Data Structures and Algorithms (DSA) Course 8

Lists / Graph Theory / Trees

Iulian Năstac

Recapitulation

- It is considered the following type:

```
typedef struct nod
        { <statements>;
          struct nod *next;
        } NOD;
```

Circular simple linked list
(Recapitulation)

A simple linked list contains:

first – the pointer to the node that has no predecessor;

last – the pointer to the node that has no successor.

We know that: **last -> next = 0**;

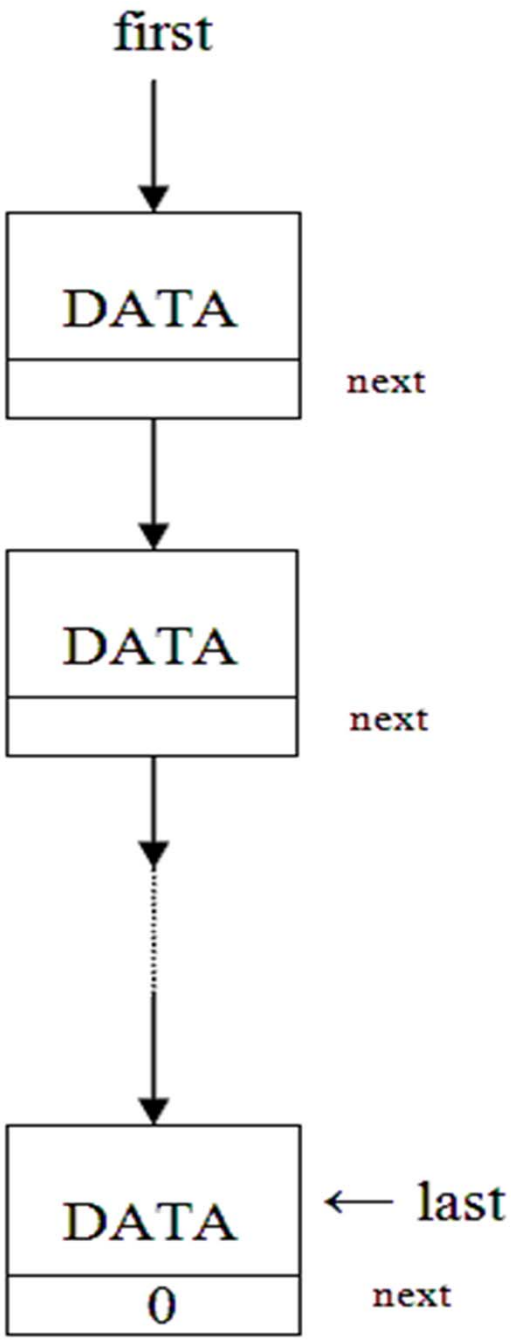
But if **last -> next = first** , then we can obtain a **circular linked list**.

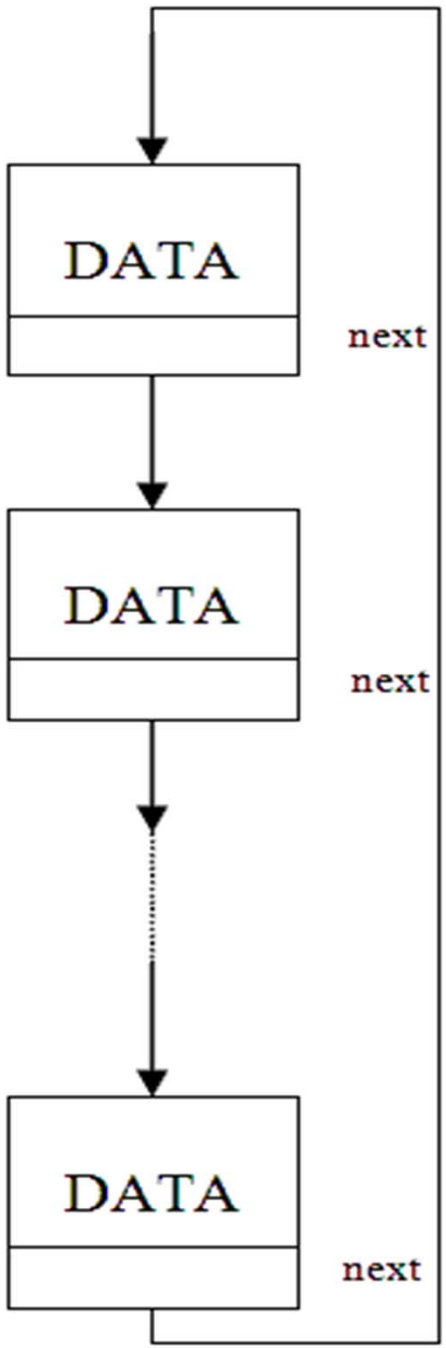
(Recapitulation)

For the management of the nodes, it is used a global variable (denoted **pcirc**) that points to an arbitrary node of the list:

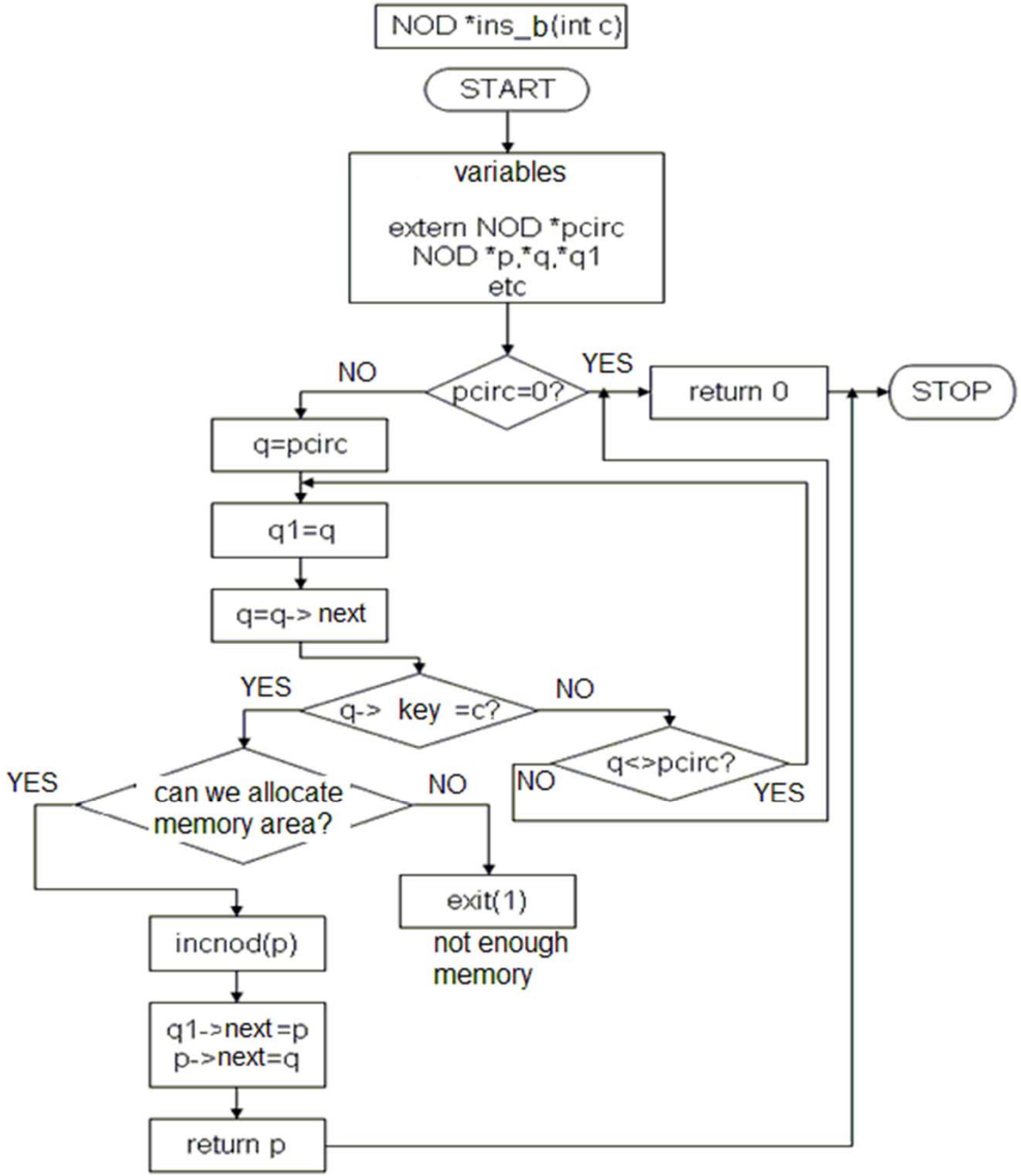
NOD *pcirc;

where NOD is the common type of the list nodes

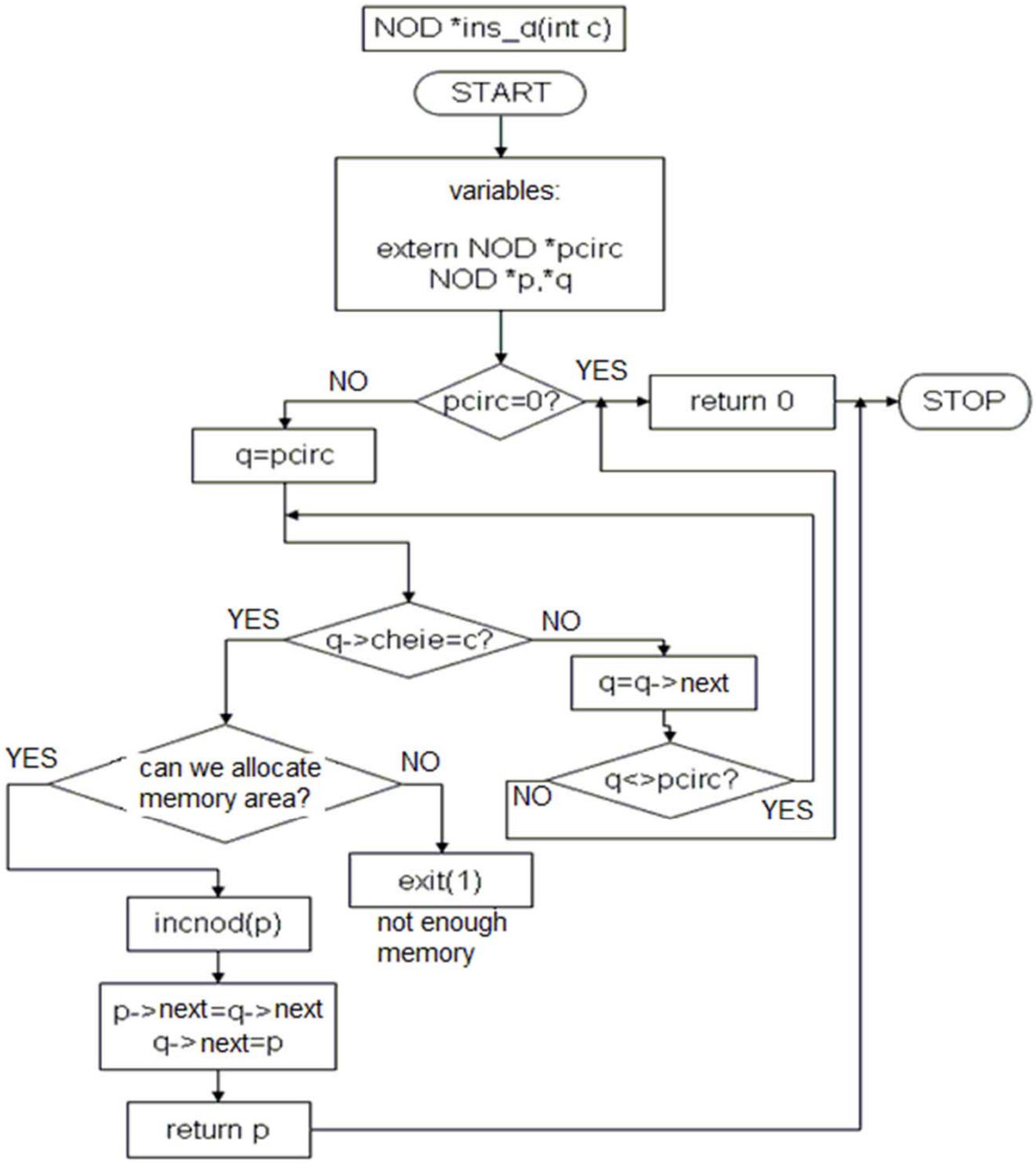

first
DATA
next
DATA
next
DATA
last
0
next


DATA
next
DATA
next
DATA
next

Inserting before a node specified by a key



Inserting after a node specified by a key



+ Finding a node by counting!…

Josephus problem

(variant for a program in C)

The steps of the algorithm are:

1) The numbering starts with node immediately following the **pcirc** pointer and delete the ***n***-th node from the list.

2) Repeat step 1, continuing with the node immediately following the deleted one, until the list is reduced to a single node.

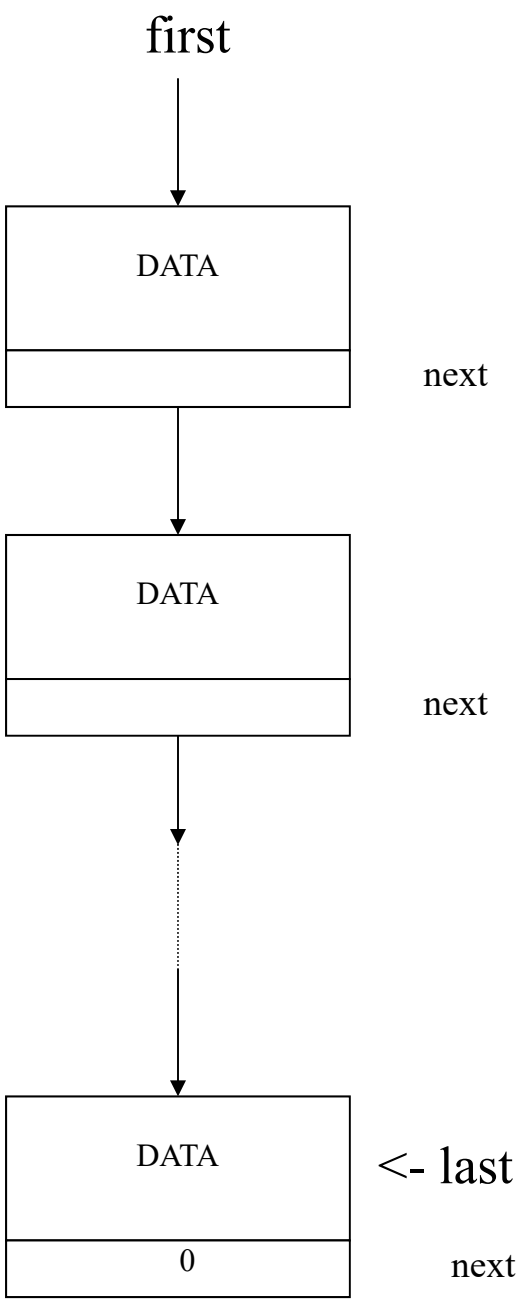
Doubly linked list

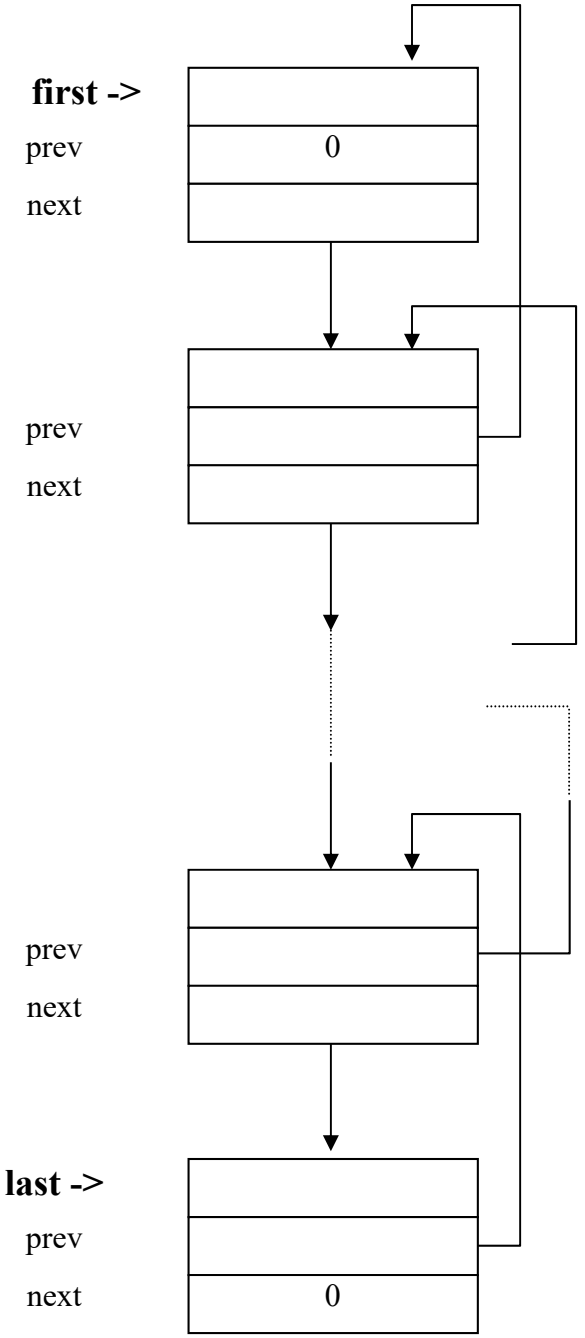
- A doubly-linked list is a linked data structure that consists of a set of sequentially linked nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' **previous** and **next** links, respectively, point to some kind of terminator (typically **null**).

Notes:

- It is considered the following type:

```
typedef struct nod
        { <statements>;
          struct nod *prev;
          struct nod *next;
        } NOD;
```


first
DATA
next
DATA
next
DATA
<- last
0
next


first ->
prev
0
next
prev
next
prev
next
last ->
prev
next
0

```
first -> prev = 0;
```

```
last -> next = 0;
```

Operations related to a doubly linked list:

a) creation of a doubly linked list;

b) access to any node of the list;

c) inserting a node in a doubly linked list;

d) deleting a node from a doubly linked list;

e) deleting a doubly linked list.

Creation of a doubly linked list

1. At first, the doubly linked list is empty :

   first = last = 0;

2. A memory area is allocated (with **malloc**) in the heap memory for the current node.

3. Are there data to upload them in the current node **p**?

- NO → returns from function (after using **elibnod(p)**);
- YES → loading node with current data (by using **incnod(p)**) and jump to step 4;

4. If the list:

• is empty:

```
first = last = p;
p->prev = p->next = 0;
```

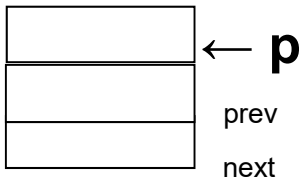
•is not empty:

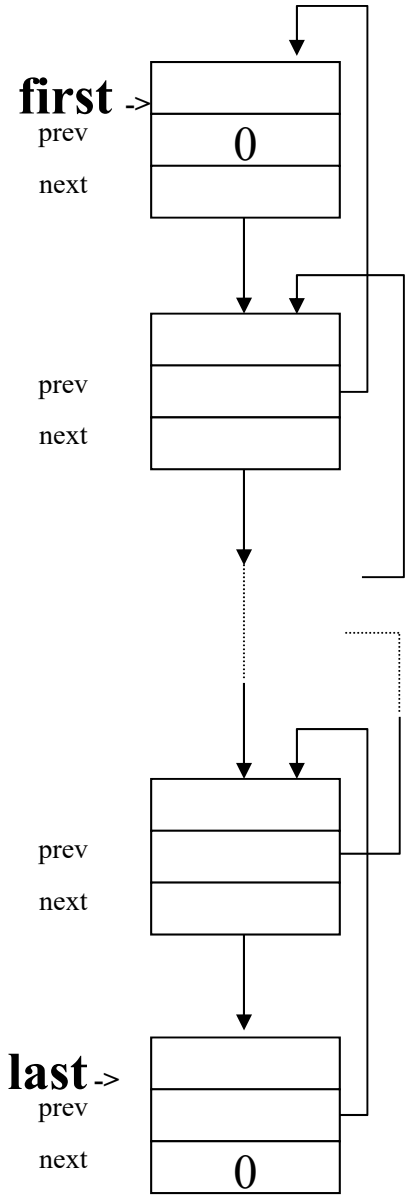
```
last->next = p;
p->prev = last;
p->next = 0;
last = p;
```

Current node is inserted after the one pointed by **last**. Afterwards **last** will indicate to the new node, which was inserted.

6) Jump to step 2.

Adding a new node






```
last->next = p;
p->prev = last;
p->next = 0;
last = p;
```

Access to a node of a doubly linked list

Usually, we prefer an **access** function that searches a node, by using a numerical key:

```
typedef struct nod
                    { <statements>;
                      type key; /* like integer */
                      struct  nod *prev;
                      struct  nod *next;
                    } NOD;
```

Inserting a node in a doubly linked list

1. Inserting before the first node
2. Inserting before a node specified by a key
3. Inserting after a node specified by a key
4. Inserting after the last node

Deleting a node from a doubly linked list

a) deleting the first node of the doubly linked list;

b) deleting a specified node with a key;

c) deleting the last node of the doubly linked list.

The doubly linked lists have various applications:

- High security programs - the rapid restoration of missed links (without loss of nodes);
- Long lists, where the search can start from both ends;
- Banking programs, etc.

Graph theory

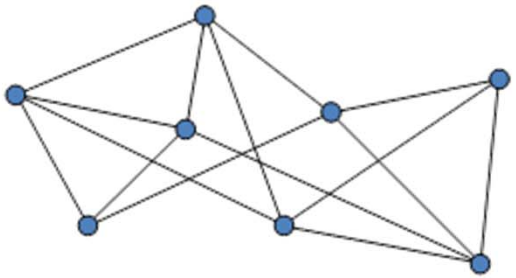
Definition:

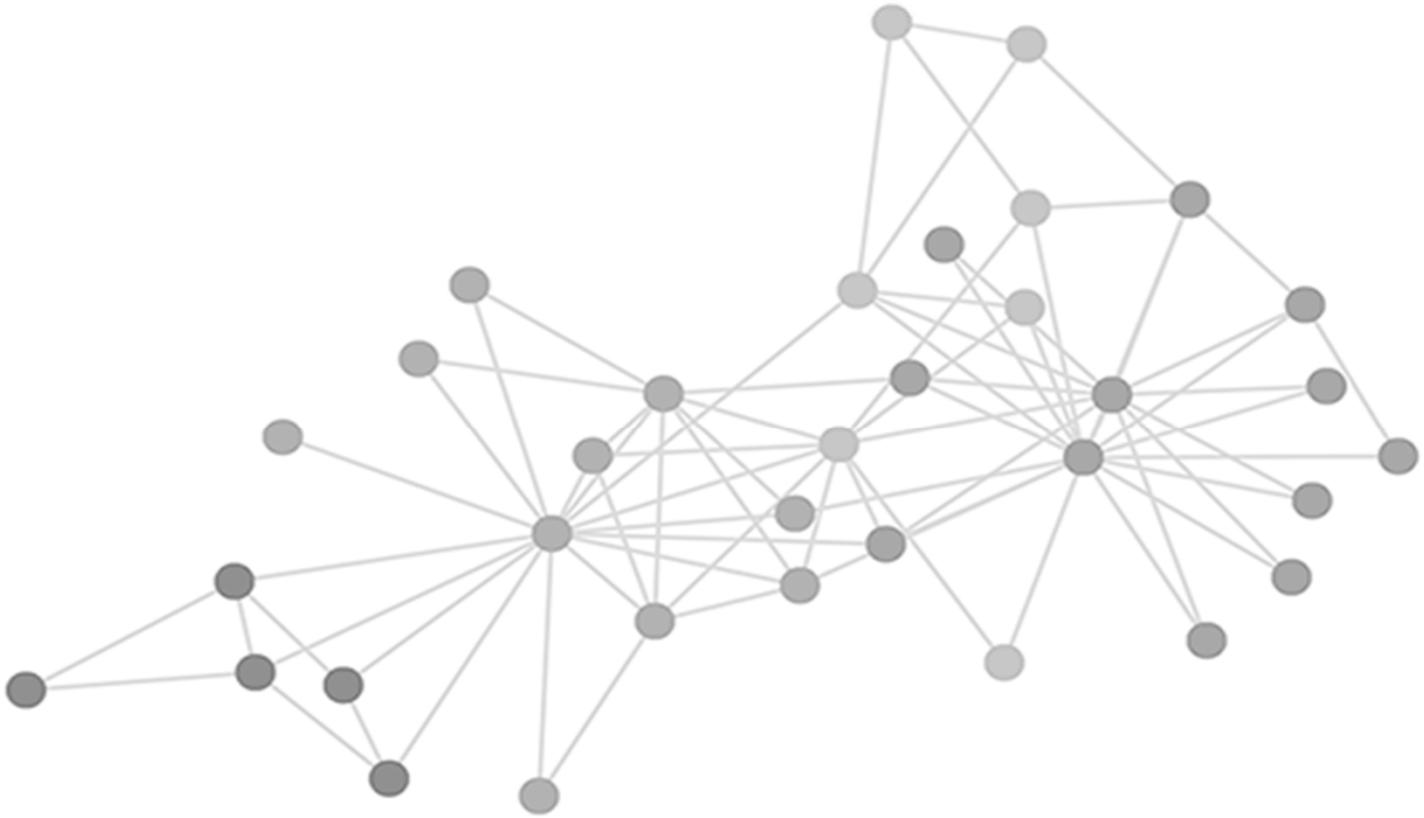
Usually a graph is a pair like:

$$G = \langle V, M \rangle$$

where V is a set of vertices (or nodes), and $M \subseteq V \times V$ is a set of edges (lines or links).

Examples





- A graph may be **undirected**, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be **directed** from one vertex to another

Usually, the line from the node **a** to the node **b** is denoted with:

- ordered pair **(a, b)** if the graph is directed;

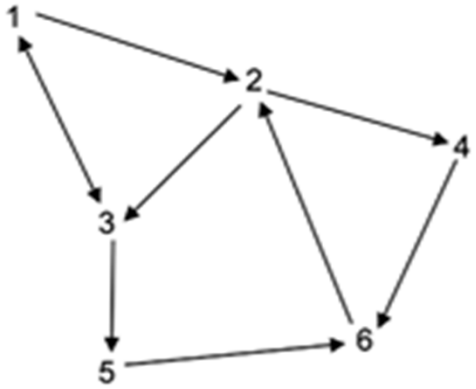
- unordered pair **{a, b}** if the graph is undirected.

In practical applications we can find different kinds of graphs:

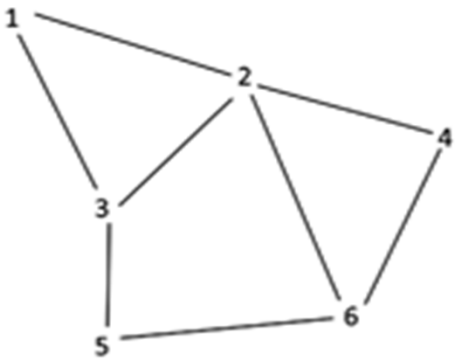
- directed,

- undirected,

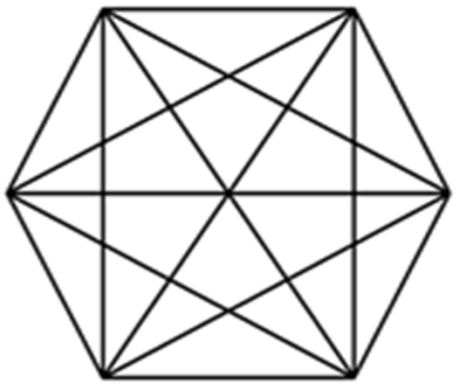
- mixed.



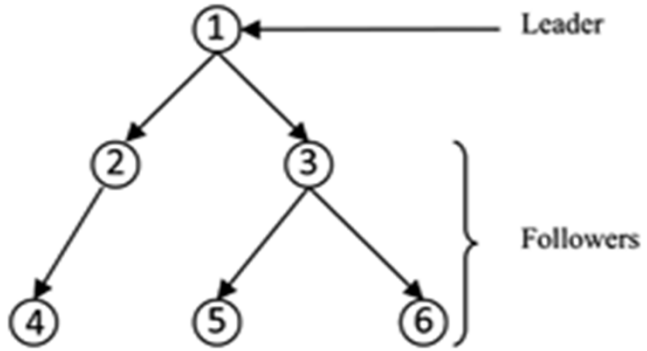
a Directed graph.



b Undirected graph

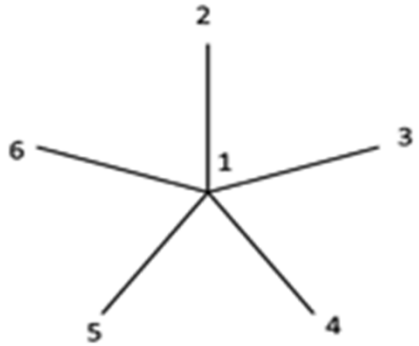


c Complete graph K^6

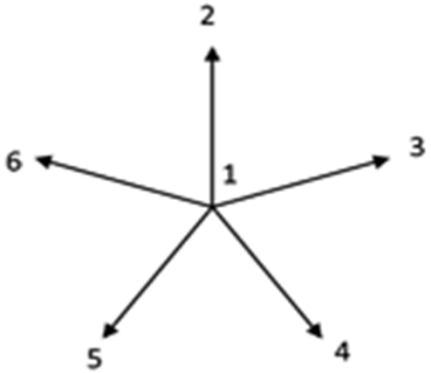


Directed tree graph (formation graph)

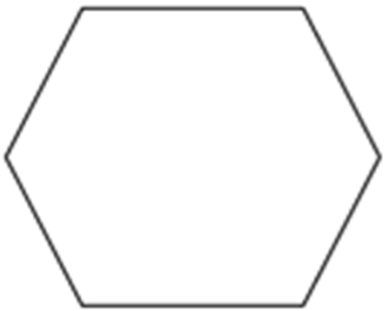
d



e Undirected star $K_{1,5}$

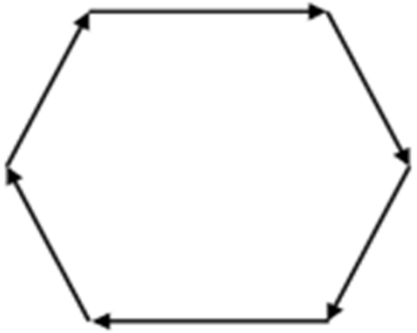


f Directed star

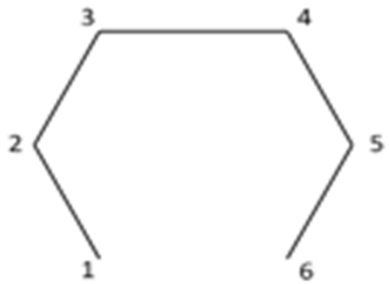


Undirected 6-cycle.
(2-regular graph)

g



h Directed 6-cycle
(6-periodic graph)



i Undirected path P^5

A **path** (route) is a sequence of edges of the following forms :

- $(a_1, a_2), (a_2, a_3), (a_3, a_4), \ldots , (a_{n-1}, a_n)$ if the graph is directed

- $\{a_1, a_2\}, \{a_2, a_3\}, \{a_3, a_4\}, \ldots , \{a_{n-1}, a_n\}$ if the graph is undirected

Definitions

- The length of a path = the number of edges.
- A simple path = a path in which the peaks are not repeated.
- Cycle = is a simple path except the first and last peak, which are the same.
- Directed acyclic graph = is a directed graph with no directed cycles.

We call a subgraph **G'** (of the graph **G**):

$$\mathbf{G' = <V', M'>}$$

where $\mathbf{V' \subseteq V}$, and $\mathbf{M' \subseteq M}$ (the vertices are a subset of the vertex set of **G**, and the edges are a subset of the initial edge set).

A **partial graph G"** spans a initial graph G, and usually it has the same vertex set, but a diminished number of edges.

$$\mathbf{G'' = <V, M''>}$$

$\mathbf{M'' \subseteq M}$ (but **G"** has the same vertex set **V**).

Connectivity

- If it is possible to establish a path from any vertex to any other vertex of a graph, the graph is said to be **connected**; otherwise, the graph is **disconnected**.
- A graph is **totally disconnected** if there is no path connecting any pair of vertices (this is just another name to describe an empty graph or independent set).

Representations

Different data structures for the representation of graphs are used in practice:

- **Adjacency list**
- **Adjacency matrix**
- **Incidence matrix**

Adjacency list

- Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices.
- This data structure allows the storage of additional data on the vertices.
- Additional data can be stored if edges are also stored as objects, in which case each vertex stores its incident edges and each edge stores its incident vertices.

Adjacency matrix

- A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices.
- Data on edges and vertices must be stored externally.
- Only the cost for one edge can be stored between each pair of vertices.

Incidence matrix

- A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges.
- The entries indicate whether the vertex at a row is incident to the edge at a column.

TREES

Tree structure

- A tree structure or tree diagram is a way of representing the hierarchical nature of a structure in a graphical form.

- It is named a "tree structure" because the classic representation resembles a tree, even though the chart is generally upside down compared to an actual tree, with the "root" at the top and the "leaves" at the bottom.

- Definition 1:

  A tree is a directed graph, which has an acyclic structure and it is connected (from the root to every terminal node - or leaves).

- Definition 2:

  A **tree** is somehow similar with a list, being a collection of recursive data structures that has a dynamic nature.

- **Definition 3**:

By tree we understand a finite and non-empty group of elements called nodes:

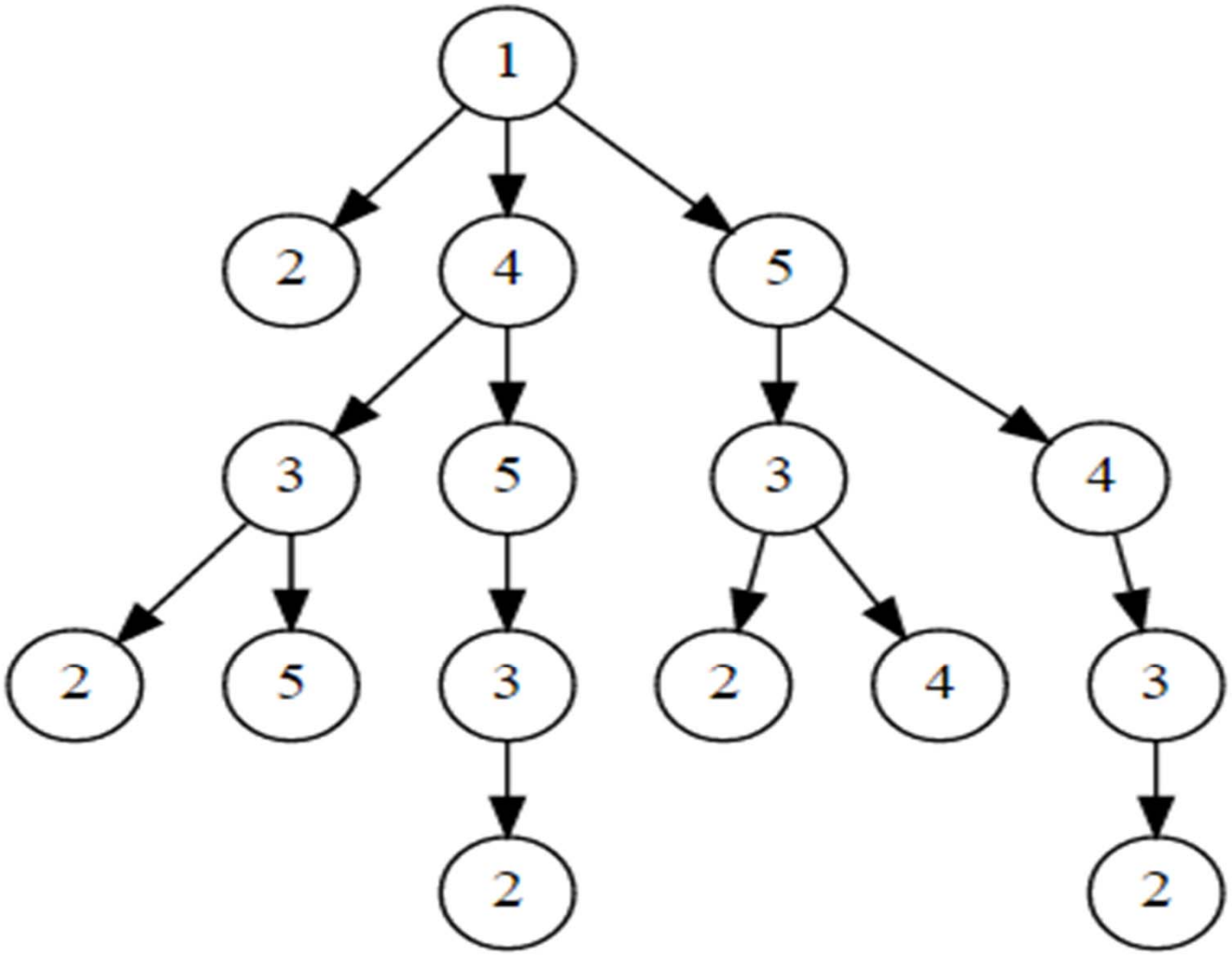
TREE = {A1, A2, A3, ..., An}, where n> 0,

which has the following properties:

- there is only one node, which is called the **root** of the tree;

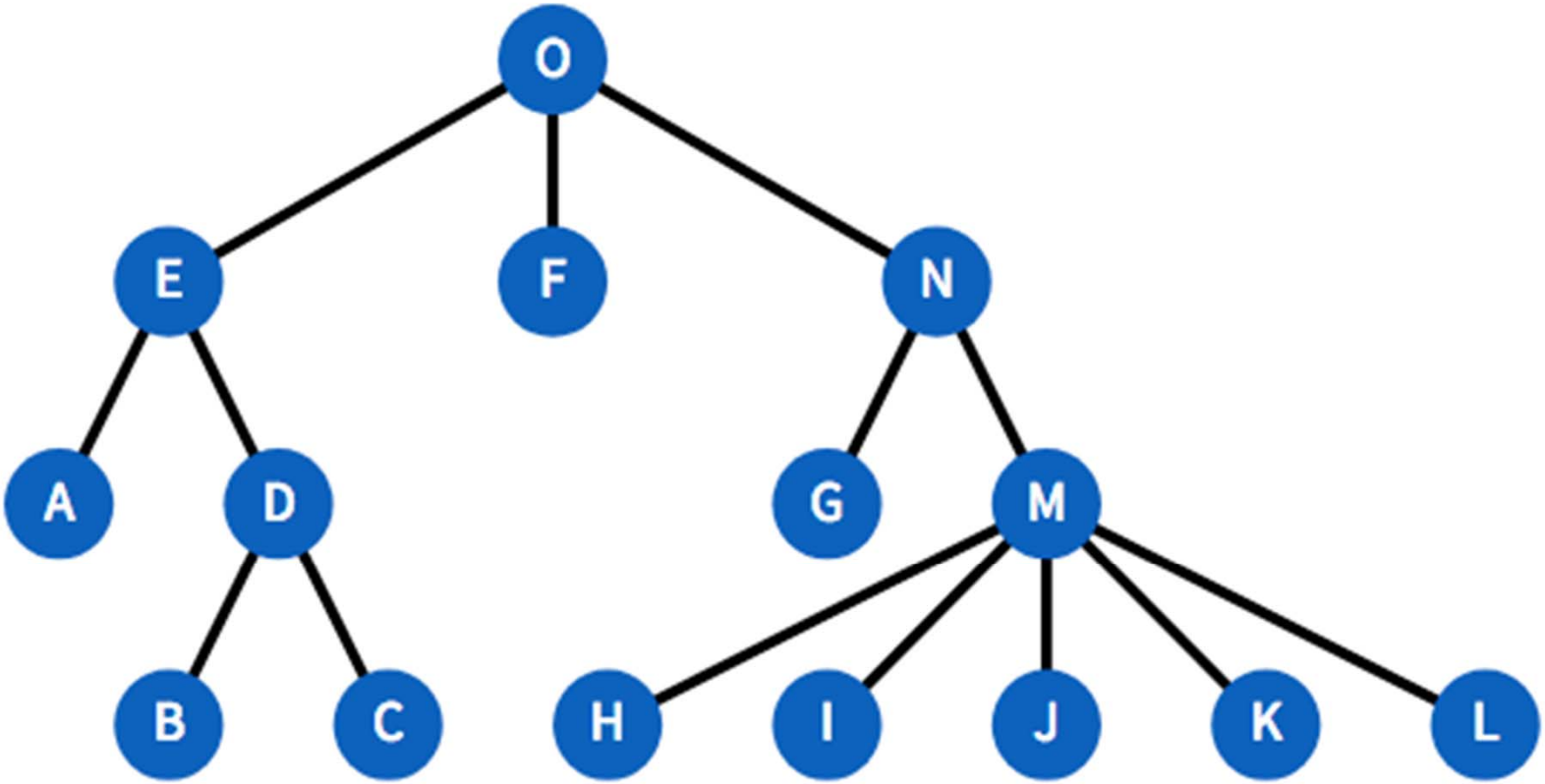
- the rest of the nodes can be grouped in **subsets** of the initial tree, which also form trees. Those trees are called **subtrees** of the root.

Example of a tree as a particular graph



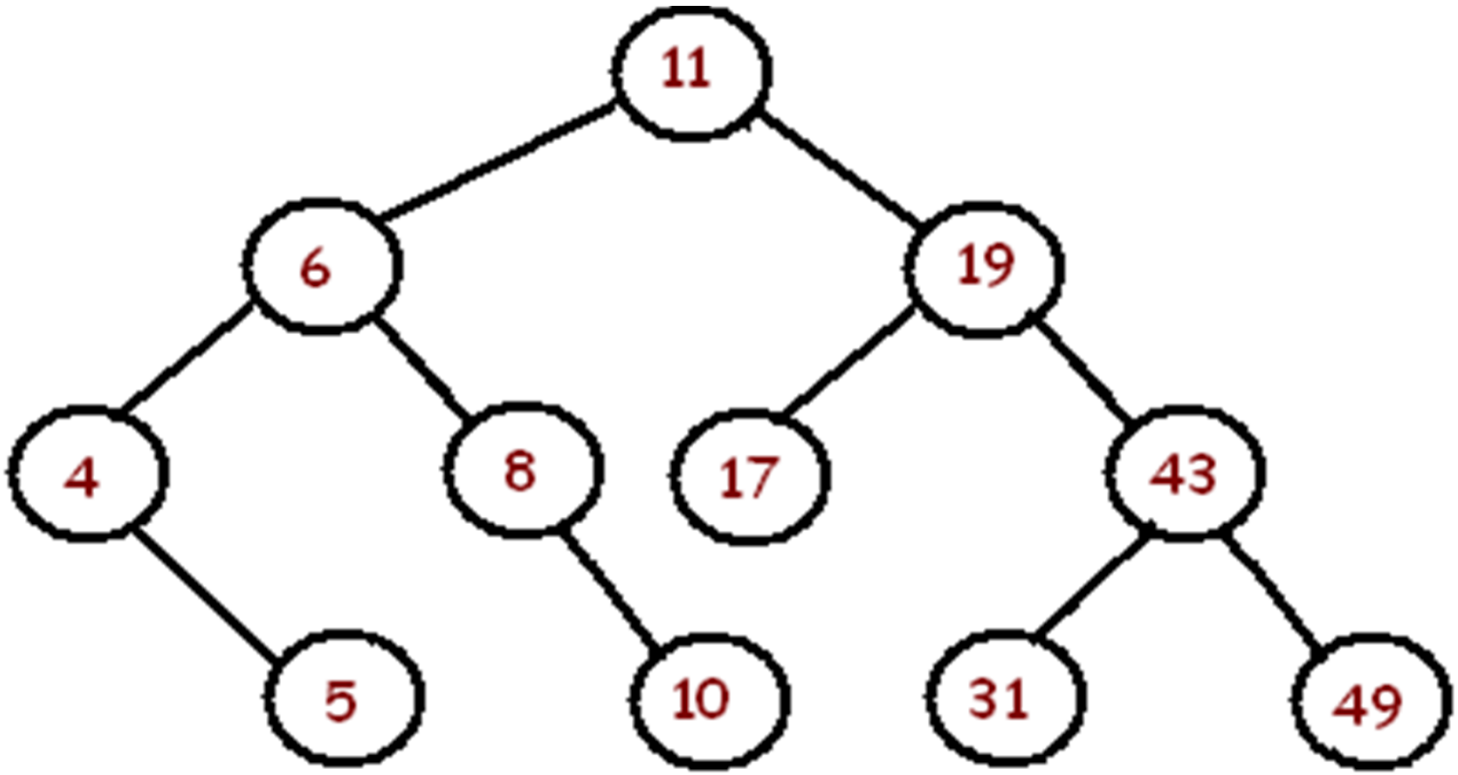
The nodes in a tree

- Each node in a tree has zero or more child nodes, which are below it in the tree (trees are usually drawn growing downwards).
- A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent.
- An internal node (also known as an inner node) is any node of a tree that has child nodes. Similarly, a terminal node (also known as a leaf node) is any node that does not have child nodes.
- The topmost node in a tree is called the root node.


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

Ordering

- An **ordered tree** is a rooted tree for which an ordering is specified for the children of each vertex (node).


11
6
19
4
8
17
43
5
10
31
49

Binary tree

- A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

- There are maximum two disjoint groups for every parental node (each one being a binary tree).

Notes:

- Usually, one of the groups is called **the left subtree** of the root, and the other one **the right subtree**.
- The binary tree is ordered, because in each node, the left subtree is considered to precede the right subtree.
- In other words, we can say that the left descendant is older that the right one.

Notes (cont.)

- Sometimes, a node of a binary tree can have only one descendant. This can be the left subtree or the right subtree.

- The two possibilities are considered distinct.

Transformation

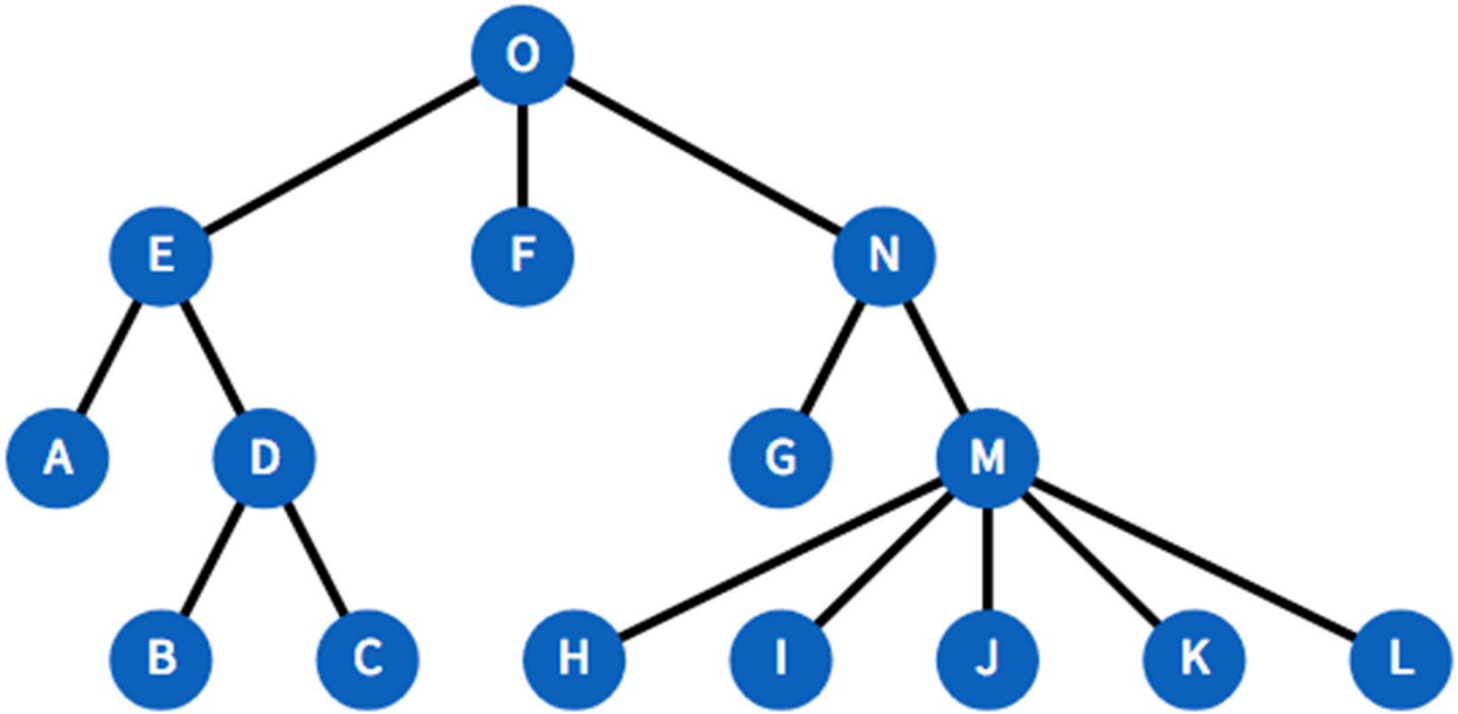
- A binary tree it cannot be defined as a particular case of an ordered tree. Usually, a classic tree is never empty, while a binary tree can be empty, sometimes.

- Any ordered tree can be always represented through a binary tree.

Conversion of an classic ordered tree in a binary tree

1. Firstly, we have to link, between them, all brothers descendants of the same parent node and suppress the links with the parent, except the first son.
2. Then, the former prime son node (the older one) becomes the left son of the parent, and the other former brothers become (in a sequentially way) the roots of the right subtrees. Each of the brothers becomes downward the right son of its former big brother.

Try to turn this tree into a binary tree



Using structures for building a binary tree

The node of a binary tree can be represented as another structural data type, called NOD, which is defined as follows:

```
typedef struct node
{
        <statements>
        struct node * left;
        struct node * right;
} NOD;
```

where:

- **left** - is the pointer to the left son of the current node;
- **right** - is the pointer to the right son of the same node.

In applications with binary trees we can define several operations such as:

1. Inserting a leaf node in a binary tree;
2. Access to a node of a tree;
3. Traversal the tree;
4. Delete a tree.

- The operations of insertion and access to a node are based on a **criteria** that defines the place in the tree where the node in question can be inserted or found (according with the current operation which is involved).
- This **criterion** is dependent on the specific problem where the binary tree concept is applied.