Lecture 2: Trees, binary trees, recursion

BE5B33ALG — Algorithms

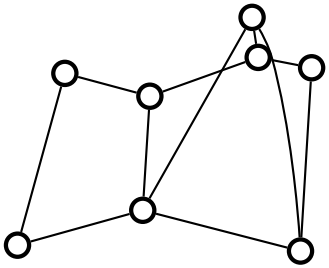
Ing. Robert Pěnička, Ph.D.
doc. RNDr. Daniel Průša, Ph.D.

Faculty of Electrical Engineering
Czech Technical University in Prague

Lecture 2: Trees, binary trees, recursion
Robert Pěnička, Daniel Průša
Trees
Binary Trees
Recursion
Simple backtracking examples

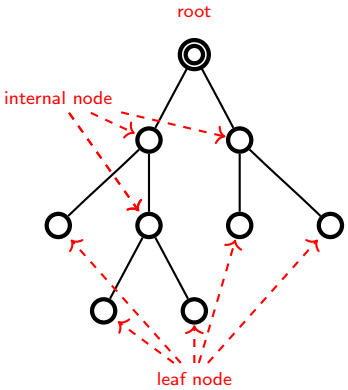
- Graph $G = (V, E)$
- V is a set of nodes (vertices)
- E is a set of edges (connections between nodes)
- Node (Vertex)

- Edge

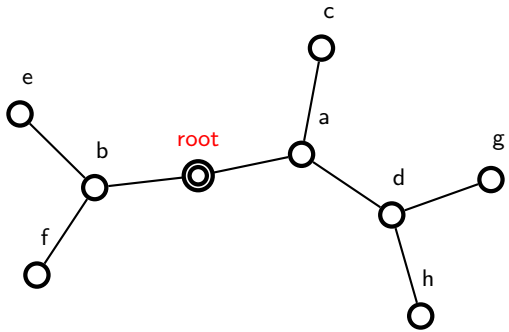


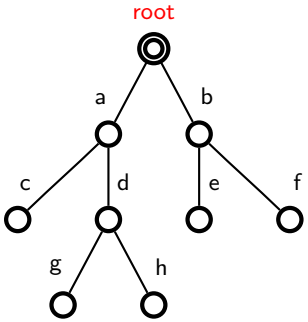
Tree

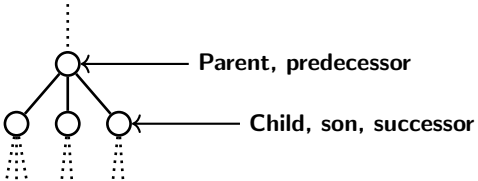
- Tree is a special type of graph
- **Connected** — there is a path between any two nodes.
- **Acyclic** — there are no cycles (loops).
- There is exactly one path path between any of its two nodes.
- Removing any edge disconnects the tree into two graphs.
- Number of edges is always less by one than the number of nodes.



Rooted Tree

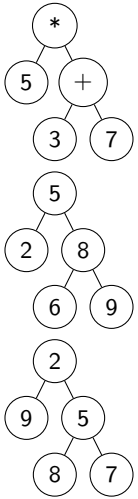

c
e
a
root
b
g
d
f
h


root
a
b
c
d
e
f
g
h

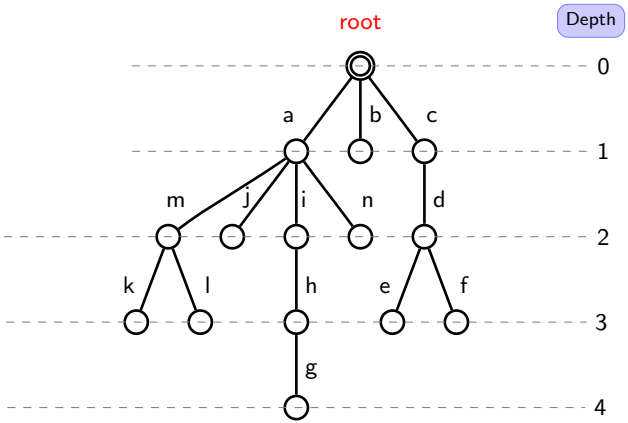

Parent, predecessor
Child, son, successor

What can be represented with a tree?

- Arithmetic expression
- Data structure (BST, heap)
- Decision tree (botanical key)
- Hierarchy of entities (employees, family tree)
- Recursive calls
- State space

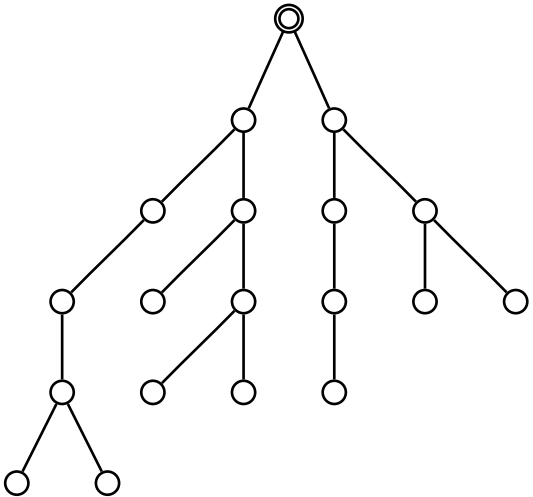


Tree Depth of rooted tree

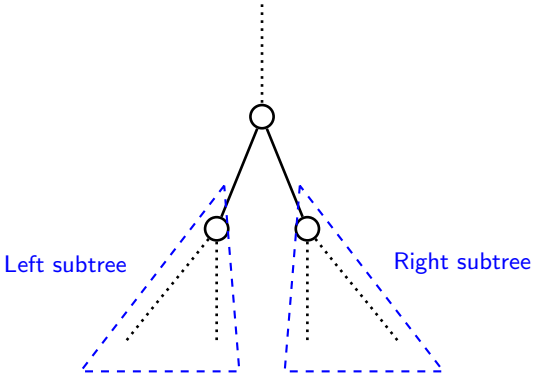


- Depth of a node is edge-length from root to node.
- Root depth is 0.
- Depth of empty tree is -1.
- Depth of a tree is maximal depth of any node.

Binary tree

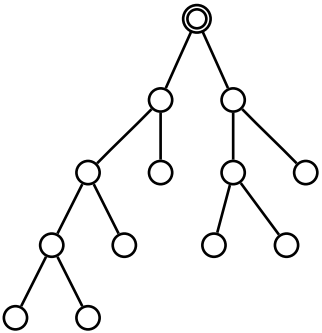


- Binary tree — each node has at most 2 children (0, 1 or 2).

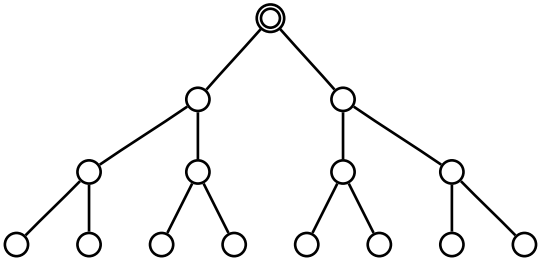


Lecture 2: Trees, binary trees, recursion
Robert Pěnička, Daniel Průša
Trees
Binary Trees
Recursion
Simple backtracking examples

- In regular binary tree, each node has 0 or 2 children. Not 1 child.

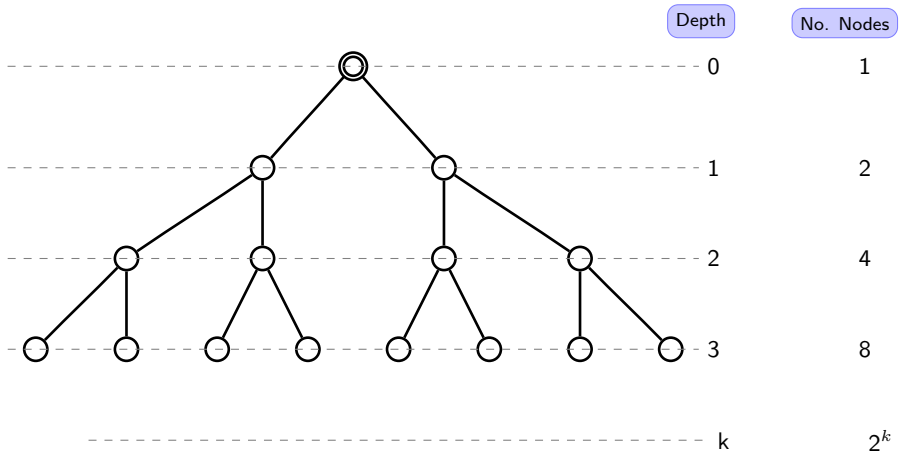


- In balanced tree the depths of all leaves are (approximately) the same.



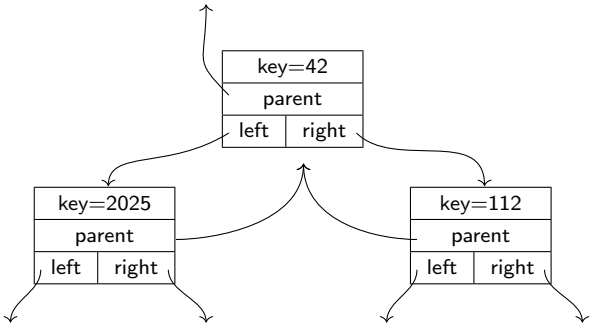
Depth of a balanced regular binary tree

- Number of nodes is $2^{depth+1} - 1$.
- Depth is $\log_2(|nodes| + 1) - 1 \approx \log_2(|nodes|)$



Tree node structure

- Each node contains a key, reference to its left and right children, and reference to its parent.
- Parent might not be neccessary in some implementations (depends how the tree is traversed).




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

Random tree generation

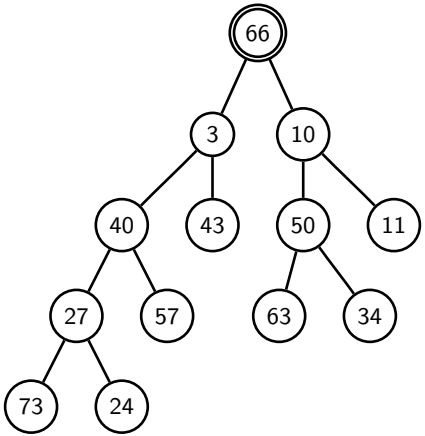
- Reference to the parent is not needed for now (no backward traversal).
- The randrange(n) returns a pseudorandom integer in the range from 0 to n-1.

```
@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
```

Lecture 2: Trees, binary trees, recursion
Robert Pěnička, Daniel Průša
Trees
Binary Trees
Recursion
Simple backtracking examples

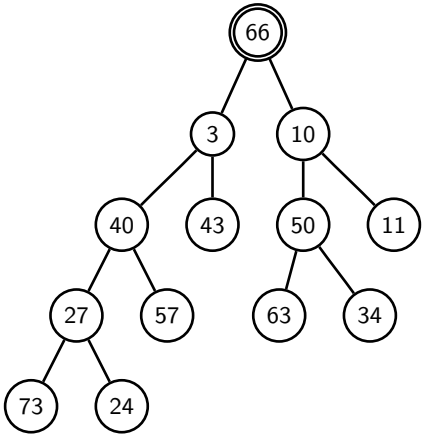
Example call

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



Inorder binary tree traversal


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



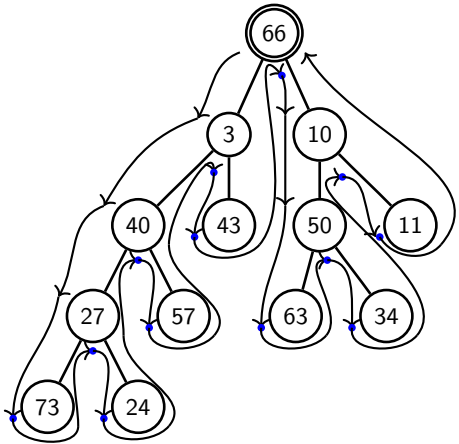
Output: 73 27 24 40 57 3 43 66 63 50 34 10 11

Inorder binary tree traversal

Print point: •

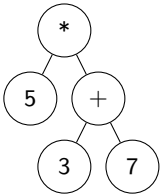
Movement direction: ⟶

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



Output: 73 27 24 40 57 3 43 66 63 50 34 10 11

Use of traversals



The tree represents the expression:

$$5 * (3 + 7)$$

- **Inorder:** $5 * 3 + 7$

  the represented expression cannot be uniquely determined

- **Preorder:** $* \; 5 \; + \; 3 \; 7$

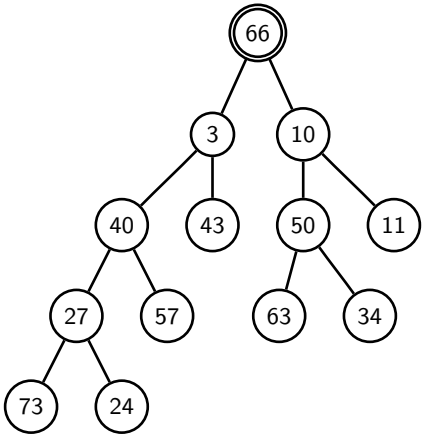
  Lisp

- **Postorder:** $5 \; 3 \; 7 \; + \; *$

  reverse Polish notation, stack calculator

Preorder binary tree traversal


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



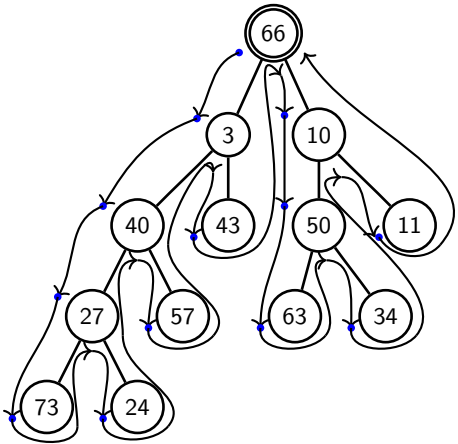
Output: 66 3 40 27 73 24 57 43 10 50 63 34 11

Preorder binary tree traversal

Print point: •

Movement direction: ⟶

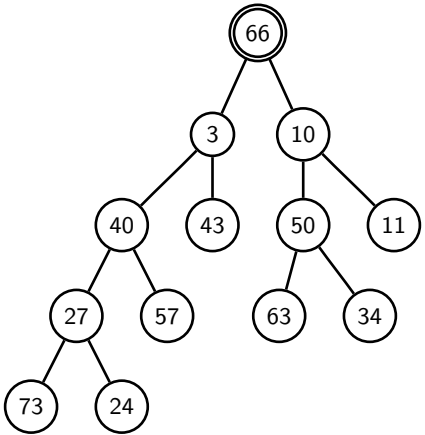
```
def listPreOrder( self, node ):
    if node == None:
        return
    print( node.key, end = " " )
    self.listPreOrder( node.left )
    self.listPreOrder( node.right )
```



Output: 66 3 40 27 73 24 57 43 10 50 63 34 11

Postorder binary tree traversal


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



Output: 73 24 27 57 40 43 3 63 34 50 11 10 66

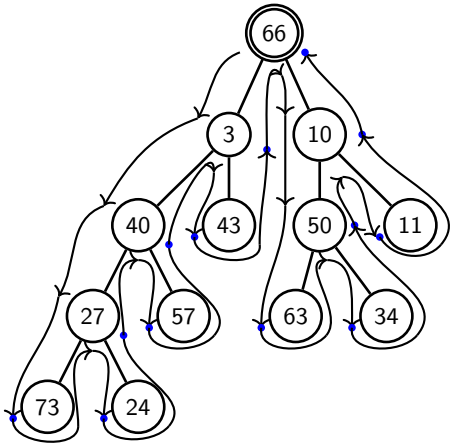
Postorder binary tree traversal

Print point: •

Movement direction: ⟶

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

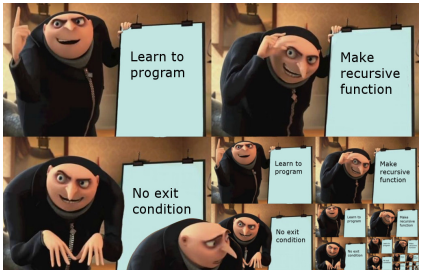
•



Output: 73 24 27 57 40 43 3 63 34 50 11 10 66

Lecture 2: Trees, binary trees, recursion
Robert Pěnička, Daniel Průša
Trees
Binary Trees
Recursion
Simple backtracking examples

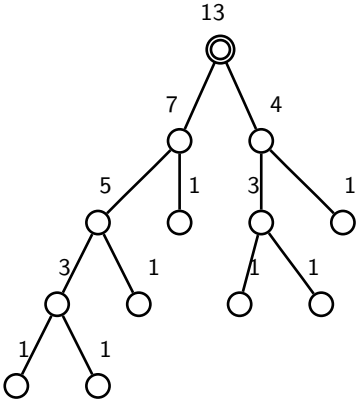
- Recursion is when a function calls itself in order to solve a problem.
- Instead of solving the whole problem at once, we can break it down into smaller subproblems that are solved recursively.
- It always has to have two key parts:
  - Base case — a simple stopping condition so it does not call itself forever.
  - Recursive step — the function calling itself with a smaller or simpler input.



Source: rabbithole of Reddit u/SingleWalnut

Recursion examples: Tree size (= number of nodes) recursively

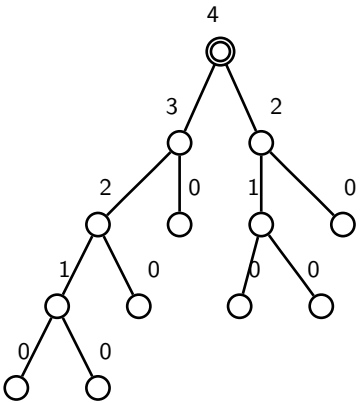
- We want to count number of nodes in a tree.
- Base case — if the tree is empty, the count is 0.
- Recursive step — the count is 1 (increment for the current node) plus the count of nodes in the left and right subtrees.




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

Recursion examples: Tree depth recursively

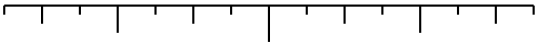
- We want to find the maximum depth of a tree.
- Base case — if the tree is empty, the depth is -1.
- Recursive step — the depth is 1 (increment for the current depth) plus the maximum depth of the left and right subtrees.




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

Recursion examples: Marking a ruler

- We want to mark the positions of a ruler.
- Base case — if the ruler length is 0, there is nothing to mark.
- Recursive step — mark the current position, then recursively mark the rest.



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

Call ruler(4) Output: 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

Recursion examples: Marking a ruler vs Inorder tree traversal

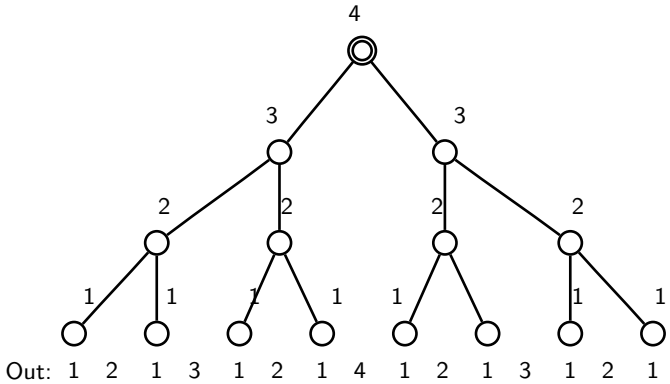
- Ruler and inorder traversal of a tree are structurally identical.

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

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

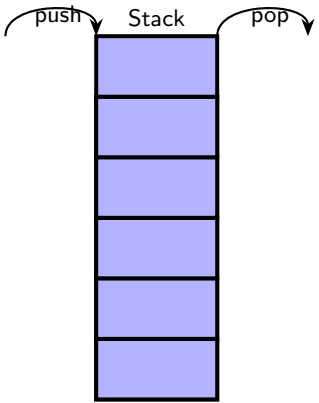
Recursion examples: Marking a ruler call


```
if n == 0: return
  ruler(n - 1)
  print(n)
  ruler(n - 1)
```

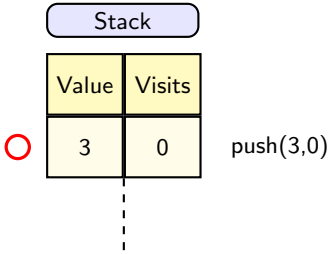


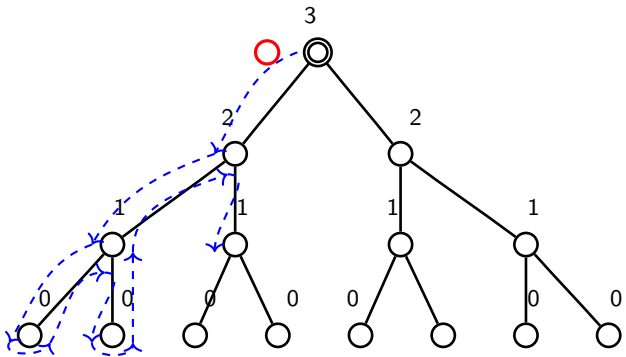
Stack

- Stack is a linear data structure.
- It follows the **LIFO (Last In, First Out)** principle.
- Most basic operations are push and pop.
- 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.

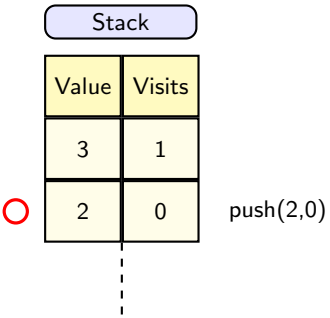


Stack implements recursion

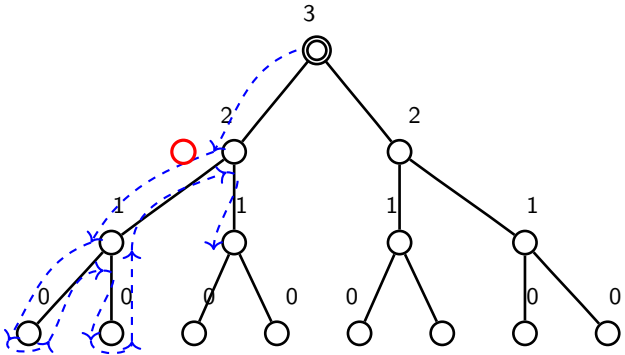




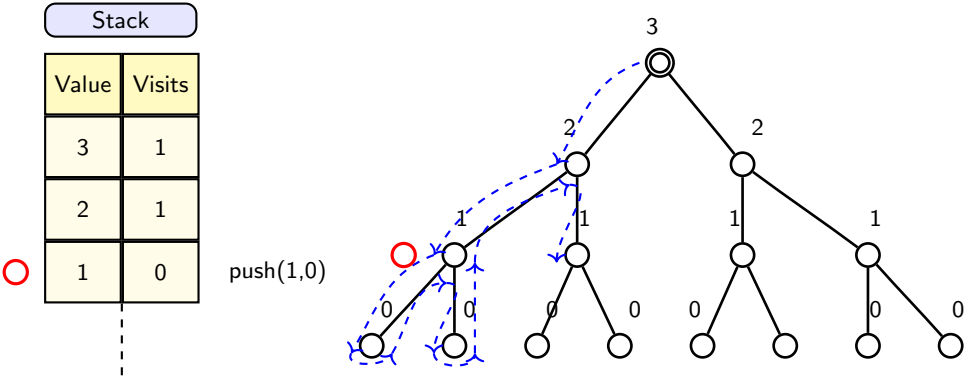
Stack implements recursion



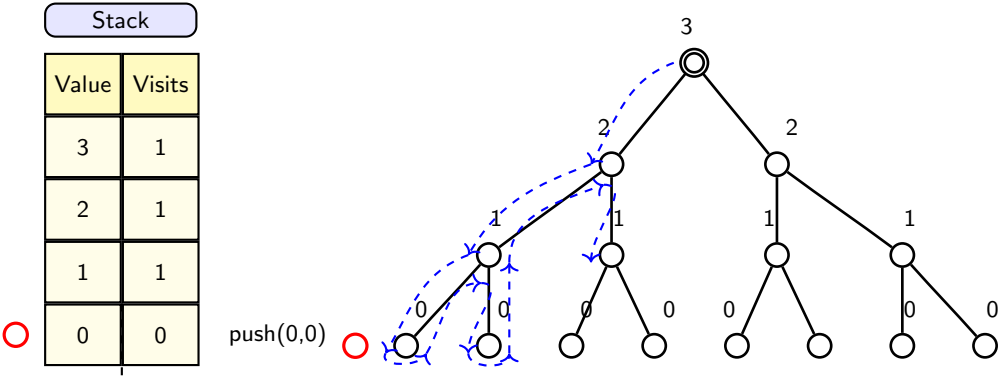
push(2,0)



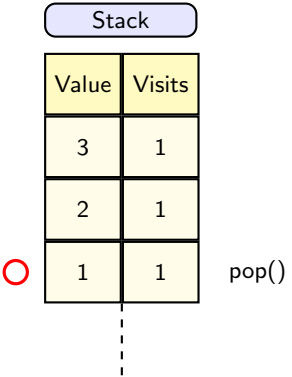
Stack implements recursion

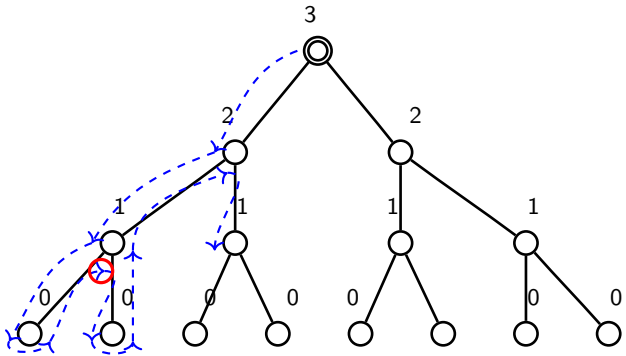


Stack implements recursion

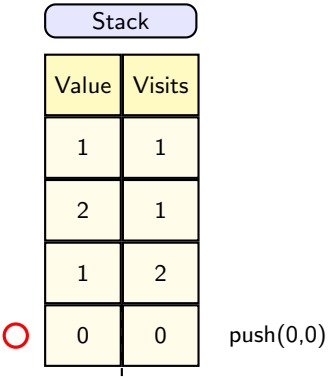


Stack implements recursion

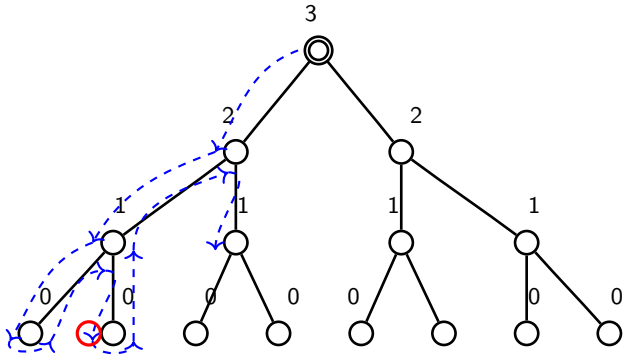




Stack implements recursion

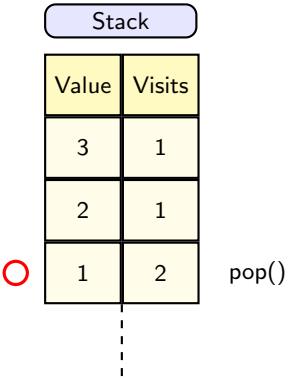


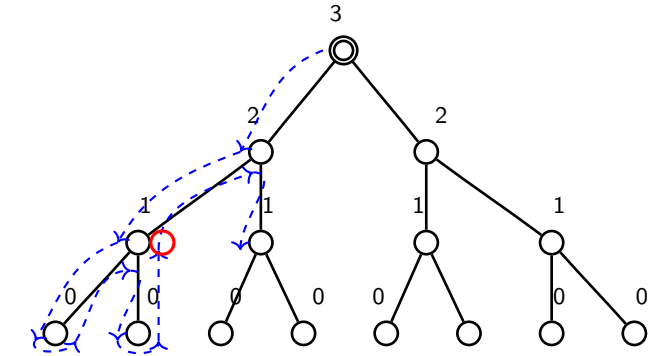
push(0,0)



Output: 1

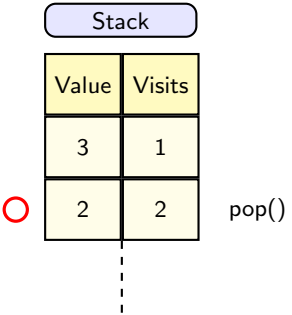
Stack implements recursion

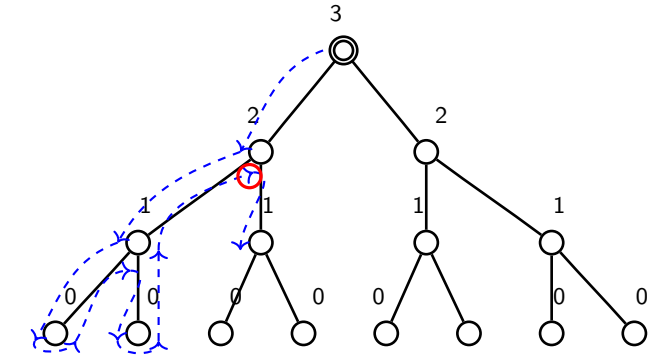




Output: 1

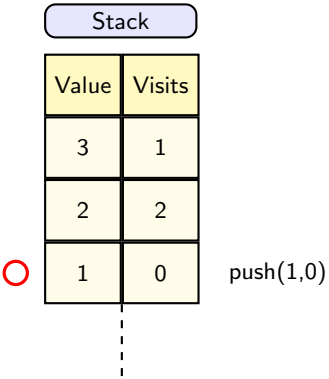
Stack implements recursion



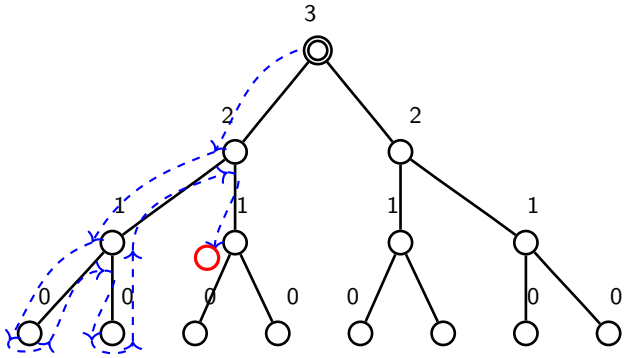


Output: 1

Stack implements recursion



push(1,0)



Output: 1 2

Ruler without recursive calls


```
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()
```

Backtracking

- **Backtracking** is a general algorithmic technique for solving problems incrementally.
- It builds candidates for solutions step by step, and **abandons (backtracks)** as soon as it determines that the partial candidate cannot lead to a valid solution.
- Often used in:
  - Combinatorial search problems (e.g., subset sum, knapsack, N-Queens).
  - Constraint satisfaction (e.g., Sudoku, puzzles).

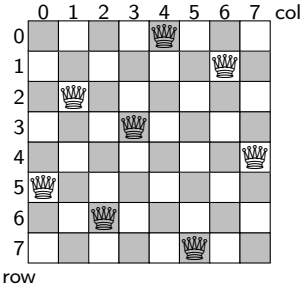
Key Idea

- Explore all possibilities, but prune paths early when they cannot succeed.
- Traverses this search tree recursively, from the root down, in depth-first order

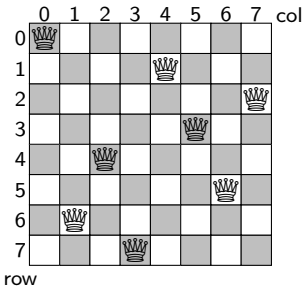
Easy backtrack problem: 8 Queens Puzzle

Problem

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



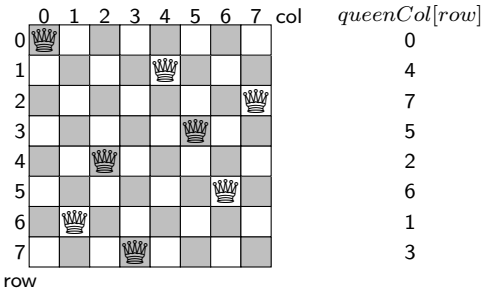
Example solution with queen positions.



Another valid solution.

Easy backtrack problem: 8 Queens Puzzle

- The single data structure `queenCol[row]` stores the column index of the queen placed in each row.
- Row conflicts are impossible since we place only one queen per row.



Lecture 2: Trees, binary trees, recursion
Robert Pěnička, Daniel Průša
Trees
Binary Trees
Recursion
Simple backtracking examples

N queens puzzle (N $\times$ 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**

Checking if a Queen's Position is Safe

Idea: When placing a queen at position (r, c):

- It must not share the same column with any previously placed queen.
- It must not share the same diagonal with any previously placed queen.
- We can check the $queenCol[r]$ array for all earlier rows to verify this.

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

Key Observation

- Row conflicts are impossible since we place only one queen per row.
- A queen is safe if no previously placed queen shares its **column** or **diagonal**.

Placing the queens

Placing qeen at position (row, col) and recursively placing queens in the next row:

```
def putQueen(row, col):
    queenCol[row] = col;
    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
```

Running the whole algorithm:

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