# Travaux dirigés d'algorithmique.
# DEUG MIAS 2ème année

Loïc Yon

2ème semestre 2000

## Table des matières

# 1 Calcul du PGCD de deux nombres

## 1.1 Algorithme récursif simple

### 1.1.1 Justification

On ne reviendra pas sur les propriétés du PGCD vues et revues en mathématiques.

### 1.1.2 Algorithme récursif

```
PGCD(A,B)
DEBUT
    SI (A<B) ALORS ECHANGER(A, B) FSI
    SI (B=1) OU (A=B) ALORS RETOURNER B FSI

    RETOURNER PGCD(A-B,B)
FIN
```

**Remarque** La procédure ECHANGER effectue trois opérations élémentaires (trois affectations pour un échange).

### 1.1.3 Correction et finitude

Cet algorithme est **correct** (i.e. donne le bon résultat) car il s'appuie sur des formules mathématiques.

Cet algorithme est **fini** car **si** la condition d'ordre sur A et B est respectée, A-B est un compteur positif entier décroissant vers 0. (Quand $A - B = 0$, on a $A = B$ donc l'algorithme s'arrête)

### 1.1.4 Complexité

Pour simplifier, on peut définir la complexité d'un algorithme récursif comme le nombre d'appel à la fonction récursive elle-même.
Dans le pire des cas, B vaut 2, A s'écrit $2k$ ou $2k-1$, ce qui va donner $k$ appels donc la complexité de l'algorithme est en $O(A)$.

A=6, B=2, PGCD(6,2)=PGCD(4,2)=PGCD(2,2)=2
A=7, B=2, PGCD(7,2)=PGCD(5,2)=PGCD(3,2)=PGCD(1,2)=1

Par suite, si $B \leq A-2$, on enlève au moins 2 à A et si $B = A-1$, l'algorithme s'arrête à l'itération suivante.

## 1.2 Algorithme récursif d'Euclide

### 1.2.1 Algorithme récursif

```
EUCL(A,B)
DEBUT
    SI (B=0) ALORS RETOURNER A FSI

    RETOURNER EUCL(B, A mod B)
FIN
```

**Remarque 1** Cet algorithme est bien plus rapide que le premier. En effet, on peut écrire A mod B = A - B.Q avec Q entier. Alors il faut Q appels de PGCD(A-B, B) pour faire UN appel de EUCL(B, A mod B).

**Remarque 2** Il n'est pas nécessaire d'ordonner les nombres A et B. Si A est plus petit que B, un appel supplémentaire d'EUCL inverse A et B.

#### 1.2.2 Nombres de Fibonacci

On applique l'algorithme d'Euclide aux nombres suivants : $A = F_i$ et $B = F_{i-1}$ pour $i \geq 1$. On remarque que A mod B vaut $F_{i-2}$ si $i \geq 2$.

$$EUCL(F_i, F_{i-1}) = EUCL(F_{i-1}, F_{i-2}) = \cdots = EUCL(F_1, F_0) = 1$$

Ce qui donne a priori i appels. En effet, la cascade d'appels ne s'arrête que lorsque B est égal à 0 et qui est obtenu avec $F_0$.

En fait, dans la pratique, on donne à l'algorithme deux nombres consécutifs de la suite de Fibonacci. Prenons un exemple avec $F_5 = 5$ et $F_6 = 8$ issus de la suite ( 0, 1, 1, 2, 3, 5, 8). Alors on a :

PGCD(8,5)=PGCD(5,3)=PGCD(3,2)=PGCD(2,1)=PGCD(1,0)=1.

On remarque que PGCD($F_1$,$F_2$) n'est jamais appelé, donc le nombre d'appels exact est de $(i-1)$ pour $i > 0$.

#### 1.2.3 Complexité

Si l'on considère l'encadrement donné, la complexité de l'algorithme est en $O(\log F_i)$. Cela veut dire que si l'on utilise des nombres entiers codés sur n bits, il y aura $O(n)$ appels récursifs de la fonction EUCL.

De plus, si l'on considère qu'une multiplication ou une division est en $O(n)$, le nombre d'opération total sera en $O(n^3)$.

Avec, l'inégalité sur $F_i$, on peut monter que :

$$\frac{\ln F_i - 1}{k} \leq nbiter \leq \frac{\ln F_i + 1}{k}$$

### 1.3 Entiers codés par leur décomposition

#### 1.3.1 Complexité

On écrit A et B sous la forme suivante :

$$A = \prod_{i=1}^{n} p_i^{\alpha_i} \quad B = \prod_{i=1}^{n} p_i^{\beta_i} \quad PGCD(A, B) = \prod_{i=1}^{n} p_i^{\min(\alpha_i, \beta_i)}$$

On autorise des puissances nulles des nombres premiers pour que les nombres soient définis sur le même ensemble de nombres premiers $p_i$.

La complexité du calcul du PGCD est alors très simple, elle est en $O(n)$. Dans le cas général, si A est défini avec n nombres premiers et B avec m nombres premiers, l'algorithme est en $O(n+m)$.

**Remarque** Il n'existe pas d'algorithme qui donne en temps polynômial la décomposition d'un nombre entier en facteurs premiers. Si la décomposition n'est pas connue, il est donc peu efficace d'utiliser cette méthode.

#### 1.3.2 Algorithme ?

Pour écrire un algorithme, il faut d'abord choisir une structure de données pour stocker les décompositions.

**Solution 1** On suppose que A et B sont décomposés sur le même ensemble de nombres premiers. Dans un tableau P[ ], on stockera les nombres premiers $P[i] = p_i$. Les tableaux A[ ] et B[ ] contiendront respectivement les indices pour chaque nombre A et B.

Pour les solutions suivantes, on prend en compte le cas plus général où A et B ne sont pas décomposés sur la même base.

**Solution 2** Le tableau A[ ] et B[ ] seront définis comme suit :

$A[\,]$ ou $B[\,] = [p_0, \alpha_0, \ldots, p_i, \alpha_i, \ldots]$

Il faudra utiliser un indice de progression pour chaque tableau.

**Solution 3** La dernière solution proposée est la plus élégante des trois mais aussi la plus "difficile" à programmer. A chaque nombre A et B est associé une liste chainées de couples $(p_i, \alpha_i)$ :

$A \rightarrow \ldots \rightarrow (p_i, \alpha_i) \rightarrow \ldots \rightarrow NIL$

Là encore, il faudra un pointeur de progression sur chaque nombre.

# 2 Recherche par dichotomie

On recherche un élément dans un tableau d'entiers déjà trié.

## 2.1 Dichotomie simple

### 2.1.1 Exemples

Tableau : (2,3,5,6,9,14,15,16,18,22,25,28)

Application à la valeur 6

DICHO(1, 12, 6)
milieu=6 valeur=14 DICHO(1,5,6)
milieu=3 valeur= 5 DICHO(4,5,6)
milieu=4 valeur= 6
Résultat : **4**

Application à la valeur 27

DICHO(1,12,27)
milieu= 6 valeur=14 DICHO( 7,12,27)
milieu= 9 valeur=18 DICHO(10,10,27)
milieu=11 valeur=25 DICHO(12,12,27)
Résultat : **-1**

### 2.1.2 Algorithme

DICHO(index1,index2,valeur)
  *% condition d'arrêt : égalité des indices*
  SI (index1≥index2)
  ALORS
    SI (tab[index2]=valeur)
    ALORS
      RETOURNER index2
    FSI
    RETOURNER -1 *% échec*

```
        FSI

        milieu := (index1+index2)/2  % Calcul de l'indice milieu
        SI (tab[milieu]=valeur) ALORS RETOURNER milieu FSI
        SI (tab[milieu]≤valeur) % Recherche à droite
        ALORS
            RETOURNER DICHO(milieu -1, index2, valeur)
        FSI

        % Recherche à gauche
        RETOURNER DICHO(index1, milieu+1, valeur)
    FIN
```

**Données :** un tableau **tab** de valeurs indexé de un à max. Ce tableau peut être passé en paramètre.

**Initialisation :** l'appel de la recherche dichotomique se fait avec :
- **index1** = 1,
- **index2** = max

et **valeur** est la valeur recherchée.

**Résultat :** la fonction renvoie l'**index** de la valeur dans le tableau et **-1** si la valeur n'est pas présente dans le tableau.

**Remarque** La ligne de test avec la valeur du milieu n'est pas obligatoire. Dans ce cas, il faut inclure la valeur milieu dans l'un des intervalles de recherche.

### 2.1.3 Complexité

Démontrons que la complexité de cet algorithme est en $\log n$.
Les pires des cas pour cet algorithme sont les suivants : la valeur cherchée n'est pas présente dans le tableau ou bien c'est la dernière valeur testée, c'est-à-dire que la valeur n'est pas trouvée ou les indices sont égaux. Le nombre d'itérations à effectuer est alors de n+1 avec un tableau de taille $2^n$. Par suite, toute taille de tableau k peut-être encadrée comme suit : $2^n \leq k \leq 2^{n+1} - 1$. La complexité est donc en $\log k$.

## 2.2 Recherche dichotomique par interpolation

### 2.2.1 Calcul de l'indice probable $\overline{\imath}$

On prend $x$ la valeur à chercher dans une suite d'éléments régulièrment espacés sur l'intervalle [min, max]. On appelle $i_{min}$ et $i_{max}$ les indices repectivement du premier et du dernier élément de la suite. On cherche $\overline{\imath}$ l'index probable de $x$.

La relation de proportionalité suivante est vérifiée :

$$\frac{x - min}{max - min} = \frac{\overline{\imath} - i_{min}}{i_{max} - i_{min}}$$

d'où le résultat suivant :

$$\overline{\imath} = \frac{x - min}{max - min} \times (i_{max} - i_{min}) + i_{min}$$

### 2.2.2 Procédure

Il suffit de remplacer la ligne de calcul de "milieu" par la ligne suivante :

$$milieu := (valeur - tab[index1])/(tab[index2] - tab[index1]) * (index2 - index1) + index1$$

### 2.2.3 Application

Application à la valeur 6

DICHO2(1, 12, 6)
milieu=2,69 valeur= 3 DICHO2(3,12,6)
milieu=3,39 valeur= 5 DICHO2(4,12,6)
milieu=4 valeur= 6
Résultat : **4**

Application à la valeur 27

DICHO2(1, 12, 27)
milieu=11,57 valeur= 25 DICHO2(12,12,27)
Résultat : **-1**

# 3 Tris simples

## 3.1 Tri par insertion

Voilà l'algorithme de tri officiel :

INSERT (tableau tab)
DÉBUT
 $tab[0] := -\infty$
 POUR i de 1 À $n-1$ FAIRE
  $j := i+1$
  TANT QUE (tab[j] < tab[j-1]) FAIRE
   ECHANGER(tab[j-1], tab[j])
   $j := j-1$
  FTANTQUE
 FPOUR
FIN

**Remarque** Dans le premier élément du tableau, tab[0], on place une sentinelle de valeur plus petite que toutes les valeurs possibles.

Voici un autre algorithme, tout aussi valable...

INSERT2 (tableau A)
DÉBUT
 POUR j DE 2 À n FAIRE *% n est la taille de A*
  clé:= $A[j]$
  $i := j-1$
  TANT QUE ($i$ >0 ET A[i]>clé) FAIRE
   $A[i+1] := A[i]$
   $i := i-1$

```
        FTANTQUE
        A[i + 1] := clé
    FPOUR
FIN
```

## 3.2 Tri à bulles

### 3.2.1 Description

À chaque étape i, on ramène en $i^{\text{ème}}$ position le $i^{\text{ème}}$ plus petit élément. Il n'est donc pas nécessaire de parcourir en entier le tableau, on peut oublier les $(i-1)$ éléments déjà placés.

### 3.2.2 Condition d'arrêt

La condition d'arrêt de cette méthode est toute simple : le tableau doit être trié, c'est-à-dire qu'aucune permutation d'élément ne doit être réalisable.

### 3.2.3 Algorithme

```
BULLE
DÉBUT
    fin := FAUX
    i := 1
    TANT QUE (NON fin) FAIRE
        fin := VRAI
        POUR j DE n À i+1 PAS -1 FAIRE
            SI (tab[j] < tab[j-1]) ALORS
                ECHANGER(tab[j-1], tab[j])
                fin := FAUX
            FSI
        FPOUR
        i := i + 1
    FTANTQUE
FIN
```

## 3.3 Tri par sélection ou échange

On ne donne l'algorithme que pour rappel et pour calculer sa complexité dans la question suivante.

```
SELECTION
DÉBUT
    POUR i DE 1 À n-1 FAIRE
        pos_min := i
        cle_min := tab[i]
        POUR j DE i + 1 À n FAIRE
            SI (tab[j] < tab[pos_min])
            ALORS
                cle_min := tab[j]
                pos_min := j
            FSI
```

FPOUR
ECHANGE(tab[i], tab[pos_min])
FPOUR
FIN

## 3.4 Comparaison des tris élémentaires

On remarque que les algorithmes de tri présentés sont d'autant plus efficaces que les tableaux sont triés. Le pire des cas correspond à une entrée triée dans le sens inverse à celui prévu.

### 3.4.1 Tableaux déjà triés

**Tri par insertion :** $n-1$ tests

Une boucle complète POUR est effectuée avec un seul test TANT QUE (et pas d'échange).

**Tri à bulles :** $n-1$ tests

Un seul passage dans la boucle TANT QUE est réalisé avec à l'intérieur, une boucle complète POUR (1 test en interne, et pas d'échange)

### 3.4.2 Tableaux triés en ordre inverse

**Tri par insertion :** $\frac{n^2+n-2}{2}$ tests et $\frac{(n-1)n}{2}$ échanges

La boucle TANT QUE s'arrête quand $j = 1$ et alors il n'y a pas d'échange. À chaque itération de la boucle POUR, il y a donc $i+1$ tests et $i$ échanges.

**Tri à bulles :** $\frac{n.(n-1)}{2}$ tests et $\frac{n.(n-1)}{2}$ échanges

L'algorithme s'arrête quand le booléen fin ne peut plus être mis à FAUX, c'est-à-dire quand $i+1 > n$. i=n suffit pour déconnecter la boucle POUR et alors i vaut n+1 après la boucle TANT QUE. Il y aura donc eu $n+1$ tests sur la boucle TANT QUE elle-même. Sur ces $n+1$ boucles, n sont complexes et contiennent des boucles POUR. Chaque boucle POUR réalise $n-i$ tests (condition SI) et échanges.

### 3.4.3 Tri par sélection

On remarque que la complexité du tri par sélection (ou échange) ne change pas que le tableau en entrée soit trié ou pas.

Il y a $n-1$ échanges (un par itération principale). De plus, pour chaque itération de i, il y a $n-i$ tests sur le tableau.

Soit un total de $n-1$ échanges et $\frac{n(n+1)}{2}$ tests.

Le tri par sélection semble être meilleur dans le pire des cas mais il n'est pas efficace quand le tableau est deja trié. Dans tous les cas, les algorithmes présentés sont tous en $O(n^2)$ donc peu rapides. On veillera donc à utiliser d'autres algorithmes de tri, sauf si l'on manque de temps ou si la taille des données ne l'exige pas.

# 4 Tri par dénombrement

Le tri par dénombrement est un tri dit "linéaire". Il est applicable lorque l'intervalle des valeurs à trier n'est pas trop grand.

DENOMBREMENT(A, B, n, k)
DEBUT
  POUR $i$ DE 1 À k FAIRE $C[i] := 0$ FPOUR
  POUR $i$ DE 1 À n FAIRE $C[A[i]] := 1$ FPOUR
  POUR $i$ DE 2 À k FAIRE $C[i] := C[i] + C[i-1]$ FPOUR
  POUR $i$ DE 1 À n FAIRE $B[C[A[i]]] := A[i]$ FPOUR
FIN

## 4.1 Exemple

Soit A=[15,6,20,9,8,10,3] n=7 (k=20)

Boucle POUR 1 : C = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Boucle POUR 2 : C = [0,0,1,0,0,1,0,1,1,1,0,0,0,0,1,0,0,0,0,1]
Boucle POUR 3 : C = [0,0,1,1,1,2,2,3,4,5,5,5,5,5,6,6,6,6,6,7]
Boucle POUR 4 : B = [3,6,8,9,10,15,20]

## 4.2 Complexité

Boucle POUR 1 : k affectations
Boucle POUR 2 : n affectations
Boucle POUR 3 : k-1 affectations et k-1 additions
Boucle POUR 4 : n affectations

Soit une complexité finale de $3k + 2n - 2$

Si $k = O(n^2)$ alors la complexité est en $O(n^2)$.
Si $k = O(n)$ alors la complexité est en $O(n)$.

En général, k est de l'ordre de n, donc la complexité moyenne de l'algorithme est en $O(k)$. En cours, on va vu que les algorithmes étaient en général en $O(n.\log n)$. Donc le tri par dénombrement peut être plus rapide qu'un tri usuel. En fait, la limitation d'un tri usuel vient du nombre de comparaisons, or, dans le tri par dénombrement, il n'y a aucune comparaison.

## 4.3 Généralisation

On s'intéresse maintenant au cas où les éléments du tableau A ne sont pas tous distincts. L'algorithme est le suivant :

DENOMBREMENT_GEN(A, B, n, k)
DEBUT
  POUR $i$ DE 1 À k FAIRE $C[i] := 0$ FPOUR
  POUR $i$ DE 1 À n FAIRE $C[A[i]] := C[A[i]] + 1$ FPOUR
  POUR $i$ DE 2 À k FAIRE $C[i] := C[i] + C[i-1]$ FPOUR
  POUR $i$ DE 1 À n FAIRE
    $B[C[A[i]]] := A[i]$
    $C[A[i]] := C[A[i]] - 1$
  FPOUR
FIN

Si on prend l'exemple du tableau A à trier où A=[15,6,15,9,9,8,3,9,3,2,2,1]

Les différentes étapes donnent les vecteurs suivants :
C=[ 1, 2, 2, 0, 0, 1, 0, 1, 3, 0, 0, 0, 0, 0, 2]
C=[ 1, 3, 5, 5, 5, 6, 6, 7,10,10,10,10,10,10,12]
et le résultat attendu suivant : B=[1,2,2,3,3,6,8,9,9,9,15,15]

# 5 Tri par éclatement-fusion

## 5.1 Algorithme

```
TRI_EF(T, i, j)
DEBUT
    SI (i < j) ALORS
        milieu := (i + j - 1)/2
        TRI_EF(T, i, milieu)
        TRI_EF(T, milieu+1, j)
        FUSION(T, i, milieu, j)
    FSI
FIN

FUSION(T, g, m, d)
DEBUT
    POUR i DE g À m FAIRE R[i] := T[i] FPOUR
    POUR i DE m+1 À d FAIRE
        R[i] := T[d - (i - m - 1)]
    FPOUR
    i := g
    j := d
    POUR k DE g À d FAIRE
        SI R[i] < R[j] ALORS
            T[k] := R[i]
            i := i+1
        SINON
            T[k] := R[j]
            j := j-1
        FSI
    FPOUR
FIN
```

## 5.2 Exemple

```
TRI(T, 1,14)
    milieu := 7
    TRI(T,1,7)
        milieu := 3
        TRI(T, 1, 3)
            milieu := 1
            TRI(T,1,1)
            TRI(T,2,3)
                milieu := 2
                TRI(T,2,2)
                TRI(T,3,3)
                FUSION(T,2,2,3)
            FUSION(T,1,1,3)
        TRI(T,4,7)
            milieu := 5
            TRI(T,4,5)
                milieu := 4
                TRI(T,4,4)
                TRI(T,5,5)
                FUSION(T,4,4,5)
            TRI(T,6,7)
                milieu := 6
                TRI(T,6,6)
                TRI(T,7,7)
                FUSION(T,6,6,7)
            FUSION(T,4,5,7)
        FUSION(T,1,3,7)
    TRI(T,8,14)
```

```
        milieu := 10
        TRI(T,8,10)
            milieu := 8 TRI(T,8,8)
            TRI(T,9,10)
                milieu := 9
                TRI(T,9,9)
                TRI(T,10,10)
                FUSION(T,9,9,10)
            FUSION(T,8,8,10)
        TRI(T,11,14)
            milieu := 12
            TRI(T,11,12)
                milieu := 11 TRI(T,11,11)
                TRI(T,12,12)
                FUSION(T,11,11,12)
            TRI(T,13,14)
                milieu := 13
                TRI(T,13,13)
                TRI(T,14,14)
                FUSION(T,13,13,14)
            FUSION(T,11,12,14)
        FUSION(T,8,10,14)
    FUSION(T,1,7,14))
```

### 5.3 Complexité

**Fusion** Par fusion, il y a $d-g+1$ tests et $2(d-g+1)$ affectations.

**Complexité globale** On se place à l'itération i et on suppose que l'on trie $n = d-g+1$ éléments. La complexité de l'itération i est la somme de la complexité du tri sur la partie gauche, du tri sur la partie droite et de la fusion entre les deux parties. Le tri sur la partie gauche (itération i+1) s'effectue sur $\lfloor n/2 \rfloor$ et la partie droite sur $\lceil n/2 \rceil$. Si l'on ne considère que les tests pour la fusion, on peut appliquer la formule et en déduire que la complexité du tri est en $O(n.\log n)$.

## 6 Tri rapide

### 6.1 Algorithme

```
TRI_RAPIDE(tableau T, i, j)
DÉBUT
    SI (i < j) ALORS
        pivot := PLACER(T, i, j)
        TRI_RAPIDE(T, i, pivot-1)
        TRI_RAPIDE(T, pivot+1, j)
    FSI
FIN

PLACER(T, g, d)
DÉBUT
    i1 := g + 1
    i2 := d
    TANT QUE (i1 ≤ i2) FAIRE
        TANT QUE (T[i1] ≤ T[g]) FAIREi1 := i1 + 1 FTANTQUE
        TANT QUE (T[i2] > T[g]) FAIREi2 := i2 − 1 FTANTQUE
        SI (i1 < i2) ALORS
            ECHANGER(T, i1, i2)
            i1 := i1 + 1
            i2 := i2 − 1
        FSI
```

```
        FTANTQUE
        ECHANGER (T, g, i2)
        RETOURNER i2
    FIN
```

**Remarque** Une sentinelle est nécessaire pour que l'algorithme s'arrête. Il ne faut pas que l'élément pivot soit le plus grand de la liste à trier.

```
    PLACER2(T, g, d)
    DÉBUT
        x := T[g]
        i := g − 1
        j := d + 1
        TANT QUE VRAI FAIRE
            RÉPÉTER j := j − 1 JUSQU'À (T[j] ≤ x) FRÉPÉTER
            RÉPÉTER i := i + 1 JUSQU'À (T[i] ≥ x) FRÉPÉTER
            SI (i < j)
                ALORS ECHANGER(T, i, j)
                SINON RETOURNER j
            FSI
    FIN
```

## 6.2 Exemple

On va faire tourner l'algorithme de tri rapide avec le tableau suivant :
*(12,28,6,1,22,10,2,13,4,5,3,24,20,11)*

```
TRI_RAPIDE(1,14)
    PLACER(1,14) (4,11,6,1,3,10,2,5,12,13,22,24,20,28) pivot = 9
    TRI_RAPIDE(1,8)
        PLACER(1,8) (1,2,3,4,6,10,11,5) pivot = 4
        TRI_RAPIDE(1,3)
            PLACER(1,3) (1,2,3) pivot = 1
            TRI_RAPIDE(1,0)
            TRI_RAPIDE(2,3)
                PLACER(2,3) (2, 3) pivot = 2
                TRI_RAPIDE(2,1) TRI_RAPIDE(3,3)
        TRI_RAPIDE(5,8)
            PLACER(5,8) (5,6,11,10) pivot = 6
            TRI_RAPIDE(5,5)
            TRI_RAPIDE(7,8)
                PLACER(7,8) (10, 11) pivot = 8
                TRI_RAPIDE(7,7)
                TRI_RAPIDE(9,8)
    TRI_RAPIDE(10,14)
        PLACER(10,14) (13,22,24,20,28) pivot = 10
        TRI_RAPIDE(10,9)
        TRI_RAPIDE(11,14)
            PLACER(11,14) (20, 22, 24, 28) pivot = 12
            TRI_RAPIDE(11,11)
            TRI_RAPIDE(13,14)
                PLACER(13,14) (24, 28) pivot = 13
                TRI_RAPIDE(13,12)
                TRI_RAPIDE(14,14)
```

## 6.3 Pire des cas

Le pire des cas pour l'algorithme du tri rapide est quand chaque décomposition donne 1 et $n - 1$. Ceci correspond à des tableaux déjà triés.

## 6.4 Algorithme stochastique

On dit d'un algorithme qu'il est **stochastique** si son comportement est déterminé non seulement par l'entrée mais aussi par les valeurs produites par un générateur de nombres aléatoires.

Par suite, aucune entrée particulière (même volontaire) ne provoque un comportement dans le pire des cas (on écarte le cas trivial où toutes les valeurs sont égales). En revanche, le pire des cas dépend du générateur.

Par exemple, on choisira un pivot au hasard dans la liste des nombres à trier. Même si le coût de la perturbation de l'entrée s'ajoute au coût du tri, perturber l'entrée est "toujours" valable.

## 6.5 Tri par l'élément médian

**Exemple** *12,28,6,1,22,10,2,13,4,5,3,24,20,11*
Première itération identique : valeur médiane de 12, 28 et 6 : 12.
*4,11,6,1,3,10,2,5,**12**,13,22,24,20,28*
*6,11,4,1,3,10,2,5*
*2,4,5,1,3,**6**,10,11*
*4,2,5,1,3*
*1,2,3,**4**,5*
*22,13,20,24,28*
*20,13,**22**,24,28*
Tout ce qui n'est pas encore trié est trié par le tri à bulle.

**Variante**

```
TRI_BIS (tab : tableau ; g, d :indices )
DÉBUT
    SI (d − g + 1 ≤ 3) ALORS BULLE_FIN(tab, g, d)
    SINON
        pivot := MEDIAN(tab, g)
        ECHANGER(tab[g], tab[pivot])
        pivot := PLACER(tab, g, d)
        TRI_BIS(tab, g, pivot-1)
        TRI_BIS(tab, pivot+1, d)
    FSI
FIN
```

**Remarque** Ce tri n'améliore que d'un facteur constant l'expression $O(n.\log n)$

# 7 File implémentée par liste chaînée

La *file* est un type abstrait de données linéaire : elle regroupe une collection d'éléments de même nature. Le premier élément entré est aussi le premier élément qui sort (gestion FIFO ou "First In First Out"). La pile, par exemple, utilise une autre politique d'entrée-sortie. Le dernier élément entré est le premier sorti (gestion LIFO ou "Last In, First Out").

## 7.1 Différentes structures de données

On peut modéliser les files en s'appuyant sur différentes structures de données : des tableaux ou des listes chaînées par exemple. Un exemple de modélisation avec une tableau a été vu en cours, on va donc s'attacher à décrire des modélisations possibles par liste chaînée.

**Modèle 1** On considère que la liste chaînée est définie grâce à deux pointeurs : un pointeur de tête et un pointeur de queue. Les opérations d'insertion et de supression à ces endroits vont donc être très faciles.

**Modèle 2** On considère que la file est définie grâce à un unique pointeur : le pointeur de tête et la fin de la liste chaînée est bien marquée par un pointeur nul. L'opération (insertion ou suppression) qui se déroulera en tête sera triviale mais il faudra chercher la fin de la liste pour l'autre opération, ce qui implique que cette dernière ne sera pas en $O(1)$.

**Modèle 3** On considère que la file est définie grâce à un unique pointeur mais cette fois la liste est circulaire.

**Modèle 4** On reprend le modèle précédent mais cette fois, la liste contient toujours au moins un élément dit fictif. La présence de cet élément diminue le nombre de cas limites.

## 7.2 Description du modèle

On va décrire très précisément le modèle 3. Il faut insister sur le fait que la donnée du pointeur définit complètement une **file**, on va donc appeler celui-ci file. Attention toutefois, à ne pas utiliser ce mot-clef en *Turbo Pascal* où le terme désigne déjà le type *fichier*.

### 7.2.1 Algorithme et pointeurs

Pour ne pas s'attacher à un langage particulier, il faut redéfinir les opérations communes sur les pointeurs :
- <u>ALLOUER</u> (pointeur) permet d'allouer la mémoire nécessaire au pointeur *pointeur*.
- <u>LIBÉRER</u> (pointeur) permet de rendre au système la mémoire que désigne le pointeur *pointeur*.
- <u>CONTENU</u> (pointeur) permet de connaître le contenu que désigne le pointeur *pointeur*. C'est l'équivalent de l'instruction pascal *pointeur*ˆ
- <u>ADRESSE</u> (objet) permet de connaître l'adresse en mémoire de l'objet *objet*. C'est l'équivalent de l'instruction pascal ˆ*objet*

On appelle *NIL* la valeur du pointeur nul.

### 7.2.2 Opérations élémentaires

La file s'appuie sur une liste chaînée qui est composée de cellules (ou blocs) définies à la figure 1. En *Turbo Pascal*, le champ objet sera une copie de l'objet. Dans le cas général, le champ objet ne sera qu'un pointeur sur l'objet.

La liste vide est représentée à la figure 2 et la file à un seul élément à la figure 3.

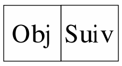


FIG. 1 – Format de cellule

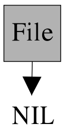


FIG. 2 – File vide

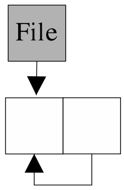


FIG. 3 – File à un seul élément

Des files "normales" sont représentées aux figures 4 et 5. L'élément indicé "un" est le premier a avoir été placé dans la file, l'élément numéro "trois" a été le dernier. On remarque par conséquent que le pointeur *file* désigne en fait le dernier élément de la file.

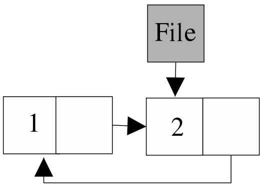


FIG. 4 – File à deux éléments

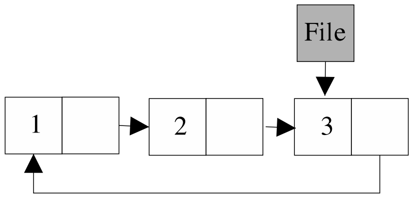


FIG. 5 – File à trois éléments

Il va donc s'agir d'insérer un élément en fin de la liste chaînée ("enfiler un élément") et de supprimer un élément en tête de liste chaînée. L'élément "trois" est d'abord enfilé à la figure 6, puis l'élément "un" est défilé à la figure 7.

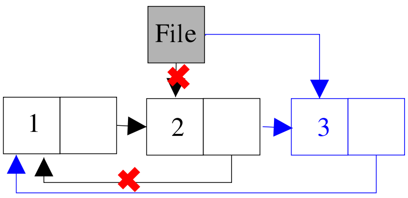


FIG. 6 – Enfiler un élément

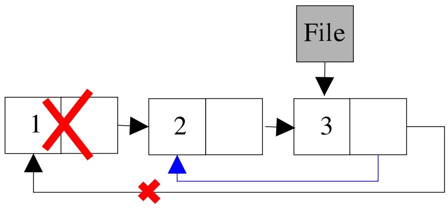


FIG. 7 – Défiler un élément

## 7.3 Primitives

Certaines situations peuvent provoquer des situations d'erreurs. Celles-ci sont signalées par le mot-clef "PROBLÈME". En *Turbo Pascal*, on pourra se contenter d'afficher un message d'erreur. Avec un language plus évolué, on pourra utiliser un système de gestion d'erreurs (levée et interception d'exceptions par exemple).

### 7.3.1 EST_FILE_VIDE

EST_FILE_VIDE (file *file*)
<u>DÉBUT</u>
    <u>RETOURNER</u> (file = nil)
<u>FIN</u>

C'est l'opération la plus facile à écrire. Par définition, le file est vide lorsque le pointeur *file* vaut NIL.

#### 7.3.2 FILE_VIDE

Cette procédure vide la file de tous ses éléments. Dans l'algorithme, il faut distinguer les cas où la file est déjà vide, n'a qu'un seul élément et a plusieurs éléments.

```
FILE_VIDE (file file)
DÉBUT
    SI NON EST_FILE_VIDE(file) ALORS
        TANT QUE contenu(file).suivant ≠ file FAIRE
            temp := contenu(file).suivant
            libérer (file)
            file := temp
        FTANTQUE
        libérer(file)
    FSI
    file := nil
FIN
```

Bien entendu, vider la file avec une boucle

```
TANT QUE NON(EST_FILE_VIDE(file)) FAIRE
    DEFILER(file)
FTANTQUE
```

est tout aussi valable.Cette opération est en $O(n)$ où n est la taille de la file.

#### 7.3.3 ENFILER

Cette procédure permet d'ajouter un nouvel élément à la file.

```
ENFILER (file file, objet X)
DÉBUT
    allouer C
    contenu(C).objet := X
    SI EST_FILE_VIDE(file) ALORS
        contenu(C).suivant := C
    SINON
        contenu(C).suivant := contenu(file).suivant
        contenu(file).suivant := C
    FSI
    file := C
FIN
```

Cette opération ne dépend pas de la taille de la file, donc elle s'effectue en $O(1)$.

#### 7.3.4 TETE_FILE

Cette fonction permet de consulter la valeur du premier élément entré dans la file.

```
TETE_FILE (file file)
DÉBUT
    SI EST_FILE_VIDE(file) ALORS PROBLEME
    SINON
        RETOURNER contenu((contenu(file).suivant)).element
    FSI
FIN
```

Cette opération ne dépend pas de la taille de la file, donc elle s'effectue en $O(1)$.

#### 7.3.5 DEFILER

Cette procédure enlève le premier élément entré dans la file. Il faut d'abord tester si la file n'a qu'un seul élément.

```
DEFILER (file file)
DÉBUT
    SI EST_FILE_VIDE(file) ALORS PROBLEME
    SINON
        SI contenu(file).suivant = file ALORS
            liberer(file)
            file := nil
        SINON
            temp := contenu(contenu(file).suivant).suivant
            liberer (contenu(file).suivant)
            contenu (file).suivant := temp ;
        FSI
    FSI
FIN
```

Cette opération ne dépend pas de la taille de la file, donc elle s'effectue en $O(1)$.

**Remarque 1** Le pointeur *file* permet un accès direct sur le début et la fin de la liste, ce qui permet d'avoir des accès en $O(1)$ sur la liste pour les opérations usuelles.

**Remarque 2** Il n'est pas nécessaire d'utiliser une file doublement chaînée pour **ce** type de file dans la mesure où l'on a déjà les performances voulues. Ce serait par contre nécessaire si l'accès à des éléments intermédaires était requis.

### 7.4 Exemple

On va représenter l'exemple sous forme de calendrier des événements.

9h00 : personne
9h08 : Arrivée d'Alfred. servi aussitôt. fin de service à 9h11.
9h11 : Alfred est servi.
9h12 : Arrivée de Bertram. servi aussitôt. fin de service à 9h15.
9h13 : Arrivée de Gertrude. mise en attente.
9h14 : Arrivée de Khaled. mis en attente.
9h15 : Bertram est servi. Gertrude se sert. fin de service à 9h18.
9h16 : Arrivée de Myriam. mise en attente.
9h18 : Gertude est servie. Khaled se sert. fin de service à 9h21.
9h21 : Khaled est servi. Myriam se sert. fin de service à 9h24.
Nathalie arrive. mise en attente
9h24 : Myriam est servie. Nathalie se sert. fin de service à 9h27.
9h26 : Arrivée de Peter. mis en attente.
9h27 : Nathalie est servie. Peter se sert. fin de service à 9h30.
Arrivée de Zoé. mise en attente
9h30 : Peter est servi. Zoé se sert. fin de service à 9h33
9h33 : Zoé est servie

# 8 Insuffisance des structures linéaires

## 8.1 Files de priorité

On va implémenter la file de priorité grâce à une simple liste chaînée. La liste va être triée selon la priorité de l'élément. On suppose que plus un élément est important plus sa priorité est proche de zéro. L'opération *extrairemin* prendra donc un temps constant car il suffira de renvoyer le premier élément de la liste et de supprimer la cellule correspondante.

```
EXTRAIRE_MIN (filep)
DÉBUT
    SI EST_FILE_VIDE(filep) ALORS PROBLEME
    SINON
        temp = contenu(filep).suivant % on sauve la tête de la liste
        filep := contenu(filep).objet
        libérer(filep) % on enlève le min de la liste
        filep = temp
        RETOURNER objet
    FSI
FIN
```

La procédure d'insertion est un tout petit peu plus complexe car la liste doit être parcourue pour trouver la bonne place de l'élément à insérer. Il faut distinguer trois cas : la file est vide et l'élément est inséré en tête, l'élément est le plus important et doit aussi être inséré en tête, l'élément est de priorité quelconque et peut être inséré n'importe où (sauf en tête).

```
INSERTION(filep, priorité P, élément E)
DÉBUT
    allouer(C)    contenu(C).suivant = nil
    contenu(C).priorité = P    contenu(C).élément = E
    SI EST_FILE_VIDE(filep)
    ALORS filep = C
    SINON
        SI contenu(filep).priorité ¿ P
        ALORS
            contenu(C).suivant = filep
            filep = C
        SINON
            precedent = filep
            courant = contenu(filep).suivant
            TANT QUE (courant ≠ nil) ET (contenu(courant).priorité
            ≤ P) FAIRE
                precedent = courant
                courant = contenu(courant).suivant
            FTANTQUEcontenu(C).suivant = courant
            contenu(precedent).suivant = C
        FSI
    FSI
```

Si l'on suppose que les insertions se font de manière aléatoire, la complexité moyenne de la primitive *insertion* est en $O(n)$.

## 8.2 Dictionnaire

Un dictionnaire est un ensemble dynamique qui supporte les opérations élémentaires telles que l'insertion, la suppression et le test d'appartenance d'un élément. On va utiliser une file pour implémenter le dictionnaire. L'élément va être une chaîne de caractères correspondant au nom à stocker ou bien un ensemble clé-valeur où la clé est le nom et la valeur par exemple sa définition. Dans tous les cas, on suppose que l'on dispose d'un ordre total sur les éléments.

```
EST_DICO_VIDE (dico dico)
DÉBUT
    RETOURNER (dico = nil)
FIN

DICO_VIDE (dico dico)
DÉBUT
    TANT QUE NON EST_DICO_VIDE(file) FAIRE
        temp = dico
        dico = contenu(dico).suivant
        libérer(temp)
    FTANTQUE
    dico = nil
FIN
```

Pour la recherche, on renvoie le premier élément qui correspond à la clé.

```
RECHERCHER(dico dico, élément X)
DÉBUT
    courant = dico
    TANT QUE NON EST_DICO_VIDE(courant)
    OU (contenu(courant).element < X) FAIRE
        courant := contenu(courant).suivant
    FTANTQUE
    SI NON EST_DICO_VIDe(courant) ET (contenu(courant).element = X)
    ALORS RETOURNER courant
    FSI
    RETOURNER nil
FIN
```

L'insertion dans un dictionnaire est quasiment identique à l'insertion dans la file de priorité (la priorité est remplacée par la clé).

```
INSERTION(dico dico, élément E)
DÉBUT
    allouer(C)
    DICO_VIDE(contenu(C).suivant)
    contenu(C).élément = E
    SI EST_DICO_VIDE(dico)
    ALORS dico = C
    SINON
        SI contenu(dico).élément > E ALORS
            contenu(C).suivant = dico
            filep = C
        SINON
```

```
                precedent = dico
                courant = contenu(dico).suivant
                TANT QUE NON EST_DICO_VIDE(courant)
                ET (contenu(courant).élément < E) FAIRE
                    precedent = courant
                    courant = contenu(courant).suivant
                FTANTQUE
                contenu(C).suivant = courant
                contenu(precedent).suivant = C
            FSI
        FSI
    FIN
```

Pour la supression d'un élément, il faut envisager trois cas possibles, le dictionnaire est vide, l'élément est en tête et l'élément est à une place quelconque (sauf en tête). De plus, il y a une recherche préliminaire et il est possible que l'élément ne se trouve pas dans le dictionnaire. Notons qu'il n'est pas possible d'utiliser la recherche précédente (telle qu'elle a été écrite) car il est nécessaire de connaître le prédécesseur de l'élément à supprimer.

```
SUPPRESSION(dico dico, élément E)
DÉBUT
    SI EST_DICO_VIDE(dico) ALORS "PROBLEME"
    SINON
        SI contenu(dico).élément = E ALORS
            temp = dico
            dico = contenu(dico).suivant
            libérer(temp)
        SINON
            precedent = dico
            courant = contenu(dico).suivant
            TANT QUE NON EST_DICO_VIDE(courant)
            ET (contenu(courant).élément ≤ E) FAIRE
                precedent = courant
                courant = contenu(courant).suivant
            FTANTQUE
            SI NON EST_DICO_VIDE(courant) ET
            (contenu(courant).élément=E) ALORS
                contenu(precedent).suivant = contenu(courant).suivant
                libérer(courant)
            FSI
        FSI
    FSI
FIN
```

### 8.3 Fusion de deux dictionnaires

Ce problème a déjà été vu à deux reprises : en TP, pour le calcul du PGCD lorque la décomposition en nombres premiers des opérandes est connue et en TD quand le tri par éclatement-fusion a été étudié.

```
FUSION (dico A, dico B, dico R)
DÉBUT
    % création de l'élément fictif
    allouer(R)
    DICO_VIDE(contenu(R).suivant)
    courantA = A
    courantB = B
    courantR = R
    TANT QUE NON EST_DICO_VIDE(courantA) ET NON EST_DICO_VIDE(courantB)
        allouer(C)
        DICO_VIDE(contenu(C).suivant)
        contenu(courantR).suivant = C
        courantR = C
        SI contenu(A).élément<contenu(B).élément ALORS
            contenu(courantR).élément = contenu(courantA).élément
            courantA = contenu(courantA).suivant
        SINON
            contenu(courantR).élément = contenu(courantB).élément
            courantB = contenu(courantB).suivant
    FTANTQUE

    TANT QUE NON EST_DICO_VIDE(courantA) FAIRE
        allouer(C)
        DICO_VIDE(contenu(C).suivant)
        contenu(courantR).suivant = C
        courantR = C
        contenu(courantR).élément = contenu(courantA).élément
        courantA = contenu(courantA).suivant
    FTANTQUE

    TANT QUE NON EST_DICO_VIDE(courantB) nil FAIRE
        allouer(C)
        DICO_VIDE(contenu(C).suivant)
        contenu(courantR).suivant = C
        courantR = C
        contenu(courantR).élément = contenu(courantB).élément
        courantB = contenu(courantB).suivant
    FTANTQUE
    % on suprime l'élément fictif
    courantR = R
    R = contenu(R).suivant
    libérer(courantR)

    RETOURNER R
FIN
```

# 9 Arbres binaires de recherche

## 9.1 Primitives des arbres

Les primitives d'accès sont :
- EST_ARBRE_VIDE(arbre A) : booléen
- CONTENU(arbre A) : élément
- RACINE(arbre A) : sommet (ou arbre ? ?)
- SAG(arbre A) : arbre
- SAD(arbre A ) : arbre

Les primitives de construction sont :
- ARBRE_VIDE(arbre A) crée un arbre vide
- FAIRE_ARBRE(élément X, arbre G, D) : arbre crée un arbre de racine de contenu X et de sous-arbres gauhe et droit, respectivement les arbres G et D.
- FIXER_CONTENU(élément X, arbre A) fixe le contenu de la racine de A
- FAIRE_FEUILLE(élément X) : arbre crée une feuille de contenu X (les sous-arbres gauche et droit sont initialisés à nil)

Pour notre implémentation, arbre est un pointeur sur une cellule qui dispose des champs suivants : un champ **cont** pour le contenu, un champ **gauche** qui est un pointeur sur le sous-arbre gauche et un champ **droit** qui est un pointeur sur le sous-arbre droit

## 9.2 Dictionnaire

On veut implémenter la structure Dictionnaire avec des arbres.

```
EST_DICO_VIDE (dico D)
DÉBUT
    RETOURNER (EST_ARBRE_VIDE(D))
FIN
```

```
DICO_VIDE (dico D)
DÉBUT
    ARBRE_VIDE(D)
FIN
```

Il est possible d'écrire deux versions de la recherche d'un élément : une version récursive et une version non récursive Voila la version récursive ...

```
RECHERCHER(dico D ; élément X)
DÉBUT
    SI (EST_ARBRE_VIDE(D)) OU (CONTENU(D)=X)
    ALORS RETOURNER D
    FSI
    SI CONTENU(D) < X
    ALORS RETOURNER RECHERCHER(SAD(D), X)
    SINON RETOURNER RECHERCHER(SAG(D), X)
    FSI
FIN
```

... et la version non récursive :

```
RECHERCHER2 (dico D, élément X)
DÉBUT
    courant = D
    TANT QUE ( NON EST_ARBRE_VIDE(courant))) ET CONTENU(courant)≠X
    FAIRE
        SI X< CONTENU(courant)
        ALORS courant := SAG(courant)
        SINON courant := SAD(courant)
        FSI
    FTANTQUE

    RETOURNER courant
FIN

INSÉRER(dico D, élément X)
DÉBUT
    C := FAIRE_FEUILLE(X)
    DICO_VIDE(precedent)
    courant := D
    TANT QUE NON EST_DICO_VIDE(courant) FAIRE
        precedent = courant
        SI CONTENU(courant) < X
        ALORS courant := SAD(courant)
        SINON courant := SAG(courant)
        FSI
    FTANTQUE
    SI EST_DICO_VIDE(precedent)
    ALORS D = C
    SINON
        SI X< CONTENU(precedent)
        ALORS contenu(precedent).gauche := C
        SINON contenu(precedent).droit := C
        FSI
    FSI
FIN
```

```
SUPPRESSION (dico D, élément X)
DÉBUT
    % Recherche de l'élément X dans le dictionnaire
    DICO_VIDE(precedent)
    courant = D
    TANT QUE NON EST_DICO_VIDE(courant) ET CONTENU(courant)≠X
        precedent := courant
        SI X< CONTENU(courant)
        ALORS courant := SAG(courant)
        SINON courant := SAD(courant)
        FSI
    FTANTQUE
    % ou moins une des branches manque
    SI EST_DICO_VIDE(SAG(courant)) OU EST_DICO_VIDE((courant))
    ALORS
        SI EST_DICO_VIDE(precedent)
        ALORS
            SI NON EST_DICO_VIDE(SAG(courant))
            ALORS D := SAG(courant)
            SINON D := SAD(courant)
            FSI
        SINON
            SI SAD(precedent)=courant ALORS
              SI NON EST_DICO_VIDE(SAG(courant))
              ALORS contenu(precedent).droit := SAG(courant)
              SINON contenu(precedent).droit :=SAD(courant)
              FSI
            SINON
              SI NON EST_DICO_VIDE(SAG(courant))
              ALORS contenu(precedent).gauche := SAG(courant)
              SINON contenu(precedent).gauche :=SAD(courant)
              FSI
            FSI
        FSI
        liberer(courant)
        DICO_VIDE(COURANT)
    SINON
        % les deux sous-arbres sont définis
        % recherche du prédécesseur de courant
        Sp := SAG(courant)
        TANT QUE NON EST_DICO_VIDE(SAD(Sp)) FAIRE
            Sp := SAD(Sp)
        FTANTQUE
        FIXER_CONTENU(CONTENU(Sp), courant)
        SUPPRIMER(SAG(courant), CONTENU(Sp))
    FSI
FIN
```

Pour la suppression du prédécesseur Sp, il n'y a pas de problème car Sp est une feuille ou n'a pas de sous-arbre droit

# 10 Équilibrage des arbres AVL

On appelle les arbres AVL les arbres binaires de recherche étudiés par Adelson-Velskii et Landis dans les années 1960, qui ont la propriété suivante : la différence de hauteur des deux sous-arbres pour toute racine est de au plus 1 en valeur absolue. Pour chaque racine R, on peut calculer son degré $d(R) = h(G) - h(D)$ où h(G) et h(D) sont les hauteurs respectives des sous-arbres gauche et droit de R. On manipule les arbres avec des rotations simples ou doubles

Les figures 8 et 9 représentent le même arbre après rotation simple. On obtient 9 après une rotation droite de 8 et réciproquement, 8 est obtenu par une rotation gauche de 9.

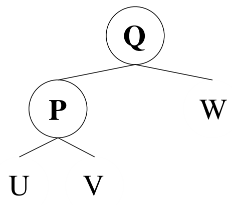


FIG. 8 – Arbre gauche

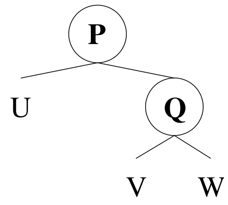


FIG. 9 – Arbre droit

On passe de 10 à 11 par une rotation double, dite gauche-droite. Une rotation gauche-droite est une rotation gauche du sous-arbre gauche suivie d'une rotation droite de l'ensemble.

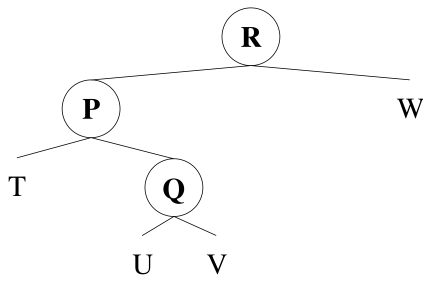


FIG. 10 – Arbre déséquilibré

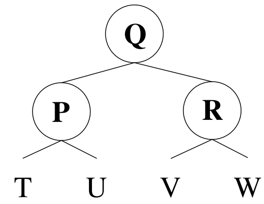


FIG. 11 – Arbre rééquilibré

Pour la suite, on s'appuie sur l'exercice II de l'examen de Juin 1998.

### 10.0.1 Construction de A

On veut construire l'arbre constitué des éléments suivants : 12, 9.1, 10.5, 14, 11, 17, 9.5, 6.5, 7

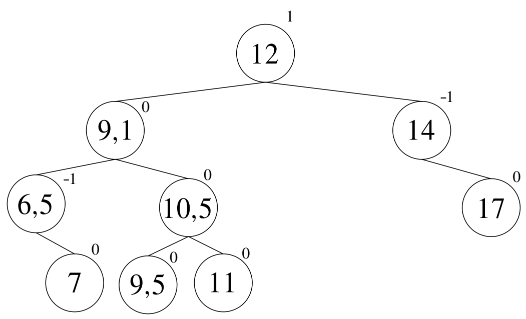


FIG. 12 – Arbre original

Cet arbre est équilibré au sens AVL : toutes les racines ont un degré d'au plus 1.

## 10.1 Rééquilibrage de l'arbre

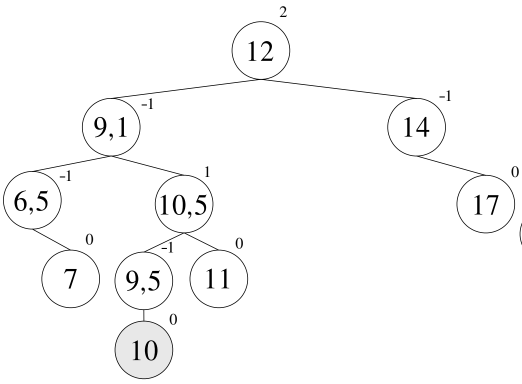


FIG. 13 – Ajout de 10

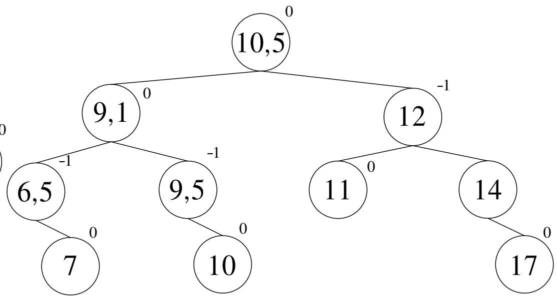


FIG. 14 – Arbre après rééquilibrage

Si l'on ajoute la note 10 de l'étudiant Popocateptl, l'arbre est déséquilibré et la racine 12 a un degré 2. Avec une rotation gauche-droite, l'arbre est à nouveau équilibré. Il faut faire une rotation double car 12 et 9,1 ont des degrés de signe différent.

## 10.2 Affichage ordonné

Pour afficher le contenu de l'arbre, il faut afficher celui-ci suivant l'ordre infixe.
La procédure AFFICHER affiche la racine de l'arbre A.

```
AFFICHER(arbre A)
DÉBUT
    SI NON EST_ARBRE_VIDE(A) ALORS
        afficher CONTENU(A).nom
        afficher CONTENU(A).prenom
        afficher CONTENU(A).note
    FSI
FIN
```

La procédure AFFICH+ affiche les notes des étudiants dans l'ordre croissant.

```
AFFICH+(arbre A)
DÉBUT
    SI NON EST_ARBRE_VIDE(A) ALORS
        AFFICH+ (SAG(A))
        AFFICHER(A)
        AFFICH+ (SAD(A))
    FSI
FIN
```

La procédure AFFICH- les affiche dans l'ordre décroissant.

```
AFFICH-(arbre A)
DÉBUT
    SI NON EST_ARBRE_VIDE(A) ALORS
        AFFICH- (SAD(A))
        AFFICHER(A)
        AFFICH- (SAG(A))
    FSI
FIN
```

### 10.3 Affichage d'un sous-ensemble

```
RECHERCHE_ANCETRE(arbre A, notes nx, ny)
DÉBUT
    SI NON EST_ARBRE_VIDE(A) ALORS
        SI CONTENU(a).note>ny ALORS
            RETOURNER RECHERCHE_ANCETRE(SAG(A), nx, ny)
        FSI
        SI CONTENU(a).note<nx ALORS
            RETOURNER RECHERCHE_ANCETRE(SAD(A), nx, ny)
        FSI
    FSI
    RETOURNER (A)
FIN

AFFICHAGE_ENTRE(arbre A, notes nx, ny)
DÉBUT
    r = RECHERCHE_ANCETRE(A, nx, ny)
    SI NON EST_ARBRE_VIDE(r) ALORS
        SI CONTENU(r).note>nx ALORS
            AFFICHAGE_GAUCHE(SAG(r), nx)
        FSI
        SI (CONTENU(r).note>nx) ET (CONTENU(r).note< ny) ALORS
            AFFICHER(r)
        FSI
        SI CONTENU(r).note<ny ALORS
            AFFICHAGE_DROIT(SAD(r), ny)
        FSI
    FSI
FIN

AFFICHAGE_GAUCHE(arbre A, note nx)
DÉBUT
    SI CONTENU(A).note>nx ALORS
        AFFICHAGE_GAUCHE(SAG(A), nx)
        AFFICHER(A)
        AFFICH+ (SAD(A))
    SINON
        SI CONTENU(A).note = nx
        ALORS AFFICH+(SAD(A))
        SINON AFFICHAGE_GAUCHE(SAD(A), nx)
        FSI
    FSI
FIN

AFFICHAGE_DROIT(arbre A, note ny)
DÉBUT
    SI CONTENU(A).note<ny ALORS
```

```
            AFFICH+ (SAG(A))
            AFFICHER(A)
            AFFICHAGE_DROIT(SAD(A), ny)
        SINON
            SI CONTENU(A).note = ny
            ALORS AFFICH+(SAG(A))
            SINON AFFICHAGE_DROIT(SAG(A), ny)
            FSI
        FSI
    FIN
```