# ALGORITHMIQUE - STRUCTURE DE DONNEES

PLAN

AVERTISSEMENT : ce cours est un cours d'algorithmique et structures de données et non pas un cours sur le langage Python même si les algorithmes sont donnés dans ce langage.

# 1 - NOTION D'ALGORITHME

Un algorithme est la manière de résoudre un problème. Le problème doit être bien défini (cahier des charges, …) et l'algorithme doit aussi être bien défini et suffisamment formel.

Pour détailler un algorithme, on peut employer un pseudo code ou l'écrire directement dans un langage de programmation.

# 2 - LANGAGES DE PROGRAMMATION

Premiers langages : FORTRAN, COBOL et LISP

## 2.1 - Différents types de langages :

- langages de bas niveau et langage d'assemblage
- langages procéduraux : Algol*, Pascal, C, Ada, Fortran, Cobol, …
- langages orientés objets : JAVA, C++, Python, …
- langages fonctionnels ou applicatifs : LISP, CAML, …
- langages divers : SQL (BD), Prolog (logique), langages de bibliothèques de fonctions (R, SAS, MATHLAB, MATHEMATICA, …)

## 2.2 - Langages interprétés vs langages compilés

Un langage interprété prend les instructions une à une et les exécute.

Un langage compilé sera traduit (par un programme appelé un compilateur) en langage machine une fois pour toute.

Un langage compilé est plus rapide qu'un langage interprété

Langages interprétés : Python, JAVA, Lisp, Prolog, …
Langages compilés : C/C++, Fortran, Cobol, …

Le langage C sert de référence et les temps d'exécution sont exprimés par un rapport par rapport à C. Parmi quelques langages on peut estimer :

- Fortran : en moyenne un peu meilleur que C
- JavaScript : entre 2 à 3 fois plus lents que C
- Python : langage environ 15 fois plus lent que C
- langages les moins performants : langages associés aux bibliothèques mathématiques mais les fonctions de ces bibliothèques sont elles très efficaces (souvent écrite en C/C++ ou Fortran)

## 2.3 - Quelques langages utilisés dans cette formation :

Python (pour ce cours), C, JAVA, Prolog, SQL, C++, OCAML, langage de la bibliothèque R, Perl, ...

## 3 - LES TYPES :

Toutes les variables et les constantes utilisées possède un type bien défini. Beaucoup de langages imposent la déclaration des variables utilisées mais pas Python du fait de son typage dynamique.

(Py) indique que ce type existe en Python.

### 3.1 - Les types simples :

- le type booléen (Py) : vrai / faux
- le type entier (Py) : exemples : 1, -10, +12, 0
- le type réel (Py) : exemples : 3.14, -10.5, 1e+3, -2e-4
- le type caractère (Py) : exemples : 'A' , 'a', ' ', '='

### 3.2 - Les types composés ou complexes :

- le type chaîne de caractères (Py)
- le type tableau (Py commun avec les listes)
- le type liste (Py)
- les structures (Py : les classes)
- les tuples (Py)
- les dictionnaires (Py)
- le type nombre complexe (Py)
- le(s) type(s) fichier (Py : accès séquentiel uniquement)
- ...

### 3.3 - Les listes python :

Les listes font partie des « conteneurs » python avec les tuples et les dictionnaires.

Les listes python peuvent être utilisées comme des listes avec des fonctions et des méthodes spécialisées et comme des tableaux classiques. Dans ce cas les indices commencent à 0 (indice du premier élément).

Une liste constante est une liste de valeurs entre crochets :
l = [1, 2, 'a', True, [3.15], None,'Fin']. Ici :
l[1] = 2, l[4] = [3.15], l[1:] = [2, 'a', True, [3.15], None,'Fin']
l[2:5] =  ['a', True, [3.15]]

Opérations sur les listes :

- concaténation : [1,2]+[3,4] = [1,2,3,4]
- ajout d'un élément : l.append(5)
- taille d'une liste : len(l)
- ...

Dans ce cours, on va souvent les utiliser comme des tableaux, On les utilisera également pour représenter les arbres.

## 4 - LES EXPRESSIONS :

## 4.1 - Les expressions logiques :

- Valeurs de vérité (constantes) (Py) : False, True
- Négation (Py) : not
- Le ou et le et logique (Py) : or, and
- Autres opérateurs logiques inutiles dans ce cours,
- La comparaison des deux valeurs de même type produit un résultat logique. Opérateurs de comparaison : ==,!=, <, <=, >, >=

## 4.2 - Les expressions numériques :

- Les constantes numériques et les variables de type entier ou réel sont des expressions numériques
- Si x et y représentent des expressions numériques, alors : (x), -x, +x, x+y, x-y, x*y, x/y, x//y, x%y, x**y sont aussi des expressions numériques

## 4.3 - Priorité des opérateurs :

Par priorité décroissante :

- les parenthèses,
- les opérateurs unaires : not, + et - unaires ,
- l'opérateur d'exponentiation : **
- les opérateurs multiplicatifs : /, //, *, and, ...
- les opérateurs additifs : -, +, or, …

En cas d'égalité des priorités, l'évaluation se fait de gauche à

droite.

## 5 - PRINCIPALES INSTRUCTIONS (python) :

### 5.0 - Commentaires :

Les commentaires ne sont pas des instructions mais permettent d'expliquer le code. Un commentaire commence par le caractère dièse (#) et se termine à la fin de la ligne.

### 5.1 - L'affectation :

Une affectation permet de donner (ou modifier) la valeur d'une variable . La syntaxe est : <var> = <exp> où <var> est le nom de la variable est <exp> l'expression que l'on souhaite donner à la variable. Ceci est la forme simple d'une affectation.

Exemples :

- toto = 4
- titi = 'a'
- machaine = "ceci est une chaîne"
- x = 3.14

### 5.2 - Affichage d'une expression :

Avec la forme simple de l'instruction print de python :

- print("toto = ",toto)
- print("Coucou")
- print(toto**2)

5.3 - Saisie d'une valeur :

Syntaxe : <var> = input(<chaîne>)

Exemples :
- nom = input('Donner le nom : ')
- n = input('Nombre d'éléments : ')
  n = int(n)
- x = input('valeur de x : ')
  x = float(x)

5.4 - Instruction conditionnelle :

Syntaxe :
if <expression logique 1> :
    <instruction>+
elif <expression logique 2> :
    <instruction>+
…….....
else :
    <instruction>+

Remarques :
- l'indentation est très importante en python, c'est elle qui définit la fin des parties 'elif', 'else' et le corps des

boucles pour les instructions itératives. Attention aux tabulations : ce ne sont pas des espaces
- les parties 'elif' et 'else' sont facultatives
- les expressions doivent être de type booléen

## 5.5 - Instructions itératives :

### Instruction « tantque » :

Syntaxe :
while <expression logique> :
    <instruction>+

Exemple :
```
while x > 0 :
    print(x)
    x = x-y
```

Remarques : identiques à celles de l'instruction conditionnelle

### Instruction « for » : parcours de conteneurs :

Parcours d'une liste :
```
for element in l :
        <instruction>+
```

Parcours d'un intervalle d'entier :
```
for i in range(1:10) :
```

<instruction>+

## 5.6 - Fichiers « texte » :

Ouverture :
En écriture : f = open("nom_fichier", "w")
En lecture : f = open("nom_fichier", "r")

Fermeture : f.close()

Écriture : f.write(<expression>)
C'est la forme la plus simple d'écriture dans un fichier, mais elle est suffisante pour ce cours.

Lecture :
Comme pour les lectures avec "input", on lit une chaîne de caractères que l'on doit ensuite convertir en entier ou réel :

- ch = f.readline() : lit une ligne dans la chaîne ch
- ch = f.read() : lit tout le fichier dans ch
- lch = f.readlines() : lit tout le fichier dans la liste de chaînes lch

## 5.7 - Les exceptions :

Elles permettent de récupérer des erreurs d'exécution. Une exception est déclenchée par un événement anormal : division par 0, conversion impossible, … Quand elle se produit elle provoque une erreur d'exécution si elle n'est pas

prévue dans le programme sinon le code correspondant est exécuté.

Forme générale :

```
try :
   <instruction risquée>+
except <nom exception> :
    <instruction>+
except <nom exception> :
    <instruction>+
```

Exemple pour la saisie d'un entier :

```
def saisie(message) :
  ok = False
  while (not ok) :
    try :
      n = input(message)
      n = int(n) # instruction risquée
      ok = True
    except ValueError :
    print("pas un entier, recommencez")
  return n
```

## 5.8 - Les fonctions :

Une fonction est une partie de programme indépendante réalisant un travail précis résolvant un sous-problème. On

distingue les véritables fonctions qui renvoient une valeur des procédures qui ne renvoient pas de valeur. Les fonctions, au sens large, peuvent avoir des paramètres.

Exemple : valeur absolue :

```
def  vabs(x) :
        if x < 0 :
                return -x
        else :
                return x
```

## 6  - LES SÉQUENCES LINÉAIRES :

Dans toute cette partie, on utilisera les listes python comme des tableaux.

### 6.1 - Opérations sur les séquences :

- création
- recherche / modification
- ajout
- suppression
- interclassement
- intersection
- différence
- extraction
- tri (interne)

### 6.2 - Recherche d'un élément dans une séquence :

On recherche un élément x dans une séquence seq représentée par un tableau (liste).

Recherche séquentielle :

```
def rech_seq (seq, x) :
# cette fonction renvoie l'indice de l'élément si il est présent
# et renvoie -1 sinon
        n = len(seq)
        i = 0
        while (i < n) and (seq[i]!= x) :
                i = i+1
        if (i >= n) :
                return -1      # élément absent
        else :
                return i
# fin de la fonction
```

Remarque : il existe une variante où l'on commence par ajouter l'élément recherché à la fin, cela permet d'économiser le test i < n car on est certain de trouver l'élément. A faire en exercice

Recherche dichotomique :
On suppose le tableau trié (en ordre croissant)

Les paramètres sont les mêmes, même chose pour la valeur

renvoyée.

```
def rech_dicho (seq,x) :
        n = len(seq)
        bi = 0    # borne inférieure de l'intervalle de recherche
        bs = n-1    # borne supérieure
        while bi <= bs :  #tq intervalle non vide
                mi = (bi+bs) // 2  # indice du milieu
                if seq[mi] >  x :
                        bs = mi-1
                elif seq|mi] < x :
                        bi = mi+1
                else :    # élément trouvé
                        return mi
         # fin de la boucle
         return -1
# fin de la fonction
```

Comparaison des complexités :

Les complexités permettent d'évaluer les performances des algorithmes en place mémoire et en temps (cours spécifique en INFO4). Ici les complexités en place sont les mêmes. Pour étudier la complexité en temps, on sélectionne une opération caractéristique du problème puis on évalue le nombre de fois où elle est exécutée. Souvent, on ne pourra la dénombrer précisément et on se limitera à un ordre de grandeur asymptotique : O(f(n)).
On distingue la complexité au mieux (peu intéressante), la

complexité au pire et la complexité moyenne.

Dans ces fonctions de recherche d'un élément, une opération caractéristique est la comparaison entre deux éléments.

On peut voir que la complexité moyenne (lorsque l'élément est toujours présent) est de n/2 pour la recherche séquentielle et de $1{,}5\log_2(n)$ pour la recherche dichotomique. Prendre des exemples avec n = 8, 16, 100, 1000, ...

Recherches auto adaptatives :

L'idée consiste à mettre les éléments les plus souvent recherchés en début de tableau.

## 6.2 - Ajout d'un élément dans une séquence triée :

Dans cet algorithme, on ne vérifie pas si la séquence est triée.

```
# ajout d'un élément dans un tableau (liste) trié
# premier paramètre : le tableau d'origine
# qui contiendra le résultat à la fin
# deuxième paramètre : l'élément à ajouter
def ajoutrie(t, x) :
    ind = len(t)-1    # indice du dernier élément
```

```
    if x > t[ind] :
        t.append(x)
    else :
        t.append(t[ind])
        ind = ind-1
        while (ind >= 0) and (t[ind] > x) :
            t[ind+1] = t[ind]
            ind = ind-1
        # fin du while
        t[ind+1] = x
    # fin de la conditionnelle
    return t
# fin de la fonction
```

Remarques :

- cet algorithme permet de créer une séquence triée par ajouts successifs à partir de la liste vide.

- On peut gagner un peu de temps en faisant une recherche dichotomique pour savoir où insérer le nouvel élément. Ceci est peu intéressant dans les langages autres que python car on sera obligé de faire séquentiellement les déplacements.

## 6.3 - Interclassement de deux fichiers triés :

```
# Interclassement de 2 fichiers
def interclasser() :
    f1  = open("toto.txt","r")
```

```
    f2 = open("titi.txt","r")
    f12 = open("tototiti.txt","w")
    article1 = f1.readline()
    article2 = f2.readline()
    # boucle principale
    while (article1 != "") and (article2 != "") :
        if article1 < article2 :
            f12.write(article1)
            article1 = f1.readline()
        elif article2 < article1 :
            f12.write(article2)
            article2 = f2.readline()
        else : # ils sont égaux
            f12.write(article1)
            article1 = f1.readline()
            article2 = f2.readline()

    #déversement fichier non vide
    while (article1 != "") :
        f12.write(article1)
        article1 = f1.readline()
    while (article2 != "") :
        f12.write(article2)
        article2 = f2.readline()
    # fermeture des fichiers
    f1.close()
    f2.close()
    f12.close()
# fin de la fonction
```

Sur le même schéma, pour des fichiers triés :

- intersection
- différence
- différence symétrique

## 6.5 - Les tris internes par comparaisons :

On s'intéresse uniquement aux tris internes par comparaisons. Les algorithmes qui suivent opèrent un tri en ordre croissant. Il existe de très nombreuses méthodes de tri plus ou moins performantes. Il n'existe pas une méthode de tri toujours meilleure (plus rapide) que les autres : cela dépend de la taille des données et d'informations complémentaires (données presque triées, …). La complexité en temps minimale théorique pour un tri par comparaison est O(n.log(n)).

### 6.5.1 - Tri par sélection du minimum :

Le principe est simple : on recherche le minimum, on le place au début et on recommence sur la fin de la séquence. Tri simple mais peu efficace.

```
# Tri par sélection du minimum
# Paramètre t : tableau (liste) à trier
# Valeur renvoyée : tableau trié
def trisel(t) :
    n = len(t)              # taille de la liste
    for i in range(n-1) :    #  recherche du ième plus petit
                             #  élément
        j = i
        for k in range(i+1,n) :
            if t[k] < t[j] :
```

```
            j = k
        # end if
    # end for
    t[i], t[j] = t[j], t[i]    # échange des deux valeurs
# end for
return t
# Fin de la fonction trisel
```

## 6.5.2 - Tri par insertion :

```
# Fonction de tri par insertion séquentielle
# Paramètre t : tableau (liste) à trier
# Valeur renvoyée : tableau trié
def triinsertion(t) :
    n = len(t)              # taille de la liste
    i = 1
    while i < n :
        k = i-1
        x = t[i]  # élément à insérer dans la partie triée :
                  #  indices entre 0 et i-1
        while (k >= 0) and (t[k] > x) :
            t[k+1] = t[k]
            k = k-1
        # end while
        t[k+1] = x
        i = i+1
    # end while
    return t
 # Fin de la fonction triinsert
```

L’un des meilleurs tris … quand on a peu d'éléments (15).

## 6.5.3 - Tri à bulles :

Un échange est fait chaque fois que deux éléments consécutifs ne sont pas dans le bon ordre. Les éléments les plus petits remontent très vite vers le début de la séquence, d'où le nom de ce tri. La complexité de ce tri dépend du nombre d'inversions dans le tableau initial mais en général, il n'est pas très rapide.

```
# Tri à bulles (un peu optimisé)
# Paramètre t : tableau (liste) à trier
# Valeur renvoyée : tableau trié
def tribulles(t) :
    n = len(t)              # taille de la liste
    i = 0
    permut = True        # indicateur de permutation dans la
                         #  boucle interne
    while (i < n-1) and (permut) :
        j = n-1             # indice du dernier élément
        permut = False
        while j > i :
            if t[j] < t[j-1] :    # permutation de 2 éléments
                                  # consécutifs dans le mauvais ordre
                t[j-1], t[j] = t[j], t[j-1]
                permut = True
            # end if
```

```
            j = j-1
        # end while
        i = i+1
    # end while
    return t
# Fin de la fonction tribulles
```

6.5.4 - Tris de Shell / Shell-Metzner :

C'est une forme optimisée du tri par insertion mais ici les éléments d'une sous-liste ne sont pas consécutifs distants d'un pas. La complexité dépend de la suite des pas : elle va de $O(n^2)$ à $O(n.\log^2 n)$ et peut-être à $O(n.\sqrt{n})$

```
# Tri de Shell
# Paramètre : tableau (liste) - ES
# Valeur de sortie : tableau trié
def trishell(t) :
    n = len(t)
    pas = n//2
    while (pas > 0) :
        for i in range(pas,n) :
            j = i-pas
            while (j >= 0) :
                if (t[j] > t[j+pas]) :
                    t[j], t[j+pas] = t[j+pas], t[j]
                    j = j-pas
                else :
                    j = -1
```

```
                # end if
            # end while
        # end for
         pas = pas//2
    # end while
    return t
# Fin de la fonction
```

## 6.5.5 - Tri rapide :

C'est un tri par dichotomie :

- on partage le tableau en deux sous-tableaux tels que tous les éléments du premier soient inférieurs à tous les éléments du second
- On réitère sur chacun des sous-tableaux jusqu'à l'obtention de sous-tableaux vides ou ne comportant qu'un seul élément
- Dans la pratique dès qu'un sous-tableau devient petit, on lui applique un tri simple (tri par insertion), le tri rapide étant lent sur de petits tableau

La dichotomie se fait par rapport à un élément appelé pivot. Le choix du pivot est important et peut faire augmenter la complexité. Un pivot parfait provoquerait une dichotomie en deux parties de même effectif. Un mauvais pivot peut engendrer une partie vide et une partie avec tous les éléments. Dans la version suivante, on prend comme pivot le premier élément du sous-tableau

```
# dichotomie d'un sous-tableau en fonction d'un pivot
# Paramètres :      bi - entier - E - borne inférieure
#                   bs - entier - E - borne supérieure
#                   t - tableau (liste) - ES - tableau complet
# Valeur renvoyée : k - entier - indice définitif du pivot

def dicho(bi, bs, t) :
    pivot = t[bi]
    # on prend pour pivot le premier élément
    n = len(t)
    i = bi
    k = bs
    while (i <= k) :
        while (t[k] > t[bi]) :
            k = k-1
        # end while
        while (i < n) and (t[i] <= t[bi]) :
            i = i+1
        # end while
        if (i < k) :
            t[i], t[k] = t[k], t[i]
            k = k-1
            i = i+1
        # end if
    # end while
    t[bi], t[k] = t[k], t[bi]
    return k
# Fin de la fonction dicho
```

```
# Fonction récursive de tri
# Paramètres :      bi - entier - E - borne inférieure
#                   bs - entier - E - borne supérieure
#                   t - tableau (liste) - ES tableau à trier
def trirap1(bi,bs,t) :
    if (bi < bs) :
        k = dicho(bi,bs,t)
        trirap1(bi,k-1,t)
        trirap1(k+1,bs,t)
    # end if
    return t
# Fin de la fonction trirap1


# Fonction principale du tri rapide
# paramètre : t - tableau (liste) - ES - tableau à trier
def trirapide(t) :
    t = trirap1(0,len(t)-1,t)
    return t
# Fin de la fonction principale
```

Dans le cas général, le tri rapide est le meilleur des tris (pas de démonstration mais nombreuses expérimentations). Ce tri étant récursif, il faut aussi regarder la complexité en place.

- Complexité en temps :
- en moyenne : O(N.log(n))

- au pire : $O(n^2)$

Complexité en place : entre O(log(n)) et 0(n)

Cette place est utilisée par la pile de sauvegarde de la récursivité. Le tri rapide n'est pas véritablement un tri sur place à cause de cette pile.

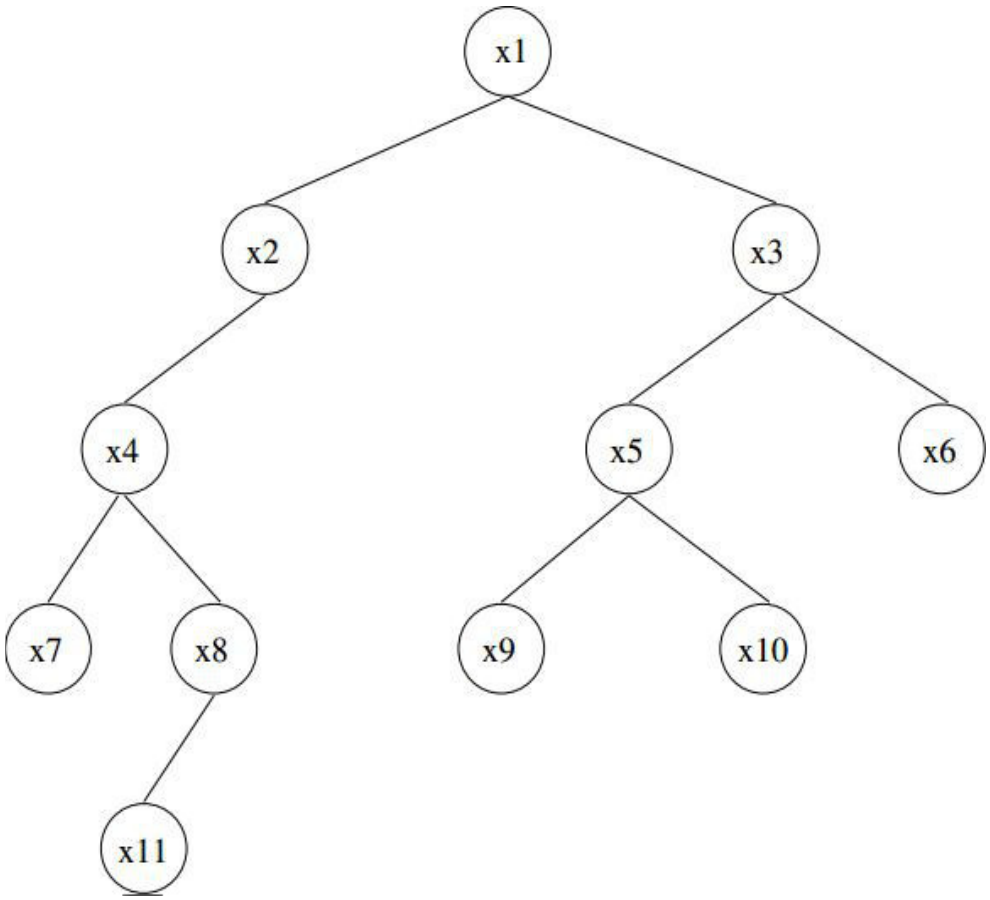
## 6.5.6 - Tri par tas :

Avant de présenter son principe, il faut introduire la notion d'arbre binaire parfait, nous reviendrons plus tard sur les arbres.

### Notion d'arbre binaire :

Un arbre binaire peut être défini [GAU 1988] comme étant soit vide soit de la forme <x,B1,B2> où B1 et B2 sont des arbres binaires disjoints et x un nœud appelé racine.

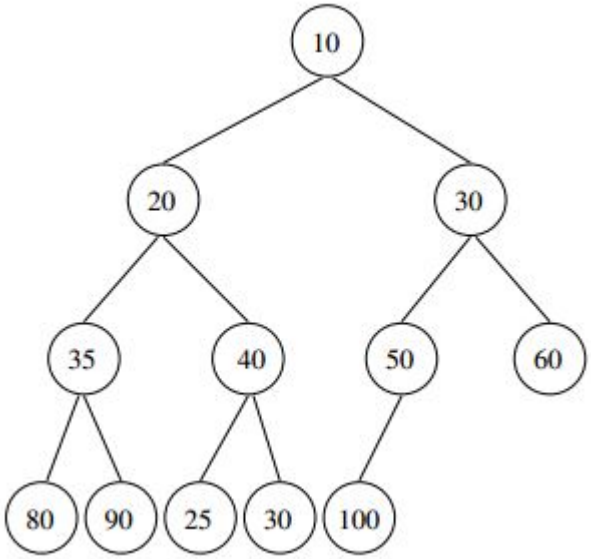
Le sous arbre B1 (respectivement B2) est appelé sous arbre gauche (respectivement droit). Le fils droit (respectivement gauche) est la racine du sous arbre droit (respectivement gauche).




Arbre binaire parfait :

Un arbre binaire de profondeur p est parfait si et seulement si :

- il est complet jusqu'à la profondeur p-1,
- toutes ses feuilles sont aux profondeurs p et p-1,
- ses feuilles de profondeur p sont regroupées à gauche.




Un arbre binaire parfait se range très facilement dans un tableau. Pour l'arbre ci-dessus :

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 30 | 35 | 40 | 50 | 60 | 80 | 90 | 25 | 30 | 100 |

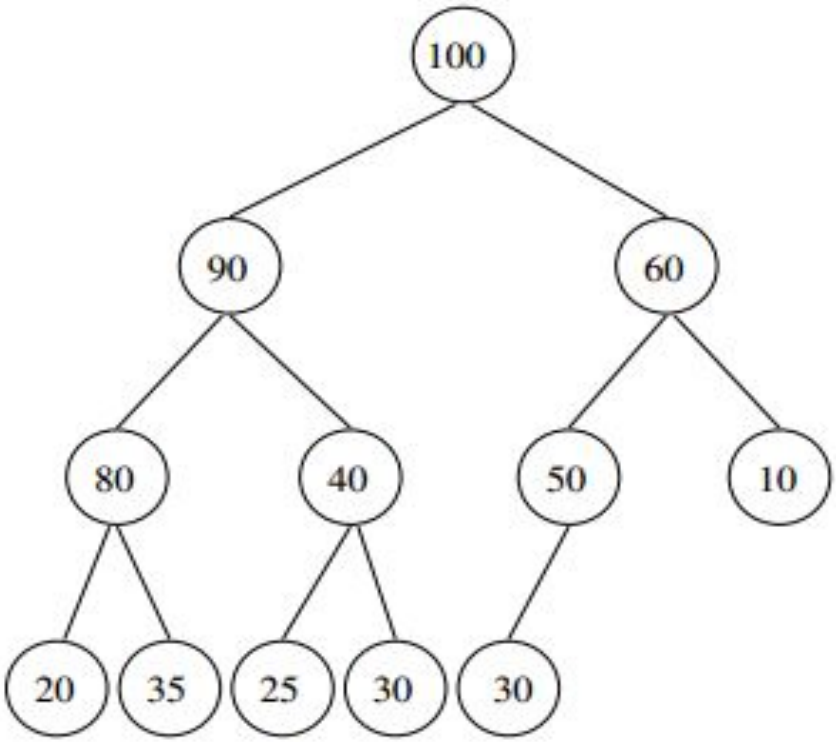
<u>Propriétés des indices pour un tableau commençant à l'indice 0 (plus simple quand commence à 1) :</u>

- racine : indice
- pour un nœud d'indice i :
  - indice du fils droit : 2(i+1)-1
  - indice du fils gauche : 2(i+1)
  - indice du père (sauf pour la racine) : (i-1)/2
- indice du premier nœud avec fils : n/2-1

Tas :

Un tas est un arbre binaire parfait partiellement ordonné. Pour tout nœud, sa valeur doit être supérieure à la valeur de chacun de ses fils. En conséquence, la valeur maximum d'un tas est à la racine de l'arbre.

Pour l'exemple :




Principe du tri par tas :

Les éléments non triés (tous au début) sont organisés en tas, le grand grand est donc à la racine. A chaque étape, on supprime la racine de l'arbre pour la mettre à la fin du tableau. On reconstitue l'arbre en remplaçant la racine par le dernier élément non trié du tableau puis on reconstruit le tas

Construction du tas initial :

Le tableau représente déjà un arbre binaire parfait. Il suffit de le réordonner. Pour ce faire, on utilise une fonction réordonnant un sous-arbre de racine rac :

```
# Fonction ordonnant un sous-arbre
# Paramètres : rac : entier
# indice de la racine du sous-arbre
#              n : entier taille du
#                  tableau de l'arbre
#              t : liste / tableau arbre
#                  à réordonner
# Valeur retournée : tableau réordonné
def ordonner (rac, n, t) :
    i = rac
    fin = (i >= (n-1)//2)
    while (not fin) :
        fin = True
        k = 2*(i+1)
        if (k > n-1) :
            j = k-1
        elif (t[k-1] > t[k]) :
           j = k-1
        else :
            j=k
        # j contient maintenant l'indice
        # du plus grand fils du nœud i
        if (t[i] < t[j]) :
            t[i], t[j] = t[j], t[i]
```

```
                # échange des valeurs
                if (j < (n-1)//2) :
                    i = j
                    fin = False
    # FIN tantque
    return t
# Fin de la fonction ordonner
```

Construction du tas :

Il faut réordonner tous les sous-arbres en partant du bas jusqu'à la racine :

```
# Fonction créant le tas initial
# Paramètre : tableau (liste)
# tableau à trier
# Valeur renvoyée : tableau(liste) :
# tableau réorganisé en tas
def creertas(t) :
    n = len(t)
    k = n//2 - 1
    while (k >= 0) :
        t = ordonner(k,n,t)
        k = k-1
    # end while
    return t
# Fin de la fonction creertas
```

Reconstruction du tas après suppression de son maximum :

```
# reconstruction du tas après l'échange
# entre le premier et le dernier élément
#  Paramètres : p : entier nombre
#   d'éléments non triés (taille du tas)
#                t : tableau (liste)
#              tableau contenant le tas
# Valeur renvoyée : tableau (liste)  tas
# reconstitué
def reconstruiretas(p, t) :
    i = 0
    while  (i <= p//2 - 1) :
        k = 2*(i+1)
        if (k > p-1) :
            j = k-1
        elif (t[k-1] > t[k]) :
           j = k-1
        else :
            j=k
        # j : indice du plus grand fils
        if (t[i] < t[j]) :
            t[i], t[j] = t[j], t[i]
            # échange des valeurs
            i = j
        else :
            break
    return t
# Fin de la fonction reconstruiretas
```

Il ne reste plus qu'à écrire la fonction principale du tri :

```
# Fonction principale de tri par tas
# Paramètre : t - tableau (liste)
# tableau initial
# Valeur renvoyée : tableau trié
def tritas (t) :
    t = creertas(t)
    k = len(t)
    while (k > 1) :
        k = k-1
        t[k], t[0] = t[0], t[k]
        t = reconstruiretas(k,t)
    # end while
    return t
# Fin de la fonction tritas
```

Complexité :

En moyenne et au pire : O(n.log(n)). C'est donc en théorie le meilleur des tris par comparaisons quand n est grand. Mais en pratique, le tri rapide est 2 à 3 fois plus rapide que le tri par tas. A noter que le tri par tas n'est pas récursif et que par conséquent il ne nécessite pas de place supplémentaire pour la pile.

## 6.6 - Tris sans comparaisons :

Les tris sans comparaisons ne sont pas applicables à tous les types de tableaux. Les éléments ou les clés (champs des éléments sur lesquels on fait le tri) doivent être discrètes et

bornées. On doit impérativement connaître le nombre et la liste des différentes valeurs possibles. Ces méthodes ne sont pas nécessairement des tris sur place.

Par contre, leurs complexités en temps sont toujours en O(n).

### 6.6.1 - Tri par comptage :

On va compter le nombre d'occurrences de chaque valeur puis les remettre triées dans le tableau. Cette méthode nécessite donc un deuxième tableau.

```
# Tri par comptage
# paramètre 1 (t) : tableau à trier
# paramètre 2 (p) : nombre de valeurs
# différentes possibles
# on suppose que les différentes valeurs
# sont les entiers de 0 à p-1

def tricomptage(t,p) :
    # initialisation à 0 des nombres
    # d'occurrences
    nbocc = [0]*p

    # calcul du nombre d'occurrences
    for elt in t :
        if (elt < 0) or (elt >= p) :
            # valeur interdite
```

```
            return None
        else :
            nbocc[elt]+=1

    # détermination du résultat
    t1 = []
    for i in range(p) :
        t1 = t1+[i]*nbocc[i]
    return t1
```

Exercices : écrire des variantes avec des valeurs ne commençant pas à 0 puis avec les différentes valeurs dans une liste.

## 6.6.2 - Tri par paquets :

On fait une liste par valeur puis on concatène toutes les listes.

```
# Tri par paquets
# paramètre 1 (t) : tableau à trier
# paramètre 2 (p) : nombre de valeurs
# différentes possibles
# on suppose que les différentes valeurs
# sont les entiers de 0 à p-1

def tripaquets(t,p) :
    # initialisation des p listes à vide
    # paquets = [[]]*p
```

```
    # pourquoi cela ne marche pas
    paquets = []
    for i in range(p) :
        paquets.append([])

    # affectation de chaque élément à
    # une liste
    for elt in t :
        if (elt < 0) or (elt >= p) :
            # valeur interdite
            return None
        else :
            paquets[elt].append(elt)

    # détermination du résultat
    t1 = []
    for i in range(p) :
        t1 = t1+paquets[i]
    return t1
```

Mêmes exercices qu'avant.

### 6.6.3 - Drapeau tricolore :

On dispose de n boules physiques (pas virtuelles) rouges, vertes ou bleues que l'on souhaite réordonner dans cet ordre. On codera ces couleurs par les entiers 0 (rouge), 1 (vert) et 2 (bleu).
Sont exclus les tris par comparaisons et le tri par comptage

(ne marche pas avec des objets physiques). Le tri par paquets marche ici mais on peut faire un tri sur place en O(n) en organisant judicieusement le tableau des boules.

| 0 | | | n-1 |
|---|---|---|---|
| boules rouges | boules vertes | boules non triées | boules bleues |

- Comment traiter la première boule de la partie non triée en fonction de sa couleur
- En déduire un programme.

## 7. LA RÉCURSIVITÉ

### 7.1 - Exemples :

Factorielle n :

Approche itérative : pour mémoire (fait en cours)

Approche récursive :
2 (ou 3) cas à traiter :

- (éventuellement) n n'est pas un entier >= 0 : None, cela peut être précisé avant l'appel
- n = 0 ou n = 1 : 1
- n > 0: n.((n-1) !)

```
def factrec(n) :
    if  n <= 1 :
```

```
        return 1
    else :
        return n*factrec(n-1)
```

Appel principal :

```
  if (n >= 0) :
        res = factrec(n)
    else :
        res = None
```

Recherche dichotomique :

```
def dichorec(bi, bs, t, x) :
    if bs < bi :
        return -1
    else :
        mi = (bi+bs)//2
        if t[mi] > x :
            return dichorec(bi,mi-1,t,x)
        elif t[mi] < x :
            return dichorec(mi+1,bs,t,x)
        else : # égalité
            return mi
```

on écrit aussi une fonction principale :

```
def dicho2 (t, x)
    n = len(t)
    return dicorec(0, n-1, t, x)
```

## 7.2 - Résolution récursive de problèmes :

L'analyse d'un problème conduit habituellement à sa décomposition en sous problèmes. Quand l'un (ou plusieurs) de ces problèmes est le problème initial, la méthode de résolution est dite récursive. Quand on emploie cette technique, il faut toujours préciser quel est l'appel principal.
La récursivité en algorithmique se traduit par des sous programmes qui peuvent s'appeler eux mêmes. De nombreux langages gèrent la récursivité (ADA, Pascal, C, python, ...). Dans le cas contraire (FORTRAN, COBOL, ...), il faudra la simuler.

Remarques :

- il faut toujours précisé l'appel principal

- il existe de la récursivité croisée : A0 → A1 → ... → Ai → … → An → A0

- l'intérêt de la récursivité réside dans la facilité de conception des algorithmes. Au niveau de la programmation il est peut être préférable de l'éliminer, car elle est souvent gourmande en temps et en place mémoire. Maintenant, beaucoup de compilateurs font d'eux mêmes les optimisations.

- Il existe une forme dégénérée de récursivité, la récursivité à droite : quand l'appel récursif est la dernière instruction exécutée du programme appelant.

Dans ce cas, il est très facile d'éliminer la récursivité (absence de sauvegarde).

## 7.3 - Mécanismes :

A l'appel récursif :

- sauvegarde des variables locales et des paramètres dans une pile
- mécanisme habituel d'appel de sous-programme

Au retour d'un sous-programme récursif :

- mécanisme habituel de retour de sous-programme
- restitution des objets sauvegardés dans la pile

Quand le langage n'est pas récursif, il faut gérer :

- la pile de sauvegarde
- la mise en place des paramètres
- la pile des adresses de retour

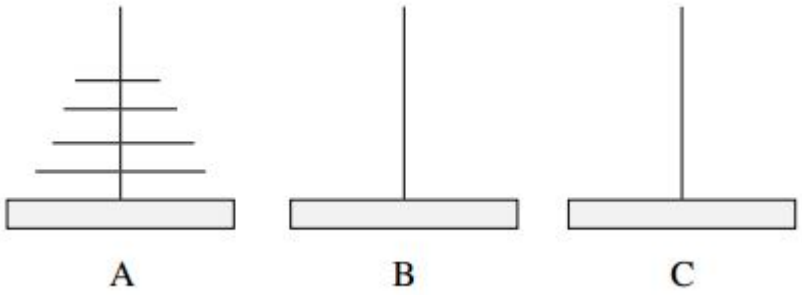
## 7.4 - Applications :

### Tours de Hanoï :

N disques concentriques sont placés par diamètre décroissant autour d'un piquet A. Le but consiste à faire passer tous les disques sur un piquet C en utilisant un piquet intermédiaire B en respectant les règles suivantes :

- on ne peut déplacer qu'un disque à la fois,
- il est interdit de poser un disque sur un disque de diamètre inférieur.

Situation initiale :




Ce problème est assez difficile à résoudre directement de façon itérative mais la version récursive est simple :

```
def Hanoï(n, a, c, b) :
  if n == 1 :
    print('disq 1 de ',a,' vers ',c)
  else :
    Hanoï(n-1,a,b,c)
    print('disq ',n,' de ',a,' vers ',c)
    Hanoï(n-1,b,c,a)
# Appel principal : Hanoï(4,'A','C','B')
```

Fibonacci :

fib(0) = 0 ; fib(1) = 1 ; fib(n) = fib(n-1)+fib(n-1) si n > 1

La version récursive paraît évidente :

```
def fibrec(n) :
    if n <= 2 : # on considère  n >=0
        return n
    else :
```

```
        return fibrec(n-1)+fibrec(n-2)
# appel principal : fibrec(10)
```

Questions :

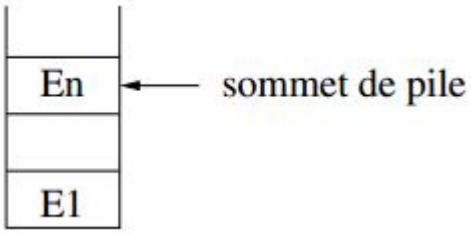
- tracer l'arbre des appels récursifs pour n=5
- même question pour n = 12 (bon courage)
- en déduire pourquoi cette version est calamiteuse en termes de complexité en temps
- en écrire une version itérative en O(n)

## 8. LES PILES ET LES FILES D'ATTENTE

Ce sont des structures de données internes avec des accès (ajout, suppression, recherche) bien définis. En python elles seront représentées par des listes.

### 8.1 - Les piles :

On ne peut ajouter un élément qu'à la fin de la pile et on ne peut accéder (ou supprimer) qu'au dernier élément. On parle d'accès LIFO : Last In First Out.




Opérations sur les piles :

| DONNÉE(S) | OPÉRATION | RÉSULTAT(S) |
|---|---|---|
| / | création | pile vide |
| pile + élément | ajout | pile |
| pile | suppression simple | pile / None |
| pile | suppression 2 | pile + élément / None |
| pile | consultation | élément / None |
| pile | test de pile vide | booléen |

Le résultat None est obtenu pour une suppression dans une pile vide ou pour la consultation du sommet d'une pile vide.

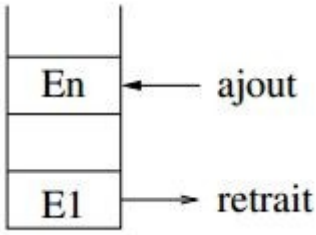
Les algorithmes de ces opérations sont immédiats avec les listes python. Dans les autres langages, les piles sont aussi faciles à coder en les représentant avec des tableaux ou éventuellement avec des listes chaînées (pointeurs).

Les piles sont surtout utilisées pour les versions itératives des algorithmes récursifs.

Exercice : évaluation d'une expression numérique suffixée

## 8.2 - Les files d'attente :

On ne peut ajouter un élément qu'à la fin de la file. On ne peut accéder qu'au premier élément et supprimer que le premier élément. On parle d'accès FIFO : First In, First Out.




Les opérations sur les files sont les mêmes que sur les piles et leurs algorithmes sont aussi simples.

Les files d'attente sont très utilisées en informatique, en particulier pour les systèmes d'exploitation.

## 9. L'ADRESSAGE DISPERSÉ

### 9.1 - Principe :

Les opérations de recherche (mise à jour), ajout et suppression d'un élément dans une ensemble sont trois opérations très utilisés et doivent être rapides. Les méthodes déjà étudiées les font avec une complexité en O(n) pour au mois l'une de ces opérations, ce qui est trop long : on souhaite avoir une complexité constante O(a) ou logarithmique O(log(n)). Ceci est réalisé par des méthodes d'adressage dispersé ou par des méthodes utilisant les arbres. Ces deux techniques sont utilisées pour l'indexation dans les SGBD.

Le principe de l'adressage dispersé consiste à obtenir en temps constant l'adresse d'un élément en fonction de sa valeur ou de la valeur de sa clé dans une table. Pour ce faire

on va associer à chaque élément ei ($0 \leq i < n$) d'un ensemble E, une valeur f(ei) qui sera l'adresse de ei dans une table à M places ou M entrées. La fonction f prend ses valeurs dans l'ensemble {1,2,..., m}. Cette fonction est appelée fonction de dispersion on encore fonction de hachage.

Si cette fonction est injective, ce qui est rare, tout se passe bien : deux éléments différents ont des valeurs de dispersion (donc des adresses) différentes. Dans le cas contraire,
$\exists$ ei, ej $\in$ E, ei $\neq$ ej / f(ei) = f(ej) : il se produit une collision (deux éléments différents ont la même adresse). Nous verrons par la suite comment résoudre les collisions.

## 9.2 - Fonction de dispersion :

Une fonction de dispersion f, est une fonction sur l'ensemble E des éléments à valeurs dans un intervalle de l'ensemble des entiers naturels. La quantité f(ei) est appelée valeur de dispersion de l'élément ei. La fonction de dispersion n'est jamais injective dans la pratique.

On définit la fonction **"même valeur de dispersion"**, notée R, sur l'ensemble E : (ei R ej) $\Leftrightarrow$ (f(ei) = f(ej)). R est une relation d'équivalence à M classes, chaque classe correspond à une entrée de la table.

Une bonne fonction doit répartir uniformément les valeurs de dispersion entre 1 et M. De plus, son calcul doit être rapide (en temps constant par rapport à N et M).

Fonction habituelle pour les chaînes de caractères : somme des codes ASCII modulo M :

```
def fdisp(M, mot):
    h=0
    for lettre in mot :
        h+=ord(lettre)
        # ord est une fonction renvoyant
        # le code ASCII d'un caractère
    return h%M
```

Cette fonction est très mauvaise quand il y a beaucoup d'anagrammes. Il existe d'autres techniques permettant d'avoir des fonctions de dispersion efficaces.

## 9.3 - Résolution des collisions :

Il existe deux types de méthodes : les méthodes directes qui calculent une nouvelle adresse et les méthodes indirectes qui mettent dans une liste les éléments d'une même classe d'équivalence de la relation R.

### 9.3.1 - Les méthodes directes :

Il est impératif, pour ce type de méthodes, que le nombre M d'entrées dans la table soit au moins égal au nombre N d'éléments. Il s'agit alors de construire une fonction F d'essais successifs.

$F : E \rightarrow \{1, \ldots, M\}^M$

ei → permutation de {1, …, M}
F(ei) = {$f_1$(ei), …, $f_j$(ei), …, $f_M$(ei)} où les $f_j$(ei) sont 2 à 2 distinctes.

Les différentes méthodes de résolution directes se distinguent selon le choix de la fonction d'essais successifs.

Algorithme général de recherche ajout d'un élément e dans une table (liste) T :

```
def rechajout(table, e) :
    # f[j] : fonction fj
    j = 1
    adr = f[1][e]
    while (table[adr]!= None) &
          (table[adr]!= e) :
        j += 1
        adr = f[j][e]
    if T[adr] == e :
        # traitement
    else :
        # ajout
```

Le hachage linéaire est une technique assez utilisée qui consiste à essayer la place suivante. En notant g la fonction de dispersion initiale, on prend :
$f_1(e) = g(e)$ et
$f_j(e) = (g(e)+j-1) \% M$

Évaluation :

Les méthodes directes sont efficaces lorsque le nombre de collisions est faible et nécessitent moins de place que les méthodes indirectes (chaînage). Le principal inconvénient est le risque assez important de collisions secondaires (emplacement de la table occupé par un élément d'une autre classe d'équivalence). Il faut éviter de faire la même séquence d'essais pour les éléments d'une même classe (inconvénient du hachage linéaire) en utilisant par exemple une deuxième fonction de hachage.

### 9.3.2 - Les méthodes indirectes :

Dans ce type de méthodes les éléments d'une même classe d'équivalence seront stockés dans une liste. Cette liste peut être triée ou non.

La table sera donc une liste de listes. Dans les langages sans listes, les listes seront codées avec des pointeurs.

## 10. LES ARBRES

### 10.1 - Introduction :

Un arbre est un graphe sans cycle tel que :

- il existe un et un seul sommet sur lequel n'arrive aucun arc
- tout autre sommet a exactement un arc incident
- il existe un chemin unique de la racine à tout sommet

Un arbre est un ensemble d'éléments appelés nœuds ou sommets organisés de façon hiérarchique à partir d'un nœud spécial sans prédécesseurs, la racine.

Notations et vocabulaire :

- un nœud y est dit fils du nœud x (x est le père de y) si et seulement si il existe un arc allant de x à y ;
- tout nœud à l'exception de la racine possède exactement un père
- un nœud peut posséder zéro, un ou plusieurs fils, un nœud sans fils est appelé feuille.

Profondeur d'un nœud :

- prof(racine) = 0
- prof(x) = prof(y)+1 si y est le père de x ($x \neq$ racine)

Profondeur d'un arbre A :

$prof(A) = \max\{prof(x) / x \in A\}$

Applications :

- arbres généalogiques (qui sont rarement de arbres surtout dans les familles royales...)
- tournois à élimination directe
- indexation de fichiers (SGBD)
- organisation des fichiers dans les systèmes d'exploitation
- sémantique
- mathématiques
- modélisation de problèmes

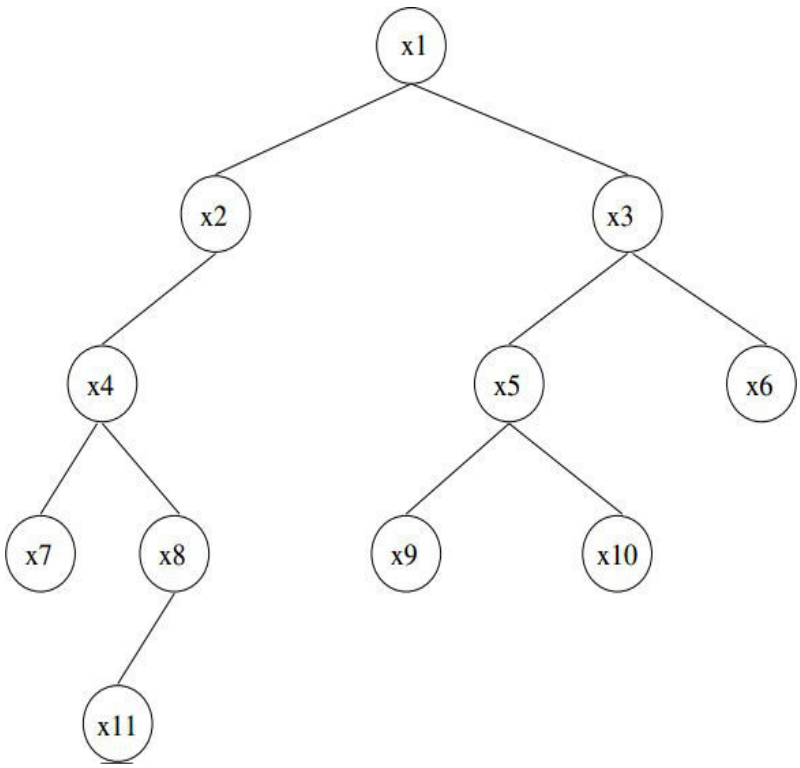
- …

## 10.2 - Les arbres binaires :

### 10.2.1 - Définition et codage :

Un arbre binaire est un arbre dont tout nœud possède au plus deux fils. Nous en avons vu un autre définition lors du tri par tas : un arbre binaire est soit vide, soit de la forme <x,B1,B2> où B1 et B2 sont des arbres binaires disjoints et x est un nœud appelé racine.

Un exemple d'arbre binaire :




Représentations internes :

- avec un tableau : convient surtout pour les arbres binaires complets ou parfaits ;

- avec des pointeurs : représentation classique en C ou Pascal ;
- avec des listes : dans tous les langages disposant de listes : python, Prolog, Lisp, .. ;
- certains langages (Prolog,...) possèdent un type arbre et proposent un codage spécifique ;
- représentations classiques des graphes : très mal adaptées pour les opérations de recherche, ajout, suppression.

En python, nous représenterons un nœud par une liste de 3 éléments :

- premier élément de la liste : valeur du nœud ,
- deuxième élément de la liste : sous-arbre gauche (liste ou None)
- troisième élément de la liste : sous-arbre droit (liste ou None)

La représentation avec des listes permet de coder facilement les arbres généraux.

Pour l'arbre précédent :
[x1, [x2, [x4, [x7,None,None], [x8, [x11,None,None], None], [x3, [x5, [x9,None,None], [x10,None,None]], [x6,None,None]]]

10.2.2 - Parcours :

En profondeur à gauche :

```
# parcours préfixé en profondeur à
# gauche
def prefixe(A) :
    if A!= None :
        print(A[0])
        prefixe(A[1])
        prefixe(A[2])
# Appel principal avec la liste complète
```

pour l'exemple : x1, x2, x4, x7, x8, x11, x3, x5, x9, x10, x6

```
# parcours infixé en profondeur à
# gauche
def infixe(A) :
    if A!= None :
        infixe(A[1])
        print(A[0])
        infixe(A[2])
# Appel principal avec la liste complète
```

pour l'exemple : x7, x4, x11, x8, x2, x1, x9, x5, x10, x3, x6

```
# parcours suffixé en profondeur à
# gauche
def suffixe(A) :
    if A!= None :
        suffixe(A[1])
        suffixe(A[2])
        print(A[0])
# Appel principal avec la liste complète
```

pour l'exemple : x7, x11, x8, x4, x2, x9, x10, x5, x6, x3, x1

Parcours divers :

- parcours en profondeur à droite,
- parcours en largeur (par niveau)
- parcours en fonction d'une fonction de guidage (heuristique)

### 10.2.3 - Les arbres binaires complets :

Un arbre binaire complet de profondeur p est un arbre binaire de profondeur p dont tous les nœuds de profondeur 0 à p-1 possèdent exactement 2 fils. Un arbre complet de profondeur p contient exactement $2^{p+1}$ - 1 nœuds.

Un tel arbre est souvent représenté de façon contiguë dans un tableau, soit selon l'ordre préfixé soit sous la forme utilisée pour le tri par, soit sous une autre forme (par exemple en ordre préfixé).

## 10.3 - Les arbres binaires de recherche :

### 10.3.1 - Définition :

Un arbre binaire de recherche (ABR) est un arbre binaire tel qu'en chacun de ses nœuds x :

- tous les nœuds du sous-arbre gauche de x ont une valeur inférieure ou égale à la valeur de x
- tous les nœuds du sous-arbre droit de x ont une valeur

supérieure à la valeur de x

Par conséquent, le parcours infixé d'un ABR produit la suite des valeurs des nœuds triée en ordre croissant.

10.3.2 - Recherche :

Dans un arbre binaire simple, on peut être amené à parcourir tout l'arbre pour rechercher un élément. Avec un ABR, dans le pire des cas, on va simplement parcourir un chemin entre la racine et une feuille.

```
# élément recherché x :
# arbre : A
def recherche_ABR(A,x):
    if A == None:
        return False
    elif x < A[0]: # à gauche
        return recherche_ABR(x0,A[1])
    elif x > A[0]: # à droite
        return recherche_ABR(x0, A[2])
    else: # gagné (x0 trouvé dans A)
        return True
```

Appel principal : avec tout l'arbre
Complexité en temps : O(p) où p est la profondeur de l'arbre

10.3.3 - Ajout :

On va ajouter un élément dans une feuille de l'ABR. Il

existe un algorithme d'ajout à la racine qui est plus compliqué et n'a pas d'intérêt particulier.

```
# élément à ajouter x :
# arbre : A
def ajoutABR(A,x) :
  nouveau=[x,None,None]
  if A == None :
    return nouveau
  else :
    fin = False
    noeud = A
    while (fin != True) :
      if x <= noeud[0] :
        if noeud[1] != None :
          noeud = noeud[1]
        else :
          noeud[1] = nouveau
          fin = True
        else :
          if noeud[2] != None :
            noeud = noeud[2]
          else :
            noeud[2] = nouveau
            fin = True
    return A
```

Complexité en temps : O(p) où p est la profondeur de l'arbre

## 10.3.4 - Suppression :

On va toujours se ramener à la suppression physique d'une feuille ou d'un nœud n'ayant qu'un seul fils. Il est trop difficile de restructurer l'arbre sinon. Lorsqu'on a alloué dynamiquement de la mémoire pour stocker l'arbre, il ne faut pas oublier de libérer la place occupée par le nœud supprimé. C'est le cas en langage C mais pas en python. On a 3 cas :

- le nœud à supprimer est une feuille : on la supprime directement
- le nœud à supprimer n'a qu'un fils : on remplace le nœud par son fils
- la valeur à supprimer est dans un nœud qui a deux fils : on commence par supprimer le nœud contenant la plus grande valeur de son sous-arbre gauche en mémorisant cette plus grande valeur qui est ensuite placée dans le nœud contenant la valeur à supprimer. On peut aussi le faire avec la plus petite valeur du sous-arbre droit.

### Algorithme de recherche du maximum d'un sous-arbre avec destruction du nœud :

```
# renvoie le maximum de l'ABR A et
# supprime ce maximum de A
# cette fonction ne doit être appelée
# que par suppABR :
# A ne peut donc pas être vide
def maxsupABR(A) :
    assert A != None
```

```
    if A[2] == None :
        return A[1], A[0]
    else :
        A[2], res = maxsupABR(A[2])
        return A, res
# retour de deux valeurs, le nouvel
# arbre et le maximum
```

Algorithme général de suppression :

```
# suppression de x dans ABR A
def suppABR(A,x) :
    if A == None : # élément absent !
        return None
    elif A[0] > x :
        A[1] = suppABR(A[1],x)
        return A
    elif A[0] < x:
        A[2] = suppABR(A[2],x)
        return A
    else : # A[0] == x
        if A[1] == None :
            # pas de fils gauche
            return A[2]
        elif A[2] == None :
            # pas de fils droit
            return A[1]
        else : # fils gauche et droit
            A[1], A[0] = maxsupABR(A[1])
            return A
```

Complexité en temps : O(p) où p est la profondeur de l'arbre

## 10.4 - Les arbres AVL :

### 10.4.1 - Introduction :

La complexité des opérations de recherche, de suppression ou d'ajout dans un ABR est O(p) où p est la profondeur de l'arbre. Or p est compris entre log2(n) et n et dans ce dernier cas la complexité est mauvaise. De plus un arbre parfaitement équilibré peut rapidement dégénérer après une série de suppressions ou d'ajouts.

* un arbre binaire est dit **H-équilibré** si en tout nœud, les profondeurs des sous-arbres gauche et droit diffèrent au plus de 1.

* fonction mesurant le déséquilibre :
  Des(∅) = 0 et
  Des(<x,g,d>) = prof(g) - prof(d)

* un arbre binaire A est H-équilibré si et seulement si, pour chacun de ses sous-arbres B on a : Des(B) ∈ {-1,0,1}.

* tour arbre binaire H-équilibré ayant N nœuds a une hauteur h vérifiant : $\log_2(N+1) \leq h+1 \leq \sqrt{2}.\log_2(N+2)$

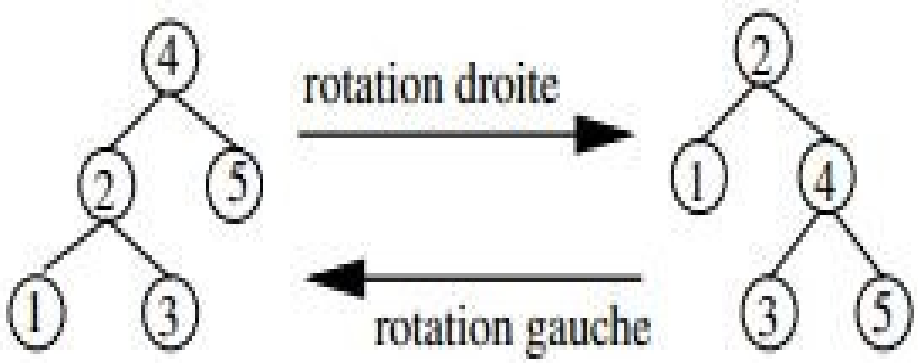
* un arbre AVL est un ABR H-équilibré. On ajoute un

champ dans le représentation interne d'un nœud : son déséquilibre. En python un nœud sera une liste de 4 valeurs.

10.4.2 - Les rotations :

Les rotations sont des opérations locales permettant de rétablir l'équilibre d'un arbre AVL après un ajout ou une suppression. Il existe 4 types de rotations : les rotations gauches, les rotations droites, les rotations gauches droites et les rotations droites gauches.

Les rotations simples :



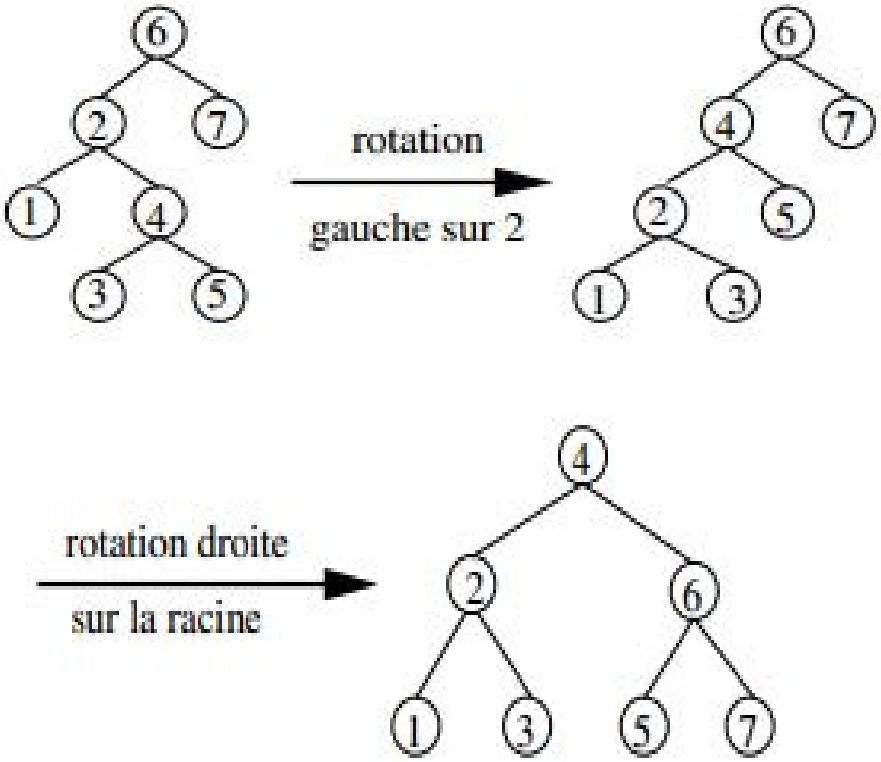

Rotation droite :

```
def rd(a) :
    b = a[1]
    a[1] = b[2]
    b[2] = a
    return b
```

Attention aux effets de bord, l'arbre passé en paramètre est détruit. La fonction pour la rotation gauche est analogue en inversant les indices (1 et 2).

Les rotations composées :

La rotation gauche droite est constituée d'une rotation gauche sur le sous-arbre gauche puis d'une rotation droite sur l'arbre. Exemple :



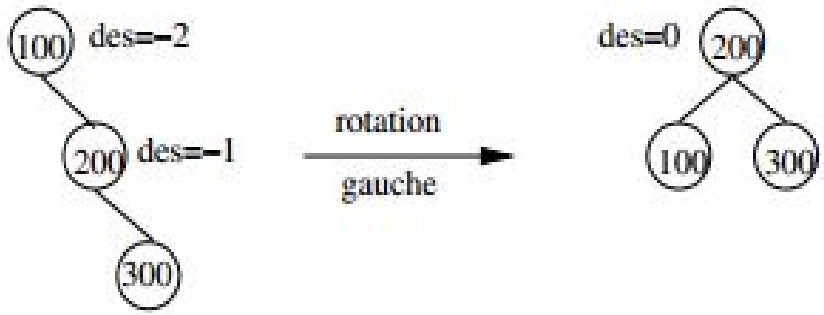

La rotation droite gauche est constituée d'une rotation droite sur le sous-arbre droit puis d'une rotation gauche sur l'arbre.

10.4.3 - Ajout :

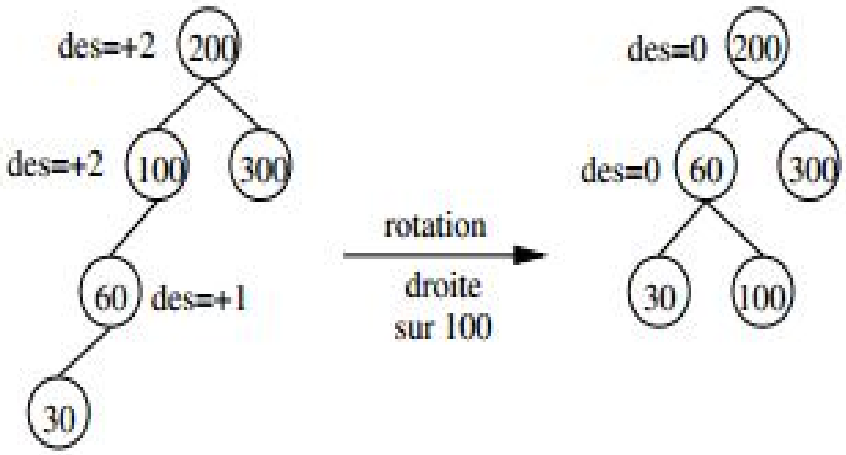
L'ajout dans les arbres AVL se fait au niveau des feuilles. Après (ou pendant) cette phase, il est nécessaire de mettre à jour le déséquilibre des nœuds sur le chemin entre la racine et le nouveau nœud puis si c'est nécessaire de rééquilibrer

l 'arbre. Une seule rotation simple ou composée suffit pour rétablir l'équilibre à condition de le faire après chaque ajout (ou suppression). Nous allons maintenant voir sur un exemple les 4 différents cas de figure :

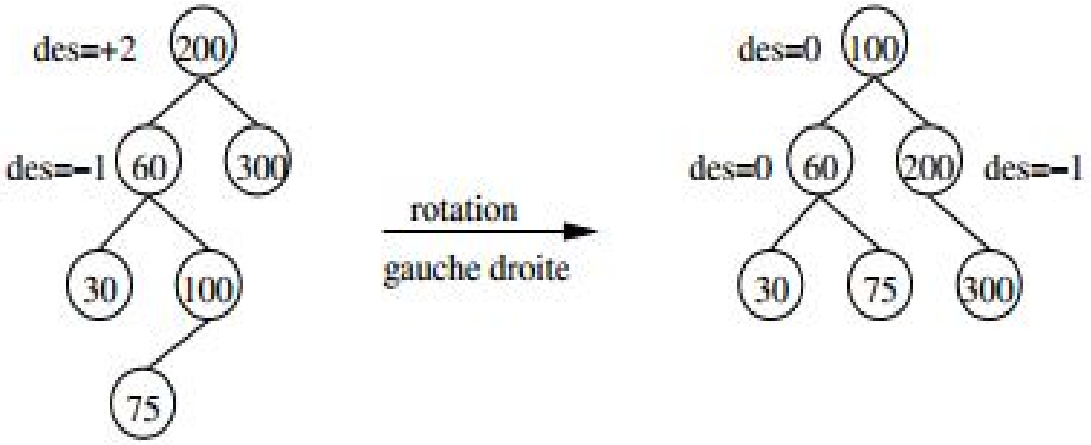
Ajouts successifs de 100, 200 et 300 dans un arbre vide :



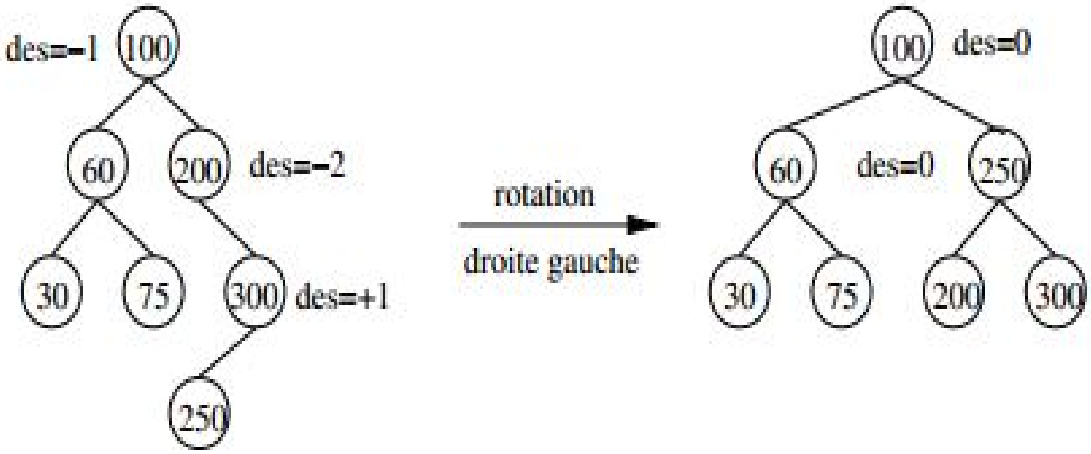

Ajouts successifs de 60 et 30 :




Ajout de 75 :




Ajout de 250 :




Récapitulatif des différents cas :

On suppose que le déséquilibre des nœuds a été mis à jour. On considère uniquement les nœuds situés sur le chemin entre la racine et le nouvel élément ajouté. Si aucun nœud n'a un déséquilibre égal à +2 ou -2 alors l'arbre est équilibré. Sinon on note N1 le nœud le plus profond de déséquilibre égal à +2 ou -2 et N2 son fils sur le chemin menant au nouveau nœud.

| des(N1) | des(N2) | traitement |
|---|---|---|

| | | |
|---|---|---|
| 2 | 1 | rotation droite sur N1 |
| 2 | -1 | rotation gauche droite sur N1 |
| -2 | 1 | rotation droite gauche sur N1 |
| -2 | -1 | rotation gauche sur N1 |

Algorithme synthétique :

- ajout simple (voir ABR) avec, le cas échéant le repérage du nœud le plus profond de déséquilibre -2 ou 2
- mise à jour des déséquilibres à partir de la racine si l'arbre reste équilibré ou à partir du nœud le plus profond de déséquilibre -2 ou 2 quand il existe
- rééquilibrage avec une rotation si nécessaire

Python et sa représentation d'arbre en listes ne facilite pas la mise en œuvre de l'algorithme : quelle est l'adresse ou les indices d'un nœud ? De plus le rééquilibrage, s'il est simple dans son principe comporte de nombreux cas particuliers pour la mise à jour des déséquilibres après une rotation. Parmi les rares livres qui donnent des algorithmes détaillés, on trouve des erreurs.

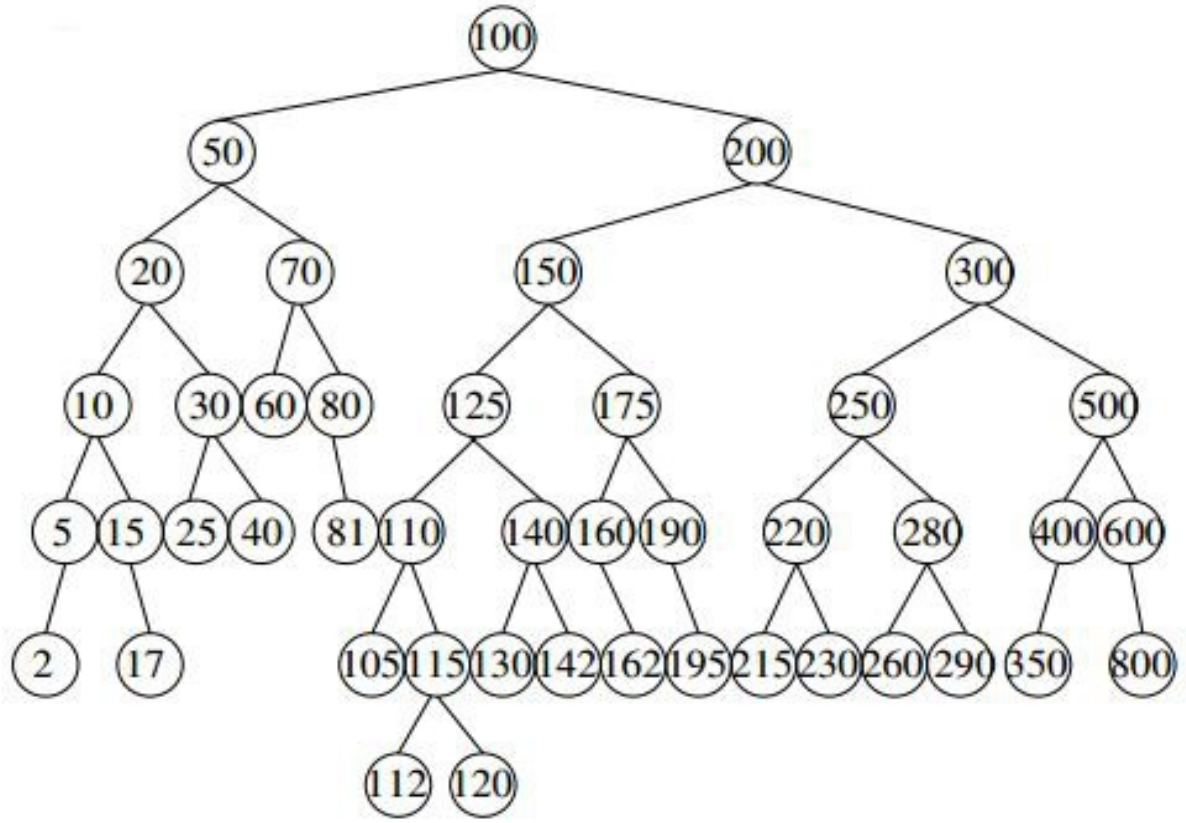
Il faut stocker les déséquilibre et non pas le recalculer pour ne pas avoir une complexité en O(n). La complexité de cet algorithme est en 0(log(n)) : parcours d'un chemin entre la racine et une feuille dans un arbre équilibré.

10.4.4 - Suppression :

On commence par supprimer l'élément sans mettre à jour la fonction de déséquilibre et sans rééquilibrer l'arbre. C'est le même algorithme que celui de la suppression dans un ABR.

On peut montrer que des rotations peuvent être nécessaires uniquement sur le chemin allant du nœud physiquement supprimé à la racine. On applique l'algorithme de rééquilibrage (voir ajout) sur tout le nœud de ce chemin ayant un déséquilibre +2 ou -2. Il faut donc conserver dans une liste le chemin en sens inverse.

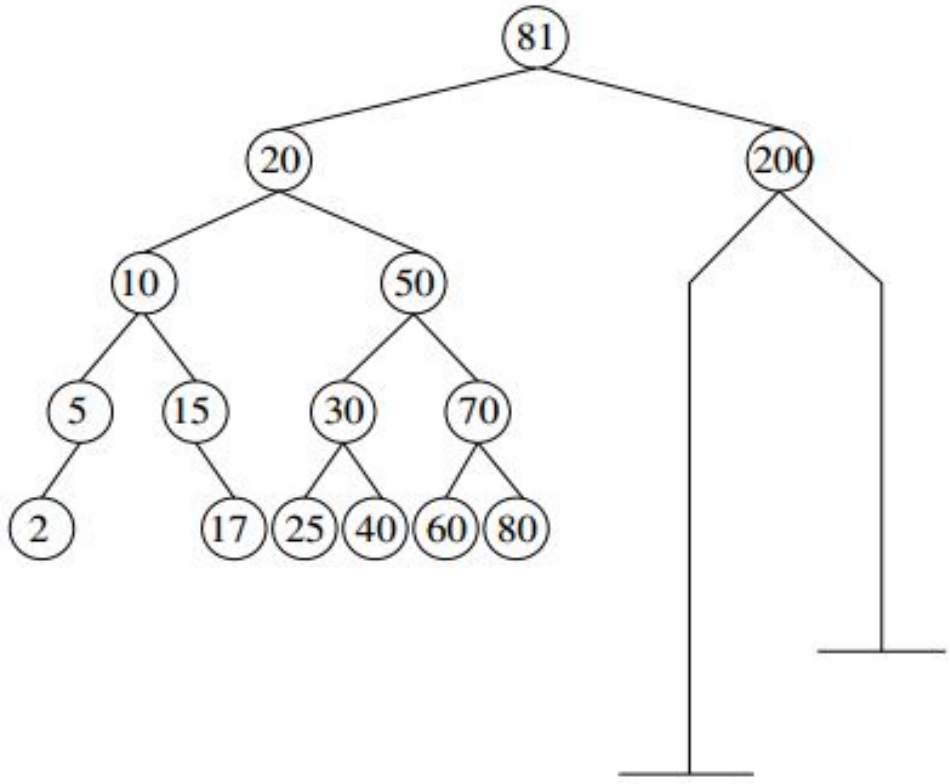
Exemple :




Soit 100 (contenu de la racine), l'élément à supprimer. On

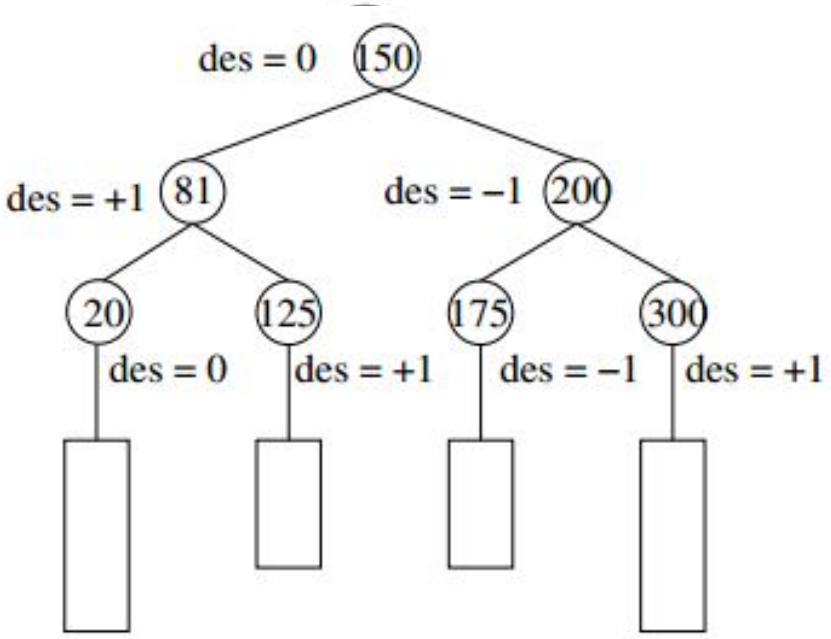
commence par appliquer l'algorithme de suppression sans se préoccuper du déséquilibre de l'arbre : suppression physique de nœud contenant 81 et recopie de 81 dans le nœud racine.

Considérons maintenant les déséquilibres sur le chemin entre le nœud supprimé et la racine : les déséquilibres des nœuds 80 et 70 sont nuls mais le déséquilibre en 50 est égal à +2. En ce nœud, la situation est +2, +1 : on applique une rotation droite sur le nœud 50 :




Après cette rotation, le déséquilibre du nœud courant, qui contient maintenant l'élément 20, est égal à 0. On poursuit donc l'algorithme en remontant vers la racine. Le déséquilibre de la racine est devenu -2. Il faut donc lui

appliquer l'algorithme de rééquilibrage. Nous sommes dans le cas de figure -2, +1 : on lui applique une rotation droite gauche :




rotation droite sur le sous-arbre droit :puis rotation gauche sur la racine :

L'arbre est maintenant équilibré.

10.4.5 - Complexités :

Les complexités au pire :

- recherche : $\log_2(n)+1$ comparaisons
- ajout : $\log_2(n)+1$ comparaisons et une rotation
- suppression : $\log_2(n)+1$ comparaisons et $\log_2(n)$ rotations
- statistiquement : une rotation pour cinq suppressions.

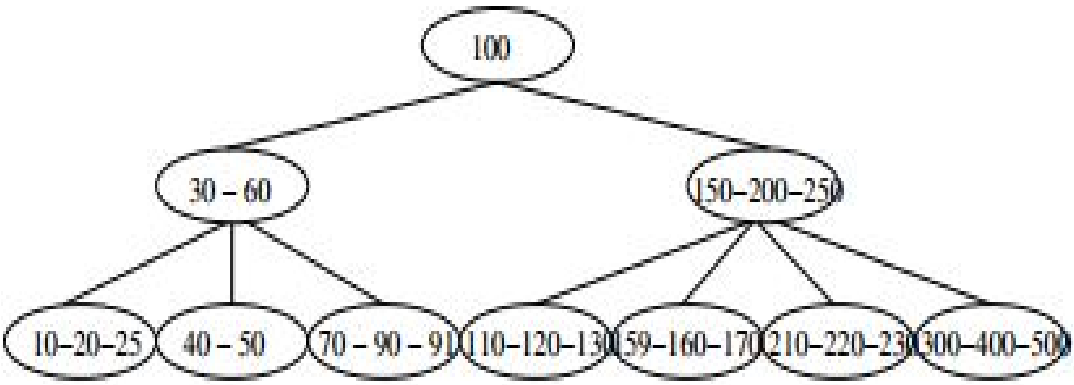
## 10.5 - Les arbres 2-3-4 :

### 10.5.1 - Définition :

Un arbre 2-3-4 (A2324) est un arbre de recherche dont chaque nœud possède de 1 à 3 éléments soit de 2 à 4 fils sauf pour les feuilles. Ceci permet d'imposer une nouvelle contrainte : toutes les feuilles doivent être à la même profondeur. La hauteur h d'un arbre 2-3-4 possède la propriété suivante (n : nombre d'éléments) :

$$\log_4(n+1) \leq h+1 \leq \log_2(n+1)$$

Exemple :




Représentation interne :

Il existe plusieurs façons de représenter un arbre 2-3-4 en python : soit directement avec une liste soit avec un ABR.

Directement avec une liste : chaque nœud sera codé par une liste comprenant dans l'ordre les valeurs stockées dans le nœud puis les sous-arbres. Soit pour l'exemple :

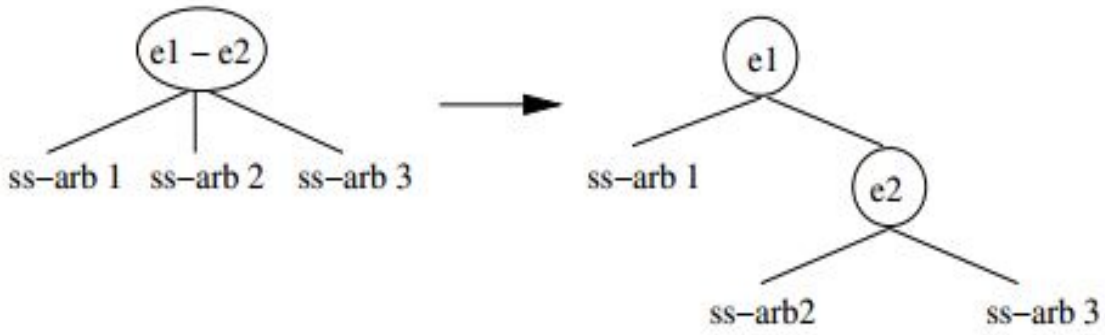
```
[100,     [30,60,   [10,20,25,None,None,None,None],
                    [40,50,None,None,None],
                    [70,90,91,[None,None,None,None]
          ],
          [150,200,250,
                    [110,120,130,None,None,None,None],
                    [159,160,170,None,None,None,None],
                    [210 ,220,230,None,None,None,None],
                    [300,400,500,None,None,None,None]
          ]
]
```

Ici l'indentation ne sert que pour améliorer la lisibilité. On peut imaginer d'autres variantes en changeant l'ordre d'un nœud.

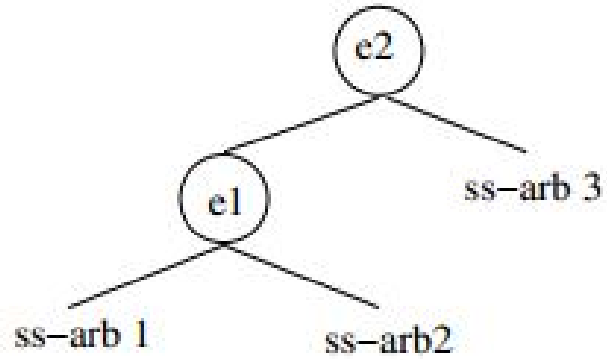
Avec un arbre binaire de recherche bicolore : chaque nœud et ses sous-arbres vont être représenter avec des arbres AVL. Trois cas sont possibles :

- Le nœud ne contient qu'un seul élément : le nœud sera représenter de façon identique, c'est un nœud normal d'un arbre AVL.

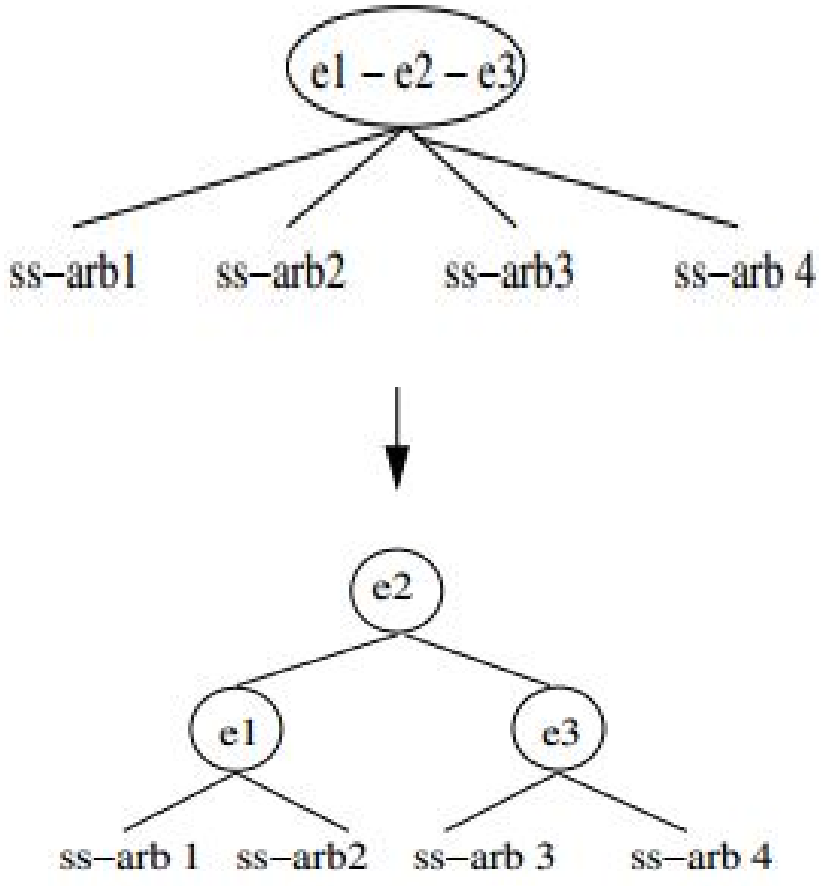
- Le nœud contient 2 éléments : deux représentations au choix sont possibles :




ou encore :




- Le nœud contient 3 éléments : une seule représentation :



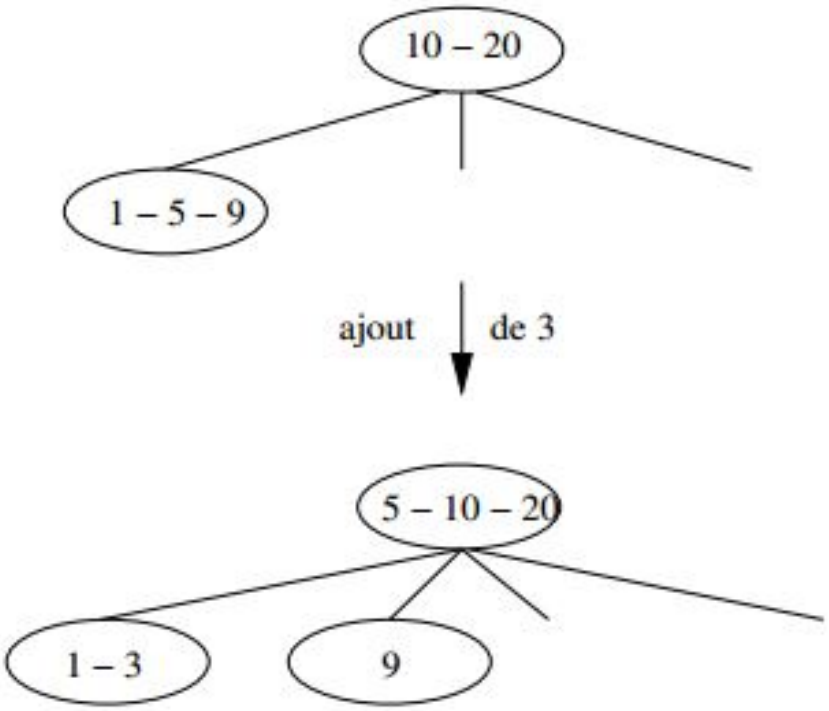
Dans cette représentation, un nœud est dit **rouge** si et seulement si son élément est un jumeau de l'élément de son père, dans le cas contraire, il est dit **blanc**.

Exercices :

- Dessiner l'arbre binaire de recherche obtenu à partir de l'arbre 2-3-4 de l'exemple en précisant la couleur de chaque nœud. Vérifier que c'est bien un arbre AVL.
- Écrire les algorithmes (ou programmes python) recherchant un élément dans un arbre 2-3-4 pour les deux représentations.

10.5.2 - Ajout :

L'ajout d'un élément dans un arbre 2-3-4 se fait toujours dans une feuille. Lorsque la feuille ne contient qu'un ou deux éléments, l'ajout est très facile. Dans le cas contraire (trois éléments), il faut la faire éclater. Il existe deux méthodes : l'une avec éclatement à la remontée et l'autre avec éclatement à la descente.




Éclatement à la remontée :
Le principe consiste à faire éclater le nœud en faisant remonter l''élément médian. Cette technique peut provoquer des éclatements en cascade.

Éclatement à la descente :
Dans la phase de descente, on éclate tous les nœuds ayant 3 éléments sur le chemin allant de la racine à la feuille où le nouvel élément sera inséré.

Avantages de l'éclatement à la descente :

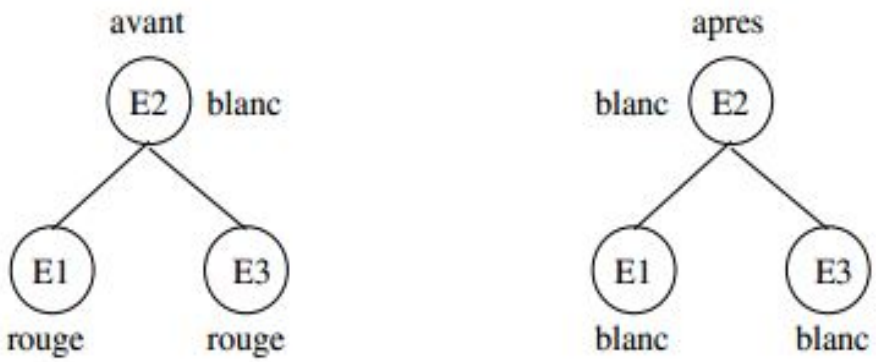
- ajout relativement simple,
- version itérative sans pile ;
- accès concurrents simplifiés.

Inconvénients :

- taux de remplissage plus faible
- et donc profondeur plus importante.

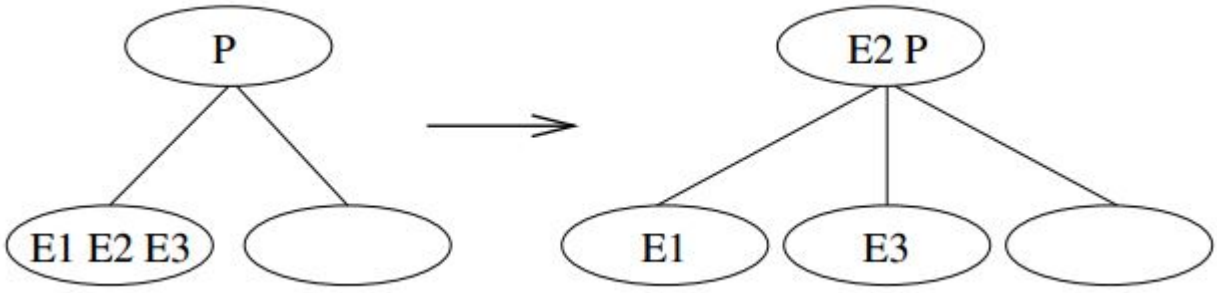
Étude de l'éclatement d'un nœud :

On se place dans le contexte où l'arbre 2-3-4 est représenté par un ABR bicolore. On y étudie l'éclatement d'un nœud possédant 3 éléments.

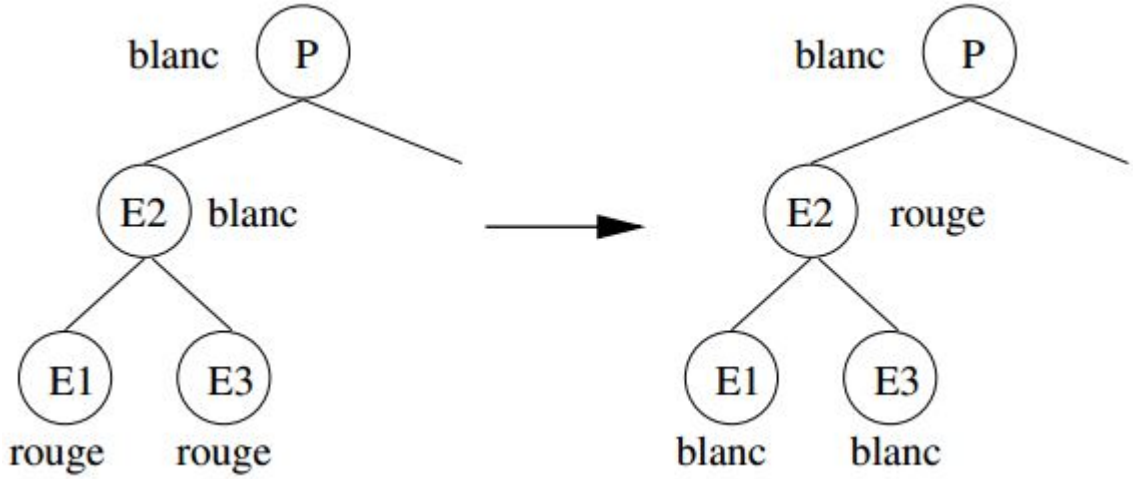



Comme le montre ce dessin seule la couleur change mais ceci risque de faire apparaître deux nœuds rouges consécutifs. La hauteur de l'arbre ne serait plus logarithmique. Pour y remédier, on utilise des rotations.

**Cas où le père possède un seul élément :**
Éclatement du nœud :



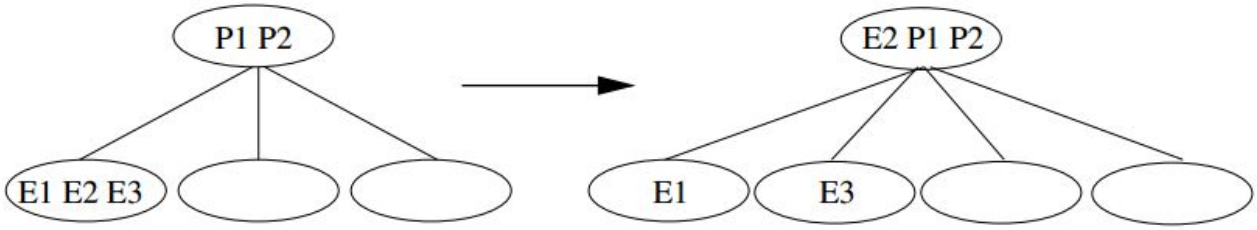
Au niveau de sa représentation :



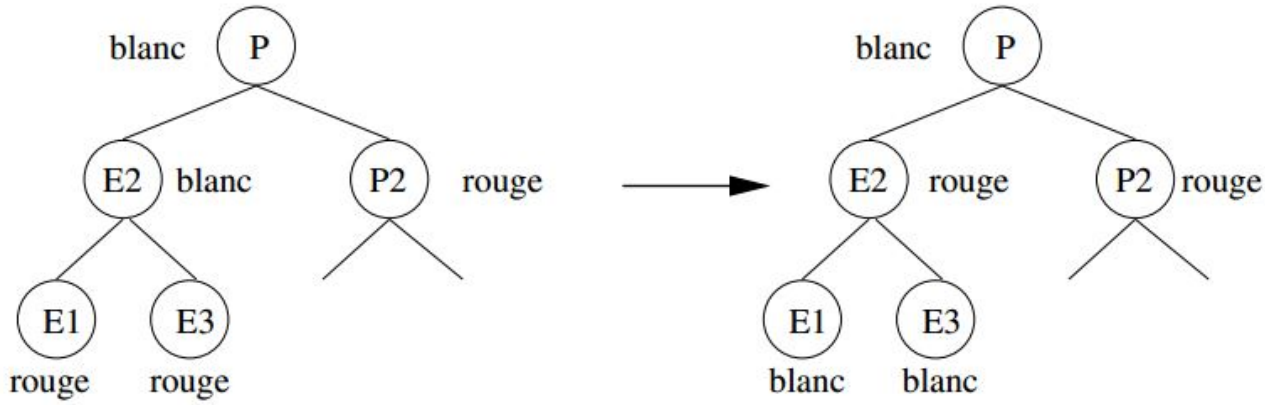

Aucun problème ici. Le cas de l'éclatement du nœud fils droit est analogue : absence de 2 nœuds rouges consécutifs.

**Cas où le père a 2 éléments :**

<u>Premier cas : éclatement du nœud de droite :</u>

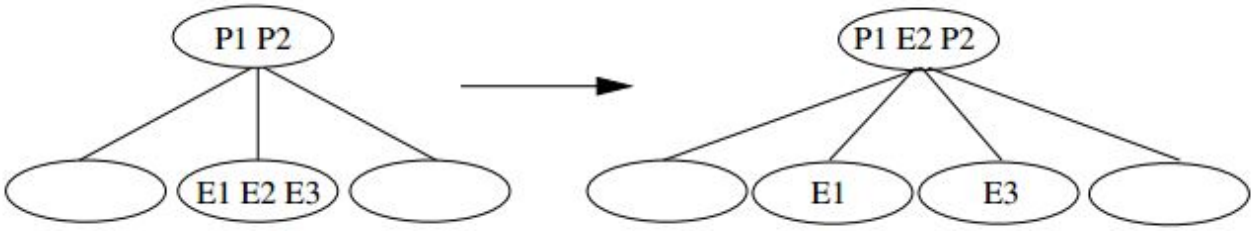



Représentation :

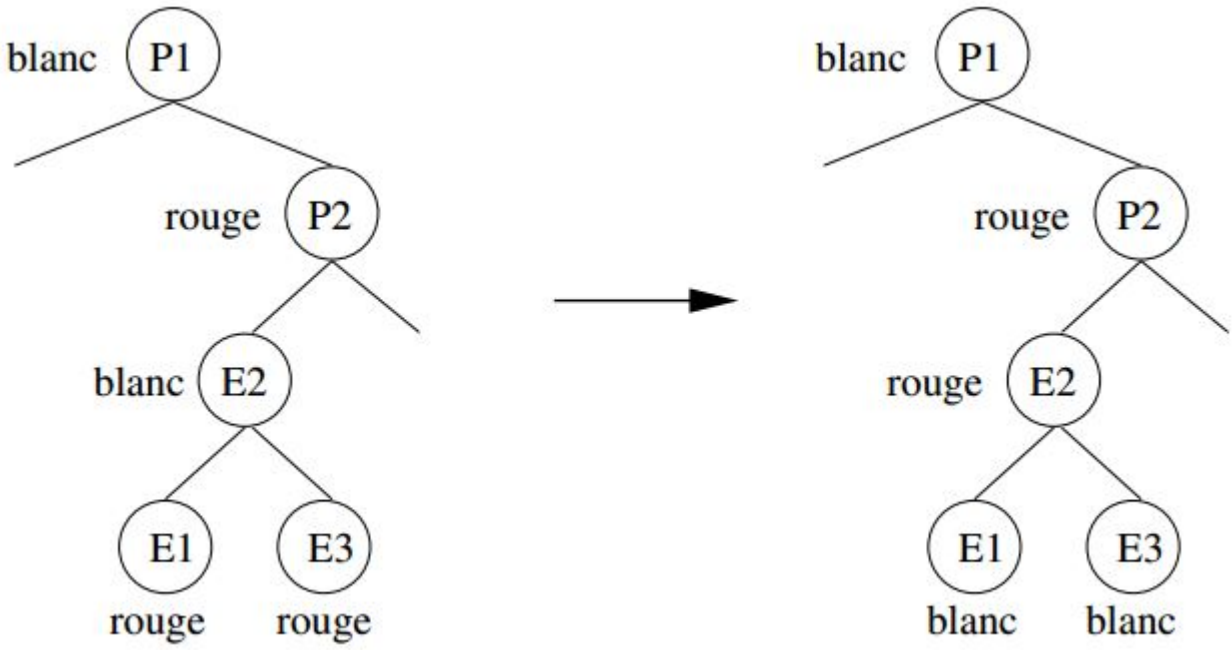


Donc ici aussi, il n'y a pas de problèmes.

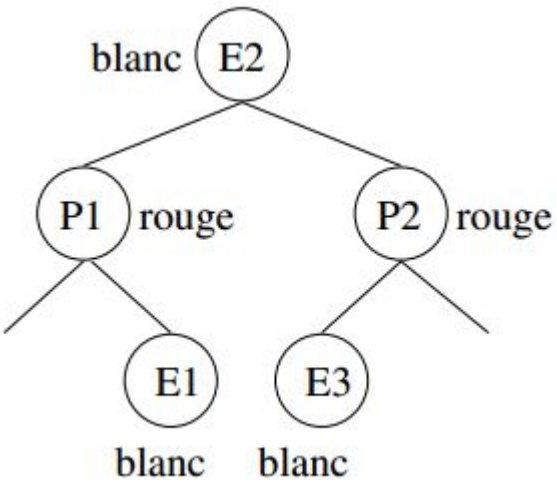
Deuxième cas : éclatement du nœud du milieu :




Représentation :

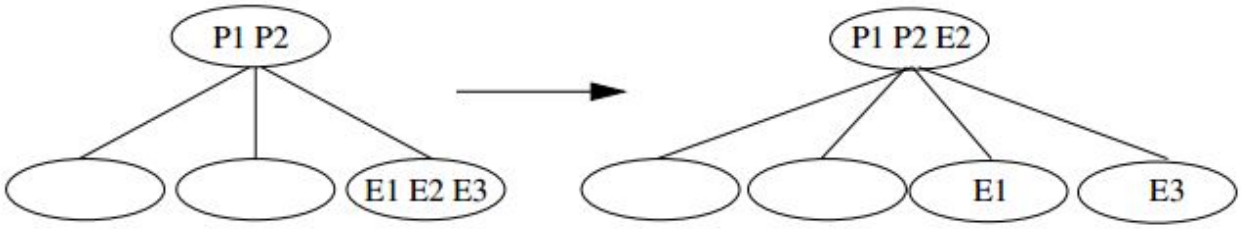



Problème : apparition de deux nœuds rouges consécutifs : on applique une rotation droite gauche en P1.

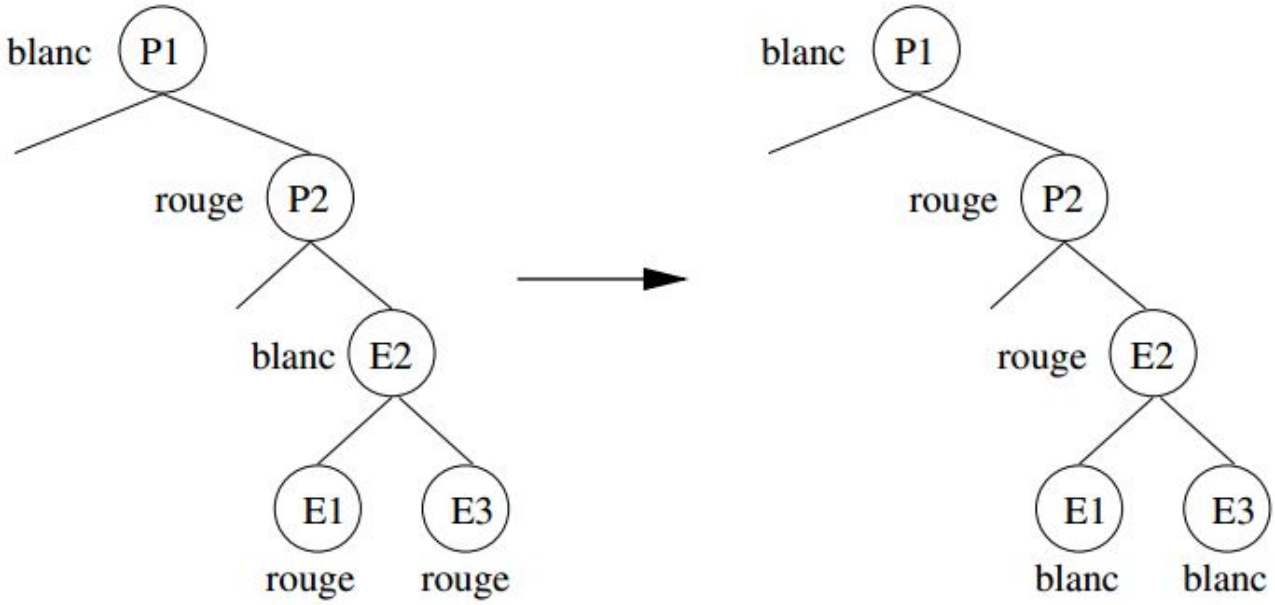



L'arbre est maintenant rééquilibré.

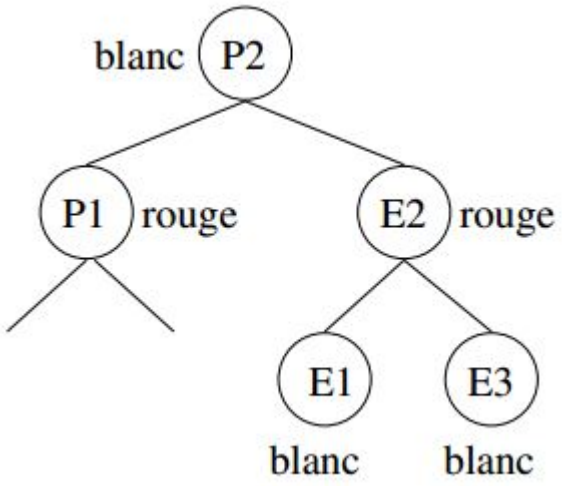
<u>Troisième cas : le nœud à éclater est à droite :</u>




Représentation :



Problème : apparition de deux nœuds rouges consécutifs : on applique une rotation gauche en P1.




L'arbre est maintenant rééquilibré.

**Cas où le père possède trois éléments :**

Ce cas de figure est impossible avec l'algorithme d'éclatement à la descente. Il peut se rencontrer avec l'autre algorithme : on applique la même technique au père.

L'algorithme ne présente pas de difficultés mais comporte de nombreux cas particuliers.

Suppression d'un élément :

La suppression d'un élément peut poser problème et amener à restructurer l'arbre. La suppression dans un arbre 2-3-4 utilise la même technique que pour la suppression dans un B-arbre étudiée dans la partie suivante.

## 10.6 _ Les B-arbres :

### 10.6.1 - Définition :

Les B-arbres généralisent les arbres 2-3-4 qui n'en sont d'ailleurs pas. Ils sont souvent utilisés pour structurer des fichiers indexés dans les SGBD. Un B-arbre de degré m est un arbre général de recherche possédant les propriétés suivantes :

- toutes les feuilles sont au même niveau
- tout nœud possède au plus 2m éléments
- tout nœud à l'exception de la racine possède au moins m éléments

En général m est compris entre 20 et 500.

Question : nombres minimal et maximum d'éléments dans un B-arbre en fonction de m et de la profondeur p.

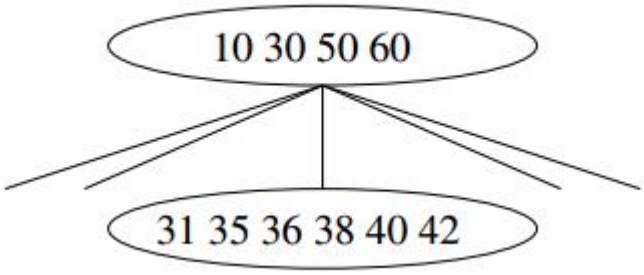
Représentations internes : il existe plusieurs représentations internes possibles : généralisation de celle des arbres 2-3-4, arbres binaires, ...

Recherche d'un élément : une fois la représentation choisie, cet algorithme ne pose pas de problèmes. L'écrire en exercice avec la généralisation aux B-arbres de la représentation interne des arbres 2-3-4.

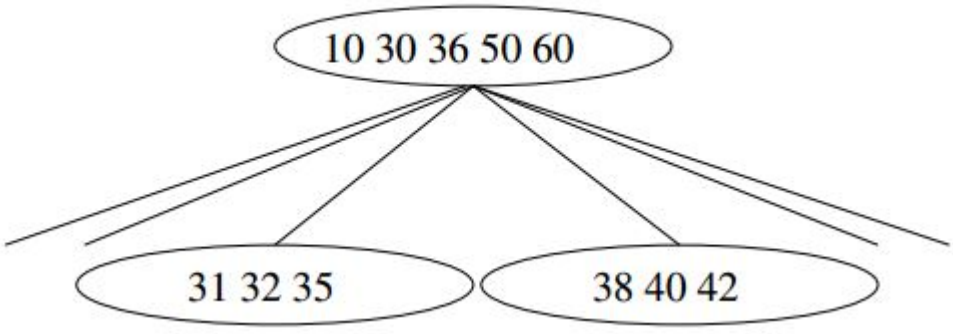
### 10.6.3 - Ajout :

Le principe est identique à celui de l'ajout d'un élément dans un arbre 2-3-4. En cas de problème (feuille contenant déjà 2m éléments), on retrouve l'algorithme avec éclatement(s) à la remontée ou plus souvent à la descente afin de mieux gérer les accès concurrents (SGBD).

L'éclatement d'un nœud fait remonter sa valeur médiane. Exemple avec m = 3 :




L'ajout de 32 provoque la remontée de 36 :




10.6.4 - Suppression :

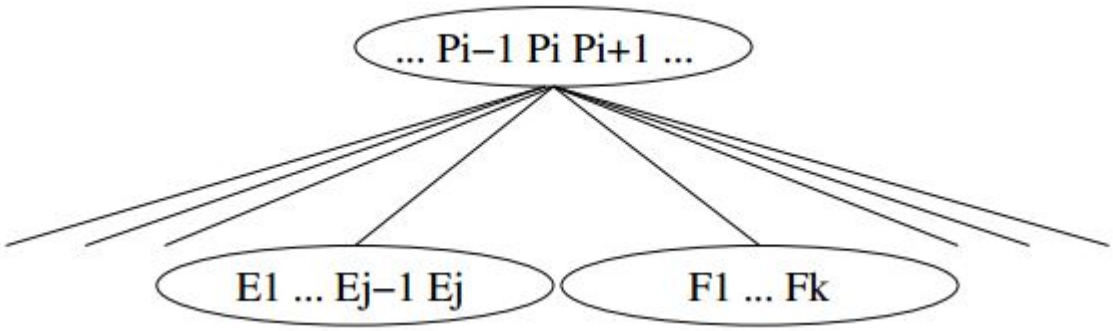
Afin de ne pas avoir à restructurer trop lourdement l'arbre, on se ramène toujours à supprimer un élément dans une feuille. Quand l'élément à supprimer n'est pas contenu dans une feuille, on le remplace par l'élément assurant la

séparation avec le sous-arbre voisin. Cet élément peut, par exemple, être le dernier élément de la feuille la plus à droite du sous-arbre à gauche de l'élément à supprimer. Si la feuille contenait exactement m élément, sauf si c'est la racine, il faut rééquilibrer le B-arbre. Pour ce faire, on utilise les opérations de répartition et ou de regroupement.

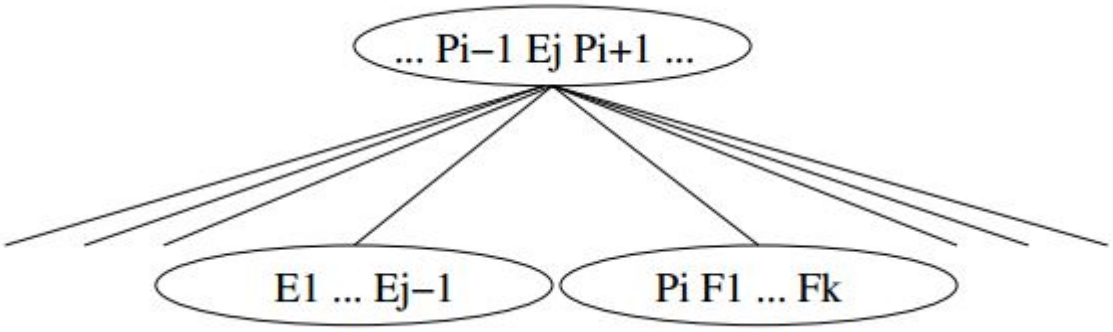
Répartition de gauche à droite :

L'idée consiste à faire passer un élément d'une feuille à sa voisine de droite :

Situation initiale :



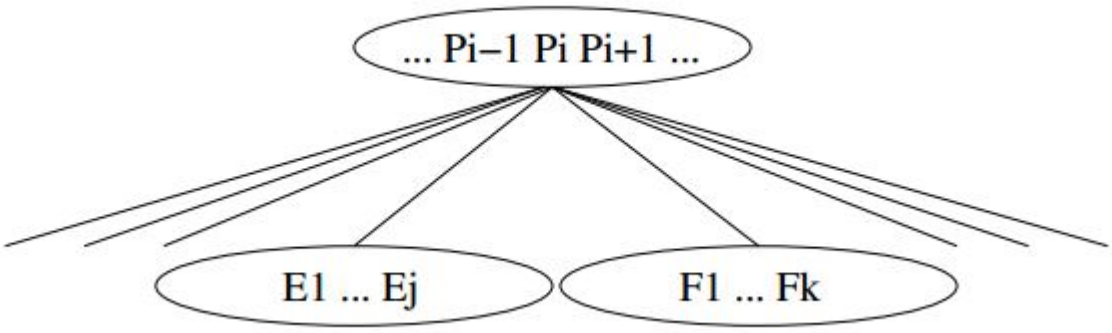

Situation après répartition :



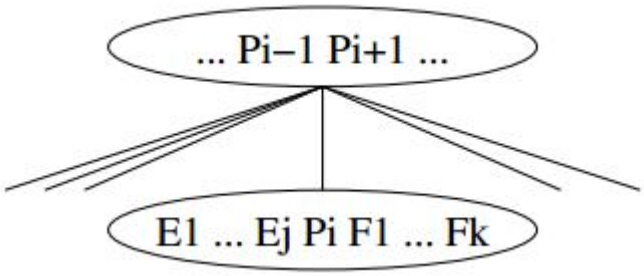

La répartition de droite à gauche est analogue (inverse).

Regroupement :

Situation initiale :




Situation après le regroupement :




Principe du rééquilibrage :

Rappelons que ce rééquilibrage n'est fait que si la feuille où a eu lieu la suppression physique n'a plus que m-1 éléments.

- Si l'une des deux feuilles voisines a au moins m+1 éléments, on réalise une répartition de gauche à droite ou de droite à gauche.
- Si les deux feuilles voisines possèdent m éléments chacune, on réalise alors un regroupement avec l'une

d'entre elles. Si nécessaire, on traite le père de la même manière (répartition, regroupement). On procède ainsi jusqu'à la racine si nécessaire.

- En cas de regroupement sous la racine et si la racine ne possédait qu'un seul élément, le nouveau nœud devient la racine du B-arbre.

## 10.7 - Les arbres de classification :

Plusieurs méthodes d'analyse de données et ou de data mining produisent des arbres comme par exemple les arbres de décisions ou les arbres résultats de CAH (Classification Ascendante Hiérarchique). Nous allons ici surtout traiter les arbres issus de CAH. Le but de la CAH consiste à trouver une suite de partitions emboîtées d'un ensemble E de N éléments. Les feuilles de l'arbre sont les éléments de E et les nœuds sont des classes constituées des feuilles sous les nœuds. Ces arbres sont produits avec des méthodes utilisant des distances (souvent basées sur l'inertie), des similarités ou des probabilités (voir cours d'analyse des données).

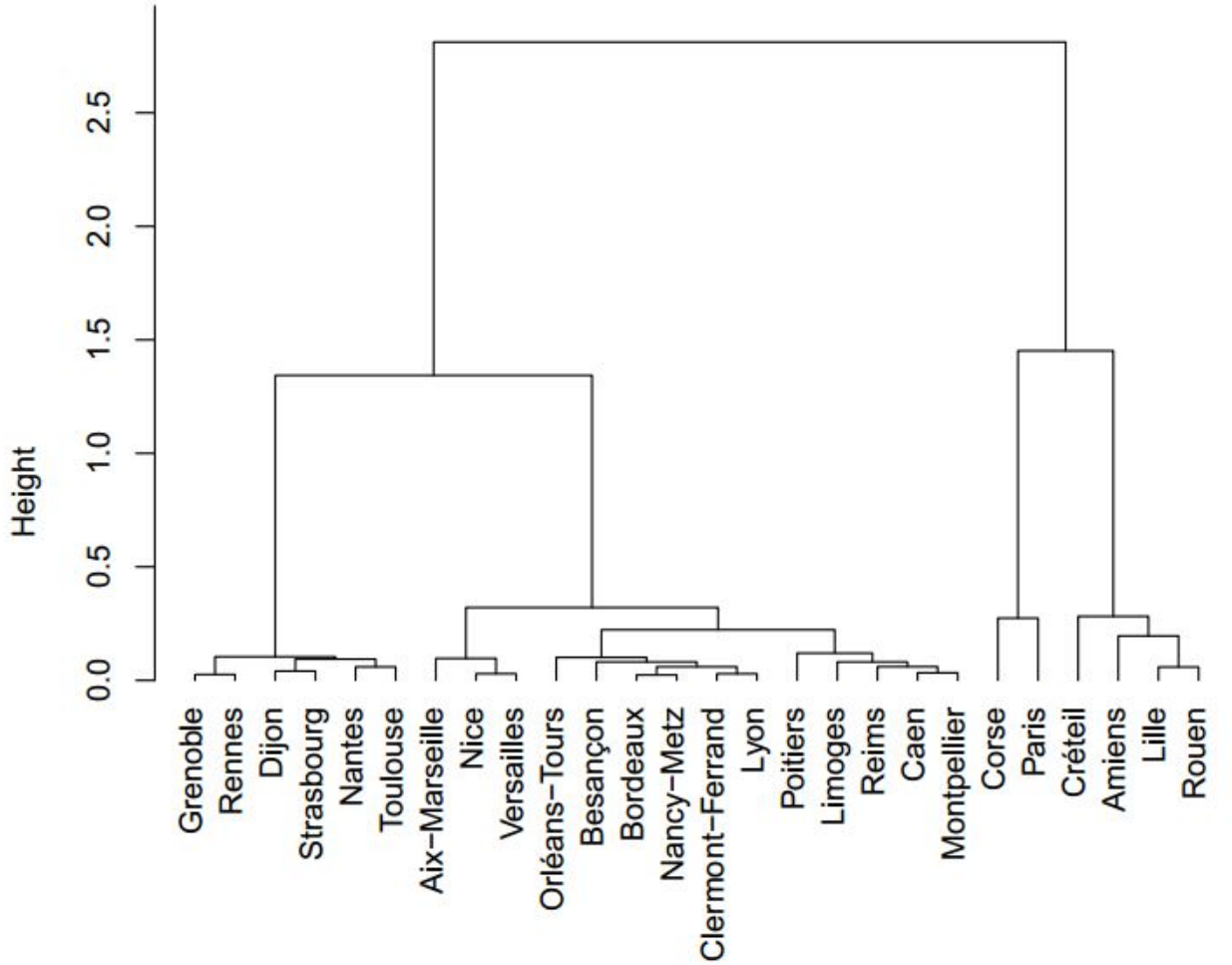
Exemple :

L'arbre a été calculé par le logiciel R sur les pourcentages de réussite au Bac 2002 par série et par académie.

| | L | ES | S | STIL | SMS | STT | BTH | STA |
|---|---|---|---|---|---|---|---|---|
| 1 - Aix-Marseille | 81.2 | 78.7 | 78.7 | 71.5 | 73.6 | 80.0 | 84.2 | 69.2 |
| 2 - Amiens | 77.7 | 74.7 | 73.2 | 71.7 | 66.8 | 70.3 | 98.6 | 77.5 |
| 3 - Besançon | 82.8 | 78.9 | 80.8 | 80.4 | 85.9 | 76.3 | 97.3 | 78.8 |
| 4 - Bordeaux | 83.2 | 80.7 | 79.0 | 77.9 | 86.6 | 76.7 | 86.0 | 76.7 |
| 5 - Caen | 80.0 | 77.8 | 78.9 | 73.7 | 86.4 | 79.4 | 94.2 | 82.7 |
| 6 - Clermont-Ferrand | 82.1 | 81.9 | 82.5 | 77.1 | 86.2 | 80.3 | 89.5 | 76.7 |
| 7 - Corse | 85.9 | 81.8 | 73.7 | 72.6 | 79.9 | 78.4 | 70.4 | 16.7 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 - Créteil | 74.6 | 71.1 | 71.1 | 67.4 | 66.8 | 67.6 | 83.0 | 75.3 |
| 9 - Dijon | 84.0 | 82.8 | 83.1 | 78.9 | 85.5 | 80.9 | 97.5 | 77.6 |
| 10 - Grenoble | 87.2 | 85.8 | 85.0 | 83.2 | 86.9 | 83.6 | 91.1 | 84.5 |
| 11 - Lille | 78.2 | 76.5 | 79.8 | 67.3 | 75.6 | 74.5 | 87.3 | 75.8 |
| 12 - Limoges | 81.3 | 76.3 | 80.1 | 72.5 | 87.5 | 74.6 | 90.6 | 66.5 |
| 13 - Lyon | 83.0 | 81.1 | 82.6 | 78.2 | 83.4 | 76.0 | 91.4 | 81.1 |
| 14 - Montpellier | 83.2 | 78.6 | 79.7 | 73.9 | 82.3 | 80.8 | 93.2 | 79.3 |
| 15 - Nancy-Metz | 82.8 | 79.0 | 80.1 | 75.0 | 87.3 | 75.7 | 90.0 | 75.5 |
| 16 - Nantes | 84.4 | 82.6 | 83.4 | 83.8 | 89.4 | 83.7 | 91.3 | 83.7 |
| 17 - Nice | 82.3 | 80.2 | 79.5 | 72.9 | 76.2 | 75.2 | 88.5 | 69.4 |
| 18 - Orléans-Tours | 79.5 | 78.2 | 81.4 | 77.2 | 84.6 | 79.7 | 84.2 | 78.7 |
| 19 - Paris | 80.3 | 77.9 | 79.4 | 74.1 | 77.6 | 71.4 | 81.6 | 0.0 |
| 20 - Poitiers | 81.1 | 81.2 | 80.0 | 74.9 | 88.9 | 84.0 | 94.3 | 72.5 |
| 21 - Reims | 83.8 | 76.9 | 76.9 | 71.1 | 81.3 | 77.9 | 95.2 | 69.6 |
| 22 - Rennes | 89.0 | 85.9 | 84.4 | 81.2 | 87.2 | 84.5 | 96.2 | 82.0 |
| 23 - Rouen | 79.5 | 73.4 | 75.9 | 66.6 | 73.1 | 74.5 | 91.7 | 82.8 |
| 24 - Strasbourg | 87.2 | 83.9 | 85.0 | 78.0 | 88.1 | 81.1 | 93.6 | 81.2 |
| 25 - Toulouse | 85.1 | 84.1 | 83.0 | 81.0 | 83.0 | 81.6 | 85.5 | 72.8 |
| 26 - Versailles | 82.9 | 79.8 | 80.7 | 71.5 | 77.5 | 73.0 | 91.6 | 85.5 |

# Arbre :



Remarques :

- ces arbres ne sont pas forcément binaires en théorie mais les algorithmes des principaux logiciels, dont R, produisent des arbres toujours binaires ;
- pour agréger n éléments en une seule classe, il y a exactement n-1 agrégations binaires ; par conséquent, pour représenter cet arbre il suffit de deux tableaux de n-1 éléments (ou d'un tableau de structures) ;
- la hauteur (height) est la distance entre les deux classes agrégées ;
- pour obtenir une partition, il suffit de couper l'arbre au niveau choisi. Des indices statistiques permettent de sélectionner les meilleurs niveaux pour couper l'arbre ;
- dans le contexte de la recherche d'une partition, on peut échanger les sous arbres gauche et droit.

Représentation aîné - benjamin :

Cette représentation est la représentation classique fils gauche, fils droit. Elle ne permet que la représentation des arbres binaires. Il en existe deux variantes.

Logiciel R : voici la représentation de l'arbre donné en exemple :

```
        FG   FD          FG   FD          FG   FD
 [1,]   -4  -15   [2,]  -10  -22   [3,]   -6  -13
 [4,]  -17  -26   [5,]   -5  -14   [6,]   -9  -24
 [7,]  -11  -23   [8,]  -16  -25   [9,]    1    3
[10,]  -21    5  [11,]  -12   10  [12,]   -3    9
```

```
[13,]    6    8      [14,]   -1    4      [15,]  -18   12
[16,]    2   13      [17,]  -20   11      [18,]   -2    7
[19,]   15   17      [20,]   -7  -19      [21,]   -8   18
[22,]   14   19      [23,]   16   22      [24,]   20   21
[25,]   23   24
```

Les nombres entre accolades sont les numéros des classes numérotées de 1 à n-1 où n est le nombre d'éléments ; les classes sont triées par distances d'agrégation croissantes, non données ici. Les nombres négatifs sont les numéros des individus et les nombres positifs (colonnes 2 et 3) sont les numéros des classes. Par exemple [1,]  -4  -15 signifie que la classe 1 est constituée des éléments 4 et 15 (Bordeaux et Nancy-Metz) ou encore [10,]  -21    5 signifie que la classe 10 est constituée de l'élément 21 (Reims) et de la classe 5. La dernière classe (25) est constituée de tout l'ensemble E.

<u>Variante :</u>

Le principe est le même mais les agrégations sont numérotées de n+1 à 2n-1, le nombre i ($1 \leq i \leq n$) représente l'élément i. Dans cette représentation, il n'y a pas de nombres négatifs. Dans l'exemple, avec n = 26, le codage de l'arbre est :

```
      FG   FD          FG   FD          FG   FD
 27    4   15    28   10   22    29    6   13
 30   17   26    31    5   14    32    9   24
 33   11   23    34   16   25    35   27   29
 36   21   31    37   12   36    38    3   35
 39   32   34    40    1   30    41   18   38
 42   28   39    43   20   37    44    2   33
 45   41   43    46    7   19    47    8   46
 48   40   45    49   42   48    50   46   47
 51   49   50
```

Représentation polonaise inverse :

La représentation polonaise inverse est une représentation préfixée à ne pas confondre avec le parcours préfixé d'un arbre. De façon analogue, on peut définir une représentation suffixée. Ces représentations nécessitent un tableau de 2N-1 éléments, ce qui est équivalent à la représentation aîné-benjamin. Par contre, elle nécessite, tout au moins en CAH, de nombreux transferts de zones de tableaux ce qui est une perte de temps. La représentation polonaise inverse permet de coder des arbres non binaires, ce que ne permettent pas les représentations aîné-benjamin.

Principe :
La représentation polonaise inverse est constituée de 2n-1 entiers, que l'arbre soit binaire ou non. Les entiers positifs sont les numéros des éléments et les entiers négatifs (de -1 à 1-N) sont les numéros des agrégations (des nœuds). Après chaque nombre négatif, on a son sous arbre gauche puis son sous arbre droit.

Exemple :

```
-25 -23 -16  -2  10  22 -13  -6   9  24
 -8  16  25 -22 -14   1  -4  17  26 -19
-15  18 -12   3  -9  -1   4  15  -3   6
 13 -17  20 -11  12 -10  21  -5   5  14
-24 -20   7  19 -21   8 -18   2  -7  11
 23
```

Agrégations multiples :

La représentation polonaise inverse permet de coder des agrégations multiples. Par exemple pour exprimer que les individus 10, 40 et 50 s'agrègent au niveau 5 : -5 -5 10 40 50 ce qui est équivalent à -5 10 -5 40 50.

D'une manière générale si p classes $c_1$, ..., $c_i$, ..., $c_p$ s'agrègent au niveau n, la représentation sera : -n ... -n $c_1$, ..., $c_i$, ..., $c_p$ avec p-1 fois -n au début et en supposant que les classes $c_i$ sont déjà en représentation polonaise inverse.

## 11 - LES GRAPHES

Nous nous limitons ici aux structures de données permettant de représenter les graphes. Les graphes et leurs algorithmes sont étudiés de façon mathématique dans un autre cours : celui de théorie des graphes ou de recherche opérationnelle. Un graphe G est constitué d'un ensemble X de sommets et d'un ensemble U d'arcs (cas orienté) ou d'arêtes (cas non orienté) : G = (X,U). On considère par la suite que les sommets sont numérotés de 0 à n-1 où n est le nombre de sommets.

### 11.1 - Matrice booléenne :

Le graphe est représenté par une matrice carrée n x n. Soit M cette matrice :

- M[i][j] = vrai si et seulement si $(x_i,x_j) \in U$

- M[i][j] = faux sinon

Dans le cas d'un graphe valué, on remplace la valeur booléenne par la valeur de l'arc (+∞ quand il n'y a pas d'arc). Question : proposer une méthode de stockage sans redondance pour les graphes non orientés.

Avantages :
- accès rapide aux successeurs d'un sommet
- accès rapide aux prédécesseurs
- test rapide de l'existence d'un arc
- mise à jour rapide de l'ensemble des arcs
- stockage sur fichier

Inconvénients :
- mal adapté à la mise à jour des sommets
- matrice souvent creuse
- perte de place dans le cas non orienté

## 11.2 - Dictionnaire des arcs :

On stocke la liste des arcs. Un arc est représenté dans le cas non valué par un couple (sommet origine, sommet extrémité) et dans le cas d'un graphe valué par un triplet (sommet origine, sommet extrémité, valeur). cette liste peut être triée ou non. Elle est représentée dans un tableau ou avec une liste.. Toutes les opérations de recherche dans une telle structure sont longues et coûteuses. Les opérations de mise à jour sont rapides mais doivent souvent être précédées d'une recherche.

Avantages :

- stockage possible sur fichier
- recherche rapide si trié sur le critère de recherche

Inconvénients :

- recherche longue si le dictionnaire n'est pas trié suivant le critère de recherche
- mise à jour longue
- test d'existence d'un arc assez long
- problème pour les sommets isolés

## 9.3 - Dictionnaire des successeurs ou des prédécesseurs :

On dispose de la liste des successeurs (ou des prédécesseurs) de chaque sommet. Ce dictionnaire est souvent représenté par un tableau de listes. Chaque liste contenant les successeurs (ou prédécesseurs) d'un sommet donné. Chaque enregistrement d'une liste contient le numéro du sommet extrémité (ou origine pour le dictionnaire des prédécesseurs) et la valeur de l'arc dans le cas valué.

Avantages :

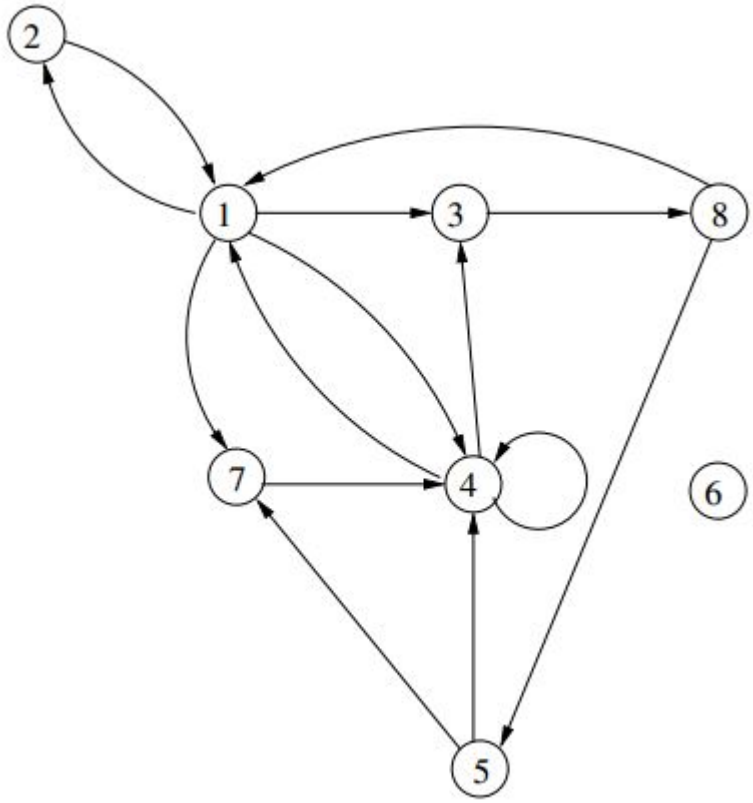
- accès rapide aux successeurs (ou aux prédécesseurs)
- test d'existence d'un arc assez rapidement
- pas de perte de place
- mise à jour assez rapide de l'ensemble des arcs si connaissance de l'origine.

Inconvénients :

- accès long aux prédécesseurs (ou aux successeurs)
- mise à jour pénible de l'ensemble des sommets
- peu adapté à un stockage sur fichier

## 11.4 - Exemple :

Soit le graphe suivant :




Dictionnaire des arcs :

| | | | | | | |
|---|---|---|---|---|---|---|
| (1,2) | (1,3) | (1,4) | (1,7) | (2,1) | (3,8) | (4,1) |
| (4,3) | (4,4) | (5,4) | (5,7) | (7,4) | (8,1) | (8,5) |

Matrice booléenne :

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | | V | V | V | | | V | |
| 2 | V | | | | | | | |
| 3 | | | | | | | | V |
| 4 | V | | V | V | | | | |
| 5 | | | | V | | | V | |
| 6 | | | | | | | | |
| 7 | | | | V | | | | |
| 8 | V | | | | V | | | |

Dictionnaire des prédécesseurs sous la forme d'une liste de listes :
[[2,4,8], [1], [1,4], [1,4,5,7], [8], [], [1,5], [3]]

## 12. LES AUTOMATES FINIS

Les automates finis ont été définis en théorie des langages et en compilation. Ils reconnaissent des langages assez simples : ceux qui peuvent être formalisés par des expressions rationnelles aussi appelées expressions régulières. C'est le cas par exemple des identificateurs, des mots clés d'un langage (analyse lexicale). Par contre, ils ne sont pas assez puissants pour décrire un langage de programmation tel que C ou JAVA. Ici nous ne traiterons pas des aspects théorie des langages et compilation mais nous verrons que ces automates sont utiles en algorithmique.

## 12.1 - Les expressions rationnelles :

Définition d'une expression régulière ou rationnelle :
Soit V un vocabulaire fini et non vide de symboles
$V = \{a_1,...,a_n\}$, on introduit un vocabulaire auxiliaire
$V' = \{*, +, (, ), \varnothing\}$

On appelle alors expression régulière sur V toute chaîne de $V \cup V'$ répondant aux conditions suivantes :

- tout symbole de V est une expression régulière sur V (de même $\varnothing$),
- si $\alpha$ et $\beta$ sont deux expressions régulières sur V alors : $\alpha \mid \beta$, $\alpha\ \beta$ , $\alpha^*$, $\alpha^+$ et $(\alpha)$ sont aussi des expressions régulières sur V,
- toutes les expressions régulières sur V sont obtenues à partir d'un nombre fini d'applications des deux règles précédentes.

Exemple : les expressions arithmétiques :

V = {nb, +, -, *, /, #} où nb représente un nombre. Une expression régulière décrivant notre langage est : nb ((+|-|*|/) nb)*

Dans la pratique, le vocabulaire auxiliaire est enrichi aux métacaractères de Lex qui est un générateur d'analyseurs lexicaux à partir d'expressions rationnelles. En voici quelques exemples :

- point (.) : n'importe quel caractères excepté le retour-chariot,

- \n : caractère retour-chariot,
- - : délimiteur d'intervalle, par exemple, une lettre sera reconnue par [a-zA-Z] et un entier par [0-9],
- ? : zéro ou une occurrence,
- ...

Les langages qui peuvent être décrits par une expression rationnelle sont appelés langages réguliers ou rationnels.

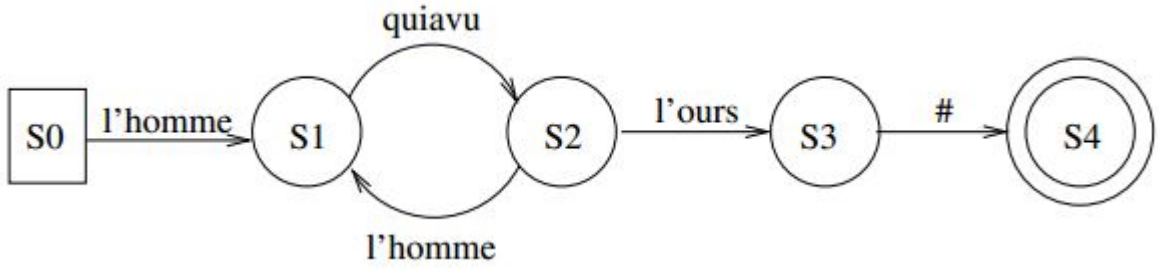
## 12.2 - Les automates :

Un automate d'états fini est un graphe orienté dont les sommets sont appelés états et dont les arcs sont étiquetés par des symboles du vocabulaire V et sont appelés transitions. Il existe un sommet initial unique (souvent numéroté état 0) et un ou plusieurs états finals.

Une chaîne donnée appartient au langage considéré si et seulement si, en la parcourant entièrement, il existe un chemin dans ce graphe entre l'état initial et un état final.

Exemple :
Soit l'expression régulière **(l'homme quiavu)$^+$ l'ours** #} sur le vocabulaire V = {l'homme, quiavu, l'ours, #}

On peut représenter son automate par un graphe :




ou par sa table de transitions :

| | l'homme | quiavu | l'ours | # |
|---|---|---|---|---|
| S0 | 1 | -1 | -1 | -1 |
| S1 | -1 | 2 | -1 | -1 |
| S2 | 1 | -1 | 3 | -1 |
| S3 | -1 | -& | -1 | 4 |
| S4 | -1 | -1 | -1 | -1 |

Plusieurs algorithmes existent pour produire automatiquement un automate fini à partir d'une expression rationnelle.
L'automate peut être déterministe ou non déterministe. Un automate est non déterministe si on y trouve une transition avec la chaîne vide ε ou si plusieurs états sont accessibles à partir du même état avec le même symbole. Il existe des algorithmes pour rendre déterministe n'importe quel automate fini. Dans la suite, nous ne considérerons que des automates déterministes.

Algorithme général pour un automate déterministe :

On suppose définies les fonctions suivantes :

- liresymb() : fonction de lecture du prochain symbole,
- transit(état, symb) : fonction entière correspondant à la table des transitions,
- final (état) : fonction booléenne qui indique si un état est final ou non.

On suppose également que l'état initial est l'état 0.

```
def automate()
    etat = 0 # état initial
    while (etat!= -1) and
                        (not final(etat)) :
        symb = liresymb()
        etat = transit(etat, symb)
    if final(etat) :
        return True
    else :
        return False
```

En théorie, avant de conclure à l'appartenance de la chaîne au langage, il faudrait s'assurer d'avoir parcouru entièrement la chaîne. En pratique le dernier symbole noté dièse (#) représente souvent la marque de fin de fichier ou la marque de fin de ligne.

Actions associées à l'automate :

On peut associer des actions aux transitions afin de réaliser certains calculs. Par exemple, dans l'histoire de l'ours, on veut déterminer le nombre de personnes intermédiaires

pour savoir quel crédit apporter à cette histoire.

Actions à réaliser :

- au début : initialiser à 0 nbint soit entre S0 et S1, soit en initialisation,
- quand on trouve un nouvel intermédiaire : incrémenter nbint entre S2 et S1,
- à la fin : afficher nbint soit entre S3 et S4, soit en traitement final.

On va numéroter les actions correspondant aux différentes transitions et les reporter dans une table des actions :

| | l'homme | quiavu | lours | # |
|---|---|---|---|---|
| S0 | 1 | -1 | -1 | -1 |
| S1 | -1 | 0 | -1 | -1 |
| S2 | 2 | -1 | 0 | -1 |
| S3 | -1 | -1 | -1 | 3 |
| S4 | -1 | -1 | -1 | -1 |

On peut ensuite écrire un sous programme regroupant les différentes actions :

```
def liste_actions(numaction) :
  if numaction == 1 :
    nbint = 0
  elif numaction == 2 :
```

```
    nbint =  nbint+1
  elif numaction == 3 :
    print ("Nombre d'intermédiaires : ",
           nbint)
  # sinon on ne fait rien
```

Pour intégrer les actions à l'algorithme général de l'automate, on insère *liste_actions(action(etat,symb))* avant le changement d'état. La fonction *action* correspond à la table des actions.

Application à l'algorithmique :

Même si les automates finis ont été conçus dans le cadre de la théorie des langages et sont utilisés pour l'analyse lexicale en compilation, ils peuvent traiter de nombreux problèmes algorithmiques. Pour cela, il suffit de pouvoir modéliser le problème par un automate sans nécessairement passer par une expression rationnelle. Dans ce cadre, l'aspect analyse syntaxique (appartenance au langage) peut être secondaire par rapport à l'aspect calculs.

# BIBLIOGRAPHIE

Il existe de nombreux livres et sites internet traitant d'algorithmie et de structures de données. Ici, sont données quelques références de livres présents à la B.U. technologie.

C. CARREZ - Des structures aux bases de données - Dunod Informatique, 1990.

J. COURTIN et I. KOWARSKI - Initiation à l'algorithmique et aux structures de données - volumes 1, 2 et 3 - Dunod Informatique, 1990.

M.C. GAUDEL, M. SORIA et C. FROIDEVAUX - Types de données et algorithmes - volumes 1 et 2 - Collection didactique INRIA, 1988.

D.E. KNUTH - The art of computer programming : sorting and searching - Addison-Wesley, 1973.