Data Structures and Algorithms (DSA) Course 9 Trees

Iulian Năstac

Doubly linked list

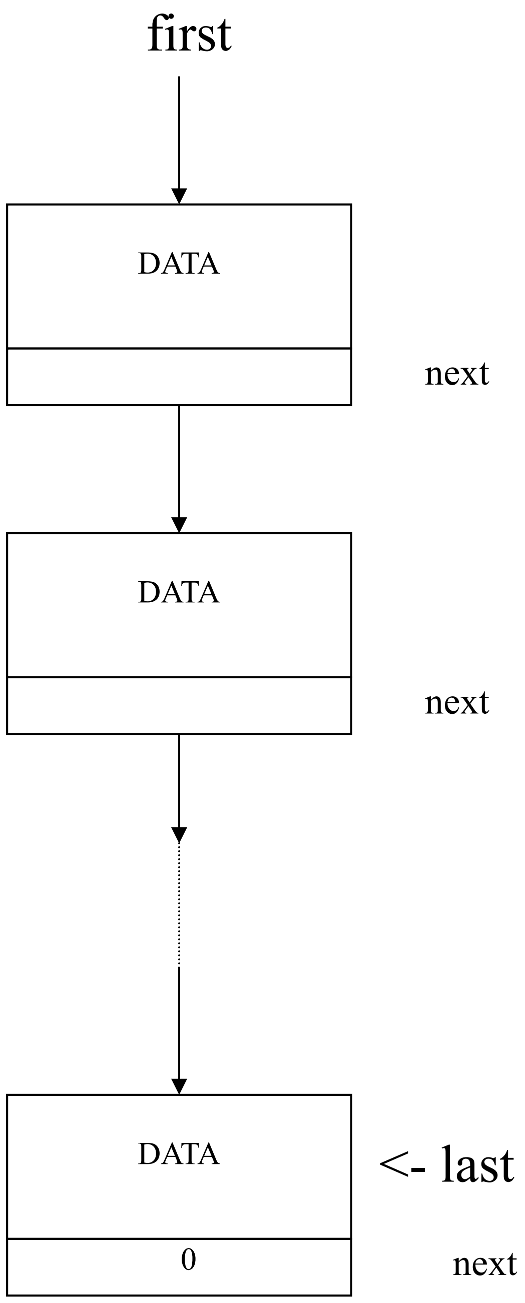
Recapitulation

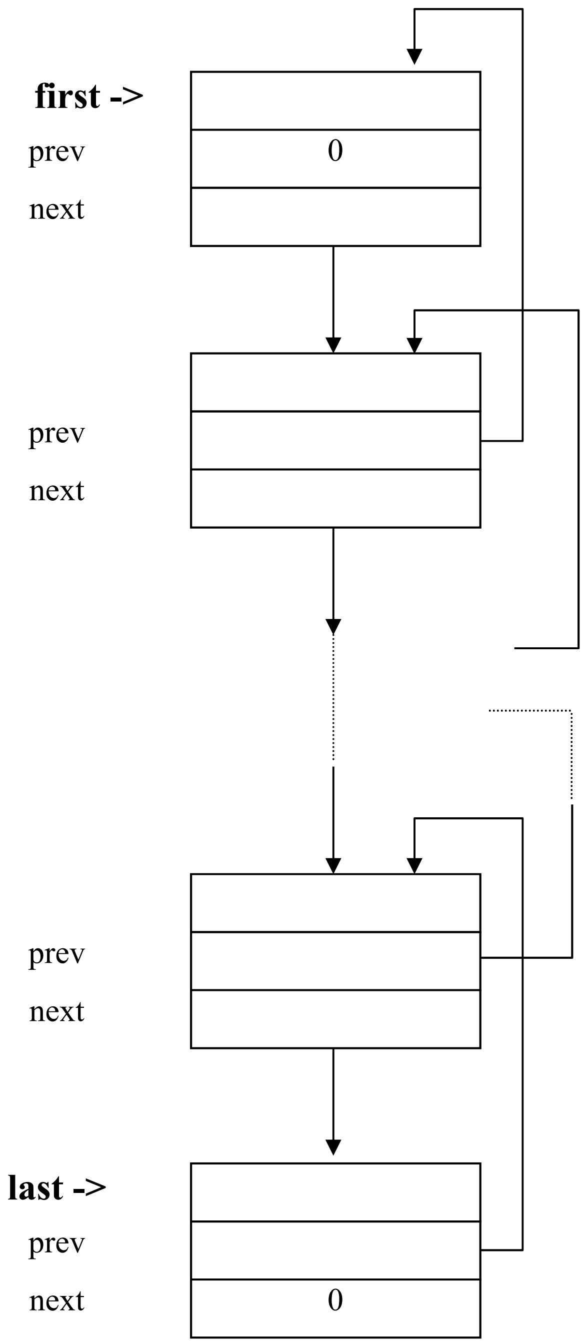
- A doubly-linked list is a linked data structure that consists of a set of sequentially linked nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' **previous** and **next** links, respectively,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

Graph theory

Recapitulation

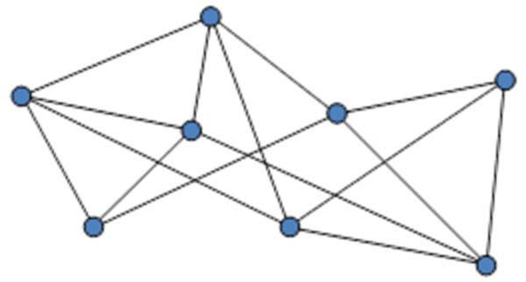
Definition:

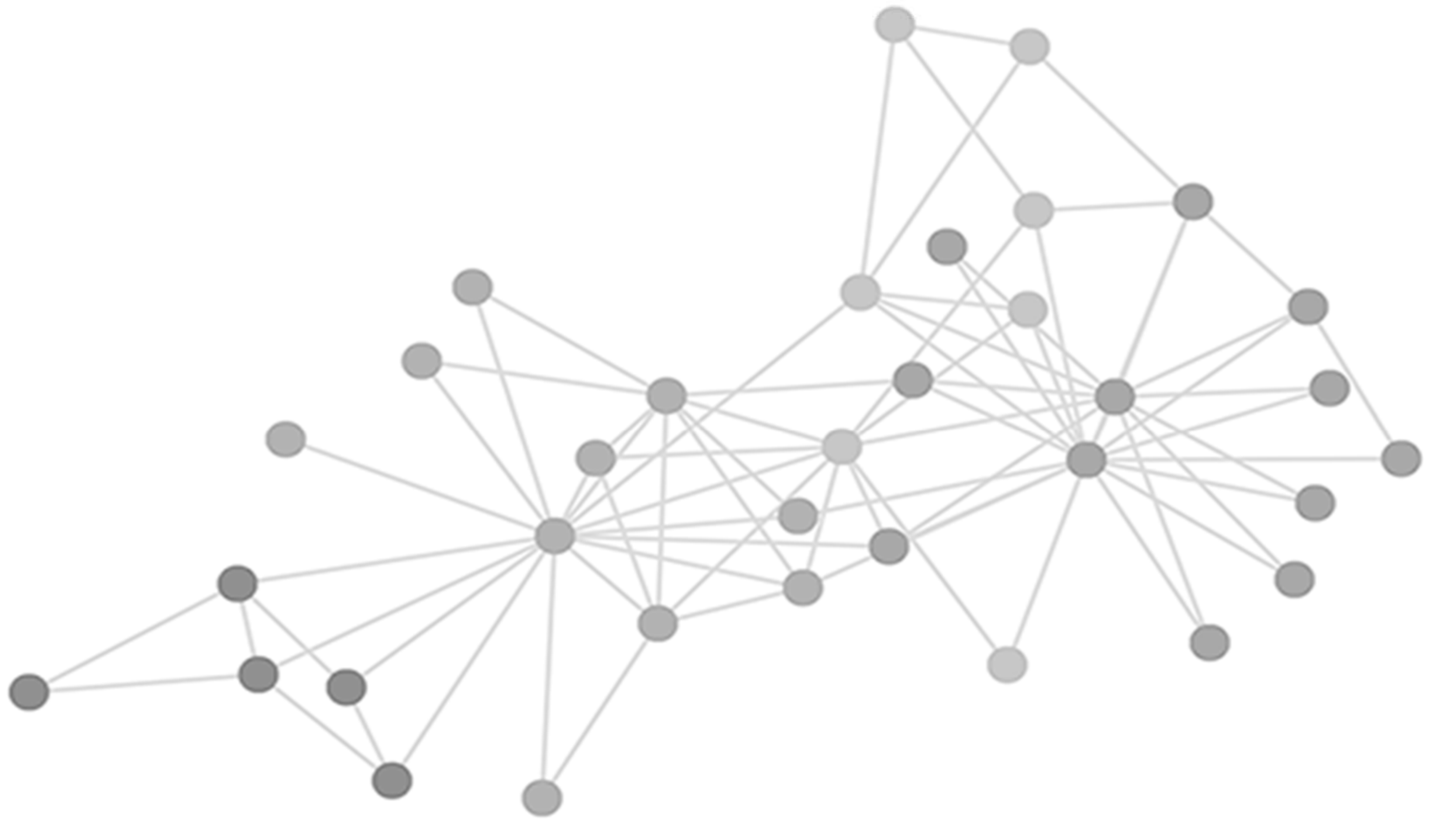
Usually a graph is a pair like:

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

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;

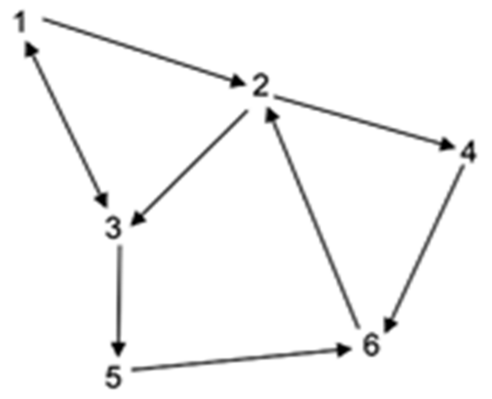
- unordered pair **{a, b}** if the graph is undirected.

In practical applications we can find different kinds of graphs:

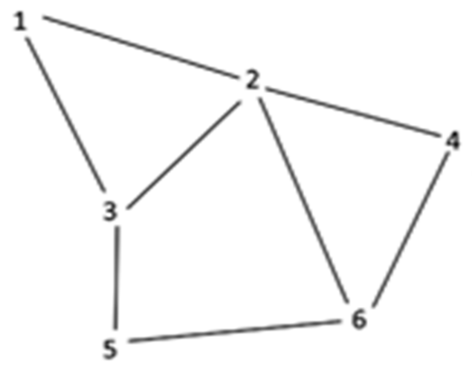
- directed,

- undirected,

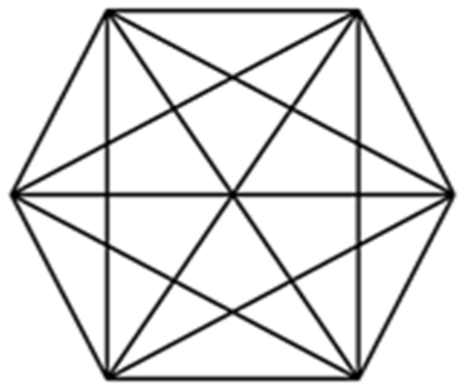
- mixed.



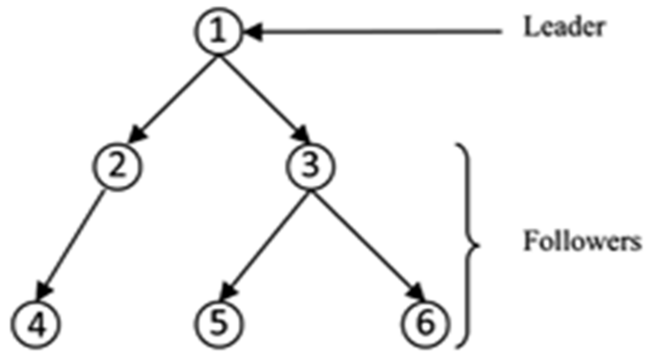
a Directed graph.



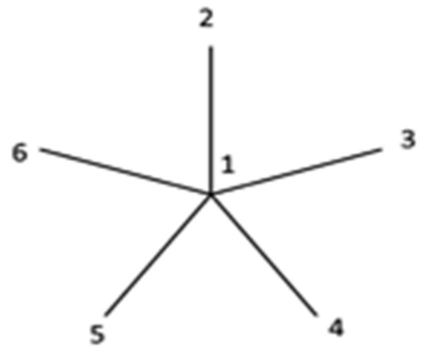
b Undirected graph



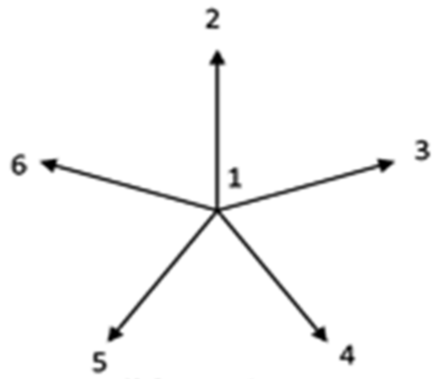
c Complete graph K^6



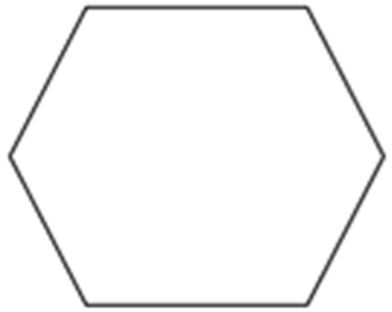
Directed tree graph (formation graph)
d



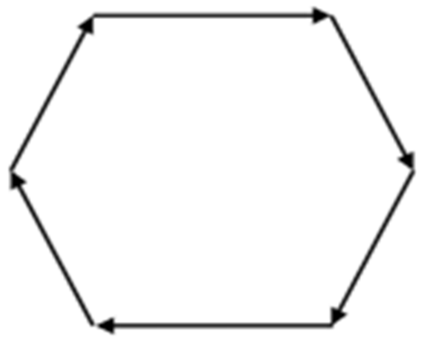
e Undirected star $K_{1,5}$



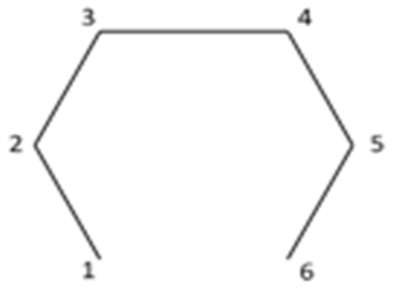
f Directed star



g Undirected 6-cycle.
(2-regular graph)



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**):

$$\mathbf{G' = \langle V', M' \rangle}$$

where $\mathbf{V'} \subseteq \mathbf{V}$, and $\mathbf{M'} \subseteq \mathbf{M}$ (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.

G'' = <V, M''>

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:

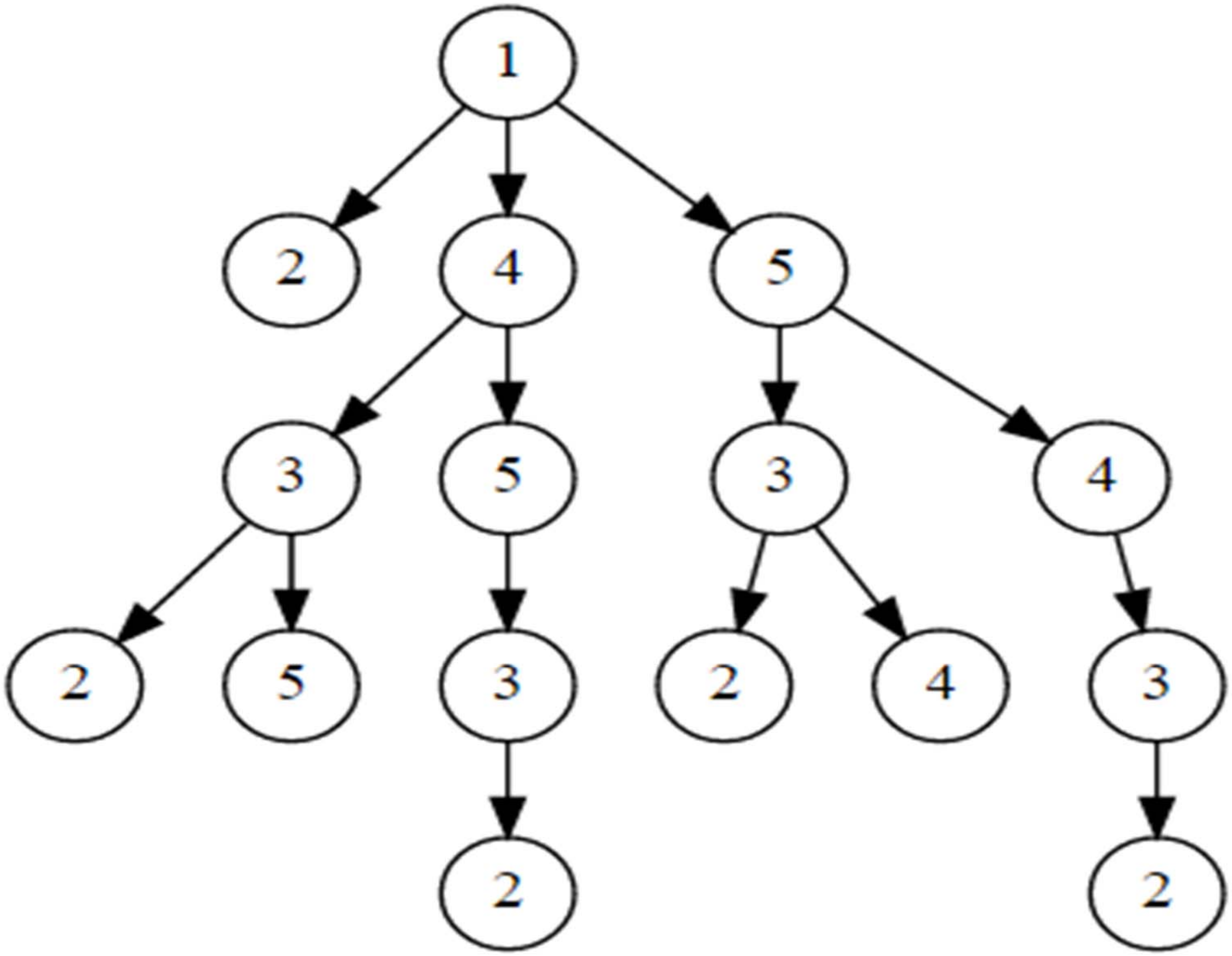
TREE = {A1, A2, A3, ..., An}, where n> 0,

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.

Ordering

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

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.

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.

The criterion function

This function has two parameters, which are pointers of NOD type. Considering p1 as the first parameter of the criterion function and p2 the second one, then the **criterion** function will return:

-1 - if p2 indicates to a data of NOD type which can be inserted in the left subtree of the node pointed by p1;

1 - if p2 indicates to a data of NOD type which can be inserted in the right subtree of the node pointed by p1;

0 - if p2 is equivalent with p1.

More explanations

When we are building a tree, it is established a criterion to find the position in which will be inserted the new current node in the tree – i.e. for the corresponding node of the last acquired value (or set of values).

Example:

Let us consider the following set of numbers:

20, 30, 5, 20, 4, 30, 7, 40, 25, 28, ...

We have to build a tree, in which its nodes will contain the previous numbers along with their frequencies.

Basically all nodes from this tree will have two useful fields :

- the number;

- the frequency.

A way to solve the problem:

a. **p1** - is a pointer to a node from the tree to which the inserting is to be linked (p1 indicates initially to the root of the tree)

b. **p2** – is a pointer to the current node (the node that will be inserted)

c. if **p2->val < p1->val**, then it tries to insert the current node into the left subtree of the node indicated by p1

d. if **p2->val > p1->val**, then it tries to insert the current node into the right subtree of the node indicated by p1

e. if **p2->val = p1->val**, then the current node will not be inserted in the tree, because it already exists a corresponding node for the current value.

The current node is no longer inserted in the tree in the **e** case (when the **criterion** function returns zero). In this case, the nodes pointed by p1 and p2 we consider to be **equivalent**.

```
typedef struct nod
{
        int nr;
        int frequency;
        struct nod * left;
        struct nod * right;
} NOD;
```

In the case of the previous example, the criterion function is:

```
int criterion(NOD *p1, NOD *p2)
{
        if(p2->nr < p1->nr) return(-1);
         if(p2->nr > p1->nr) return(1);
        return(0);
}
```

Function for treating the equivalence

- Usually, when we have two equivalent nodes, **p1** is incremented (or processed in a specific way) and **p2** is eliminated.
- To achieve such processing is necessary to call a function that takes as parameters the pointers p1 and p2, and returns a NOD type pointer (usually returns the value of p1 after deleting the p2).
- We call this function: **equivalence**. It is dependent on the specific issue to be solved by the program.

Example:

```
NOD *equivalence(NOD *q, NOD *p)
{
        q -> frequency ++;
        elibnod(p);
        return(q);
}
```

Other useful functions

- In addition to the functions listed previously, we also use other specific functions for operations on binary trees.
- Typical examples of functions : **elibnod** and **incnod**
- Some functions use a global variable that is a pointer to the root of the tree.

An example of function that is used in laboratory:

```
void elibnod(NOD *p)
/* Release the heap memory areas allocated by a
pointer type node p */
{
 free(p -> word);
 free(p);
}
```

The entry in the tree

- We denote **proot** a global variable to the root of the binary tree.

- It is defined as:

```
NOD *proot;
```

Next we use some functions based on the global variable **proot**.

Inserting a leaf node in a binary tree

The function **insnod** inserts a node in the tree, according to the following steps:

1. It is allocated a memory area for the node to be inserted in the tree. Consider **p** being the pointer for this memory.

2. By calling the **incnod** function, we have to fill the node with data. If **incnod** returns 1, then jump to step 3. Otherwise the function returns the value zero (after deleting p).

3. Assignments are made:

p->left = p->right = 0

since the new node is a leaf one.

4. **q = proot**

5. Find the position in the tree where the insertion will be made (find the possible parent for the node which will be inserted):

i = criterion(q, p)

6. If **i<0**, then jump to step 7; otherwise jump to step 8.

7. Try to insert the current node to the left subtree of the root **q**.
- If **q -> left** is zero, then the current node becomes the left leaf of q (**q->left = p**). Afterwards the function returns the value of **p**.
- Otherwise **q = q->left** , and jump to step 5.

8. If **i>0**, then jump to step 9; otherwise jump to step 10.

9. Try to insert the current node to the right subtree of the root **q**.
- If **q -> right** is zero, then the current node becomes the right leaf of q (**q->right = p**). Afterwards the function returns the value of **p**.
- Otherwise **q = q->right** , and jump to step 5.

10. The current node cannot be inserted into the binary tree. Call the **equivalence** function.

2. The access to a node of a tree

- Access to a node implies a criterion for locating the relevant node in the tree.
- It will be used the **criterion** function.
- The function which performs the search will identify an equivalent node in the tree, with the one pointed by **p** (used as input parameter in the searching function).

The searching function (denoted **search**) will return:

- a pointer to the equivalent node in the tree;

- 0, if there is no such equivalent node.

```
NOD *search(NOD *p)
{
        extern NOD *proot;
        NOD *q;
        int i;
        if (proot = = 0) return 0;  /*the tree is empty*/
        for (q = proot; q;  )
          {
                if ( (i = criterion(q, p)) = = 0)    return q;
                else   if (i < 0)       q = q -> left;
                        else            q = q -> right;
          }
        return 0;
}
```

3. Tree traversal

- Tree traversal is a form of graph traversal and refers to the process of visiting (examining or updating) each node in a tree data structure, exactly once, in a systematic way.
- Such traversals are classified by the order in which the nodes are visited.
- The following algorithms are described for a binary tree, but they may be generalized to other trees as well.

Traversing the nodes of a binary tree can be done in several ways:

- **Pre-order**
- **In-order (symmetric)**
- **Post-order**

Note: In the following, displaying may be replaced by processing (which is a more general task).

Pre-order

- Display the data part of root element (or current element)
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.

In-order (symmetric)

- Traverse the left subtree by recursively calling the in-order function.
- Display the data part of root element (or current element).
- Traverse the right subtree by recursively calling the in-order function.

Post-order

- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
- Display the data part of root element (or current element).

- The access to a node allows the processing of the information contained in the respective node. For this you can call a function that is dependent of the specific problem that implies traversal of the tree.

- In the following we will use the **process** function.

```
void process( NOD *p)
```

Pre-order

```
void preord(NOD *p)
{
 if(p != 0)
       {
        process(p);
        preord(p -> left);
        preord(p -> right);
       }
}
```

In-order (symmetric)

```
void inord(NOD *p)
{
 if(p != 0)
       {
        inord(p -> left);
        process(p);
        inord(p -> right);
       }
}
```

Post-order

```
void postord (NOD *p)
{
 if(p != 0)
       {
        postord (p -> left);
        postord (p -> right);
        process(p);
       }
}
```

Example:

The same problem with a series of numbers:

20, 30, 5, 20, 4, 30, 7, 40, 25, 28, ...

Notes:

- The program from the laboratory (with nodes containing words together with their frequency of occurrence in a text) is solved more easily by using a tree (because the search operation in a list is less efficient).

- The search process in a tree requires fewer steps than the search in a list.

4. Deleting a binary tree

- to delete a binary tree is required to traverse it and delete each node of that tree (in a specific way).

- it is used the **elibnod** function.

- the tree will be traversed in **postorder**

Deleting a binary tree in postorder

```
void delete_tree(NOD *p)
{
 if(p != 0)
       {
        delete_tree (p -> left);
        delete_tree (p -> right);
        elibnod(p);
       }
}
```

Notes:

- The **delete_tree** function does not assign zero to the global variable (to **proot**).

- This assignment will be required immediately after the call of the **delete_tree** function.

```
proot=0;
```

- A **degenerate binary tree** – is the one where all of the nodes (except the last leaf) contain only one sub node.

5. Deleting a node specified by a key

- The key is a part of every node and it is unique for each of them:

```
typedef struct nod
        {
                < statements ; >
                type key;
                struct nod *left;
                struct nod *right;
        }  NOD;
```

  where the key type can be **char**, **int**, **float** or **double**.

- **Especially leaf nodes can be deleted!**

- Deleting a node that is not a leaf involves more complicated operations to restore the binary tree structure.

```
void search_delete(NOD *p, int c)    /* the key is an integer here */
{
      if (p != 0)
        {
            if (( p -> left = = p -> right) && ( p -> left = = 0 ) && ( p -> key = = c))
                        {
                            elibnod (p);
                            return;
                        }
             search_delete(p -> left, c);
             search_delete(p -> right, c);
        }
}
```

Note: This operation, which includes search and delete, is processed in **pre-order**.

Notes:

- At a closer look, from the above function it lacks something: when a leaf node identified was deleted from his parent, then the father should have the zero value for the recursive pointer that correspond to the deleted direction (node)!

- How can you solve this problem?

Supplementary note

- In some applications are required specific flattening trees (having a low height) but with large numbers of nodes.
- In such cases it may replace a binary tree with another tree that allows a greater number of direct offsprings (descendants).
- The procedure is relatively simple, and instead of two recursive pointers (to the left subtree and to the right one also), we can use a vector of recursive pointers (having a certain length), which allows (for the new type of structure) to define an arbitrary number of direct descendants.