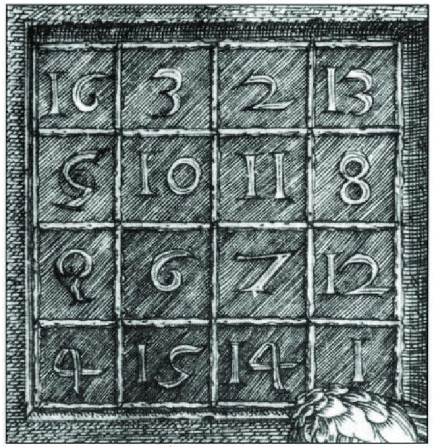
# Logic and Discrete Structures -LDS



Course 8 – Trees. Touples

S.l. Dr. Eng. Cătălin Iapă

catalin.iapa@cs.upt.ro

## What have we covered so far?

Functions

Recursive functions

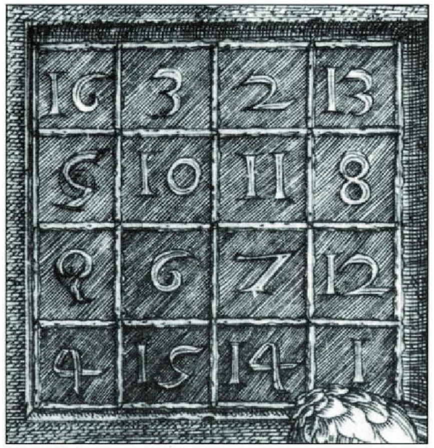
Lists

Sets

Relations

Dictionaries

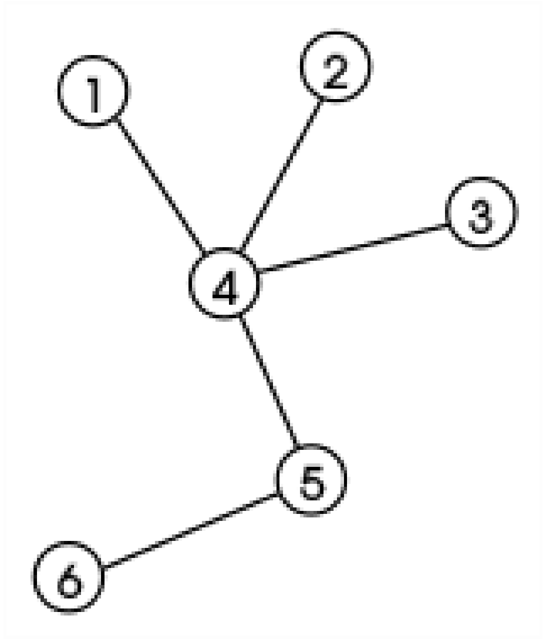
Graphs



# Trees

A tree is a connected undirected graph without cycles.

Trees are the simplest graphs in terms of structure in the class of connected graphs, and they are also the most commonly used in practice.



http://en.wikipedia.org/wiki/File:Tree_graph.svg

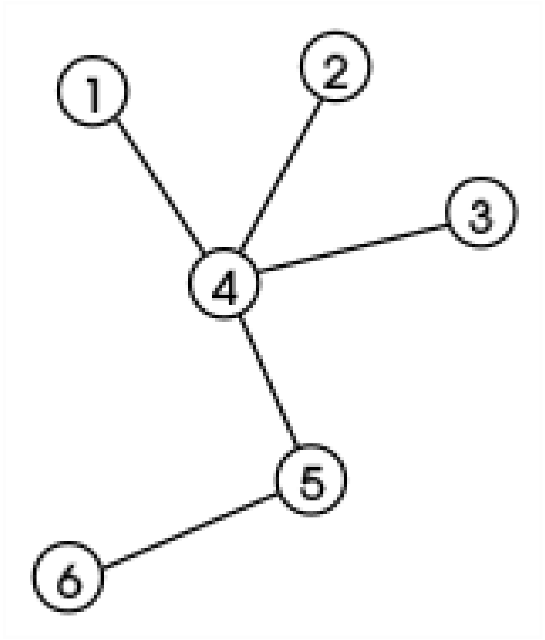
# Trees

A tree is a connected undirected graph without cycles.

connected = path between any 2 vertices (of 1 or more steps)

It is composed of vertices and edges.

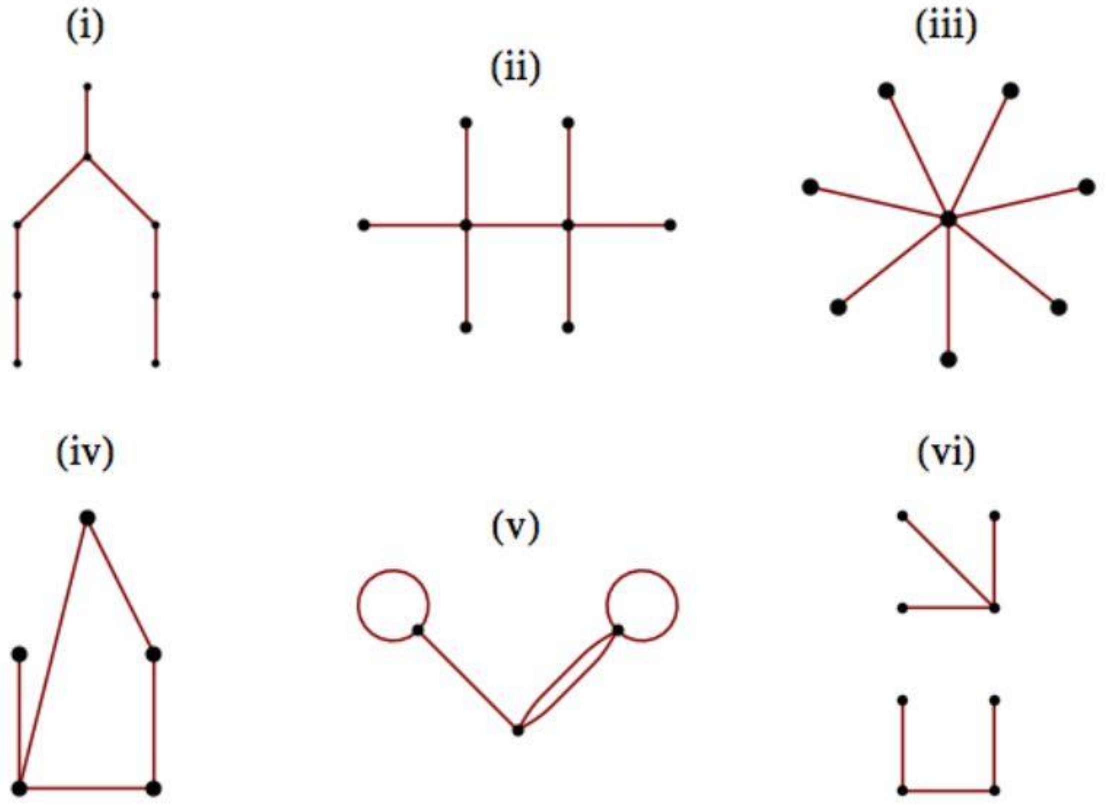
A tree with n vertices has n - 1 edges.



http://en.wikipedia.org/wiki/File:Tree_graph.svg

# Trees exemples

- (i) - tree
- (ii) - tree
- (iii) - tree
- (iv) - is not tree
- (v) - is not tree
- (vi) - is not tree (is forest)



# Forest

A type of graph closely related to the concept of a tree, but not fulfilling all the conditions of a tree, is the forest.

A forest is an undirected unconnected graph whose connected components are trees.

# Conditions for a graph to be a tree

If we have the graph G = (V, E) undirected and cycle-free, and |V| = n, the following propositions are equivalent:

- G is a tree
- For every 2 distinct vertices in V, there is only one path between them
- G is connected, and if we have an edge e, then the graph (V, E - {e}) is not connected
- G contains no cycles, but if we add an extra edge we have a cycle
- G is connected, and |E| = n-1

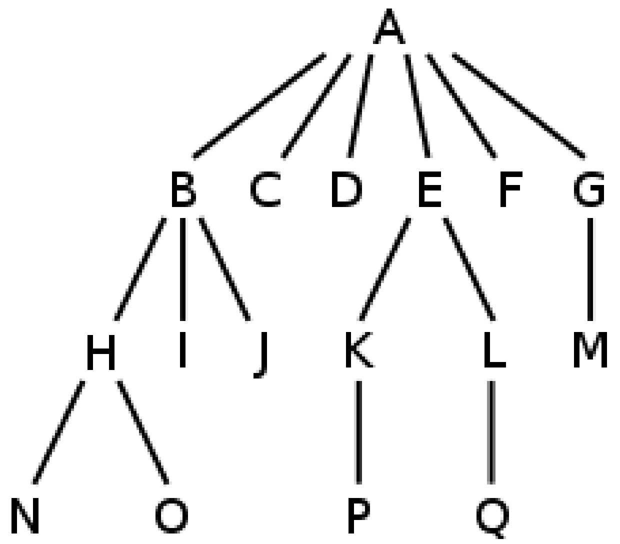
# Rooted tree

Typically, we identify a specific node called the root, and orient the edges in the same direction away from the root.

Any node other than the root has a unique parent.

A node can have multiple children.

Nodes without children are called leaf nodes.



Imagine: http://en.wikipedia.org/wiki/File:N-ary_to_binary.svg

## Trees in computer science

Trees are a natural way of representing hierarchical structures:

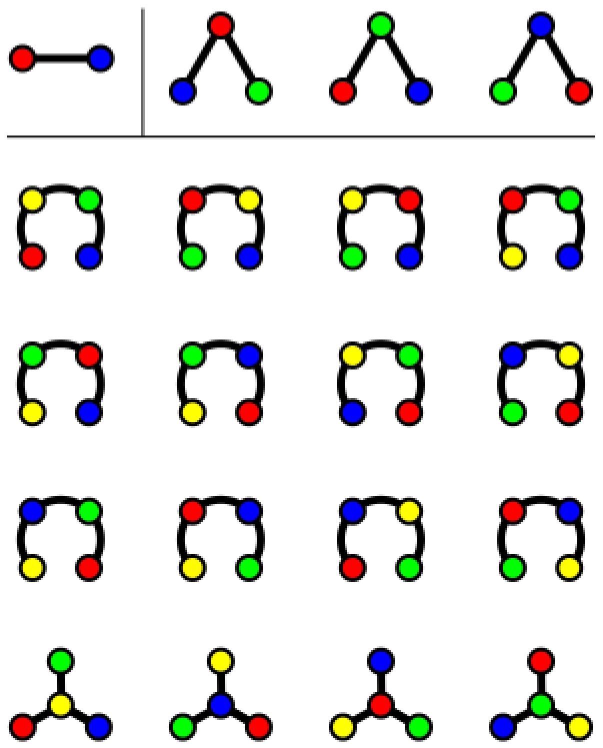
- file system (subtrees are catalogs)
- syntactic tree in a grammar (e.g. expression)
- class hierarchy in object-oriented programming (OOP)
- XML files (elements contain other elements)

# Ordered and unordered trees

The order between children may (e.g. syntax tree) or may not matter

Unordered trees with 2 - 4 nodes - in the figure:

There are $n^{n-2}$ unordered trees with n nodes (Cayley's formula)



Imagine: http://en.wikipedia.org/wiki/File:Cayley’s_formula_2-4.svg

## Trees - recursive structures

A tree is either:

- an empty tree or
- a node with 0 or more subtrees

  ⇒ a list of subtrees (leaves have empty list)

Depending on the problem, nodes contain information

## Binary trees

In a binary tree, each node has at most two children, identified as the left and right child

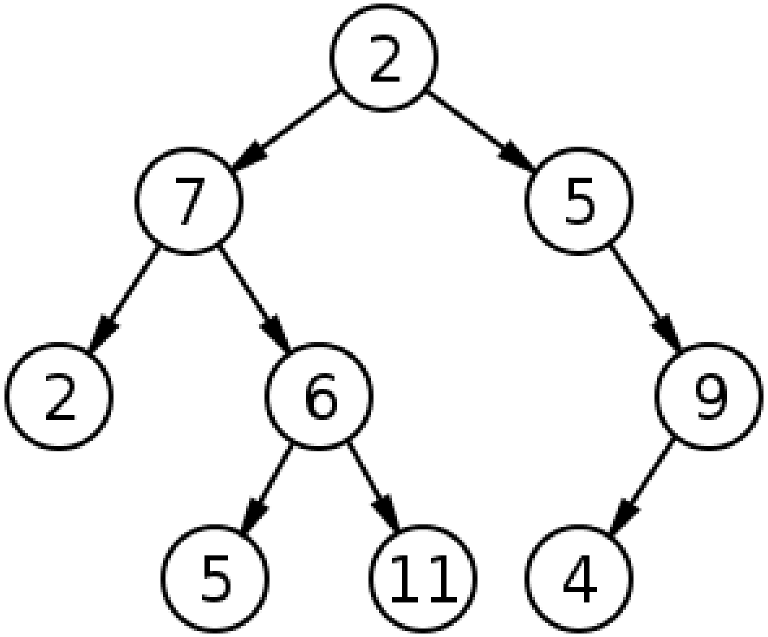
(any/both may be missing)

⇒ a binary tree is:

- empty tree or
- a node with at most two subtrees

# Binary trees

A binary tree of height n has at most $2^{n+1} - 1$ nodes

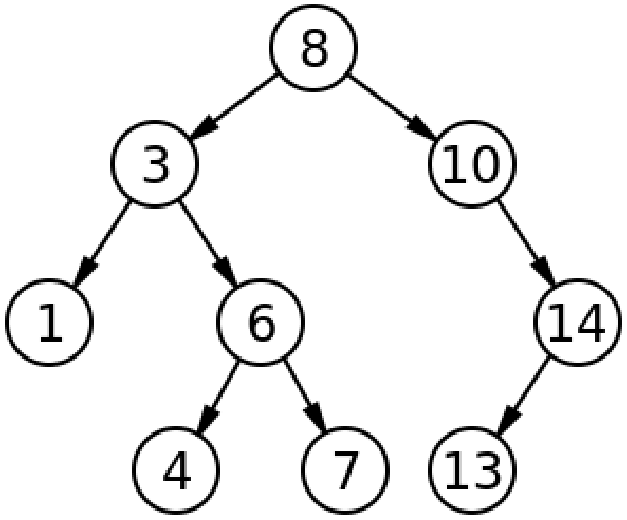


# Binary Search Trees (BST)

Binary search trees are binary trees that store sorted values.

For each node, relative to the value in the root:

- the left subtree has smaller values
- the right subtree has higher values



The search is done recursively, always comparing the searched element with the root of the current subtree:

- if they are equal we have found the element in the tree
- if < the current root, continue the search in the left subtree
- if it is > the current root, continue the search in the right subtree

# Sorting using search trees

Search trees can be used to sort a string of objects that can be ordered.

First create the search tree with the objects in the string:

- the first object will be the root of the tree
- the following objects are placed in the left or right subtree, depending on the value

And then we traverse the search tree in-order (left tree, root, right tree) and we get the ordered string objects.

## Parcurgerea arborilor

în *preordine*: *rădăcina*, subarborele *stâng*, subarborele *drept*

în *inordine*: subarborele *stâng*, *rădăcina*, subarborele *drept*

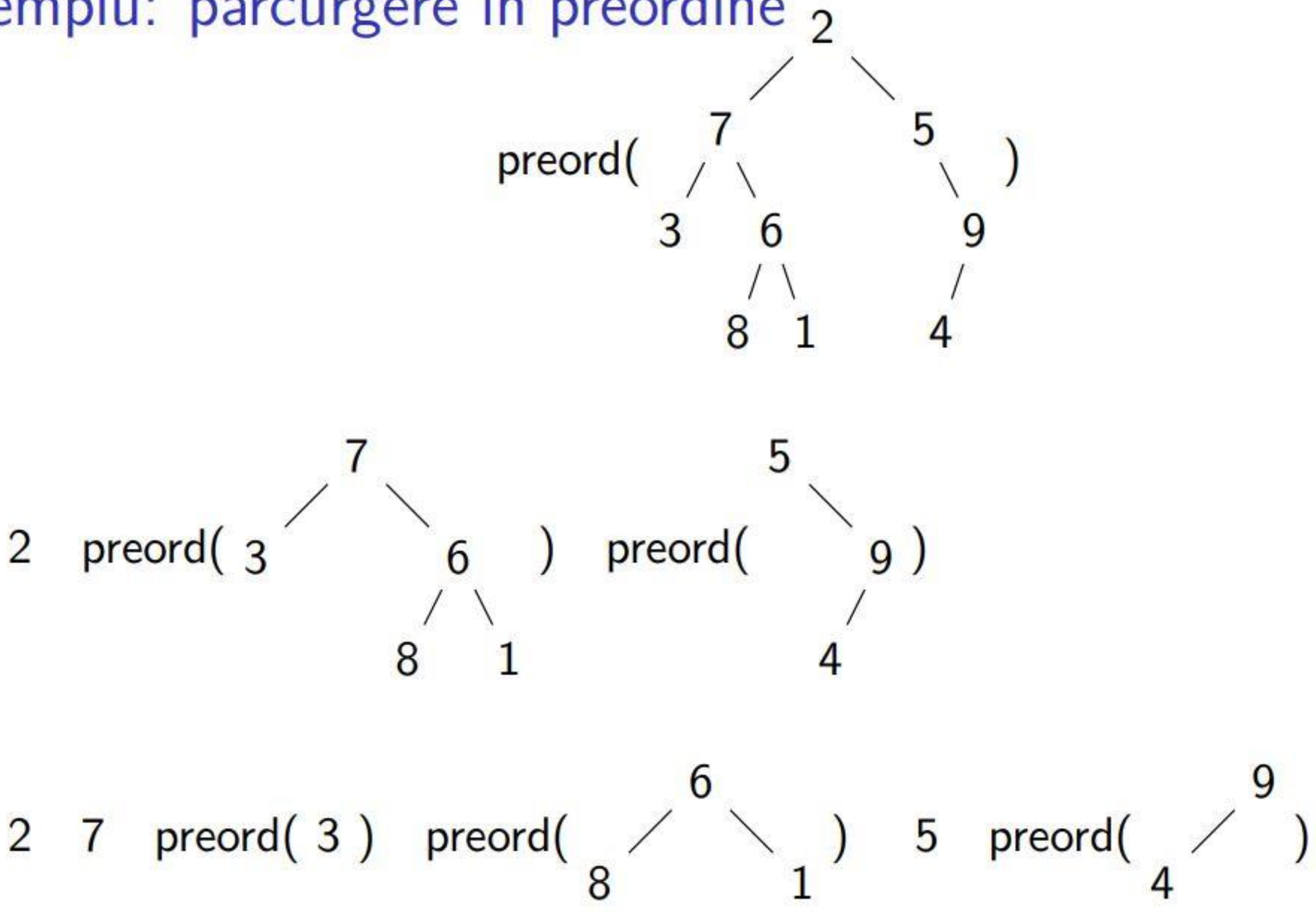
în *postordine*: subarborele *stâng*, subarborele *drept*, *rădăcina*

*subarborii* se parcurg și ei tot în ordinea dată (pre/in/post ordine)!

Pentru expresii, obținem astfel formele prefix, infix și postfix

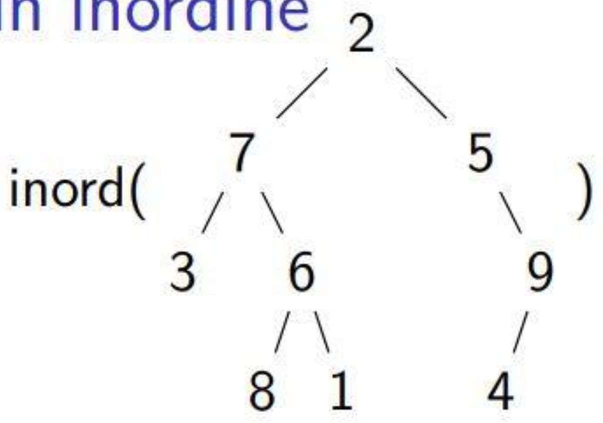
Parcurgerea în pre-/ post-ordine e definită la fel pentru orice arbori (nu doar binari).

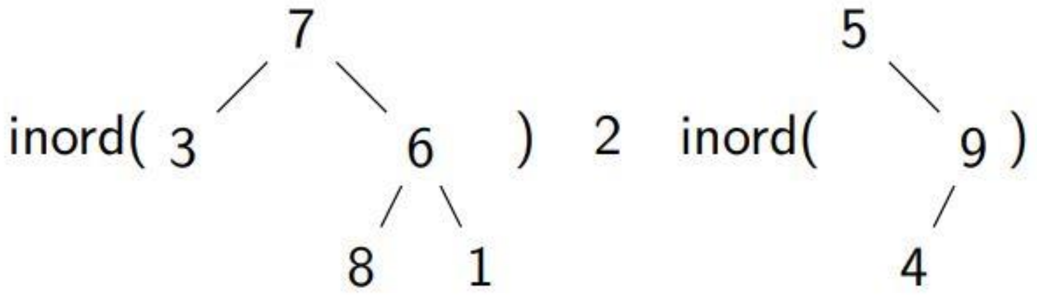
## Exemplu: parcurgere în preordine

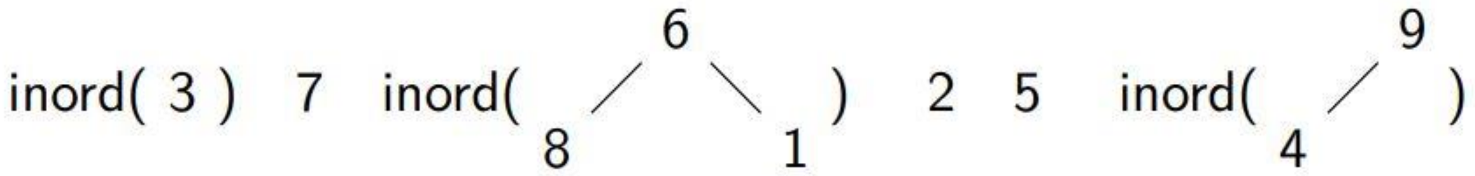

preord( 2 7 5 3 6 9 8 1 4 )
2 preord( 7 3 6 8 1 ) preord( 5 9 4 )
2 7 preord( 3 ) preord( 6 8 1 ) 5 preord( 9 4 )

2 7 3 6 8 1 5 9 4

## Exemplu: parcurgere în inordine

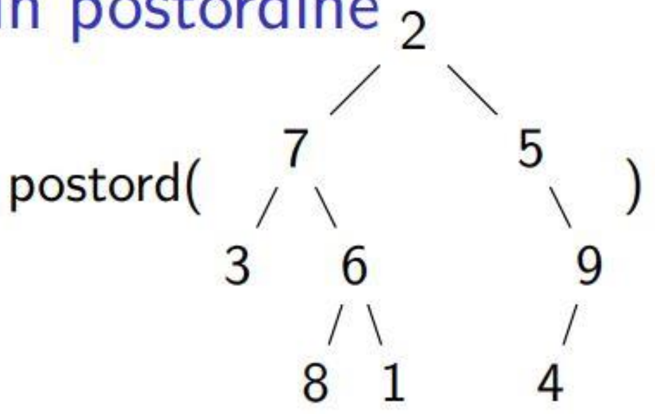


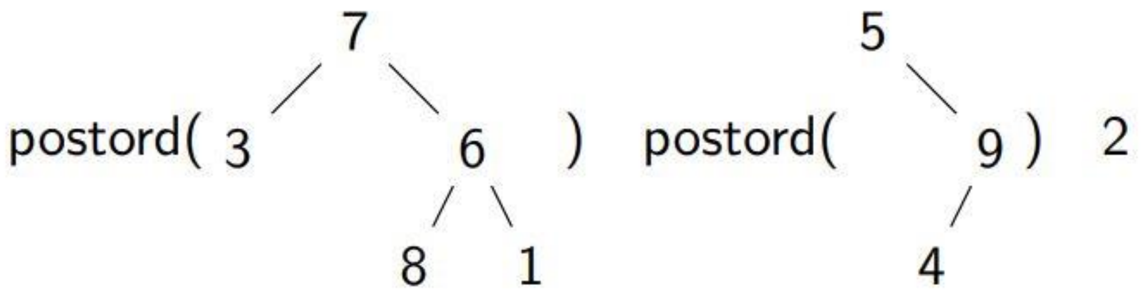


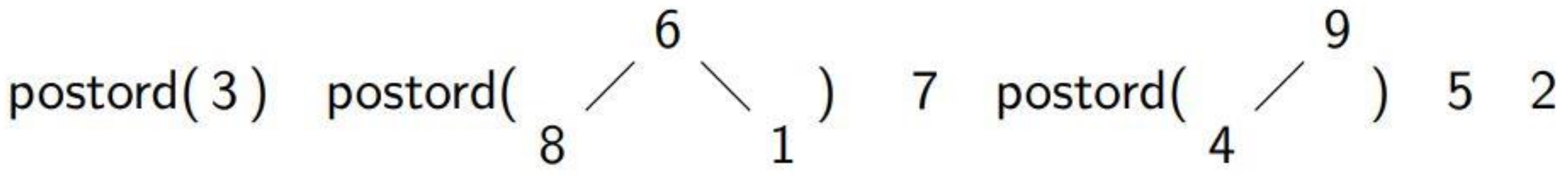


3 7 8 6 1 2 5 4 9

## Exemplu: parcurgere în postordine







3 8 1 6 7 4 9 5 2

# Representation of a tree

To represent a tree, for each node we will have a dictionary that will contain two pairs: the value of the node and the list of values of its children.

The tree will be represented by a list containing all its nodes in the form:

One node:

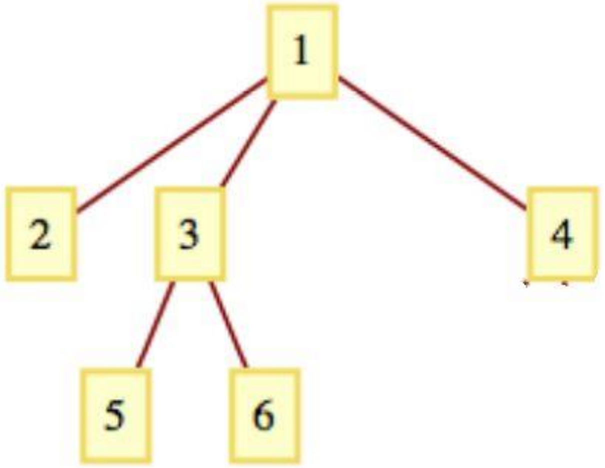
{"valore" : None, "copii" : []}

The tree:

[{"valore" : None, "copii" : […]}, …]

## Representation of a tree - example

```
a_tree= [
    {"value" : 1, "children" : [2, 3, 4]},
    {"value" : 2, "children" : []},
    {"value" : 3, "children" : [5, 6]},
    {"value" : 4, "children" : []},
    {"value" : 5, "children" : []},
]
```



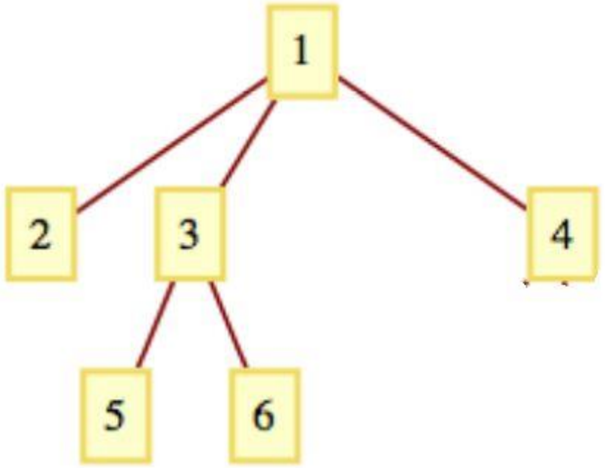
## Representation of a tree

Another way to represent the tree is that the list of children of a node directly holds the information as a dictionary list, not just as a list of values.

In this way we make use of the recursive structure of a tree.

# Representation of a tree - example

```
a_tree = { "value": 1, "children":
          [
              { "value": 2, "children": []},
              { "value": 3, "children":
                  [
                      { "value" :5, "children": []},
                      { "value" :6, "children": []}
                  ]
              },
              { "value": 4, "children": []}
          ]
      }
```

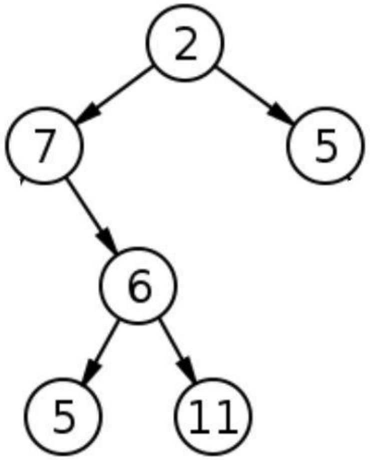


# Representation of a binary tree

A binary tree can be represented recursively as a dictionary with 3 pairs: value, left tree and right tree.

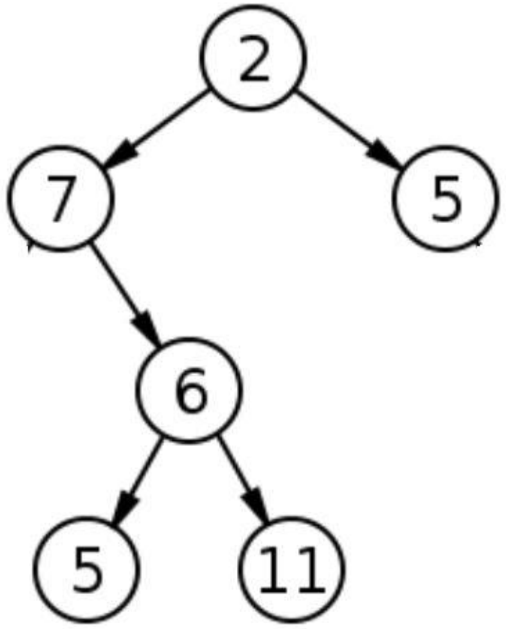
*tree = {"value": None, "left": None, "right": None}*

```
tree2 = { "value" : 2, "left":
        {
            "value": 7, "left": None, "right":
            {
                "value": 6, "left":
                {
                    "value": 5, "left": None, "right": None
                }, "right":
                {
                    "value":11, "left": None, "right": None
                },
            },
        }, "right":
            {
                "value": 5, "left": None, "right": None
            }
    }
```



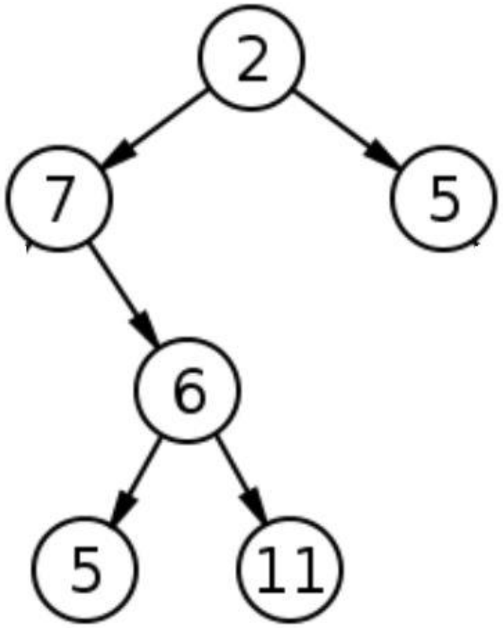
# Preorder traversal

```
def rsd(tree):
    if (tree != None):
        return [tree["value"]] + rsd(tree["left"]) + rsd(tree["right"])
    else:
        return []


print(rsd(binary_tree))
# [2, 7, 6, 5, 11, 5]
```



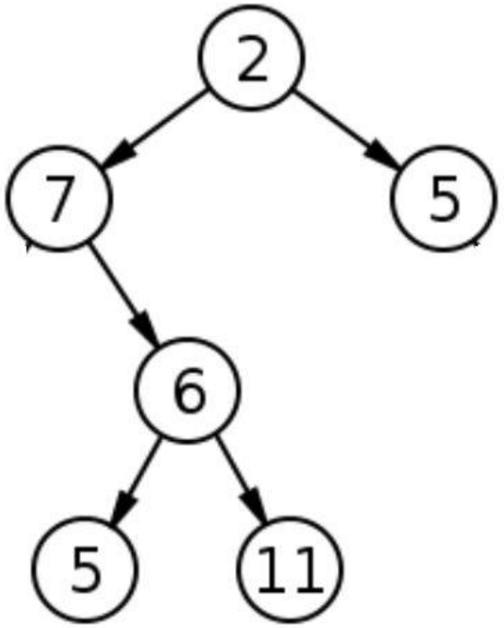
# Inorder traversal

```
def srd(tree):
    if (tree != None):
        return  srd(tree["left"]) + [tree["value"]] + srd(tree["right"])
    else:
        return []


print(srd(binary_tree))
# [7, 5, 6, 11, 2, 5]
```



# Postorder traversal

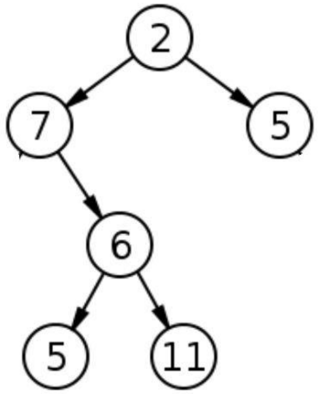
```
def sdr(tree):
    if (tree != None):
        return  sdr(tree["left"]) + sdr(tree["right"]) + [tree["value"]]
    else:
        return []


print(sdr(binary_tree))
#[5, 11, 6, 7, 5, 2]
```



# Adding a new node

Adding a new node to a parent and a specific position:

```
def adaugare_nod_pozitie(parinte, nod_nou, pozitie):
    if (parinte[pozitie] == None):
        parinte[pozitie] = nod_nou
    return parinte
```

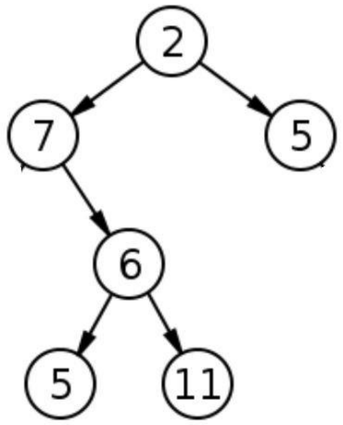


```
binary_tree["left"]=adaugare_nod_pozitie(binary_tree["left"],
{"value": 100, "left": None, "right": None}, "left")
print(rsd(binary_tree))
#[2, 7, 100, 6, 5, 11, 5]
```

## Adding a new node

Adding a new node to the binary search tree:

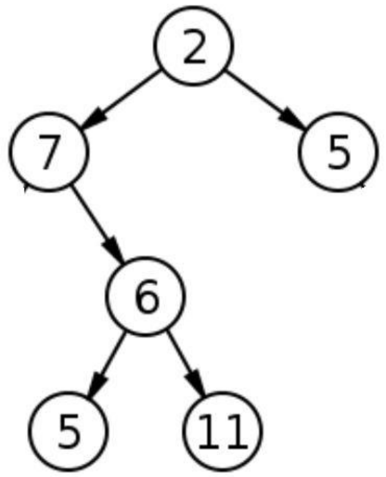
```
 def adaugare_nod(tree, nod_nou):
    if (tree == None):
        return nod_nou
    if (nod_nou
["value"]<tree["value"]):
        tree["left"] = adaugare_nod(tree["left"], nod_nou)
    else:
        tree["right"] = adaugare_nod(tree["right"], nod_nou)
    return tree

print(rsd(adaugare_nod(binary_tree,{"value": 1, "left": None,
"right": None})))
#[2, 7, 1, 6, 5, 11, 5]
```



# Deleting a node/sub-tree

Deleting a node (or subtree) from a given parent as a parameter:

```
def stergere_nod(parinte, valoare_nod):
    if (parinte["left"]["value"] == valoare_nod):
        parinte["left"] = None
    elif(parinte["right"]["value"] == valoare_nod):
        parinte["right"] = None

stergere_nod(binary_tree, 5)
print(rsd(binary_tree))
#[2, 7, 6, 5, 11]
```



## Tuples in PYTHON

A tuple is a collection of predefined data in PYTHON (in addition to lists, sets and dictionaries).

A tuple is an ordered collection of data and cannot be changed after creation.

A tuple is written in round brackets:

*tuple = (2, 5, 7, 1, 5)*

# Tuples in PYTHON

*The elements of a tuple:*

- *are ordered (can be accessed by positive or negative index)*
- *cannot be changed after creation*
- *allow duplicates*

# Tuples in PYTHON

The number of elements in the tuple can be found with the len() function:

```
a = (1, 6, 8)
print(len(a)) # 3
```

To create a single-element tuple you need to put round brackets and a comma at the end:

```
tuple = (5,)
```

# Tuples in PYTHON

We can also create a tuple with the tuple() constructor:

```
a = tuple((4, 6, 8))
b = tuple([“Arad”, “Timisoara”])
```

Elements are accessed by indexes:

```
print(a[0])                    # 4
print(b[1])                    # Timisoara
print(b[-2])                   # Arad
print(a[1:3])                  # (6, 8)
print(9 in a)                  # False
print(8 in a)                  # True
```

## Tuples in PYTHON

No elements can be added to the tuple and no elements can be deleted after its creation.

These operations are allowed when working with lists.

If we want to create a new tuple with different elements we can transform it into a list and process it:

```
a = (3, 5, 7, 3)
L = list(a)
#processing the list L
```

## Tuples in PYTHON

We can extract elements from the tuple and then process them independently:

```
t = (3, 3, 6, 8)
a, b, c, d = t
print(a)                # 3
print(b)                #3
print(c)                #6
```

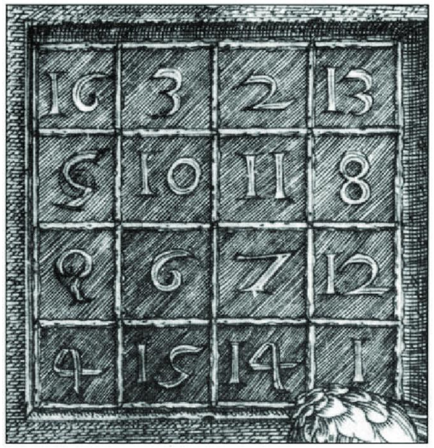
If the number of elements in the tuple is bigger, it is mandatory to use an asterisk at the last object:

```
a, b, *c = t            # c = (6, 8)
```

# Tuples in PYTHON

We can create a new tuple with elements from 2 or more other tuples:

```
a = (1, 2, 3)
b = (“a”, “b”, “c”)
c = a + b

print(c)                    # (1, 2, 3, “a”, “b”, “c”)
```



Thank you!

## Bibliography

- The content of the course is mainly based on the material from the LSD course taught by Prof. Dr. Eng. Marius Minea and S.l. Dr. Eng. Casandra Holotescu (http://staff.cs.upt.ro/~marius/curs/lsd/index.html)