Data Structures and Algorithms (DSA)
Course 11
Trees

Iulian Năstac

4. Deleting a binary tree (Recapitulation)

- 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 (Recapitulation)

```
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:

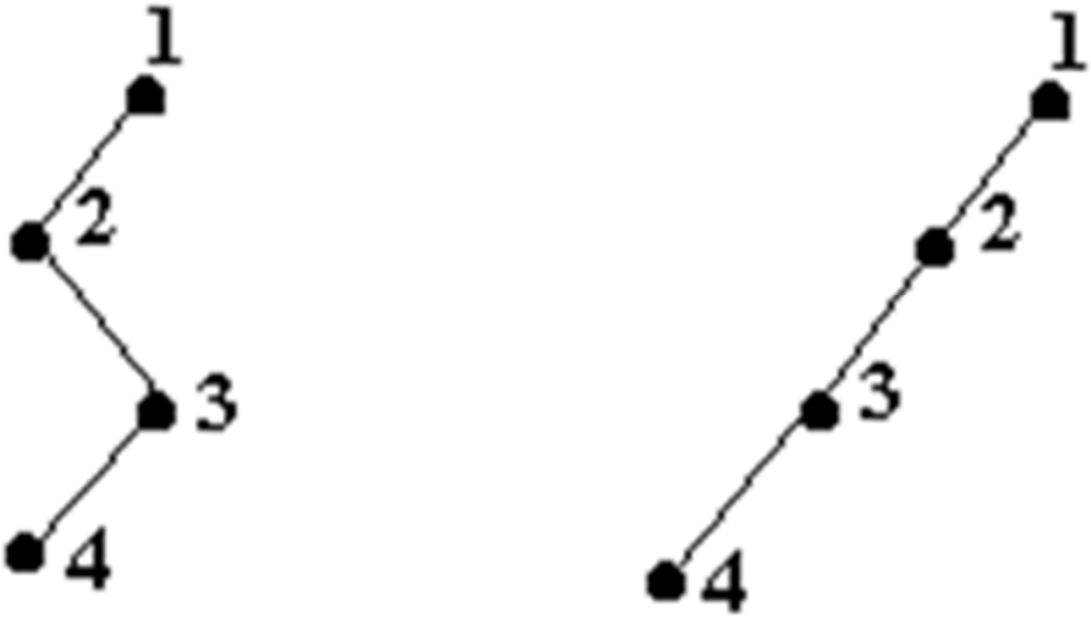
- At a closer look, from the above function it lacks something: when a leaf node identified was deleted from his parent, then the father should have the zero value for the recursive pointer that correspond to the deleted direction (node)!

- How can you solve this problem?

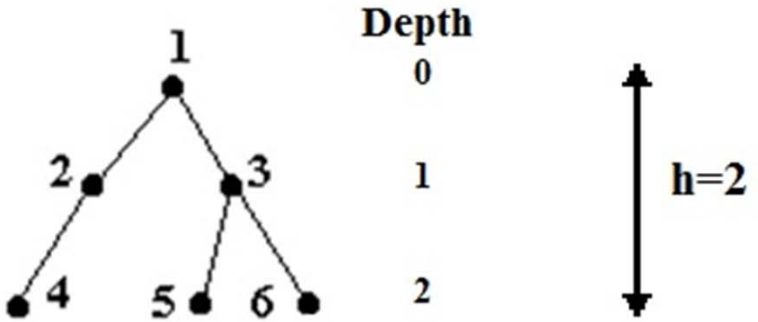
Supplementary note

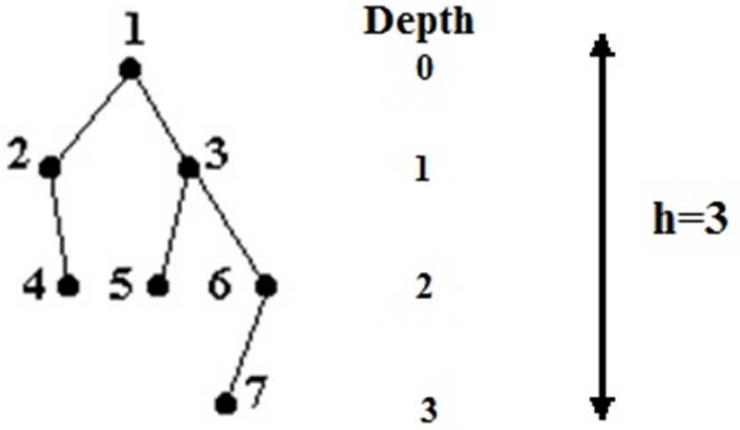
- In some applications are required specific flattening trees (having a low height) but with large numbers of nodes.
- In such cases it may replace a binary tree with another tree that allows a greater number of direct offsprings (descendants).
- The procedure is relatively simple, and instead of two 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.

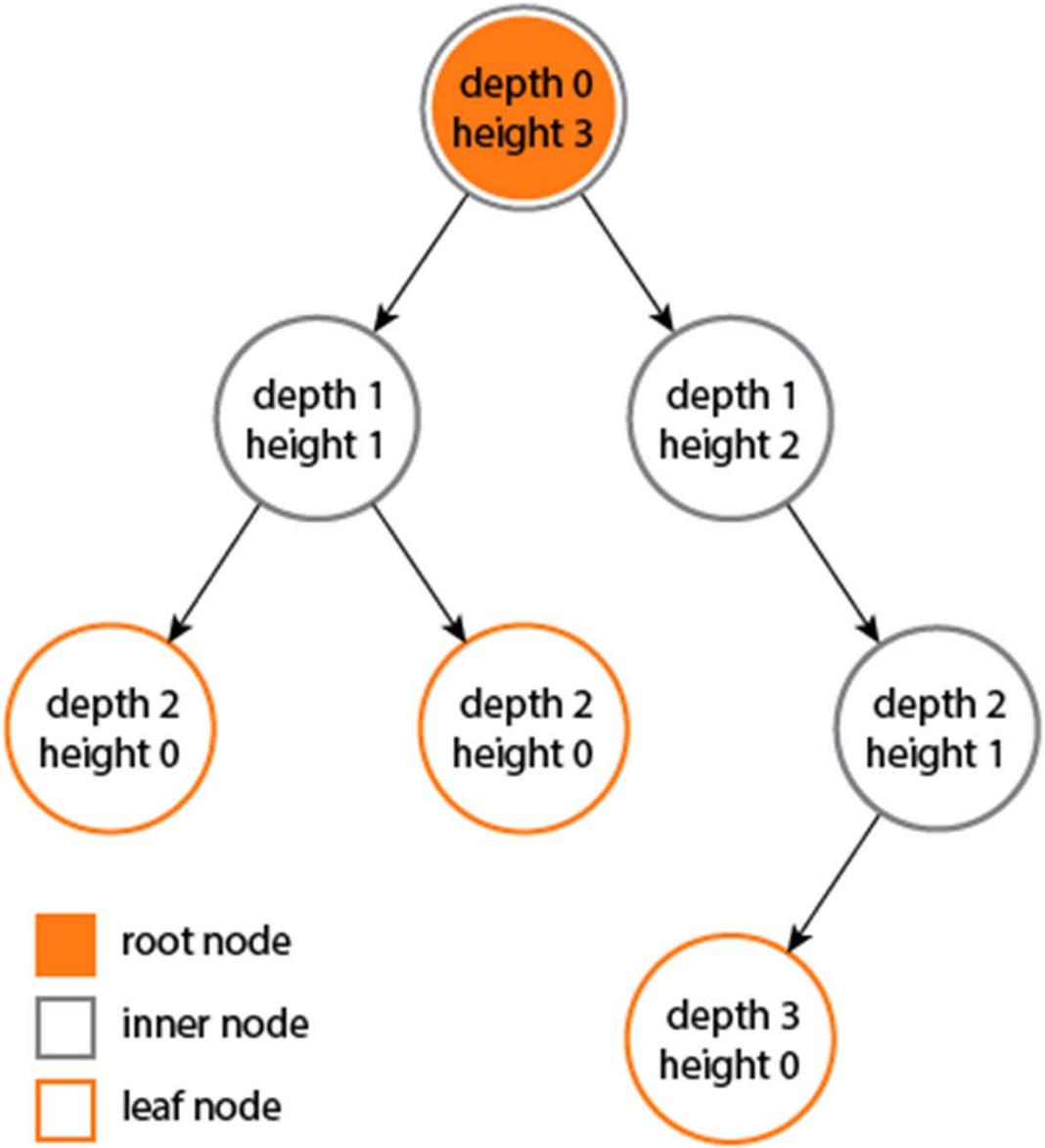


Depth (or Height) of a Binary Tree (Recapitulation)





- 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.

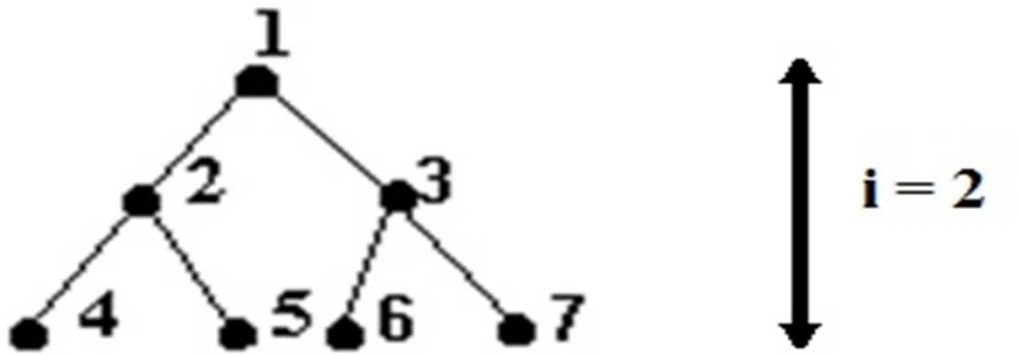
- The **height** of a node is the number of edges on the *longest path* from the node to a leaf. The lowest leaf node will have a height of 0.

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

Special binary trees

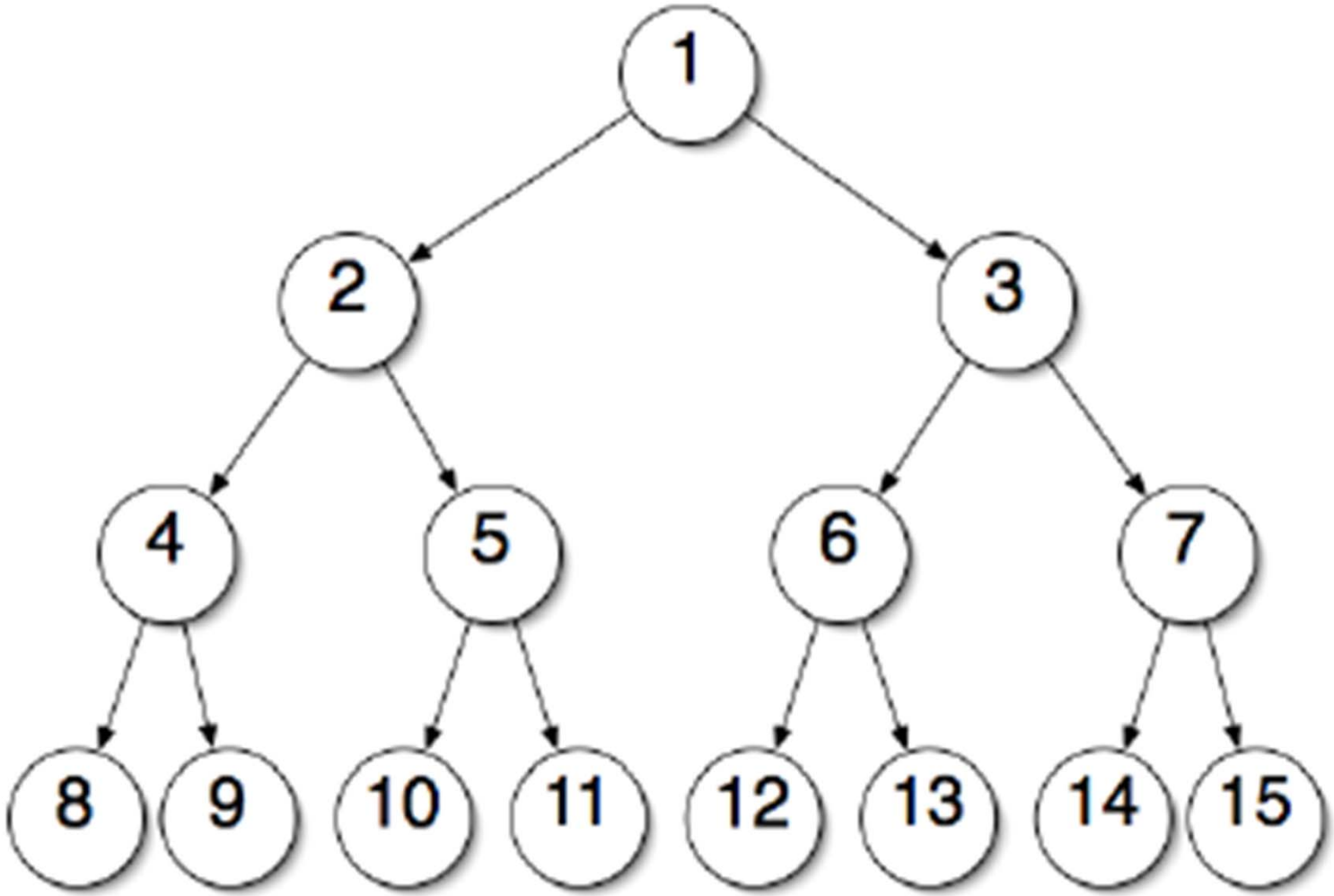
(Recapitulation)

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

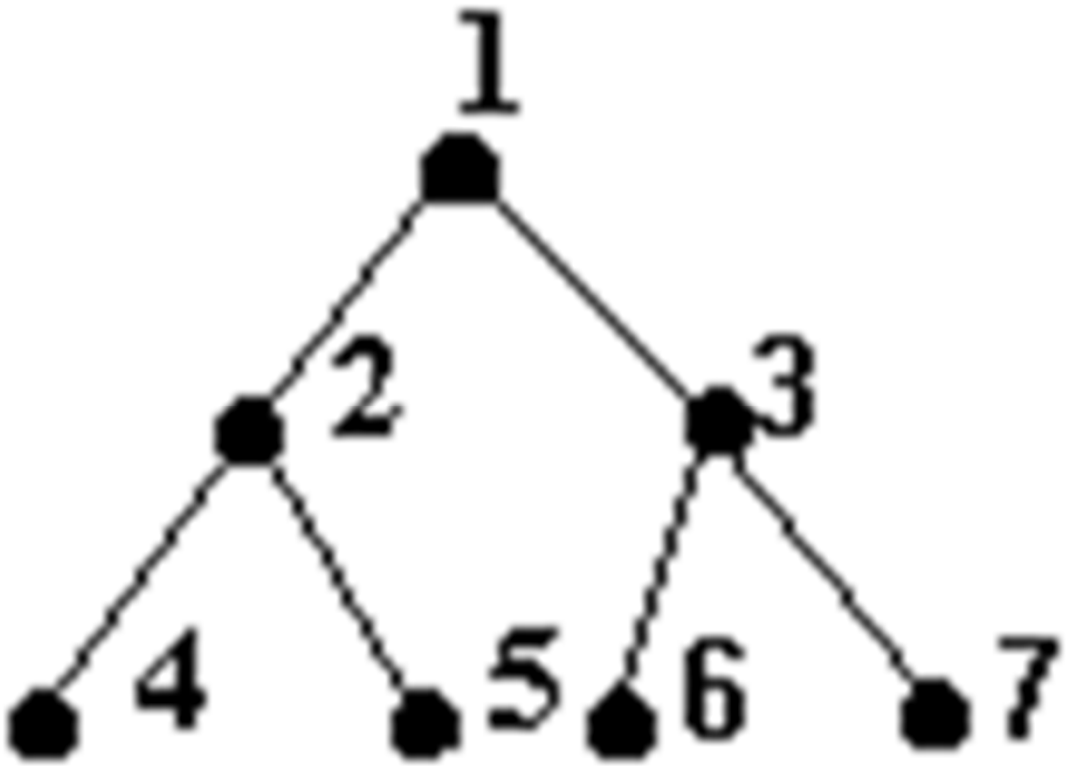
A full binary tree is the one that has the maximum number of vertices ($2^{i+1}-1$) for a specified height of **i**.

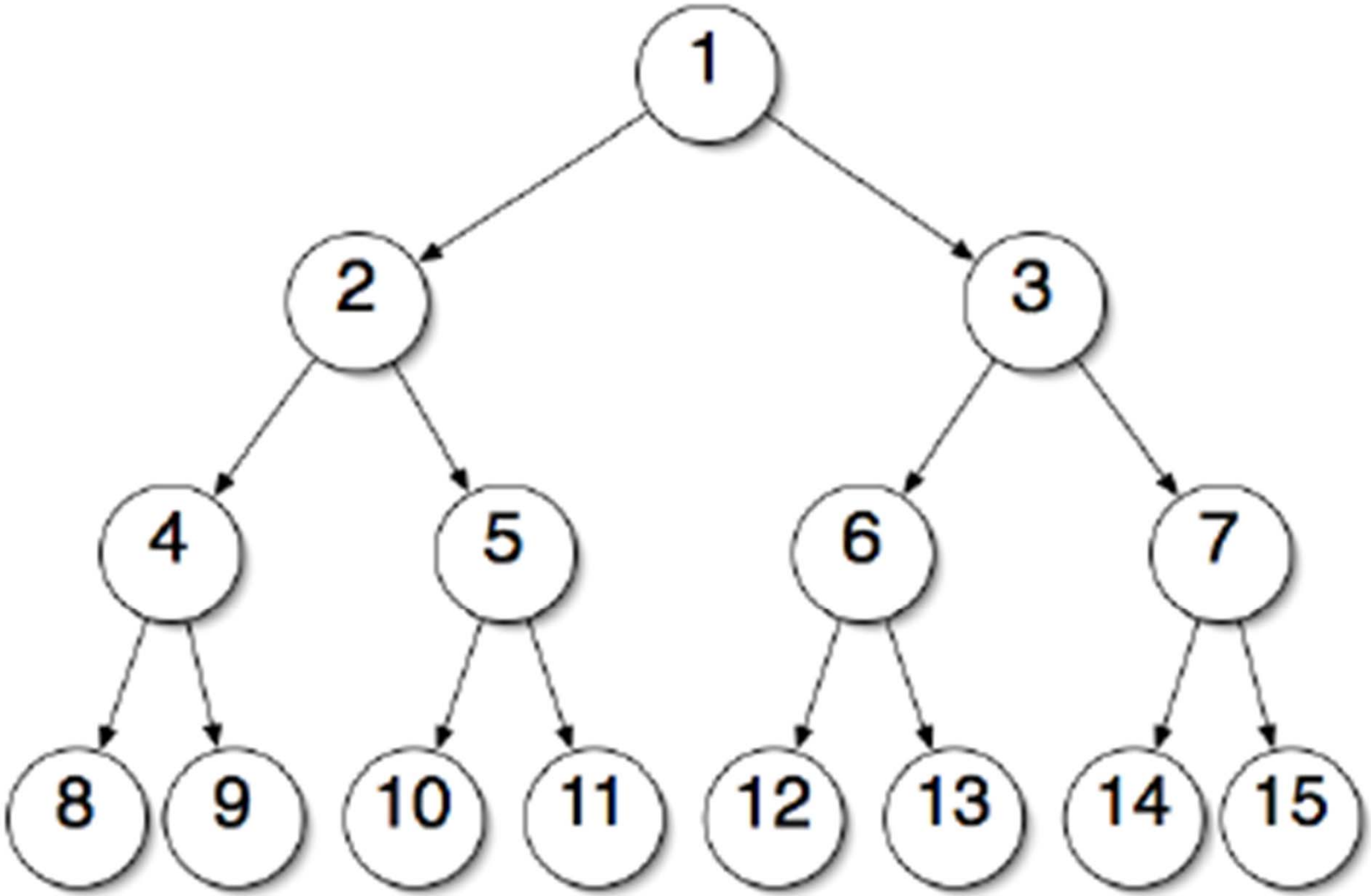
For example, a full binary tree of height 2 is as follows :

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

Notes:

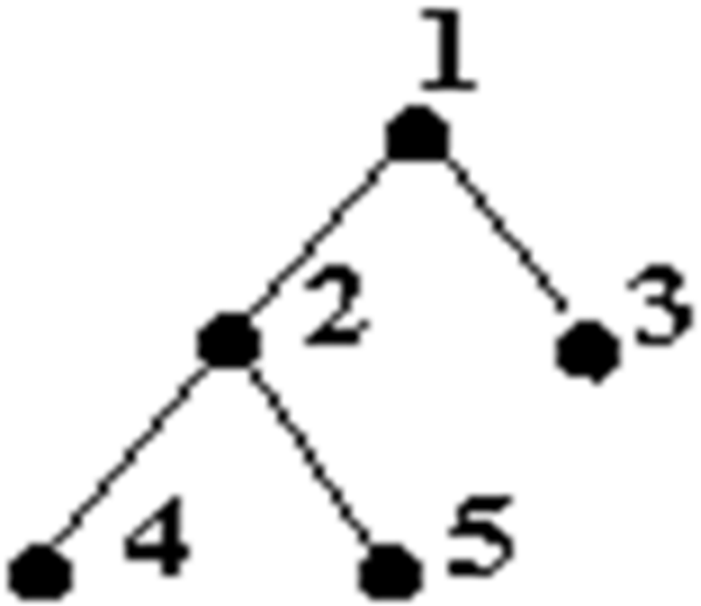
- The full binary tree nodes are numbered from the left to right according with their depth.




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

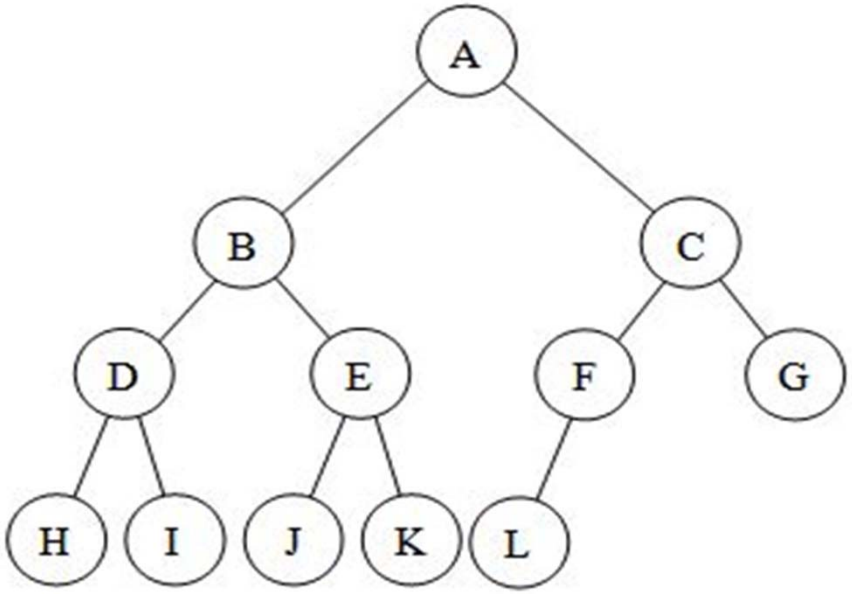
Complete binary tree
(Recapitulation)

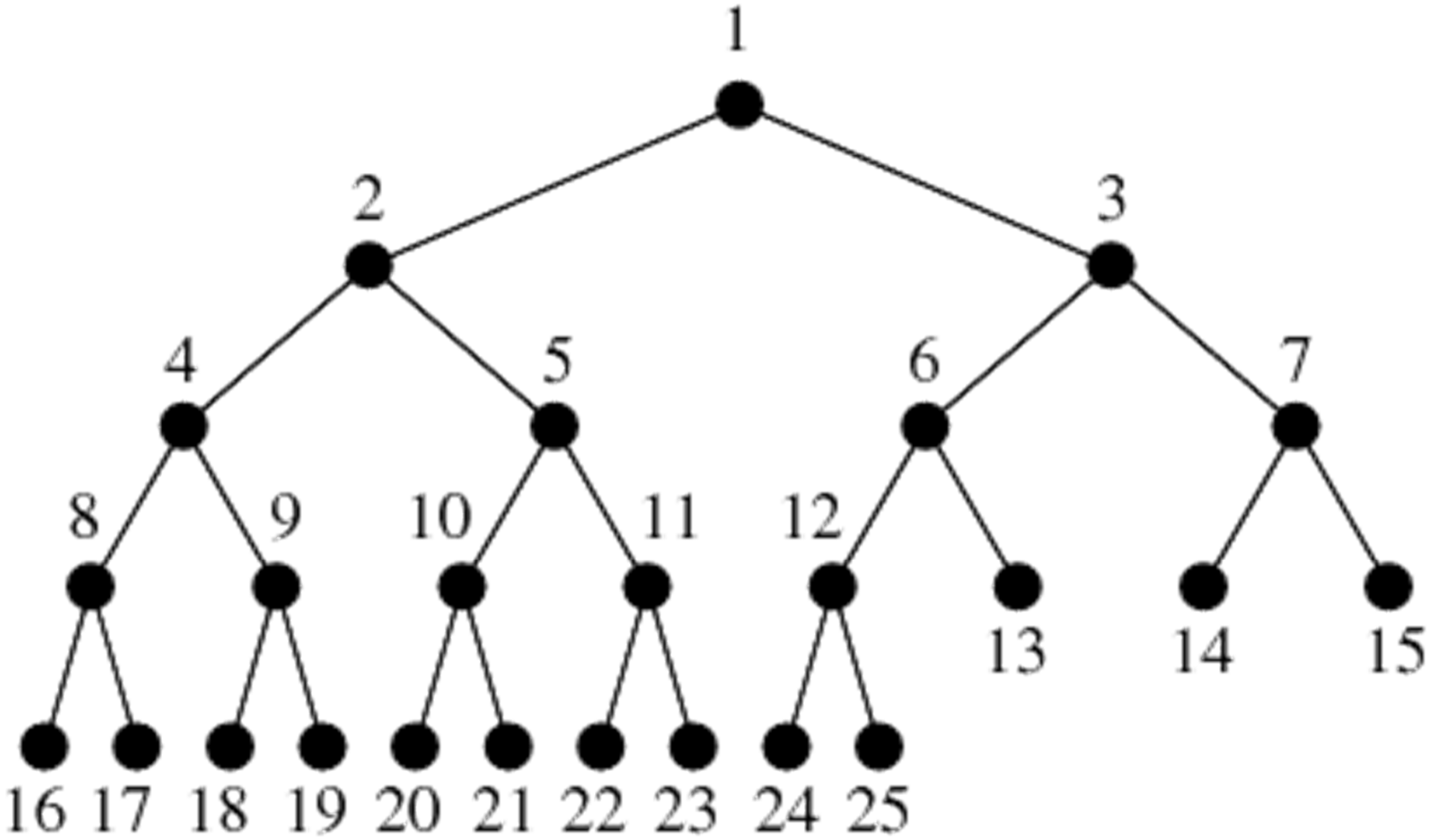
A binary tree with ***n*** nodes that has a height of ***i*** is called as being a **complete binary tree** if it is obtained from a full binary tree with a height of ***i***, in which there are eliminated the last consecutive nodes, numbered with ***n+1***, ***n+2***, ... up to $2^{i+1}-1$.



Notes:

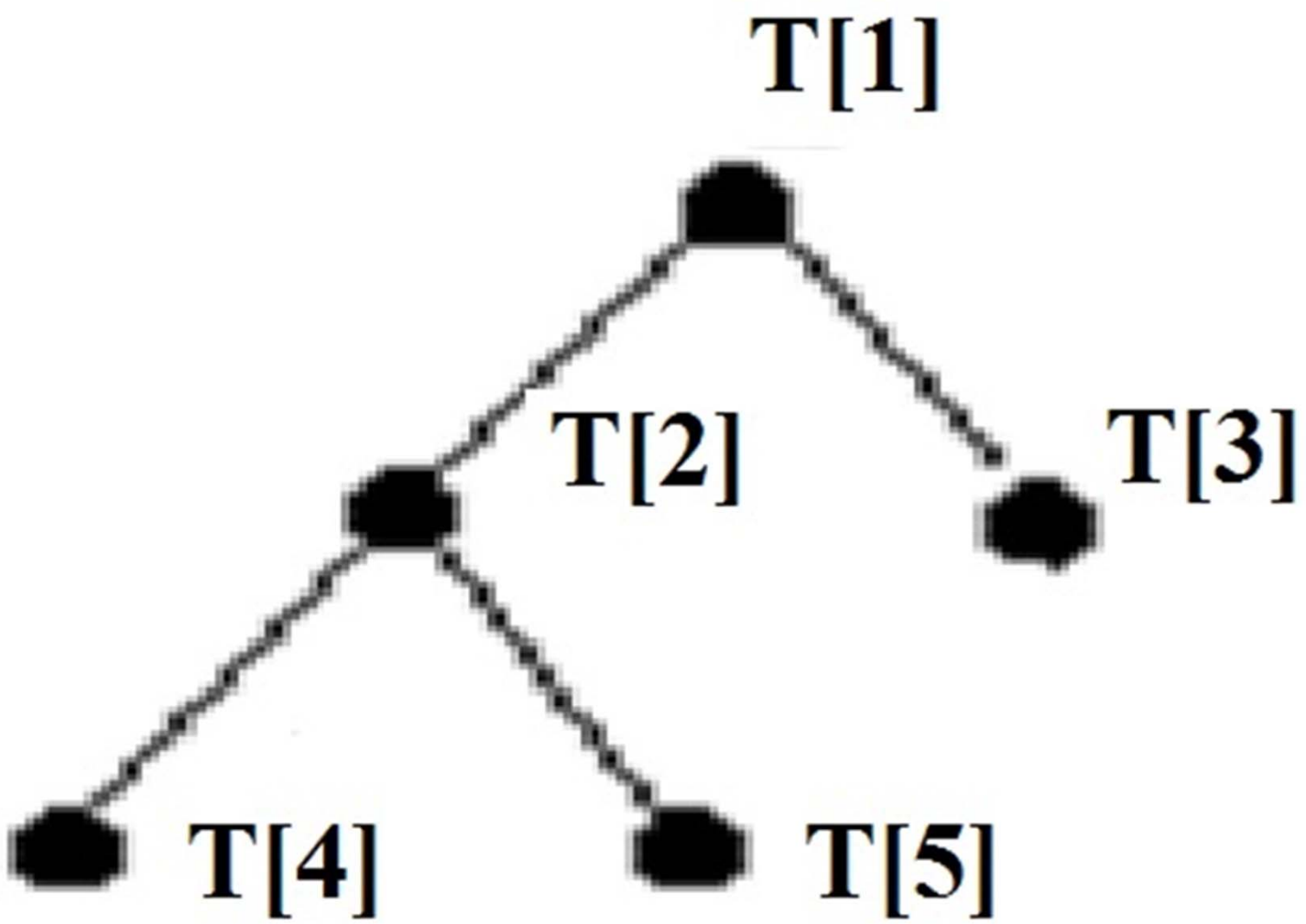
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.




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.



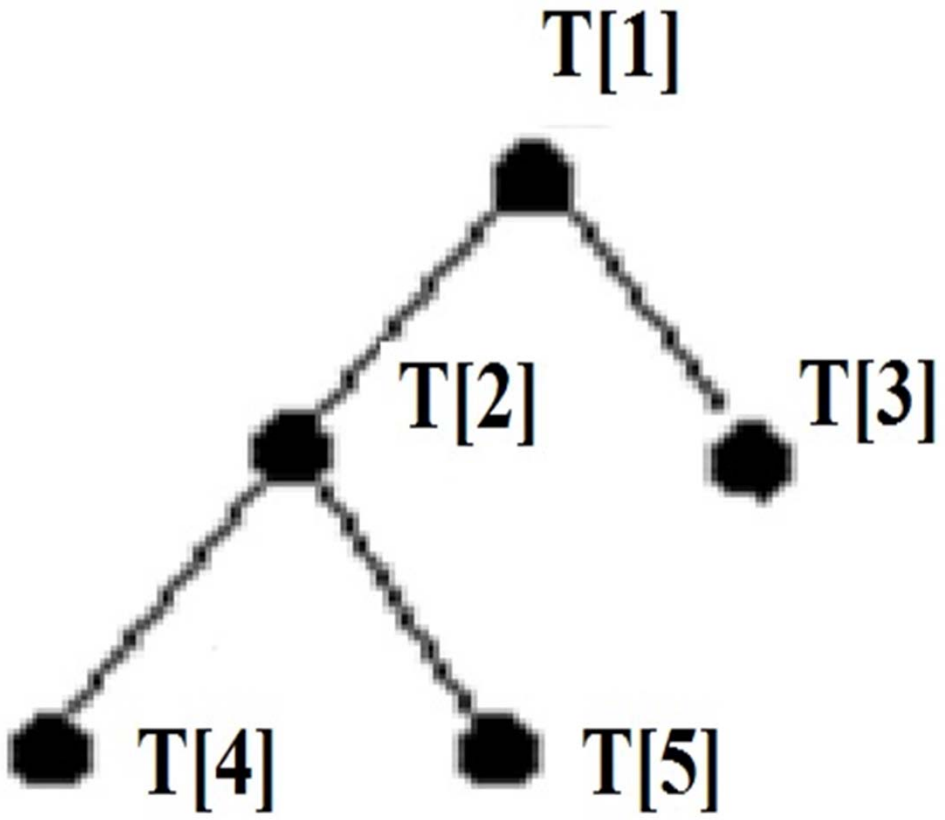
• **Warning**: This is a generic vector, which it begins with T [1] (not with T [0], as usual in the C programing).

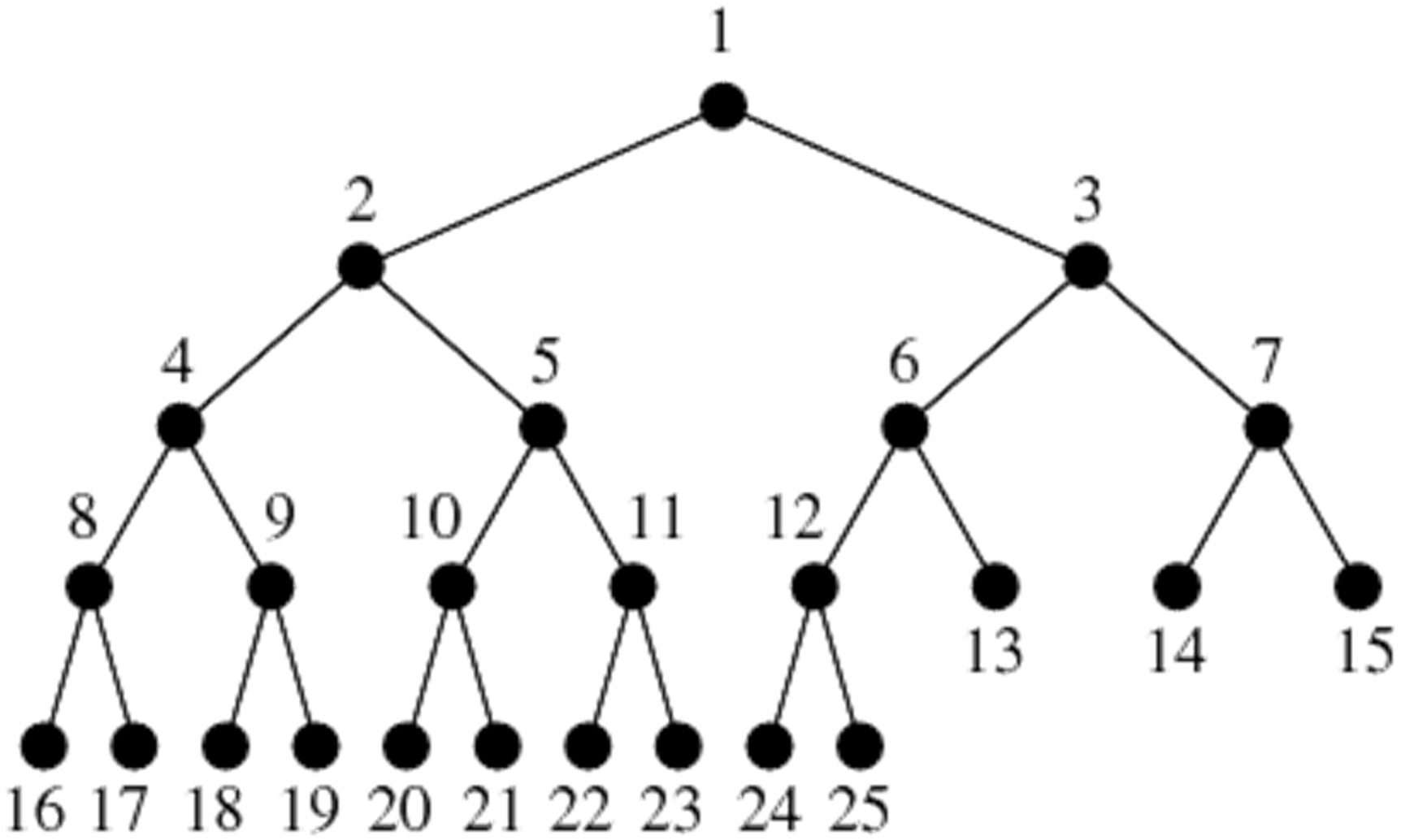
• We can make the necessary changes when we will write the code in C.

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:

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

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

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 (Recapitulation)

- 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

(Recapitulation)

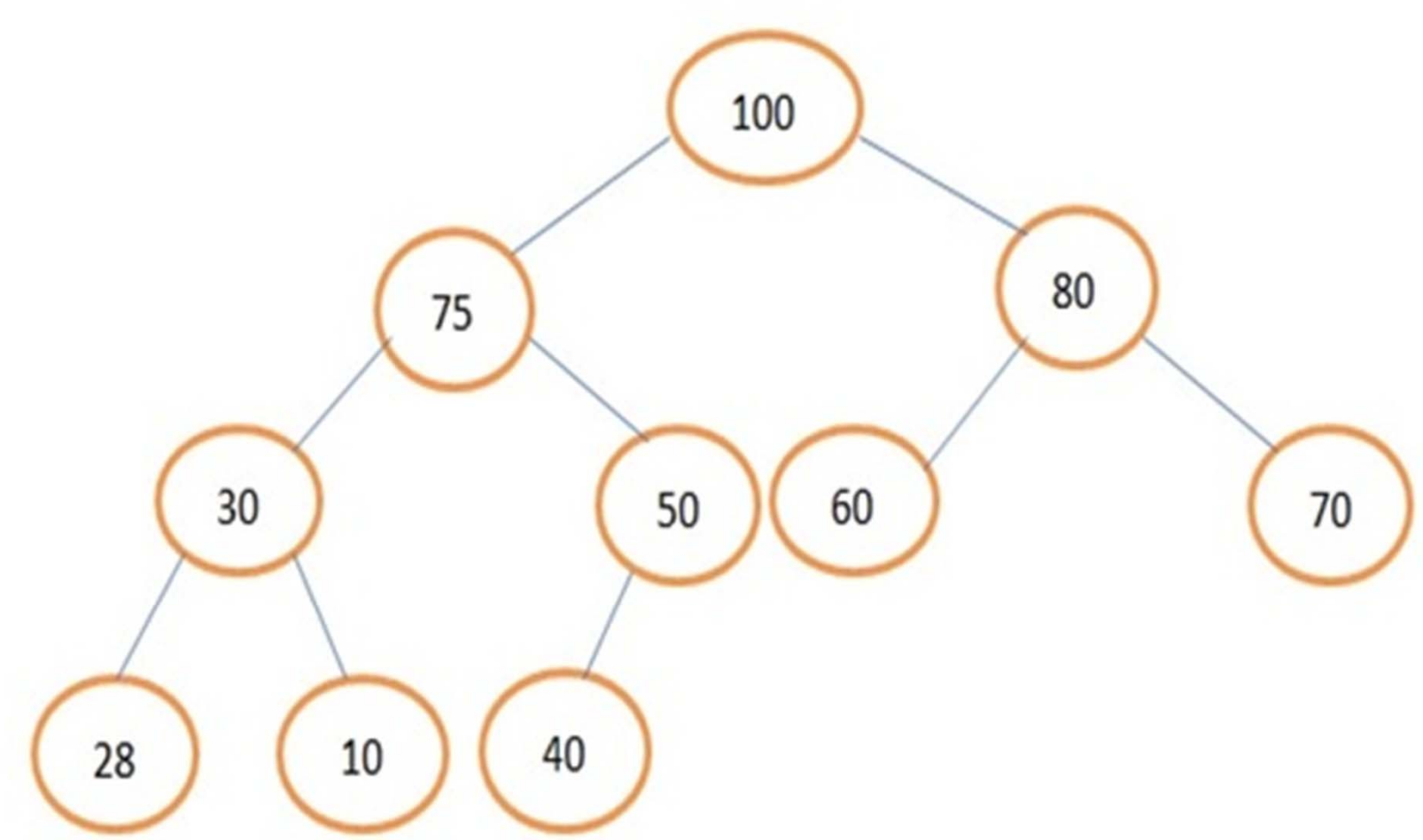
A **heap** is a specialized tree-based data structure that satisfies the **heap property**:

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

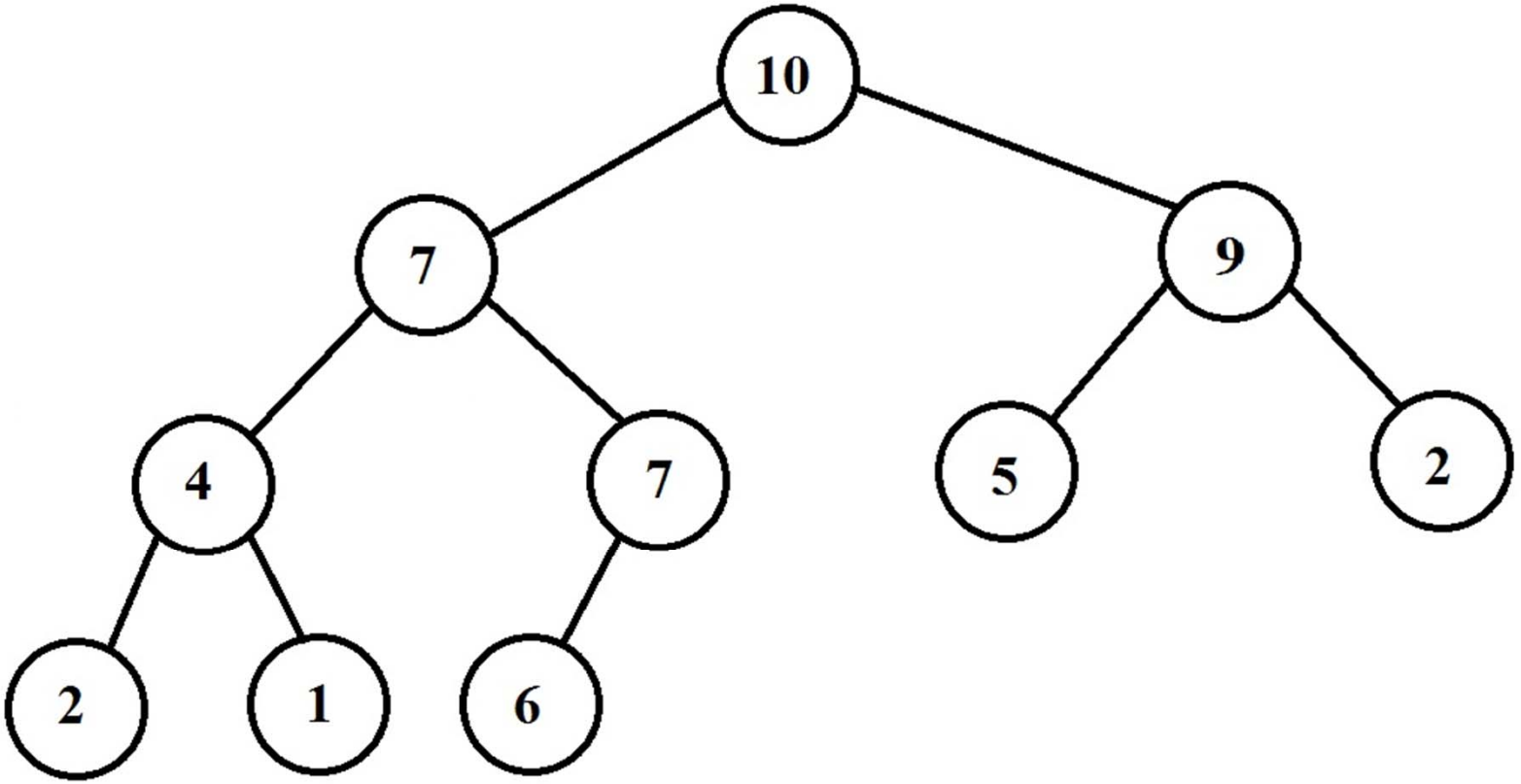
- Heaps can be classified further as either a "**max heap**" or a "**min heap**".
- In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node.
- In a min heap, the keys of parent nodes are less than or equal to those of the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

Restore the heap property

We consider T[1..n] as being a heap.

Having **i**, 1≤i≤n, we can assign to T[i] the value ν, and then we can restore the heap property.

```
void restore_heap (T[1..n], i, ν)
{local variable x;
  x ← T[i];
  T[i] ← ν;
  if  ν < x       then sift_down(T, i);
                  else sift_up(T,i);
}
```

The heap is a useful model for:

- Find the maximum item of a max-heap or a minimum item of a min-heap.
- Adding a new node to the heap.
- Change the value of a node (with **restore_heap**).

Previous operations can be used to implement a dynamic list of priorities:

- The node value of a corresponding element will indicate its priority.
- The event with the highest probability will always be at the root of the heap.
- The priority of a node can be changed dynamically.

These are some principles underlying the database programs.

Examples of useful functions:

1) **The function for finding the maximum value**:

```
find_maxim (T[1..n])
{  return T[1];
}
```

2) **The function to extract a maximum (and remove it):**

```
extr_max (T[1..n])
{ var loc x;
  x ← T[1];
  T[1] ← T[n];
  sift_down (T[1..n-1], 1);
  return x;
}
```

3) **Insertion of a new element in the heap**:

```
insert (T[1..n], ν)
{
    T[n+1] ← ν;
    sift_up (T[1..n+1], n+1);
}
```

Notes: **in C language**, it is not taken into consideration the maximum number of elements of an array used as a parameter function, so that, for example, for the position of sift-up, sift-down, etc., will have to take into account an additional parameter.

Example: **sift_down (T[ ], *n*, i)**

where **n** indicates the index of the last element of the vector (the numbering should start from **0** in C language!).

How can we create a heap from an unordered vector T[1..n] ?

- A less effective solution is to start from a heap of one element and add items one by one.

```
slow_make_heap(T[1..n])
{
   for (i=2; i ≤ n; i++)
        sift_up (T[1..i], i);
}
```

But there is another linear algorithm that works better (in terms of order / efficiency):

```
make_heap(T[1..n])
{
        for (i = n div 2; i ≥ 1; i - -)
                sift_down (T[ ], i);
}
```

Example:

- Starting from the following vector (which is not a heap):

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

Through a sequence of few steps we can obtain:

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]

Remember

- 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.

Note:

- All procedures, used to handle a heap, are changed accordingly to this new situation.

Warning!

- The heap data structure is very attractive, but it does have limitations.

- There are operations that can not be performed effectively in a heap.

The disadvantages of a heap:

- The tree has to be a complete one.

- Finding a peak with a certain value is inefficient (because there may be several nodes with the same value placed on different branches/ or levels in the heap).

• An extension of the concept of heap is possible for the complete trees with more than two children.

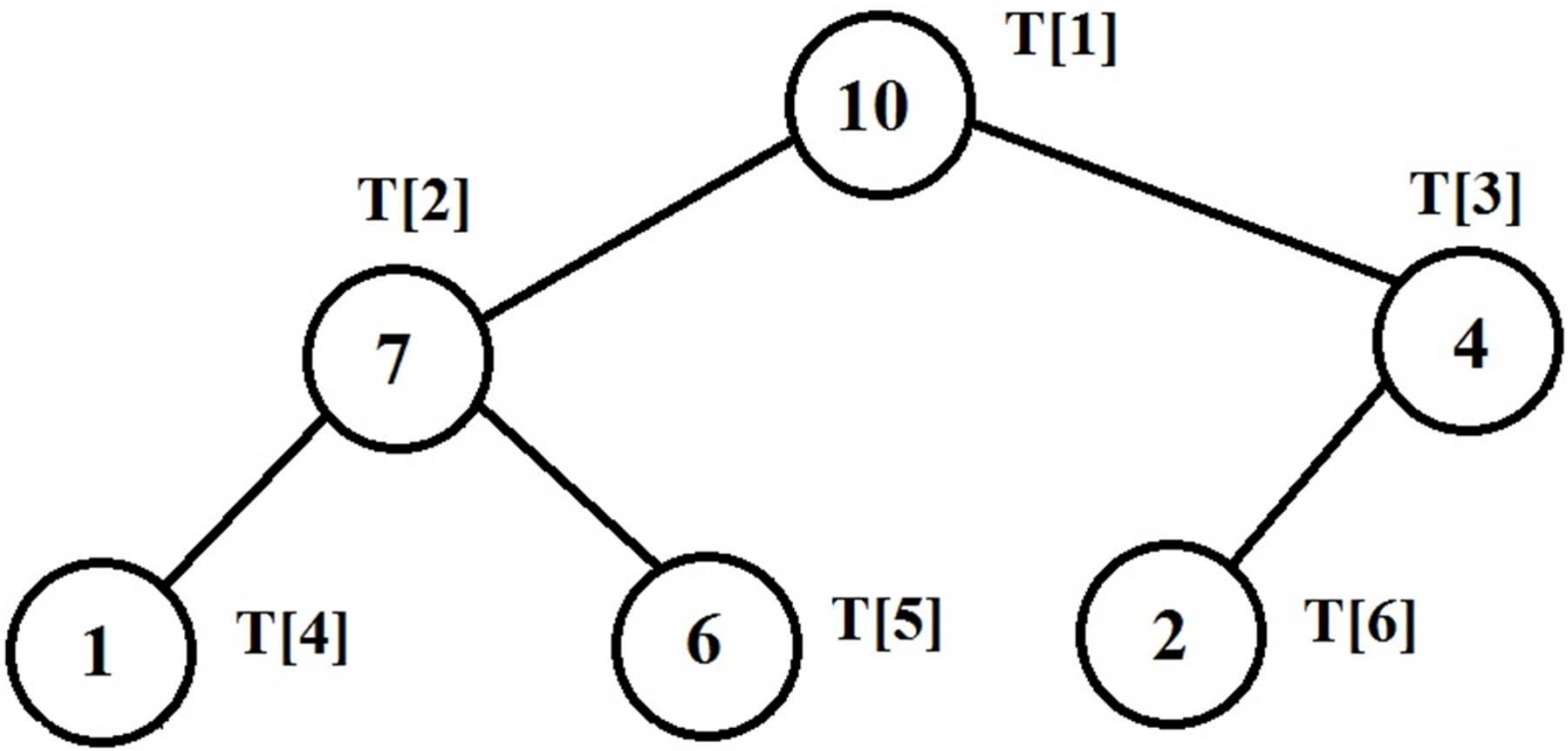
• Such an approach will accelerate the sift-down procedure.

The main application of the heap concept: A new sorting technique (**heapsort**)

- The **heapsort** algorithm was invented by J. W. J. Williams in 1964. In the same year, R. W. Floyd published an improved version that could sort an array in-place, continuing his earlier research into the treesort algorithm.

```
heapsort (T[1..n])
{
        make_heap(T);
        for( i = n; i ≥ 2; i - -)
                {
                 T[1] ↔ T[i];
                 sift_down (T[1…i-1], 1)
                }
}
```

Example:



Other ordering (or sorting) algorithms:

- **Insertion**
- **Selection**
- **Merge sort**
- **Quicksort**
- **etc.**

Ordering methods

- Ordering = arranging items of the same kind, class or nature, in an ordered sequence.

- For this purpose we consider that the data are a collection of items of a certain type and each item comprises one or even more values (variables), which are decisive in the ordering that is performed. Such a value is called **key**.

For the C programming language, a sorting algorithm can be achieved by one of the following methods:

1. Arranging data (which are sorted) so that their keys will finally correspond to the desired order.

2. By ordering an array of pointers (to the data that must be sorted) in order to form an ordered set.

Note:

- In the following we will discuss only about sorting unidimensional arrays (vectors) with numerical data.