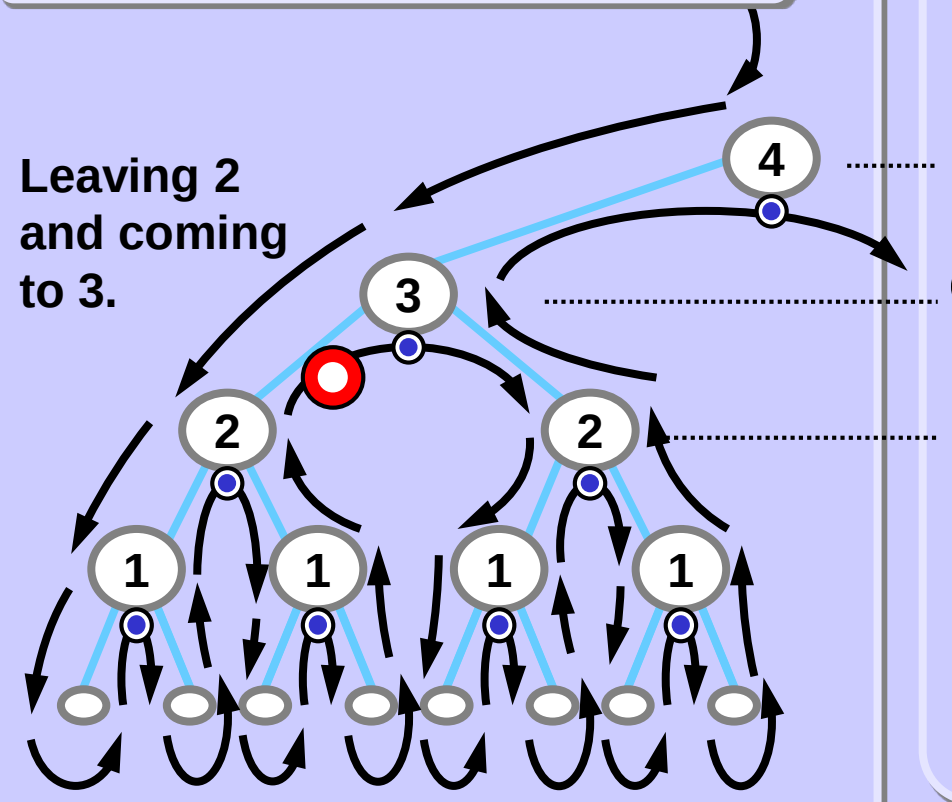
Output

etc…

Stack implements recursion

… after a while …

Recursion tree traversal



Stack

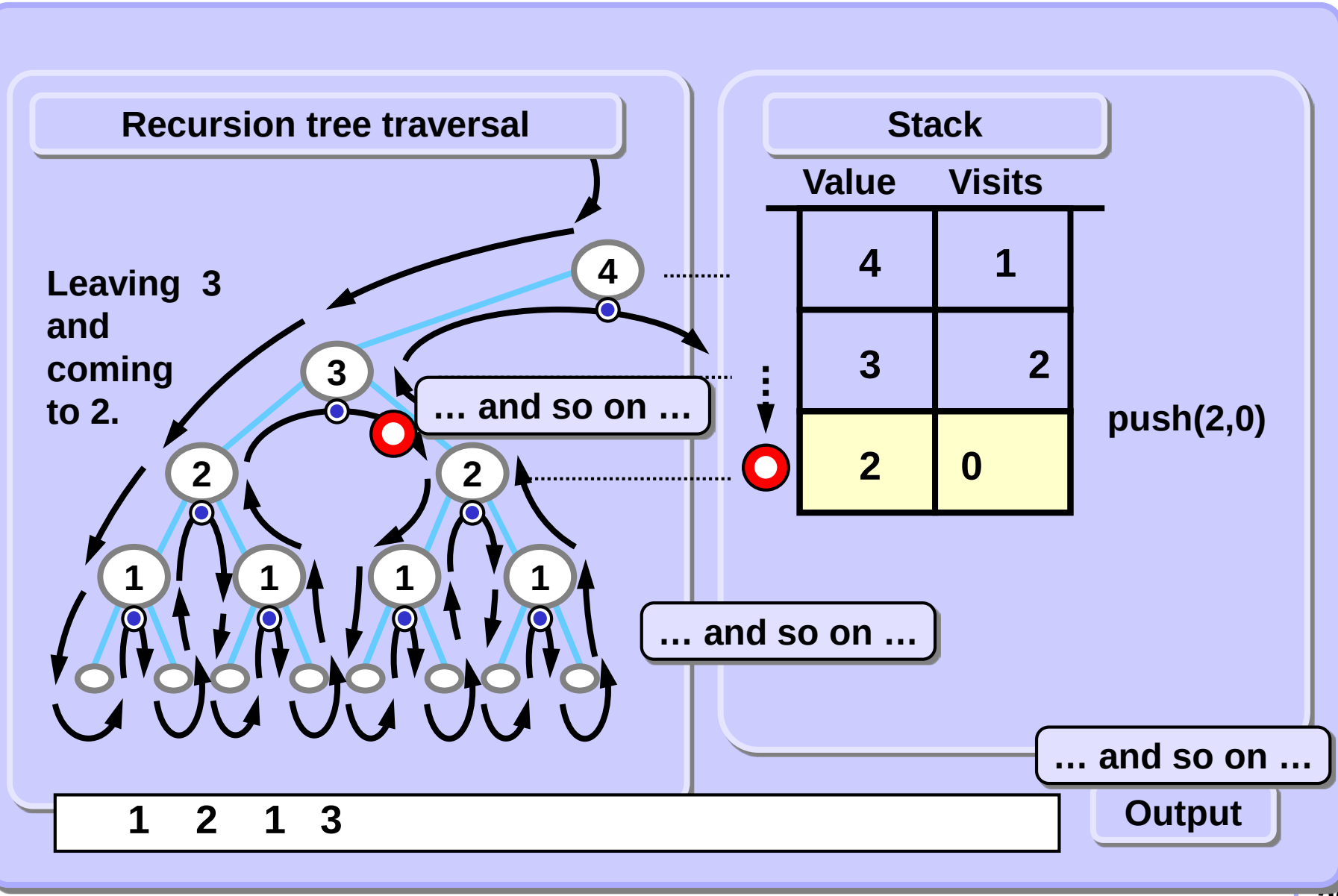
Value	Visits
4	1
3	1
2	2

pop()

1 2 1

Output

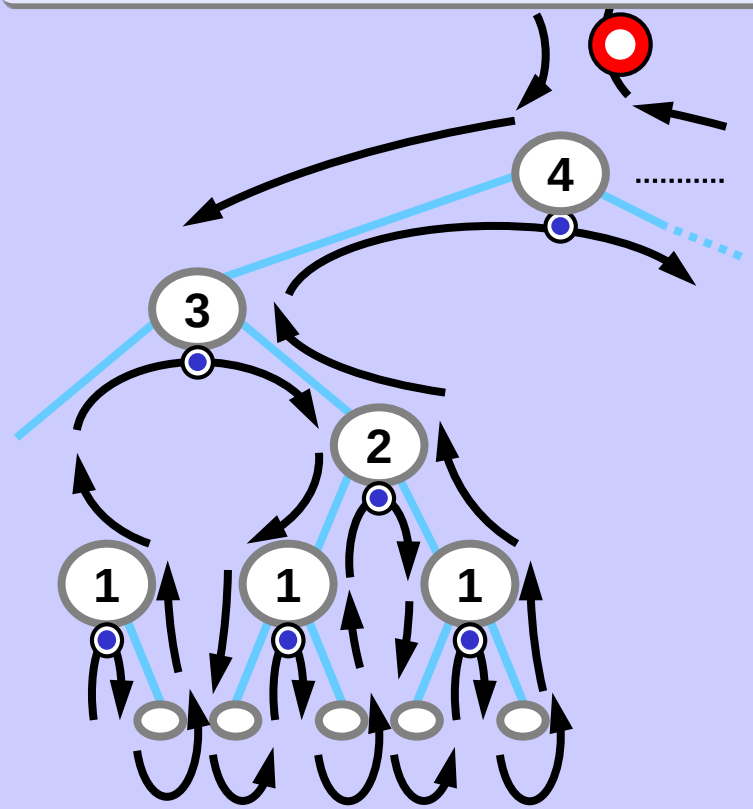
Stack implements recursion



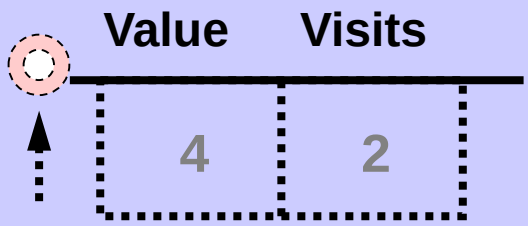
Stack implements recursion

… after another while … completed.

Recursion tree traversal



Stack



pop()

(empty == true)

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

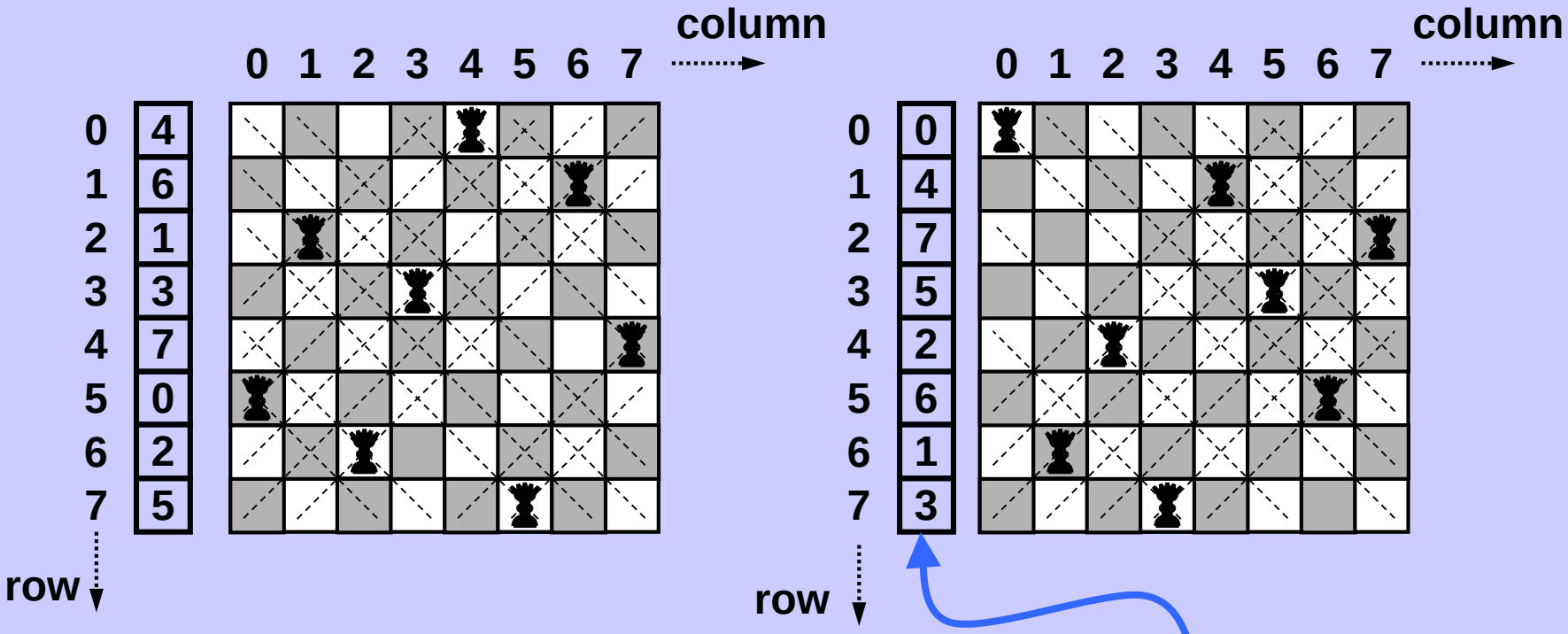
A little more compact code

```
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.

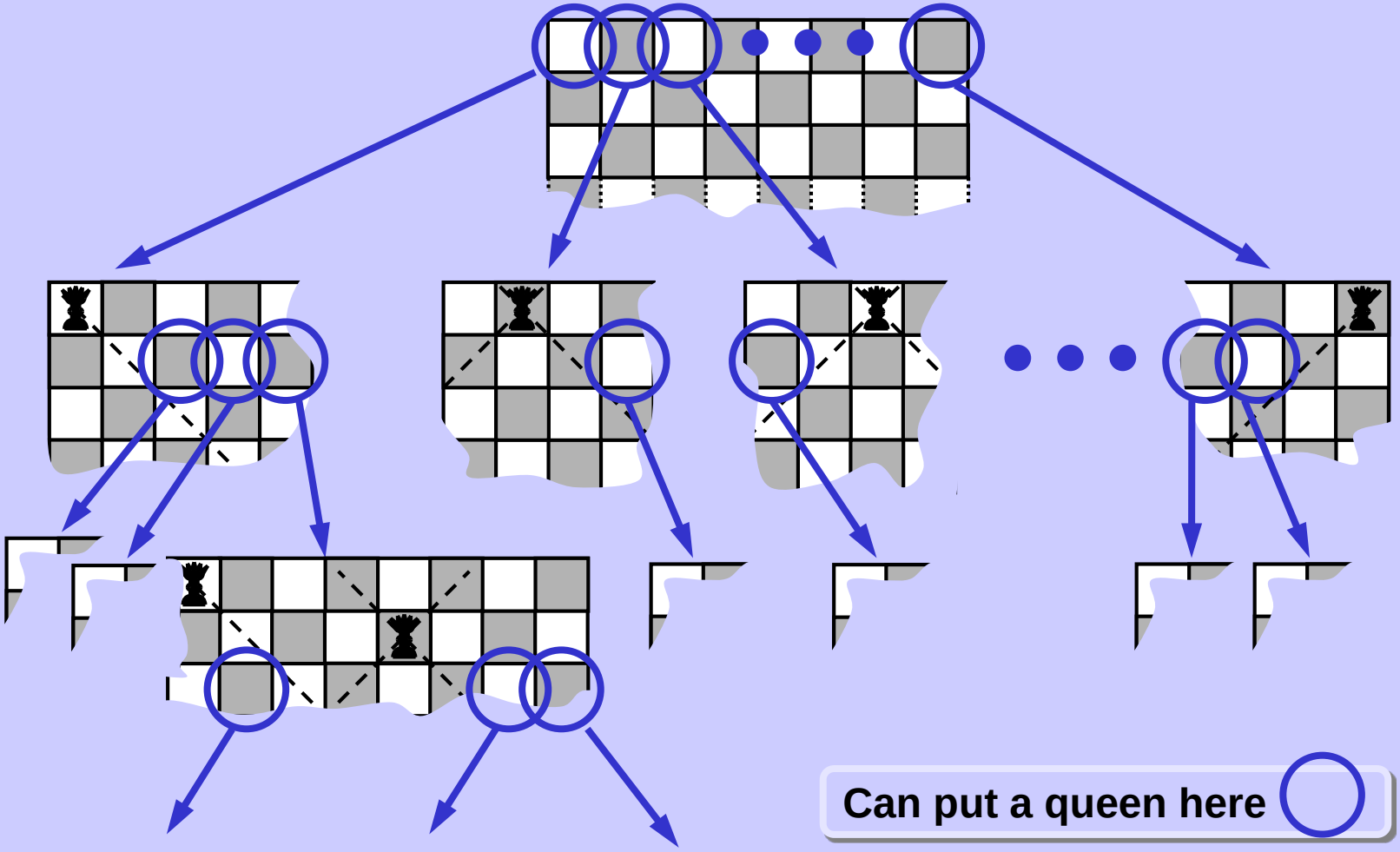
Some solutions



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

Easy backtrack problem 8 queens puzzle

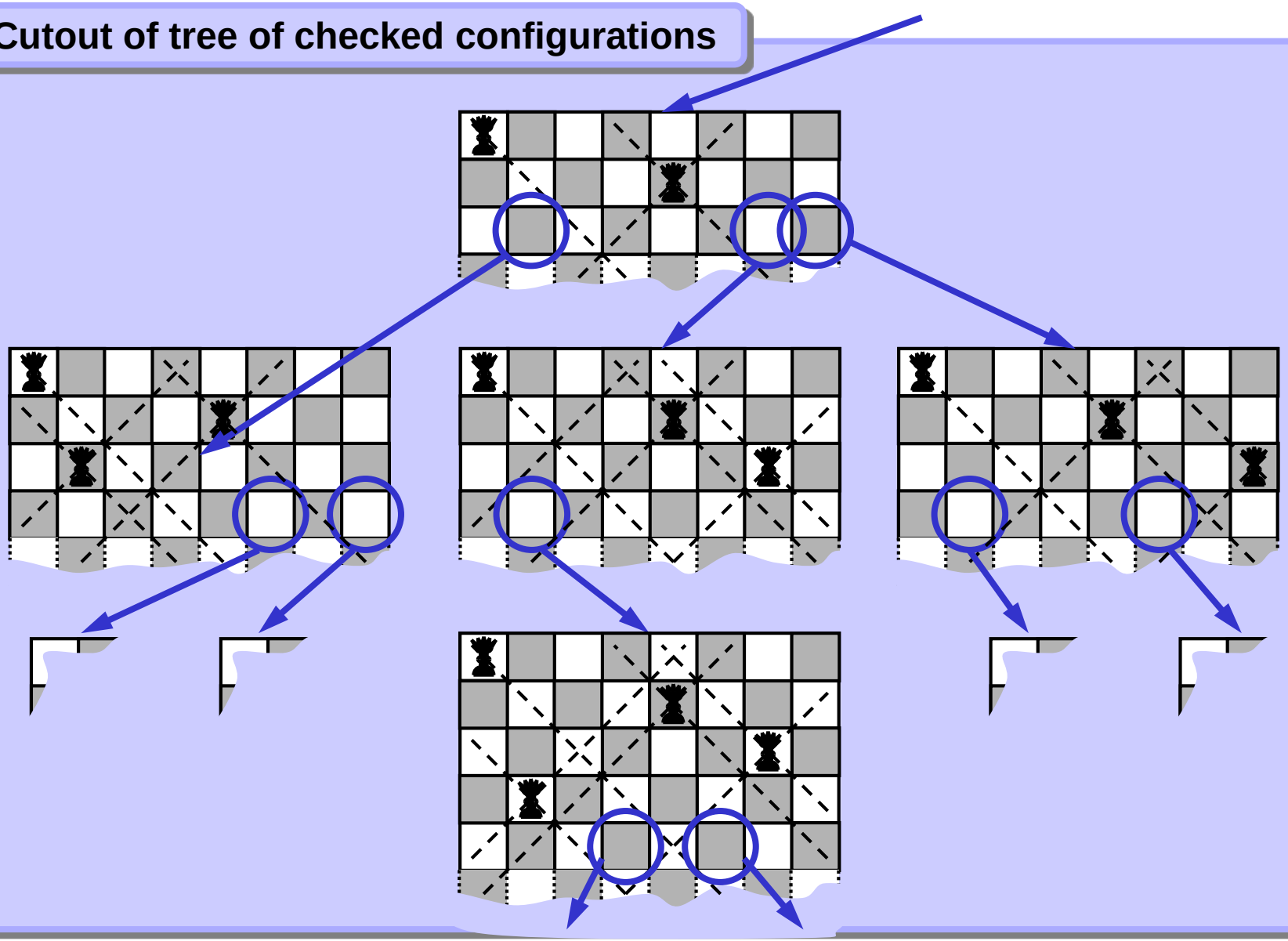
Tree of checked configurations (a root and a few successors)



Can put a queen here

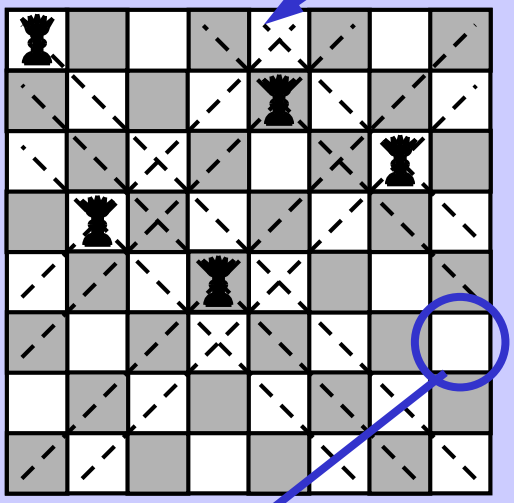
Easy backtrack problem 8 queens puzzle

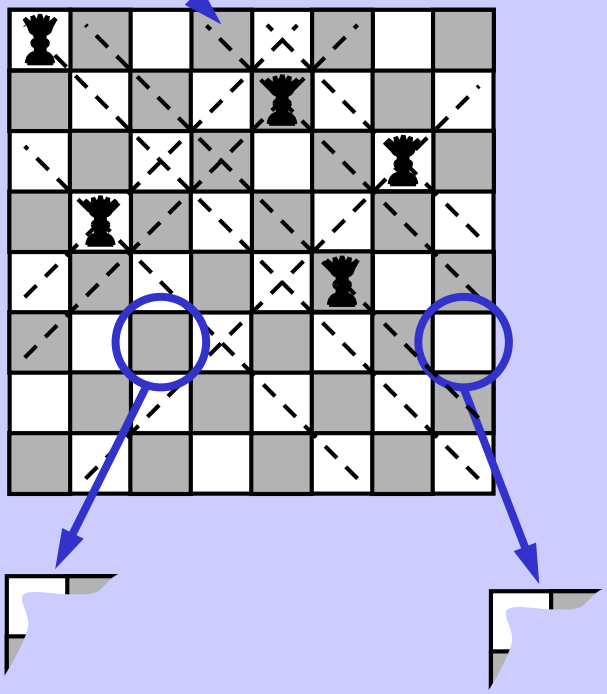
Cutout of tree of checked configurations

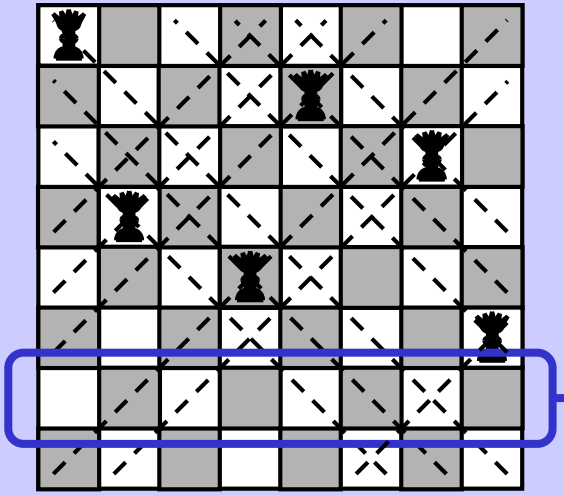


Easy backtrack problem 8 queens puzzle

Cutout of tree of checked configurations







Stop and backtrack

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 .
```

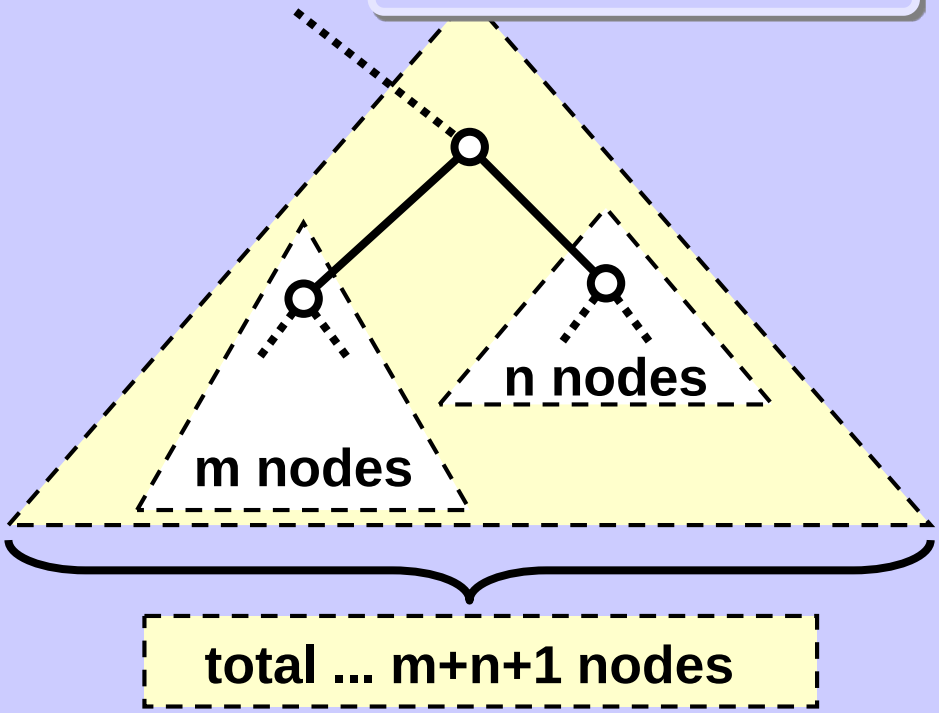
```
 . . Q . .
 . . . . Q
 . Q . . .
 . . . Q .
 Q . . . .
```

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

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

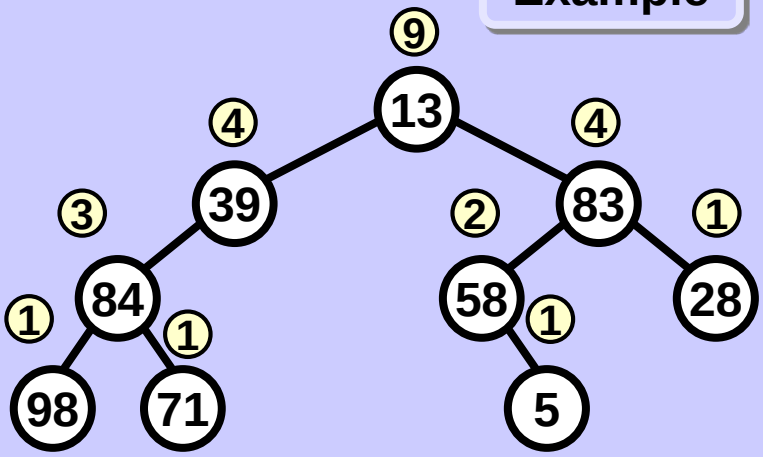
Tree size (= number of nodes)

A tree or a subtree



total ... m+n+1 nodes

Example



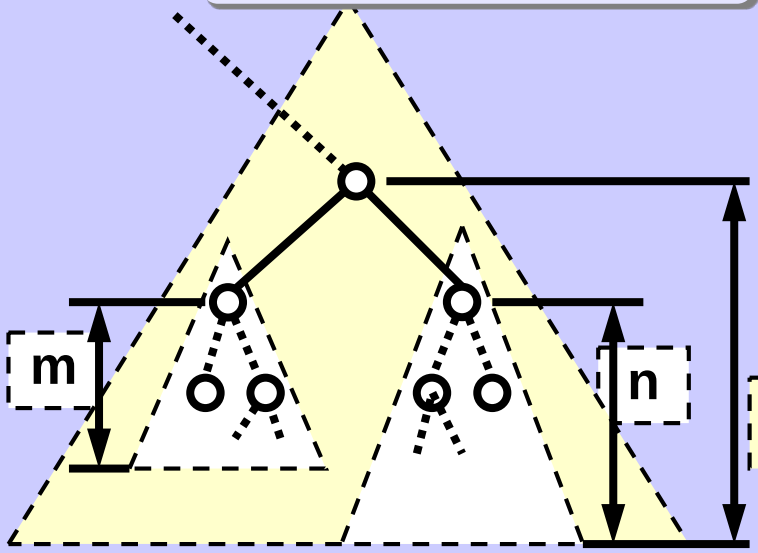
Exercise suggestion:
Implement with stack and no recursion.

Recursively

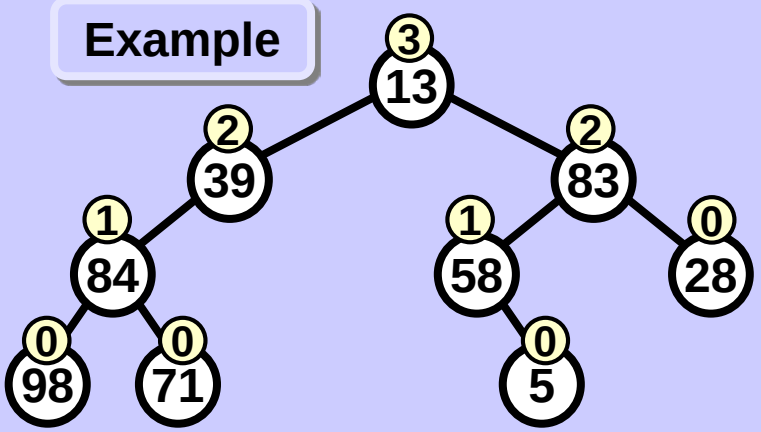
```
def count( self, node ):
  if node == None: return 0
  return 1 + self.count(node.left) + self.count(node.right)
```

Tree depth (= max depth of a node)

A tree or a subtee



Example



Exercise suggestion: Implement with stack and no recursion.

Recursively

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