Data Structures and Algorithms (DSA)
Course 10
Trees

Iulian Năstac

TREES

(Recapitulation)

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.

The nodes in a tree (Recapitulation)

- 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
(Recapitulation)

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

Binary tree
(Recapitulation)

- 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).

Transformation
(Recapitulation)

- 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 (Recapitulation)

1. Links between them all brothers descendants of the same parent node and suppress their links with the parent, except the first son.
2. The former prime son node son becomes the left son of the parent, and the other former brothers become sequentially, the roots of 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. Accessing a node from a tree;
3. Traversing binary tree;
4. Delete a tree.

Recapitulation

- 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.

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
(Recapitulation)

- 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
Recapitulation

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

- It is defined as:

NOD *proot;

Inserting a leaf node in a binary tree

Recapitulation

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->right** , 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 (prad = = 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:

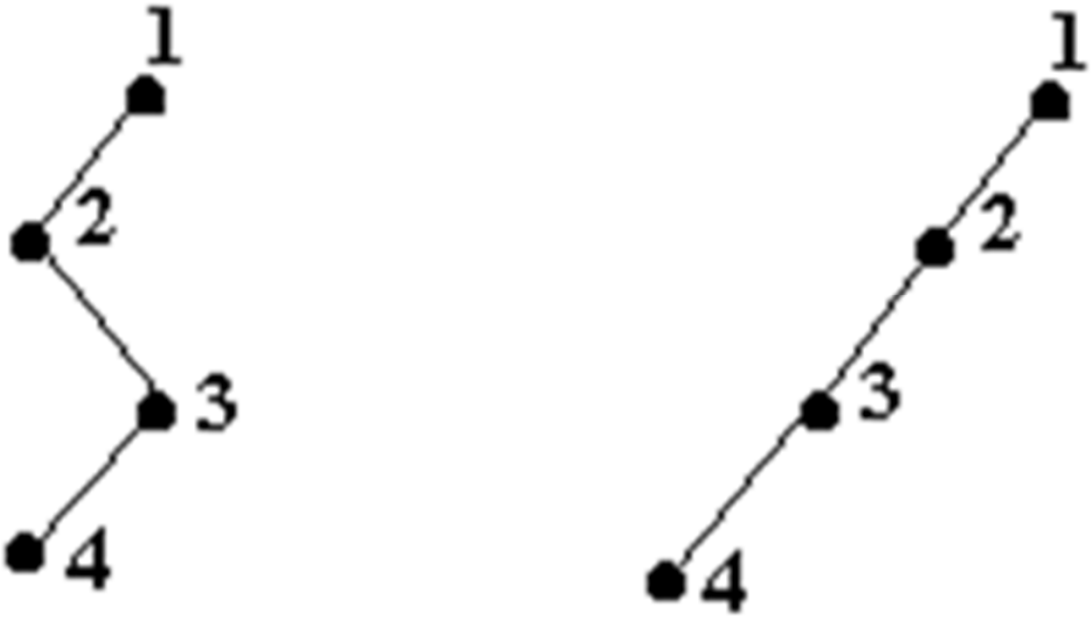
- At a closer look, from the above function it lacks something: when a leaf node identified was deleted from his parent, then the father should have the zero value for the recursive pointer that correspond to the deleted direction (node)!

- How can you solve this problem?

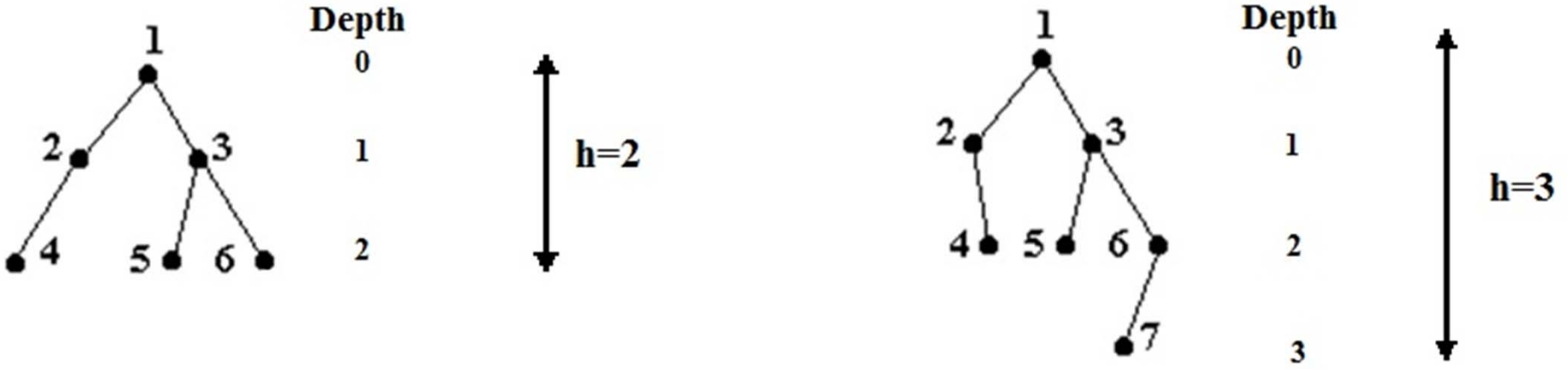
Supplementary note

- 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.

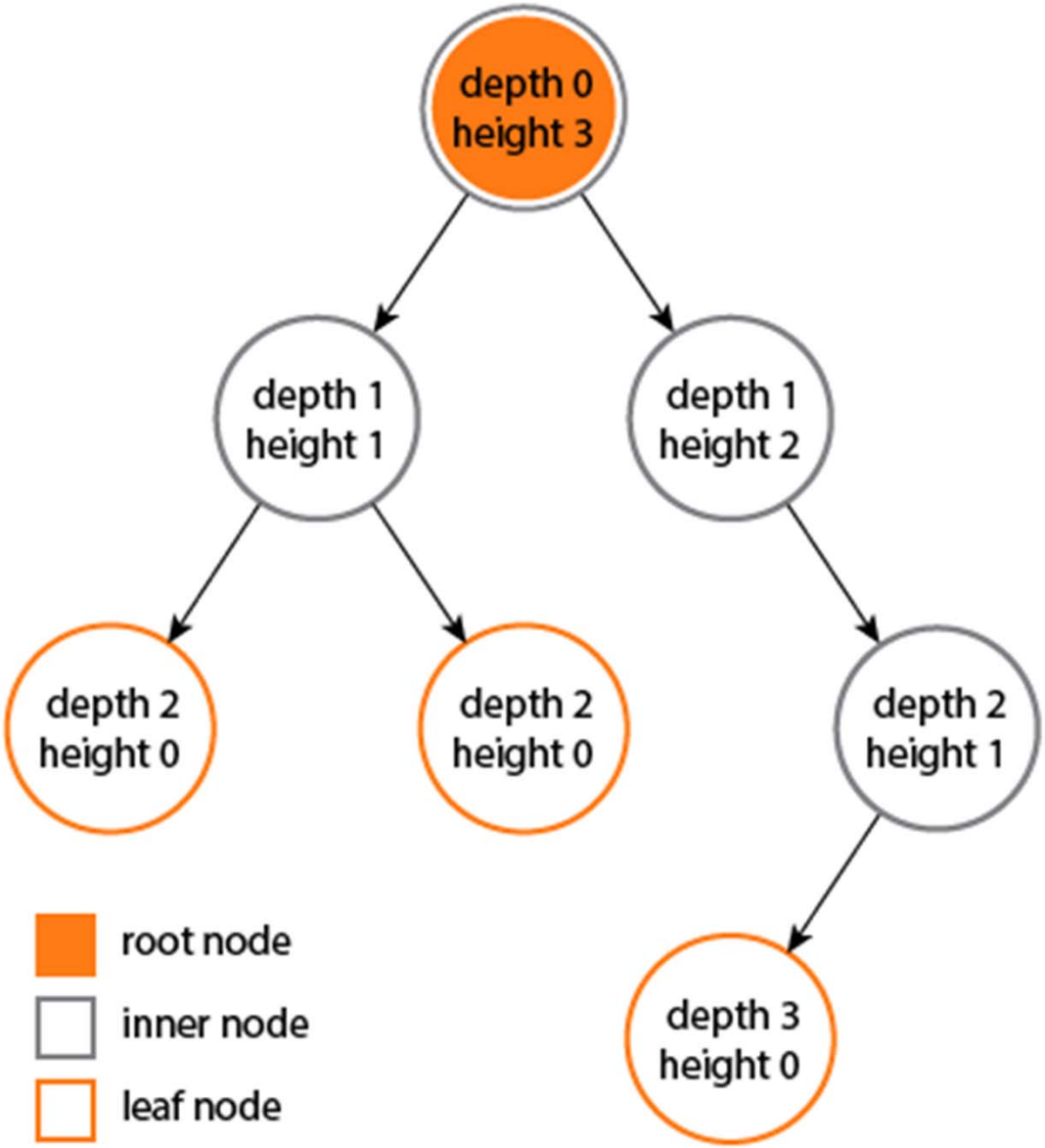
- A **degenerate binary tree** – is the one where all of the nodes (except the last leaf) contain only one sub node.



Depth (or Height) of a Binary Tree



- Usually, we can denote the depth of a binary tree by using **h** (or **i**)


depth 0
height 3
depth 1
height 1
depth 1
height 2
depth 2
height 0
depth 2
height 0
depth 2
height 1
depth 3
height 0
root node
inner node
leaf node

Depth and height

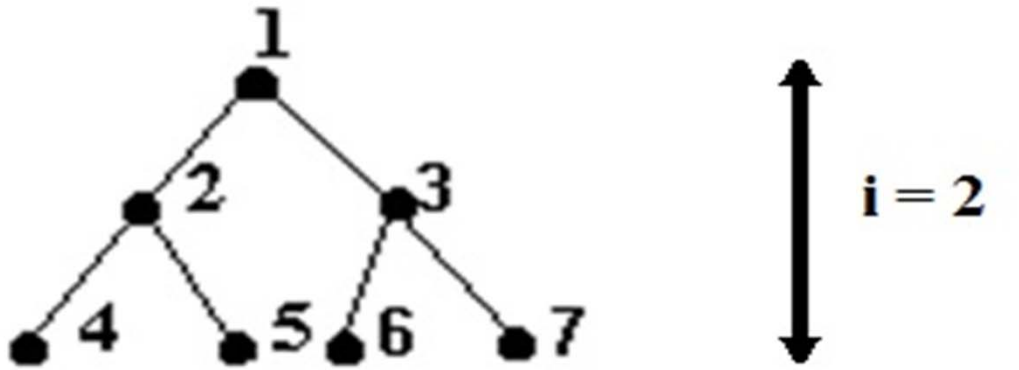
- The **depth** of a node is the number of edges from the node to the tree's root node. A root node will have a depth of 0.

- The **height** of a node is the number of edges on the *longest path* from the node to a leaf. The lowest leaf node will have a height of 0.

- For a binary tree, the height and depth (globally) measure the same thing.

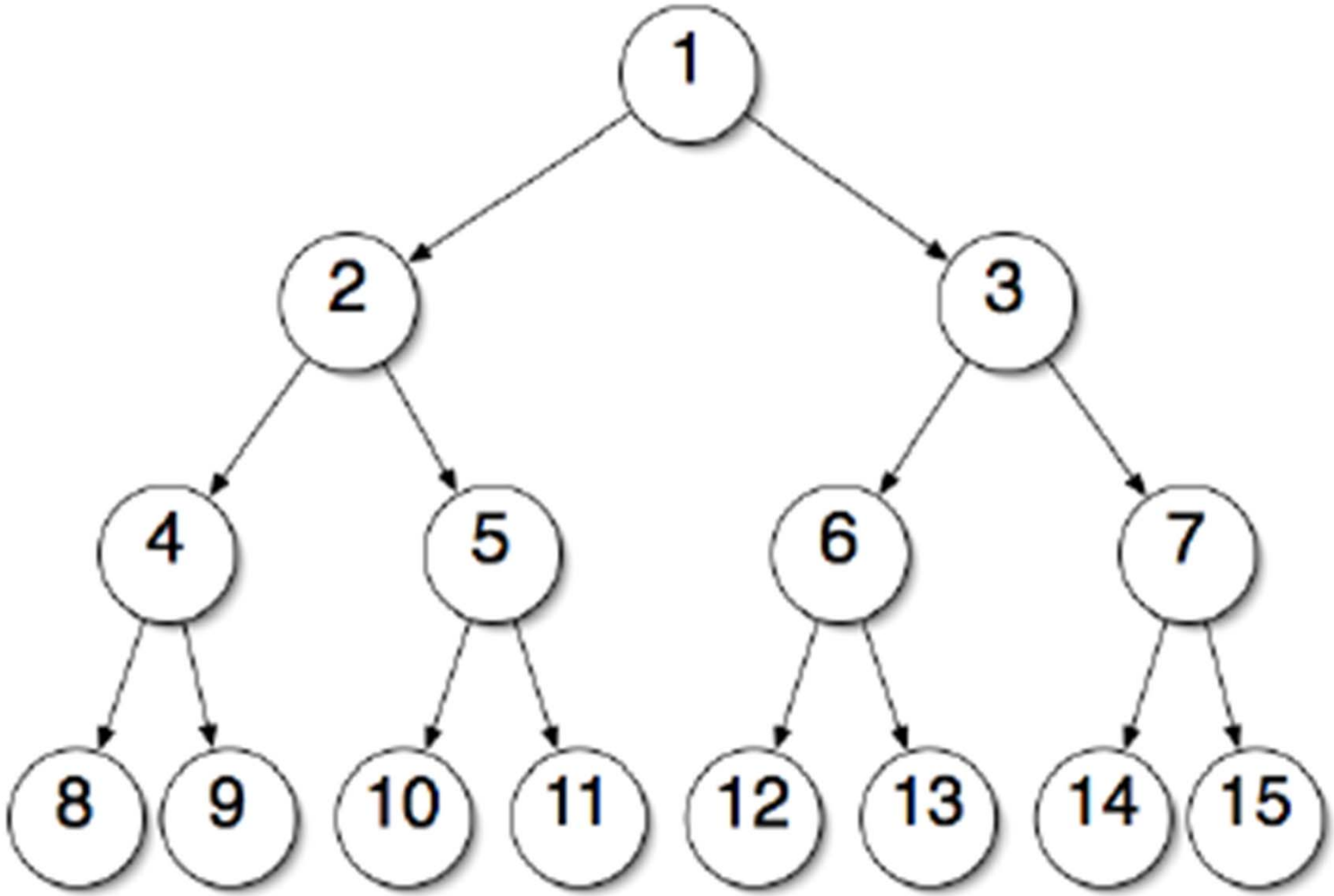
Special binary trees

P: A binary tree with the height **i** could have a maximum number of $2^{i+1}-1$ nodes.



Notes:

A level with the depth **k** could have maximum 2^k nodes


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

A proof by induction:

- It easily observed that previous declaration is valid for ***i*** = 0, 1, 2 ...
- Therefore, we must prove the validity of the declaration for the depth of ***i*** will implies a validity for ***i + 1***

  We rely on the fact that any level of depth ***k*** has a maximum of 2^k nodes.

It is easy to notice that for the height of **i+1** we get:

No_of_nodes = $2^{i+1} - 1 + 2^{i+1} = 2^{i+1}(1 + 1) - 1 =$

$2^{i+1}.2 - 1 = 2^{i+2} - 1$ nodes

The full binary tree

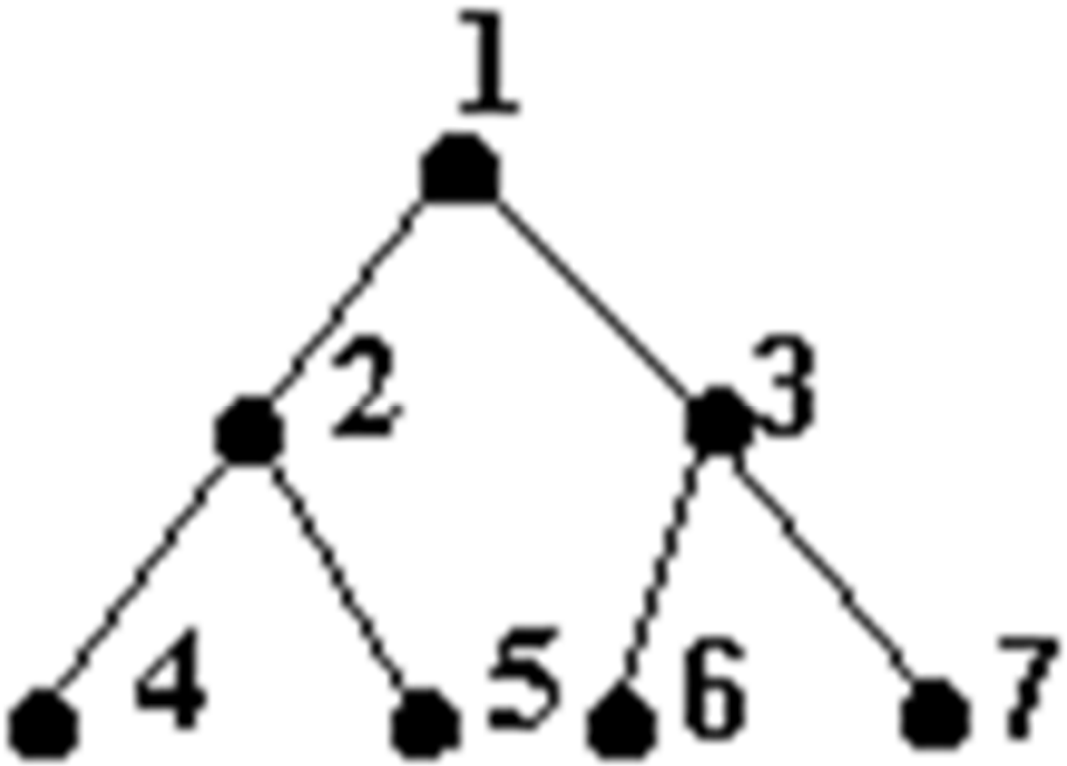
A full binary tree is the one that has the maximum number of vertices ($2^{i+1}-1$) for a specified height of **i**.

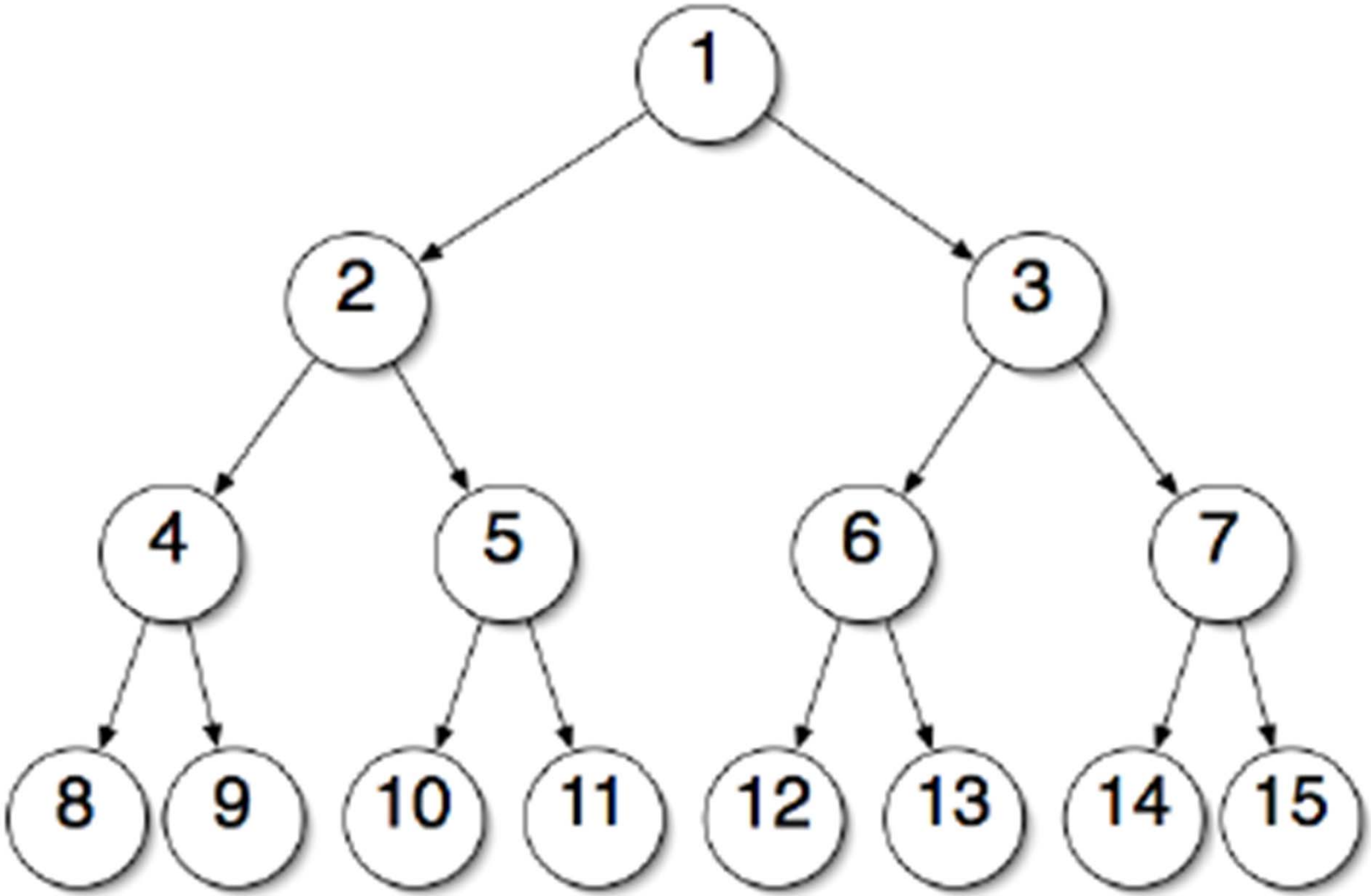
For example, a full binary tree of height 2 is as follows :

A full binary tree is that one in which every node has two children (excepting the last level with the leaves).

Notes:

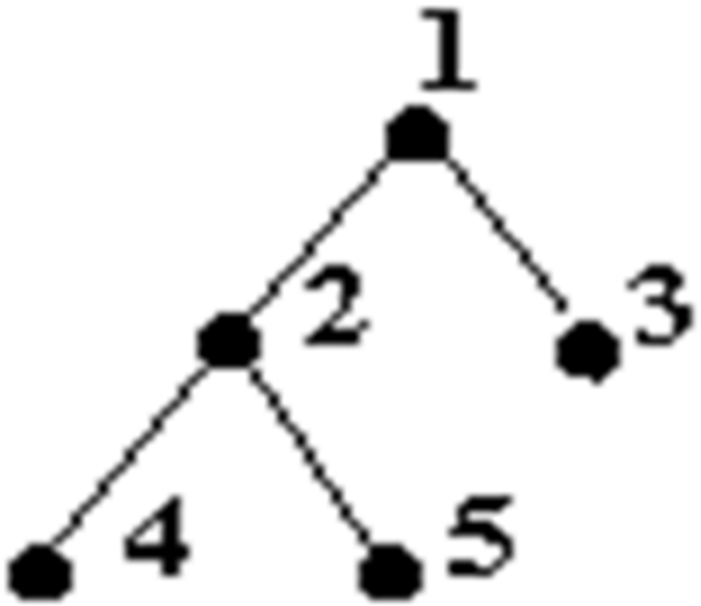
- The full binary tree nodes are numbered from the left to right according with their depth.




1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

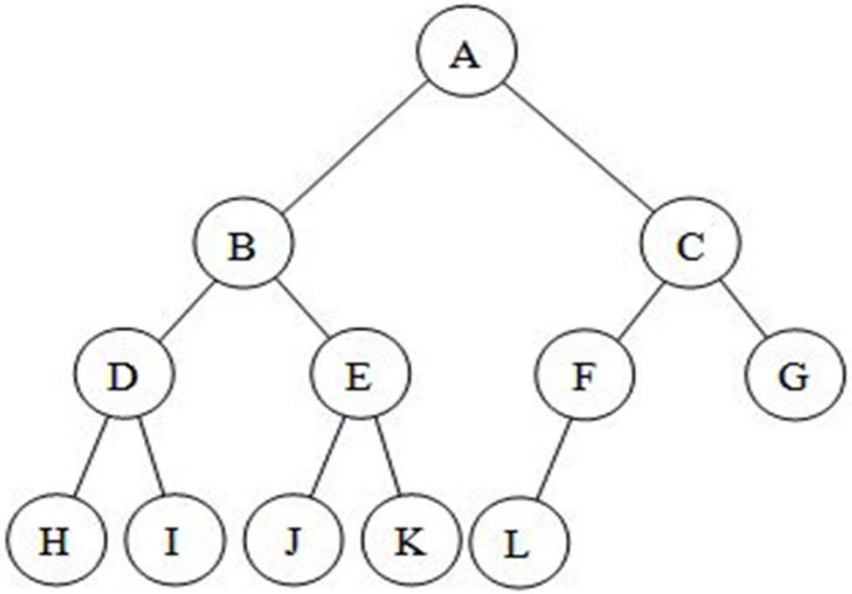
Complete binary tree

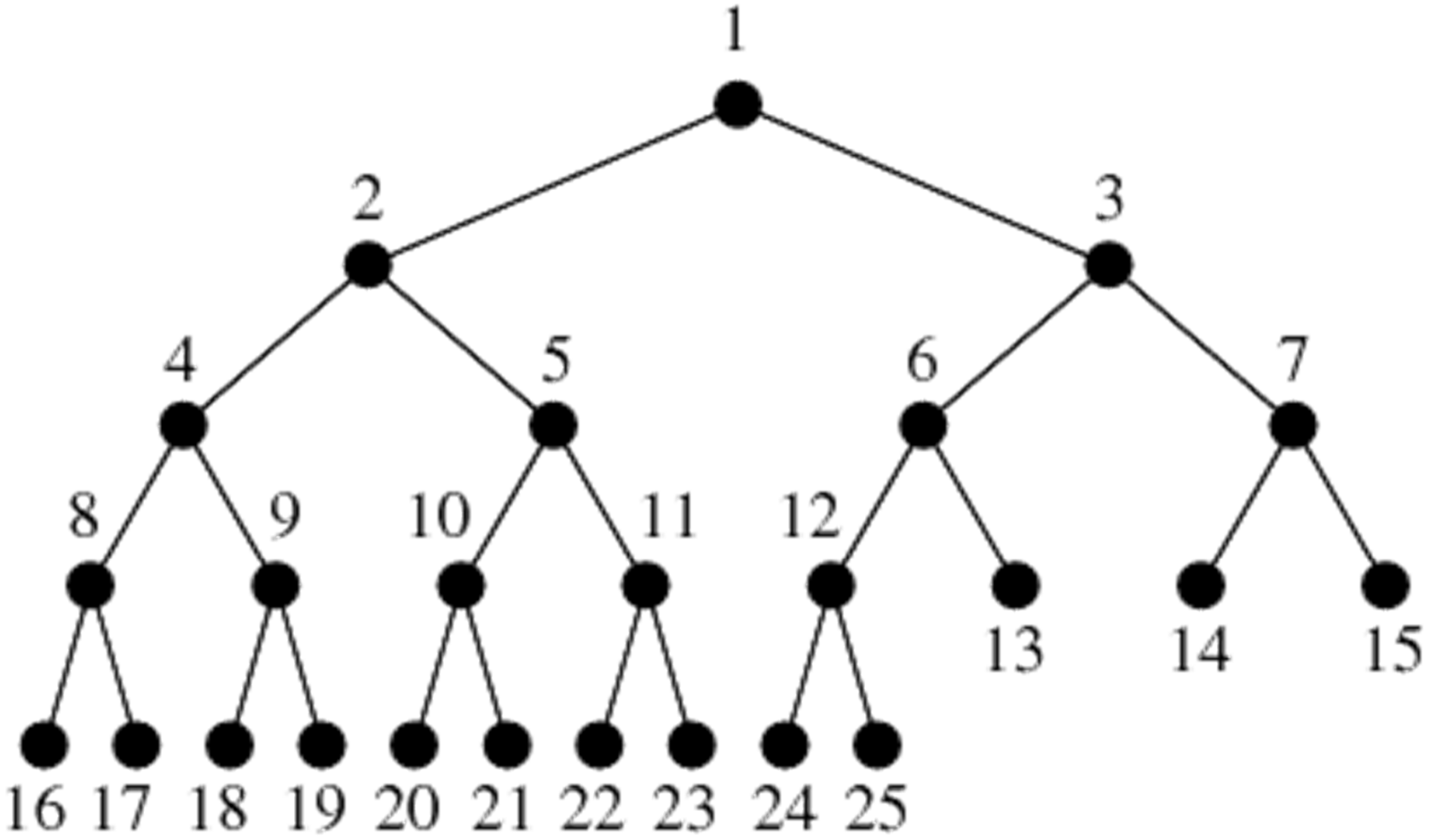
A binary tree with ***n*** nodes that has a height of ***i*** is called as being a **complete binary tree** if it is obtained from a full binary tree with a height of ***i***, in which there are eliminated the last consecutive nodes, numbered with ***n+1***, ***n+2***, ... up to $2^{i+1}-1$.



Notes:

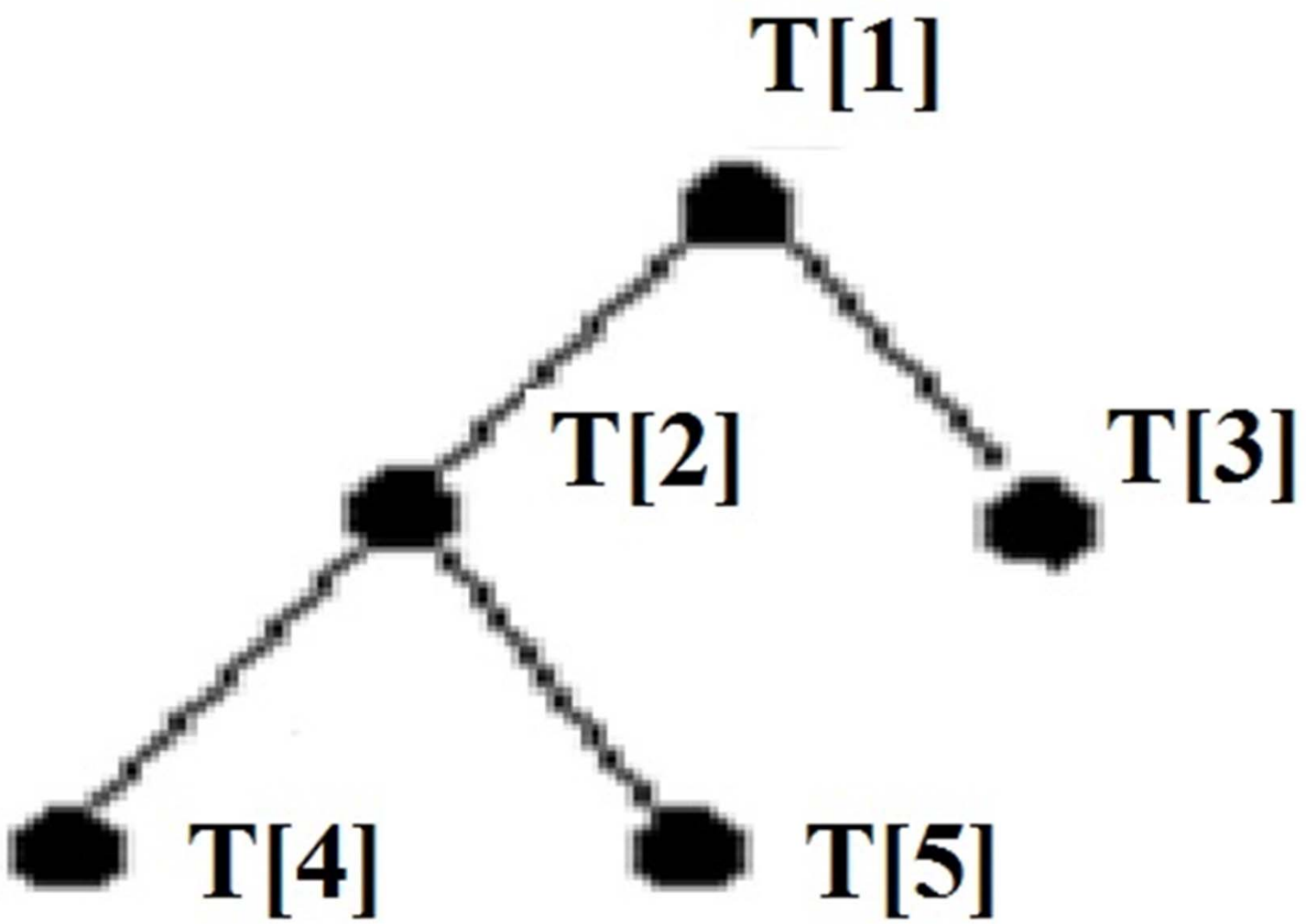
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.




1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

Notes:

- A complete binary tree can be sequentially represented using a vector (noted **T**), in which the nodes of depth **k**, from left to right, are inserted in the following positions: **T[2^k]**, **T[2^k+1]**, …, **T[2^{k+1}-1]**, excepting the final level, which may be incomplete.



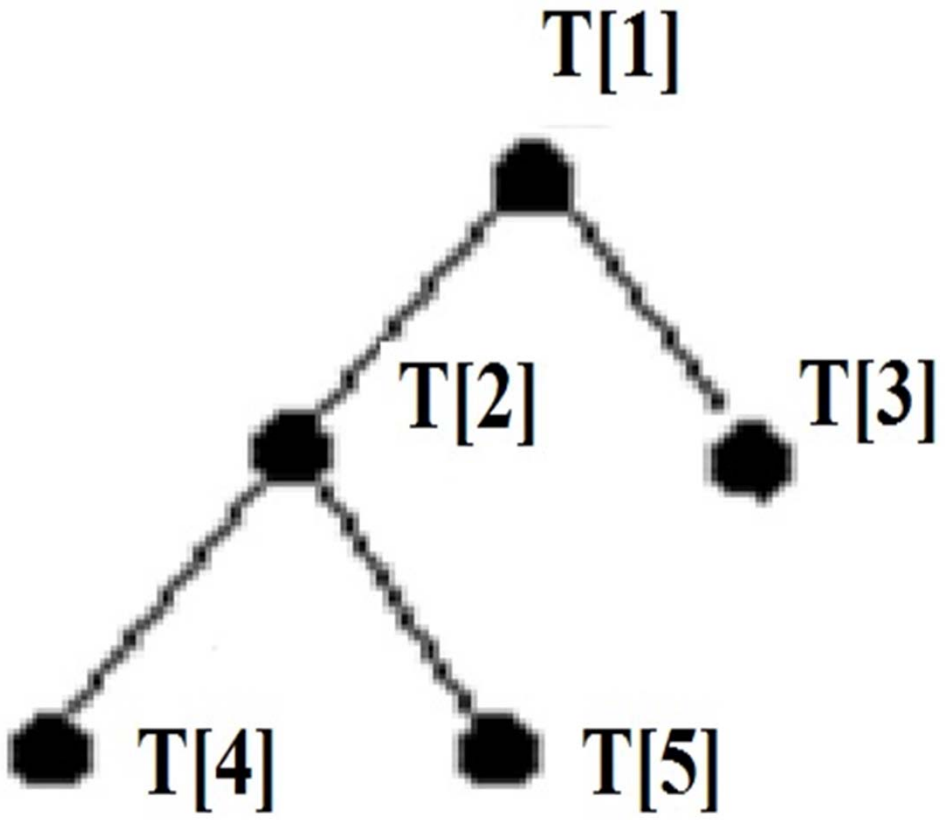
• **Warning**: This is a generic vector, which it begins with T [1] (not with T [0], as usual in the C programing).

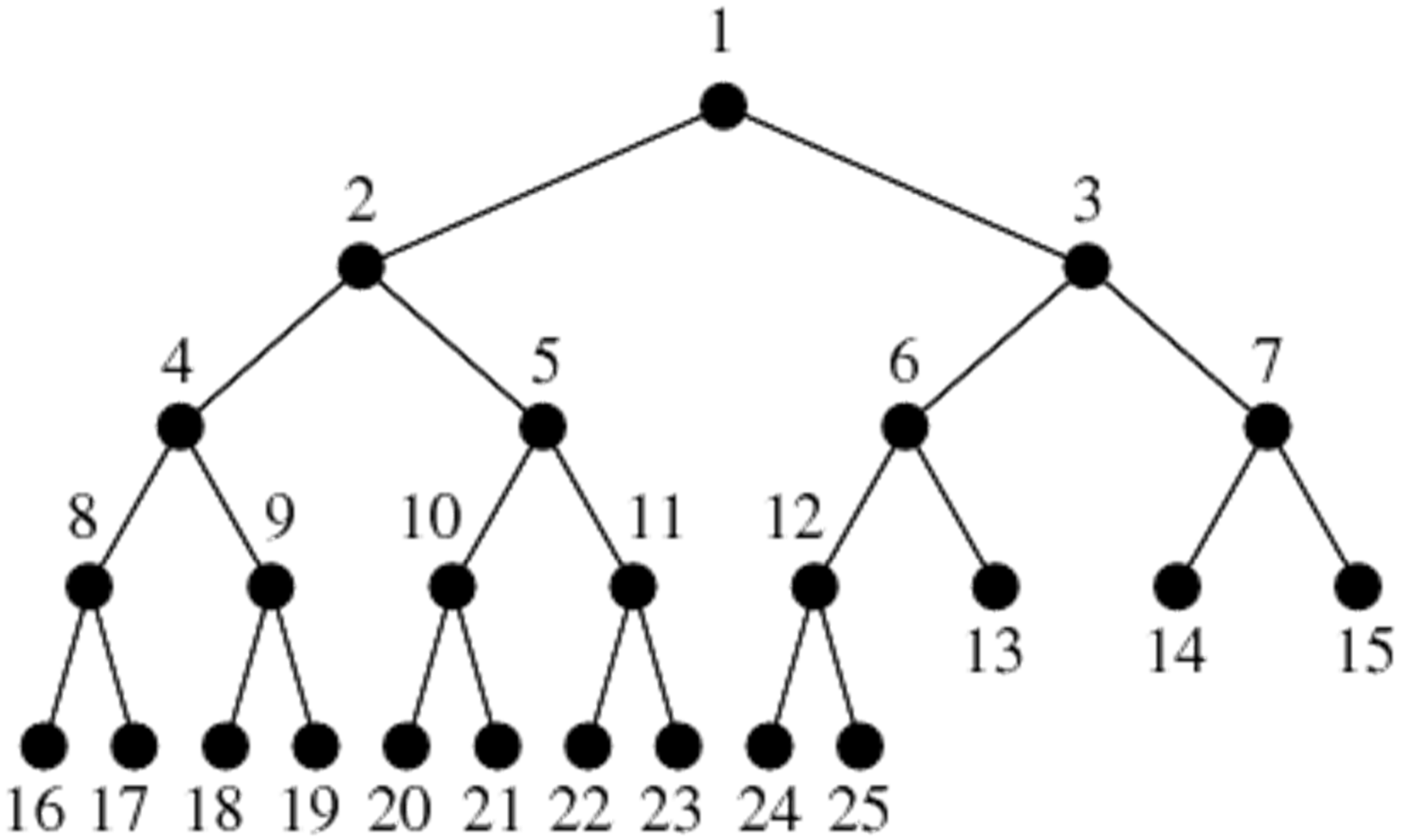
• We can make the necessary changes when we will write the code in C.

Notes:

- The parent of a node from **T[i]**, i>1, can be found in **T[i div 2]** .

- The sons of a node from **T[i]**, can be found (if exist) in **T[2·i]** and **T[2·i + 1]** .




1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

We can define:

$$\lfloor x \rfloor = \max\{n \mid n \le x, n \in Z\}$$

$$\lceil x \rceil = \min\{n \mid n \ge x, n \in Z\}$$

Useful properties:

1. $x-1<\lfloor x\rfloor\le x\le\lceil x\rceil<x+1$ $\forall\ x\in R;$

2. $\left\lfloor\frac{n}{2}\right\rfloor+\left\lceil\frac{n}{2}\right\rceil=n$ $\forall\ n\in Z;$

3. $\left\lceil\left\lceil\frac{n}{a}\right\rceil\Big/b\right\rceil=\left\lceil\frac{n}{ab}\right\rceil$ and $\left\lfloor\left\lfloor\frac{n}{a}\right\rfloor\Big/b\right\rfloor=\left\lfloor\frac{n}{ab}\right\rfloor$ $\forall a, b, n\in Z;$ $a, b\neq 0$

4. $\left\lfloor\frac{n}{m}\right\rfloor=\left\lceil\frac{n-m+1}{m}\right\rceil$ and $\left\lceil\frac{n}{m}\right\rceil=\left\lfloor\frac{n+m-1}{m}\right\rfloor$ $\forall\ n, m\in N^{*};$

The height of a complete binary tree

- We have to demonstrate that the height of a complete binary tree with **n** vertices is:

$$i = \lfloor \log_2 n \rfloor$$

Demonstration:

- Considering:
  - **n** – number of nodes;
  - **i** – height of the tree.
- We already know that:
  - $n_{max} = 2^{i+1}-1$ (when the binary tree is full)
  - $n_{min} = 2^{i}$ (when on the last level we have only one node)

- It results that:

$$i = \log_2 n_{\min} \quad => \quad i = \lfloor \log_2 n_{\min} \rfloor \quad \textbf{(1)}$$

Since the logarithmic function is increasing then:

$$\lfloor \log_2 n_{\max} \rfloor = \lfloor \log_2 (2^{i+1} - 1) \rfloor < \lfloor \log_2 2^{i+1} \rfloor = i + 1$$

Therefore:

$$\lfloor \log_2 n_{\max} \rfloor < i + 1 \qquad \textbf{(2)}$$

From (1) and (2) it result that:

$$i = \lfloor \log_2 n_{\min} \rfloor \leq \lfloor \log_2 n_{\max} \rfloor < i + 1$$

$$\Rightarrow \qquad i = \lfloor \log_2 n \rfloor$$

The heap tree

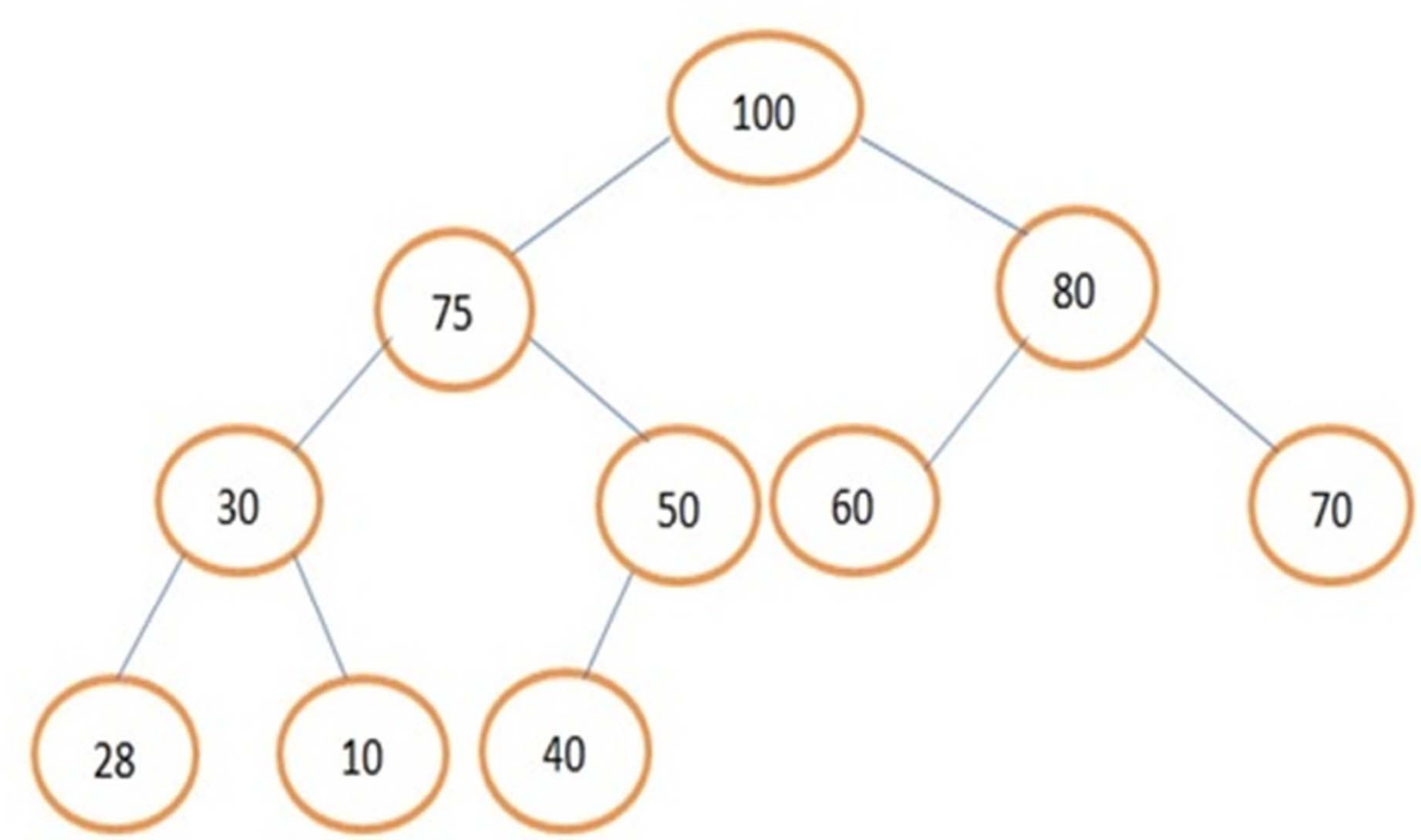
A **heap** is a specialized tree-based data structure that satisfies the **heap property**:

If *A* is a parent node of *B* then the key of node *A* is ordered with respect to the key of node *B*. The same ordering is applied across the entire heap.

The heap is not a classic binary tree!

There is no order between left and right son of a father inside of a heap…

Example of a heap tree



Heap tree types

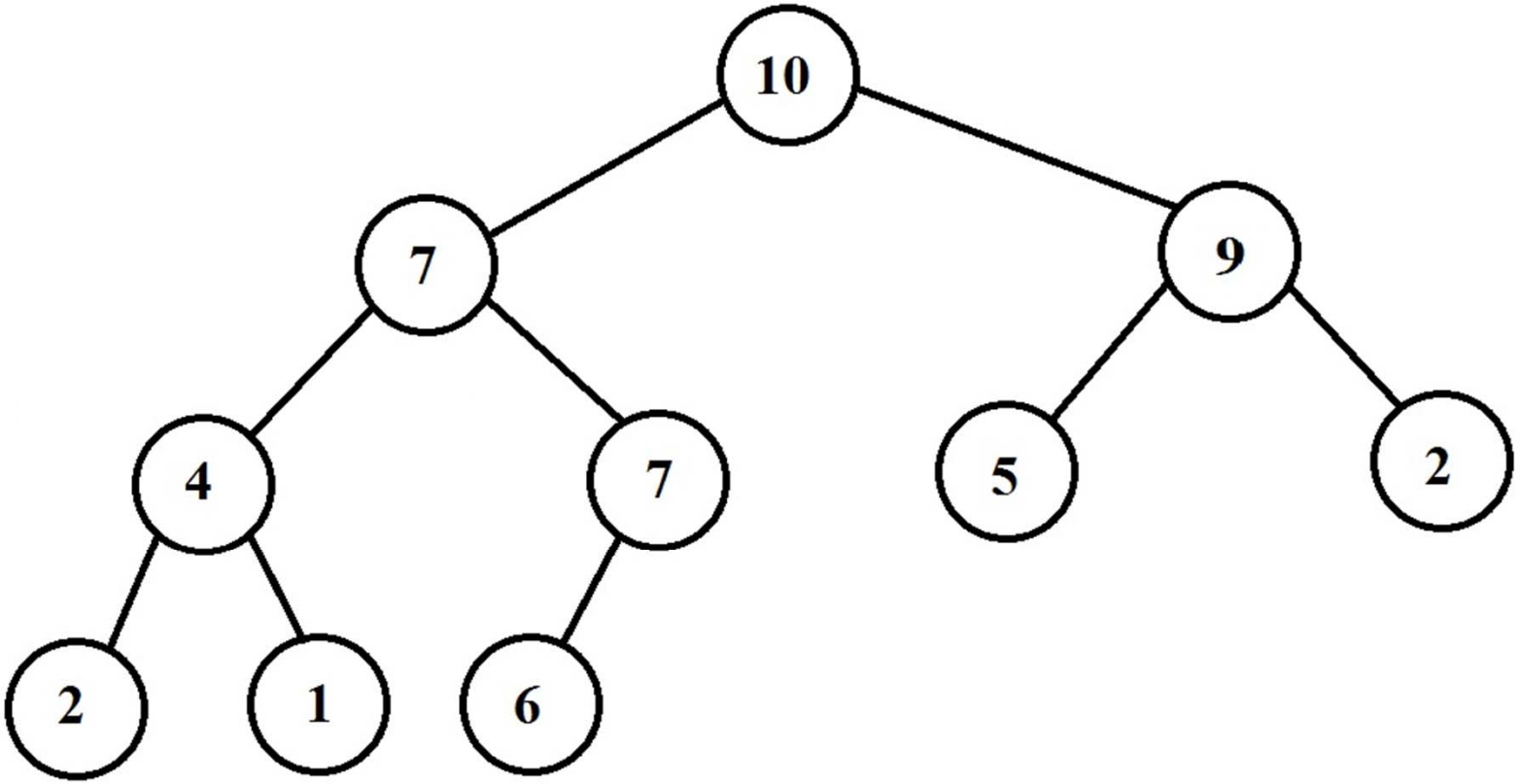
- Heaps can be classified further as either a "**max heap**" or a "**min heap**".
- In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node.
- In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node.

Notes:

- Usually, in many applications, a **max heap** is simply called **heap tree**

- Any **heap tree** can be represented by a vector (one-dimensional array)

Example:



This heap can be represented by the following vector:

10	7	9	4	7	5	2	2	1	6
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Notes:

- In a heap tree we can make modifications at the node level (changing the value of the current node).
- Thus the value of a node can be increased or decreased, resulting a canceling of the specific order inside the heap tree.
- The order of the heap can be simply restored through two operations called ***sift-down*** and ***sift-up***.

sift-up (percolate) in a heap

sift-up = means to move a node up in the tree, as long as needed; used to restore heap condition after insertion.

- Called "sift" because node moves up the tree until it reaches the correct level, as in a sieve. Often incorrectly called "shift-up".

- It is also said that the changed value was filtered (percolated) to his new position.

sift-down in a heap

***sift-down* = moves a node down in the tree, similar to sift-up; used to restore heap condition after deletion or replacement**.

- If a node value decreases so that it becomes lower than the elder son, it is enough to change between them these two values, and continue the process (downward) until the heap property is restored.
- It is said that the changed value was sieved (sift down) to his new position.

Note:

- Next, the vast majority of functions will be written in a **pseudocode** version

Pseudocode of the sift-down function

```
void sift_down (T[1…n], i)
{ int k, x, j;
  k ← i ;
  do      {
            j ← k ;
            if ((2j ≤ n) ∧ (T[2j] > T[k]))  then k ← 2j;
            if ((2j+1 ≤ n) ∧ (T[2j+1] > T[k]))  then k ← 2j+1;
            x ← T[j];
            T[j] ← T[k];
            T[k] ← x
          } while (j ≠ k)
}
```

Pseudocode of the sift-up function

```
void sift_up (T[1…n], i)
{ int k, j, x;
  k ← i ;
  do      {
            j ← k ;
            if ((j > 1) ∧ (T[ j div 2] < T[k]))  then k ← j div 2;
            x ← T[ j];
            T[ j] ← T[k];
            T[k] ← x
          } while (j ≠ k)
}
```