CHAPTER 3: TREES

Contents

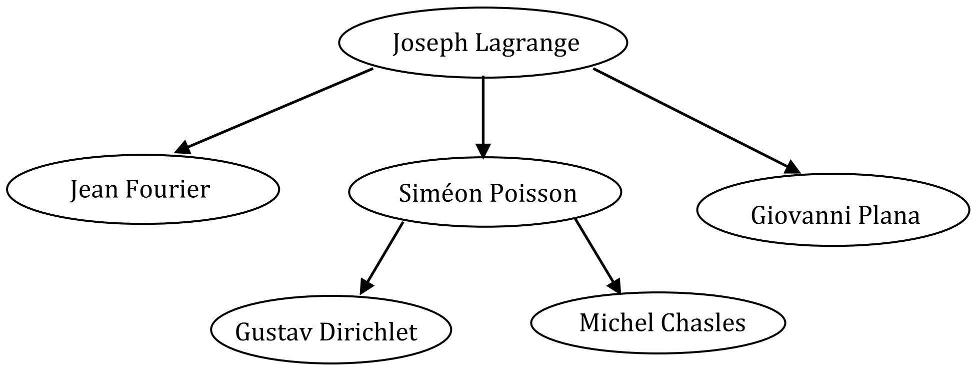
1. Reminders

For large amounts of data (Big data), the linear complexity of list operations is prohibitive. We study in this chapter, hierarchical or tree data structures for which the running time of most operations is of the order of O(logN). We start by recalling the definitions relating to trees in general and binary trees in particular, then we show how to use binary trees to develop an efficient search algorithm with a complexity of O(logN). We end with another application of binary trees namely heaps as well as a new efficient sorting algorithm called heap sort.

Definitions

A tree is defined recursively as a structure composed of elements called nodes linked by a "Parent/Child" relationship; it contains a particular node called root (the only one that has no parent node), as well as an ordered sequence (which can be empty) of disjoint sub-trees $A_1, A_2, \ldots., A_m$**.** A node can take any type of data: simple (numeric, character, etc.) or composite (structure, class), etc.

A typical example of a tree is the family tree where the nodes represent people and the relationship between nodes is the classic “kinship” relationship. The following figure illustrates the schematization of part of the mathematical family tree (http://www.genealogy.ams.org). The relation "kinship" is interpreted as "student or doctoral student of":



It is worth recalling the following definitions relating to trees:
- The **children** of a node are the roots of its sub-trees, for example the "doctoral students" of Lagrange are Fourier, Poisson and Plana.
- The **parent** of a node (except the root) is the unique node of which it is a child.
- An **internal node** is a node other than the root which has at least one child, for example Poisson.
- A **leaf** is a node that has no children.

- The **ancestors** of a node are the nodes that connect it to the root, including the root and the node itself.
- The **descendants** of a node are the nodes which belong to the subtree having this node as root.
- The **depth** of a node is the length of the path connecting it to the root.
- The **height** of a tree is the maximum depth of its nodes.
- The **size** of a tree is the total number of its nodes.

Tree traversal procedures

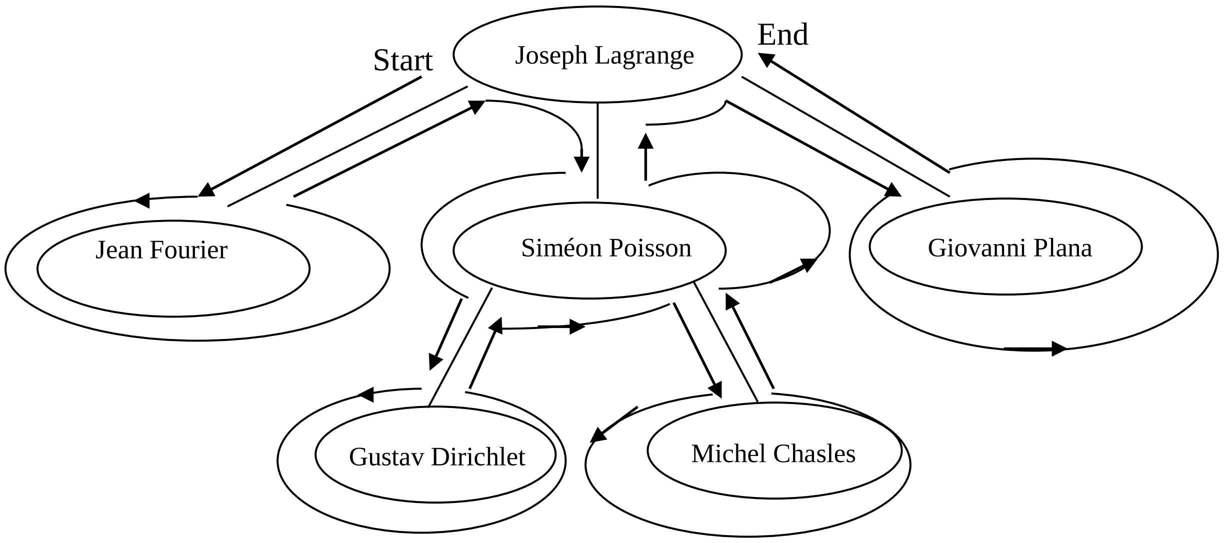
It is about examining once and only once all the nodes of a tree. It is worth mentioning:

Preorder (prefix) traversal "Root-Left-Right policy"

This involves recursively implementing the following steps:

- Explore the root
- **Preorder** traversal of the first subtree (the leftmost)
- …
- **Preorder** traversal of the last subtree (the rightmost)

In the previous example, we explore the nodes *Lagrange, Fourier, Poisson, Dirichlet, Chasles, Plana*.



The figure above illustrates the preorder traversal as an examination of the nodes from the first encounter only.

Postorder (postfix) traversal "Left-Right-Root policy"

In this traversal, we implement recursively the following steps:

- Postorder traversal of the first subtree
- …
- …
- Postorder traversal of the last subtree
- Explore the root

The postorder traversal of the previous tree lead to: *Dirichlet, Chasles, Poisson, Plana, Lagrange*. This operation is similar to the previous one except that the examination of the nodes is done at the last encounter.

Level order traversal

This traversal consists of exploring the tree from the root (level 0), then the nodes of level 1 from left to right and so on. This traversal order is considered as a breath traversal while preorder and postorder procedures are considered as a depth traversal.

The level order traversal of the previous tree lead to *Lagrange, Fourier, Poisson, Plana, Dirichlet, Chasles*.

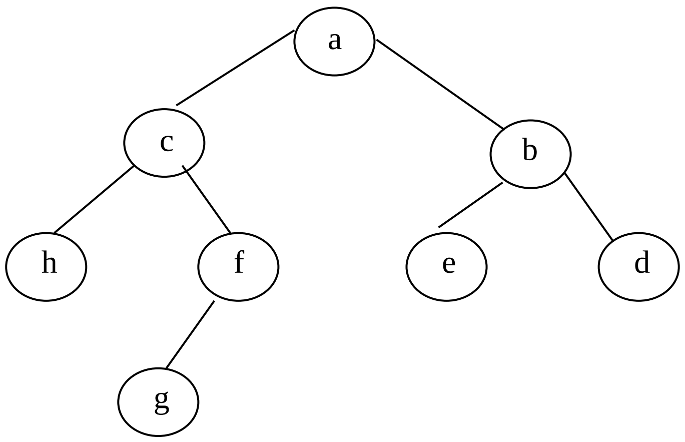
Some operations on trees

We can cite among the operations on trees:

- **initTree()**: return an empty tree.
- **parent (n,T)**: function that return the parent of the node n of tree T. In the case of root, a null node is returned.
- **leftmostChild(n,T)**: return the leftmost child of the node n of tree T. In the case of a leaf, the function return a null node.
- **rightmostChild(n,T)**: same but return the rightmost child.
- **root(T):** return the root node. In the case of an empty tree, the function return the null node.
- **isLeaf(n,T)**: tests if node n is a leaf.
- **nbLeaves(T)**: return the number of leaves of the tree T.

2. Binary trees

A binary tree can be either empty or composed of the root node which possibly points to two subtrees at most. At each level of the binary tree, the two disjoint binary subtrees are called left subtree and right subtree. A binary tree often denotes a non-empty tree. The traversal procedures for general trees remain valid for binary trees and the terminology used is slightly modified to take account of certain specificities of binary trees. Indeed, when we designate the children of a node, we talk about the left child and the right child in the case where one or both exist.



Note: A tree is called:

- **homogeneous** when all nodes have two or zero children.
- **degenerated** if all nodes have only one child.
- **complete** if each level of the tree is completely filled (that is level i contain 2^i nodes).
- **perfect** when all the levels except possibly the last are filled, and the leaves of the last level are grouped from the left.
- **balanced** if for each node the difference between the left subtree height and the right subtree height does not exceed 1.

To implement a binary tree, each node is associated with a data structure containing the data and two pointers that point to the two child nodes. By convention, a null pointer indicates an empty tree (or subtree). In this way, it is sufficient to memorize the address of the root.

2.1. Data structures

Using arrays (Static representation)
Binary trees can be represented by the following data structure:

```
const int MaxNodes=1000;

struct Node{
 int element;
 int leftChild;
 int rightChild;
};

Node Tree[MaxNodes];
```

Using pointers (Dynamic representation)
Instead of using integer fields to point to the left and right children, pointers can be used as follows:

```
struct Node {
 int element;
 Node *leftChild, *rightChild, *parent;
};
```

Thus a binary tree is represented by a linked structure in which each node is an object. In addition to the element (key) and pointer fields of the data, each node contains three fields: leftChild, rightChild and parent. These fields point to the left subtree, right subtree and the parent of the node respectively if they exist. Otherwise, one or more of these fields contains NULL. The root is the only node whose parent field is NULL.

The parent pointer facilitates the traversal "leaves → root". Another variant consists in deleting this pointer to save memory space and to simplify the update procedures, but to the detriment of the traversal of the tree. Thus we can represent the binary tree more simply as follows:

```
struct Node {
 int element;
 Node *leftChild, *rightChild;
};
```

Note: A binary tree is initialized by: **`Node *root=NULL;`**

2.2. Binary tree traversal

In addition to the pre-order and post-order traversals already studied for general trees, a third type of depth traversal can be used in the case of binary trees: The in-order (infix) traversal.

⇒ **PREORDER (PREFIX) TRAVERSAL: "Root-Left-Right policy"**
This involves recursively implementing the following steps:
- Explore the root
- Preorder traversal of the left subtree
- Preorder traversal of the right subtree

The preorder traversal of the given binary tree is: a, c, h, f, g, b, e, d.

⇒ **INORDER (INFIX) TRAVERSAL: "Left-Root-Right policy"**
This involves recursively implementing the following steps:
- Inorder traversal of the left subtree
- Explore the root
- Inorder traversal of the right subtree

The inorder traversal of the given binary tree is: h, c, g, f, a, e, b, d.

⇒ **POSTORDER (POSTFIX) TRAVERSAL: "Left-Right-Root policy"**
This involves recursively implementing the following steps:
- Postorder traversal of the left subtree
- Postorder traversal of the right subtree
- Explore the root

The postorder traversal of the given binary tree is: h, g, f, c, e, d, b, a.

Theses traversals can be easily implemented with the following routines:

```
void preorderTraversal(Node *root)
{ if (root!=NULL) {
    cout<<root->element<<endl;
    preorderTraversal(root->leftChild);
    preorderTraversal(root->rightChild); }
}

void inorderTraversal(Node *root)
{ if (root!=NULL) {
    inorderTraversal(root->leftChild);
    cout<<root->element<<endl;
    inorderTraversal(root->rightChild); }
}

void postorderTraversal(Node *root)
{ if (root!=NULL) {
    postorderTraversal(root->leftChild);
    postorderTraversal(root->rightChild); }
    cout<<root->element<<endl;
}
```

Exercise: Propose an iterative implementation of these 3 traversals.

2.3. Create/Drop a binary tree

Recall that a tree is a recursive data structure; indeed a binary tree is formed from a root whose left and right children are also trees. This allows to build a tree in a very simple way thanks to the following routines:

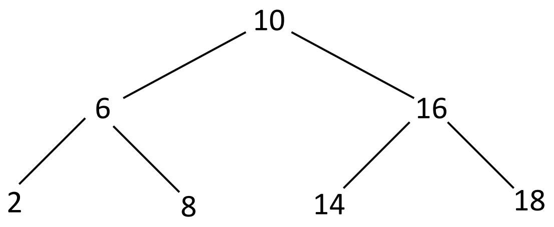
```
Node *newNode(int x) {
      Node *newNd = new Node;
      newNd->element = x;
      newNd->leftChild = newNd->rightChild = NULL;
      return newNd;
}
```

```
Node *join(Node *left, Node *right, int valNode) {
      Node *newNd = newNode(valNode);
      newNd->leftChild = left;
      newNd->rightChild = right;
      return newNd;
}
```

To drop a tree and recover the allocated space; we run the following recursive routine:

```
void drop(Node *p) {
      if (p==NULL) return;
      drop(p->leftChild);
      drop(p->rightChild);
      delete p;
}
```

The following program implements the routines described previously to build the following tree. The memory space is recovered at the end of the program.



```
int main() {
      Node *LST = join(newNode(2),newNode(8),6);
      Node *RST = join(newNode(14),newNode(18),16);
      Node *root = join(LST,RST,10);
      inorderTraversal(root);
      drop(root);
}
```

2.4. Operations on binary trees

Here are implementation details of some operations on binary trees based on the dynamic representation:

- Tests if a node is a leaf:

```
bool isLeaf(Node *p)
{ return (p->leftChild==NULL && p->rightChild==NULL); }
```

- Tree size:

```
int size(Node *root)
{ if (root==NULL) return 0;
  else return 1+size(root->leftChild)+size(root->rightChild);
}
```

- Number of leaves of a binary tree:

```
int nbLeaves(Node *root)
{ if (root==NULL) return 0;
  else if (isLeaf(root)) return 1;
       else return nbLeaves(root->leftChild)+nbLeaves(root->rightChild);
}
```

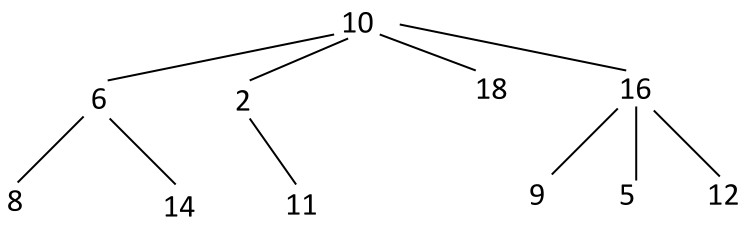
2.5. Representation of general tree as binary tree

Although binary trees are the most widely used, general trees also have important applications. The routines seen previously can easily be generalized for general trees. However, it is possible to convert any tree into a binary tree by following these steps:

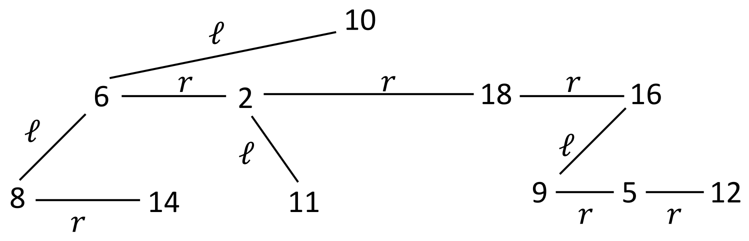
1. Remove all edges from each node of the general tree, except the leftmost edge (edges ℓ for left).
2. Draw edges from a node to the node that is on the right, if it exists, that is located at the same level (sibling nodes; edges r for right).
3. Generate the binary tree from the obtained edges.

Example:

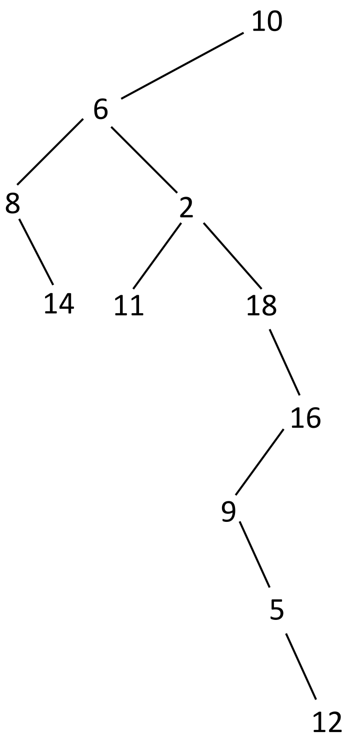
Let convert the following tree into a binary tree:



Steps 1 and 2:



Step 3:



3. Implementations

3.1. N-ary trees

N-ary trees (or general trees) can be implemented using arrays and also using linked lists. Each implementation has its own advantages and disadvantages in terms of its ability and efficiency to implement the classic tree operations described earlier.

Using arrays

Let T a tree with *n* nodes designed by 0,1,…,*n*-1. An efficient data structure that can implement easily the operation ***parent(n,A)*** is an array A that element A[i] denote the parent of node i. Therefore we can implement a tree by an array A defined as follows:

A[i] = j	if the node j is the parent of node i
A[i] = -1	if node i is the root.

It is clear with this representation that the **parent** operation is always of complexity O(1). However, with this representation, operations using information on children are more complicated. Furthermore, we cannot impose an order on the children of a node, in particular the operations **leftmostChild**, **rightmostChild** cannot be defined.

Example: If we make each node of the tree seen in section 1, correspond to an index; we obtain:

Node	Index
J. Lagrange	0
J. Fourier	1
S. Poisson	2
G.Plana	3
G. Dirichlet	4
M. Chasles	5

$\Rightarrow$

0	1	2	3	4	5
-1	0	0	0	2	2

Using array of pointers

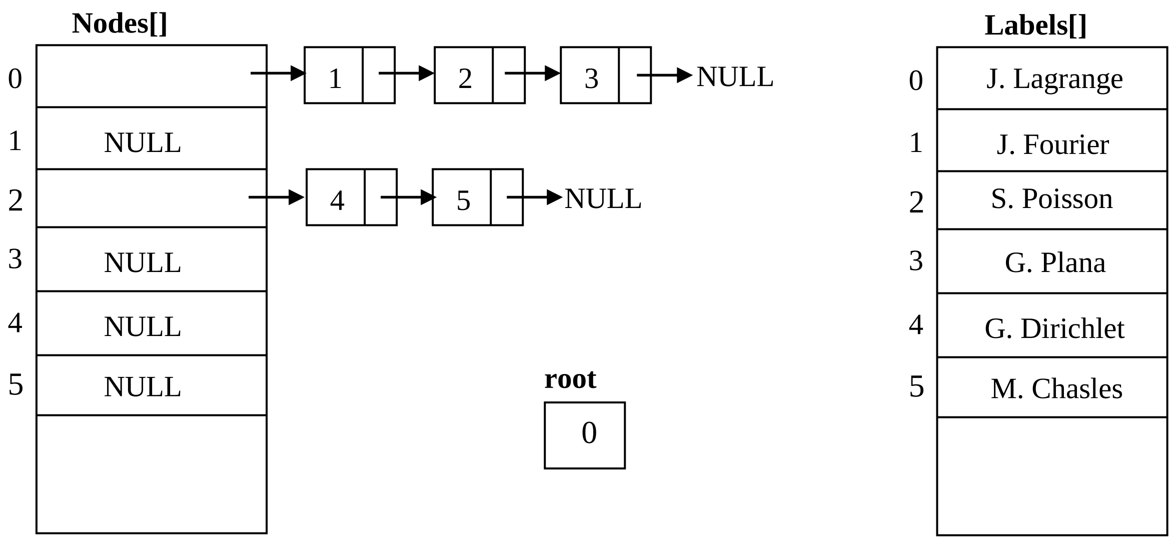
An efficient way to represent a tree is to build a list of children of each node which can be implemented with a linked list, since the number of children varies from node to node:

```
const int MaxNodes=1000;

struct Bloc {
  int element;
  Bloc *next;
};

struct Tree {
  Bloc *Nodes[MaxNodes];
  char Labels[MaxNodes];
  int root;
};
```

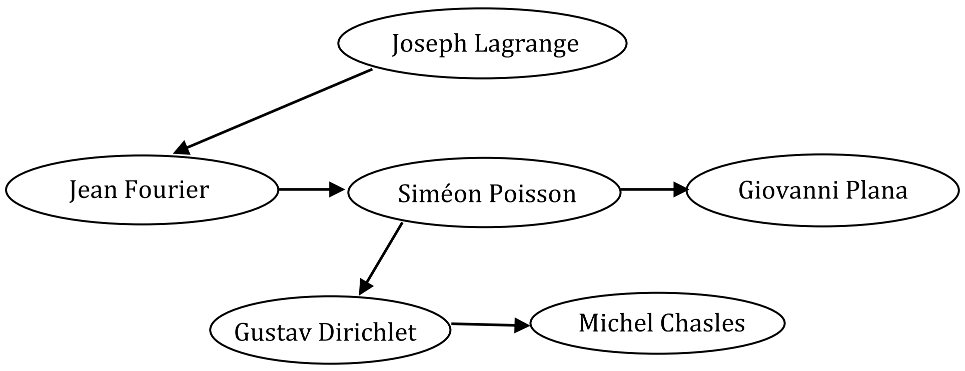
Thus the tree cited in illustration is represented as follows:



Exercise: Propose an implementation of the routines cited in section 1 based on this representation.

Dynamic representation

Since the number of children can vary from one node to another and their number is not known in advance, a dynamic representation associating the data of the node with pointers to all its children would be impractical. The solution is simple: Just keep for each node a link to the leftmost child and a link to the node representing the next "sibling" as follows:



```
struct Node {
 int element;
 Node *leftmostChild;
 Node *nextSibling;
};
```

Here is the implementation of some routines using this representation:

- Display of children

```
void displayChildren(Node *p)
{ Node *current = p->leftmostChild;
  while (current != NULL) {
      cout << current->element << endl;
      current = current->nextSibling; }
}
```

- Postorder traversal

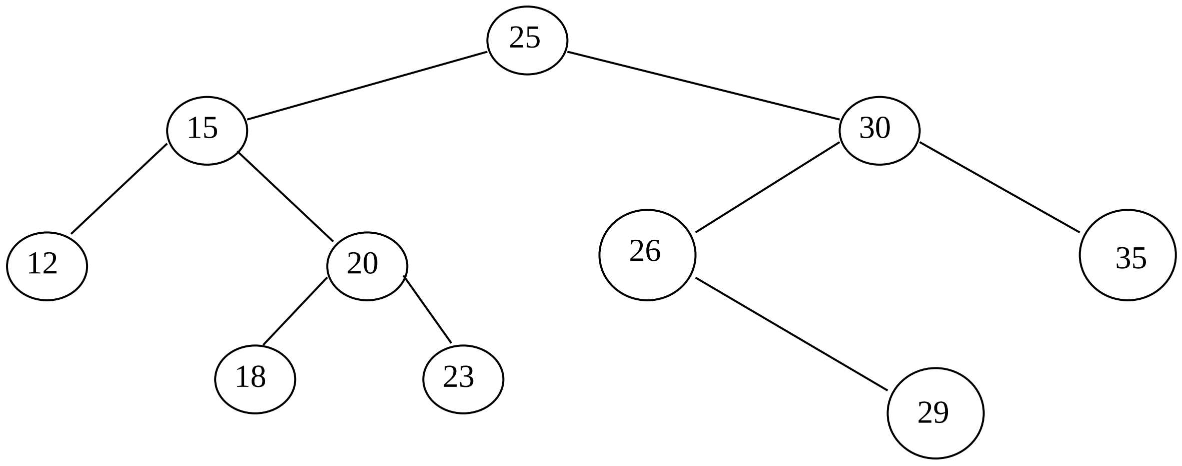
```
void postorder(Node *root)
{ if (root!=NULL) {
      postorder(root->leftmostChild);
      cout<<root->element<<endl;
      postorder(root->nextSibling);
  }
}
```

3.2. Binary Search Tree (BST)

A Binary Search Tree (BST) is a tree where the data in the tree follow a total order relation. Whatever the node, the data present in the left subtree are less than or equal to the data present in this node, which itself is less than or equal to those present in the right subtree.

This way of organizing the data allow to search elements with a O(logN) complexity if the tree is balanced. The advantage of this search method compared to the binary search in a sorted list lies in particular in the insertion operation. Indeed to maintain a sorted list, the insertion is done in the worst case (and also on average) in O(N), whereas in binary search trees, the average complexity of the insertion is O(logN) provided the tree is balanced.

Example:



By its structure, the in-order (infix) traversal of a binary search tree produces a sorted list of the data it contains. The search, insert, delete, minimum or maximum procedures can be easily implemented using recursive or iterative procedures.

- **Search**

```
bool search(Node *root , int val)
{ if (root==null) return false;
  else if (root->element==val) return true;
       else if (root->element>val)
                return search(root->leftChild,val);
            else return search(root->rightChild,val);
}
```

- **Insertion**

```
void insert(Node *&root , int val)
{
  Node *p=root, *prior=NULL;
  while (p!=NULL)
  {
   prior=p;
   if (p->element>val) p=p->leftChild;
   else p=p->rightChild;
  }
  Node *newNd=new Node;
  newNd->element=val;
  newNd->leftChild=newNd->rightChild=NULL;
  if (prior==NULL) root=newNd;
  else if (prior->element>val) prior->leftChild=newNd;
       else prior->rightChild=newNd;
}
```

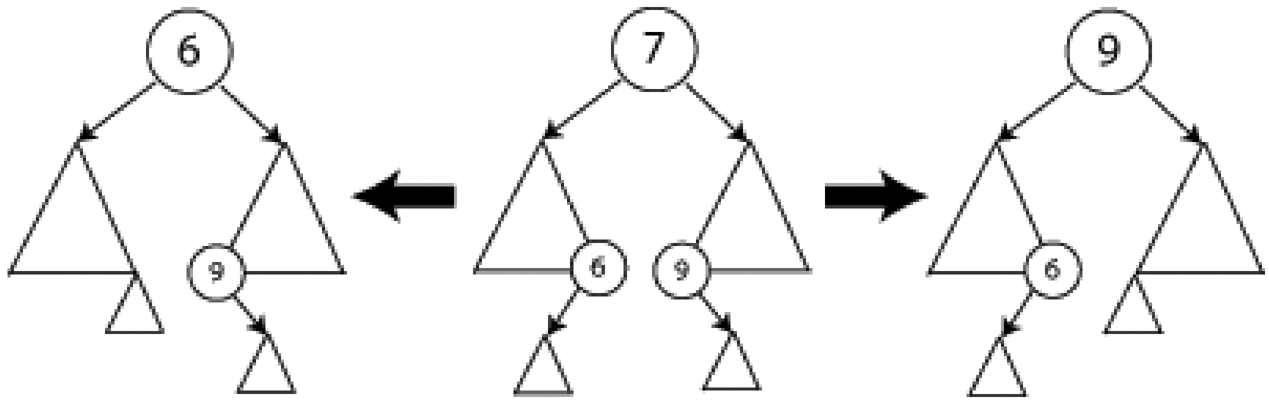
It is easy to see that the complexity of the insertion and search operations in a binary search tree (of size n) is equal, in the worst case, to the height of the tree, i.e.:

- $O(\log_2(n))$ if the tree is balanced.
- $O(n)$ if the tree is degenerated.

- **Deletion**

Three cases must be considered, once the node to delete is found:

- *Deletion of a leaf*: Just remove it from the tree since it has no child.
- *Deletion of a node with one child*: remove it by replacing it with his unique child.
- *Deletion of a node with two children*: Suppose the node to be deleted is called D (the node with value 7 in the graph below). We swap node D with its closest successor (the leftmost node of the right subtree - below, the node of value 9) or its closest predecessor (the rightmost node of the subtree left - below, the value node 6). This keeps a binary search tree structure. Then we apply the deletion procedure again to D, which is now a leaf or a node with only one child.

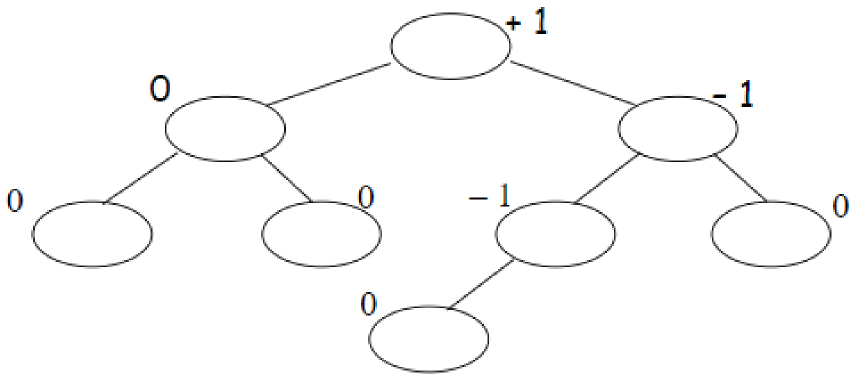


3.3. Balanced binary search tree: AVL Tree

AVL trees are self-balanced binary search trees; the heights of the subtrees of the same node differ at most by one. The time complexity of search and update procedures are $O(logn)$ in the worst case. Nonetheless, insert and delete procedure occasionally requires performing rebalancing operations called rotations.
The name "tree AVL" comes from the respective names of its two inventors, respectively Georgii Adelson-Velsky and Evgueni Landis.

Balance factor f

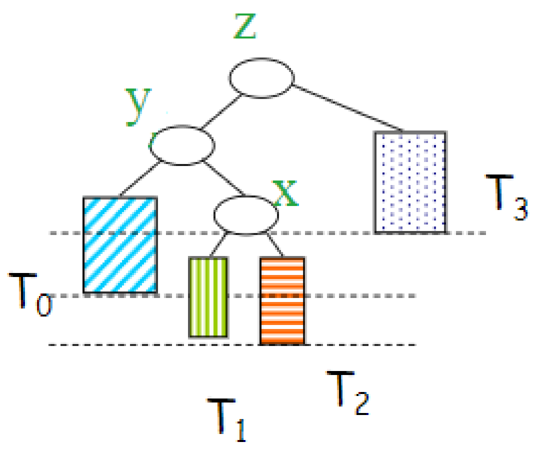
Balance factor f of a node is the difference between the height of its right subtree and that of its left subtree. A node whose balance factor is 1, 0 or -1 is considered balanced, otherwise it is considered unbalanced and requires balancing. This factor is either calculated from the respective heights of the subtrees or stored in each node of the tree (see figure below).

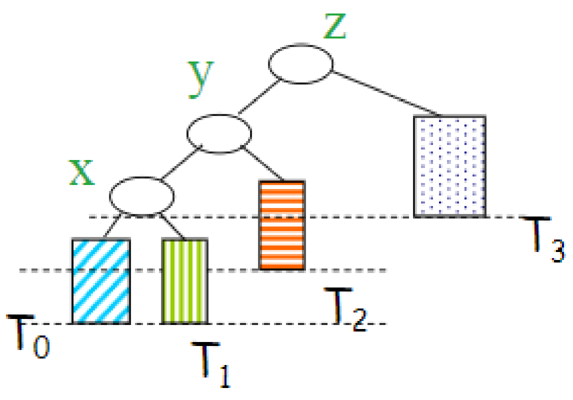


Insertion in an AVL

Inserting a node into an AVL tree is a two-step process: first, inserting the node in exactly the same way as in a binary search tree, then if the tree becomes unbalanced, perform a single (right or left) or double (left-right or right-left) rotation to balance the tree.

For this, we must identify three nodes that form a triplet (grandparent, parent and child) and four subtrees attached to these nodes. Let w be the node which has just been inserted and which caused an imbalance in the tree. We traverse the tree towards the root to identify the three nodes x, y and z ancestors of w. These three nodes can have four subtrees connected to them: T0, T1, T2 and T3 as in the examples of the figure below:

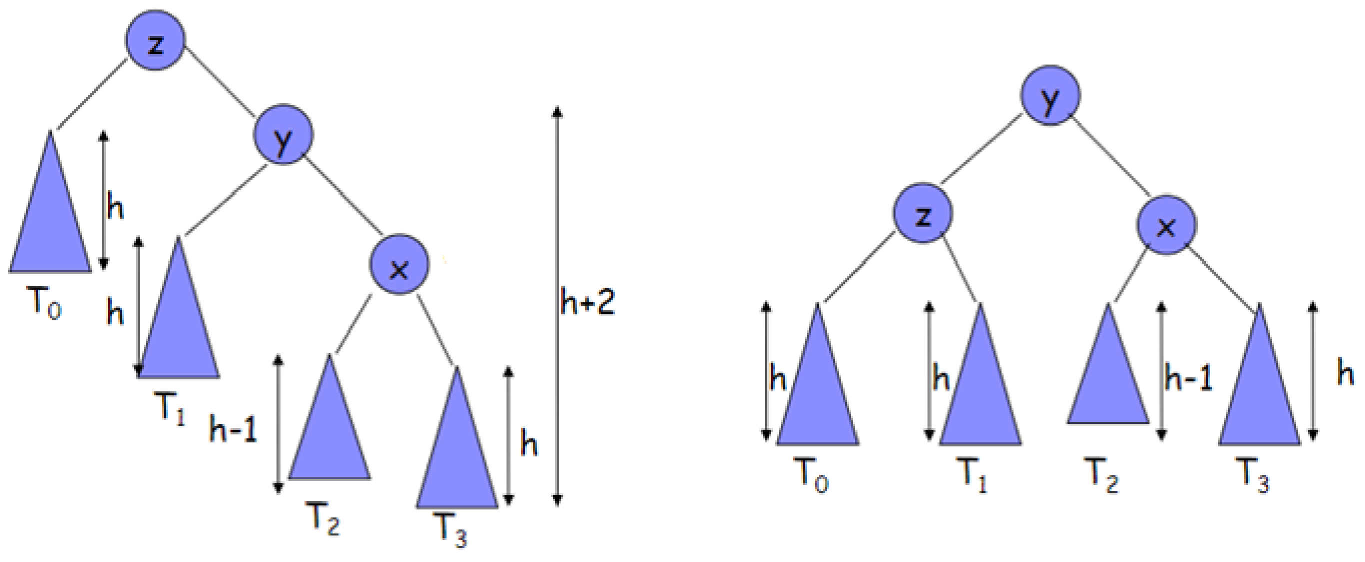




Then the rebalancing is done by considering the three nodes in the infix order according to the following four operations:

Simple left rotation

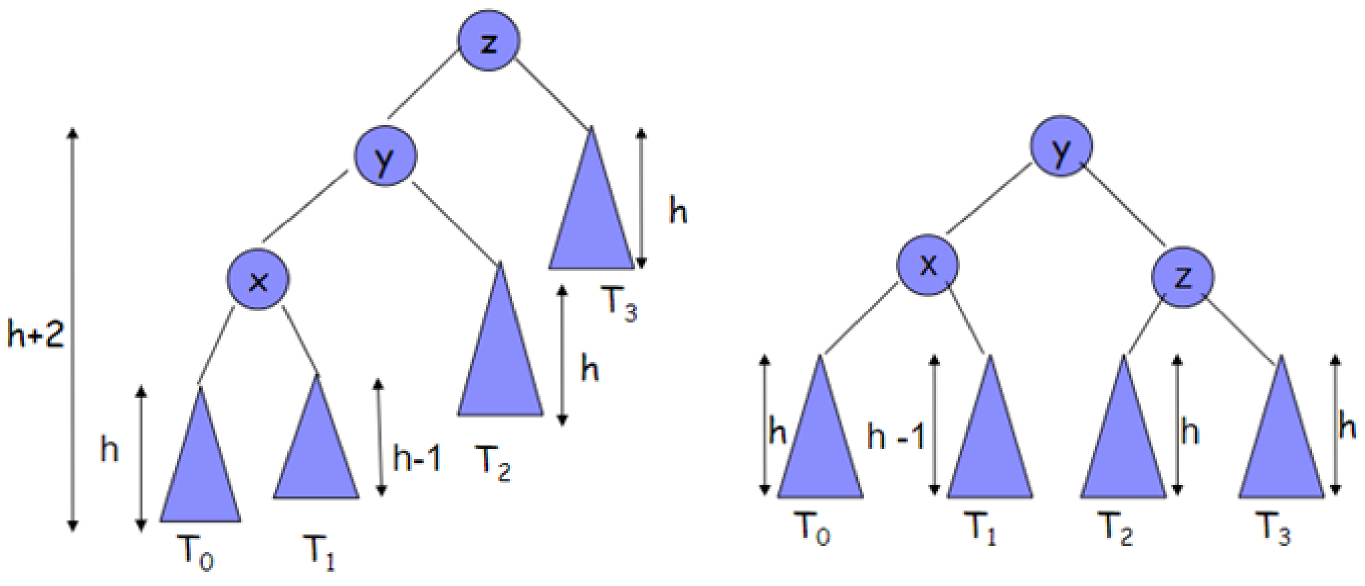
The tree with root z leans right and its right subtree with root y leans right; f(z)=2 and f(y)=1



Inorder Traversal : T0 z T1 y T2 x T3

Simple right rotation

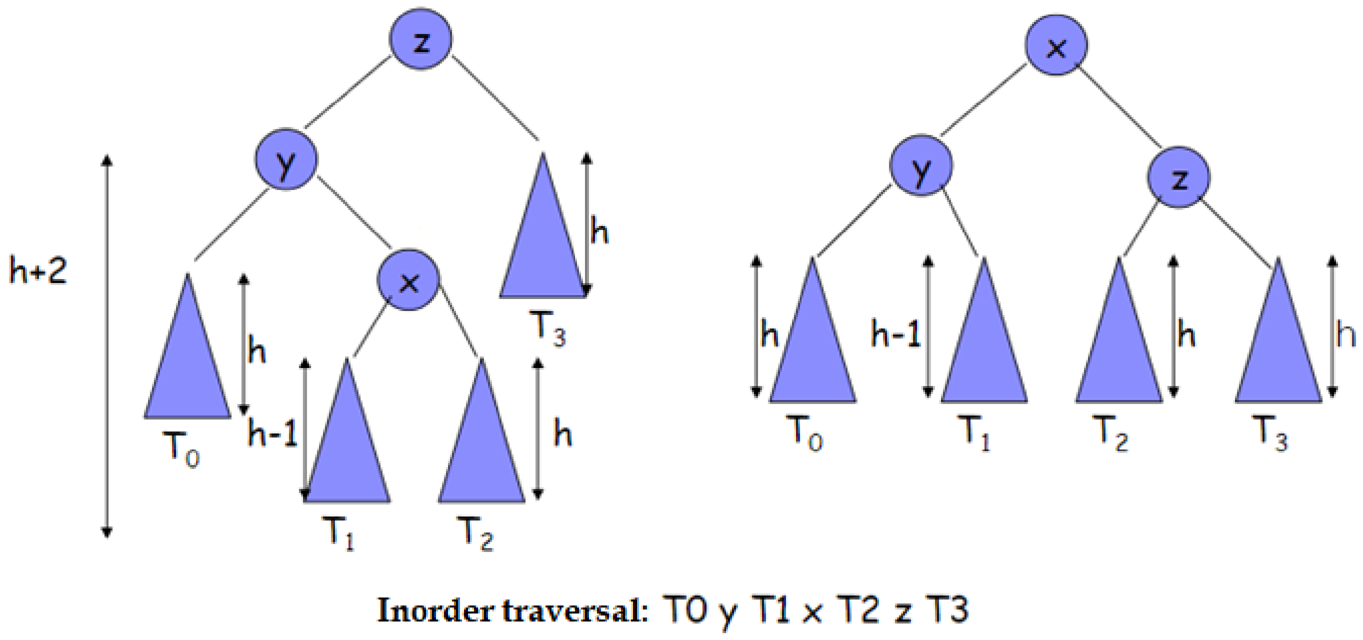
The tree with root z leans left and its left subtree with root y leans left; f(z)= –2 and f(y)= –1.



Inorder Traversal : T0 x T1 y T2 z T3

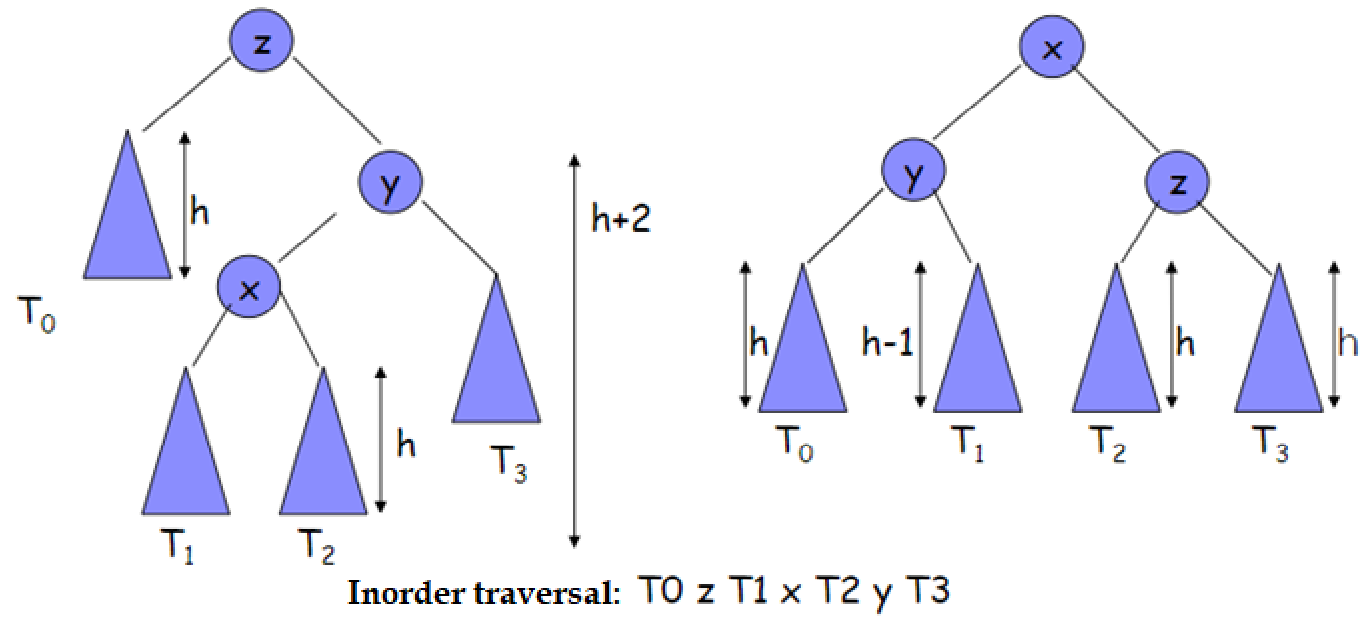
Double left-right rotation

The tree with root z leans left and its left subtree with root y leans right; f(z)= –2 and f(y)= 1 => left rotation of y followed by a right rotation of z.



Double right-left rotation

The tree with root z leans right and its right subtree with root y leans left; f(z)= 2 and f(y)= –1 => right rotation of y followed by a left rotation of z.



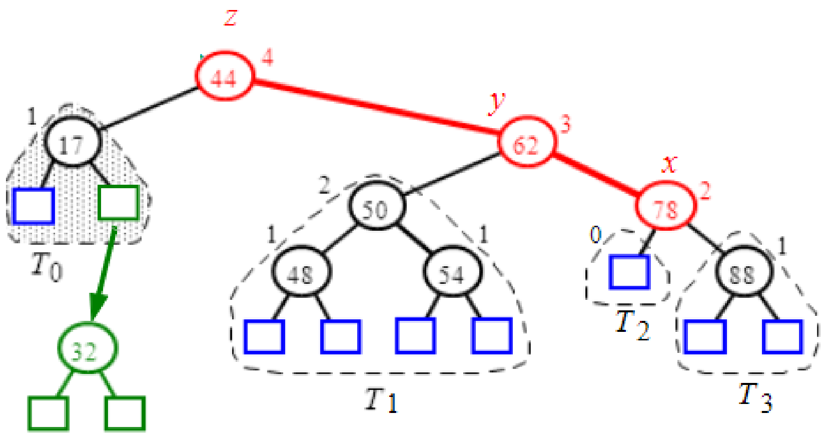
Complexity

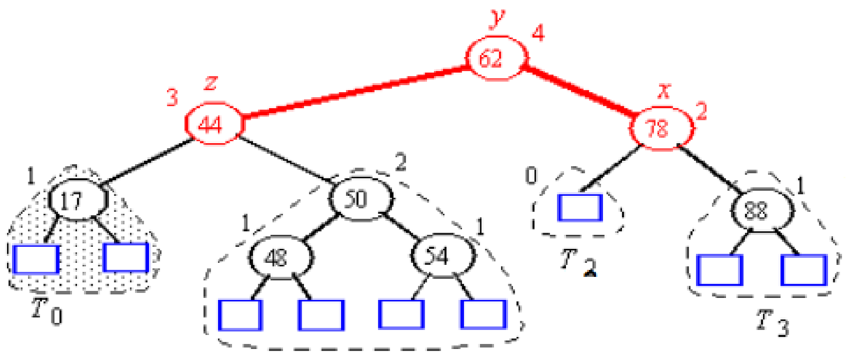
The tree being balanced, all rotation operations have a $O(logN)$ complexity. The insertion routine therefore remains of logarithmic complexity.

Deletion in an AVL

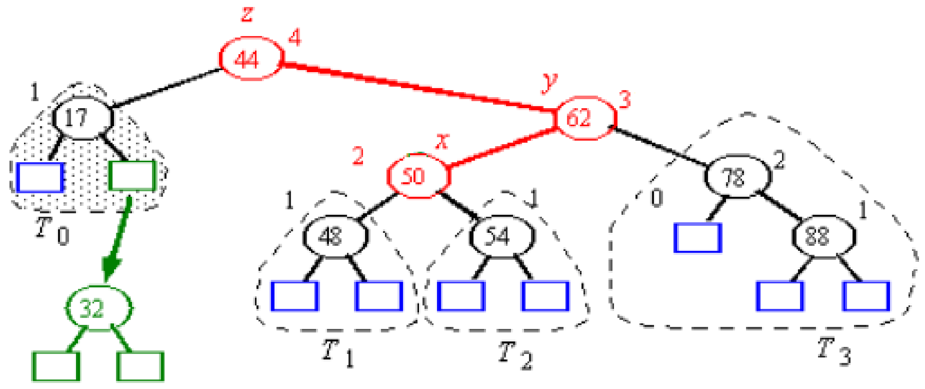
The same tree rebalancing strategy can be applied after deleting an item. Let w be the element to be deleted or the element that replaced the element to be deleted. Let z be the first unbalanced node encountered while traversing the tree to the root from w. Let y be the child of z with the greatest height and x the child of y with the greatest height (x and y are not ancestors of w). The restructuring reduces to 1 the height of the subtree initially rooted at z and thus could unbalance another node higher in the tree. Thus, one must continue to check the balance until the root is reached.

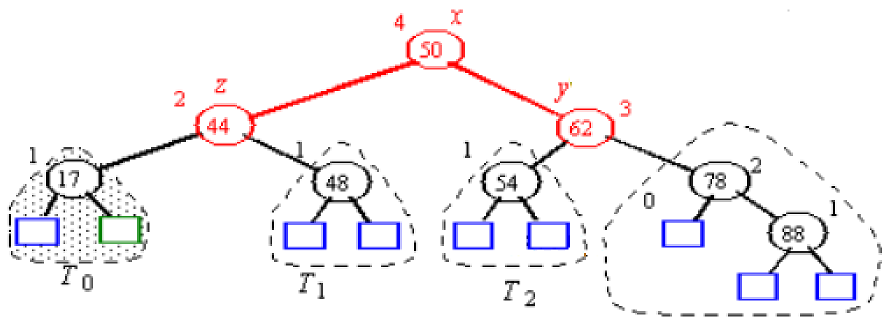
The following example shows the choice of nodes x and y and z and the rebalancing done after removing element 32.





We can choose another x (50 instead of 78 for example)

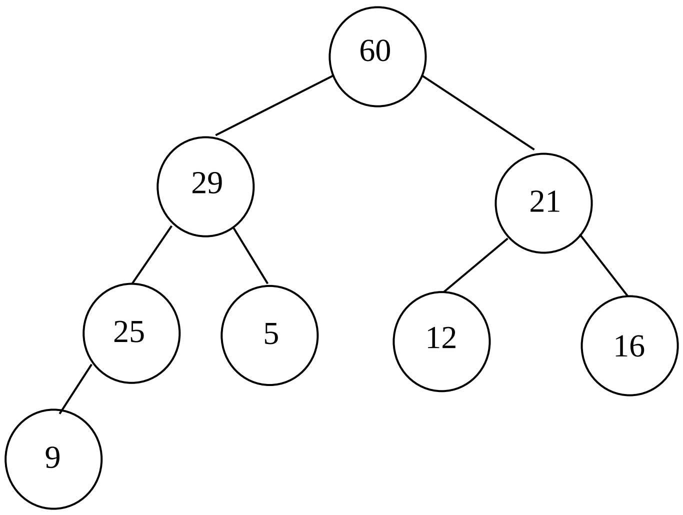




4. Heap data structure

4.1. Definition of a heap

A heap is represented as a perfect binary tree (filled from left to right). The data in the nodes must belong to a totally ordered set and must be greater than those present at the level of its children. The structure thus obtained is called max-heap as shown in the following figure. Note the difference with a binary search tree.



The heap can also be ordered in an increasing way, the structure is then called min-heap.

4.2. Implementation

For each heap structure corresponds a particular linear structure (array, list) obtained by numbering the nodes level by level and from left to right. The root occupies the first element of the structure.

It easy to verify that nodes of level i are stored in the interval $[2^i, 2^{i+1}[$. The parent of a node stored in position j, (except the root), is in the position $\lfloor j/2 \rfloor$, i.e. the integer part of j divided by 2.

The following table corresponds to the representation of the heap structure described previously:

1	2	3	4	5	6	7	8
60	29	21	25	5	12	16	9

The declaration of a heap can therefore be:

```
const int MaxSize=9999;

struct Heap
{
 int Data[MaxSize];    //Index 0 not used
 int size;
};
```

4.3. Operations on heaps

Insertion in a heap

The insertion in a heap must respect the perfect tree structure. For this, the insertion is done in the last level if it is not full. Otherwise, a new level must be created and the element to be inserted must occupy the first place in this new level. In both cases the insertion is always done at the end of the linear structure. On the other hand, to respect the fact that the data present at the level of the parent must be greater than those present in the child nodes (case of max-heap), the new element inserted must be exchanged with its parent if this principle is not respected as many times as it is necessary. The insert procedure can be implemented as follows:

```
void insertHeap(Heap &H, int key)
{ int i=H.size+1;
  while (i>=2 && key>H.Data[i/2])
  { H.Data[i]=H.Data[i/2]; i=i/2;}
  H.Data[i]=key;
  H.size++;
}
```

Complexity

The insertion of a key at position k is done first at the end of the array. This correspond to the level $\log_2(k)$ of the tree representing the heap. Therefore, in the worst case and during the insertion process, the data can go up to the top of the tree. Therefore, the complexity of the insertion is $O(\log_2(k))$ and this of course in the worst case.

Deletion in a heap

Deleting an element from a heap is done first by substituting the element to be deleted with the last element of the heap. Once the element has been deleted, the last element in the heap, which is therefore in the place of the deleted element, is compared with its children. Thus this element is possibly moved downwards until the property of the heap is respected.

Exercise: Propose an implementation of deletion in a heap and measure its complexity.

4.4 Heap Sort

Due to its structure, the top of a heap always contains the maximum value (max-heap) or the minimum value (min-heap). This important remark can be used for the development of one of the most efficient algorithms for sorting which has a complexity in the worst case, and also on average, equal to $O(n\log_2(n))$.

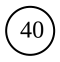
In the case of a max-heap, the principle of sorting the heap consists first of all in exchanging the data contained in the root with that contained in the last leaf of the tree which corresponds to the last cell of the array representing the heap. Consequently, after this exchange operation, the data contained in the last cell of the table is the largest and remains unchanged in the case of sorting in ascending order of the data of a heap. To complete the sorting, the algorithm must reconstitute the data whose indices are between 1 and n-1 in a sub-heap to be able to reuse the previous remark. To respect the structure of a heap, the new data that has just been inserted into the root must eventually be moved down to be larger than the data contained in its children. This operation must be repeated as many times as necessary until a leaf is reached and the data that descends is smaller than those contained in its two children.

Example: Heapsort (case of min-heap) applied to the following list:

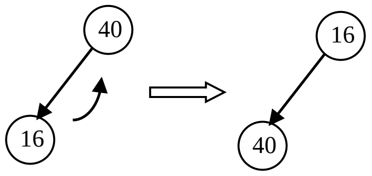
40	16	20	23	28

a) Heap building (Insertion)

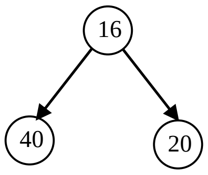
- Insertion of 40



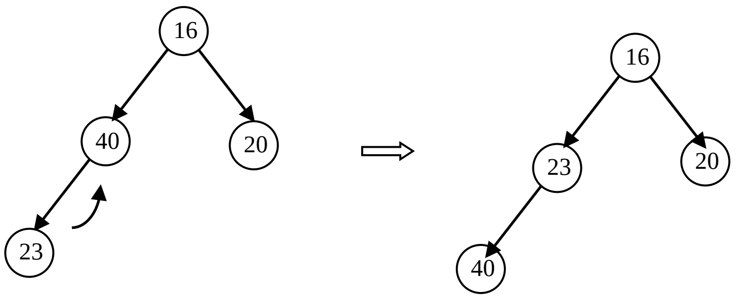
- Insertion of 16



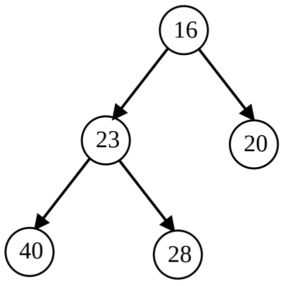
- Insertion of 20



- Insertion of 23



- Insertion of 28



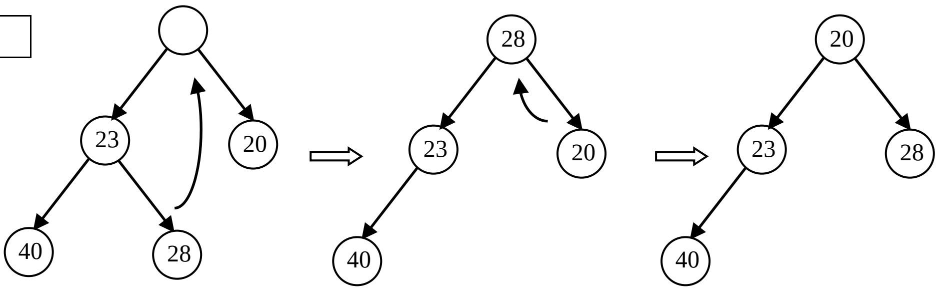
The Heap obtained is:

16	23	20	40	28

b) Heapsort (root deletion)

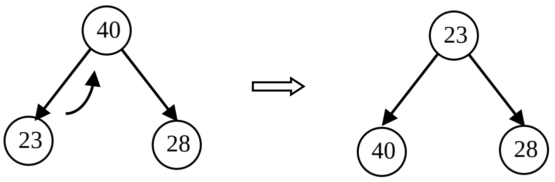
- Deletion of the root 16

16				



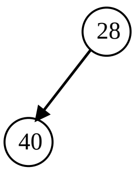
- Deletion of 20

16	20			



- Deletion of 23

16	20	23		



- Deletion of 28

16	20	23	28	

- Deletion of 40 (Heap empty)

16	20	23	28	40

Implementation details of heapsort are given below.

```
void moveDown(Heap &A, int first, int last)
{
int r,temp;
r=first;
while (r<=last/2)
    if (last==2*r)
     {
      if (A.Data[r]>A.Data[2*r])
          {
            temp=A.Data[r];
            A.Data[r]=A.Data[2*r];
            A.Data[2*r]=temp;
          }
      r=last;
     }
    else
```

```
     if (A.Data[r]>A.Data[2*r] && A.Data[2*r]<=A.Data[2*r+1])
      {
       temp=A.Data[r];
       A.Data[r]=A.Data[2*r];
       A.Data[2*r]=temp;
       r=2*r;
      }
     else
      if (A.Data[r]>A.Data[2*r+1] && A.Data[2*r+1]<=A.Data[2*r])
       {
        temp=A.Data[r];
        A.Data[r]=A.Data[2*r+1];
        A.Data[2*r+1]=temp;
        r=2*r+1;
       }
      else r=last;
}

void heapSort(Heap &A)
{
int i,temp;
for (i=A.size/2;i>=1;i--) moveDown(A,i,A.size);
for (i=A.size;i>=2;i--)
{ temp=A.Data[1];
  A.Data[1]=A.Data[i];
  A.Data[i]=temp;
  moveDown(A,1,i-1);
}
}
```

The heapsort complexity can be evaluated as follows: each move operation takes in the worst case $\log_2(p)$ where p is the number of nodes concerned by this operation. Therefore, the total complexity is O($\sum_{p=1}^{n}\log_2(p)$) = O(n$\log_2$(n)).