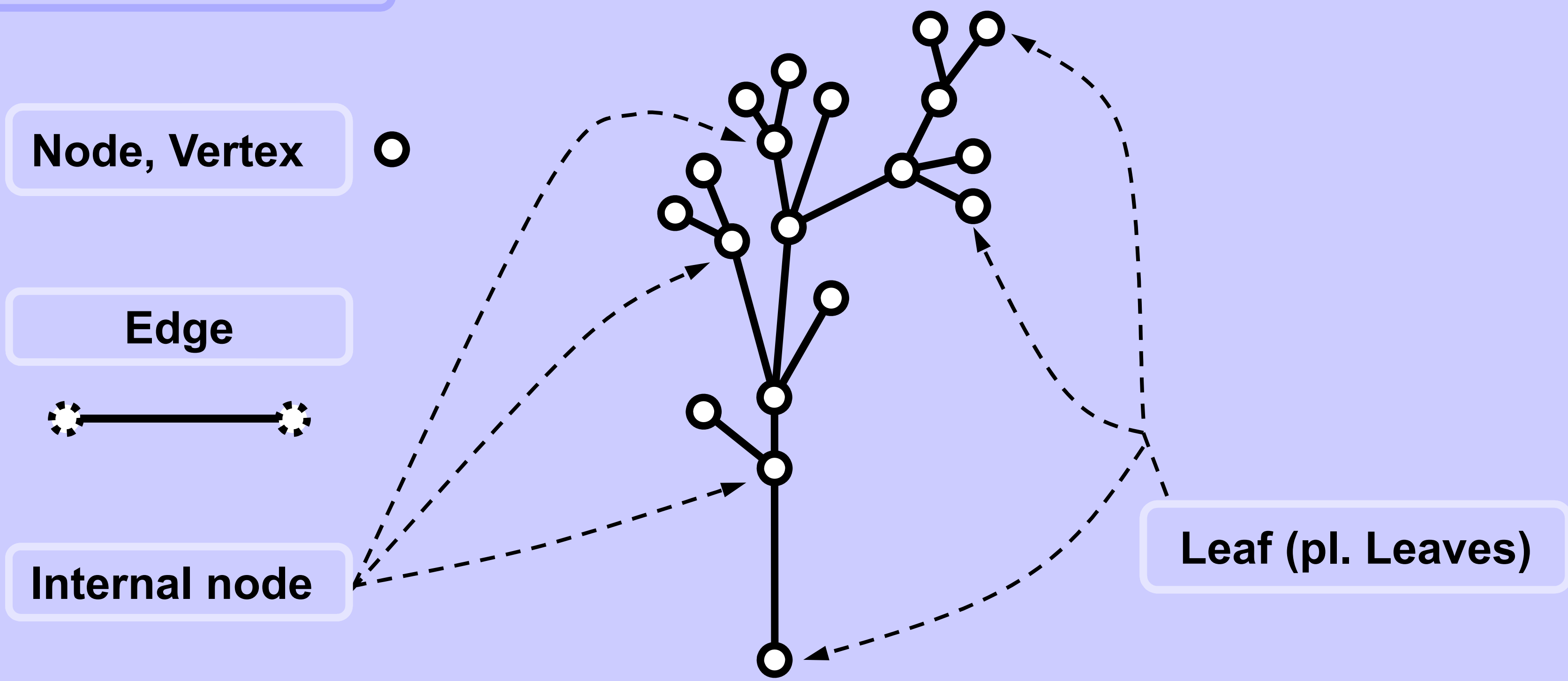
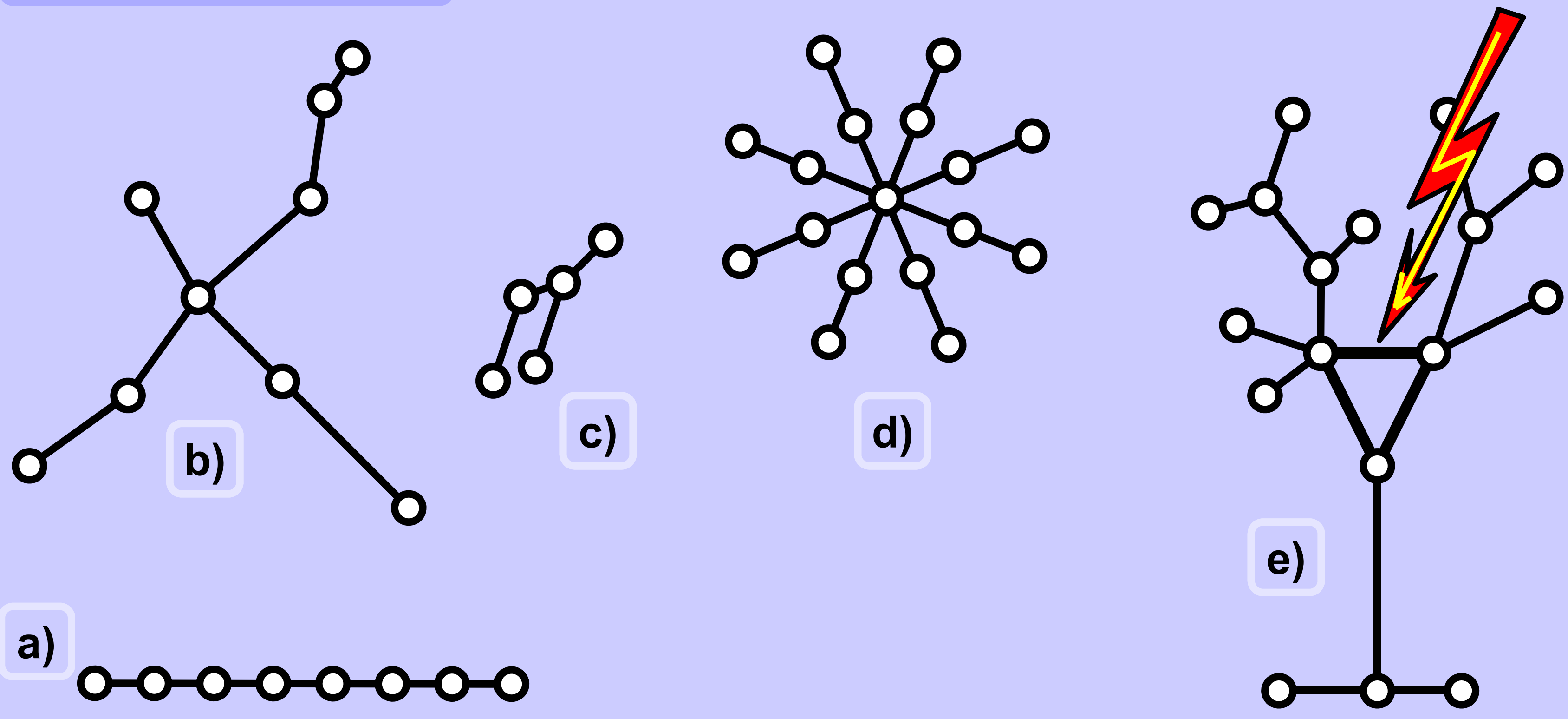
# TREES, BINARY TREES
# REALTION BETWEEN TREES AND RECURSION
# USING STACK TO IMPLEMENT RECURSION
# BACKTRACKING

## Tree



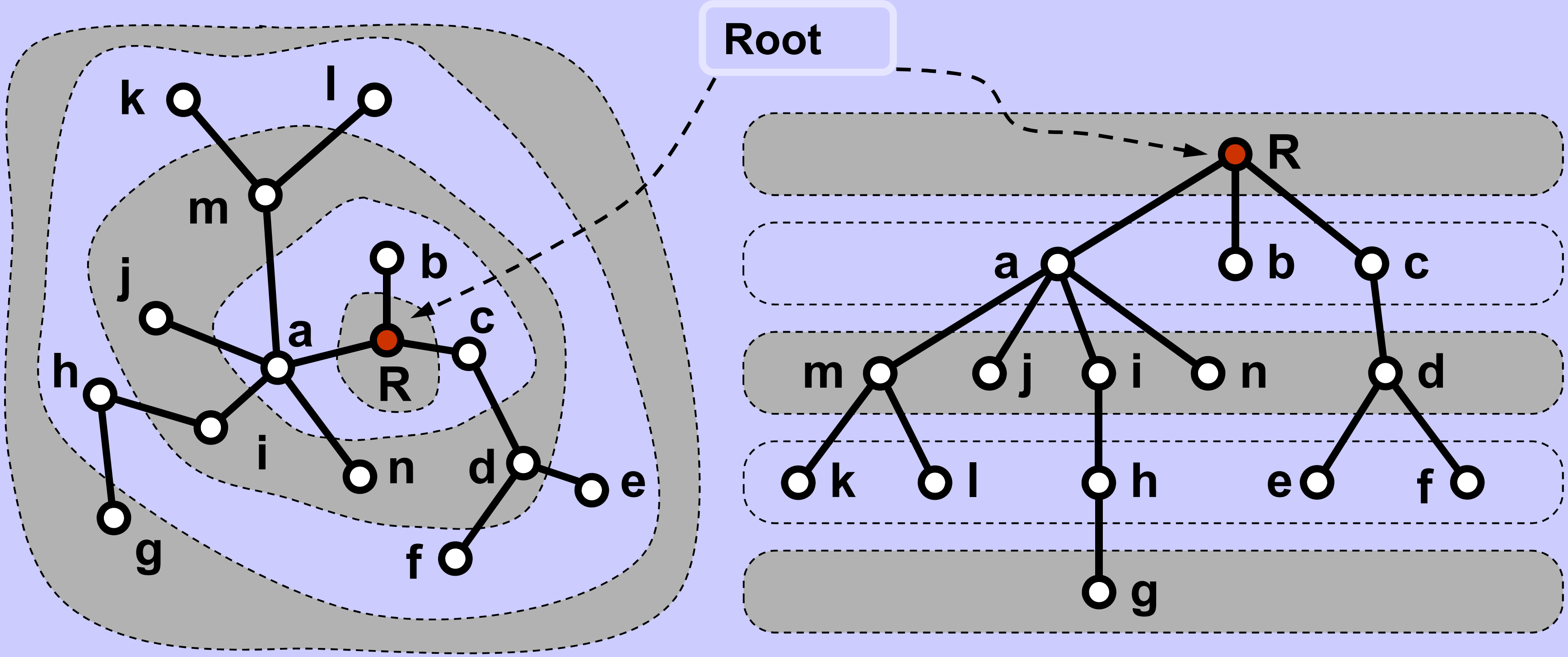
## Tree examples



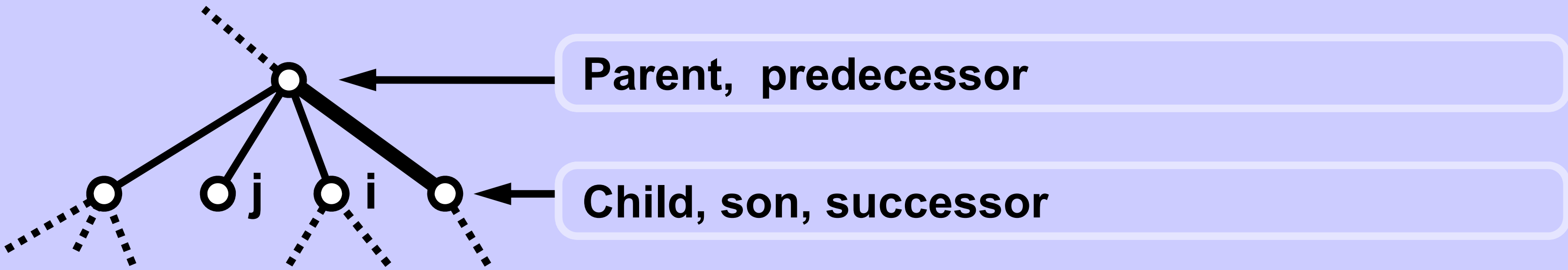

## Tree properties

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.

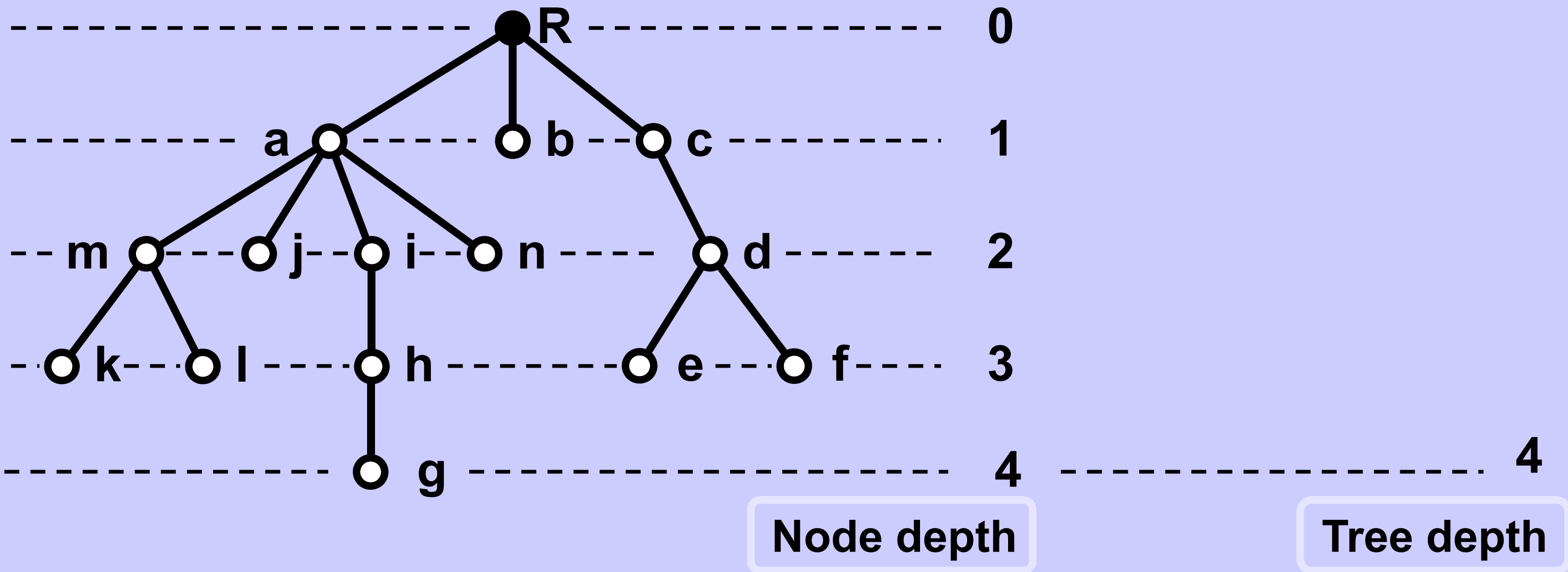
## Rooted tree



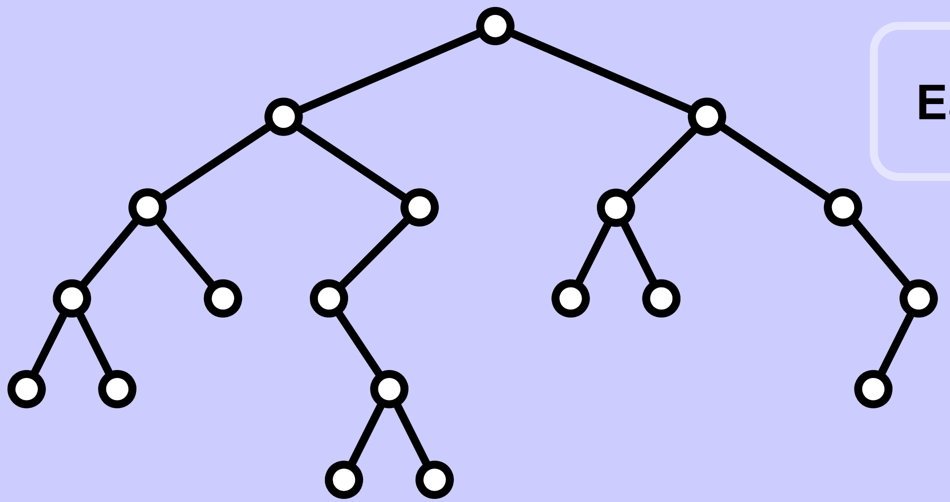

## Terminology



## Tree depth

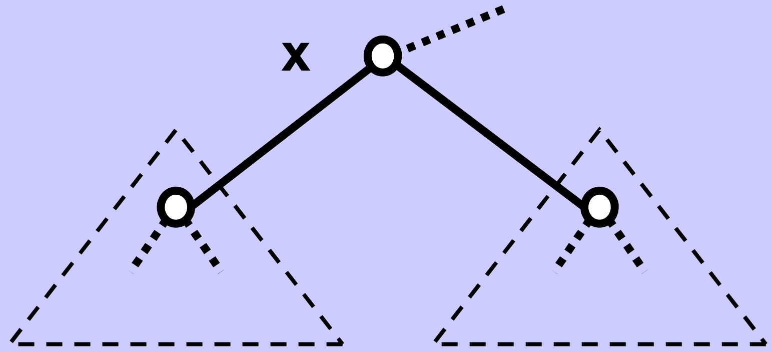


## Binary (rooted!!) tree



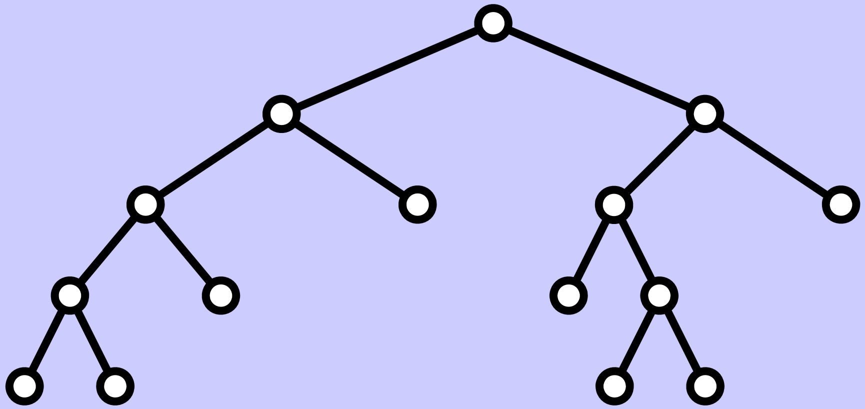
Each node has 0 or 1 or 2 children.

## Left and right subtree



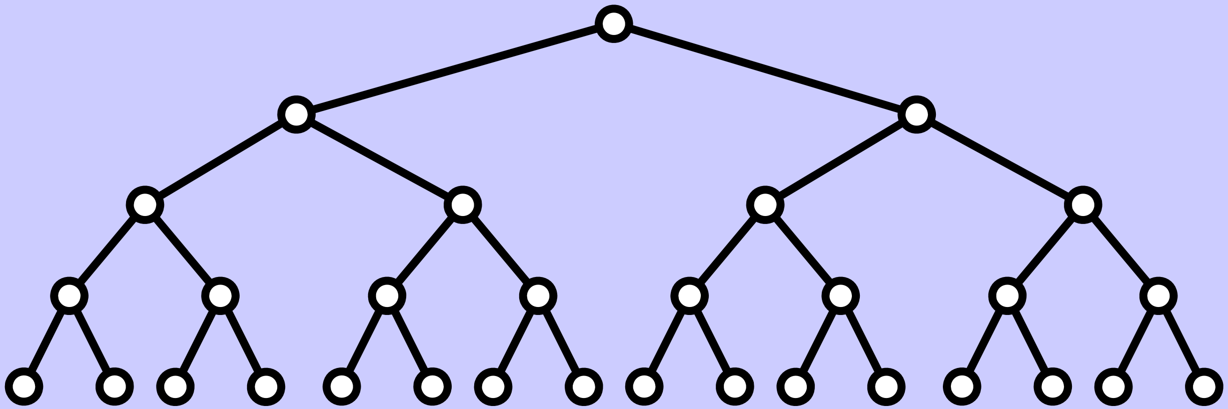

Subtree of node x ................ left ................ right

## Regular binary tree



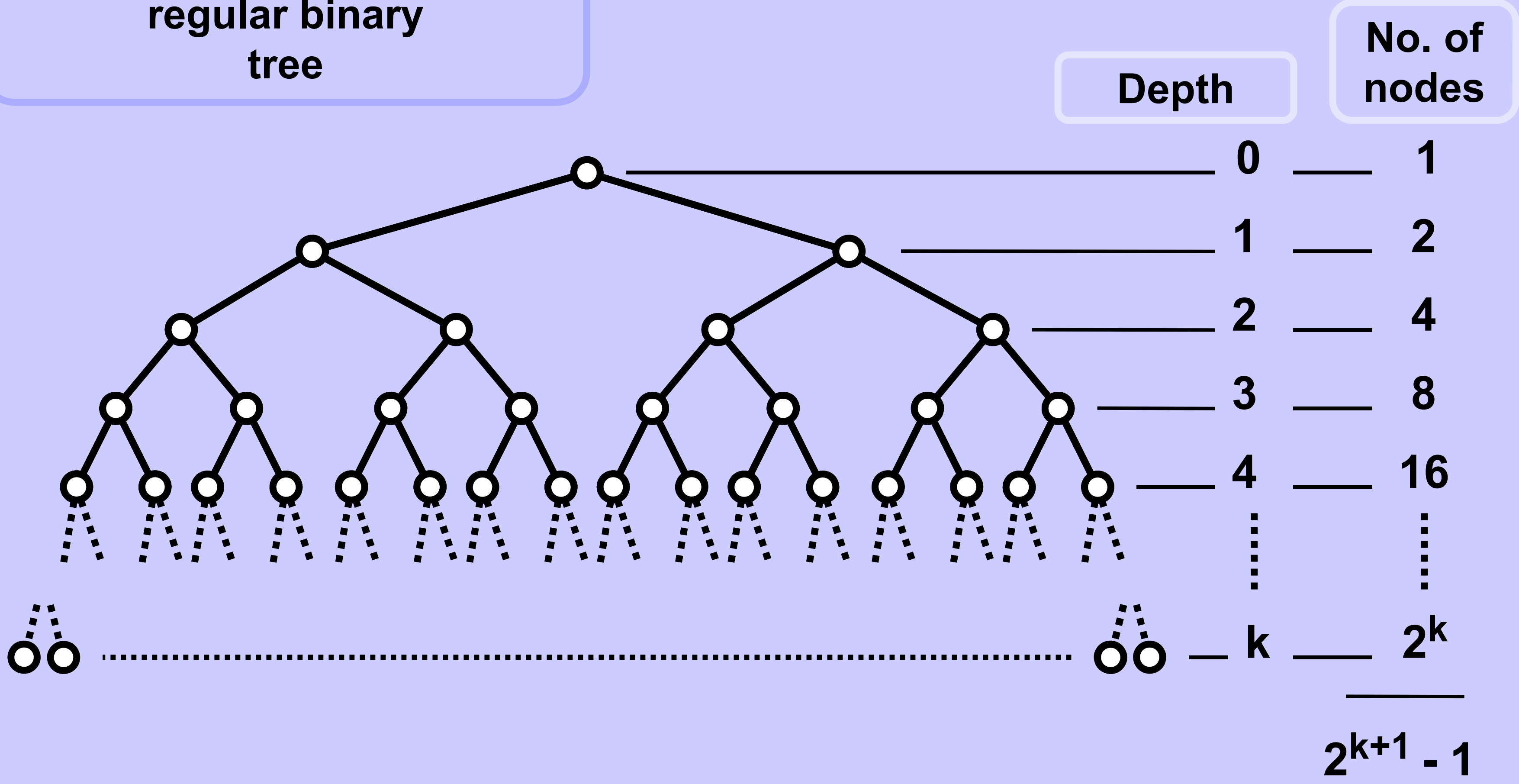
Each node has 0 or 2 children.
Not 1 child

## Balanced tree



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

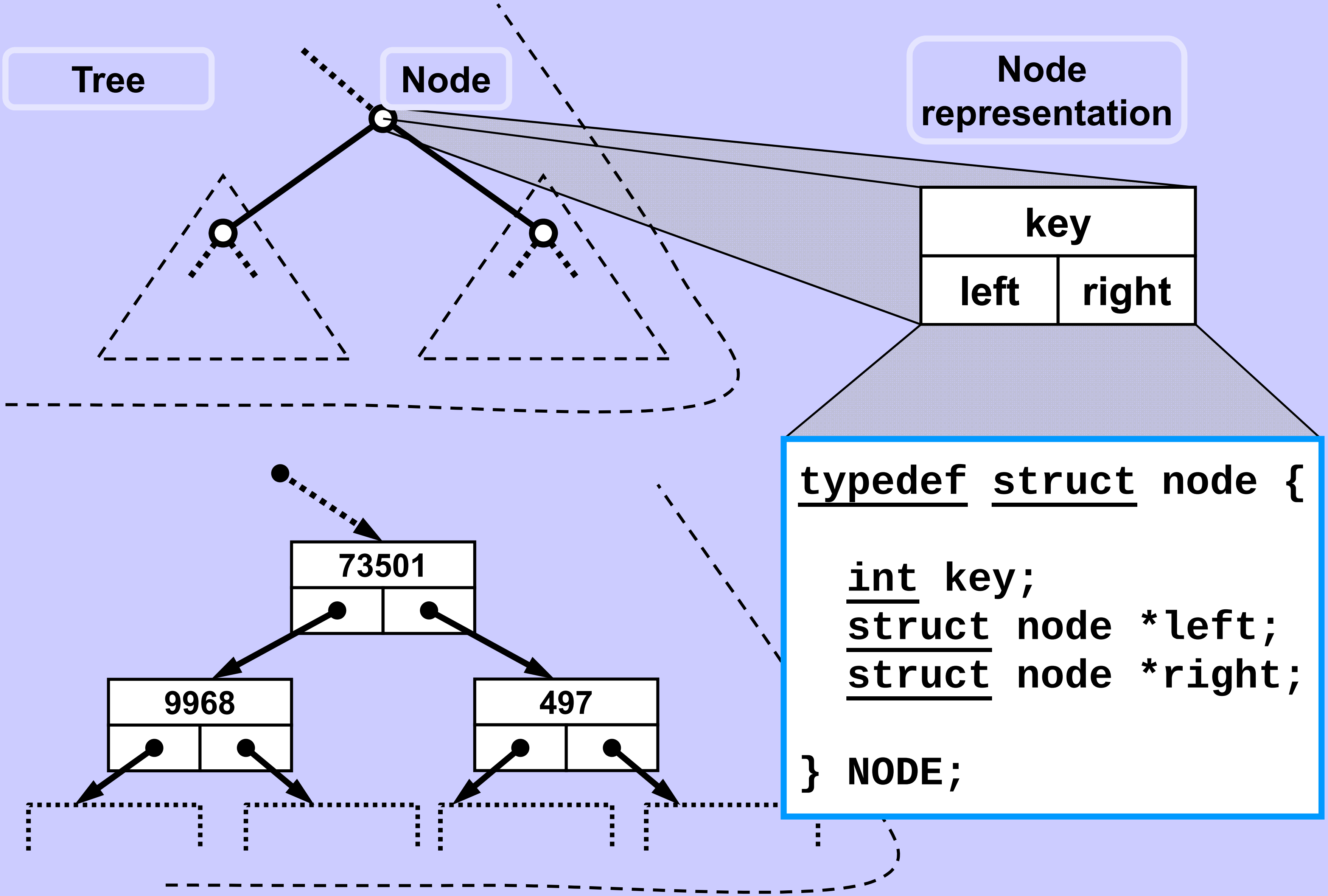
# Depth of a balanced regular binary tree




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

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

# Binary tree implementation -- C

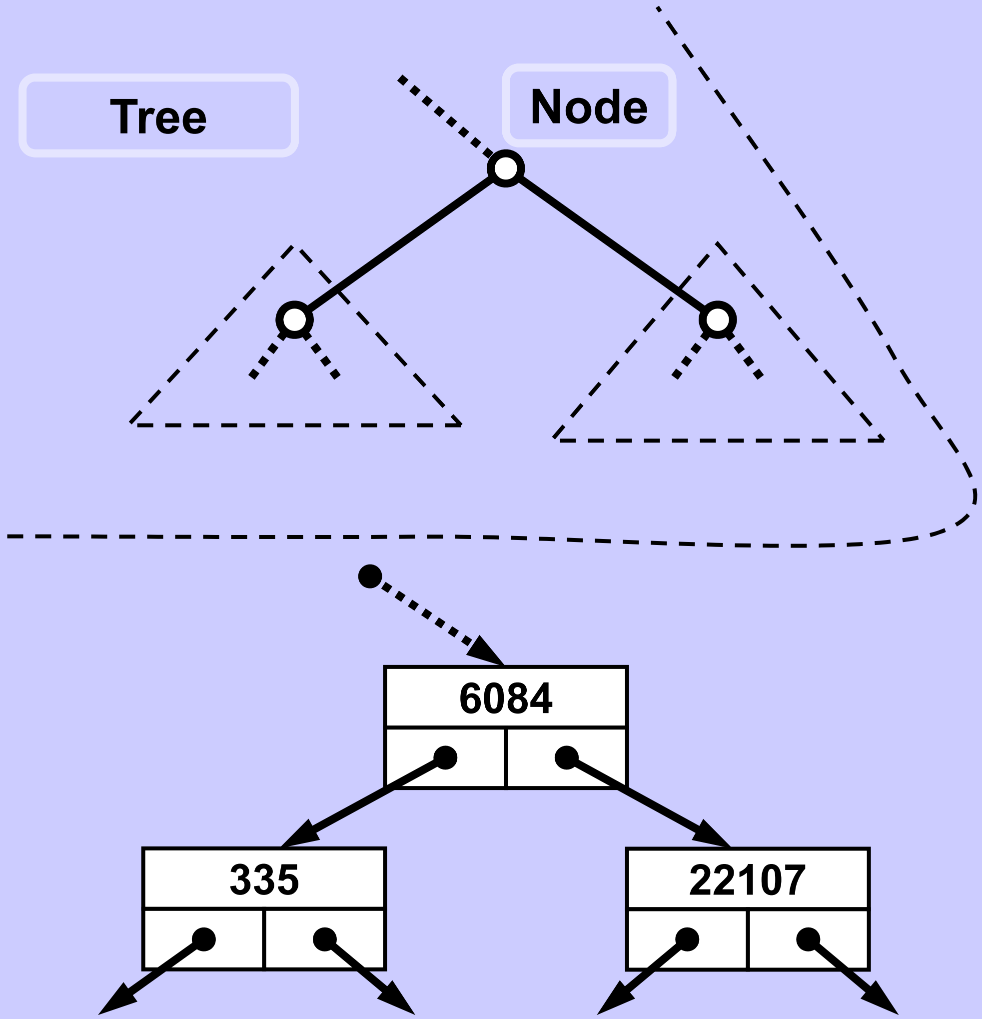



```c
typedef struct node {

   int key;
   struct node *left;
   struct node *right;

} NODE;
```

# Binary tree implementation -- Java




```
public class Node {
   public Node left;
   public Node right;
   public int key;
   public Node(int k) {
     key = k;
     left = null;
     right = null;
   }
}

public class Tree {
   public Node root;
   public Tree() {
     root = null;
   }
}
```

## Build a random binary tree -- C

```
NODE *randTree(int depth) {
   NODE *pnode;
   if ((depth <= 0) || (random(10) > 7))
      return  (NULL);                                //stop recursion
   pnode = (NODE *) malloc(sizeof(NODE)); // create node
   if (pnode == NULL) {
      printf("%s", "No memory.");
      return NULL;
      }
  pnode->left = randTree(depth-1);     // make left subtree
  pnode->key = random(100);            // some value
  pnode->right = randTree(depth-1);    // make right subtree
  return pnode;                        // all done
 }
```

Example of function call

```
NODE *root;
root = randTree(4);
```

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

## Build a random binary tree -- Java

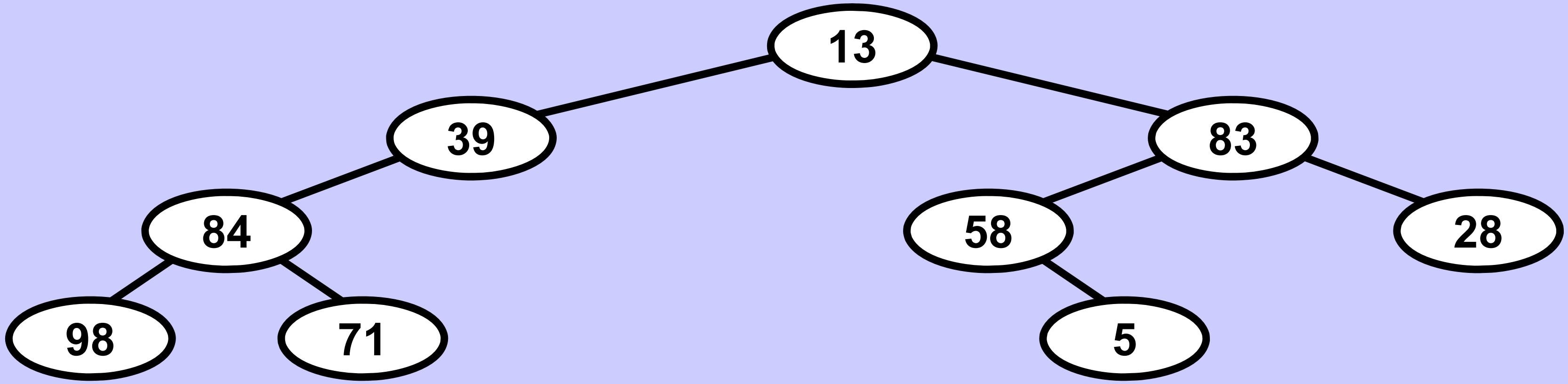
```
public Node randTree(int depth) {
  Node node;
  if ((depth <= 0) || ((int) Math.random()*10 > 7))
     return null;
                                   // create node with a key value
  node = new Node((int)(Math.random()*100));

  node.left = randTree(depth-1);  // create left subtree
  node.right = randTree(depth-1); // create right subtree
  return node;                    // all done
}
```

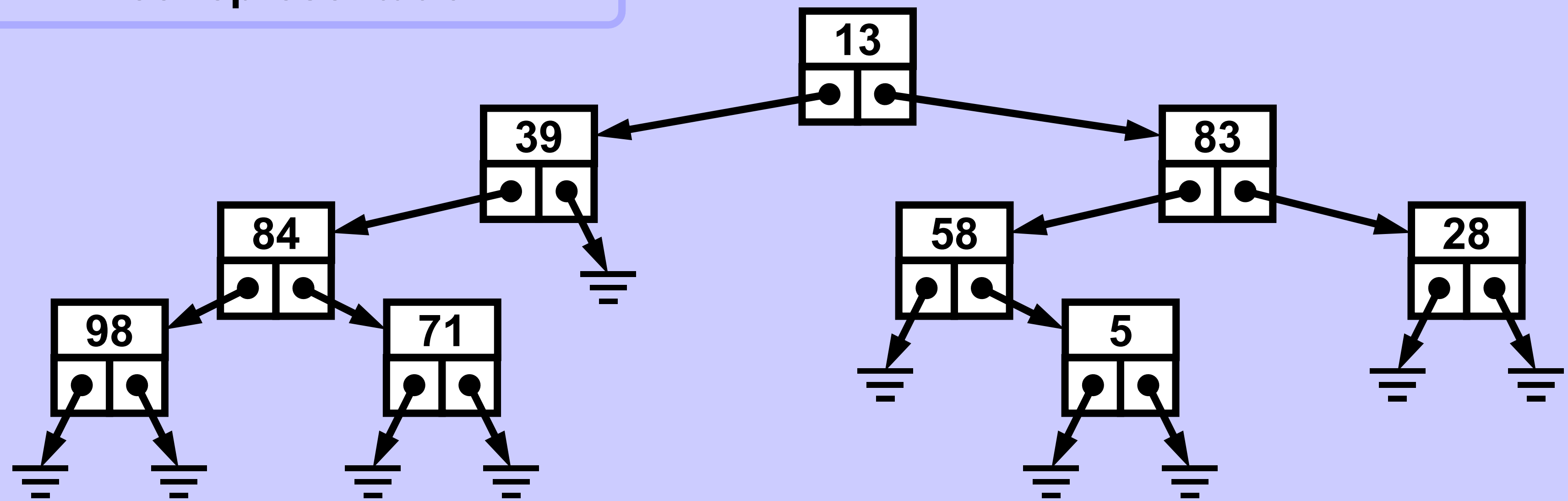
Example of function call

```
Node root;
root = randTree(4);
```

## Random binary tree

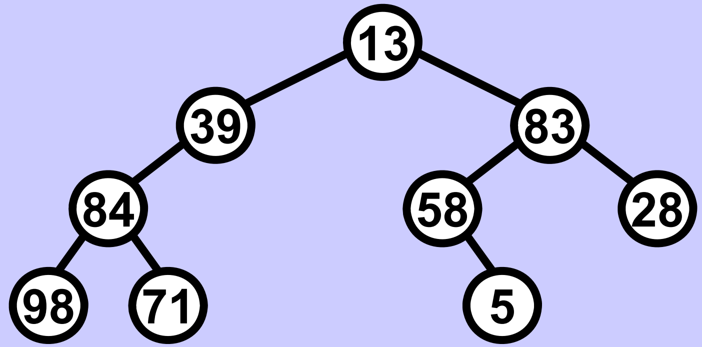



## Tree representation



# Inorder traversal of a binary tree

Tree




INORDER traversal

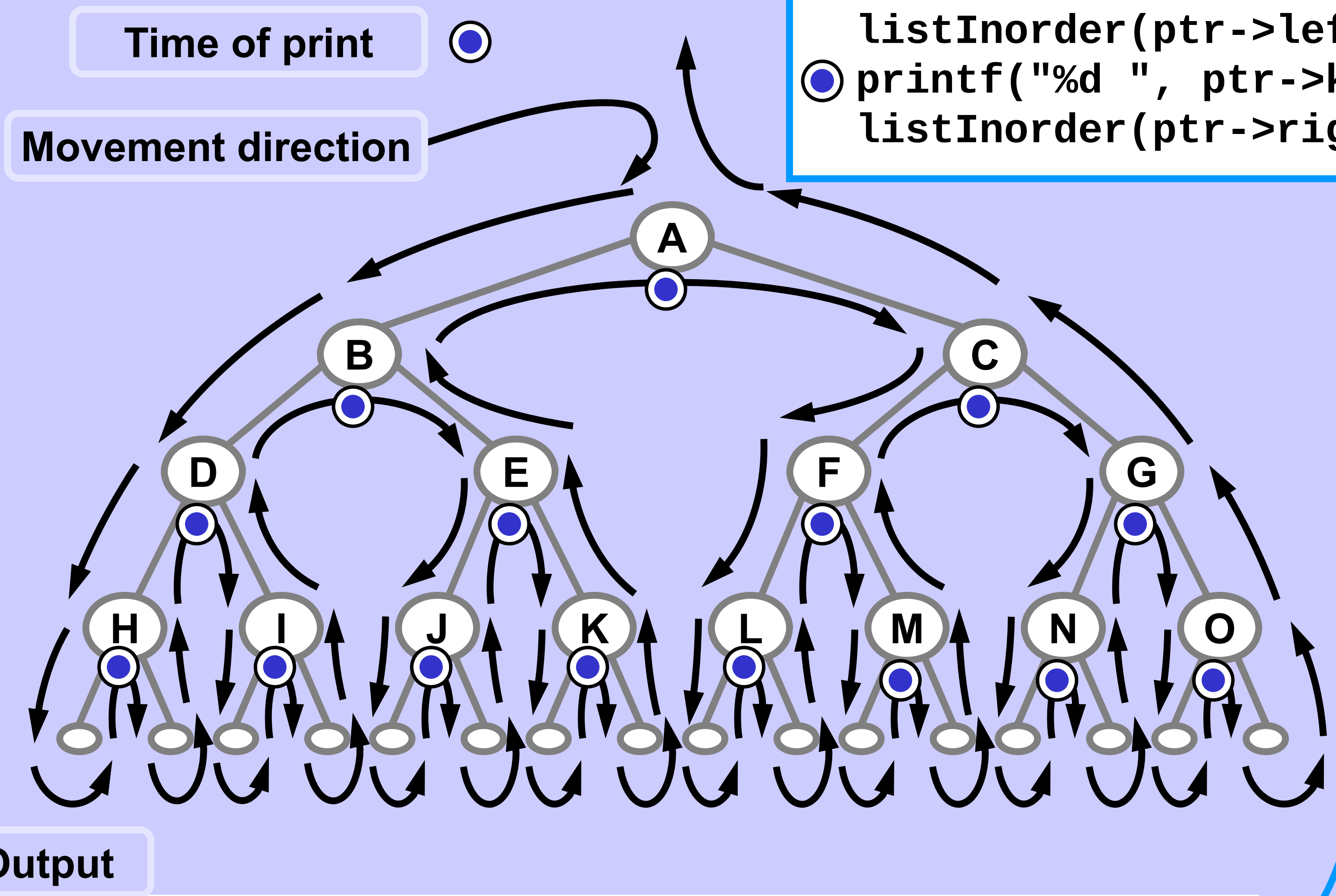
```
void listInorder( NODE *ptr) {
   if (ptr == NULL) return;
   listInorder(ptr->left);
   printf("%d ", ptr->key);
   listInorder(ptr->right);
}
```

Output

98 84 71 39 13 58 5 83 28

# Movement in the tree during inorder traversal

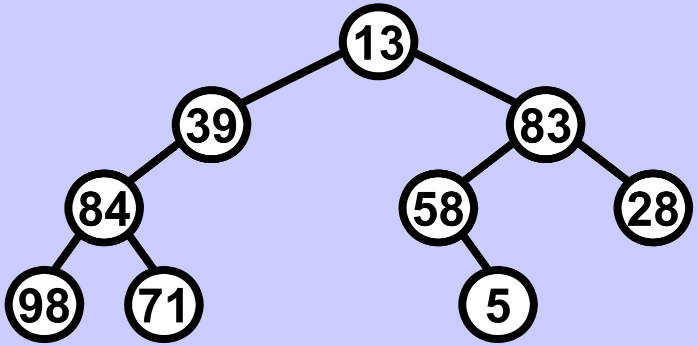
```
listInorder(ptr->left);
printf("%d ", ptr->key);
listInorder(ptr->right);
```




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

# Preorder traversal of a binary tree

**Tree**



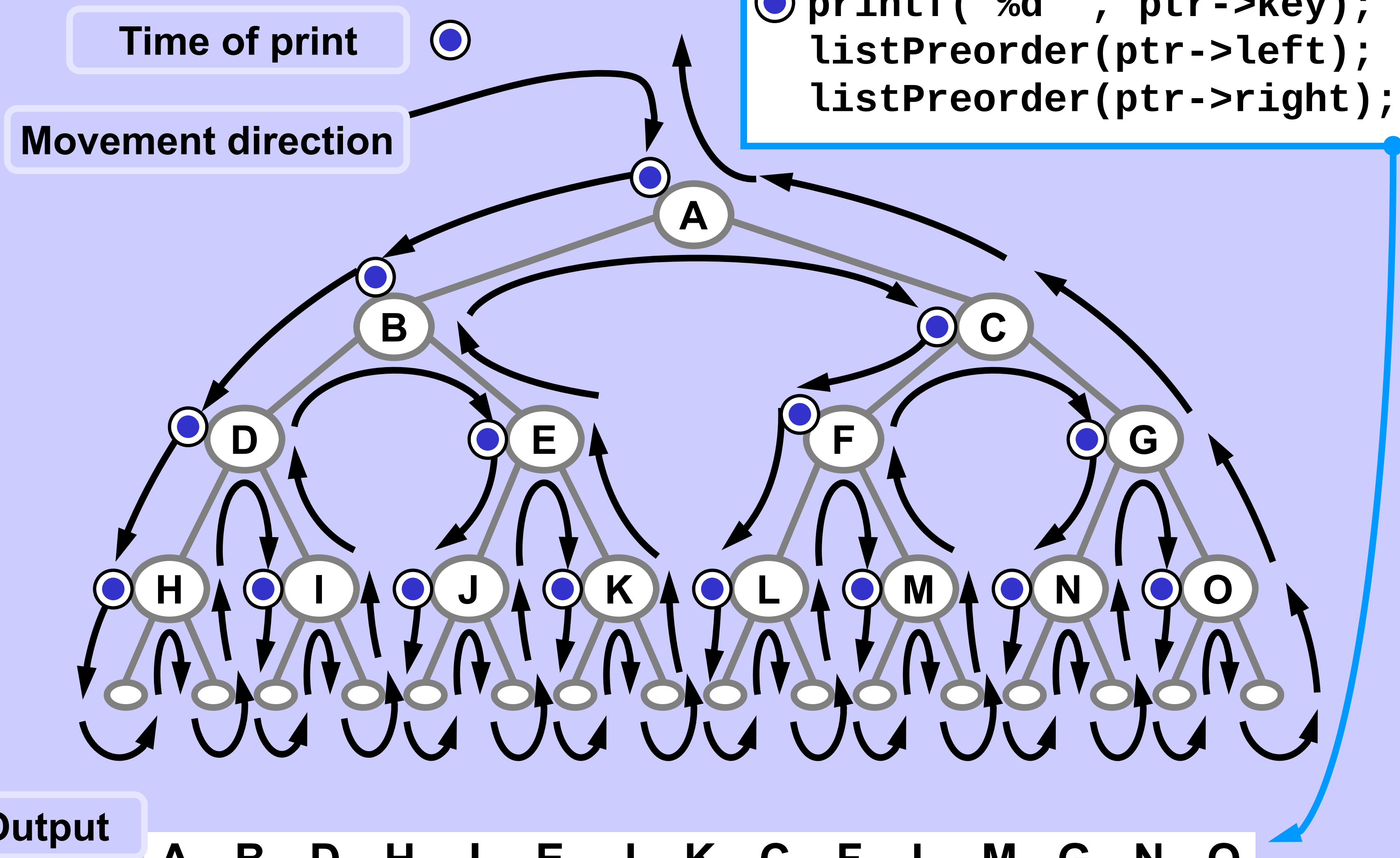

**PREORDER traversal**

```
void listPreorder( NODE *ptr) {
  if (ptr == NULL) return;
  printf("%d ", ptr->key);
  listPreorder(ptr->left);
  listPreorder(ptr->right);
}
```

**Output**

13 39 84 98 71 83 58 5 28

## Movement in the tree during preorder traversal

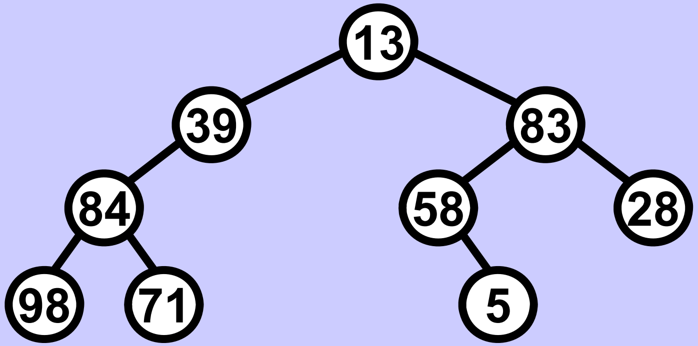



```
printf("%d ", ptr->key);
listPreorder(ptr->left);
listPreorder(ptr->right);
```

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

# Postorder traversal of a binary tree

**Tree**




**POSTORDER traversal**

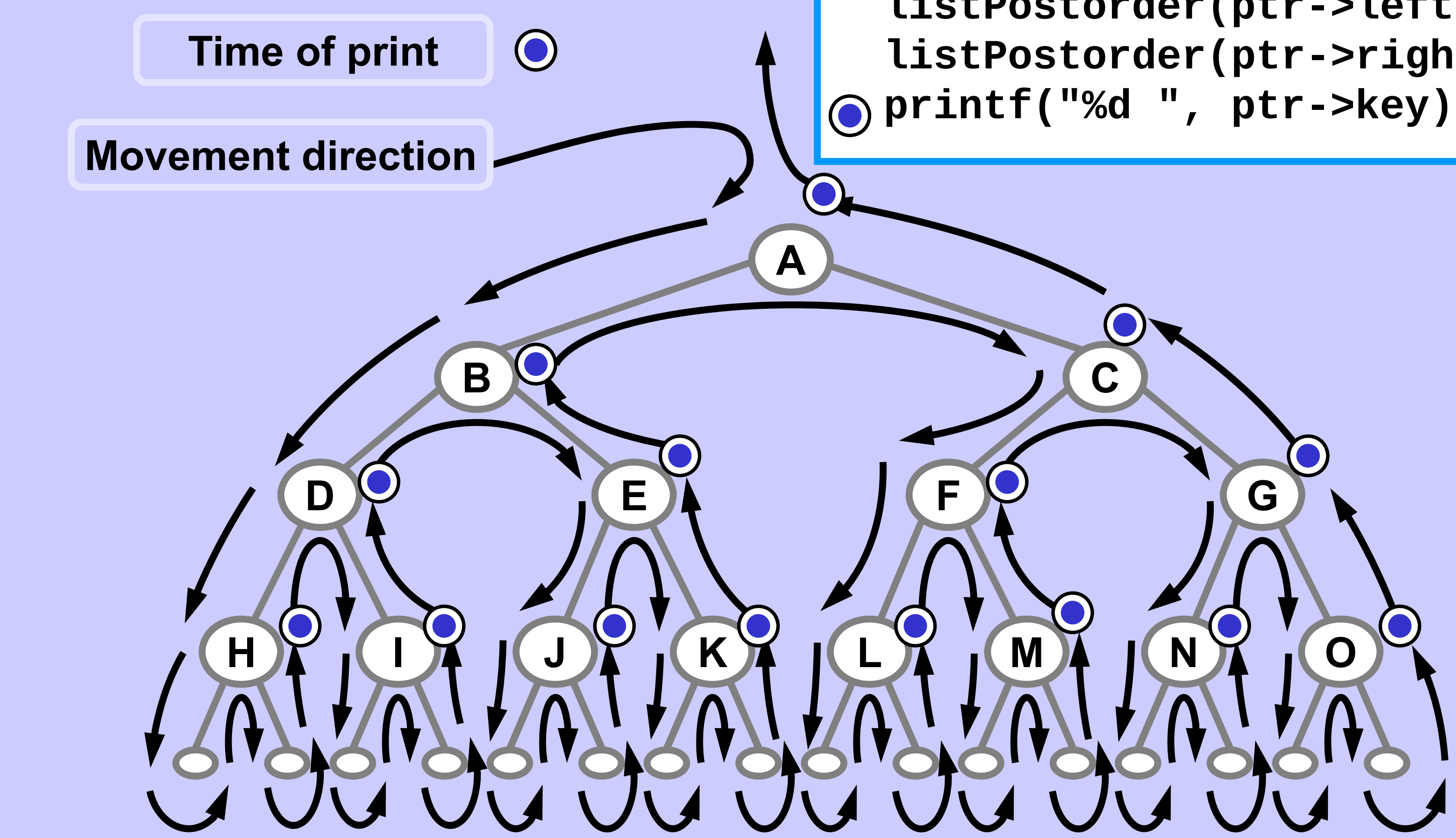
```
void listPostorder( NODE *ptr) {
   if (ptr == NULL) return;
   listPostorder(ptr->left);
   listPostorder(ptr->right);
   printf("%d ", ptr->key);
}
```

**Output**

98 71 84 39 5 58 28 83 13

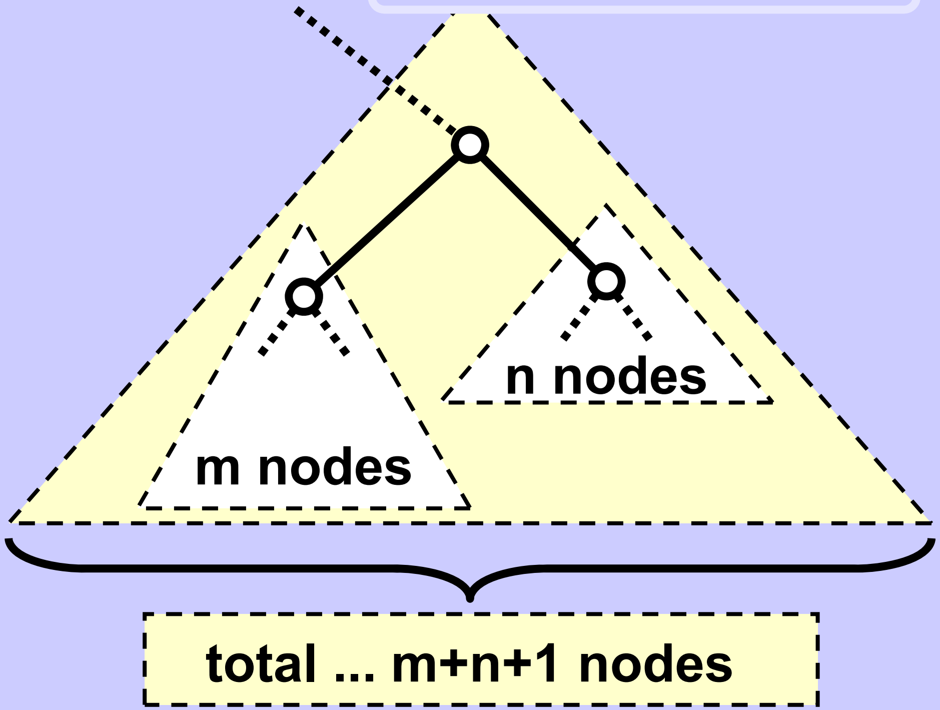
# Movement in the tree during postorder traversal

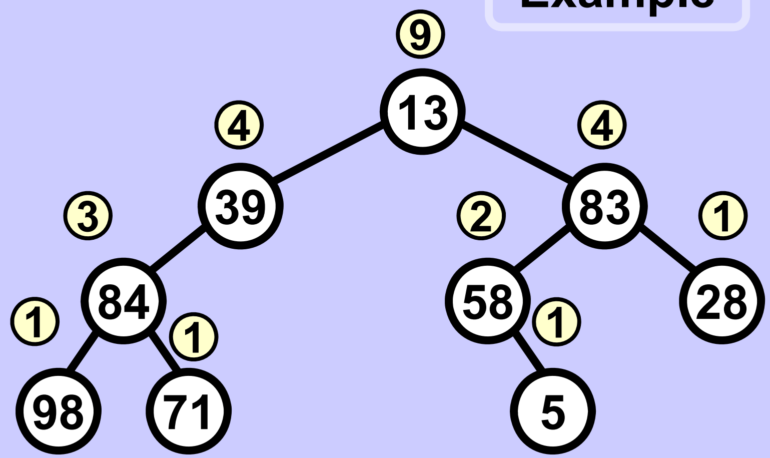
```
listPostorder(ptr->left);
listPostorder(ptr->right);
printf("%d ", ptr->key);
```




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

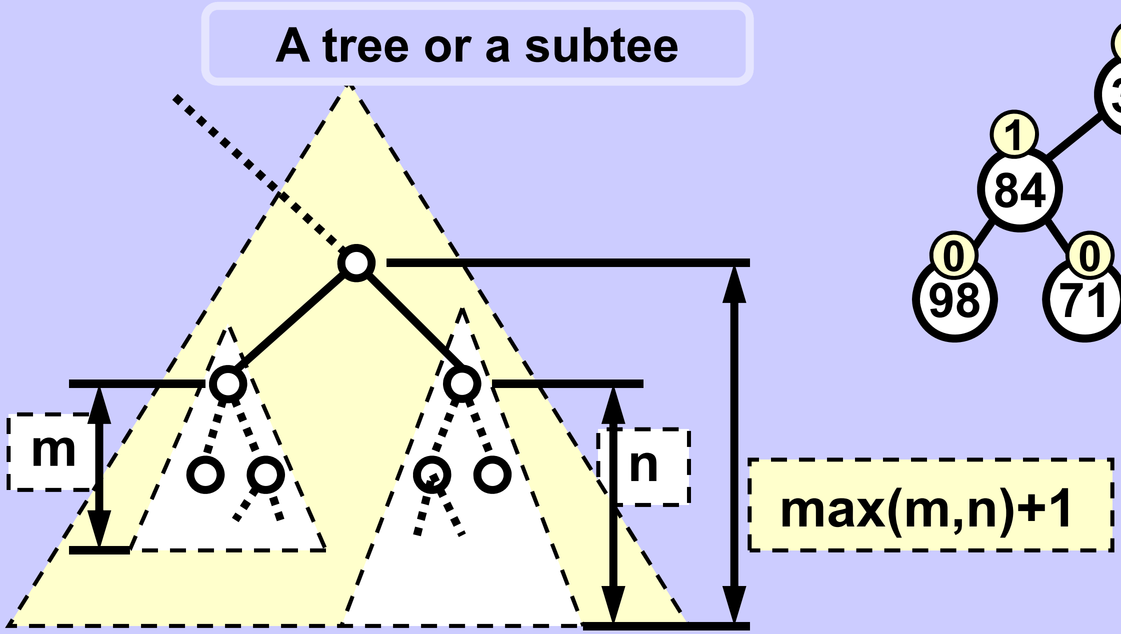
# Tree size (= number of nodes) recursively

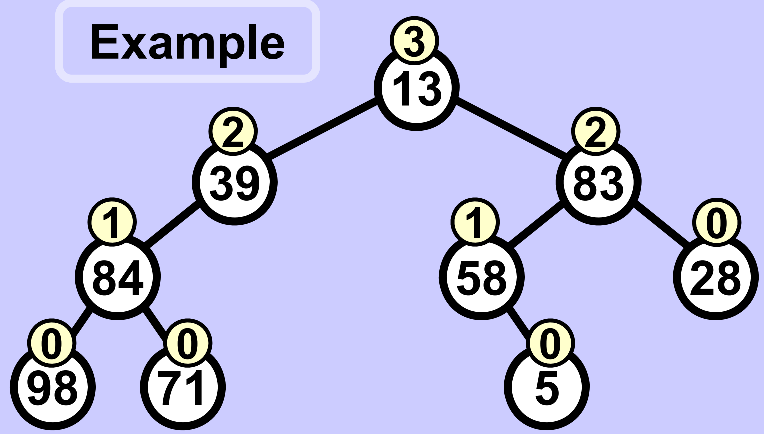



```
int count(NODE *ptr) {
  if (ptr == NULL) return (0);
  return (count(ptr->left) + count(ptr->right)+1);
}
```

# Tree depth (= max depth of a node) recursively



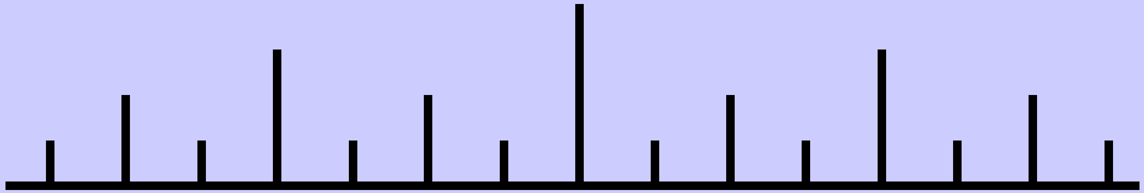




```
int depth(NODE *ptr) {
  if (ptr == NULL) return (-1);
  return ( max(depth(ptr->left), depth(ptr->right) )+1 );
}
```

# Simple recursive example

## Binary ruler

**Ruler notches**



**Notch lengths**

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

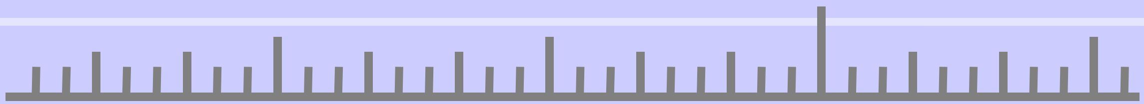
**Print the lengths of all notches**

```
void ruler(int val) {
  if (val < 1) return;

  ruler(val-1);
  print(val);
  ruler(val-1);
}
```

```
Call: ruler(4);
```

**Exercise: Ternary ruler:**



# Simple recursive example

## Binary ruler vs. Inorder traversal

**Ruler**

```
void ruler(int val) {
  if (val < 1) return;

  ruler(val-1);
  print(val);
  ruler(val-1);
}
```

**Inorder**

```
void listInorder( NODE *ptr) {
  if (ptr == NULL) return;

  listInorder(ptr->left);
  printf("%d ", ptr->key);
  listInorder(ptr->right);
}
```

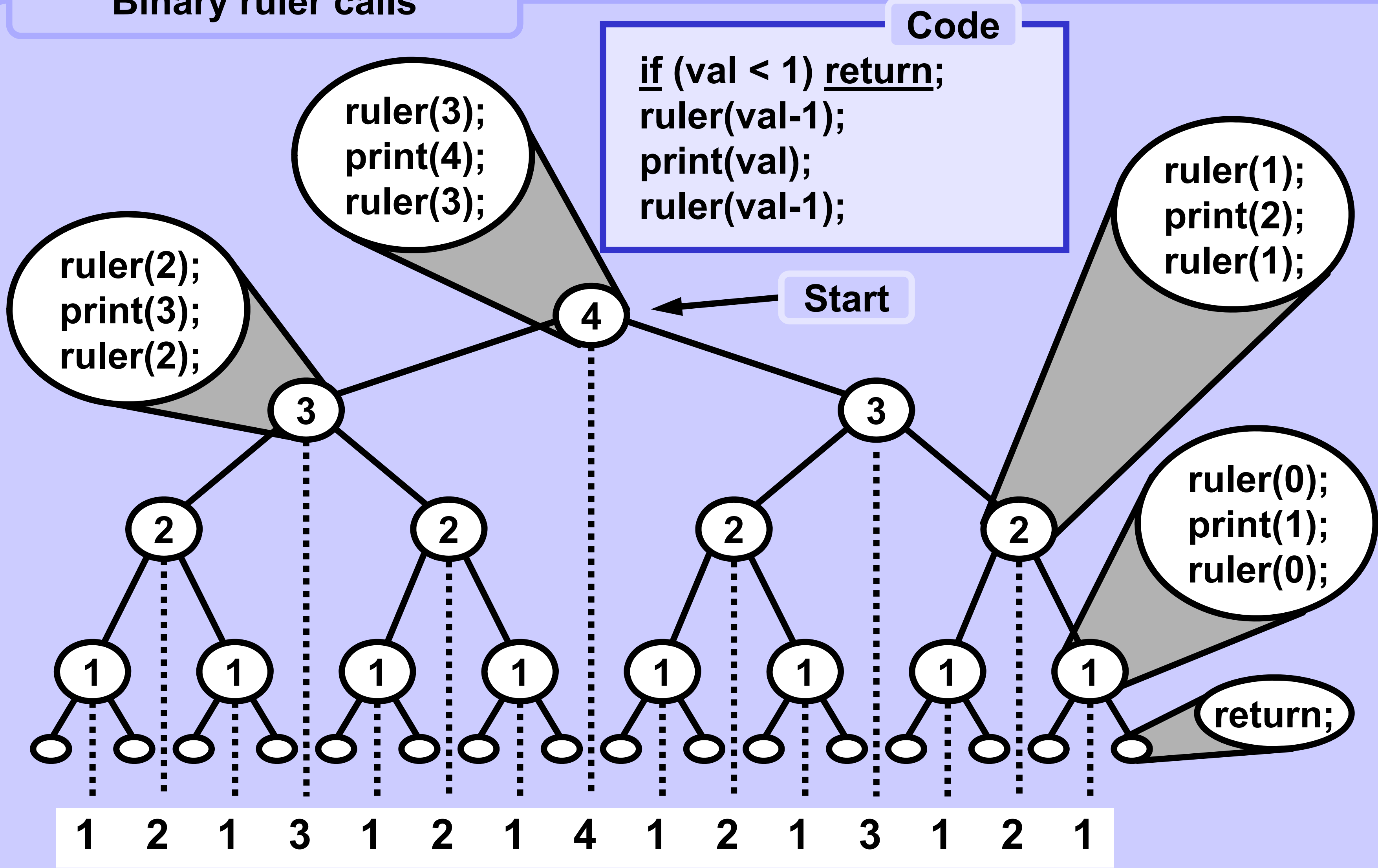
**Structurally identical!**

**Ruler output**

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 3 | 1 | 2 | 1 | 4 | 1 | 2 | 1 | 3 | 1 | 2 | 1 |

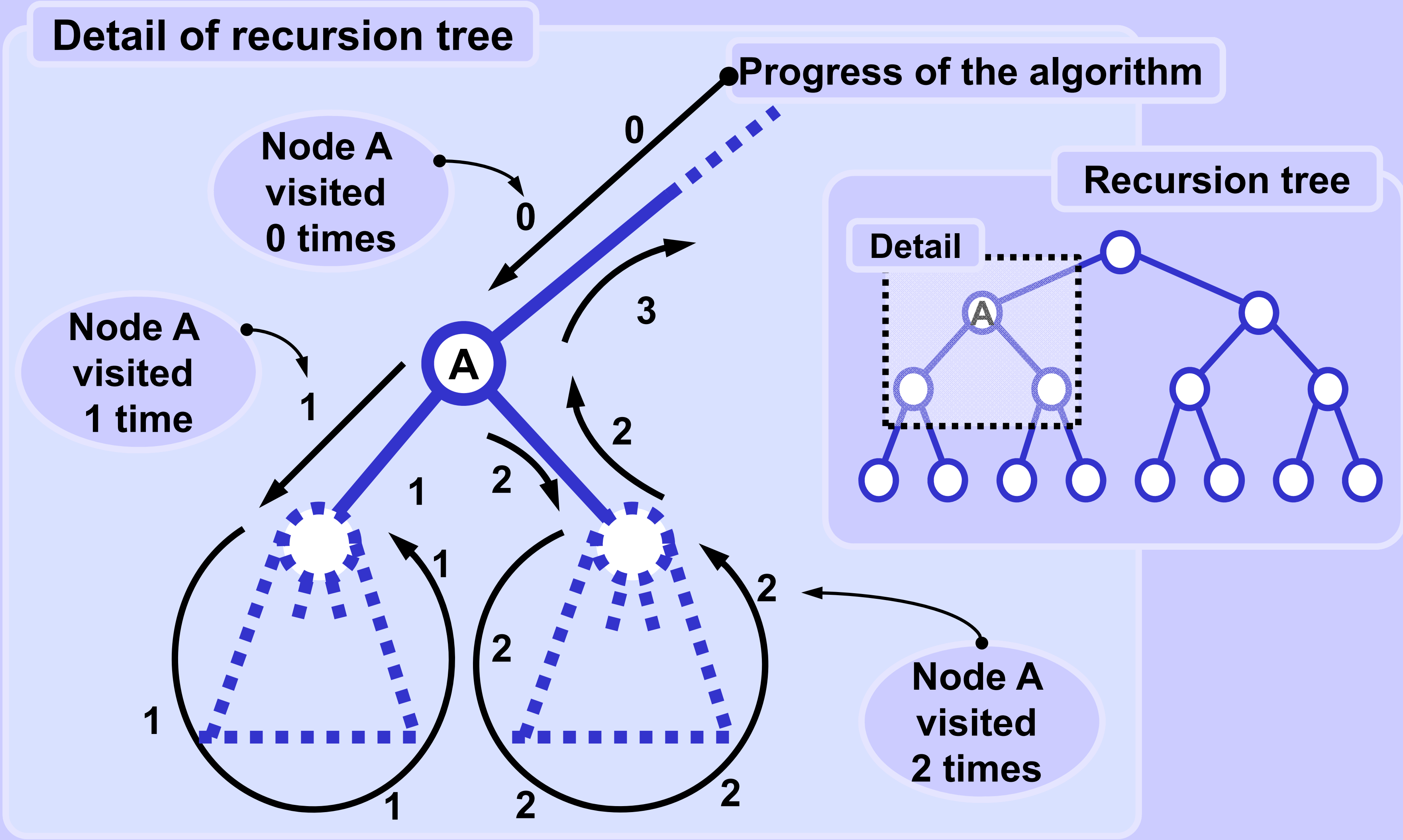
# Simple recursive example

## 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.**

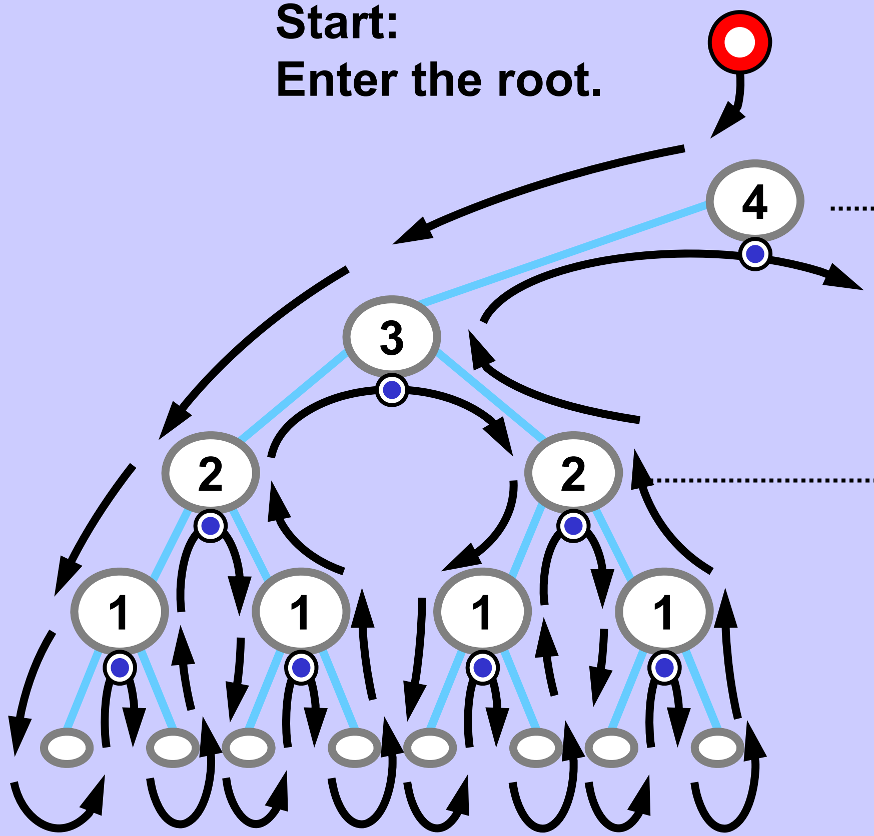
## Standard approach

**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



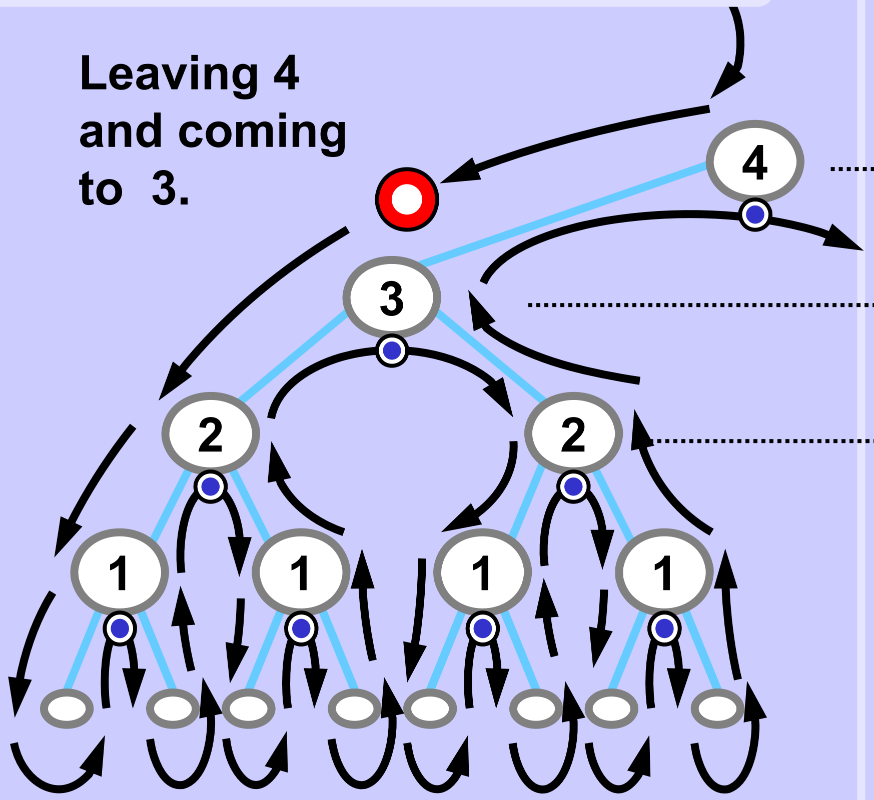

Stack

| Value | Visits |
|---|---|
| 4 | 0 |

push(4,0)

Output

# Stack implements recursion




## Stack

| Value | Visits |
|---|---|
| 4 | 1 |
| 3 | 0 |

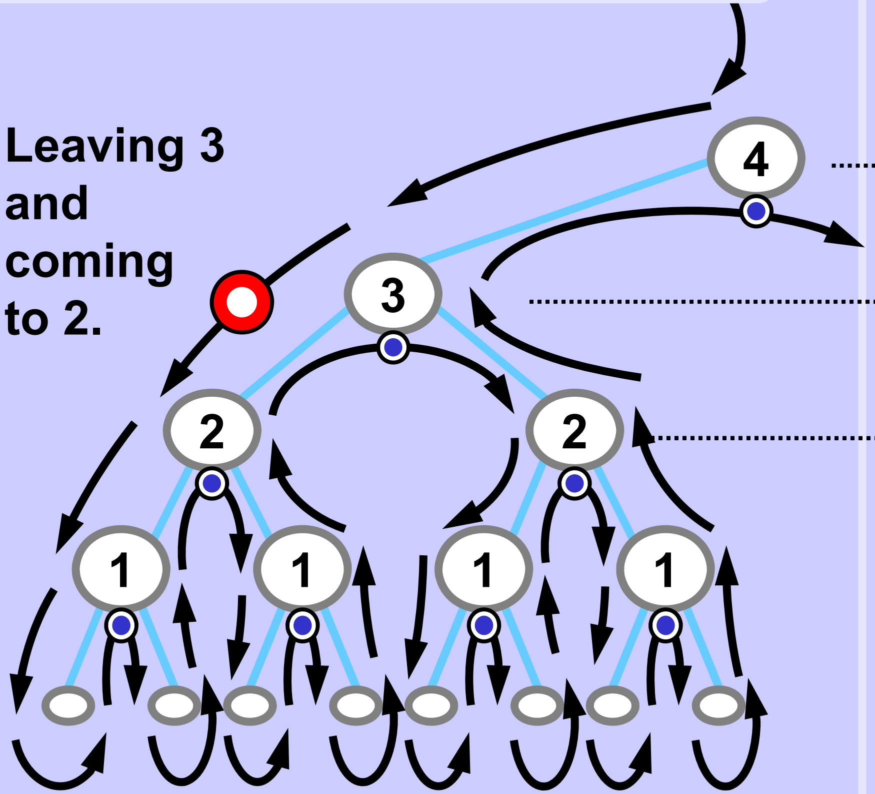
push(3,0)

Output

# Stack implements recursion

## Recursion tree traversal

Leaving 3 and coming to 2.




## Stack

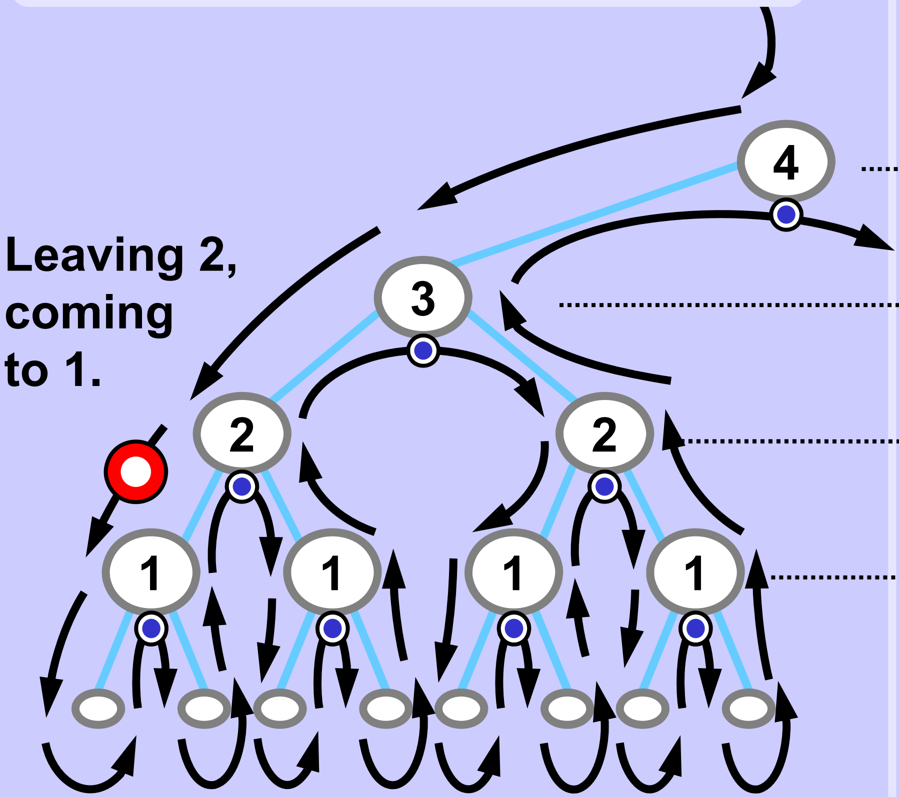
| Value | Visits |
|---|---|
| 4 | 1 |
| 3 | 1 |
| 2 | 0 |

push(2,0)

Output

# Stack implements recursion

## Recursion tree traversal




## Stack

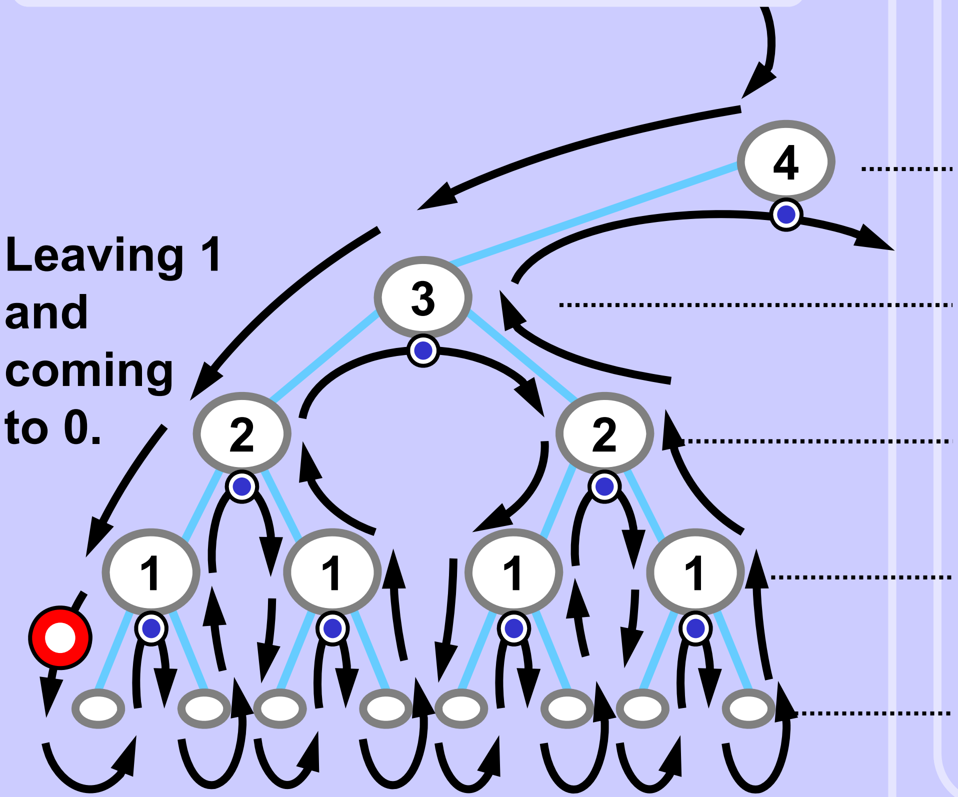
| Value | Visits |
|---|---|
| 4 | 1 |
| 3 | 1 |
| 2 | 1 |
| 1 | 0 |

push(1,0)

Output

# Stack implements recursion

## Recursion tree traversal




## Stack

| Value | Visits |
|---|---|
| 4 | 1 |
| 3 | 1 |
| 2 | 1 |
| 1 | 1 |
| 0 | 0 |

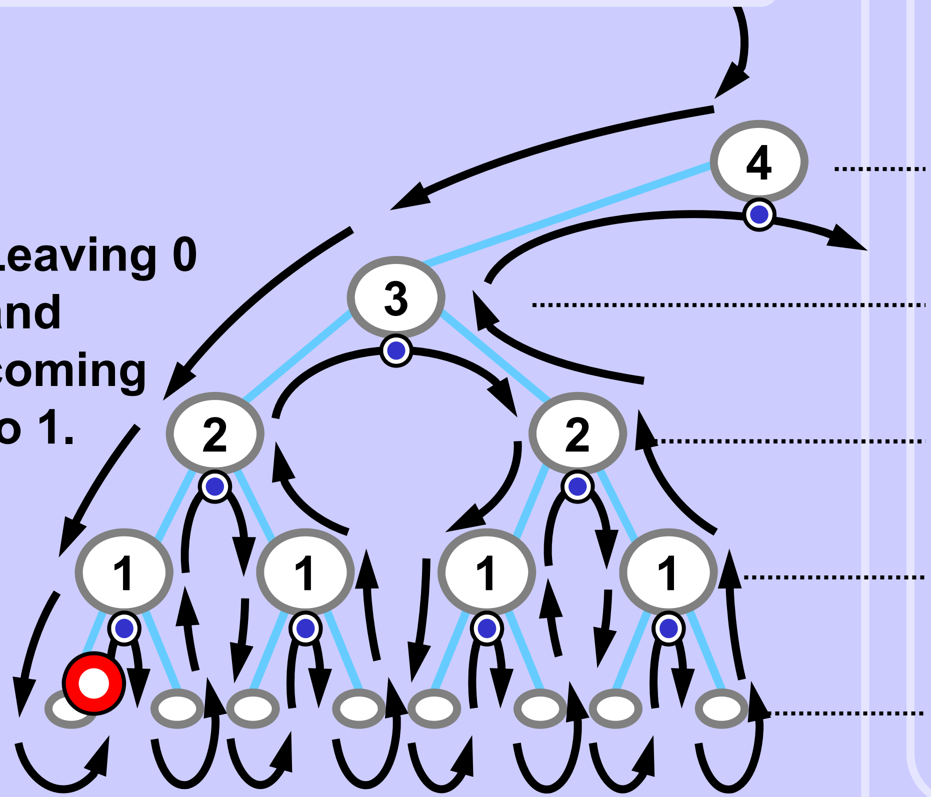
push(0,0)

Output

# Stack implements recursion

## Recursion tree traversal

Leaving 0 and coming to 1.




## Stack

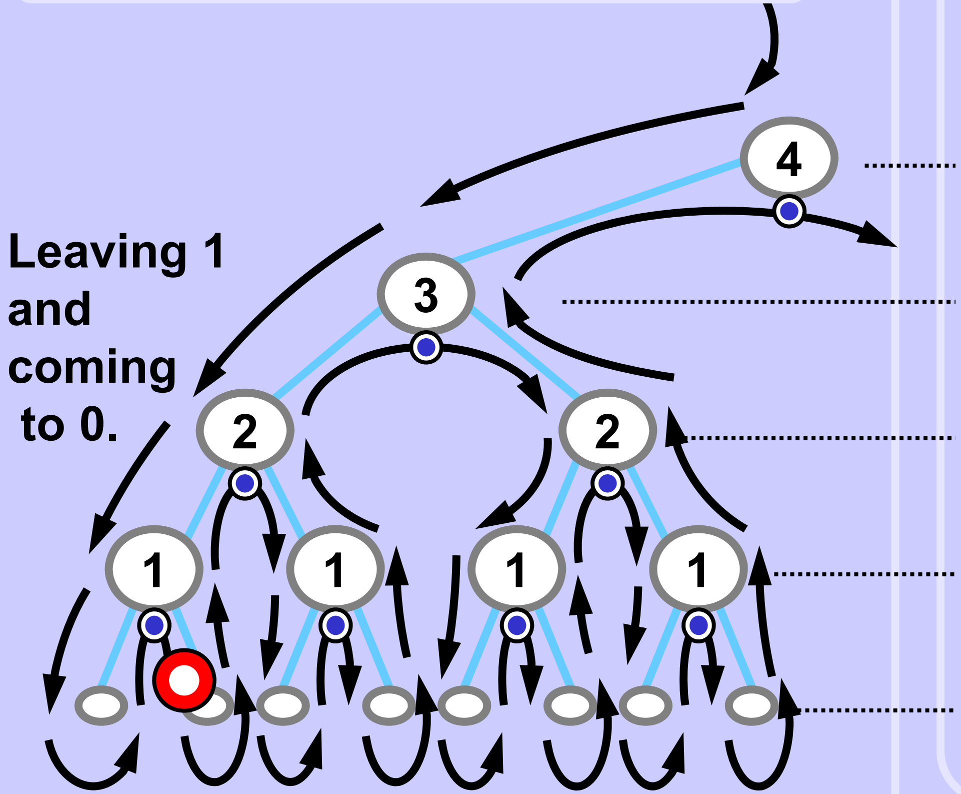
| Value | Visits |
| --- | --- |
| 4 | 1 |
| 3 | 1 |
| 2 | 1 |
| 1 | 1 |
| 0 | 0 |

pop()

Output

# Stack implements recursion

## Recursion tree traversal




## Stack

| Value | Visits |
|---|---|
| 4 | 1 |
| 3 | 1 |
| 2 | 1 |
| 1 | 2 |
| 0 | 0 |

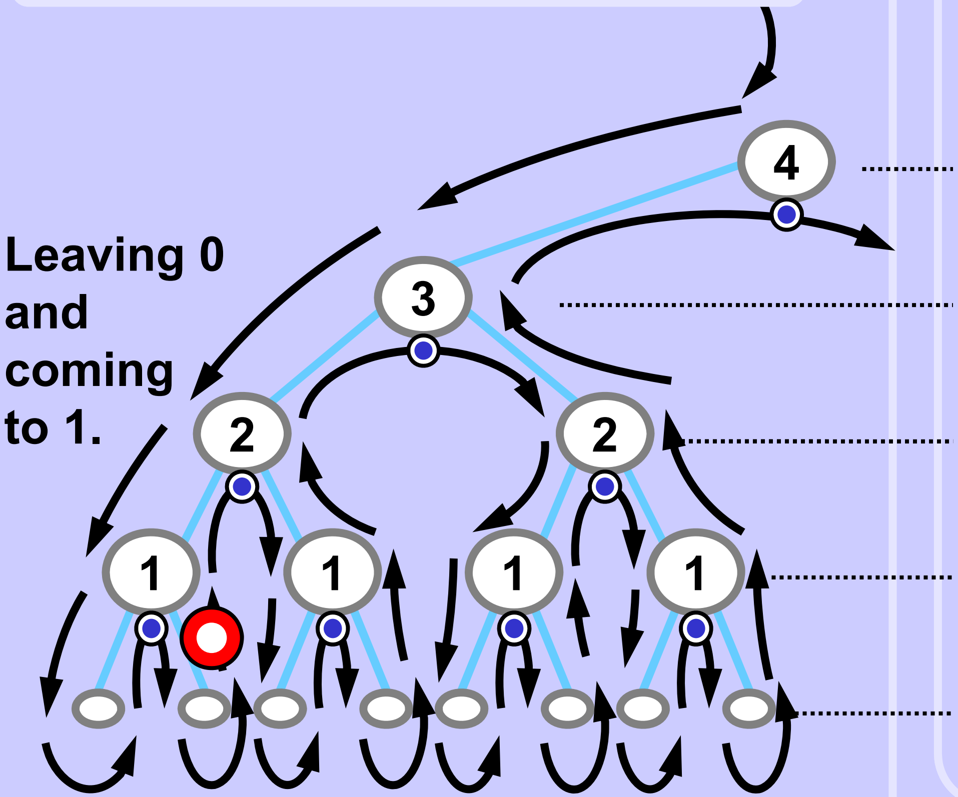
push(0,0)

Output

1

# Stack implements recursion

## Recursion tree traversal




## Stack

| Value | Visits |
|---|---|
| 4 | 1 |
| 3 | 1 |
| 2 | 1 |
| 1 | 2 |
| 0 | 0 |

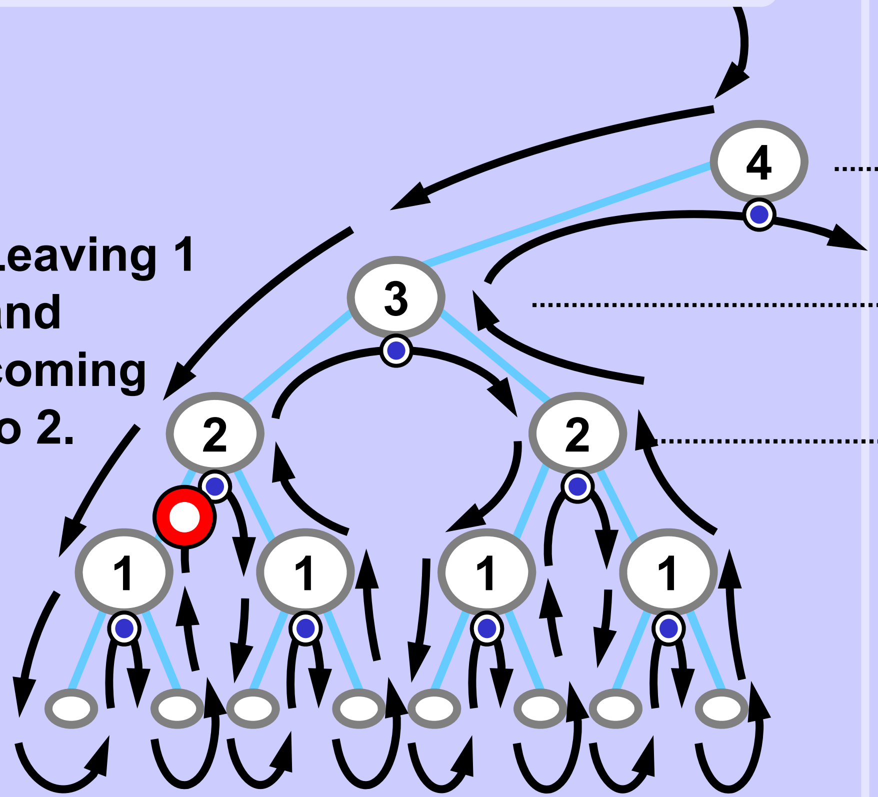
pop()

Output: 1

# Stack implements recursion

## Recursion tree traversal

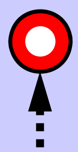
Leaving 1 and coming to 2.




## Stack

| Value | Visits |
|---|---|
| 4 | 1 |
| 3 | 1 |
| 2 | 1 |
| 1 | 2 |

pop()

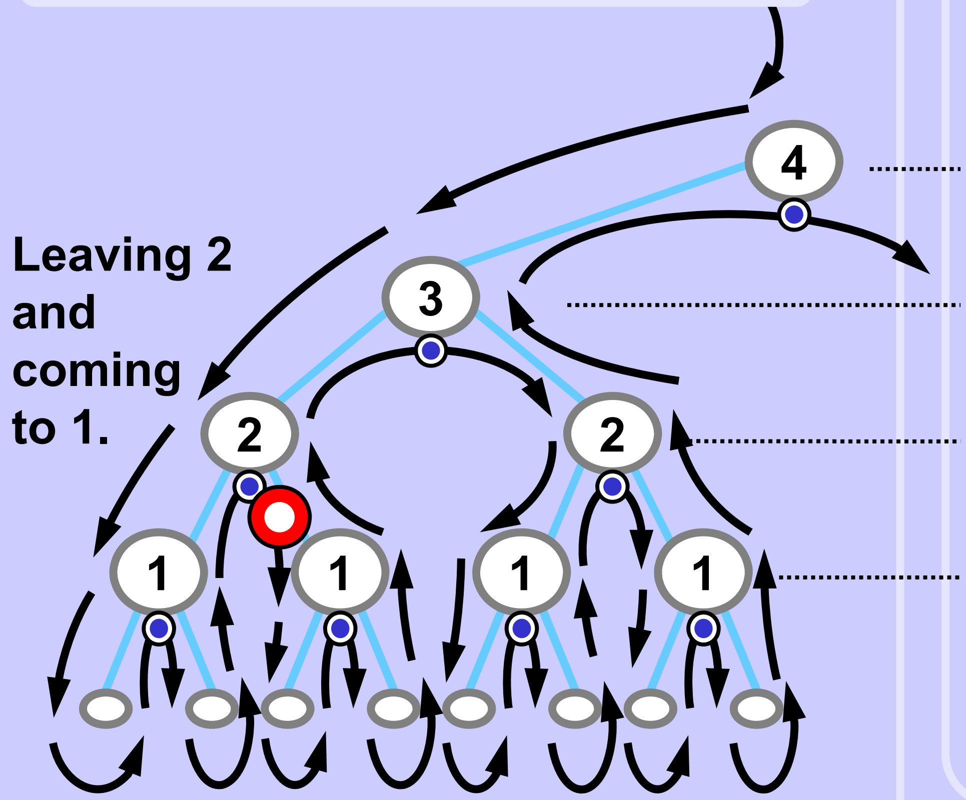


1

Output

# Stack implements recursion

## Recursion tree traversal




## Stack

| Value | Visits |
|---|---|
| 4 | 1 |
| 3 | 1 |
| 2 | 2 |
| 1 | 0 |

push(1,0)

Output: 1 2

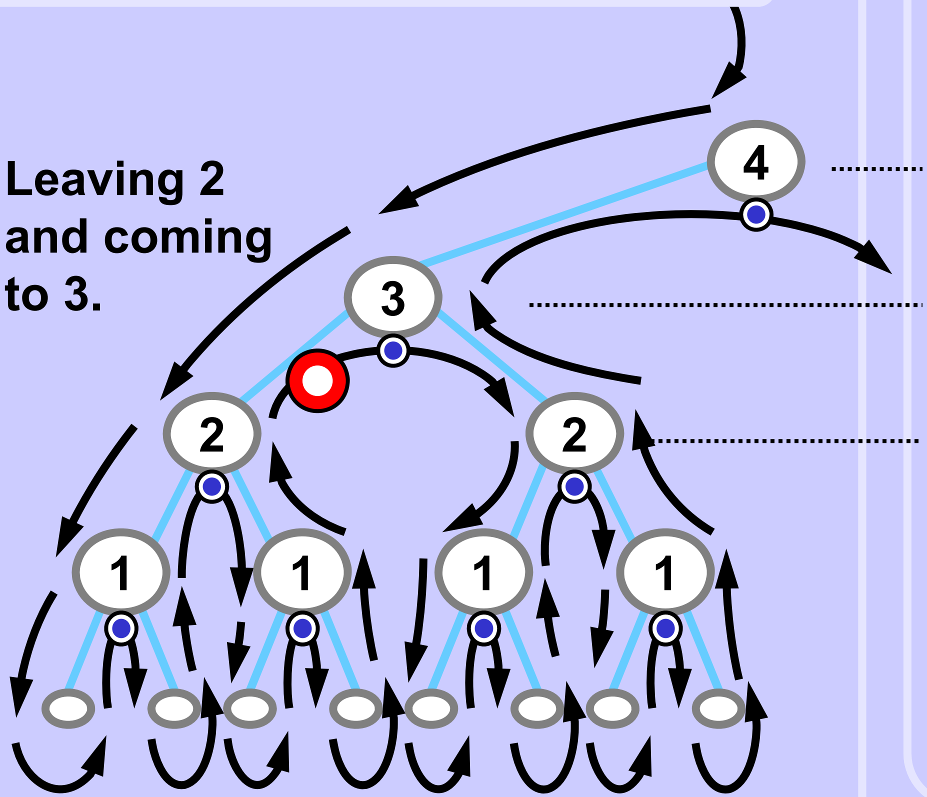
atd...

# 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()

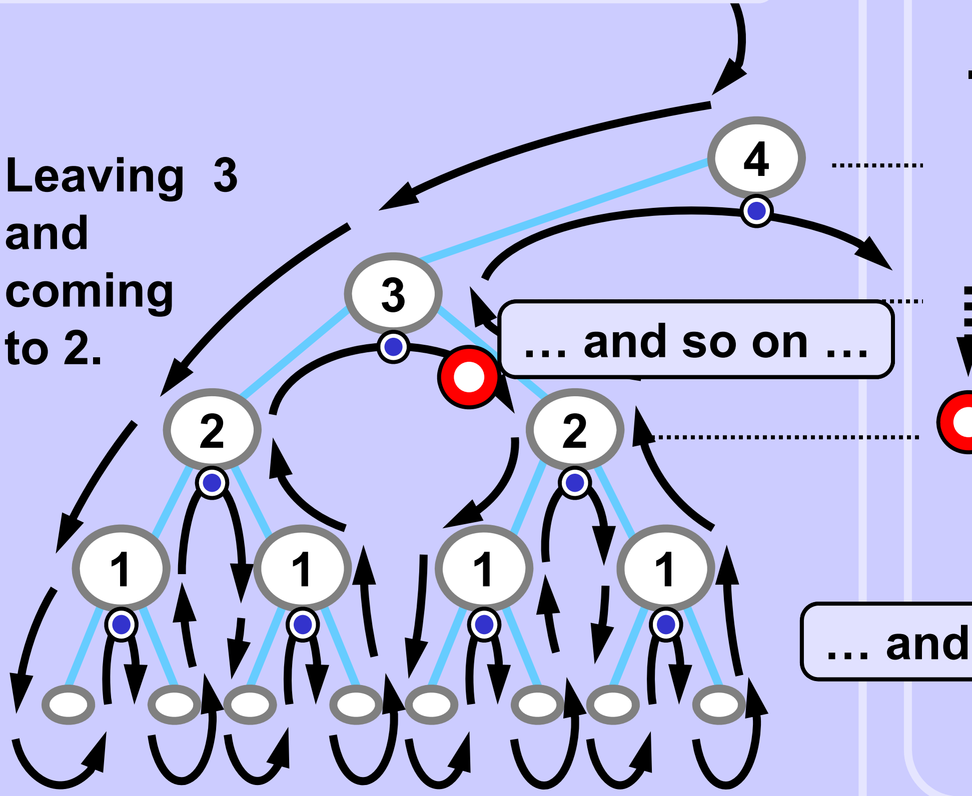
**Output**

1 2 1

# Stack implements recursion

## Recursion tree traversal

Leaving 3
and
coming
to 2.




... and so on ...

## Stack

| Value | Visits |
|---|---|
| 4 | 1 |
| 3 | 2 |
| 2 | 0 |

push(2,0)

... and so on ...

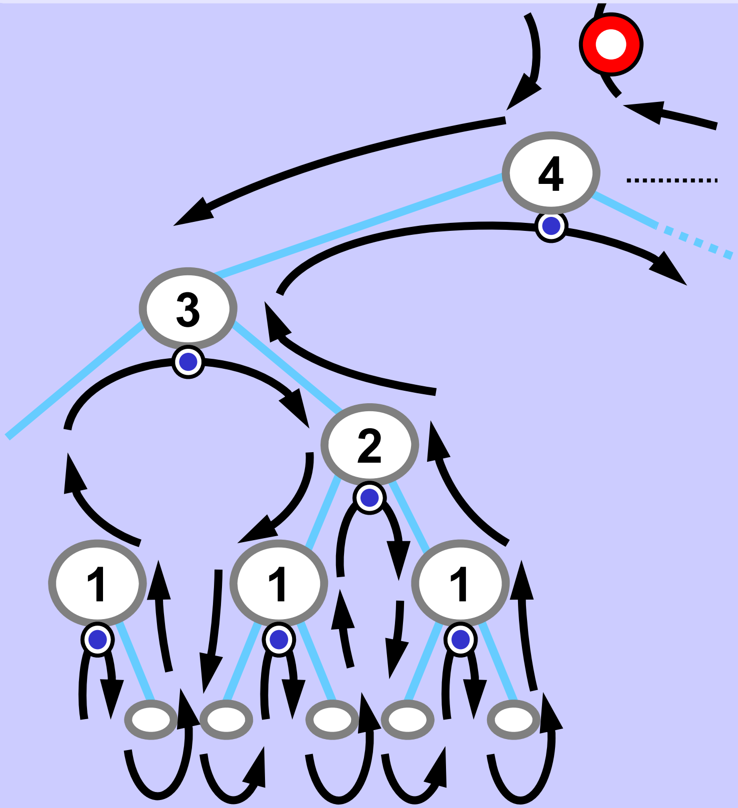
Output

1 2 1 3

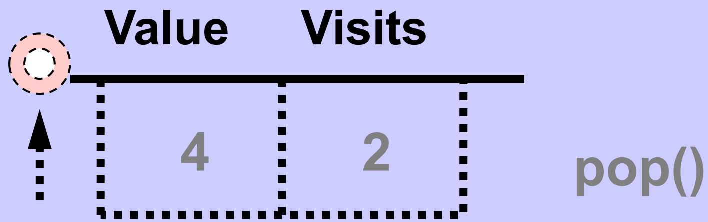
## Stack implements recursion

… after another while … completed.

Recursion tree traversal




Stack




(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

```
stack.init();
stack.top.value = N; stack.top.visits = 0;
while (!stack.empty()) {
      if (stack.top.value == 0) stack.pop();
      if (stack.top.visits == 0) {
          stack.top.visits++;
          stack.push(stack.top.value-1,0);
          }
      if (stack.top.visits == 1) {
          print(stack.top.value);
          stack.top.visits++;
          stack.push(stack.top.value-1,0);
          }
      if (stack.top.visits == 2) stack.pop();
}
```

## Stack implements recursion

### Recursive ruler without recursive calls
### Easy implementation with arrays

```
int stackVal[10];  int stackVis[10];
void ruler2(int N) {
  int SP = 0;                               // stack pointer
  stackVal[SP] = N; stackVis[SP] = 0;       // init
  while (SP >= 0) {                         // while unempty
    if (stackVal[SP] == 0) SP--;            // pop: in leaf
    if (stackVis[SP] == 0) {                // first visit
      stackVis[SP]++; SP++;
      stackVal[SP] = stackVal[SP-1]-1;      // go left
      stackVis[SP] = 0;
    }
    if (stackVis[SP] == 1) {                // second visit
      printf("%d ", stackVal[SP]);          // process the node
      stackVis[SP]++; SP++;
      stackVal[SP] = stackVal[SP-1]-1;      // go right
      stackVis[SP] = 0;
    }
    if (stackVis[SP] == 2) SP--;            // pop: node done
} }
```

## Stack implements recursion

**Recursive ruler without recursive calls**
**Easy implementation with arrays**

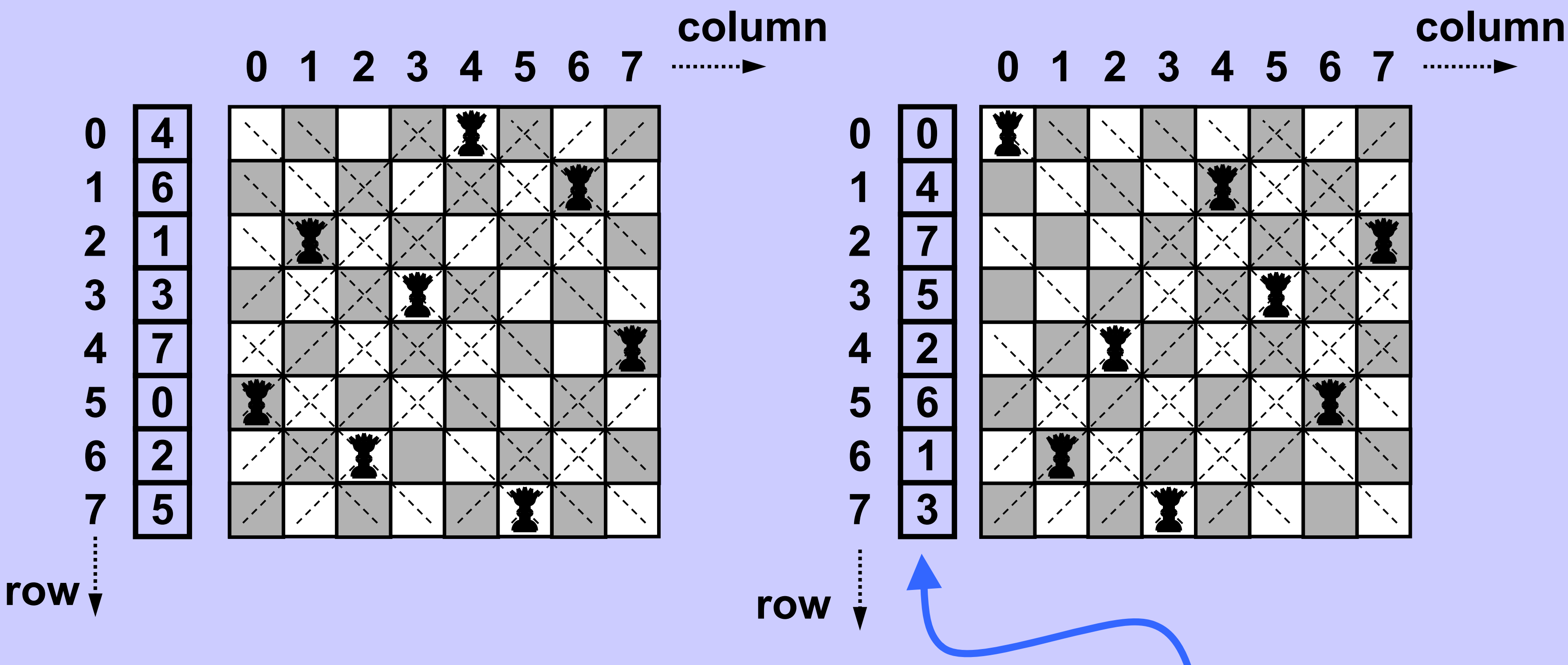
### A little more compact code

```
int stackVal[10];  int stackVis[10];

void ruler2(int N) {
  int SP = 0;                               // stack pointer
  stackVal[SP] = N; stackVis[SP] = 0;       // init
  while (SP >= 0) {                         // while unempty
    if (stackVal[SP] == 0) SP--;            // pop: in leaf
    if (stackVis[SP] == 2) SP--;            // pop: node done
    else {
    if (stackVis[SP] == 1)                  // if second visit
      printf("%d ", stackVal[SP]);          // process the node
    stackVis[SP]++; SP++;                   // otherwise
    stackVal[SP] = stackVal[SP-1]-1;        // go deeper
    stackVis[SP] = 0;
} } }
```

# Easy backtrack problem 8 queens puzzle

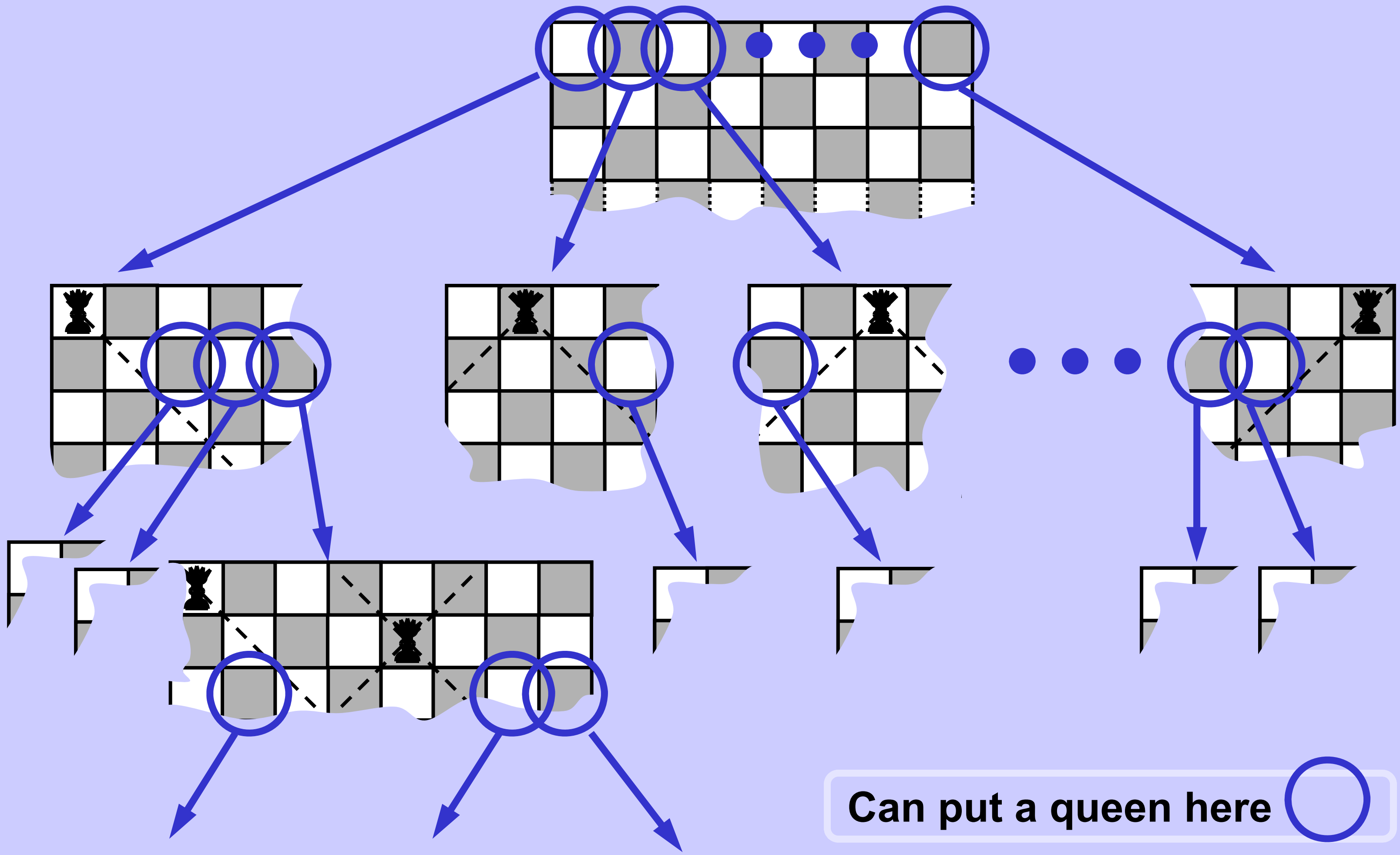
## Some solutions




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

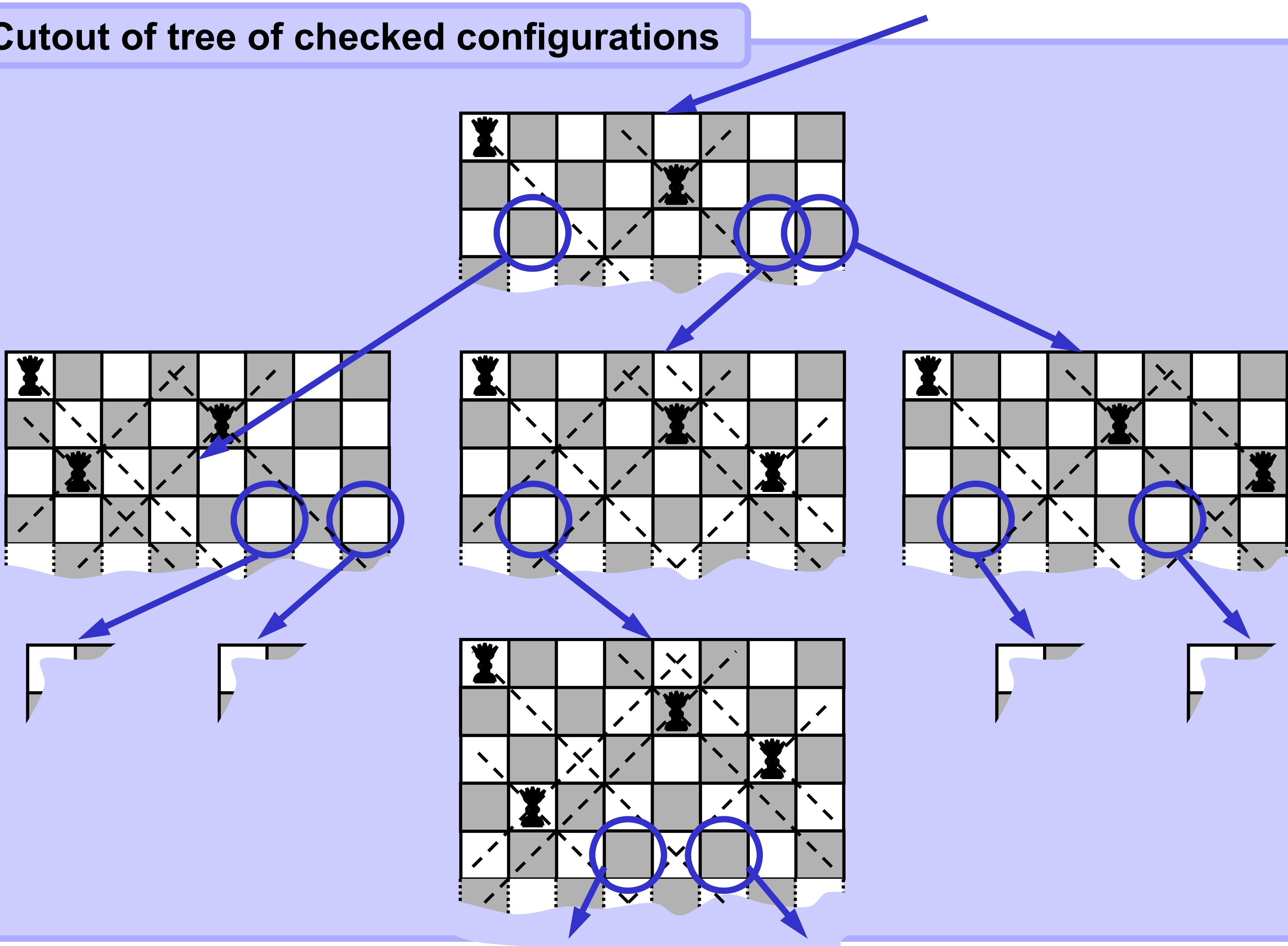
# Easy backtrack problem 8 queens puzzle

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



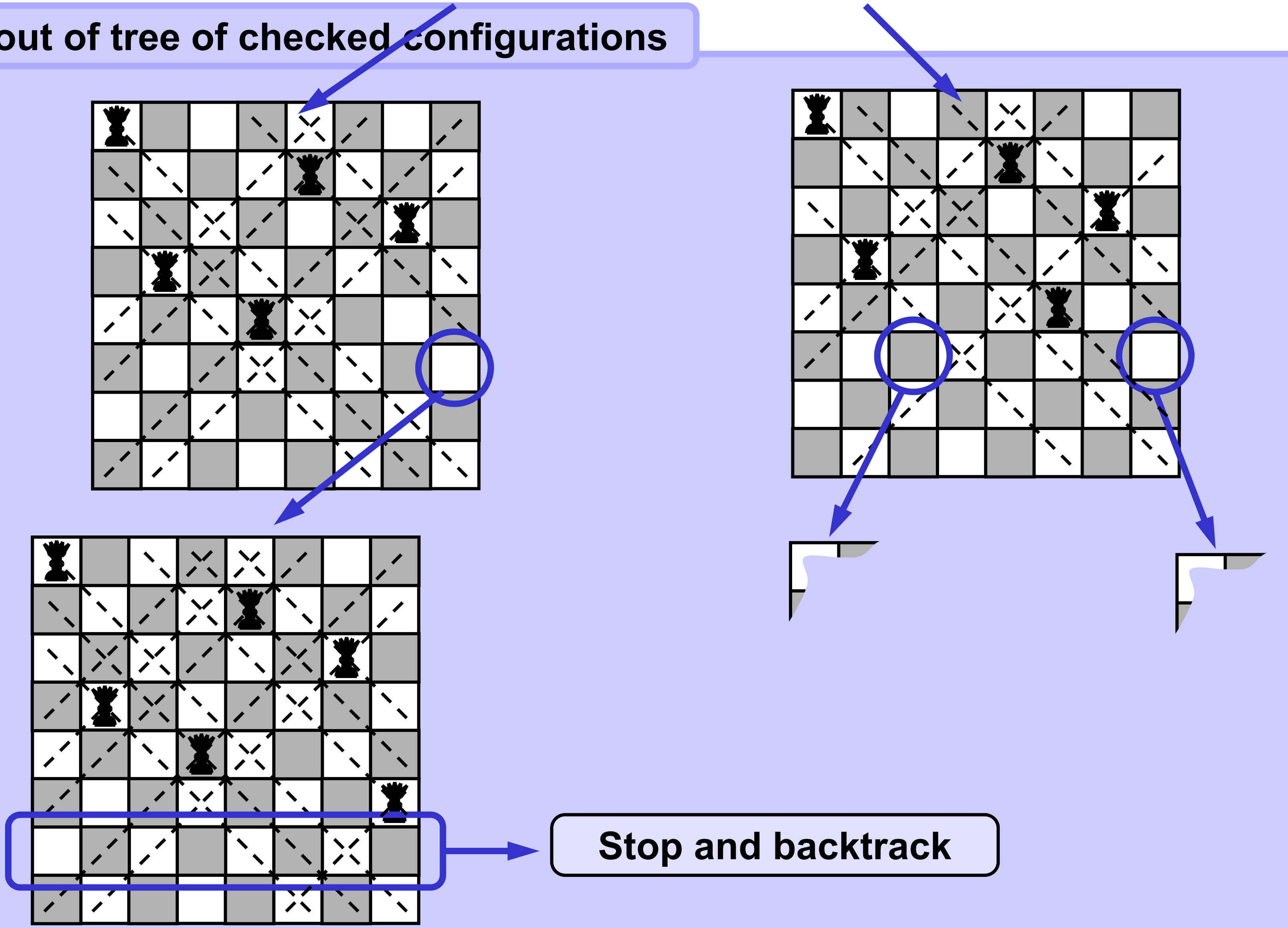
# Easy backtrack problem 8 queens puzzle

## 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

```
boolean positionOK(int r, int c) {       // r: row, c: column
  for (int i = 0; i < r; i++)
    if ((queenCol[i] == c) ||            // same column or
      (abs(r-i) == abs(queenCol[i]-c))) // same diagonal
      return false;
  return true;
}

void putQueen(int row, int col) {
  queenCol[row] = col;                // put a queen there
  if (++row == N)                     // if solved
    print(queenCol);                  // output solution
  else
    for(col = 0; col < N; col++)      // test all columns
      if (positionOK(row, col))       // if free
          putQueen(row, col);         // next row recursion
}
```

```
Call: for(int col = 0; col < 8; col++)
         putQueen(0, col);
```