Chapter 5

Hierarchical Structures

5.1 Trees

5.1.1 Introduction

In the tables we have:

+ Direct access by index (quick)

- Insertion and deletion require offsets

In linear linked lists we have:

+ Insertion and deletion are only done by modification of chaining

- Slow sequential access

Trees represent a compromise between the two:

+ Relatively quick access to an item from its key

- Inexpensive addition and deletion

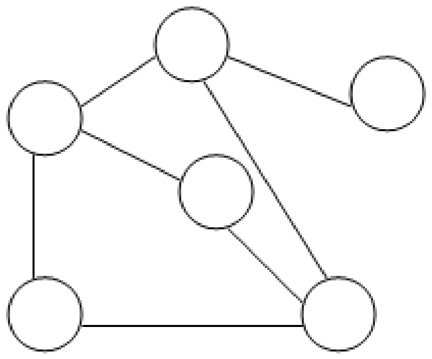
In addition, many computer processing operations are tree-like in nature such as family trees, hierarchy of functions in a company, representation of arithmetic expressions, etc.

5.1.2 Definitions

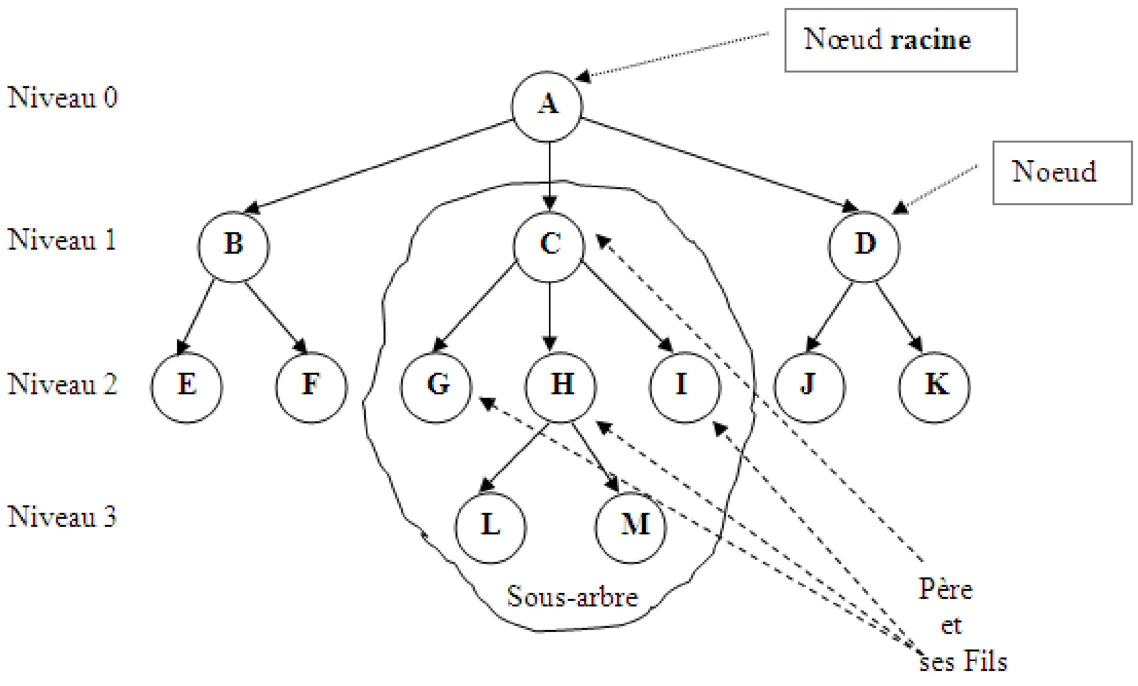
5.1.2.1 Definition of a tree

A tree is a non-linear structure, it is a graph without cycle where each node has at most one predecessor.

Graph



Tree



- The predecessor if it exists is called father (father of C = A, father of L = H)
- The successor if it exists is called son (son of A = { B,C,D }, son of H= {L,M })
- The node which has no predecessor is called root (A)
- The node which has no successor is called leaf (E,F,G,L,J,...)
- Descendants of C={G,H,I,L,M}, of B={E,F},...
- Ascendants of L={H,C,A t}, E={B,A},...

5.1.2.2 Size of a tree

This is the number of nodes it has.

- Size of previous tree = 13
- An empty tree is of size 0.

5.1.2.3 Level of a node

- Root level = 0
- The level of each node is equal to the level of its father plus 1
- Level of E,F,G,H,I,J,K = 2

5.1.2.4 Depth (Height) of a tree

- This is the maximum level in this tree.
- Depth of previous tree = 3

5.1.2.5 Degree of a node

- The degree of a node is equal to the number of its children.
- Degree of (A = 3, B =2, C = 3, E= 0, H=2,...)

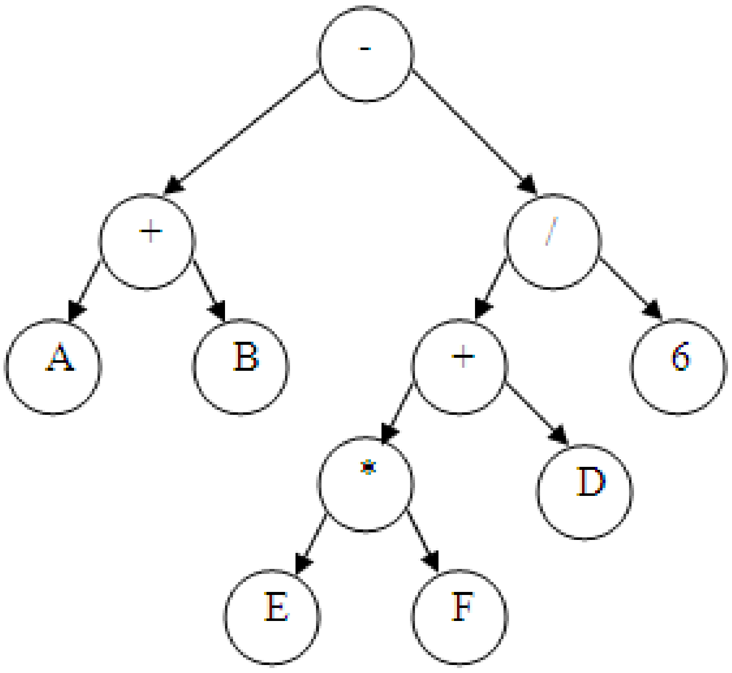
5.1.2.6 Degree of a tree

- This is the maximum degree of its nodes.
- Degree of previous tree = 3.

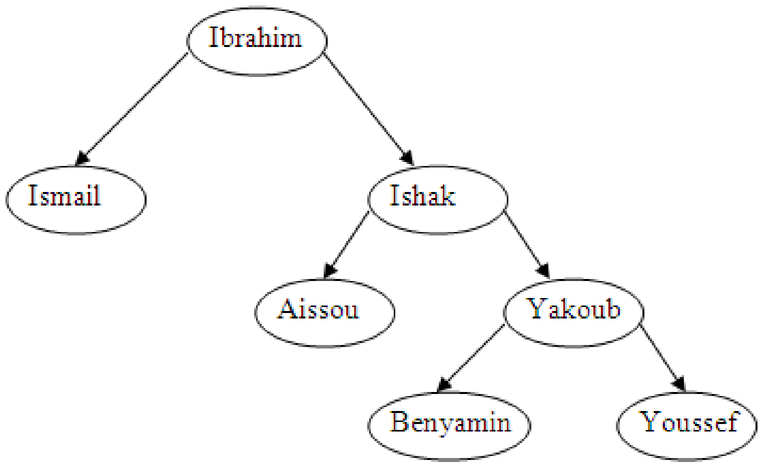
5.1.3 Using trees

- Representation of arithmetic expressions

(A+B)*c - (d+E*f)/6



- Representation of a family tree



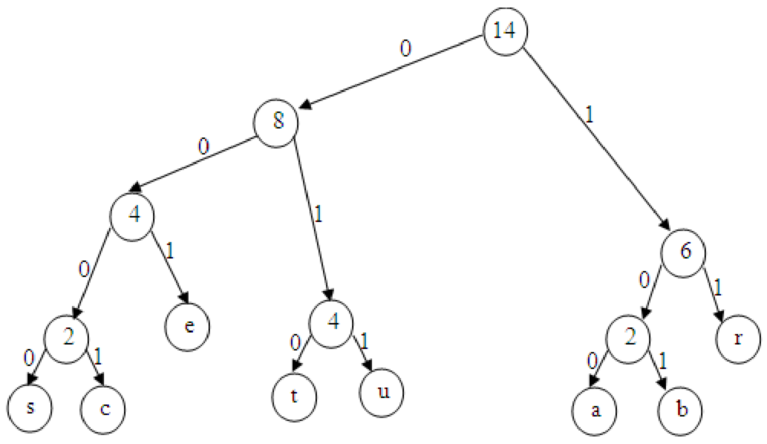
- Coding

  Example: code the string tree structure

  1. Constructs the character frequency table

Character	Frequency	Character	Frequency
s	1	c	1
t	2	e	2
r	4	a	1
u	2	b	1

2. Builds the code tree



3. Build the code table

Character	Code	Character	Code
s	0000	c	0001
t	010	e	001
r	11	a	100
u	011	b	101

4. Encode the string:

"tree structure" $\Rightarrow$ 0000 010 11 011 0001 010 011 11 001 100 11 101 11 001

5.1.4 Tree implementation

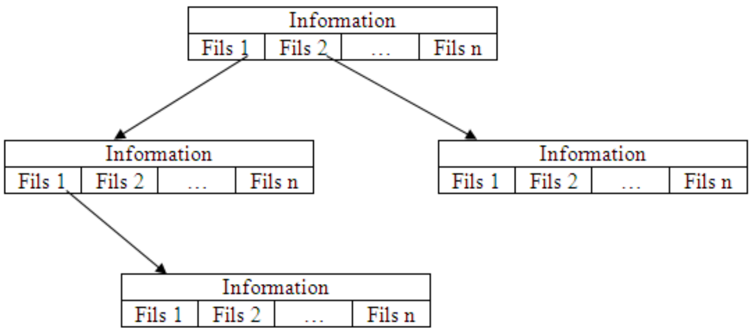
Trees can be represented by arrays, non-linear lists, or both:

5.1.4.1 Static representation

The tree in the first example can be represented by a table as follows:

Num	Information	Son 1	Son 2	Son 3
1	A	2	3	4
2	B	5	6	0
3	C	7	8	9
4	D	10	11	0
5	E	0	0	0
6	F	0	0	0
7	G	0	0	0
8	H	12	13	0
9	I	0	0	0
10	J	0	0	0
11	K	0	0	0
12	L	0	0	0
13	M	0	0	0

5.1.4.2 Dynamic Representation



```
Type  TNode = Structure
        Info : Basic_Type;
        Children : Array[1..NbChildren] of Pointer(TNode);
End;
Var  Root : Pointer(TNode);
```

5.1.5 Treatments on trees

5.1.5.1 Tree Traversals

The traversal of a tree consists of passing through all its nodes. There are two types of traversals:

1. Depth First Traversal (DFT)
   In a Depth First Traversal (DFT), you go as deep as possible into the tree then, once a leaf has been reached, you go back up to explore the other branches, starting with the "lowest" branch among those not yet covered. The algorithm is as follows:

```
Procedure DFT( node : Pointer(TNode));
Begin
    If (node ≠ Nil) Then
        For $i$ From 1 to NbChildren Do
            DFT($node->Children[i]$)
        End For;
    End If;
End;
```

Depth First Traversal can be done in two ways:

- Prefix DFT: where we display the father before his sons.
- Postfix DFT: where we display the children before their father.

The corresponding recursive algorithms are:

```
Procedure PrefixeDFT( node : Pointer(TNode));
Begin
    If (node$\neq$ Nil) Then
        write(node->Value);
        For $i$ From 1 to NbChildren Do
            PrefixeDFT($node->Children[i]$)
        End For;
    End If;
End;
```

```
Procedure PostfixeDFT( node : Pointer(TNode));
Begin
    If (node $\neq$ Nil) Then
        For $i$ From 1 to NbChildren Do
            PostfixeDFT($Fils_i$(node))
        End For;
        write($node->Children[i]$);
    End If;
End;
```

The prefix depth traversal of the tree in the first example gives:

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

While postfix depth traversal gives:

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

2. Breadth First Traversal (BFT)

   In a breadth first traversal, all nodes at level i must have been visited before the first node at level $i+1$ is visited. Such a traversal requires that we remember all the branches that remain to be visited. To do this, we use a queue.

```
Procedure BFT( node : Pointer(TNode));
Var  N : Pointer(TNode);
Begin
    If (node $\neq$ Nil) Then
        InitFile;
        Enfiler(node) ;
        While (Non(FileVide)) Do
            Défiler(N);
            Afficher(Valeur(N));
            For i From 1 to NbFils Do
                If ($Fils_i$(N) $\neq$ Nil) Then
                    Enfiler($Fils_i$(N));
                End If;
            End For;
        End While;
    End If;
End;
```

Applying this algorithm to the tree of the first example gives

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

5.1.5.2 Search an element

```
Function Search( node : Pointer(TNode); Val : Typeqq) : Boolean;
Var  i : integer;
      Exist : Boolean;
Begin
    If (node = Nil) Then
        Search ← False;
    Else
        If (Valeur(node)=Val) Then
            Rechercher ← Tue;
        Else
            i ← 1;
            Exist ← False;
            While ((i ≤ NbChildren) and not Exit) Do
                Exist ← Search($Fils_i$(Node), Val);
                i ← i + 1;
            End While;
            ; Search ← Exist;
        End If;
    End If;
End;
```

5.1.5.3 Calculating the size of a tree

```
Function Taille( node : Pointer(TNode)) : entier;
Var  i,S : entier;
Begin
    If (node = Nil) Then
        Taille ← 0;
    Else
        S ← 1;
        For i From 1 to NbFils Do
            S ← S + Taille($Fils_i$(node));
        End For;
        Taille ← S;
    End If;
End;
```

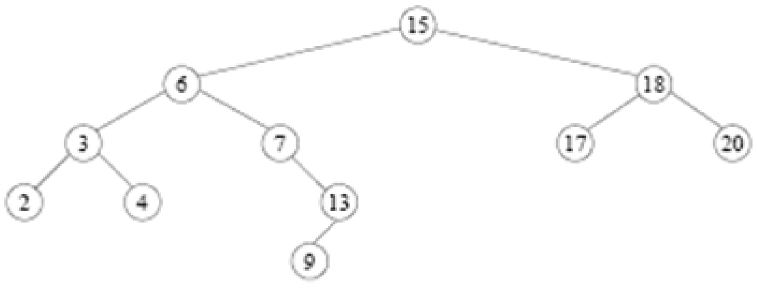
5.1.6 Typologie des arbres

- **m-arity Trees:** an m-arity tree of order n is a tree where the maximum **degree** of a node is equal to m.
- **B-Tree:** A B tree of order n is a tree where:
  - the root has at least 2 children
  - each node, other than the root, has between $n/2$ and n children
  - all leaf nodes are at the same level
- **Binary Trees:** this is a tree where the maximum degree of a node is equal to 2.
- **Binary search tree:** this is a binary tree where the key of each node is greater than those of its left descendants, and lower than those of its right descendants.

5.2 Binary search trees

5.2.1 Definition

Binary search trees are used to speed up search in m-ary trees. A binary search tree is a binary tree satisfying the following property: let x and y be two nodes of the tree, if y is a node of the left sub-tree of x, then $clé(y) \leq clé(x)$, if y is a node of the right sub-tree of x, then $cl\ acutee(y) \geq clé(x)$.



A node therefore has at most one left child and one right child.

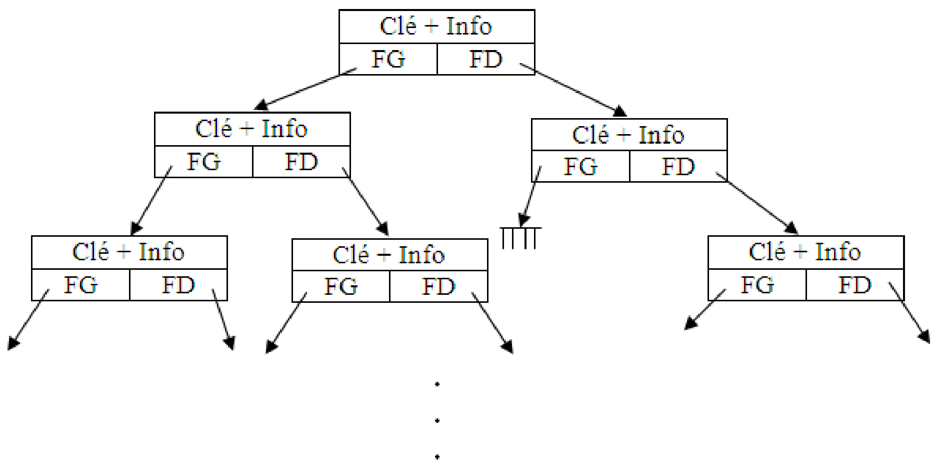
5.2.2 Implementation of BST (Binary Search Trees)

Binary search trees are implemented in the same way as m-ary ones (static or dynamic)

5.2.2.1 Static Representation

Num	Information	Left Child	Right Child
1	15	2	3
2	6	4	5
3	18	6	7
4	3	8	9
5	7	0	10
6	17	0	0
7	20	0	0
8	2	0	0
9	4	0	0
10	13	11	0
11	9	0	0

5.2.2.2 Dynamic Representation



```
Type TNode = Record
        Key : integer;
        Info : Basic_Type;
        LC,RC : Pointer(TNode);//the left child and the right child
End;
Var root : Pointer(TNode);
```

5.2.3 Basic algorithms on BST

5.2.3.1 Traversals

As with m-ary trees, the BST traversal can be done in **Depth** or **Breadth**:

- In Depth (Depth First Traversal)

```
Procedure DFT( node : Pointer(TNode));
Begin
    If (node $\neq$ Nil) Then
        PP(LC(node));
        PP(RC(node));
    End If;
End;
```

Listing the elements of the tree in depth can be done by:

- prefix (pre-order): father LC RC,
- infix (in-order) : LC father RC,
- postfix (post-order) : LC RC father.

```
Procedure prefixDFT( node : Pointer(TNode));
Begin
    If (node $\neq$ Nil) Then
        Write(node->LKey);
        prefixDFT(LC(node));
        prefixDFT(RC(node));
    End If;
End;
```

Trace : 15 6 3 2 4 7 13 9 18 17 20

```
Procedure infixDFT( node : Pointer(TNode));
Begin
    If (node $\neq$ Nil) Then
        infixDFT(node->LC);
        Write(node->Key);
        infixDFT(node->RC);
    End If;
End;
```

Trace : 2 3 4 6 7 9 13 15 17 18 20

```
Procedure postfixDFT( node : Pointer(TNode));
Begin
    If (node $\neq$ Nil) Then
        postfixDFT(node->LC);
        postfixDFT(node->RC);
        Write(node->LC);
    End If;
End;
```

Trace : 2 4 3 9 13 7 6 17 20 18 15

- In breadth (BFT: Breadth First Traversal)

```
Procedure BFT( node : Pointer(TNode));
Var  N : Pointer(TNode);
Begin
    If (node $\neq$ Nil) Then
        InitQueue; // this creates an empty queue
        Enqueue(node) ;
        While (Not(EmptyQueue)) Do
            Dequeue(N);
            Write(N->Key);
            If (N->LC $\neq$ Nil) Then
                Enqueue(N->LC);
            End If;
            If (N->RC $\neq$ Nil) Then
                Enqueue(N->RC);
            End If;
        End While;
    End If;
End;
```

5.2.3.2 Search

```
Function Serach( node : Pointer(TNode); xKey : integer) : Boolean;
Begin
    If ((node = Nil)) Then
        Search ← False;
    Else
        If ((node->Key)=xKey) Then
            Search ← True
        Else
            If ((node->Key) > xKey ) Then
                Search ← Search(node->LC);
            Else
                Search ← Search(node->RC);
            End If;
        End If;
    End If;
End;
```

5.2.3.3 Insertion

The element to be added is inserted where it would have been found if it had been present in the tree. The insertion algorithm therefore searches for the element in the tree and, when it comes to the conclusion that the element does not belong to the tree (the algorithm ends up on NIL), it inserts the element as child of the last node visited.

5.2.3.4 Suppression

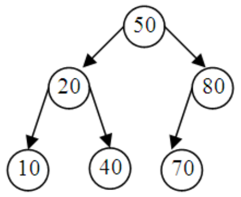
Several scenarios can be found: either to delete node N:

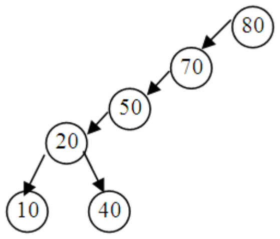
Cas			**Action**
LC(N)	**RC(N)**	**Example**	
Nil	Nil	Leafs (2,4,17)	Replace N by Nil
Nil	$\neq$ Nil	7	Replace N by RC(N)
$\neq$ Nil	Nil	13	Replace N by LC(N)
$\neq$ Nil	$\neq$ Nil	6	1- Find the smallest descendant to the right of N let be P(7) 2- Replace Value(N) by Value(P) (6 $\leftarrow$7) 3- Replace P by RC(P) (7 $\leftarrow$13)

Exercise: Give the tree after removing 3 then 15.

5.2.4 Balancing

consider the two following BSTs (Binary Search Trees):





These two BSTs contain the same elements, but are organized differently. The depth of the first is less than that of the second. If we are looking for element 10 we will have to browse 3 elements (50, 20, 10) in the first tree, on the other hand in the second, we will have to browse 5 elements (80, 70, 50, 20,10). The first tree is said to be more balanced. If we calculate the complexity of the search algorithm in a binary search tree, we will find O(h) where h is the height of the tree. So the more balanced the tree, the lower the height and the faster the search.
We say that an BST is balanced if for any node of the tree the difference between the height of the left sub-tree and the right sub-tree is at most equal to 1. It is always advisable to work on a balanced tree to guarantee the fastest possible search. The balancing operation can be done each time a new node is inserted or each time the imbalance reaches a certain threshold to avoid the cost of the balancing operation which requires a reorganization of the tree.

5.3 Heaps

5.3.1 Introduction

To implement a priority queue, often used in operating systems, we can use:

- An ordinary queue (without priority), insertion will then be simple (at the end) in O(1), but removal will require searching for the highest priority element, in O(n).
- A sorted array (or list) where deletion will be O(1) (the first element), but insertions will require O(n).

Heaps provide the solution to this problem.

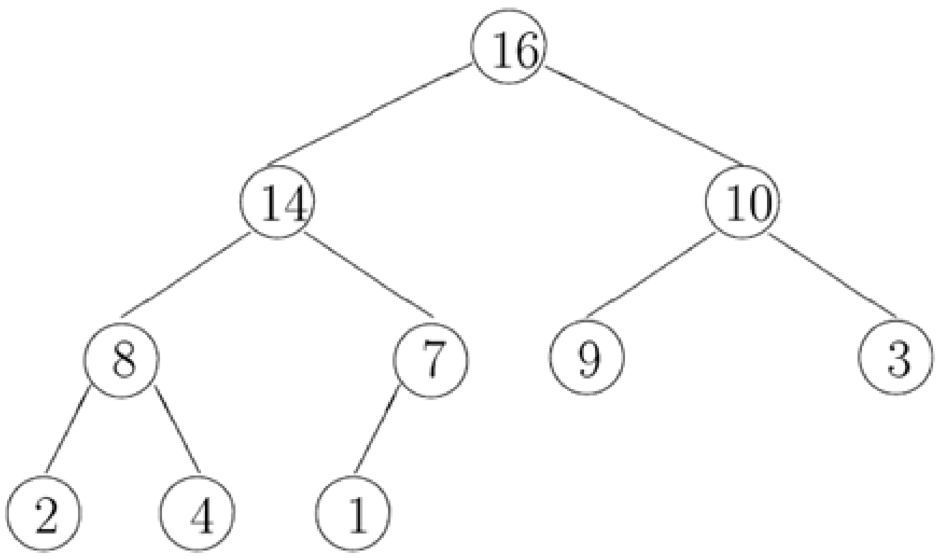
5.3.2 Definition

A heap is a tree that satisfies the following two properties:

1. It is a complete binary tree, that is to say a binary tree in which all levels are filled except possibly the last where the elements are arranged as far to the left as possible.
2. The key of any node is greater than that of its descendants.

The highest priority element is therefore always at the root.

Example of a heap:



5.3.3 Heap operations

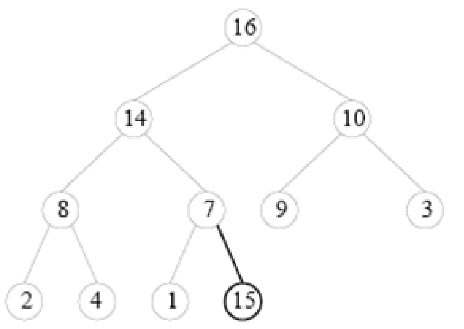
To implement a priority queue as a heap, we must define the add and delete operations.

5.3.3.1 Addition

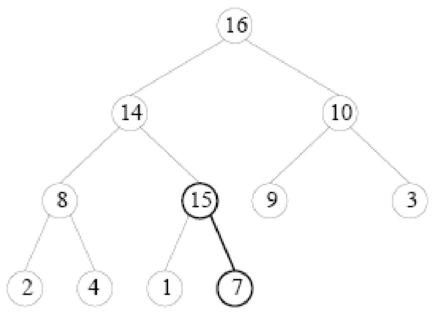
To add a new element to the priority queue, i.e. to the heap, we must:

1. Create a node containing the value of this element,
2. Attach this node in the last level in the first empty place as far to the left as possible (create a new level if necessary). We always obtain a complete binary tree but not necessarily a heap.
3. Compare the key of the new node with that of its father and swap them if necessary, then repeat the process until there are no more elements to swap.

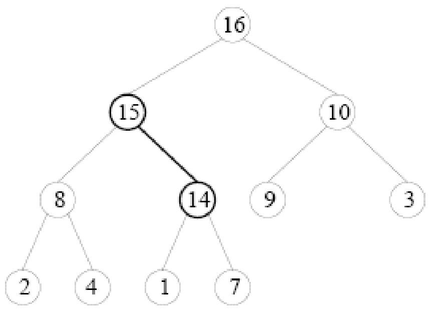
Example: either to add priority element 15:



- 1 -



- 2 -



- 3 -

The complexity of this operation is $O(h = Log_2(n))$ if n is the number of elements.

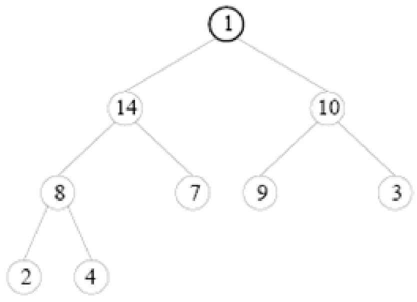
5.3.3.2 Deletion

The highest priority element is always at the root, so removal (i.e., removal) involves reading the root and then deleting it. To do this we must:

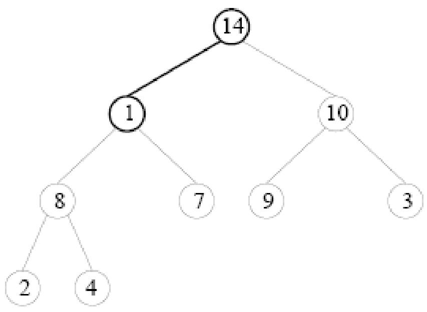
1. Replace the value of the root with the value of the rightmost item in the last level.
2. Delete this element from the tree (rightmost in the last level), we obtain a binary tree but not necessarily a heap.

3. We compare the value of the root with the values of its two children and we permute it with the largest. We start the process again until there are no more elements to swap.

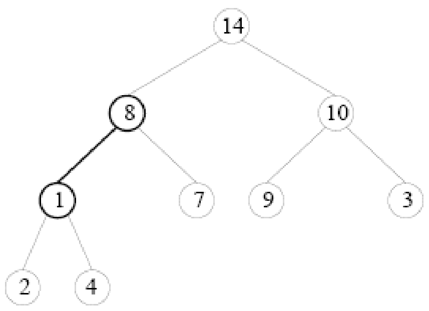
Exzmple:



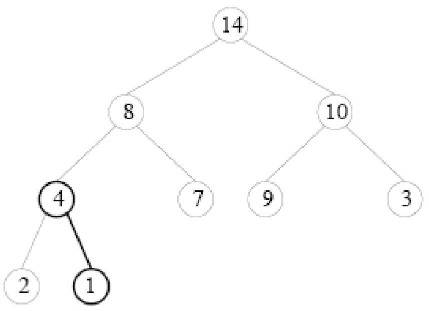
- 1 -



- 2 -



- 3 -

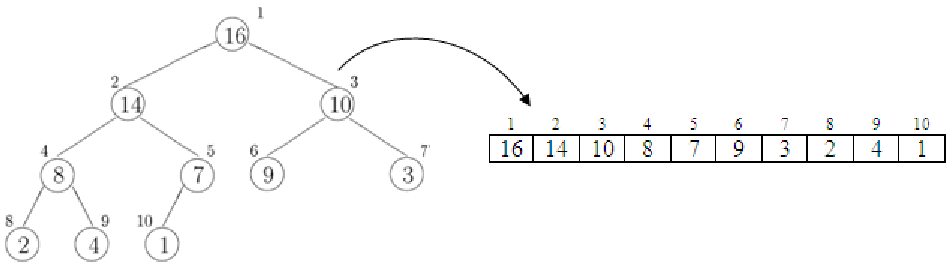


- 4 -

The deletion is also in $O(h = Log_2(n))$.

5.3.4 Heap implementation

Heaps can be implemented dynamically exactly like trees. However, a very effective static representation using arrays is widely used in practice, it consists of arranging the elements of the heap in an array according to a breadth first traversal:



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

We notice in the resulting table that the left child of an element with index i is always

found if it exists at position $2i$, and its right child is found at position $(2i + 1)$ and its father is at position $i/2$. Add and remove operations on the static heap are done in the same way as in the case of the dynamic heap. With this principle the addition and removal operations are carried out in a very simple and extremely efficient manner.
Heaps are used even for sorting arrays: we add the elements of an array to a heap, then remove them in descending order.