ALGORITHMIQUE ET PROGRAMMATION

Philippe PETER

17 juillet 2018

Table des matières

1 Introduction **7**

2 Algorithme et langages de programmation **9**
2.1 Notion d'algorithme 9
2.2 Langages de programmation 9
2.2.1 Différents types de langages 9
2.2.2 Langages interprétés vs langages compilés 10

3 Rapide aperçu du langage Python **11**
3.1 Les types 11
3.2 Les expressions 11
3.2.1 Les expressions numériques 11
3.2.2 Les expressions booléennes 12
3.2.3 Les listes Python 12
3.2.4 Les pointeurs cachés (les références) 12
3.3 Les instructions 13
3.3.1 Affectation 13
3.3.2 Entrées / sorties 13
3.3.3 Instruction conditionnelle simple 13
3.3.4 Instruction conditionnelle généralisée 13
3.3.5 Instructions itératives 14
3.3.6 Documentation 14
3.4 Objets de types composés 15
3.4.1 Les tableaux 15
3.4.2 Les structures ou enregistrements 15
3.5 Les fichiers 15
3.5.1 Les fichiers séquentiels sans structure 15
3.5.2 Les fichiers séquentiels structurés 16
3.6 Les sous-programmes 16
3.6.1 Les procédures 16
3.6.2 Les fonctions 18
3.7 Les classes python 19
3.7.1 Syntaxe 20
3.7.2 La méthode __init__ 20
3.7.3 Les autres méthodes 21

4 Les listes linéaires **23**
4.1 Introduction . . . 23
4.2 Principales opérations sur les séquences . . . 23
4.3 Schémas d'algorithme . . . 23
4.3.1 Les files à nombre d'éléments déterminé . . . 24
4.3.2 Les files à sentinelle exclue . . . 24
4.3.3 Les files à sentinelle inluse . . . 24
4.4 Notations pour les algorithmes . . . 24
4.5 Algorithme de recherche . . . 25
4.5.1 Recherche séquentielle . . . 25
4.5.2 Recherche dichotomique . . . 25
4.5.3 Recherches auto adaptatives . . . 26
4.5.4 Complexité moyenne . . . 26
4.6 Ajout d'un élément . . . 26
4.7 Interclassement de deux séquences triées . . . 27
4.8 Intersection de deux séquences triées . . . 27
4.9 Les tris internes élémentaires . . . 28
4.9.1 Tri par sélection du minimum . . . 28
4.9.2 Tri à bulles . . . 28
4.9.3 Tri par insertion séquentielle . . . 29
4.9.4 Evaluation de ces méthodes . . . 29
4.9.5 Tri Shell . . . 29
4.9.6 Tris sans comparaison entre éléments . . . 30

5 La récursivité **33**
5.1 Exemples . . . 33
5.1.1 Factorielle n (n!) . . . 33
5.1.2 Suite de Fibonacci . . . 33
5.1.3 Recherche dichotomique . . . 33
5.2 Résolution récursive de problèmes . . . 34
5.3 Mécanismes . . . 34
5.3.1 A l'appel récursif . . . 34
5.3.2 Au retour d'un sous programme récursif . . . 34
5.3.3 Quand le langage n'est pas récursif . . . 35
5.4 Applications . . . 35
5.4.1 Les tours de Hanoi . . . 35
5.4.2 Le tri rapide (quicksort) . . . 35

6 Les listes chaînées **39**
6.1 Les listes simplement chaînées . . . 39
6.1.1 Structure . . . 39
6.1.2 Représentation interne . . . 39
6.1.3 Définition d'une liste avec des classes . . . 40
6.1.4 Exercices . . . 41
6.2 Les liste circulaires . . . 42
6.3 Les listes doublement chaînées . . . 42

7 Les piles et les files **45**
7.1 Les piles . . . 45
7.1.1 Mode d'accès aux éléments . . . 45
7.1.2 Opérations sur les piles . . . 45
7.1.3 Représentation interne . . . 46
7.1.4 Algorithmes . . . 46
7.1.5 Utilisation des piles . . . 47
7.2 Les files d'attente . . . 47
7.2.1 Mode d'accès aux éléments . . . 47
7.2.2 Opérations sur les files d'attente . . . 47
7.2.3 Représentation interne . . . 48
7.2.4 Algorithmes avec des liste Python . . . 48
7.2.5 Utilisation des files d'attente . . . 48

8 L'adressage dispersé **49**
8.1 Principe . . . 49
8.2 Fonction de dispersion . . . 49
8.2.1 Définition . . . 49
8.2.2 Choix de la fonction de dispersion . . . 49
8.3 Résolution des collisions . . . 50
8.3.1 Les méthodes directes . . . 50
8.3.2 Les méthodes indirectes . . . 51

9 Les arbres **53**
9.1 Introduction . . . 53
9.1.1 Définitions . . . 53
9.1.2 Notations et vocabulaire . . . 53
9.1.3 Applications . . . 53
9.2 Les arbres binaires . . . 54
9.2.1 Définition . . . 54
9.2.2 Représentation interne . . . 54
9.2.3 Exemple . . . 55
9.2.4 Parcours . . . 55
9.2.5 Ecriture sous forme totalement parenthésée . . . 56
9.2.6 Arbres binaires complets de profondeur p . . . 56
9.2.7 Représentation d'un arbre quelconque par un arbre binaire 57
9.3 Les arbres binaires de recherche . . . 57
9.3.1 Définition . . . 57
9.3.2 Recherche d'un élément dans un arbre binaire de recherche 58
9.3.3 Ajout d'un élément dans un arbre binaire de recherche . . 58
9.3.4 Suppression d'un élément dans un arbre binaire de recherche 59
9.4 Les arbres AVL . . . 60
9.4.1 Introduction . . . 60
9.4.2 Les rotations . . . 61
9.4.3 Recherche dans un arbre AVL . . . 62
9.4.4 Ajout dans un arbre AVL . . . 62
9.4.5 Suppression dans un arbre AVL . . . 67
9.4.6 Complexité au pire . . . 70
9.5 Les arbres (a-b) . . . 70
9.6 Les arbres 2-3-4 . . . 71

9.6.1 Introduction . . . 71
9.6.2 Représentation interne . . . 71
9.6.3 Recherche dans un arbre 2-3-4 . . . 73
9.6.4 Ajout d'un élément dans un arbre 2-3-4 . . . 73
9.6.5 Etude de l'éclatement d'un noeud . . . 74
9.7 Les arbres balancés ou B-arbres . . . 77
9.7.1 Définition . . . 77
9.7.2 Représentation interne et recherche d'un élément . . . 77
9.7.3 Ajout dans un B-arbre . . . 78
9.7.4 Suppression dans un B-arbre . . . 79
9.8 Les arbres B+ . . . 81
9.9 Le tri par tas (heapsort) . . . 83
9.9.1 Arbre binaire parfait . . . 83
9.9.2 Construction initiale du tas . . . 84
9.9.3 Reconstruction du tas après la suppression de son maximum 85
9.9.4 Algorithme final . . . 85
9.9.5 Complexité . . . 86
9.9.6 Exemple avec le tableau précédent . . . 86
9.10 Les arbres de classification . . . 88
9.10.1 Représentation aîné - benjamin . . . 89
9.10.2 Représentation polonaise inverse . . . 90

10 Représentation des graphes 93
10.1 Matrice booléenne . . . 93
10.2 Dictionnaire des arcs . . . 94
10.3 Dictionnaire des successeurs ou des prédécesseurs . . . 94
10.4 Exemple . . . 95

11 Les automates finis ou automates à états 97
11.1 Les expressions rationnelles . . . 97
11.2 Automate fini . . . 98
11.2.1 Algorithme général pour un automate déterministe . . . 98
11.2.2 Actions associées à l'automate . . . 99
11.3 Application à l'algorithmique . . . 100

12 Les tris externes 101
12.1 Introduction . . . 101
12.2 Construction des monotonies . . . 101
12.2.1 Monotonies de même taille . . . 101
12.2.2 Monotonies de taille variable . . . 101
12.3 Interclassement des monotonies . . . 103
12.3.1 Algorithme . . . 103
12.3.2 Placement des monotonies . . . 103
12.3.3 Le tri équilibré . . . 103
12.3.4 Le tri polyphasé . . . 103

Chapitre 1

Introduction

Le but de ce cours consiste, dans un premier temps à présenter les outils nécessaires pour manipuler des structures de données plus complexes. Dans un deuxième temps, ce cours présentera les listes chaînées, l'adressage dispersé les structures d'arbres. Dans ce polycopié les exemples seront donnés dans le langage Python. Le but de ce cours n'est pas l'apprentissage du langage Python. Ce langage ne sert qu'à coder les algorithmes. Dans la mesure du possible, nous écrirons des programmes en python procédural sans utiliser les concepts objets, nous serons toutefois amener à introduire les classes pour l'étude des listes chaînées et des arbres.

Chapitre 2

Algorithme et langages de programmation

2.1 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 langage algorithmique (ou pseudo code) ou l'écrire directement dans un langage de programmation, c'est le choix fait ici.

Un algorithme est constitué d'une combinaison de composants opératoires pouvant être :

- des composants élémentaires :
  - affectation,
  - entrées / sorties,
- des structures de base :
  - séquence,
  - conditionnelle,
  - itération.

2.2 Langages de programmation

Premiers langages : FORTRAN, COBOL et LISP

2.2.1 Différents types de langages

On peut regrouper les langages en plusieurs catégories :

- langages de bas niveau et 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, ...)

2.2.2 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. Il existe pour certains langages, à la fois des interpréteurs et des compilateurs. On citer quelques langages interprétés : Python, JAVA, Lisp, Prolog et quelques langages compilés : C/C++, Fortran, Cobol, Pascal.
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 et Julia : équivalents (un peu meilleurs) au C
- JavaScript et Go : entre 2 à 3 fois plus lents que le C
- Python : langage environ 15 fois plus lent que le 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)

Quelques langages utilisés dans cette formation : Python, C, JAVA, Prolog, SQL, C++, OCAML, langage de la bibliothèque R, Perl, ...

Chapitre 3

Rapide aperçu du langage Python

Nous ne présentons ici que les instructions et structures de bases du langage

3.1 Les types

Python possède tous les types de base classiques et trois types plus complexes : les listes, les tuples et les dictionnaires. Il est aussi possible de définir des structures plus complexes avec les classes, ce que nous éviterons dans ce cours qui n'est pas un cours sur la POO. Principaux types :

- le type entier (int) , exemples de constantes : 10, -12, +15, 0
- le type réel (float), exemples de constantes : 1.1, -1.5e-4, 1e9, -1.2, 0
- le type booléen (bool), deux valeurs de vérité : False et True
- le type caractère (str), exemples : "c", "1", ";", " "
- le type chaîne de caractères (str), exemples : "ceci est une chaîne", "" A noter que python ne différencie pas le type caractère du type chaîne de caractères
- le type liste (list), exemples : [1,2], [], [12, "a', [True, False], 15]
- les types tuples (tuple) ne seront pas utilisés dans ce cours et les types dictionnaires ne le seront que très peu..

3.2 Les expressions

3.2.1 Les expressions numériques

- Les constantes numériques et les variables de type réel ou entier 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 mod y et x**y sont aussi des expressions numériques
- L'ordre décroissant de priorité des opérateurs est :
  - L'ordre induit par les parenthèses.
  - Les opérateurs unaires (à un seul opérande).
  - l'opérateur exponentiel : **

  - Les opérateurs multiplicatifs : *, /, mod.
  - Les opérateurs additifs : + et -.
- Certains langages ont un typage fort et distinguent les expressions réelles des expressions entières.

3.2.2 Les expressions booléennes

Les règles sont analogues aux règles des expressions numériques. Les deux constantes booléennes sont **True** et **False**. Les opérateurs booléens sont par ordre de priorité décroissant : **not**, **and** et **or**. De plus la mise en relation de deux expressions de même type (sous réserve de l'existence d'une relation d'ordre) produit une expression booléenne. Les opérateurs relationnels licites pour les expressions de type simple ou de type chaîne sont : ==, !=, <, <=, > et >=.

3.2.3 Les listes Python

Les listes font partie des « conteneurs » python avec les tuples et les dictionnaires.
Les listes python peuvent 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 : maliste = [1, 2, 'a', True, [3.15], None,'Fin']. Ici : maliste[1] = 2, maliste[4] = [3.15], maliste[1 :] = [2, 'a', True, [3.15], None,'Fin'] maliste[2 :5] = ['a', True, [3.15]]
Opérations sur les listes (fonctions et méthodes) :

- concaténation : [1,2]+[3,4] = [1,2,3,4]
- ajout d'un élément : maliste.append(5)
- taille d'une liste : len(maliste)
- extraction de sous liste entre d et f : maliste[d,f+1]

3.2.4 Les pointeurs cachés (les références)

Le pointeurs n'existent pas en python mais sont sous-jacents. Les objets modifiables, comme les listes, sont repérés en interne par leur référence : un pointeur. Considérons le test suivant :

```
a = [1,2]
b = a
print("a=",a" b =",b) # affichage : a=[1, 2] b=[1, 2] : NORMAL
b.append(3) # maintenant b contient[1, 2, 3]
print("a=",a" b =",b) # affichage : a=[1, 2, 3] b=[1, 2, 3] : BIZARRE
```

En fait quand on écrit b = a, on ne recopie pas la valeur de a dans une autre variable b mais on recopie la référence de a dans b, par conséquent a et b renvoient sur le même emplacement mémoire. Si on souhaite recopier la valeur il faut utiliser copy (ou deepcopy pour les sous-listes).

```
import copy
a = [1,2]
b = copy.copy(a)
```

```
print("a=",a" b =",b) # affichage : a=[1, 2] b=[1, 2] : NORMAL
b.append(3) # maintenant b contient[1, 2, 3]
print("a=",a" b =",b) # affichage : a=[1, 2] b=[1, 2, 3] : CLASSIQUE
```

3.3 Les instructions

3.3.1 Affectation

<var> = <exp>
où <var> et <exp> représentent respectivement une variable et une expression de type compatible. Attention il n'y a pas de déclaration de variables en python (sauf la création à partir d'une classe).

3.3.2 Entrées / sorties

Entrée : <var> = input(<chaîne>)
Sortie : print(<exp>)

Exemple :

```
nom = input("Donner votre nom : ")
print("Le nom lu est : ", nom)
```

Attention : input renvoie toujours une chaîne de caractères qu'il faut ensuite traduire quand on souhaite avoir un nombre. Les conversions se font avec la fonction int pour les entiers et avec la fonction float pour les réels. Ces fonctions de conversion déclenchent une exception quand la chaîne n'est pas convertible (ne représente pas un nombre).

3.3.3 Instruction conditionnelle simple

```
if <exp>:
    <instructions1>
else:
    <instructions2>
```

où <exp> doit être une expression booléenne. La partie **else** est facultative. Les parties <instructions1>) et <instructions2>) sont délimitées par l'indentation.

3.3.4 Instruction conditionnelle généralisée

```
if <exp1>:
    <instructions1>
elif <exp2>:
    <instructions2>
...
elif <expn>:
    <instructionsn>
else:
<instructionsn+1>
```

Les <expr$_i$ > doivent être des expressions booléennes, la partie **else** est facultative. Les <instructionsi>) sont délimitées par l'indentation.

3.3.5 Instructions itératives

La simulation de boucle de traitement avec le <goto> ne sera pas évoquée ici, cette instruction n'existe d'ailleurs pas en python..

Instruction <while>

```
while <exp>:
    <instructions>
```

<exp> doit être une expression booléenne. On commence par évaluer la valeur de cette expression. si sa valeur est vrai, on exécute les instructions et l'on recommence le cycle. Si sa valeur est faux, on sort de la boucle, dans le cas où la valeur de l'expression est fausse dès la première évaluation, les instructions du corps de la boucle ne sont pas exécutées. Le corps de boucle est délimitée par l'indentation.

Instruction <for>

Cette instruction permet de parcourir un intervalle (entier) ou un conteneur, on rappelle que les conteneurs sont les listes, les tuples et les dictionnaires.

```
for <indice> in range(n):
    <instructions>
ou
for <élément> in <liste>:
    <instructions>
Exemples :
for i in range(10): #cas 1
    print("i=",i)
for j in range(1,10): #cas 2
    print("j=",j)
maliste=["toto","titi","tata"]
for e in maliste: #cas 3
    print("e=",e)
```

Dans le premier cas, on passe 10 fois dans la boucle et i varie de 0 à 9. Dans le deuxième cas, on passe 9 fois dans la boucle et j varie de 1 à 9. Enfin dans le troisième cas, on passe 3 fois dans la boucle avec e variant de "toto" à "tata".

3.3.6 Documentation

Il est très important de documenter un algorithme ou un programme. Cette documentation doit être externe (pour les projets) et interne. La documentation interne peut prendre plusieurs formes dont notamment l'auto-documentation et les commentaires. L'auto-documentation consiste à nommer les objets avec des noms explicites. Les commentaires sont obligatoires dans tous les algorithmes et programmes. D'un point de vue syntaxique, les commentaires se placent à partir de # jusqu'à la fin de la ligne.
L'indentation n'est pas une option en python car elle fait partie de la syntaxe du langage. Attention aux confusions entre les caractères espaces et les tabulation.

3.4 Objets de types composés

3.4.1 Les tableaux

Définition

Un tableau est une structure de données interne de taille finie dont tous les éléments qui le constituent sont de même type. On utilise des tableaux pour stocker des variables indicées. En python, il n'existe pas de tableau au sens habituel, par contre les listes sont indicées, leurs indices commencent toujours à 0.

3.4.2 Les structures ou enregistrements

Une structure est un agrégat de plusieurs objets de types éventuellement différents. Une structure est considérée comme une entité réunissant tous les champs qui la composent. Un champ peut être de n'importe quel type y compris de type structure ou tableau. Il est à noter que les définitions récursives sont généralement interdites dans les langages procéduraux. Il n'existe pas de type structure classique en python, par contre on peut les définir avec des classes.

3.5 Les fichiers

Un fichier est un ensemble d'informations stocké de façon permanente sur un support physique (disque). Il existe trois types d'organisation d'un fichier :

- organisation séquentielle : accès séquentiels,
- organisation relative : accès séquentiels et accès direct à un enregistrement à partir de son numéro,
- organisation indexée : accès séquentiels et accès direct indexé à un enregistrement à partir de sa clé (un champ particulier). Ce type de fichiers n'est plus utilisé (COBOL).

3.5.1 Les fichiers séquentiels sans structure

Ces fichiers contiennent des flots de caractères, ils sont aussi appelés fichiers textes. Les périphériques d'entrée et de sortie sont des fichiers textes particuliers.

Assignation

L'assignation sert à faire le lien entre le fichier physique sur disque et le fichier logique (une variable du programme). Il est indispensable de faire une assignation avant de pouvoir faire des entrées / sorties sur un fichier. En python l'assignation et l'ouverture d'un fichier sont faites en même temps.

Ouverture

L'ouverture sert à spécifier le sens des entrées / sorties (lecture, écriture ou mise à jour), l'organisation du fichier et le mode d'accès. Ces deux dernières

informations ne sont pas pertinentes pour ce cours où les fichiers manipulés seront tous d'organisation séquentielle.

En python :

```
<fichier> = open(<nom physique>,<sens>)
exemples :
monfichier = open("toto.txt","r"); # ouverture en entrée (lecture)
monfichier = open("toto.txt","w"); # ouverture en sortie (écriture)
```

fermeture

```
<fichier>.close()
Exemple : monfichier.close()
```

Instruction à faire à la fin du traitement du fichier ou avant de l'ouvrir avec d'autres attributs.

Lecture séquentielle

Les lectures sont assez spécifiques en python. En voici trois formes qui devraient suffire pour ce cours :

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

Lorsque le fichier est vide ou si on a atteint la fin du fichier, readline() renvoie la chaîne vide et readlines renvoie la liste vide.

Ecriture séquentielle

<nom logique>.write(<expression>)
C'est l'instruction la plus simple d'écriture dans un fichier mais elle est suffisante pour ce cours.

3.5.2 Les fichiers séquentiels structurés

Un tel fichier est constitué d'une séquence de composants de même type appelés articles ou enregistrements. Le type d'un enregistrement sera très souvent une structure. Ces fichiers ne peuvent être manipulés que par programme ou par des commandes spécialisées du système d'exploitation (pas par éditeurs de texte). Les instructions sont identiques à celle des fichiers text. La seule différence : on doit lire ou écrire tout un enregistrement à la fois.

3.6 Les sous-programmes

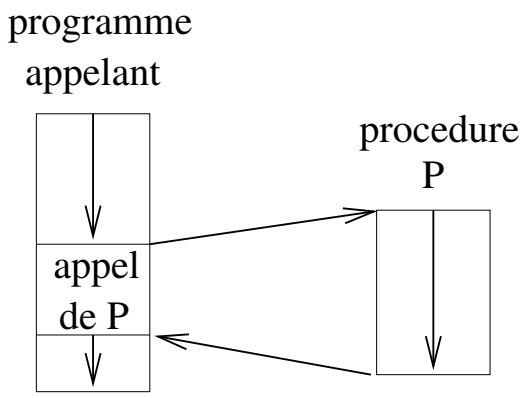
3.6.1 Les procédures

Une procédure est une partie de programme indépendante réalisant un travail précis résolvant un problème. L'intérêt des sous-programmes réside en plusieurs points :

- facilité de conception de logiciels importants par décomposition en sous problèmes,
- facilité de test,
- lisibilité du code.

Définition (déclaration) d'une procédure : grossièrement, cette opération consiste à donner un nom à un bloc d'instructions

Appel de procédure : à l'appel de la procédure dans le programme appelant, le contrôle d'exécution est passé à la procédure. Une fois l'exécution de la procédure terminée, le contrôle est rendu au programme appelant à l'instruction qui suit l'appel.



Passage de paramètres : les paramètres permettent la communication d'informations entre le programme appelant et le sous-programme. Les paramètres déclarés dans la procédure sont appelés **paramètres formels** et les paramètres fournis à la procédure par le programme appelant sont appelés **paramètres effectifs**.

Type des paramètres : dans la plupart des langages (C, JAVA, ...), on spécifie le type des paramètres. Ce n'est pas le cas en python qui dispose d'un système de typage dynamique. On peut vérifier le type d'une constante, d'une variable ou d'un objet avec les fonctions type et isinstance : type renvoie le type (int, float, bool, str, list, ...) alors que isinstance renvoie True si le premier paramètre est du type (ou une instance) du deuxième paramètre et renvoie False sinon. Outre leur forme syntaxique, type renvoie le type immédiat alors que isinstance permet de gérer les héritages. Son deuxième paramètre peut aussi être un tuple de types. Exemple pour tester si une variable a (déjà définie) est un nombre (entier ou réel) :

```
if (type(a) is int) or (type(b) is float):
    ...
# ou :
if isinstance(b,(float,int)):
    ...
```

Objets accessibles depuis un sous programme : les règles peuvent varier d'un langage à l'autre en fonction de la place syntaxique des sous programmes dans le programme :

- paramètres du sous programme,
- objets locaux (déclarés ou définis dans le sous programme),
- objets globaux (déclarés dans des blocs englobants : PASCAL), déclarés avant le main (C),
- objets externes (extern de C, COMMON de FORTRAN).

Logiquement, un sous programme devrait être indépendant de son contexte et n'utiliser que ses paramètres et des objets locaux.

syntaxe python : mécanismes d'appel de sous programme et de retour. On ajoute un entête aux instructions composant le corps de la procédure comportant le mot-clé def, le nom de la procédure et la liste de ses paramètres entre parenthèses et séparés par des virgules. Exemple :

```
def  affichcarres(n):
    for i in tange(1,n+1):
        print(i,"**2 = ",i*i)
```

3.6.2 Les fonctions

Une fonction est un sous programme qui renvoie un ou plusieurs résultats (comme par exemple la valeur absolue d'un nombre ou le pgcd de deux entiers naturels). L'utilisation et les mécanismes d'appel sont presque identiques à ceux des procédures. Nous nous limiterons à donner les différences.
Corps :

- il faut préciser la (ou les) valeur(s) résultat : ceci est réalisé par l'instruction return(<expression>). L'exécution de cette instruction termine la fonction,
- une fonction ne devrait jamais se terminer autrement qu'avec l'instruction retourner.

Appel :

- une fonction ne peut être appelée que dans une expression. L'appel peut apparaître à n'importe quel endroit où une constante du type de la valeur renvoyée est permise.
- certains langages (le C !) sont moins stricts mais il faut prendre de bonnes habitudes.

Exemple :

```
def pgcdnaif(a,b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

```
# ou en vérifiant les types :

def pgcdnaif1(a,b):
    if not(type(a) is int):
        print("ERREUR le premier paramètre n'est pas un entier")
        return None
    elif not(type(b) is int):
        print("ERREUR le deuxième paramètre n'est pas un entier")
        return None
    elif (a <= 0) or (b <= 0):
        print("ERREUR :au moins un des deux paramètres est <=0")
        return None
    else:
        while a != b:
            if a > b:
                a = a - b
            else:
                b = b - a
        return a
```

3.7 Les classes python

On introduit les classes afin de pouvoir appréhender plus facilement les listes chaînées et les arbres. On utilisera des classes simples, sans héritage en parlant le moins possible de programmation objet : ce n'est pas le but de ce cours mais ce sera fait en Java et C++ dans un autre cours. Une classe permet d'introduire un modèle de variables composé de plusieurs champs comme par exemple le nom d'une personne, son age, son numéro INSEE, ... Cela correspond un peu avec les structures en langage C mais les classes offrent beaucoup plus de possibilités. On prendra comme exemple une classe manipulant les nombres complexes :

```
class MesComplexes():
  def __init__(self,preel,pimag):
    self.reel = preel # partie réelle
    self.imag = pimag # partie imaginaire

# addition des 2 complexes x et y
  def plusc(x,y):
    # en décomposant les étapes
    s = MesComplexes(0,0)
    s.reel = x.reel+y.reel
    s.imag=x.imag+y.imag
    return s

# soustraction complexes x - y
  def moinsc(x,y):
    s = MesComplexes(x.reel-y.reel, x.imag-y.imag)
    return s

# multiplication des complexes x et y
```

```
  def multc(x,y):
    s = MesComplexes(0,0)
    s.reel = x.reel*y.reel - x.imag*y.imag
    s.imag = x.reel*y.imag + x.imag*y.reel
    return s

# division complexe x/y
# renvoie None si le module de y est nul
  def divc(x,y):
    modul2 = y.reel**2 + y.imag**2
    if modul2 < 1e-9: # seuil de précision
      return None
    else:
      dreel = (y.reel*x.reel + y.imag*x.imag)/modul2
      dimag = (y.reel*x.imag - y.imag*x.reel)/modul2
      s = MesComplexes(dreel,dimag)
      return s

# module d'un nombre complexe
  def module(self):
    return (self.reel**2+self.imag**2)**0.5

# multiplication d'un complexe par un scalaire
  def multscal(self,scal):
    ms = MesComplexes(scal*self.reel, scal*self.imag)
    return ms

# affichage d'un nombre complexe
  def affichc(self):
    print("partie réelle : ",self.reel, " partie imaginaire : ", self.imag)
```

3.7.1 Syntaxe

Syntaxiquement, une classe commence par le mot-clé *class* suivi du nom de la classe puis des parenthèses ouvrantes et fermantes (pas d'héritage) et du caractère deux point. Dans la classe on va trouver essentiellement des fonctions (appelées méthodes) travaillant sur des objets de la classe.

3.7.2 La méthode __init__

C'est une méthode particulière permettant de créer un objet (une variable) de la classe. Elle prend toujours le paramètre *self* représentant l'objet qui sera créé et des paramètres servant à initialiser les champs de l'objet.b Dans notre exemple, cette méthode a 2 paramètres outre self qui sont les parties réelle et imaginaire de l'objet à créer. Cette méthode initialise ou définit les champs d'un objet nombre complexe, ces champs sont appelés attributs, ici on a les attributs reel (partie réelle) et imag (partie imaginaire). Ainsi pour créer 2 nombres complexes, on peut écrire : a = Mescomplexe(4,3) et i = MesComplexes(0,1). Si l'on saisit b = i, on ne fait que recopier la référence de i dans b (i et b font référence à la même zone mémoire).

3.7.3 Les autres méthodes

Dans la classe, on trouve ensuite les 4 opérations sur les nombres complexe qui renvoient le résultat sous la forme d'un nombre complexe (sauf division par zéro où on retourne None. Ces fonctions s'appellent de manière habituelle.

La méthode module (d'un nombre complexe) n'a qu'un paramètre : self, l'objet sur lequel elle est appliquée. Pour l'utiliser on appliquera la syntaxe habituelle des méthodes, par exemple print(a.module()). C'est la même chose pour la dernière méthode d'affichage d'un nombre complexe.

La méthode multscal de multiplication par un scalaire possède deux paramètres, l'objet avec le paramètre self et le scalaire. L'utilisation de cette méthode suit la même syntaxe avec par exemple : a10 = a.multscal(10).

Chapitre 4

Les listes linéaires

4.1 Introduction

Une liste linéaire (aussi appelée séquence ou liste) est une structure de données constituée d'un nombre fini d'éléments de même nature. Chaque élément comprend une clef qui l'identifie et zéro ou plusieurs autres champs. Une séquence peut être stockée dans un fichier, un tableau, dans d'autres structures internes ou nulle part si elle est saisie au terminal et traitée au fur et à mesure. Le mode d'accès aux éléments d'une séquence peut être séquentiel ou sélectif. Quand il est séquentiel, on peut accéder à l'élément suivant et quelquefois revenir au premier élément (fichier). Lorsqu'il est sélectif, on peut accéder à n'importe quel élément à la seule condition de connaître son rang.

4.2 Principales opérations sur les séquences

DONNEES	OPERATION	RESULTAT
séquence et valeur de clé	recherche	élément ou vide
séquence et valeur de clé	suppression	nouvelle séquence
séquence et élément	modification	nouvelle séquence
séquence et élément	ajout	nouvelle séquence
séquence et séquence	fusion	nouvelle séquence
séquence et séquence	intersection	nouvelle séquence
séquence et séquence	différence	nouvelle séquence
séquence et séquence	différence symétrique	nouvelle séquence
séquence et critère de sélection	extraction	nouvelle séquence
séquence	tri	nouvelle séquence

4.3 Schémas d'algorithme

Les séquences traitées peuvent être regroupées en trois catégories :
- le nombre d'éléments est connu avant la boucle de traitement,
- le dernier élément est connu et il n'est pas traité,
- le dernier élément est connu et il est traité.

4.3.1 Les files à nombre d'éléments déterminé

Leur cardinal (noté N) est connu dès le début du traitement.

```
def fnd(...):
    <initialisation>
    <acquisition de n>
    for i in range(n):
        <acquisition de l'élément courant>
        <traitement de l'élément courant>
    <traitement final>
```

4.3.2 Les files à sentinelle exclue

Leur cardinal est inconnu en début de traitement. Elles se terminent par un élément spécial appelé "sentinelle". Cet élément ne doit subir aucun traitement. N permet de calculer le cardinal de la file si on en a besoin.

```
def fse(...):
    <initialisation>
    n = 0
    <acquisition du premier élément de la file>
    while <élément courant> != sentinelle:
        n = n+1
        <traitement de l'élément courant>
        <acquisition de l'élément suivant>
    <traitement final>
```

4.3.3 Les files à sentinelle inluse

Leur cardinal est aussi inconnu en début de traitement. Elles se terminent par une sentinelle qui doit ici être traitée. N permet de calculer le nombre d'éléments si on en a besoin.

```
def fsi()
    <initialisation>
    n = 0
    élément = None #sauf si sentinelle = None
    while élément != sentinelle:
        n = n+1
        <acquisition de l'élément courant>
        <traitement de l'élément courant>
    <traitement final>
```

4.4 Notations pour les algorithmes

- Type d'un élément de séquence : entier mais cela pourrait être n'importe quel type sur lequel existe une relation d'ordre
- Type d'une séquence : séquence = liste python
- Cardinal d'une séquence : n ou ni

4.5 Algorithme de recherche

4.5.1 Recherche séquentielle

On suppose la séquence dans un tableau non trié. On va faire en sorte de toujours trouver l'élément recherché. Pour ce faire, on va l'ajouter en sentinelle à la fin du tableau.

```
def rech_seq(tab, x):
   # tab : tableau (liste python) contenant la séquence
   # x : élément recherché
   # valeur renvoyée : indice de l'élément (-1 si absent)
   # n : cardinal de la séquence
     n = len(tab)
     tab.append(x) # mise en place de la sentinelle
     i = 0
     while tab[i] != x:
         i = i+1
   # conclusion
     if i < n :
         return i
     else:
         return -1 # on a trouvé la sentinelle
```

4.5.2 Recherche dichotomique

Le tableau contenant la séquence doit être obligatoirement trié (on suppose en ordre croissant). Le principe de cette méthode consiste à examiner l'élément médian de la séquence, trois cas sont possibles :

- l'élément médian est l'élément recherché : succès,
- l'élément médian est strictement supérieur à l'élément recherché : on recommence sur la première moitié de la séquence,
- l'élément médian est strictement inférieur à l'élément recherché : on recommence sur la seconde moitié de la séquence.

L'algorithme se termine soit en cas de succès, soit en cas d'échec lorsque la sous séquence à examiner devient vide.

```
def rech_dicho(tab, x):
   # tab: tableau contenant la séquence
   # x : élément recherché
   # valeur renvoyée : indice de l'élément (-1 si absent)

   # n : cardinal de la sequence
   # bi : borne inférieure de la sous séquence
   # bs : borne supérieure
   # mi : milieu
     bi = 1
     bs = n-1
     while bi <= bs:
         mi = (bi+bs)//2 #division entière
         if tab[mi] > x:
```

```
            bs = mi-1
        elif tab[mi] < x:
            bi = mi+1
        else:
            return mi
    # ftq
    # on ne passe ici qu'en cas d'échec
    return -1
```

4.5.3 Recherches auto adaptatives

Le principe de ces méthodes consiste à placer en début de séquence les éléments les plus souvent recherchés. Ces méthodes peuvent être plus ou moins sophistiquées, la plus simple consistant à placer en tête le dernier élément recherché, la pus compliquée consistant quant à elle à tenir compte des fréquences de recherche.

4.5.4 Complexité moyenne

Méthode	complexité en temps	complexité en place
Recherche séquentielle	O(N)	N
Recherche dichotomique	0(log(N))	N

4.6 Ajout d'un élément

Il s'agit d'ajouter un élément à sa place dans une séquence triée en ordre croissant. Les autres ajouts à la fin et au début étant évidents (attention toutefois au décalage en cas d'ajout en tête).

```
def ajout_trie (tab, x):
  # tab : tableau (liste python) contenant la séquence
  # x : élémentàa insérer dans la séquence

    ind = len (tab)-1 # indice du dernier élément avant l'ajout
    tab.append(x) # ajout de l'élément en dernière position
                  # uniquement pour augmenter la taille de la liste
  # décalage éventuel et recherche de l'emplacement
    while(ind >= 0) and (tab[ind] > x):
        atb[ind+1] = tab[ind]
        ind = ind-1
     ftq
  # insertion en position ind+1
     tab[ind+1] = x
```

Une recherche dichotomique est possible pour trouver le rang où insérer le nouvel élément mais le gain est faible car le décalage doit être fait séquentiellement.

4.7 Interclassement de deux séquences triées

On suppose que les deux séquences sont respectivement dans les fichiers de noms physiques "seq1.txt" et "seq2.txt". Le résultat sera placé dans le fichier "seq3.txt".

```
# Interclassement de 2 fichiers
def interclasser():
    # ouverture et assignation des fichiers
    f1  = open("seq1.txt","r")
    f2 = open("seq2.txt","r")
    f12 = open("seq3.txt","w")

    article1 = f1.readline() # premier élément de f1
    article2 = f2.readline() # premier élément de f2

    # boucle principale
    # en fin de fichier, readline renvoie la chaîne vide
    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
```

4.8 Intersection de deux séquences triées

Le principe est le même que pour la fusion. On n'ajoute un article dans F3 que lorsqu'il est présent dans F1 et F2 (pas de déversement à la fin). Le principe est identique pour les opérations de différence et de différence symétrique.

4.9 Les tris internes élémentaires

Les tris seront fait en ordre croissant. Les tris étudiés dans cette partie sont peu performants. Le tri rapide (quicksort) et le tri par tas (heapsort), beaucoup plus efficaces en général, seront étudiés après la récursivité et les arbres (pour le heapsort).

4.9.1 Tri par sélection du minimum

Le principe est simple : on recherche le minimum, on le place au début de la séquence puis on réitère sur la fin de la séquence.

```
# Tri par sélection du minimum
# Paramètre t : tableau (liste) à trier
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
# Fin de la fonction trisel
```

4.9.2 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.

```
# Tri à bulles (un peu optimisé)
# Paramètre t : tableau (liste) à trier
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
                t[j-1], t[j] = t[j], t[j-1] # consécutif dans le mauvais ordre
                permut = True
            # end if
            j = j-1
        # end while
        i = i+1
    # end while
```

```
# Fin de la fonction tribulles
```

4.9.3 Tri par insertion séquentielle

On insère l'élément courant à sa place dans la partie triée de la séquence. Pour le faire, on applique l'algorithme d'ajout d'un élément dans une séquence triée. Une recherche dichotomique est possible mais le gain est faible (décalages).

```
# Fonction de tri par insertion séquentielle
# Paramètre t : tableau (liste) à trier
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
 # Fin de la fonction triinsert
```

4.9.4 Evaluation de ces méthodes

Ces méthodes procèdent par comparaison de deux éléments successifs et possèdent la propriété suivante : après le k^{ieme} placement, les k plus petits éléments sont à leur place définitive. Ces méthodes sont peu efficaces et ne doivent pas être utilisées sur de gros tableaux.

algorithme	nb moyen de comparaisons	nb max de comparaisons	nb moyen de transferts	nb max de transferts
tri par sélection	N(N-1)/2	N(N-1)/2	3(N-1)	3(N-1)
tri à bulles	N(N-1)/2	N(N-1)/2	3N(N-1)/4	3N(N-1)/2
insertion séq.	N(N+3)/4-1	N(N-1)/2-1	N(N+7)/4-2	N(N+3)/2-2
insertion dich.	0(N.log(N))	O(N.log(N))	N(N+7)/4-2	N(N+3)/2-2

4.9.5 Tri Shell

C'est une méthode de tri par insertion où on insère un élément à sa place dans une sous liste déjà triée. Sa particularité vient du fait que les éléments d'une même sous liste sont distants d'un pas successivement égal à N/2, N/4, ..., 1. Cette méthode est souvent employée dans des programmes écrits dans des langages non récursifs. Sa complexité en nombre de comparaisons est $O(N^{1.5})$.

```
# Fonction de tri par insertion séquentielle
# Paramètre t : tableau (liste) à trier
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
 # Fin de la fonction triinsert
```

4.9.6 Tris sans comparaison entre éléments

Dans le cas de clefs discrètes et bornées, il existe des méthodes de tri ne nécessitant aucune comparaison et testant une seule fois la valeur de chaque élément. Dans ce cas, il est préférable d'employer une méthode spécifique de tri. On peut, par exemple, citer le tri par paquets : on crée un paquet (une liste python, par valeur de clé puis on concatène les paquets.

```
# Tri par paquets
# paramètre 1 (t) : tableau/liste 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 y a t il un problème ici ?
    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
```

On peut aussi compter le nombre d'occurrences de chaque clé :

```
# Tri par comptage
# paramètre 1 (t) : tableau/liste 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
```

Chapitre 5

La récursivité

5.1 Exemples

5.1.1 Factorielle n (n !)

```
def fact(n):
    if n < 0: #paramètre invalide
        return None
    elif n  <= 1:
        return 1
    else:
        return n*fact(n-1)
```

Il faut toujours préciser l'appel principale d'une fonction récursive. Appel principal : print(fact(3))

5.1.2 Suite de Fibonacci

```
def fib(n):
    if n < 0: # paramètre invalide
        return None
     elif n <= 1:
         return n
     else:
         return fib(n-1)+fib(n-2)
```

Appel principal : print(fib(5)). La complexité de cet algorithme est catastrophique avec O(2**n) alors que l'algorithme itératif est en O(n). Ceci n'est pas du à la récursivité mais à l'algorithme qui calcule de nombreuses fois les mêmes nombres.

5.1.3 Recherche dichotomique

```
# fonction récursive :
# bi : borne inférieure de l'intervalle de recherche
# bs : borne supérieure de l'intervalle de recherche
# t : tableau (liste python) dans lequel on fait la recherche
```

```
# x : valeur recherchée
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

# Fonction contenant l'appel principal :
def dicho2(t, x):
    n = len(t)
    return dicorec(0, n-1, t, x)
```

5.2 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 existe de la récursivité croisée : A0 $\rightarrow$ A1 $\rightarrow$ ...$\rightarrow$ Ai $\rightarrow$ ...$\rightarrow$ An $\rightarrow$ A0
- l'intérêt de la récursivité réside dans la facilité de conception des algorithmes. Au niveau de la programmation il est souvent préférable de l'éliminer, car elle est souvent gourmande en temps et en place mémoire. Certains 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).

5.3 Mécanismes

5.3.1 A l'appel récursif

- sauvegarde des variables locales dans une pile,
- mécanisme normal d'appel de sous programme.

5.3.2 Au retour d'un sous programme récursif

- mécanisme normal de retour de sous programme,

– restitution des objets sauvegardés dans la pile.

5.3.3 Quand le langage n'est pas récursif

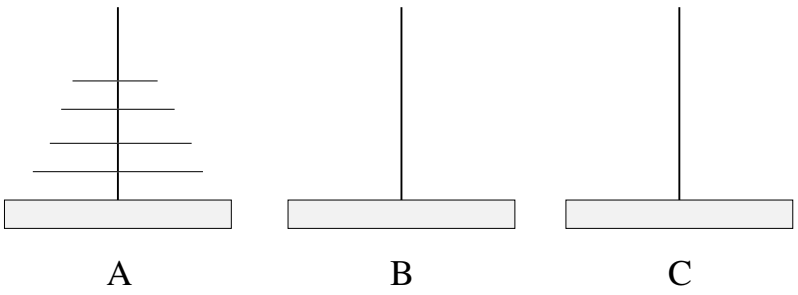
Dans ce cas, il faut gérer :
– la pile de sauvegarde,
– la pile des adresses de retour.

5.4 Applications

5.4.1 Les tours de Hanoi

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 :



Procédure récursive de résolution

```
# n : nombre de disques
# a : piquet d'origine
# c : piquet destination
# b : piquet de travail
def hanoi(n,a,c,b):
    if n>1:
       hanoi(n-1,a,b,c)
    print("Déplacer ",n," de ",a," vers ",c)
    if n>1:
       hanoi(n-1,b,c,a)
```

Appel principal : hanoi(5,'A','C','B')

5.4.2 Le tri rapide (quicksort)

Principe

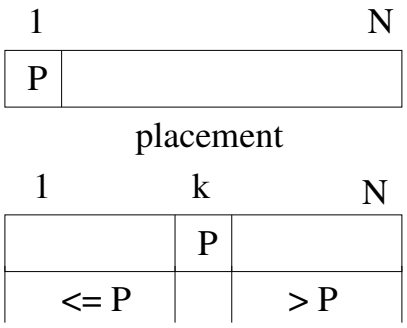
C'est un tri interne procédant par dichotomie :

- 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 réduits à un élément ou vides. Dans la pratique, on arrête ce processus de tri d'un sous tableau dès que son cardinal devient inférieur à un seuil de quelques dizaines. On applique alors une méthode de tri élémentaire, le tri rapide étant peu performant sur les petits ensembles.

Dichotomie

- choix d'un élément dans le tableau : **le pivot**,
- tous les éléments du premier sous tableau seront inférieurs ou égaux au pivot,
- tous les éléments du second sous tableau seront strictement supérieurs au pivot.

Le choix du pivot est très important. Un pivot "parfait" provoquerait une dichotomie en deux sous tableaux de même effectif (à un élément près). Un mauvais choix peut engendrer un sous tableau vide ce qui provoquerait une forte baisse de performance de l'algorithme (voir complexité au pire). Le choix du pivot doit se faire en temps constant par rapport au nombre d'éléments à trier. Cette contrainte ne permet pas l'élaboration de technique sophistiquée. En général, on prend comme pivot le premier élément du tableau (ce qui est fait dans les algorithmes qui suivent). Une méthode plus judicieuse consiste à considérer le premier élément, l'élément du milieu et le dernier élément du tableau : on sélectionnera alors l'élément médian (au sens de la relation $\leq$).



On remarquera, qu'une fois la dichotomie réalisée, que le pivot est à sa place définitive.

Algorithme de dichotomie

Cet algorithme est non récursif et nécessite la pose d'une sentinelle égale à $+\infty$ à l'indice N+1. Les notations utilisées sont les mêmes que celles utilisées pour les tris élémentaires.

```
# dichotomie d'un sous-tableau en fonction d'un pivot pour le tri rapide
# Paramètres : bi - entier - E - borne inférieure du sous-tableau
#              bs - entier - E - borne supérieure du sous-tableau
#              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 du sous-tableau : indice bi
    n = len(t)
    # i : indice montant
    # k : indice descendant
    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]  # échange de 2 variables
            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
```

Algorithme de tri

```
# Fonction récursive de tri
# Paramètres : bi - entier - E - borne inférieure du sous-tableau à trier
#              bs - entier - E - borne supérieure du sous-tableau à trier
#              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
```

Performances

En temps (nombre de comparaisons) : O(N.log(N)) en moyenne mais $O(N^2)$ au pire.
En place (pile de sauvegarde) : entre O(N) et O(log(N)), ce n'est donc pas un tri sur place.

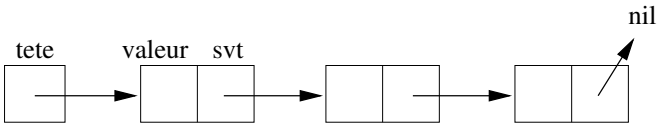
Chapitre 6

Les listes chaînées

6.1 Les listes simplement chaînées

6.1.1 Structure

Une liste chaînée constitue un autre procédé de représentation interne des séquences. Dans une liste simplement chaînée, chaque élément est relié à son successeur par un chaînage. Un nouveau champ va apparaître pour contenir l'adresse de l'élément suivant.



6.1.2 Représentation interne

Avec des listes

Les langages procéduraux ne proposent de rarement un type listes. D'autres langages comme Prolog ou Lisp ont des listes uniquement décomposable entre le premier élément et la fin de liste mais cela est suffisant pour coder une liste (chaînée). On peut accéder à n'importe quel élément en parcourant la liste ou en utilisant une fonction (Lisp) ou un prédicat (Prolog) prédéfini. Python propose un type liste où n'importe quel élément est accessible à partir de son indice. Ces types listes n'ont pas besoin de chaînage explicite, c'est le compilateur ou l'interpréteur qui s'en charge.

Avec des tableaux

On peut représenter une liste chaînée avec un tableau d'enregistrements. Chaque enregistrement contiendra un élément de la séquence. Le dernier champ de l'enregistrement recevra soit l'adresse de l'élément suivant (son indice dans le tableau), soit une adresse invalide (par exemple -1) pour indiquer qu'il s'agit du dernier élément de la séquence. De plus une variable entière indiquera l'adresse (l'indice) du premier élément de la séquence dans le tableau.

Inconvénients : il est impératif de connaitre dès le début du traitement le nombre d'éléments de la séquence (ou un majorant raisonnable de ce nombre). La gestion des places libres après un certain nombre d'ajouts et de retraits n'est pas non plus très simple. On préférera donc, si le langage de programmation le permet (C, PASCAL, ...), une représentation avec des pointeurs.

Avec des pointeurs

C'est la méthode habituelle avec tous les langages procéduraux proposant des pointeurs. Un pointeur est une adresse en mémoire centrale qui permet de repérer une zone mémoire contenant un maillon (pour une liste chaînée). Un maillon est ici constitué de deux champs : le valeur de l'élément et l'adresse de l'élément suivant. Elle est plus coûteuse que la méthode utilisant un tableau (place des pointeurs) mais est beaucoup plus souple et plus rapide en cas de modification de la liste. Cette méthode sera utilisée au prochain semestre avec le langage C.

Avec des classes

Il est possible de représenter les listes chaînées avec des classes : une classe Maillon qui contiendra une cellule et une classe ListeCh qui contiendra l'adresse de la première cellule de la liste. Cette technique est très proche de celle utilisant des pointeurs mais ici on ne manipule pas de pointeurs mais des références !
On considère maintenant que chaque maillon est constitué de la valeur de l'élément et de la référence de l'élément suivant (None si c'est le dernier élément).

6.1.3 Définition d'une liste avec des classes

```
class Maillon():
  def __init__(self, valeur=None, suivant=None):
    self.val = valeur    # de n'importe quel type (ou presque
    self.svt = suivant   # objet Maillon
# Fin de la classe "Maillon"

class ListeCh():
  def __init__(self, premier=None):
    self.tete = premier # objet Maillon
    # on pourrait aussi mettre le dernier élément de la liste
    # et le nombre d'éléments de la liste

  # affichage itératif des éléments d'une liste
  def affichLch(self):
      if self.tete:           # équivalent à self.tete != None
          courant = self.tete
          while courant:      # équivalent à courant != none
              print(courant.val,end='\t')
              courant = courant.svt
      print("\n")
  # Fin la méthode "affichLch"
```

```
  # ajout d'un élément x en tête
  def ajoutTete(self, x):
      mx =  Maillon(x,self.tete)
      self.tete = mx
  # Fin de la méthode "ajoutTete"

  # ajout d'un élément x à la fin
  def ajoutFin(self, x):
       mx =  Maillon(x)
       if not self.tete:   # liste vide
         self.tete = mx
       else:
         lpred = self.tete
         lsvt = lpred.svt
         while lsvt:
           lpred = lsvt
           lsvt = lsvt.svt
         # insertion après lpred
         lpred.svt = mx
  # Fin de la méthode "ajoutFin"

  # ajout d'un élément x dans une liste triée
  def ajoutTrie(self, x):
       mx =  Maillon(x)
       if not self.tete:        # liste vide
         self.tete = mx
       elif x <= self.tete.val: #insertion en t^te
         mx.svt = self.tete
         self.tete = mx
       else:                    # cas général
           lpred = self.tete
           lsvt = lpred.svt
           while (lsvt) and (x > lsvt.val):
             lpred = lsvt
             lsvt = lsvt.svt
           # insertion entre lpred et lsvt
           mx.svt = lsvt
           lpred.svt = mx
  # Fin de la méthode "ajoutTrie"
# Fin de la classe "ListeCh"
```

La classe Maillon contient deux attributs : la valeur stockée dans la cellule (val) et la référence de la cellule suivante (svt). La classe ListeCh ne contient qu'un attribut : la référence du premier maillon de la liste (tete).

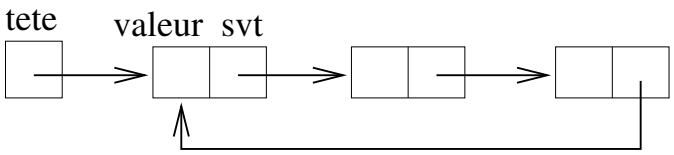
6.1.4 Exercices

- parcours de listes,
- ajout en position k,
- suppression en position k,

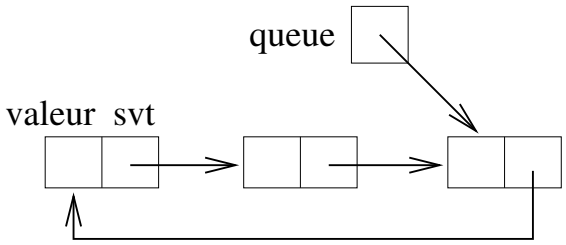
– tri d'une liste,
– interclassement de deux listes triées.

6.2 Les liste circulaires

Le principe est le même, mais le champ successeur du dernier élément contient l'adresse du premier élément.



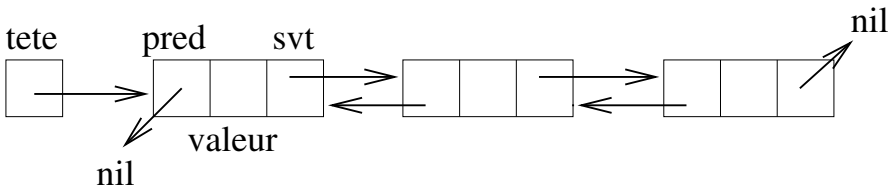
Variante : il peut être préférable de conserver l'adresse du dernier élément de la liste plutôt que l'adresse du premier élément. ceci assure un accès en temps constant à la fois au dernier élément et au premier élément de la liste.



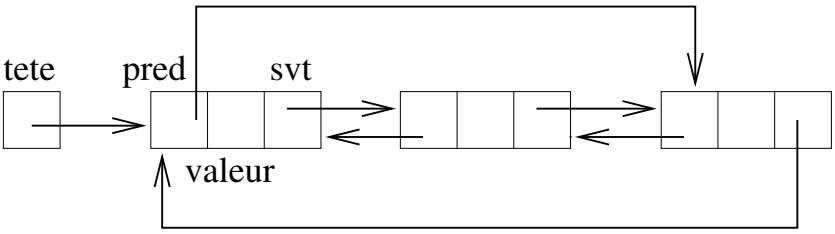
6.3 Les listes doublement chaînées

Chaque élément d'une liste doublement chaînée possède à la fois un champ contenant l'adresse de son successeur et un champ contenant l'adresse de son prédécesseur.

Liste doublement chaînée non circulaire :



Liste doublement chaînée circulaire :



Chapitre 7

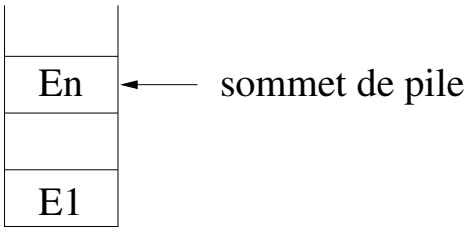
Les piles et les files

Les piles et les files d'attente sont des structures de données internes constituées d'un nombre fini éléments. La particularité de ces structures concerne leur mode d'accès.

7.1 Les piles

7.1.1 Mode d'accès aux éléments

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



7.1.2 Opérations sur les piles

Les seules opérations permises sont dans le tableau suivant :

Données	Opération	Résultat
vide	création	pile vide
pile et élément	ajout	pile
pile	suppression 1	pile
pile	suppression 2	pile et élément
pile	consultation	élément
pile	test pile vide	booléen

7.1.3 Représentation interne

Il est possible d'utiliser des listes chaînées, des tableaux ou des listes pour les langages ayant un tel type. La représentation avec des tableaux est simple et correspond au schéma précédent où le sommet de pile est un entier. L'inconvénient des tableaux vient de leur déclaration et la nécessité de devoir prévoir le nombre maximum d'éléments. Il s'agit néanmoins d'un codage très efficace en place et en temps pour les piles
Avec la représentation avec des listes, un élément de la pile sera constitué d'une structure contenant deux champs. Le premier champ sera du type des valeurs à empiler, le deuxième sera un pointeur contenant l'adresse de l'élément du dessous dans la pile. Ce type de codage évite le problème de la connaissance du nombre maximum d'éléments mais est assez lourd et nécessite beaucoup de place pour stocker les pointeurs.
Les listes (Python, Caml, Prolog) sont également bien adaptées au stockage d'une pile. La complexité en temps et en place dépend de l'implémentation des listes dans le langage.

7.1.4 Algorithmes

Les algorithmes sont immédiats avec les listes python en utilisant les méthodes append et pop sur les listes.

Avec Python

```
# création de pile vide
def creerpilevide():
  return []

# test de pile vide
def estvide(p):
  return (p == [])

# ajout de l'élément x à la pile p
def empiler(p, x):
  p.append(x)
  return p

# suppression du sommet de pile
# valeurs renvoyées : sommet de pile et nouvelle pile
# ou None si la pile était vide
def depiler(p):
  if estvide(p):
    return None
  else:
    x = p.pop()
    return x, p
```

Avec Prolog : c'est magique

Les listes prolog ont la même syntaxe que les listes python. Par contre, elles ne sont pas indicées. Pour couper une liste, on peut simplement séparer le premier élément de la fin de la liste avec l'opérateur "|". Pour cette raison, on placera le sommet de pile en tête de la liste.

```
/* crée une liste vide (paramètre libre)
ou teste si une liste est vide (paramètre lié) */
vide([]).
/* empile, dépile ou cherche le sommet de la pile
   en fonction des paramètres liés et libres */
empile_depile_sommet(Element, Pile, [Element|Pile]).
```

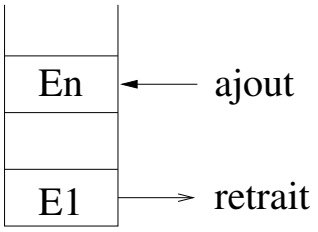
7.1.5 Utilisation des piles

Les piles sont essentiellement utilisées pour les versions itératives des algorithmes récursifs.
Exercice : Evaluation d'une expression arithmétique suffixée.

7.2 Les files d'attente

7.2.1 Mode d'accès aux éléments

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



7.2.2 Opérations sur les files d'attente

Les seules opérations permises sont dans le tableau suivant :

Donnée	Opération	Résultat
vide	création	file vide
file et élément	ajout	file
file	suppression 1	file
file	suppression 2	file et élément
file	consultation	élément
file	test file vide	booléen

7.2.3 Représentation interne

Il est possible d'utiliser des listes chaînées, des tableaux ou des listes python. Du fait de leur mode de décomposition les listes Lisp ou Prolog sont mal adapter dans ce cas. Outre le problème de la déclaration de leur taille, les tableaux ne sont pas très simples à gérer.
Avec la représentation avec des listes chaînées, un élément de la file sera constitué d'une cellule contenant deux attributs. Le premier attribut sera du type des valeurs à stocker, le deuxième sera la référence de l'élément suivant dans la file. Afin d'accéder rapidement au dernier et au premier élément, nous utiliserons une liste chaînée circulaire.

7.2.4 Algorithmes avec des liste Python

```
# création d'une file d'attente vide
def creerfilevide():
  return []

# test de file d'attente vide
def estvide(filatt):
  return (filatt == [])

# ajout de l'élément x à la file d'attente filatt
def enfiler(filatt, x): # vous pouvez changer le nom !
  filatt.append(x)
  return filatt

# suppression d'un élément dans la file d'attente filatt
# retour : élément supprimé et nouvelle file ou None
def defiler(filatt): # vous pouvez changer le nom !
  if estvide(filatt):
    return None
  else:
    pre = filatt[0]
    f = filatt[1:]
    return pre, f

# accès au premier élément de la file d'attente filatt
def premier(filatt):
  if estvide(filatt): # file vide
    return None
  else:
    return filatt[0]
```

7.2.5 Utilisation des files d'attente

Les files d'attente sont très utilisées en informatique (systèmes d'exploitation, ...).

Chapitre 8

L'adressage dispersé

8.1 Principe

l'idée 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 e_i $(1 \leq i \leq N)$ d'un ensemble E, une valeur $f(e_i)$ qui sera l'adresse de e_i dans une table à M places ou M entrées. La fonction f prend ses valeurs dans l'ensemble $\{1, 2, \ldots, M\}$. 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, $\exists e_i, e_j \in E, e_i \neq e_j / f(e_i) = f(e_j)$: il se produit une collision (deux éléments différents ont la même adresse). Nous verrons par la suite comment résoudre les collisions.

8.2 Fonction de dispersion

8.2.1 Définition

Une fonction de dispersion f, est une fonction sur l'ensemble E des éléments à valeurs sur un intervalle de l'ensemble $\mathbb{N}$ des entiers naturels. La quantité $f(e_i)$ est appelée valeur de dispersion de l'élément e_i. 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 : $(e_i R e_j) \Leftrightarrow (f(e_i) = f(e_j))$. R est une relation d'équivalence à M classes, chaque classe correspond à une entrée de la table.

8.2.2 Choix de la fonction de dispersion

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). Voici quelques méthodes :

Fonction habituelle pour les chaînes de caractères

```
def hcode(m, mot):
# m : nombre d'entrées de la table
# mot : chaine dont on souhaite calculer la valeur de dispersion
# ord : fonction python renvoyant le code asccii d'un caractère
# % : reste de la division entière
  somme = 0
  for c in mot: # pour tous les caractères du mot
    somme = somme + ord(c)
  return somme % m
```

Extraction

On extrait certains bits de la clé de l'élément. Si $M = 2^p - 1$, un mot de p bits fournit une valeur de dispersion. Le principal danger de cette technique réside dans la sélection de bits non significatifs.

Compression

Cette technique permet l'emploi de tous les bits de la clé. On les comprime sur p positions binaires en utilisant le ou logique ou le ou exclusif.

Multiplication par un réel

La valeur de dispersion d'un élément e_i sera $ent(fract(\lambda.e_i).M) + 1$ où :
- ent est une fonction retournant la partie entière,
- fract est une fonction retournant la partie fractionnaire,
- λ est un réel compris entre 0 et 1.

D'après des études théoriques, il est préférable de prendre $\lambda = \frac{\sqrt{5}-1}{2}$ ou $\lambda = \frac{3-\sqrt{5}}{2}$.

8.3 Résolution des collisions

Il existe deux types de méthodes : les méthodes directes qui calculent une nouvelle adresse et les méthodes indirectes qui réalisent un chaînage sur les éléments d'une même classe d'équivalence de la relation R.

8.3.1 Les méthodes directes

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

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

$e_i \rightarrow$ permutation de $\{1, \ldots, M\}$

$F(e_i) = \{f_1(e_i), \ldots, f_j(e_i), \ldots, f_M(e_i)\}$ où les $f_j(e_i)$ sont deux à deux distinctes.

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

Algorithme général de recherche / ajout

```
def ajout(table, element) :
     j = 1
     adr = f1(elem)
     while (table[adr]) != vide) and (table[adr] != element):
          j = j+1
          adr = fj(element)
     ftq
     if table[adr] == element:
         <traitement>
      else:
         <ajout>
# fin ajout
```

Le hachage linéaire

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

Evaluation

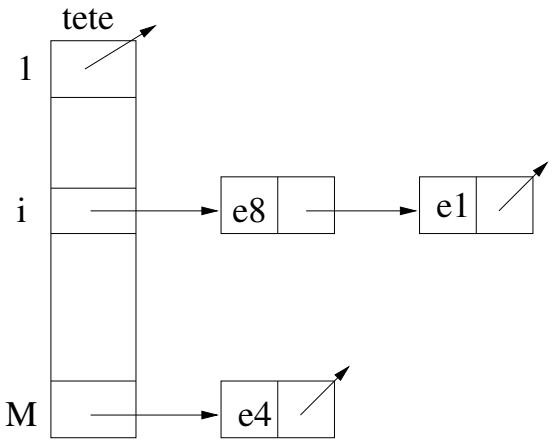
Les méthodes directes sont efficaces lorsque le nombre de collisions est faible et nécessitent moins de place que les méthodes indirectes (chaînage). Le principal inconvénient est le risque assez important de collisions secondaires (emplacement de la table occupé par un élément d'une autre classe d'équivalence). 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.

8.3.2 Les méthodes indirectes

Dans ce type de méthodes, on réalise un chaînage des collisions : on crée une liste chaînée par classe d'équivalence de la relation R. Indépendamment de l'implantation, tableaux ou pointeurs, il existe deux principales structures de données. Les listes peuvent être triées ou non. Ces méthodes sont plus intéressantes que les méthodes de résolution directes quand le nombre de collisions augmente.

Tableau de listes

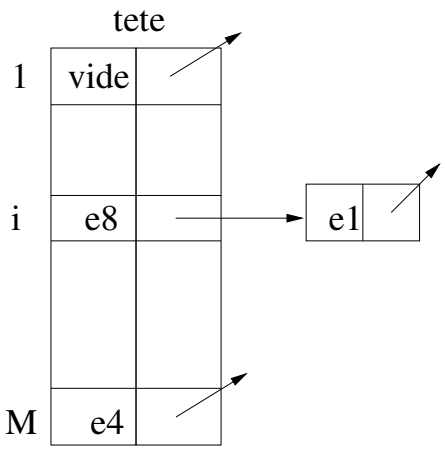
C'est la technique la plus utilisée. On dispose d'un tableau de M pointeurs (tete) dont l'élément d'indice i contient l'adresse du premier élément de la classe de valeur de dispersion i.



Les algorithmes de recherche, d'ajout et de suppression sont les mêmes que ceux utilisés pour les listes chaînées. Si f est la fonction de dispersion, l'adresse du premier élément de la liste lors du traitement de l'élément e est tete[f(e)].

Variante

La tableau *tete* contient deux champs : un champ de type élément et un de type pointeur vers l'élément suivant. Cette technique est avantageuse quand il y a peu de collisions (gain de place). Néanmoins, elle nécessite la présence d'un marqueur pour indiquer si une liste est vide ou si elle ne possède qu'un seul élément (pointeur égal à nil dans les deux cas).



Chapitre 9

Les arbres

9.1 Introduction

9.1.1 Définitions

Un arbre est un graphe sans cycle tel que :
- il existe un et un seul sommet où n'arrive aucun arc : la racine,
- 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 noeuds ou encore sommets organisé de façon hiérarchique à partir d'un noeud spécial sans prédécesseur, la racine.

9.1.2 Notations et vocabulaire

Un noeud y est dit fils du noeud x (x est le père de y) si et seulement si il existe un arc allant de x à y.

Tout noeud (à l'exception de la racine) possède exactement un père.

Un noeud peut posséder zéro, un ou plusieurs fils. Un noeud sans fils est appelé une feuille.

La profondeur d'un noeud est définie comme suit :
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) = $max\{prof(x)/x \in A\}$

9.1.3 Applications

Les applications des arbres sont très nombreuses dans tous les domaines :
- arbres généalogiques (qui sont rarement des arbres),
- tournois ou compétition de type "coupe",
- organisation interne de fichiers,
- organisation des fichiers dans les systèmes d'exploitation,
- structures internes de représentation d'ensembles,

- sémantique,
- mathématique,
- modélisation de problèmes,
- ...

9.2 Les arbres binaires

9.2.1 Définition

Un arbre binaire est un arbre dont tout noeud possède au plus deux fils. Un arbre binaire peut aussi être défini [3] comme étant soit vide soit de la forme <x,B1,B2> où B1 et B2 sont des arbres binaires disjoints et x un noeud appelé racine.

Notations : 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).

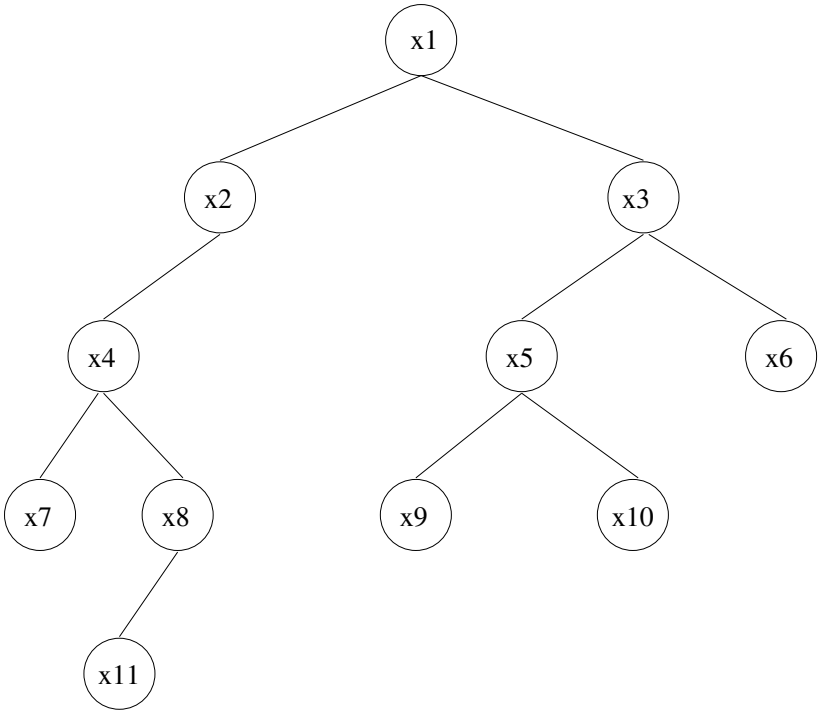
9.2.2 Représentation interne

Il existe de nombreuses manières de représenter un arbre. Un arbre est un graphe particulier : on peut utiliser les techniques de représentation de graphes (peu efficace pour les arbres). On peut également le représenter de manière contiguë de diverses façons dans un tableau (pour les arbres binaires complets ou parfaits ou encore pour les arbres de classification). Il est possible de représenter des arbres binaires avec des listes : Python, Lisp, Prolog. Dans ce cours, nous définirons un noeud comme étant composé d'une valeur et de ses deux sous-arbres puis d'un arbre composé d'un noeud racine. En C cela se fait avec des structures (et des pointeurs) et en python avec des classes (choix fait dans ce cours).

```
class Noeud:
  """ Classe définissant un noeud d'un arbre binaire """
  def __init__(self, valeur, g=None, d=None):
    self.val = valeur # type supportant une relation d'ordre
    self.g = g        # type arbre
    self.d = d        # type arbre

class Arbre:
  """Classe définissant un arbre binaire"""
  def __init__(self, rac=None):
    self.racine=rac   # type noeud
```

9.2.3 Exemple



9.2.4 Parcours

En profondeur à gauche

Il n'existe aucune justification théorique de partir d'abord à gauche. Dans la pratique il existe trois parcours : préfixé, infixé et suffixé.

Parcours préfixé

```
def affpre(self):
  """ affichage d'un arbre binaire en ordre infixé"""
  if self.racine:
    print(self.racine.val)
    self.racine.g.affpre()
    self.racine.d.affpre()
```

On traite le noeud avant de traiter ses sous-arbres. Voici ce qui est affiché pour l'exemple précédent : x1, x2, x4, x7, x8, x11, x3, x5, x9, x10, x6.

Parcours infixé

```
def affinf(self):
  """ affichage d'un arbre binaire en ordre infixé"""
  if self.racine:
    self.racine.g.affinf()
    print(self.racine.val)
```

```
      self.racine.d.affinf()
```

Chaque noeud est traité après le traitement de son sous-arbre gauche et avant le traitement de son sous-arbre droit. Voici ce qui est affiché pour l'exemple précédent : x7, x4, x11, x8, x2, x1, x9, x5, x10, x3, x6.

Parcours suffixé ou postfixé

```
  def affsuf(self):
    """ affichage d'un arbre binaire en ordre infixé"""
    if self.racine:
      self.racine.g.affsuf()
      self.racine.d.affsuf()
      print(self.racine.val)
```

Chaque noeud est traité après le traitement de ses deuw sous-arbre. Voici ce qui est affiché pour l'exemple précédent : x7, x11, x8, x4, x2, x9, x10, x5, x6, x3, x1.

Parcours divers

- par niveau (sur l'exemple : parcours suivant la numérotation des noeuds),
- parcours en fonction d'une fonction de guidage (heuristique),
- en profondeur à droite,
- ...

9.2.5 Ecriture sous forme totalement parenthésée

On suit l'algorithme de parcours classique en ajoutant des parenthèses. Algorithme pour un affichage totalement parenthésé en ordre infixé :

```
  def affpar(self):
    """Affichage totalement parenthésé"""
    if self.racine:
      print("(",end=' ')
      self.racine.g.affpar()
      print(self.racine.val,end=' ')
      self.racine.d.affpar()
      print(")",end=' ')

Appel principal : ecr_par(racine)
```

9.2.6 Arbres binaires complets de profondeur p

Un arbre binaire complet de profondeur p est un arbre binaire de profondeur p dont tous les noeuds de profondeur strictement inférieure à p possèdent exactement deux fils. Un arbre binaire complet de profondeur p contient exactement $2^{p+1} - 1$ noeuds. Un arbre binaire complet (ou presque complet) est souvent représenté de façon contiguë suivant le schéma suivant :

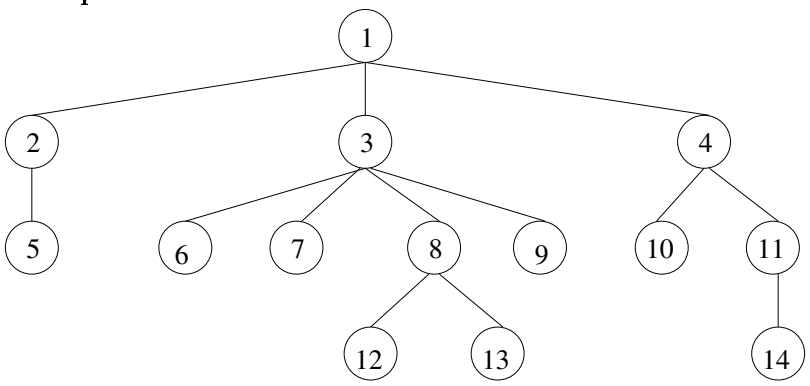
valeur du noeud	sous arbre gauche	sous arbre droit

9.2.7 Représentation d'un arbre quelconque par un arbre binaire

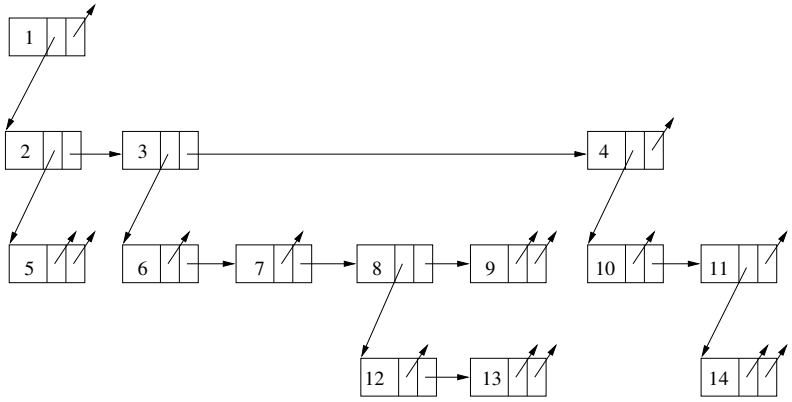
Il est possible de représenter un arbre quelconque par un arbre binaire avec la sémantique suivante :

- le fils gauche de l'arbre binaire représente le fils aîné,
- le fils droit de l'arbre binaire représente le frère.

Exemple :



Codage :



9.3 Les arbres binaires de recherche

9.3.1 Définition

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

- tous les noeuds du sous arbre gauche de x ont une valeur inférieure à la valeur de x,
- tous les noeuds 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 noeuds triée en ordre croissant. Un ABR étant un arbre binaire, sa représentation interne sera celle d'un arbre binaire.

9.3.2 Recherche d'un élément dans un arbre binaire de recherche

Cette fonction renvoie l'adresse du noeud recherché quand il existe et renvoie la valeur nil sinon. Les paramètres de cette fonction sont d'une part, la valeur recherchée (supposée entière dans l'algorithme) et d'autre part, la racine de l'arbre dans lequel on fait la recherche.

```
  def rechercheABR(self, x):
    """ recherche d'un élément x dans un arbre binaire de recherche"""
    if not self.racine:
      return None
    elif x == self.racine.val:
      return self.racine
    elif x < self.racine.val:
      return Arbre.rechercheABR(self.racine.g, x)
    else:
      return Arbre.rechercheABR(self.racine.d, x)
  ## fin de rechercheABR

Appel principal : adresse = racine.rechercheABR(x)
```

9.3.3 Ajout d'un élément dans un arbre binaire de recherche

Deux stratégies existent : soit ajouter le nouvel élément à la racine, soit ajouter cet élément à une feuille. L'ajout à la racine est plus compliqué car il nécessite de couper l'arbre et n'apporte aucun avantage particulier en termes d'équilibre de l'arbre. Seul sera présenté l'ajout à une feuille.

```
  # ajout de x dans un ABR
  def ajoutABR(self, x) :
    """ ajout dans une feuille d'un élément x dans un arbre binaire de recherche"""
    ndx = Noeud(x, Arbre(), Arbre())
    if not self.racine: # si l'arbre est vide
      self.racine = ndx
    else :
      courant = self
      # descente jusqu'à la feuille
      while courant.racine:
        pere = courant
        if x <= courant.racine.val:
          courant = courant.racine.g
        else:
          courant = courant.racine.d
      # fin du while
      # test pour savoir s'il faut ajouter à gauche ou à droite
      if x <= pere.racine.val:
        pere.racine.g = Arbre(ndx)
      else:
        pere.racine.d = Arbre(ndx)
```

```
Appel principal : racine.ajout_ABR(x0)
```

Remarque : Cet algorithme permet la construction d'un arbre binaire de recherche par ajouts successifs à partir d'un arbre vide.

9.3.4 Suppression d'un élément dans un arbre binaire de recherche

Afin d'éviter à avoir à restructurer profondément un arbre après une suppression brutale, on va toujours se ramener à supprimer physiquement un noeud ayant au maximum un fils. Si le noeud à supprimer possède deux fils, on commence par supprimer le noeud contenant la plus grande valeur de son sous arbre gauche en mémorisant cette plus grande valeur. Cette valeur est ensuite placée dans le noeud contenant la valeur à supprimer. Dans les algorithmes suivants, on suppose que l'arbre n'est pas vide.

Suppression du maximum

```
# Recherche et suppression du maximum d'un sous-arbre de racine rac
  def maxiABR(rac) :
    assert rac != None
    courant = rac
    while courant.d != None :
      pere = courant
      courant = courant.d
    if courant == rac :
      rac2 = rac.g
    else :
      pere.d = courant.g
      rac2 = rac
    # on renvoie la valeur du maximum et la racine de l'arbre
    return courant.val, rac2

L'appel principal sera fait dans l'algorithme général de suppression.
```

Suppression dans le cas général

```
  def suppABR(rac,x):
    pere = None
    courant = rac
    # boucle de recherche de l'élément à supprimer
    while (courant != None) and (courant.val != x) :
      pere = courant
      if x < courant.val :
        courant = courant.g
      else :
        courant = courant.d
    # end while

    if courant == None : # élément à supprimer absent
```

```
      print("élément ", x, " absent de l'arbre")
      return rac

    # sinon : on a trouvé l'élément à supprimer : 3 cas possibles
    elif courant.g == None : # pas de fils gauche
      # il faut supprimer courant.val
      if pere != None :
        if x < pere.val :
          pere.g = courant.d
        else :
          pere.d = courant.d
        # free courant
        return rac
      else :
        return courant.d  # et free(courant)

    elif courant.d == None : # pas de fils droit
      if pere != None :
        if x < pere.val :
          pere.g = courant.g
        else :
          pere.d = courant.g
        # free courant
        return rac
      else :
        return courant.g  # et free(courant)

    else : # 2 fils
      courant.val, courant.g = Noeud2.maxiABR(courant.g)
      return rac
  # FIN DE suppABR

Appel principal : racine = suppABR(x0)
```

9.4 Les arbres AVL

9.4.1 Introduction

Les arbres binaires de recherche ne sont performants que quand ils sont équilibrés. En effet, la recherche, comme la mise à jour, dans un ABR devient séquentielle quand ce dernier est dégénéré en liste (un seul fils par noeud) alors que dans le meilleur des cas, elle est logarithmique par rapport au nombre de noeuds. On peut noter qu'un arbre initialement équilibré peut rapidement dégénérer après une série de suppressions et d'ajouts.

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

* La fonction mesurant le déséquilibre est définie comme suit :
Des($\emptyset$) = 0 et
Des(<x,G,D>) = prof(G) - prof(D)

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

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

* Un **arbre AVL** est un arbre binaire de recherche H-équilibré.

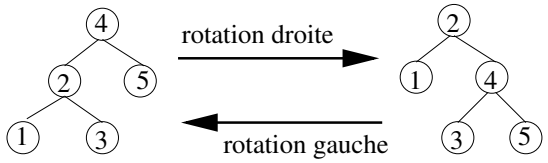
* La structure interne des arbres AVL est définie en ajoutant un attribut déséquilibre à la classe noeud :

```
class NoeudAVL:
  """ Classe définissant un noeud d'un arbre binaire """
  def __init__(self, valeur, g=None, d=None, des=0):
    self.val = valeur # type supportant une relation d'ordre
    self.g = g        # type arbre
    self.d = d        # type arbre
    self.des = des    # type entier variant de -2 à +2
```

9.4.2 Les rotations

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

Les rotations simples

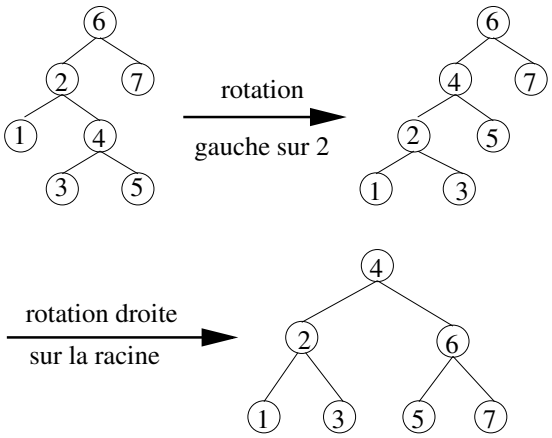



```
  def rotationdroite(self):
    b = self.racine.g
    self.racine.g = b.racine.d
    b.racine.d = self
    return b
```

Cette fonction ne met pas à jour les déséquilibres des noeuds. L'algorithme de la rotation gauche est analogue, il suffit d'inverser g et d.

Les rotations composées

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



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

9.4.3 Recherche dans un arbre AVL

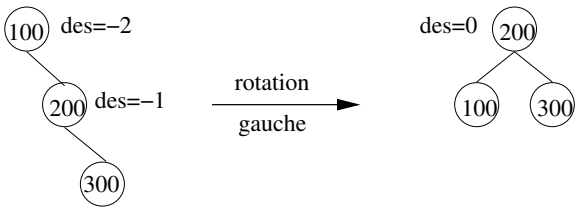
L'algorithme de recherche d'un élément est identique à celui utilisé dans le cas des arbres binaires de recherche.

9.4.4 Ajout dans un arbre AVL

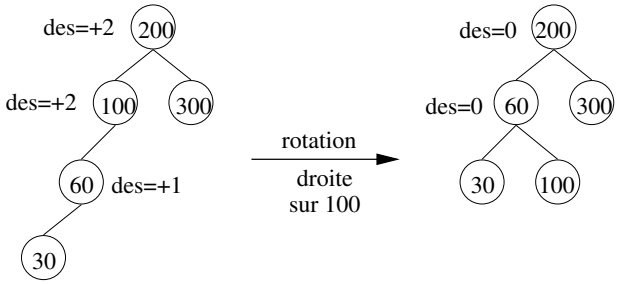
L'ajout dans les arbres AVL se fait au niveau des feuilles (algorithme identique à celui pour les arbres binaires de recherche). Après cette phase, il est nécessaire de mettre à jour le champ déséquilibre des noeuds concernés et éventuellement de rééquilibrer l'arbre. Une seule rotation suffit pour rétablir l'équilibre à condition de maintenir l'équilibre après chaque ajout. Nous allons maintenant voir sur un exemple les différents cas de figure.

Exemple

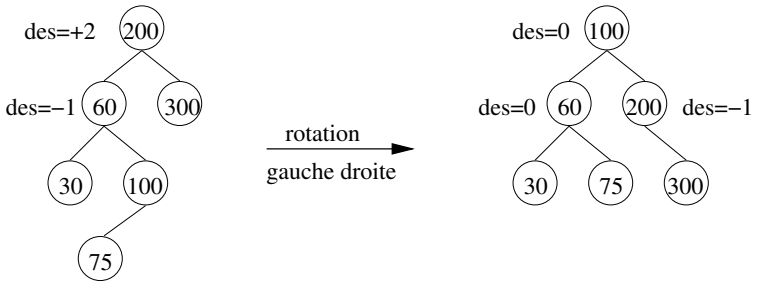
Ajouts successifs de 100, 200 et 300 :



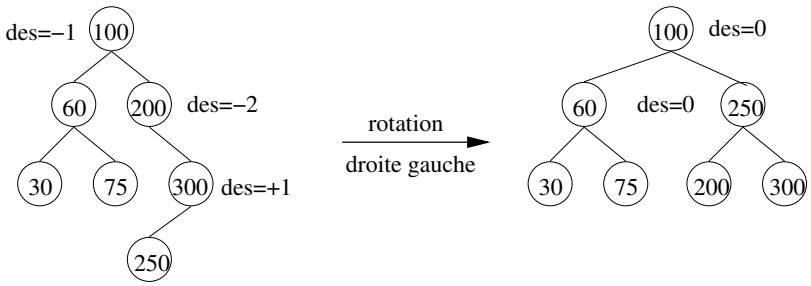
Ajouts successifs de 60 et 30 :



Ajout de 75 :



Ajout de 250 :



Récapitulatif des différents cas

On suppose que l'ajout a été fait au niveau des feuilles et que la valeur de déséquilibre de chacun des sommets a été mise à jour. On considère les noeuds situés sur le chemin entre la racine et le nouvel élément ajouté. Si aucun noeud n'a un déséquilibre égal à +2 ou à -2 alors l'arbre est équilibré (aucun traitement à faire). Sinon on note N1 le noeud le plus profond de déséquilibre égal à +2 ou à -2 et N2 son fils sur le chemin menant au nouveau noeud.

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

L'algorithme commence par parcourir itérativement l'arbre pour savoir où ajouter le nouvel élément. On procède ensuite à la réactualisation du déséquilibre des noeuds qui le nécessitent. Après avoir ajouté le nouvel élément, l'algorithme procède si nécessaire au rééquilibrage de l'arbre. Le principe du rééquilibrage est simple (une rotation) mais l'algorithme comprend de nombreux cas particuliers. Le plus compliqué (fastidieux) est la mise à jour de l'attribut "des" après une rotation. Dans les documents cités en bibliographie, l'algorithme correct détaillé est présent dans les livres de D.E. Knuth et de C. Carrez ; le livre de M.C. Gaudel, M. Soria et C. Froidevaux contient un algorithme détaillé comportant 4 erreurs de mise à jour des déséquilibre, quant au livre de J. Courtin et I. Kowarski, il ne traite pas ce genre d'arbre.
Les algorithmes suivant ne sont pas donnés en python mais en pseudo-code.

Procédure d'ajout. Cette procédure ajoute le nouvel élément et repère le noeud le plus profond, quand il exite de déséquilibre +2 ou -2.

Dictionnaire des objets :

nom	type	statut	rôle
R	AVL	paramètre E	racine de l'arbre
element	entier	paramètre E	élément à ajouter
adx	AVL	paramètre ES	adresse nouvel élément
adpb	AVL	paramètre S	adresse du noeud le plus profond en déséquilibre
padpb	AVL	paramètre S	père de adpb
courant	AVL	variable locale	adresse du noeud courant
pcourant	AVL	variable locale	père de courant

```
procedure ajout_simple(R : ES AVL, element : E entier,
                       adx : ES AVL, adpb : S AVL, padpb : S AVL)
debut
   adel := allouer()
   adx^.x := element ; adx^.des := 0
   adx^.g := nil ; adx^.d := nil
   si R == nil alors R := adx
     sinon
       courant := R ; pcourant := nil
       adpb := R ; padpb := nil
       tantque courant != nil faire
          si courant^.des != 0 alors
             adpb := courant ; padpb := pcourant
          fsi
          pcourant := courant
          si element <= courant^.x alors
              courant := courant^.g
            sinon
              courant := courant^.d
          fsi
       ftq
```

```
       si element <= pcourant^.x alors
           pcourant^.g := adx
         sinon
           pcourant^.d := adx
       fsi
   fsi
fin ajout_simple
```

Mise à jour du déséquilibre. La procédure précédente ajoute un élément sans se soucier de modifier le déséquilibre des noeuds. dans le cas où l'arbre ne serait plus H-équilibré, il existe une propriété importante. Soit N1 le noeud le plus profond de déséquilibre égal à +2 ou -2 après ajout, la profondeur de son sous-arbre est la même avant l'ajout qu'après l'ajout et rééquilibrage. Par conséquent, il suffit de commencer à mettre à jour le déséquilibre à partir de N1. Quand l'arbre demeure équilibré après l'ajout, il faut mettre à jour le déséquilibre depuis la racine (voir adpb dans l'algorithme précédent). Le premier paramètre de la procédure est l'adresse du noeud de début pour la mise à jour du déséquilibre ; le deuxième paramètre est la valeur du nouvel élément et enfin, le troisième paramètre est l'adresse de cet élément.

```
procedure maj_des(deb : E AVL, element : E entier, adx : E AVL)
debut
   courant := deb
   tantque courant != adx faire
       si element <= courant^.x alors
           courant^.des := courant^.des+1
           courant := courant^.g
         sinon
           courant^.des := courant^.des-1
           courant := courant^. d
       fsi
   ftq
fin maj_deb
```

Procédure de rééquilibrage. Le premier paramètre est le noeud sur lequel s'effectue la rotation, le deuxième paramètre étant la valeur de l'élément ajouté.

```
procedure reequilibrage (adpb : ES AVL, element : E entier)
debut
  cas
     adpb^.des == 2 :
        si adpb^.g^.des == 1 alors
            adpb := RD(adpb)
            adpb^.des := 0 ; adpb^.d^.des := 0
          sinon // adpb^.g^.des == -1
            adpb := RGD(adpb)
            cas
               adpb^.des == -1 :
                     adpb^.g^.des := 1
```

```
                    adpb^.d^.des := 0
               adpb^.des == 0 :
                    adpb^.g^.des := 0
                    adpb^.d^.des := 0
               adpb^.des == 1 :
                    adpb^.g^.des := 0
                    adpb^.d^.res = -1
            fcas
            adpb^.des := 0
        fsi
     adpb^.des == -2 :
        si adpb^.d^.des == -1 alors
            adpb := RG(adpb)
            adpb^.des := 0 ; adpb^.g^.des := 0
          sinon // adpb^.d^.des == 1
            adpb := RDG(adpb)
            cas
               adpb^.des == -1 :
                    adpb^.g^.des := 1
                    adpb^.d^.des := 0
               adpb^.des == 0 :
                    adpb^.g^.des := 0
                    adpb^.d^.des := 0
               adpb^.des == 1 :
                    adpb^.g^.des := 0
                    adpb^.d^.res := -1
            fcas
            adpb^.des := 0
        fsi
  fcas
fin reequilibrage
```

Procédure finale :

```
procedure ajout_AVL(R : ES AVL, element : E entier)
debut
   ajout_simple(R, element, adx, adpb, padpb)
   maj_des(adpb, element, adx)
   si padpb == nil alors
       R := adpb
     sinon si adpb^.x <= padpb^.x alors
       padpb^.g := adpb
     sinon
       padpb^.d := adpb
   fsi
   fsi
fin ajout_AVL
```

9.4.5 Suppression dans un arbre AVL

L'algorithme de suppression simple, sans mise à jour de la fonction de déséquilibre ni rééquilibrage de l'arbre, est identique à l'algorithme de suppression dans un arbre binaire de recherche.

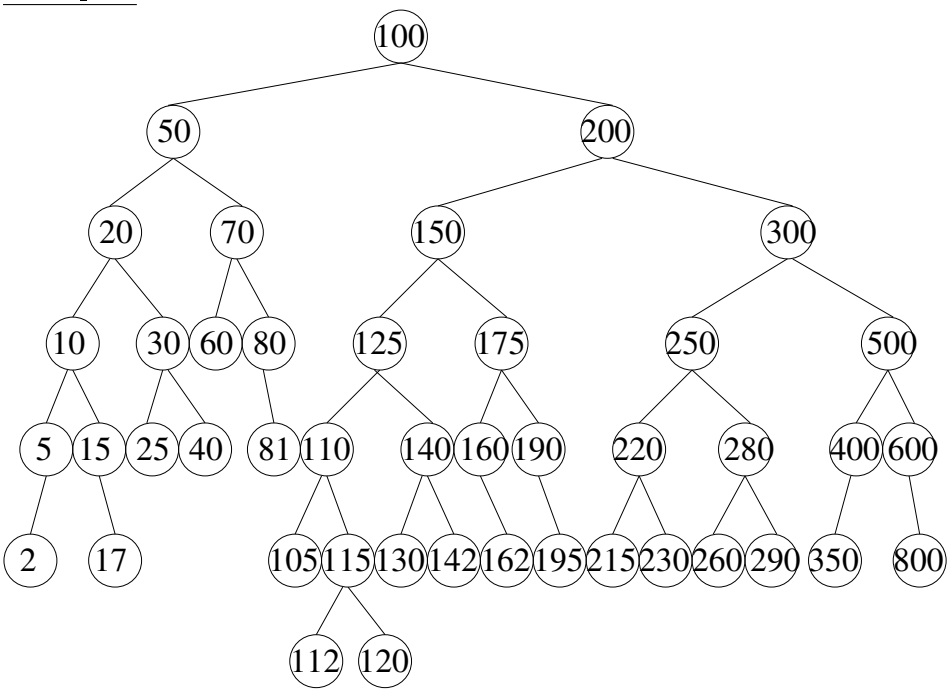
Principe du rééquilibrage

On peut montrer que des rotations peuvent être nécessaires uniquement sur le chemin allant du noeud physiquement supprimé à la racine. On applique l'algorithme de rééquilibrage (le même que celui détaillé lors de l'ajout d'un élément) sur tous les noeuds de ce chemin (à partir du noeud supprimé) ayant un déséquilibre de +2 ou de -2

Algorithme général

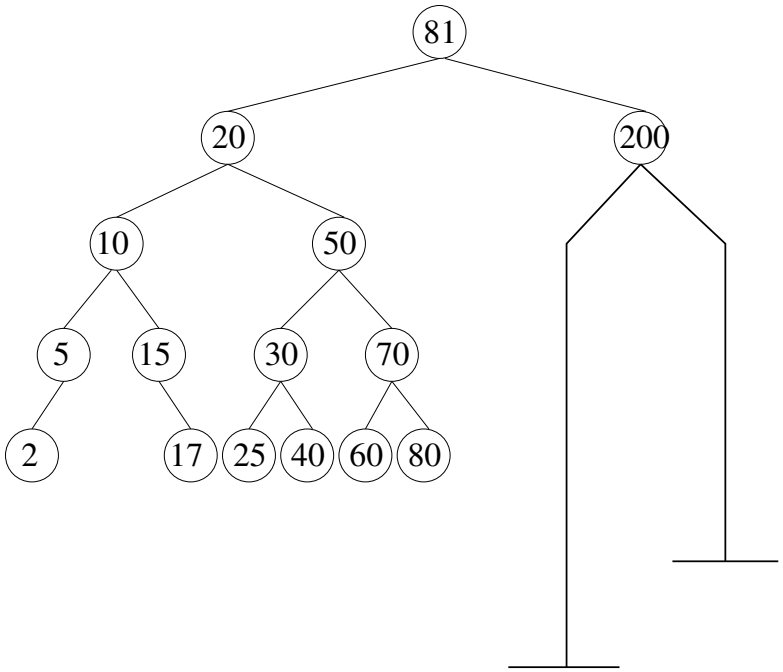
```
procedure supp_avl(R : ES AVL, elemnt : E entier)
debut
   suppression_simple(R, element, dernier)
   // identique à la suppression dans un ABR avec :
   // mise à jour du déséquilibre des noeuds concernés
   // création d'une liste contenant le chemin allant
   // du noeud supprimé à la racine, dernier : tête de liste
   courant := dernier
   tantque courant != nil faire
       reequilibrage(noeud,element)
         // ne fait rien si déséquilibre = 1, 0 ou -1
       courant := courant^.svt
   ftq
fin supp_AVL
```

Exemple :



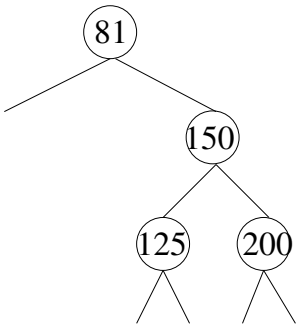
Soit 100 (contenu dans la racine), l'élément à supprimer. On commence par appliquer l'algorithme de suppression sans se préoccuper du déséquilibre de l'arbre : suppression physique du noeud contenant l'élément 81 et enregistrement de cet élément dans le noeud racine.

Considérons maintenant les déséquilibres sur le chemin entre le noeud supprimé et la racine : les déséquilibres des noeuds 80 et 70 sont égaux à 0, le déséquilibre en 50 est égal à +2. Il faut donc appliquer l'algorithme de rééquilibrage en 50. En ce noeud la situation est +2, +1 : on applique une rotation droite sur le noeud 50.

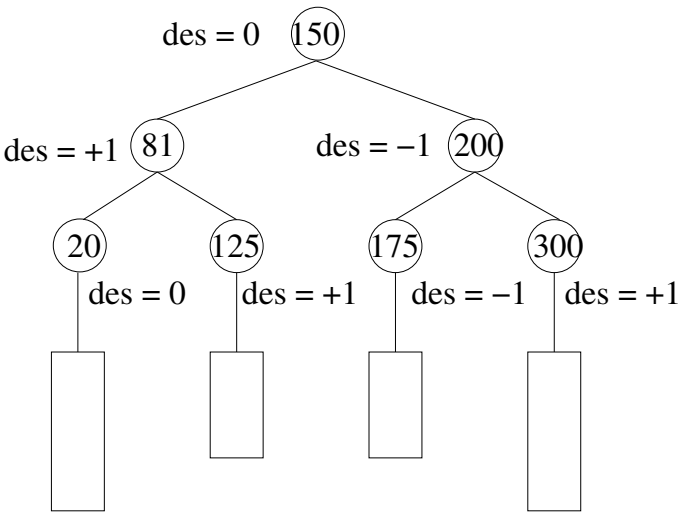


Après cette rotation, le déséquilibre du noeud courant, qui contient maintenant l'élément 20, est égal à zéro. On poursuit donc l'algorithme en remontant vers la racine. Le déséquilibre de la racine est devenu égal à -2. Il faut donc lui appliquer l'algorithme de rééquilibrage. Nous sommes dans le cas de figure -2, +1 : on applique une rotation droite gauche sur la racine.

rotation droite sur
sur le sous arbre droit



Résultat de la rotation droite gauche :



9.4.6 Complexité au pire

- Recherche : $\log_2(N)+1$ comparaisons,
- Ajout : $\log_2(N)+1$ comparaisons et une rotation,
- Suppression : $\log_2(N)+1$ comparaisons et $1{,}5.\log_2(N)$ rotations,
- Statistiquement : une rotation pour 5 suppressions.

9.5 Les arbres (a-b)

Les arbres (a-b) sont des arbres généraux de recherche. Comme dans un arbre AVL, les éléments y sont ordonnés. Les nombres a et b sont des entiers déterminant les nombres minimum et maximum de fils (et donc de valeurs) en chaque noeud. On a $a \geq 2$ et $(b+1)/2 \geq a$. Un arbre (a-b) peut être vide sinon il doit respecter les propriétés suivantes :

- la racine contient entre 1 et b-1 éléments (valeurs) ordonnés ;
- tout noeud autre que la racine contient un nombre de noeuds compris entre a-1 et b-1 éléments ordonnés ;
- toutes les feuilles sont à la même profondeur
- tous les noeuds, sauf les feuilles, ont entre a et b fils. La racine peut ne posséder que 2 fils. En fait le nombre de fils d'un noeud (non feuille) est égal à son nombre d'éléments plus 1.
- Les éléments d'un arbre (a-b) sont ordonnés en généralisant ce qui est fait pour les arbres AVL.

Dans les sections suivantes, on va étudier trois types d'arbres (a-b) :

- les arbres 2-3-4 avec a = 2 et b =4,
- les B-arbres avec a = d+1 et b = 2d+1 où d est le degré du B-arbre,
- les arbres B+ (même valeurs a et b que pour les B-arbres) même si en toute rigueur ce ne sont pas des arbres en raison d'un chaînage entre les feuilles.

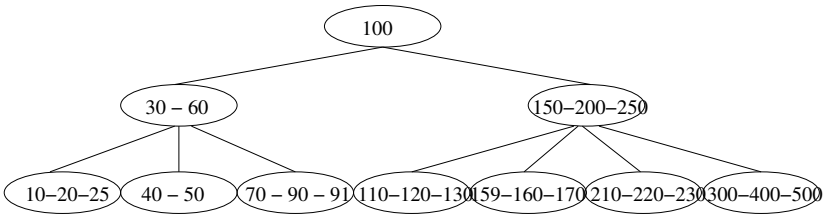
9.6 Les arbres 2-3-4

9.6.1 Introduction

Les arbres 2-3-4 sont des arbres de recherche dont chaque noeud possède de 1 à 3 éléments. Ceci permet d'imposer une contrainte supplémentaire : toutes les feuilles doivent être au même niveau.

Propriété : en notant N le nombre d'éléments et h la hauteur de l'arbre, on a : $log_4(N+1) \leq h+1 \leq log_2(N+1)$

Exemple :



9.6.2 Représentation interne

Il existe deux principales façons de représenter les arbres 2-3-4.

Première manière

```
class Noeud234a:
  """ Classe définissant un noeud d'un arbre 2-3-4 """
  def __init__(self, v1, v2, v3, g=None, m1=None, m2=None, d=None):
    self.val1 = v1 # type supportant une relation d'ordre
    self.val2 = v2 # même type que val1 (ou None)
    self.val3 = v3 # même type que val1 (ou None)
    self.g = g     # type arbre (ou None)
    self.g = m1    # type arbre (ou None)
    self.g = m1    # type arbre (ou None)
    self.d = d     # type arbre (ou None)
```

On peut également ajouter un champ entier indiquant le nombre de valeurs stockées dans le noeud.

Avec des arbres binaires de recherche bicolores

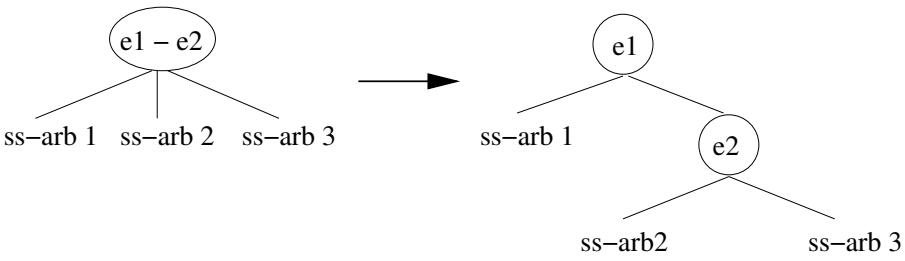
L'idée consiste à représenter chaque noeud d'un arbre 2-3-4 par un ABR :

```
class Noeud234a:
  """ Classe définissant un noeud d'un arbre 2-3-4 """
  def __init__(self, couleur, valeur g=None, d=None):
    self.coul = couleur  # blanc  : 1, rouge : 2
    self.val = valeur    # type supportant une relation d'ordre
    self.g = g           # type arbre (ou None)
    self.d = d           # type arbre (ou None)
```

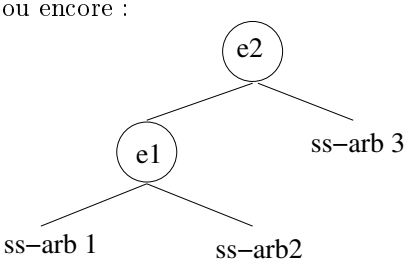
Nous allons maintenant examiner les trois cas possibles.

Noeuds contenant un seul élément : ces noeuds sont représentés de façon identique. Ce sont les noeuds habituels des arbres AVL.

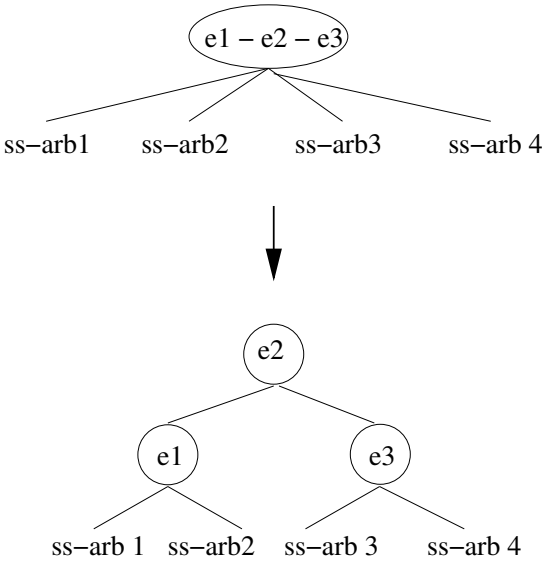
Noeuds contenant deux éléments : deux représentations sont possibles.



ou encore :



Noeuds contenant trois éléments : une représentation possible.



Dans cette représentation, un noeud 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**.

9.6.3 Recherche dans un arbre 2-3-4

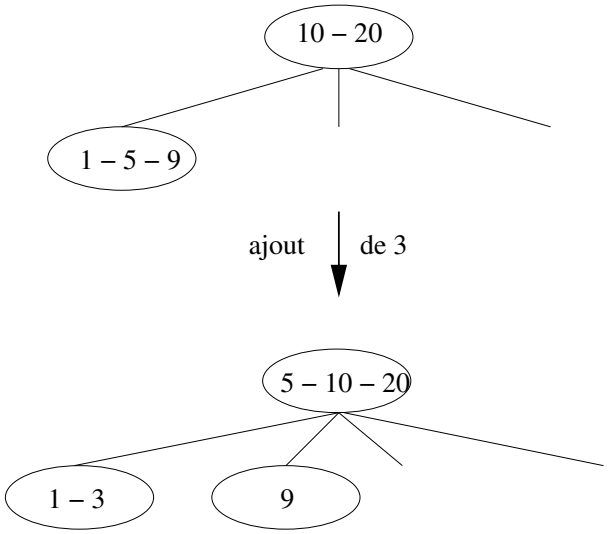
La recherche d'un élément ne présente pas de difficulté. Quand l'arbre est représenté par un arbre binaire de recherche bicolore, l'algorithme est le même que celui de la recherche dans un arbre binaire de recherche.

9.6.4 Ajout d'un élément dans un arbre 2-3-4

L'ajout dans un arbre 2-3-4 se fait toujours dans une feuille. Lorsque la feuille ne contient qu'un ou deux éléments, l'ajout ne comporte aucune difficulté. Quand la feuille contient déjà trois éléments, il faut la faire éclater. Il existe deux méthodes : l'une avec éclatement(s) à la remontée et l'autre avec éclatement(s) à la descente.

Eclatement à la remontée

Le principe consiste à faire éclater le noeud en faisant remonter son élément médian. Cette technique risque d'entraîner des éclatements en cascade.



Eclatement à la descente

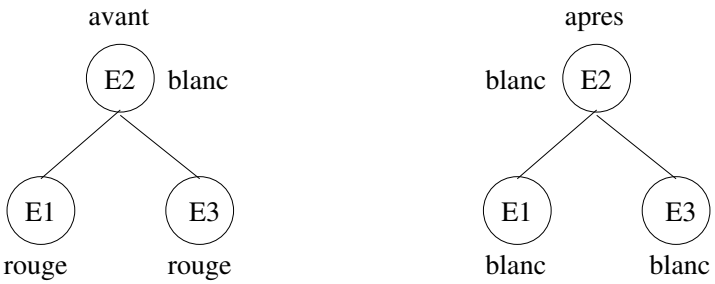
On éclate durant la phase de descente tous les noeuds contenant trois éléments sur le chemin allant de la racine à la feuille où le nouvel élément sera inséré.

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

Inconvénients :
- taux de remplissage plus faible,
- donc profondeur plus importante.

9.6.5 Etude de l'éclatement d'un noeud

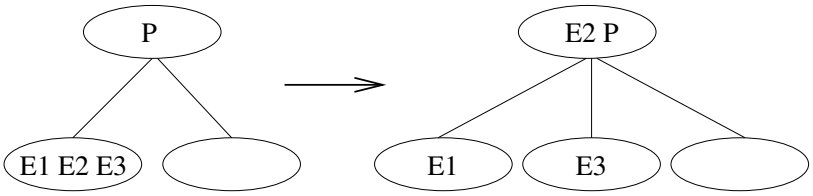
On se place dans le contexte où l'arbre 2-3-4 est représenté par un arbre binaire de recherche bicolore. On y étudie l'éclatement d'un noeud possédant trois éléments.



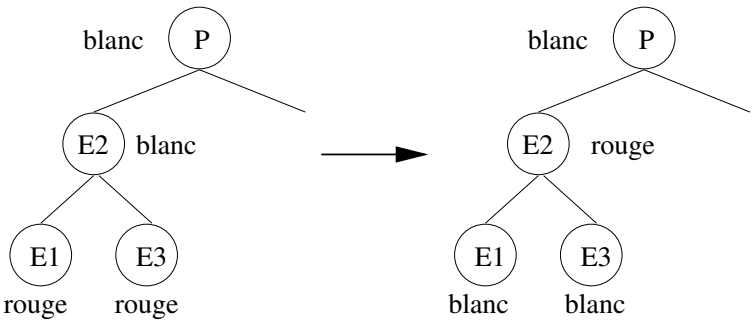
Comme le montre ce schéma, seule la couleur change mais ceci risque de faire apparaître deux noeuds rouges consécutifs. La hauteur de l'arbre ne serait plus logarithmique. Pour y remédier, on utilise des rotations.

Le père possède un élément

Eclatement du noeud :

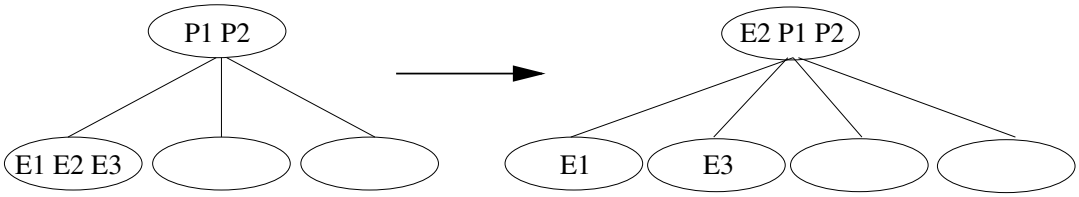


Au niveau de sa représentation :

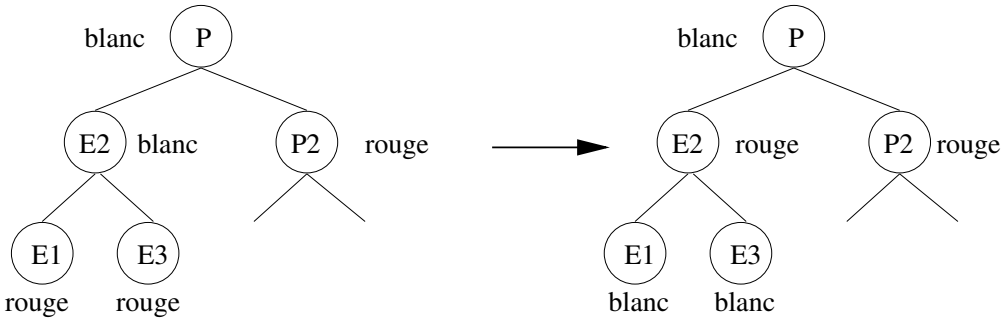


Le père possède deux éléments

Premier cas : le noeud à éclater est à droite :

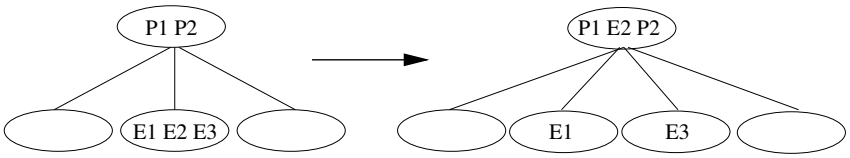


Représentation :

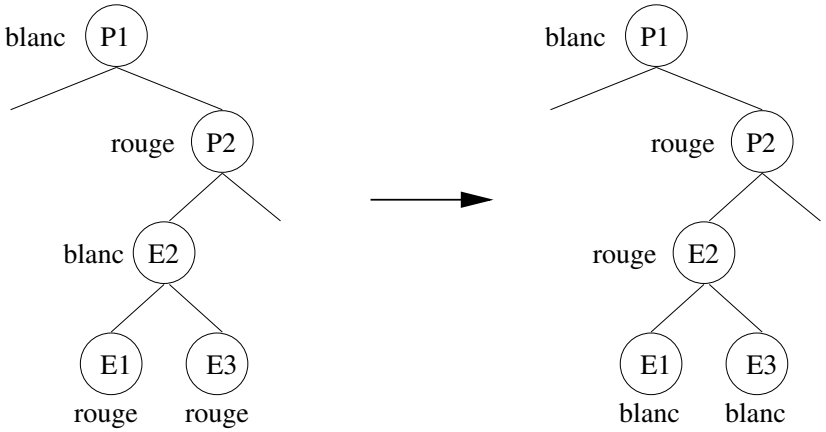


Pas de problème dans ce cas.

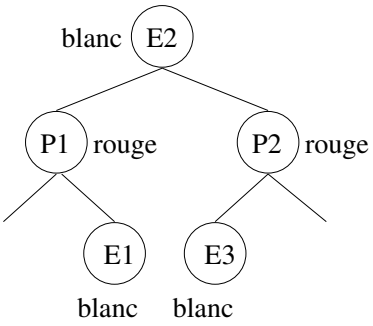
Deuxième cas : le noeud à éclater est au milieu :



Représentation :

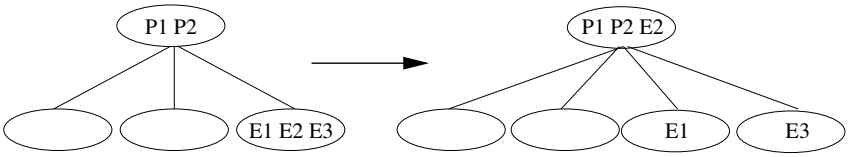


`Problème :` apparition de deux noeuds rouges consécutifs : on applique une rotation droite gauche en P1.

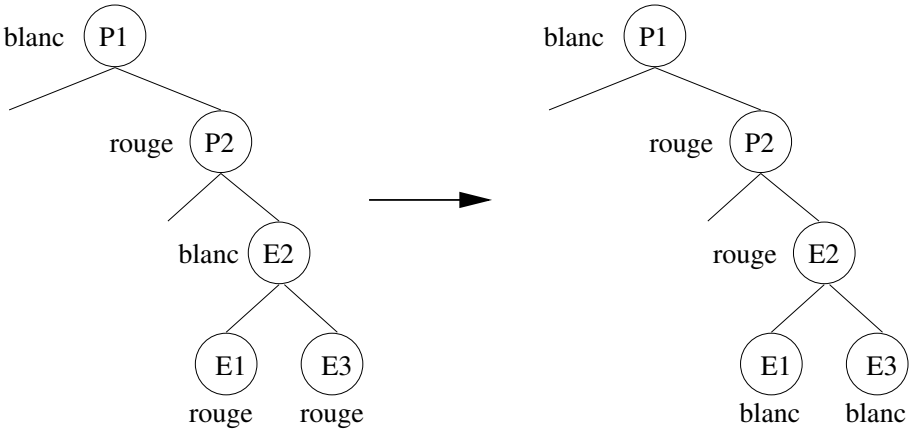


L'arbre est maintenant rééquilibré.

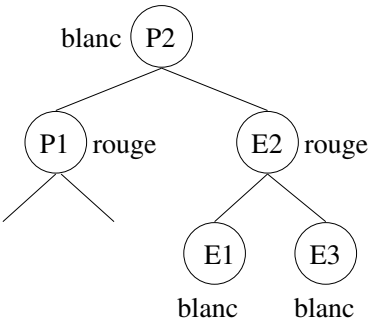
Troisième cas : le noeud à éclater est à gauche :



Représentation :



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



L'arbre est maintenant rééquilibré.

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.

Algorithme

Il ne présente pas de difficultés mais comporte de nombreux cas particuliers. A faire en exercice !

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 est un cas particulier de la suppression dans un B-arbre étudiée dans la partie suivante.

9.7 Les arbres balancés ou B-arbres

9.7.1 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 (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 noeud possède au plus 2m éléments,
- tout noeud, la racine exceptée, possède au moins m éléments.

En général : $20 \leq m \leq 500$

Question : nombres minimum et maximum d'éléments en fonction de m et de la profondeur p du B-arbre ?

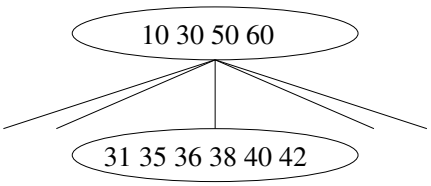
9.7.2 Représentation interne et recherche d'un élément

Il existe plusieurs représentations internes possibles : faire de la bibliographie. Une fois la représentation choisie, la recherche d'un élément ne pose aucun problème.

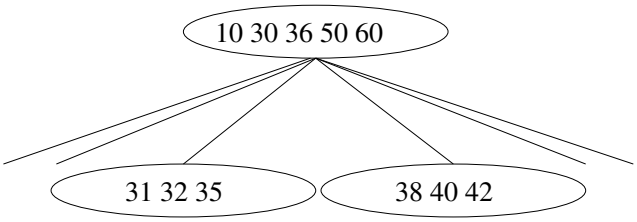
9.7.3 Ajout dans un B-arbre

Le principe est identique à celui de l'ajout dans un arbre 2-3-4. En cas de problème (feuille comportant déjà 2m éléments), on retrouve l'algorithme avec éclatement(s) à la remontée et l'algorithme avec éclatement à la descente.

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



L'ajout de 32 provoque la remontée de 36 ; question : pourquoi ne pas remonter 38 ?



Algorithme simplifié

```
def ajout_B_arbre(arbre, x):
  f = chercher_feuille(arbre, x)
  if place_libre_dans(f) :
    insérer_trié(x, f)   # cas simple
    return arbre
  else:                  # éclatement de la feuille
    val_milieu, f1, f2 = éclater(f)
    p = père(f)
    # attention, la racine peut changer
    return remonter(arbre, p, val_milieu, f1, f2)

def remonter(arbre, noeud, val, noeud1, noeud2):
  if (noeud == None):  # noeud à éclater = racine
    racine2 = éclater_racine(val_milieu, f1, f2)
    return racine2
  else:
    if place_libre_dans(noeud): # arrêt récursivité
      ajouter_dans_noeud(noeud, val_milieu, f1, f2)
      return arbre
    else:
      val_milieu, f1, f2 = réorganiser(noeud, val_milieu, f1, f2)
      p = père(noeud)
      return remonter(arbre, p, val_milieu, f1, f2)
```

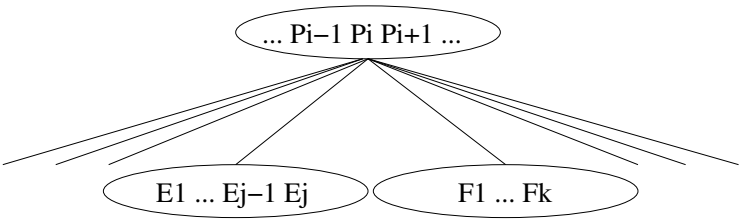
9.7.4 Suppression dans un B-arbre

Afin de na pas avoir à restructurer trop lourdement l'arbre, on se ramène toujours à supprimer un élément dans une feuille. Quand l'élément à éliminer 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 considérée contenait exactement m éléments, il faut rééquilibrer le B-arbre. Pour ce faire, on utilise les opérations de répartition et / ou de regroupement.

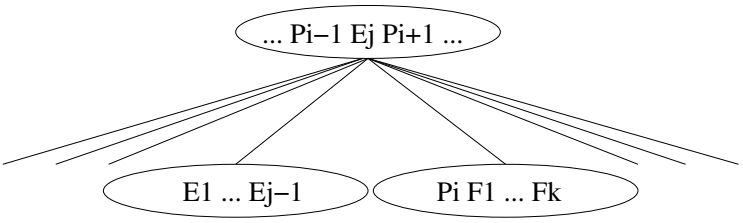
Répartition de gauche à droite

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

Situation initiale :



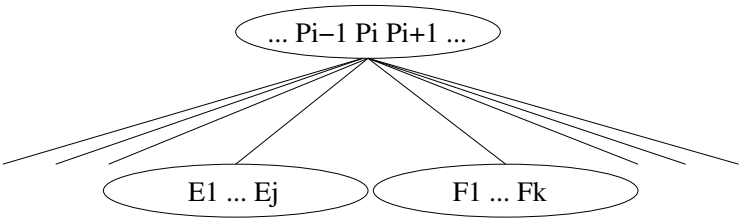
Situation après répartition :



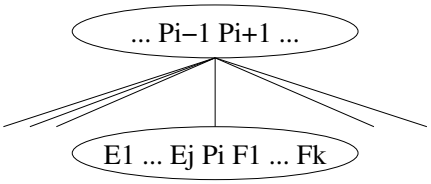
La répartition de droite à gauche est analogue (opération inverse).

Regroupement

Situation initiale :



Situation après 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 possède au moins m+1 éléments, on réalise alors 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. Ceci implique la perte d'un élément pour le père. 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 noeud devient la nouvelle racine du B-arbre.

Algorithme simplifié

```
def supp_B_arbre(arbre, x):
  noeud = chercher(arbre, x)
  if noeud == None : # valeur absente
    print("élément absent")
    return arbre # ou levée d'une exception
  elif not feuille(noeud): # l'élément n'est pas dans une feuille
    # on prend la feuille la plus à droite du sous-arbre séparant x
    # de l'élément précéeant dans le noeud
    f_droite = feuille_droite(sous_arbre(noeud))
    # on échange x et la plus grande valeur de f_droite
    échanger(x,noeud,f_droite)
    noeud = f_droite
  # maintenant on va supprimer x de noeud
  # en sachant que noeud est une feuille

  m = degre(arbre)
  p = père(noeud)
  if nb_élément(noeud) > m :
    supprimer(x, noeud) # suppression simple de x dans la feuille
    return arbre
  elif repartition_possible(pere, noeud):
    repartir(pere, noeud, x)
    if nb_élément(p) >= m:
      return arbre
    else:
      return regrouper(arbre, pere)
```

```
  else: # il faut regrouper le noeud avec son frère gauche
        # ou son frère droit
    return regrouper(arbre, noeud)
```

La fonction / méthode regrouper regrouper le noeud passé en deuxième paramètre avec un de ses frères : il en a 1 (cas particuliers) ou 2 (cas général). Cette fonction récursive peut être amenée à faire un regroupement sous la racine, dans ce cas la racine change.

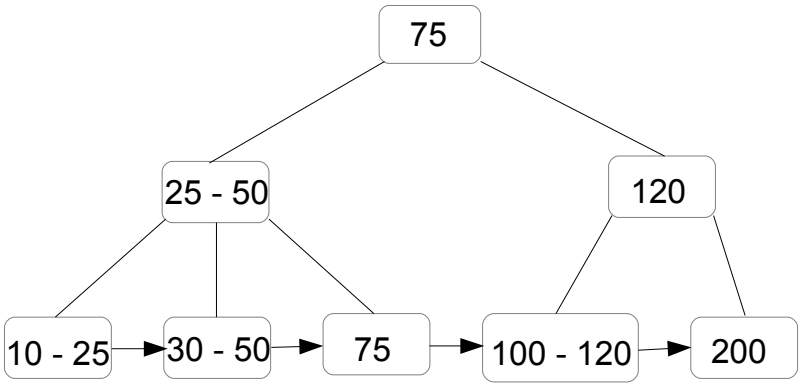
9.8 Les arbres B+

Les arbres B+ sont une variante des B-arbres presque exclusivement utilisée en base de données pour indexer des relations. On ajoute deux propriétés aux B-arbres :

- Tous les éléments présents dans un noeud n'étant pas une feuille sont dupliqués dans une feuille
- Chaque feuille est chaînée à la feuille suivante. Il en découle que les arbres B+ ne sont plus des arbres (au sens théorie des graphes) mais ce chaînage optimise les parcours complets d'une relation en base de données.

Les algorithmes de recherche, ajout, suppression découlent de ceux des B-arbres en étant un peu plus compliqués, ils pourront être étudiés dans le cours de base données.

Exemple :



9.9 Le tri par tas (heapsort)

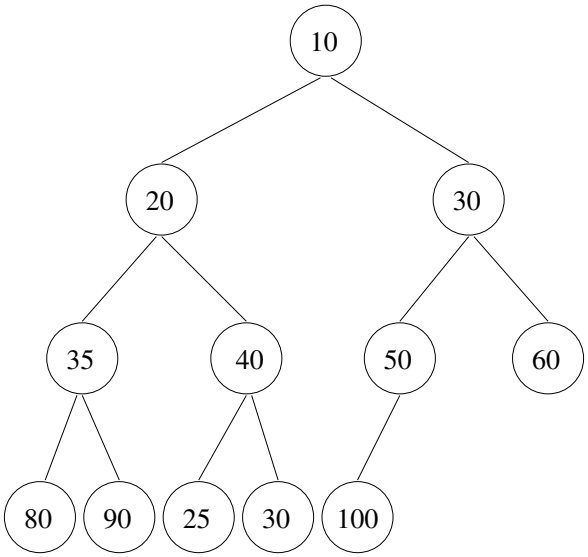
C'est une méthode de tri par sélection du minimum ou du maximum. Les éléments non encore triés sont organisés en tas.

9.9.1 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 tel arbre se représente facilement dans un tableau en l'y rangeant par niveau.

Exemple :



Représentation dans un tableau :

1	2	3	4	5	6	7	8	9	10	11	12
10	20	30	35	40	50	60	80	90	25	30	100

- La racine est à l'indice 1 du tableau (attention en langage C).
- Pour un noeud situé à l'indice i :
  - son fils gauche, si il existe, est à l'indice 2i,
  - son fils droit, si il existe, est à l'indice 2i+1,
  - son père, sauf pour la racine, est à l'indice i/2 (division entière).

Définition : un tas est un arbre binaire parfait partiellement ordonné. Pour tout noeud, 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.

9.9.2 Construction initiale du tas

On remarquera que le tableau initial représente un arbre parfait : il suffit de réordonner cet arbre. Pour ce faire, il suffit de comparer la valeur d'un noeud avec la valeur de son père et ceci à partir des feuilles. Quand les valeurs sont inversées, on les échange. Dans ca cas, il faut vérifier si la valeur du père ne doit pas descendre plus bas dans l'arbre.

Procédure ordonnant un sous arbre de racine i

```
# CARACTERISTIQUE D'UN ARBRE BINAIRE PARFAIT
# racine : indice 0
# pour un noeud 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 noeud avec fils : n/2-1

# Fonction ordonnant un sous-arbre (en tas)
# Paramètres : rac : entier - E - indice de la racine du sous-arbre
#              n : entier - E - taille du tableau de l'arbre
#              t : liste / tableau - ES - arbre à réordonner
# Valeur retournée : tableau réordonner
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 noeud 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
```

Création du tas initial

```
# Fonction créant le tas initial
# Paramètre : tableau (liste) - ES - 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
```

9.9.3 Reconstruction du tas après la suppression de son maximum

A chaque itération de l'algorithme principal, on échange le maximum avec le dernier élément du tas. Le maximum passe dans la partie triée à la fin du tableau. Il faut réorganiser le tas : faire descendre, le cas échéant, sa nouvelle racine.

```
# reconstruction du tas après l'échange entre le premier et le dernier élément
#  Paramètres : p : entier - E -nombre d'éléments non triés (taille du tas)
#               t : tableau (liste) - ES - tableau contenant le tas
# Valeur renvoyée : tableau (liste) - ras 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
```

9.9.4 Algorithme final

```
# Fonction principale de tri par tas
# Paramètre : t - tableau (liste) - ES - tableau initial
# Valeur renvoyée : tableau trié
def tritas (t) :
    t = creertas(t)
    k = len(t)
    while (k > 1) :
        k = k-1
```

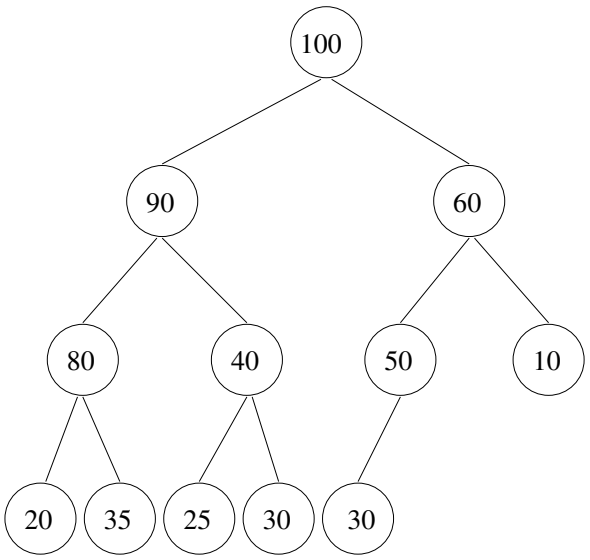
```
        t[k], t[0] = t[0], t[k]
        t = reconstruiretas(k,t)
    # end while
    return t
# Fin de la fonction tritas
```

9.9.5 Complexité

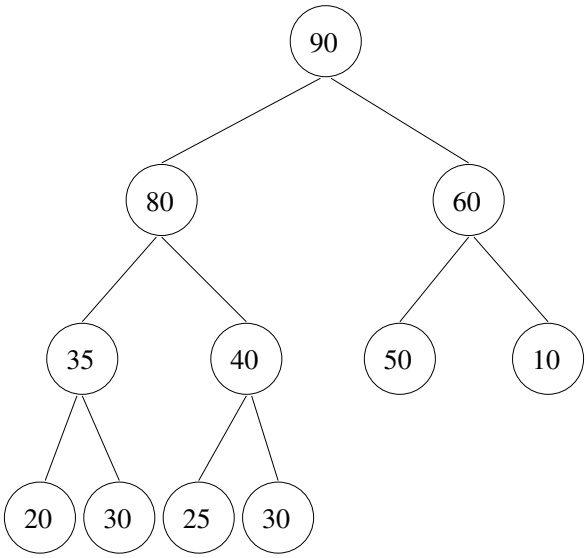
Au pire et en moyenne : O(N.log(N)). Mais expérimentalement, le tri rapide est deux à trois fois plus rapide que le tri par tas. Par contre le tri par tas n'étant pas vraiment récursif (récursivité à droite), il ne nécessite pas de pile de sauvegarde et sera donc plus intéressant pour les tris externes.

9.9.6 Exemple avec le tableau précédent

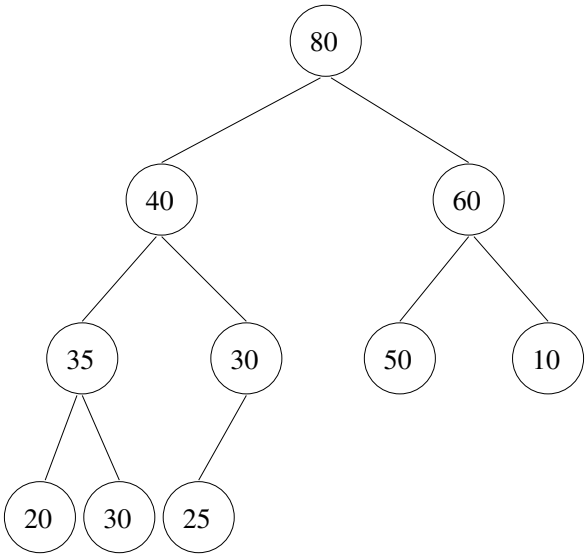
Tas initial :



Echange de 100 et 30 (le dernier élément) puis reconstruction du tas :



Echange de 90 et 30 (le dernier élément) puis reconstruction du tas :



1	2	3	4	5	6	7	8	9	10	11	12
80	40	60	35	30	50	10	20	30	25	90	100

Le tableau est trié à partir de l'indice 11.

Et ainsi de suite jusqu'au tri de tout le tableau.

9.10 Les arbres de classification

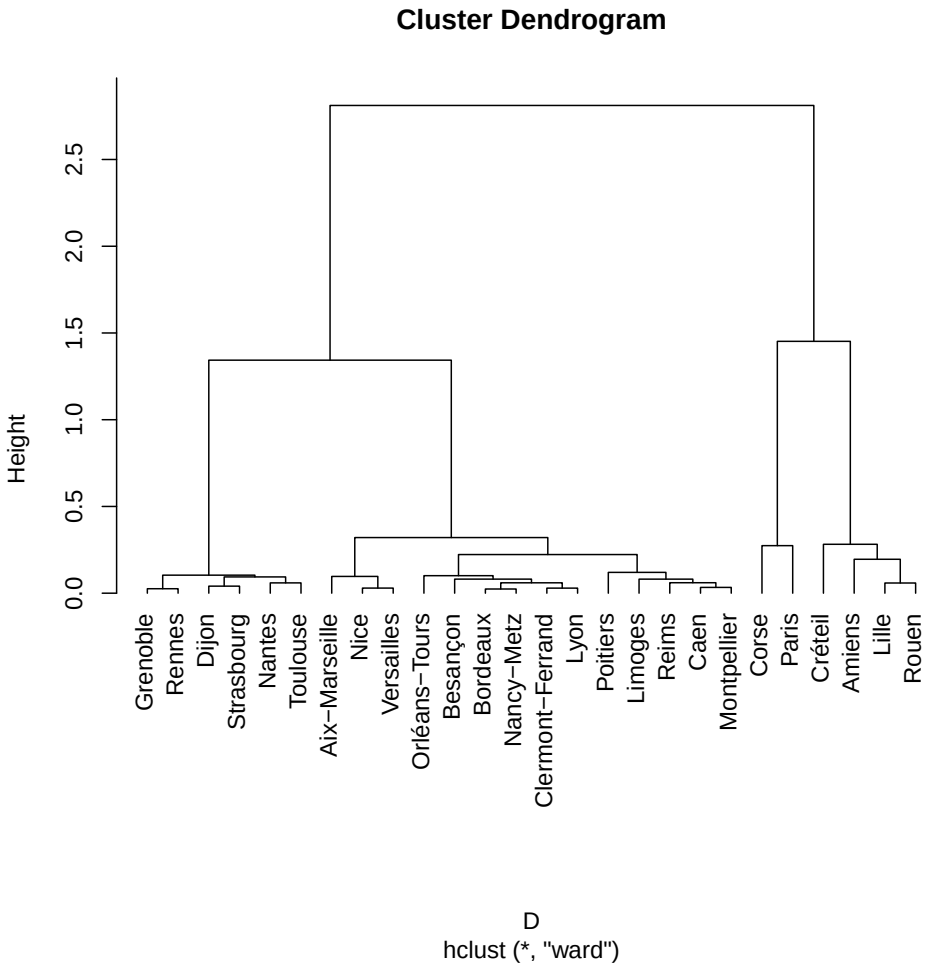
Plusieurs méthodes d'analyse de données et ou de data mining produisent des arbres comme par exemple les arbres de décisions ou les arbres résultats de CAH (Classification Ascendante Hiérarchique). Nous allons ici surtout traiter les arbres issus de CAH. Le but de la CAH consiste à trouver une suite de partitions emboîtées d'un ensemble E de N éléments. Les feuilles de l'arbre sont les éléments de E et les noeuds sont des classes constituées des feuilles sous les noeuds. 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 classes, 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 gauches et droits.

9.10.1 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.

Variante : 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
```

9.10.2 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 noeuds). 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 $c_1, \ldots, c_i, \ldots, c_p$ s'agrègent au niveau n, la représentation sera : $-n \ldots -n\ c_1, \ldots, c_i, \ldots, c_p$ avec $p-1$ fois $-n$ au début et en supposant que les classes c_i sont déjà en représentation polonaise inverse.

Chapitre 10

Représentation des 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 1 à N = card(X).

10.1 Matrice booléenne

Il s'agit de la première méthode qui vient à l'esprit. Le graphe est représenté par une matrice carrée N x N. Soit M cette matrice, on a :

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

Dans le cas d'un graphe valué, on remplace la valeur booléenne par la valeur de l'arc ($+\infty$ quand il n'y a pas d'arc).

Avantages

- accès rapide aux successeurs d'un sommet,
- accès rapide aux prédécesseurs d'un sommet,
- test rapide d'existence d'un arc,
- mise à jour rapide de l'ensemble des arcs,
- absence de problème pour les sommets isolés,
- stockage possible sur fichier.

Inconvénients

- mal adapté à la mise à jour des sommets,
- obligation de connaître le nombre de sommets,
- matrice souvent creuse,
- perte de place dans le cas non orienté.

10.2 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 chaînée. Le seul avantage de la structure de liste chaînée est sa souplesse en ce qui concerne le nombre d'arcs. 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 : c'est la structure la plus utilisée,
- recherche rapide si le tableau est trié sur le critère de recherche.

Inconvénients

- recherche longue si le tableau n'est pas trié suivant le critère de recherche,
- nécessité de connaître le nombre d'arcs,
- mise à jour longue,
- test d'existence d'un arc assez long,
- problème pour les sommets isolés.

10.3 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 chaînées. 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), la valeur de l'arc dans le cas valué et un pointeur contenant l'adresse de l'élément suivant.

Avantages

- accès rapide aux successeurs (ou prédécesseurs),
- test d'existence d'un arc assez rapide,
- les sommets isolés ne posent pas de problème,
- pas de perte de place,
- mise à jour relativement rapide de l'ensemble des arcs si connaissance de l'origine (ou extrémite pour le dictionnaire des prédécesseurs) de l'arc.

Inconvénients

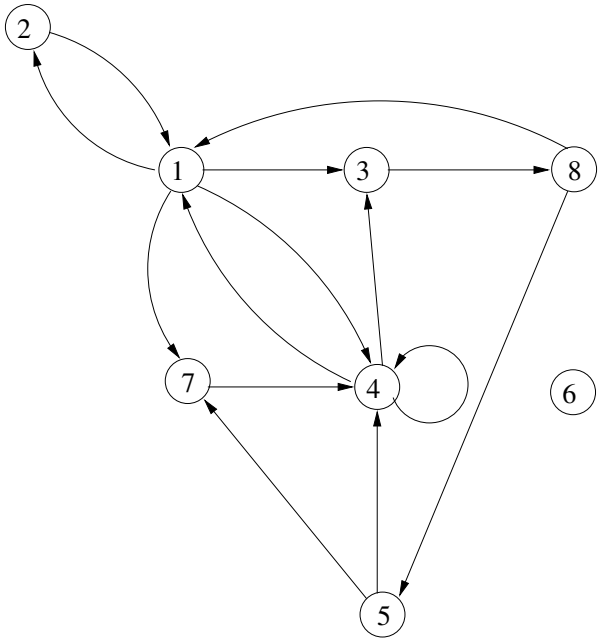
- nécessité de connaître le nombre de sommets,
- accès long aux prédécesseurs (ou aux successeurs),
- mise à jour pénible de l'ensemble des sommets,
- impossibilité de stocker des pointeurs dans un fichier.

On peut également représenter le dictionnaire des successeurs (ou des prédécesseurs) à l'aide de la représentation séquentielle indexée. Cette représentation possède l'avantage de ne pas utiliser de pointeurs : elle est donc facilement utilisable dans les langages n'ayant pas de pointeurs (comme FORTRAN : beaucoup de bibliothèque de graphes sont écrites dans ce langage) et peut être stockée sur fichier (structure malgré tout mal adaptée). Cette représentation nécessite la connaissance du nombre de sommets. Pour le reste, elle possède les mêmes avantages et les mêmes inconvénients que la structure de tableau de listes.

Elle est constituée de deux tableaux EXTREMITE et FIN. On se place dans le cas du dictionnaire des successeurs. Le tableau EXTREMITE est indexé par l'ensemble des arcs (de taille maximum N*N). Il contient tous les successeurs du premier sommet puis du deuxième sommet et ainsi de suite jusqu'au dernier sommet. Le tableau FIN est indexé par l'ensemble des sommets. FIN[i] est l'indice du dernier successeur du sommet x_i dans le tableau EXTREMITE. De plus on pose FIN[0]=0. Ainsi l'ensemble des successeurs du sommet x_i $(1 \leq i \leq N)$ est dans le tableau EXTREMITE entre les indices FIN[i-1]+1 et FIN[i]. Il existe une variante de cette représentation en ne stockant pas l'indice de fin mais l'indice de début.

10.4 Exemple

Soit le graphe suivant :



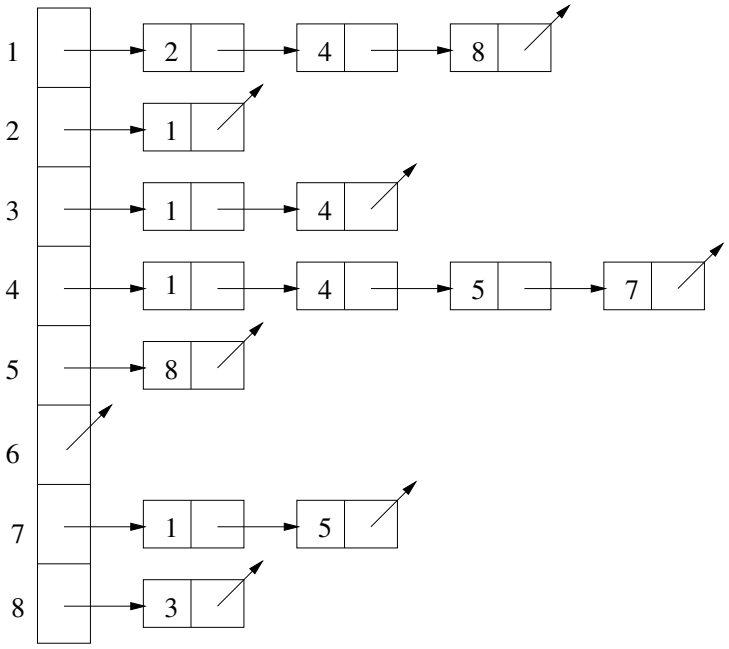
Dictionnaire des arcs :

```
(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 forme d'un tableau de listes chaînées :



Chapitre 11

Les automates finis ou automates à états

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.

11.1 Les expressions rationnelles

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

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 $\emptyset$),
- si α et β sont deux expressions régulières sur V alors : $\alpha|\beta, \alpha\beta, \alpha*$, α^+ et (α) 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 :

- . : n'importe quel caractère excepté le caractère retour-chariot,
- \n : caractère retour-chariot,

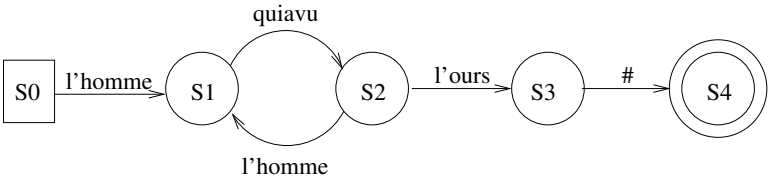
- - : délimiteur d'intervalle, par exemple, une lettre sera reconnue par [a-zA-Z] et un entier par [0-9],
- ? : zéro ou une occurrence,
- ...

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

11.2 Automate fini

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	-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 [AHO]. Dans la suite, nous ne considérerons que des automates déterministes.

11.2.1 Algorithme général pour un automate déterministe

On suppose définies les fonctions suivantes :
- lire_symb() : fonction de lecture du prochain symbole,
- transit(etat, symb) : fonction entière correspondant à la table des transitions,

– final(etat) : 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(état, 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.

11.2.2 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 S2 et S3, 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	l'ours	#
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 de personnes intermédiaires : ", nbint)
  # sinon on ne fait rien
```

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

11.3 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. Un TP sur les automates finis sera réalisé en C.

Chapitre 12

Les tris externes

12.1 Introduction

Les tris externes sont utilisés quand les données ne tiennent pas en mémoire centrale. Les méthodes de tri externe s'articulent autour de deux étapes principales :

- la construction des monotonies,
- l'interclassement de monotonies.

Une monotonie est une suite d'éléments triés entre eux. Dans un tri externe, l'opération la plus coûteuse est l'accès fichier (lecture ou écriture).

12.2 Construction des monotonies

12.2.1 Monotonies de même taille

Soient N le nombre d'éléments à trier et M le nombre d'éléments pouvant être chargés simultanément en mémoire centrale. On partitionne les N éléments en N/M sous listes. Chaque sous liste est ensuite transformée en monotonie par application d'une méthode de tri interne, pouvant être le tri par tas (attention aux tris récursifs).

Nombre de lectures : N
Nombre d'écritures : N

12.2.2 Monotonies de taille variable

Principe :

- chargement de M éléments en mémoire centrale,
- suppression du minimum non marqué (il rejoint le fichier contenant la monotonie courante),
- lecture de l'élément suivant : si il est inférieur à l'ancien minimum alors on le marque,
- on réitère jusqu'à ce que tous les éléments soient marqués (changement de monotonie) : retour à l'étape numéro 2 après avoir inversé le marquage,
- on arrête quand les N éléments ont rejoint une monotonie.

Algorithme avec utilisation du tri par tas

```
def construction_monotonies_tas(M, Fent, Fmono):
  # Fent : fichier d'entrée
  # Fmono : fichier d'écriture des monotomies
  #         dans la pratique les monotomies sont écrites
  #         alternativement dans au moins 2 fichiers

  Fent = open("donnees.txt","r")
  Fmono = open("mono.txt","w")
  T = []
  # chargement des M premiers éléments dans le tableau T
  for i in range(M) :
    x  = Fent.readline()
    T.append(x)

  creer_tas_min(M,T)   # création d'un tas avec minimum à la racine
  longueur_tas = M
  monot:= 1            # numérotation de la monotonie
  while x != "" :      # test de fin de fichier
    while (longueur_tas > 0) and (x != "") :
      Fmono.write(T[0])
      if x >= T[0] :
        T[0] = x
        reconstruire_tas_minimum(longueur_tas,T)
      else :
        longueur_tas = longueur_tas-1
        T[0] = T[longueur_tas]
        T[longueur_tas] = x

      x  = Fent.readline()
     # fin boucle interne

    if x != "" :
       monot = monot+1
       creer_tas_min(M,T)
       longueur_tas := M
  # fin boucle principale

  vider_tas_courant(Fmonot,longueur_tas,T)
  trier(longueur_tas+1,M-1,T)   // tri des éléments marqués
  monot := monot+1
  for i in range(longueur_tas,M):
    Fmono.write(TI])
  Fent.close()
  Fmono.close()
```

Intérêts :

- même nombre d'accès fichier : N lectures et N écritures.
- La longueur des monotonies est beaucoup plus grande que pour la méthode précédente. ceci va contribuer à minimiser le nombre d'interclassements et

par conséquent, le nombre d'accès fichier.

12.3 Interclassement des monotonies

12.3.1 Algorithme

L'algorithme d'interclassement de deux monotonies est déjà étudié. En exercice, étendre cet algorithme à l'interclassement de plusieurs (>2) monotonies.

12.3.2 Placement des monotonies

Il faut dans cette phase, déterminer sur quels fichiers seront placées les monotonies. Cette phase est délicate car un mauvais placement des monotonies peut entraîner un blocage du processus. Un tel blocage peut provoquer N lectures et N écritures sur fichier.

12.3.3 Le tri équilibré

Avec quatre fichiers

Deux des fichiers sont utilisés en lecture et deux en écriture. Les monotonies initiales sont stockées sur deux fichiers. Les monotonies de rang pair sont mises dans le deuxième fichier et les monotonies de rang impair sont mises dans le premier fichier. On interclasse ensuite les monotonies des deux fichiers. Les monotonies résultat sont mises à tour de rôle dans l'un puis l'autre des fichiers en écriture lors de cette phase. On recommence le processus jusqu'à l'obtention d'une monotonie unique : tous les éléments sont alors triés. Cette méthode est appelée tri équilibré à deux voies. A programmer en exercice.

Généralisation

La méthode générale est la même mais on utilise p fichiers en entrée et p fichiers en sortie. La procédure de fusion devient plus compliquée. On parle de tri équilibré à p voies. Plus p est grand, plus le nombre d'accès fichier est faible.

12.3.4 Le tri polyphasé

Cette méthode utilise p ($p \geq 3$) fichiers. A un instant donné, p-1 fichiers sont utilisés en lecture et le dernier fichier est utilisé en écriture. Dès qu'un fichier en entrée devient vide, ce fichier passe en écriture et l'ancien fichier en écriture passe en lecture. On appelle **phase** une suite de fusions où les fichiers conservent le même rôle. Le principal problème consiste à répartir les monotonies initiales sur les fichiers d'entrée afin d'éviter le problème du blocage.

Avec trois fichiers

On utilise la suite de Fibonacci. Soit fib cette fonction définie comme suit :
- fib(1) = fib(2) = 1,
- fib(n) = fib(n-1) + fib(n-2), si n > 2.

Si on est en présence de fib(n) monotonies, on place fib(n-1) monotonies sur un fichier et fib(n-2) monotonies sur l'autre fichier. Le cas échéant, on complète un fichier avec des monotonies virtuelles (vides).

Généralisation pour p fichiers

On obtient la répartition des monotonies par généralisation de la suite de Fibonacci : f(i,j) pour $1 \leq i \leq p$ représente le nombre de monotonies en entrée sur le fichier i, j phases avant la fin du tri. Cette fonction est définie comme suit :

- f(p,j) = 0,
- f(i,1) = 1 si $i < p$,
- f(i,j) = f(i+1,j-1) + f(1,j-1) si $j > 1$ et $i < p$.

Exemple avec 5 fichiers et 45 monotonies : fonction f(i,j) :

(*) j / i	1	2	3	4	F(j) = $\sum$
0	1	0	0	0	1
1	1	1	1	1	4
2	2	2	2	1	7
3	4	4	3	2	13
4	8	7	6	4	25
5	15	14	12	8	49

(*) avec renumérotation des fichiers

Utilisation de 4 monotonies virtuelles pour compléter à 49.

Exemple de fonctionnement : avec 5 fichiers et 10 monotonies. On a : F(2) $< 10) \leq$ F(3). On en déduit que trois phases seront nécessaires. On utilisera également trois monotonies virtuelles V1 à V3. Les monotonies sont notées M1 à M10.

Trois phases avant la fin (situation initiale)

F1	F2	F3	F4	F5
V1	V2	V3	M4	$\emptyset$
M1	M2	M3	M8	
M5	M6	M7		
M9	M10			

Deux phases avant la fin

F1	F2	F3	F4	F5
M5	M6	M7	$\emptyset$	M4
M9	M10			M1+M2+M3+M8

Une phase avant la fin

F1	F2	F3	F4	F5
M9	M10	$\emptyset$	M4+M5+M6+M7	M1+M2+M3+M8

Situation finale

F1	F2	F3	F4	F5
∅	∅	M1 à M10	∅	∅

Bibliographie

[1] A. V. AHO, M. S. LAM, R. SETHI et J. D. ULLMAN. *Compilers : Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, 2006.

[2] C. CARREZ. *Des structures aux bases de données.* Dunod Informatique, 1990.

[3] J. COURTIN et I. KOWARSKI. *Initiation à l'algorithmique et aux structures de données.* Volumes 1, 2 et 3. Dunod Informatique, 1990.

[4] M.C. GAUDEL, M. SORIA et C. FROIDEVAUX. *Types de données et algorithmes.* Volumes 1 et 2. Collection didactique INRIA, 1988.

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