

Université Paris Est, Créteil
Département d'Informatique



Algorithmique

Daniele Varacca

Version 1.02 du 03 Juin 2019

Adresse de l'auteur :

Daniele Varacca
LACL
Université Paris Est, Créteil
France

Email : `daniele.varacca@u-pec.fr`

Copyright © Daniele Varacca 2008 - 2019.



Cet œuvre est protégé sous une licence *creative commons Paternité - Pas d'Utilisation Commerciale - Pas de Modification 2.0 France*. Vous êtes libres :

- de reproduire, distribuer et communiquer cette création au public

Selon les conditions suivantes :

- Paternité : Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).
- Pas d'Utilisation Commerciale : Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.
- Pas de Modification. Vous n'avez pas le droit de modifier, de transformer ou d'adapter cette création.

À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le text intégral de la licence est disponible à l'adresse <http://creativecommons.org/licenses/by-nc-nd/2.0/fr/legalcode>.

Ces notes sont un ébauche d'un cours d'algorithmique. J'ai estimé nécessaire les écrire, car aucun des livres d'algorithmique à ma connaissance ne pouvait être proposé à des étudiants avec une expérience limitée en informatique.

Ces notes sont encore incomplètes, faute manque de temps. Elles contiennent également encore beaucoup de fautes de Français, et, j'espère, un peu moins de fautes de contenu.

À tous ceux qui m'aideront à corriger les erreurs, je n'offrirai pas 2.56 dollars, mais ma gratitude. En particulier je tiens à remercier Ralf Treinen, Sylvain Perifel, Luc Boasson, Christian Choffrut et Giulio Manzonetto.

DV

Table des matières

1	Premiers exemples d'algorithmes	11
1.1	Tri par insertion	12
1.2	Tri par bulles	17
1.3	Des chiffres (mais pas des lettres)	18
1.4	Tri par essais	19
1.5	Trouver le max	20
1.6	Trouver un nombre	20
2	La Complexité d'un algorithme	21
2.1	Complexité	22
2.2	L'ordre de grandeur des fonctions	22
2.3	Exemples	23
2.4	Les pas	24
3	Récursion	25
3.1	Nommer un algorithme	26
3.2	Récursion	27
3.3	Diviser pour Régner : Tri par fusion	28
3.4	Complexité du Tri par fusion	30
3.5	Le Théorème du coût récursif	31
3.6	Tri rapide	32
4	Listes, Files et Piles	33
4.1	Les objets, et les pointeurs	34
4.2	Listes chaînées	36
4.3	Objet Liste	37
4.4	Liste doublement chaînée	39
4.5	Files et Piles	40

5 Arbres	43
5.1 Arbres Binaires Simples	44
5.2 Parcours d'un arbre binaire	45
5.3 Arbres Binaires Avec Prédécesseur	45
5.4 Arbres Binaires de Recherche	45
6 Le Tas	49
6.1 Arbres parfaits et tableaux	50
6.2 Les Tas	50
6.3 Tri par Tas	53
7 Tables de hachage	57
7.1 Résoudre les conflits	58
7.2 Réduire les conflit	59
7.3 Exemple	59
8 Autres sujets	61

Introduction

Un algorithme est une suite de pas à suivre pour atteindre un objectif. Normalement l'objectif est la résolution d'un problème. On dit qu'un algorithme est *correct* pour un problème si, en suivant les pas de l'algorithme, on résout le problème. Pour des petits algorithmes, c'est relativement facile de vérifier s'ils sont corrects ou pas, mais en général il faut une preuve mathématique rigoureuse pour montrer la correction d'un algorithme.

Pour donner un exemple, si le problème est de manger un gâteau au chocolat, un algorithme correct pourrait être le suivant

- mélanger 150 grammes de sucre avec 150 grammes de beurre
- ajouter 2 oeufs, un verre de lait, un peu de sel et de la cannelle
- ajouter lentement 300 grammes de farine, 100 grammes de cacao et de la levure
- mettre dans une forme
- cuire dans un four à 180 degrés pendant 30 minutes
- servir dans une assiette
- amener la fourchette à la bouche

En principe cette description devrait être assez précise pour ne pas donner lieu à des ambiguïtés mais devrait également être assez abstraite pour ne pas spécifier trop de détails. Par exemple on ne spécifie pas s'il faut utiliser de la farine blanche ou complète, si le four doit être électrique ou à gaz. On ne dit pas à quelle heure il faut cuisiner ni les mouvements exacts des bras. Ces détails concernent l'*implémentation*¹ d'un algorithme. La ligne de démarcation entre un algorithme et une implémentation n'est pas précise. En général, les algorithmes décrivent plus abstraitement les pas à faire.

Très souvent un problème se présente en forme *paramétrique*, c'est-à-dire que tous les détails du problème ne sont pas spécifiés. En spécifiant ces détails, on obtient une *instance* du problème. En général un algorithme décrit les pas nécessaires à la solution de toutes les instances d'un problème. Par exemple, on considère le problème de connaître la somme de deux nombres entiers. Ici les paramètres du problème sont les deux nombres. Une instance du problème est connaître la somme de 45 et 17. Or, on a tous appris l'algorithme pour l'addition de deux nombres.

- écrire les deux nombres, un au dessus de l'autre
- ajouter éventuellement des zéro en tête du nombre plus court jusqu'à que les deux ont la même taille
- écrire une ligne au dessous des deux nombres
- tant que il y a des colonnes non marquées, répéter les pas suivants:
 - choisir la colonne la plus à droite parmi celles qui ne sont pas encore marquées
 - additionner les chiffres de la colonne et l'éventuel report
 - si le résultat a deux chiffres, écrire le chiffre de droite en bas de la colonne, sous la ligne, et reporter le chiffre de gauche
 - marquer la colonne
- si il y a un report qui reste, l'écrire devant les autres chiffres du résultat

1. Ce mot n'est pas toujours accepté comme Français correct, et il est parfois remplacé par "implantation", ce qui pourtant ne traduit pas correctement le mot anglais "implementation".

Cet algorithme s'applique à toute instance du problème et amène à sa solution.

Encore une fois cette description évite les ambiguïtés sans spécifier trop de détails. Par exemple on ne spécifie pas s'il faut écrire avec l'encre rouge ou noir, s'il faut utiliser le papier ou le tableau. Ces détails concernent l'implémentation.

On aurait pu choisir une autre façon de décrire le même algorithme. Serait-il donc le *même* algorithme ? Quand est-ce que deux algorithmes sont le même ? Il n'y a pas une réponse exacte. Cependant, comme il ne faut pas confondre un algorithme avec ses implémentations, il ne faut pas confondre un algorithme avec l'énoncé qui le décrit.

Quand on a plusieurs algorithmes pour résoudre un même problème, lequel faudrait-il choisir ? Lequel serait à considérer comme le meilleur ? On pourrait choisir l'algorithme le plus facile à implémenter, le plus court à décrire, le plus facile à être prouvé comme étant correct. Mais la caractéristique la plus importante est l'*efficacité*. On mesure l'efficacité d'un algorithme par le nombre de pas qu'il amène à exécuter. Bien sûr ce nombre dépend non seulement de l'algorithme même, mais aussi de l'instance spécifique du problème. Additionner 45 et 17 requerra moins de pas que additionner 123643654 et 534643556. Donc on mesurera le nombre de pas à exécuter en fonction de la dimension des données d'entrée.

En général on décrira les algorithmes de façon que chaque pas puisse être exécuté approximativement dans un même délai de temps. Cela nous amène à identifier le nombre de pas avec le *temps d'exécution*.

Bien qu'un algorithme puisse être destiné à un humain (comme pour la recette du gâteau), on s'intéresse surtout à des algorithmes destinés à être implémentés sur des machines, et en particulier sur des ordinateurs électroniques. Pour pouvoir implémenter un algorithme il faut savoir comment faire faire à l'ordinateur les pas nécessaires dans l'ordre spécifié. Cela requiert la connaissance d'un *langage de programmation*. L'étude des algorithmes est largement indépendante de l'étude des langages de programmation, car la plupart des algorithmes peuvent être implémentés dans n'importe quel langage. Toutefois certaines langages sont plus adaptés que d'autres à de différents types d'algorithme.

Dans ces notes on utilisera un *pseudo-code* pour décrire les algorithmes. Le pseudo-code est une forme intermédiaire entre la langue parlée et un vrai langage de programmation. L'utilisation du pseudo-code nous permet un certain degré de précision, sans pourtant faire un choix défini sur le langage utilisé dans l'implémentation.

Les algorithmes informatiques prévoient l'accès, la modification, la destruction et la création de données. La façon avec la quelle on organise ces données influence l'efficacité d'un algorithme. Pour cette raison, à l'étude des algorithmes s'accompagne l'étude des *structures de données*. Différentes structures sont associées à de différents algorithmes. Certaines structures permettent un accès rapide aux données existantes, mais permettent avec difficulté la création de nouvelles données. Certaines structures permettent un accès rapide aux données seulement dans un certain ordre. On verra quelles structures sont adaptées à quels algorithmes.

Chaque branche de l'informatique utilise des algorithmes différents. La vérification des programmes, la cryptographie, l'optimisation, la bioinformatique, ont leurs propres spécificités. Ce cours se donne l'objectif d'introduire quelques techniques de base.

Dans le Chapitre 1 on introduira quelques premiers exemples d'algorithmes. Dans le Chapitre 2, on présentera les techniques utilisées pour évaluer le temps d'exécution d'un algorithme. Dans le Chapitre 3, on introduira d'autres exemples d'algorithmes qui utilisent la technique de la *réursion*.

Dans la deuxième partie on présentera différents algorithmes associés à de différentes *structures de données*.

Ces notes s'inspirent du livre [CLRS02].

Chapitre 1

Premiers exemples d'algorithmes

Les premiers algorithmes qu'on va étudier sont les algorithmes de *tri*. Étant donnée une suite de n nombres, il s'agit de la trier en ordre croissant.

Pour résoudre ce problème il faut d'abord préciser quelles opérations on a le droit de faire. Imaginons donc les nombres comme s'ils étaient dans un casier, un nombre dans chaque case :

4	5	1	9	12	32	4	7	15	17
---	---	---	---	----	----	---	---	----	----

Pour ces premiers algorithmes, on aura le droit de

- copier les nombres sur un "espace de travail",
- comparer deux nombres et dire le quel est le plus grand,
- mettre un nombre dans une case, effaçant son contenu précédent.

Sur notre espace de travail on a le droit de créer une ou plusieurs copies de chaque nombre et même du casier entier.

Pour évaluer l'efficacité de nos algorithmes, il est important que les éléments à trier soient des nombres, c'est-à-dire des entités simples, faciles à manipuler, à déplacer, à effacer. On verra dans la suite comment on pourra manipuler des entités plus complexes.

Progressivement, on verra également quelles autres opérations on aura le droit de faire pour résoudre d'autres problèmes.

1.1 Tri par insertion

Cet algorithme est l'algorithme qu'on utilise quand on joue aux cartes, et on veut tenir en main les cartes en ordre croissant de gauche à droite. On prend chaque carte, une à une, et on l'insère à sa place, parmi les cartes qu'on a déjà en main. Avant de décrire cet algorithme en termes de casiers et de nombres, on va le décrire en termes de mains et de cartes.

On imagine que toutes les cartes soient, au début, devant nous sur la table. Voici une description de l'algorithme :

1. prendre la prochaine carte de la table, s'il y en a, et la mettre le plus à droite possible
2. s'il n'y a pas de cartes à gauche de la carte qu'on vient de prendre, recommencer de (1)
3. sinon comparer la carte qu'on vient de prendre avec la carte à sa gauche
4. si la carte qu'on vient de prendre est plus petite
5. échanger les deux cartes
6. recommencer de (2)
7. si la nouvelle carte est plus grande ou égale, recommencer de (1)

Il faudrait prouver formellement que cet algorithme est correct, c'est-à-dire qu'il se termine, et que à la fin les cartes sont effectivement triées. Cependant, tous ceux qui ont jamais joué aux cartes savent que cet algorithme est correct. La description qu'on a donné utilise un peu trop de mots. On pourrait la simplifier en donnant des noms aux entités, comme on fait souvent en mathématique.

1. s'il y en a, prendre la prochaine carte de la table, appelons-la c , et la mettre le plus à droite possible,
2. s'il n'y a pas de cartes à gauche de c , recommencer de (1)
3. s'il y a une carte à gauche de c , appelons-la d
4. si c est plus petite que d
5. échanger c et d
6. recommencer de (2)
7. si c est plus grande ou égale à d
8. recommencer de (1)

On change maintenant un peu la description pour se rapprocher aux langages de programmation. En lieu de écrire explicitement "recommencer de" on dit quelles instructions il faut répéter. L'indentation du texte nous dira quelles sont les instructions à répéter.

1. à partir de la premier carte sur la table, pour chaque carte c sur la table répéter :
2. mettre c en main, le plus à droite possible
3. à partir de la carte à gauche de c , pour chaque carte d en main
4. tant que c est plus petite que d répéter :
5. echanger c et d
6. passer à la prochaine carte en main
7. passer à prochaine carte sur la table

On peut maintenant sortir de la métaphore et passer des nos mains aux casiers, des carte aux nombres.

1. créer un casier vide sur l'espace de travail
2. à partir de la case 1 en montant pour chaque case i du casier principal répéter :
3. mettre le nombre de la case i du casier principal, dans la case i du casier de travail
4. à partir de la case $i - 1$ du casier de travail, en descendant, pour chaque case j du casier de travail
5. tant que le nombre de la case $j + 1$ est plus petit que le nombre de la case j répéter :
6. échanger le contenu des case j et $j + 1$
7. passer à la case suivante du casier de travail
8. passer à la case suivante du casier principal

Encore une fois trop de mots. Mieux serait de donner un nom aux casiers. On utilisera aussi la convention suivante : si A est le nom d'un casier, par $A[i]$ on dénote le nombre contenu dans la i -ème case de A . Dans la suite, on appellera C le casier à trier.

1. créer un casier vide D
2. pour chaque case i de C , répéter :
 3. copier $C[i]$ dans la case i de D .
 4. pour chaque case j de D , à partir de $j = i - 1$ en descendant,
 5. tant que $D[j + 1] < D[j]$ répéter :
 6. échanger $D[j]$ et $D[j + 1]$
 7. passer à la case suivante de D
 8. passer à la case suivante de C

Pour décrire plus formellement ces opérations, on introduit le *pseudo-code*, une notation intermédiaire entre la langue parlée et un langage de programmation. On utilisera la notation suivante :

- Quand on écrit $i \leftarrow expr$
on veut dire "prends la valeur dénotée par $expr$ et mets la comme nouvelle valeur dénotée par i . L'ancienne valeur de i est oubliée." En particulier on écrira $C[i] \leftarrow D[j]$ pour dire "prends le nombre qui se trouve dans la case j du casier D et mets le dans la case i du casier C . L'ancien contenu de la case i du casier C est effacé." Aussi on écrira $i \leftarrow i - 1$ pour décrémenter de 1 le contenu de i .
- Quand on écrit **tant que condition faire** : $instructions$
on veut dire : répète les instructions tant que la condition est vraie. L'indentation du texte nous dira quelles sont les instructions à répéter.
- Quand on écrit **pour** $i \leftarrow 1$ **à** n **faire** : $instructions$
on veut dire : commence avec $i \leftarrow 1$. Après exécute les instructions et à la fin incrémente i de 1. Répète cela jusqu'à que $i = n$, à ce moment là on s'arrête.
- Quand on écrit **si condition faire** : $instructions$
on veut dire : exécute les instructions seulement si la condition est vraie. On pourra ajouter aussi **sinon faire** : $instructions-2$
en ce cas on exécutera les instructions-2 si la condition est fausse.

Avec ces conventions on peut décrire plus précisément et plus brièvement notre algorithme de tri. On dénotera par n le nombre de cases du casier à trier.

1. créer un casier vide D de taille n
2. **pour** $i \leftarrow 1$ **à** n **faire** :
 3. $D[i] \leftarrow C[i]$
 4. $j \leftarrow i - 1$
 5. **tant que** $j > 0$ et $D[j + 1] < D[j]$ **faire** :
 6. $temp \leftarrow D[j + 1]$
 7. $D[j + 1] \leftarrow D[j]$
 8. $D[j] \leftarrow temp$
 9. $j \leftarrow j - 1$

On observe que pour échanger $D[j]$ et $D[j + 1]$ on a eu besoin de stocker une valeur dans une variable "temporaire". On peut un peu simplifier : en lieu d'insérer la valeur de $C[i]$ au début et échanger, on insère la valeur de $C[i]$ seulement à la fin, seulement quand on a trouvé sa place.

1. créer un casier vide D de taille n
2. **pour** $i \leftarrow 1$ **à** n **faire** :
3. $j \leftarrow i - 1$
4. **tant que** $j > 0$ et $C[i] < D[j]$ **faire** :
5. $D[j + 1] \leftarrow D[j]$
6. $j \leftarrow j - 1$
7. $D[j + 1] \leftarrow C[i]$

On commence maintenant à avoir quelques difficultés à comprendre ce qui se passe. Pour cela on ajoute des *commentaires*. Un commentaire sert à expliquer mieux ce qui se passe, mais il ne constitue pas un pas de l'algorithme. Un commentaire est précédé par les symboles *//*.

1. créer un casier vide D de taille n
2. **pour** $i \leftarrow 1$ **à** n **faire** :
3. $j \leftarrow i - 1$
4. **tant que** $j > 0$ et $C[i] < D[j]$ **faire** :
5. $D[j + 1] \leftarrow D[j]$
6. *// on déplace le nombre vers la droite*
7. $j \leftarrow j - 1$
8. *// on passe à la case suivante de D , en descendant*
9. $D[j + 1] \leftarrow C[i]$
10. *// quand $j = 0$ ou bien quand $C[i] \geq D[j]$ on insère le nombre*

Finalement, avec un peu d'astuce, on n'a pas besoin de copier le casier. On déplace les nombres dans le casier de départ. Il suffit de copier un nombre à la fois. Pour faire cela on utilise une seule case de l'espace de travail, qu'on appelle *cle*.

1. **pour** $i \leftarrow 1$ **à** n **faire** :
2. $cle \leftarrow C[i]$
3. $j \leftarrow i - 1$
4. **tant que** $j > 0$ et $cle < C[j]$ **faire** :
5. $C[j + 1] \leftarrow C[j]$
6. *// on déplace le nombre vers la droite*
7. $j \leftarrow j - 1$
8. *// on passe à la case suivante en descendant*
9. $C[j + 1] \leftarrow cle$
10. *// quand $j = 0$ ou bien quand $cle \geq C[j]$ on insère le nombre*

On observe que, bien que on ait présenté plusieurs textes différents, ils sont tous descriptions du même algorithme. Pour utiliser une métaphore biologique, ils sont différents *individus* de la même *espèce*.

On présente l'application de l'algorithme à l'instance du problème qui suit.

$i = 1$	$i = 2$	$i = 3$	$i = 4$
3	5	1	9

- On met $i \leftarrow 1$.
- On vérifie que $i \leq 4$, donc on exécute ce qui suit. On met $cle \leftarrow C[i]$ et $j \leftarrow i - 1$. On a que $j \leq 0$, donc on n'exécute pas la boucle intérieure, on fait $C[j + 1] \leftarrow cle$. On met $i \leftarrow 2$.
- On vérifie que $i \leq 4$, donc on exécute ce qui suit. On met $cle \leftarrow C[i]$ et $j \leftarrow i - 1$. On a que $j > 0$, mais $cle \geq C[j]$ donc on n'exécute pas la boucle intérieure, on fait $C[j + 1] \leftarrow cle$. On met $i \leftarrow 3$.
- On vérifie que $i \leq 4$, donc on exécute ce qui suit. On met $cle \leftarrow C[i]$ et $j \leftarrow i - 1$. On a que $j > 0$ et $cle < C[j]$ donc on commence la boucle intérieure
 - on met $C[j + 1] \leftarrow C[j]$ et on diminue j
 - on vérifie que $j > 0$ et que $cle < C[j]$, donc on met $C[j + 1] \leftarrow C[j]$ et on diminue j
 - on vérifie que $j \leq 0$ et donc on termine la boucle
 on fait $C[j + 1] \leftarrow cle$ et on met $i \leftarrow 4$.
- On vérifie que $i \leq 4$, donc on exécute ce qui suit. On met $cle \leftarrow C[i]$ et $j \leftarrow i - 1$. On a que $j > 0$, mais $cle \geq C[j]$ donc on n'exécute pas la boucle intérieure, on fait $C[j + 1] \leftarrow cle$. On met $i \leftarrow 5$.
- On vérifie que $i > 4$ et donc on termine la boucle

1	3	5	9
---	---	---	---

On peut compter le nombre de pas. Il y a plusieurs possibilité pour définir un pas. Ici on choisira de considérer chaque ligne du pseudo code comme un pas. A chaque tour de la boucle principale on doit :

- mettre à jour la valeur de i
- vérifier si $i \leq n$ et si c'est le cas :
- copier $C[i]$ dans cle
- initialiser j
- vérifier si $j > 0$ et $cle < C[j]$ et éventuellement faire la boucle intérieure pour la valeur courante de i
- mettre la valeur de cle dans $C[j + 1]$

A chaque tour de la boucle principale on doit donc faire 6 pas plus la boucle intérieure. A chaque tour de la boucle intérieure on doit :

- copier $C[j]$ dans $C[j + 1]$
- copier $j - 1$ dans j
- vérifier si $j > 0$ et $cle < C[j]$

Cela fait 3 pas.

Dans l'exemple ci-dessus, on exécute 4 fois la boucle principale, et deux fois la boucle intérieure, et on ajoute deux pas finaux (mise à jour de i et vérification). On a donc : $6 + 6 + 6 + (3 + 3) + 6 + 2 = 32$. Est-il efficace ? Pourrions-nous faire mieux ?

Considérons une autre instance.

9	5	3	1
---	---	---	---

Si on compte le nombre de pas ici on vérifiera que c'est 44. Pour la même longueur on a un nombre différent de pas. Cela est normal, car le nombre de pas dépend de l'ordre initial. Il n'est pas difficile de réaliser que pour un casier de longueur donnée, on exécute le plus grand nombre de pas si l'ordre initial est l'inverse de l'ordre naturel. Ceci est le *pire des cas*.

On peut calculer le nombre de pas pour toute suite en ordre inverse de n nombres. Combien de fois on exécute la boucle intérieure? La condition $cle < C[j]$ sera toujours vraie, car la suite est en ordre inverse, donc la clé sera plus petite que tous les nombres déjà triés. Donc la boucle intérieure s'arrêtera seulement quand $j = 0$.

A chaque tour i de la boucle principale on fait $6 + BI(i)$ pas, où $BI(i)$ est le nombre de pas faits par la boucle intérieure. La boucle intérieure part avec $j = i - 1$ et s'arrête quand $j = 0$. On exécute la boucle intérieure $i - 1$ fois, et chaque fois on fait 3 pas. $BI(i) = 3(i - 1) = 3i - 3$. Finalement on fait le deux pas finaux. Le temps d'exécution est donc

$$T(n) = (6 + (3 - 3)) + (6 + (6 - 3)) + \dots + (6 + (3n - 3)) + 2$$

En réordonnant les termes on obtient :

$$T(n) = 2 + (6 - 3) * n + 3 * (1 + 2 + \dots + n)$$

Quelle est la valeur de $1 + 2 + \dots + n$?

Quand le petit Carl Friedrich Gauss était à l'école primaire, son professeur demanda à sa classe d'additionner tous les nombres de 1 à 100, pour les faire rester tranquilles pendant un peu de temps. Mais le petit Gauss répondit en quelques secondes. Il avait observé qu'on peut regrouper les nombres deux à deux : $(1 + 100) + (2 + 99) + (3 + 98) + \dots + (50 + 51)$. Chaque groupe vaut 101 et il y a 50 groupes. Le résultat est donc 5050. Plus en général : $1 + 2 + \dots + n = ((n + 1) * n) / 2$ (Attention au cas où n est impair!)

Donc

$$T(n) = 2 + 3 * n + 3 * (n^2 + n) / 2 = 2 + 9/2 * n + 3/2 * n^2$$

On verra que ce qui compte le plus est le monôme $3/2 * n^2$. On dira donc que l'algorithme de tri par insertion a un temps d'exécution *quadratique* dans le *pire des cas*. Pour avoir une idée de la rapidité, si on imagine que on peut faire un pas en un microseconde, on peut trier 1000 nombres en moins de deux secondes, 10000 nombres en moins de trois minutes, mais pour trier l'annuaire de Paris (~ 1000000 de familles) il nous faudrait, dans le pire des cas, plus que deux semaines. On verra comment on peut faire mieux que cela.

1.2 Tri par bulles

Un autre algorithme de tri consiste à imaginer que le tableau soit posé verticalement sur le sol, et que les nombres les plus grands aient la tendance à remonter vers le haut, comme des bulles de gaz dans le soda. Cette intuition amène à l'algorithme de *Tri par bulles* (Bubblesort). On parcourt le tableau, et si on rencontre deux nombres adjacents qui ne sont pas dans le bon ordre, on les échange. On répète le parcours du tableau tant que on a des échanges à faire. Pour savoir quand il faut terminer, on utilise un *drapeau* qui enregistre s'il y a eu d'échanges. À chaque parcours, on commence par mettre le drapeau à 1. Si au début du parcours suivant, on voit que le drapeau est toujours à 1, on sait qu'il n'y a pas eu d'échanges, et donc on doit terminer.

```

1.  $n \leftarrow \text{longueur}(C)$ 
2.  $\text{fini} \leftarrow 0$ 
3. // initialisation du drapeau
4. tant que  $\text{fini} = 0$  faire :
5.      $\text{fini} \leftarrow 1$ 
6.     pour  $i \leftarrow 1$  à  $n - 1$  faire :
7.         si  $C[i] > C[i + 1]$  faire :
8.             // on échange les deux
9.              $\text{temp} \leftarrow C[i]$ 
10.             $C[i] \leftarrow C[i + 1]$ 
11.             $C[i + 1] \leftarrow \text{temp}$ 
12.            // si on a fait un échange ce n'est pas le dernier passage
13.            // donc on met le drapeau à 0
14.             $\text{fini} \leftarrow 0$ 

```

Encore une fois on ne donne pas une preuve formelle de la correction de cet algorithme, mais croyez-nous sur parole : l'algorithme est correct.

Combien de pas on fait dans le pire des cas ? Ici, encore une fois, le pire des cas arrive quand les nombres, au départ, se trouvent dans l'ordre inverse. En ce cas, il faut faire remonter chaque nombre, en l'échangeant avec tous les nombres qu'on n'a pas encore fait remonter. Le nombre à la position i doit faire $n - i$ échanges. Chaque échange coûte 4 pas. Il faut ajouter deux pas (mise à jour de i et vérification), répétés n fois. La boucle du dernier nombre ne fait pas d'échanges et c'est donc la dernière. Donc

$$T(n) = 2 + (2 * n + 4 * (n - 1)) + (2 * n + 4 * (n - 2)) + \dots + (2 * (n - 1) + 4 * (n - (n - 1))) + 1$$

En réordonnant en peu

$$T(n) = 1 + 2 * n^2 + 4 * (n * (n - 1) / 2) = 4 * n^2 - 2 * n + 1$$

Il s'agit donc d'un algorithme quadratique, à l'instar du tri par insertion.

1.3 Des chiffres (mais pas des lettres)

Le problème qu'on présente ici a comme données un tableau de n nombres (positifs et négatifs) et un nombre "but" x . Le problème consiste de dire si, en additionnant un sous-ensemble non vide de nombres dans le tableau, on peut obtenir x . Un algorithme conceptuellement simple pour cela consiste à essayer toutes les possibilités. Pour chaque sous-ensemble, on additionne les nombres et on voit si le résultat est égal à x . Combien de pas faut-il faire ? Dans le pire des cas (le quel c'est ?) il faut essayer tous les sous-ensembles non vides. Combien y en a-t-il ? Combien de sous-ensembles, éventuellement vides, a un ensemble avec 1 élément, appelons-le $\{a\}$? 2 : l'ensemble vide, et $\{a\}$. Combien de sous-ensembles, éventuellement vides à un ensemble avec 2 éléments, disons $\{a, b\}$? 4 : l'ensemble vide, $\{a\}$, $\{b\}$, $\{a, b\}$. Pour l'ensemble

$\{a, b, c\}$, on a 8 sous-ensembles : le vide, $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{b, c\}$, $\{a, c\}$, $\{a, b, c\}$. En effet, on peut prouver, par *induction* [Exo] que un ensemble de n éléments a 2^n sous-ensembles, et donc $2^n - 1$ sous-ensembles non vides.

Imaginons maintenant qu'ajouter les nombres d'un sous-ensemble consiste en un pas (en pratique ce sera souvent plus). Donc on doit faire deux pas pour chaque sous-ensemble non vide (ajouter et comparer), et en total on devrait faire $2 * (2^n - 1) = 2^{n+1} - 2$ pas. Si par contre on suppose qu'ajouter les nombres d'un sous-ensemble soit $n - 1$ pas (en pratique ce sera souvent moins, car les sous-ensembles sont plus petits), on devrait faire $n * (2^n - 1)$ pas. Le vrai nombre de pas sera entre les deux. (C'est $(n/2) * (2^n)$) En tout cas le temps sera supérieur à 2^n et donc on parle d'un temps d'exécution *exponentiel*.

Pour visualiser ce que cela veut dire en pratique, imaginons qu'on puisse faire un pas chaque microseconde. Pour terminer l'algorithme sur un ensemble de 50 nombres il nous faudrait 1000 ans.

Il est donc un algorithme très peu efficace. Pourrait-on faire mieux ? En effet, le meilleur algorithme connu est beaucoup plus rapide, mais il lui faut toujours de centaines d'ans pour seulement 100 nombres. On ne sait pas si on peut trouver un algorithme encore plus efficace, mais les experts pensent qu'on ne puisse pas faire beaucoup mieux.

1.4 Tri par essais

Pour continuer avec des algorithmes simples mais inefficaces, on considère l'algorithme suivant pour trier un tableau : pour chaque *permutation* (façon de placer les nombres dans le tableau), vérifier si elle est dans le bon ordre. Pour faire cela on parcourt le tableau et on vérifie si chaque nombre est plus grand que le précédent. Si c'est le cas on termine. Il faudrait spécifier comment on va construire les différentes permutations. Mais cela n'est pas très important pour évaluer le temps d'exécution dans le pire des cas. En fait, dans le pire des cas, il faudra vérifier toutes les permutations.

Combien de permutations y a-t-il ? Pour un tableau de 1 case il y a une seule permutation. Pour un tableau de 2 cases il y a 2 permutations. Pour un tableau de 3 cases il y a 6 permutations. Il n'est pas difficile de montrer (par induction) que pour un tableau de n cases il y a $n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$ permutations. Ce nombre s'appelle le *factoriel* de n , est se dénote avec l'expression $n!$ (il est si grand qu'on a besoin de l'exclamer !!) Par exemple le factoriel de 10 est plus de 3 millions.

Dans le pire de cas, il faut donc faire au moins $n!$ pas (sans compter les pas nécessaires à créer une permutation ou à vérifier si la permutation est dans le bon ordre). On pourrait montrer que cela est encore moins efficace qu'un temps exponentiel. Avec cet algorithme, pour trier 20 nombres il nous faudrait quelques millions d'années, et pour 30 nombres il ne nous suffirait pas l'âge de l'univers.

1.5 Trouver le max

On considère maintenant le problème de trouver le maximum parmi les nombres dans un tableau. Le premier algorithme qu'on propose est le suivant :

1. $temp = 0$
2. **pour** $i \leftarrow 1$ **à** n **faire** :
3. **si** $C[i] > temp$ **faire** :
4. $temp \leftarrow C[i]$
5. **renvoie** $temp$

Pour chaque case i on vérifie si notre "candidat" maximum est plus grand (ou égal) que le nombre dans i . Si non, on remplace le candidat. À la fin, le candidat sera le plus grand nombre rencontré. Dans le pire des cas on aura fait $3 * n + 2$ pas. Il s'agit en ce cas d'un algorithme *linéaire*. Cela est très efficace. On peut trouver le maximum parmi un million de nombres en 3 secondes.

On note aussi le mot du pseudo-code **renvoie**. Cela est utilisé pour renvoyer explicitement le résultat. Cela n'est pas strictement nécessaire ici, mais il le sera dans le Chapitre 3, quand on verra les algorithmes récurifs.

On pourrait être intéressés pas par le maximum même, mais par la case du tableau où il se trouve. Pour cela on a donc l'algorithme suivant :

1. $temp = 1$
2. **pour** $i \leftarrow 2$ **à** n **faire** :
3. **si** $C[i] > C[temp]$ **faire** :
4. $temp \leftarrow i$
5. **renvoie** $temp$

A noter que si le maximum apparaît plusieurs fois dans le tableau, cet algorithme renvoie la position de la première *occurrence* du maximum.

1.6 Trouver un nombre

Finalement nous considérons un tableau trié de n nombres et nous nous demandons si un certain nombre x se trouve dans le tableau. Un simple algorithme linéaire consiste à parcourir tout le tableau. Si on trouve x , tant mieux, si non on sait qu'il n'est pas là. Mais on peut faire mieux que cela. En effet l'algorithme linéaire ci-dessus n'utilise pas le fait que le tableau est trié. On reviendra sur ce problème dans les prochains chapitres.

Chapitre 2

La Complexité d'un algorithme

On a discuté informellement de l'efficacité des algorithmes. Dans ce chapitre on donne les bases pour une analyse plus formelle.

2.1 Complexité

On a vu au chapitre précédent que on s'intéresse à évaluer le temps d'exécution d'un algorithme pour plusieurs instances d'un problème. En général on s'intéresse au temps d'exécution *en fonction* de la taille des données, ou paramètres, du problème. On voudra donc donner une expression $f(n)$ qui représente le temps d'exécution d'un algorithme. Par exemple, on a vu que le tri par insertion a un temps d'exécution de $2 + 9/2 * n + 3/2 * n^2$ pour un tableau de longueur n , dans le pire des cas.

On dit que le temps d'exécution est fonction de la taille des paramètres, mais on a aussi vu que des différents paramètres peuvent avoir la même taille. Jusqu'à présent on a considéré le temps d'exécution dans le pire des cas. Mais il y a d'autres possibilités : on pourrait aussi considérer le temps d'exécution en moyenne sur toutes les données de la même taille. Une autre possibilité plus technique est de considérer ce qu'on appelle le temps *amorti*. On verra cela à la fin du cours.

Pour rester dans le pire des cas, on pourrait peut-être calculer une formule exacte pour chaque algorithme. Pourtant cela est souvent complexe à obtenir avec précision. Une formule exacte pourrait aussi dépendre de certains détails secondaires, qui changent quand on change la description formelle d'un algorithme. De plus, pour avoir une formule exacte, il faudrait connaître le temps d'exécution de chaque opération, et non pas considérer tous les pas comme équivalents. Finalement, on observe aussi que le temps d'exécution devient crucial seulement pour des données relativement grandes. Il est donc intéressant d'évaluer le temps d'exécution *asymptotique*, c'est-à-dire, pour des données dont la taille tend vers l'infini.

Pour toutes ces motivations, on ne considérera jamais les fonctions exactes qui représentent le temps d'exécution, mais on tiendra compte seulement de leur *ordre de grandeur* ou *ordre de croissance*. On a déjà vu cela informellement, en parlant des fonctions quadratiques, linéaires, exponentielles. On va maintenant préciser ce concept.

2.2 L'ordre de grandeur des fonctions

Le calcul de la complexité d'un algorithme dépend de plusieurs détails. Pour un calcul exact, il faudrait les préciser avec attention. Bien que ces détails jouent un rôle important en pratique, on préfère, dans un premier moment, s'abstraire d'eux. En conséquence on ne peut pas s'intéresser au temps d'exécution exact, mais à son *ordre de croissance*. Pour utiliser une autre expression qu'on va bien tôt définir, on s'intéresse à la complexité *asymptotique* des algorithmes.

On commence par un exemple. On veut comparer la fonction $10000 * n^2$ avec la fonction 2^n .

Quelle est la plus "grande" ? Regardons les premières valeurs :

	$10000 * n^2$	2^n
$n = 1$	10000	2
$n = 2$	40000	4
$n = 3$	90000	8
$n = 4$	160000	16
$n = 5$	250000	32

Il semblerait que $10000 * n^2$ est beaucoup plus grande. Cependant, si on regarde ce qui se passe pour de grandes valeurs de n , on remarquera que à partir de $n = 23$, la valeur de 2^n dépasse celle de $10000 * n^2$. Cela montre que, *asymptotiquement*, c'est à dire, à partir d'une certaine valeur *suffisamment grande* de n , une fonction devient plus grande que l'autre. En fait, c'est n'est pas seulement que une fonction est plus grande que l'autre, mais bien plus que cela : avec quelques outils mathématiques, on pourrait montrer que le *rapport* entre 2^n et $k * n^h$ devient de plus en plus grand : pour tout nombre c , pour n suffisamment grand on a $2^n / k * n^h > c$. Ce rapport dépasse toute borne.

On utilisera ce critère pour comparer deux fonctions $f(n)$ et $g(n)$ ¹. On dira que $f(n)$ est *au plus* de l'ordre de $g(n)$ si le rapport $f(n)/g(n)$ est *borné*, c'est-à-dire s'il existe un nombre $c > 0$ tel que pour tout n on a $f(n)/g(n) < c$. On dénotera cela par $f(n) \in O(g(n))$ ("Grand O") On dit que les deux fonctions ont *le même ordre de grandeur*, ou qu'elles sont *du même ordre* si leur rapport est borné dans les deux sens : s'ils existent deux constantes strictement positives c_1, c_2 telles que pour tout n $f(n)/g(n) > c_1$ (le rapport ne devient pas trop petit) et $f(n)/g(n) < c_2$ (le rapport ne devient pas trop grand). On écrira $f(n) \in \Theta(g(n))$ (ou $g(n) \in \Theta(f(n))$) pour dire que elles sont du même ordre.

Exercice : prouver que la relation "être du même ordre que" est une relation d'équivalence.

Exercice : prouver que $f(n)$ et $g(n)$ sont *du même ordre* si $f(n) \in O(g(n))$ et $g(n) \in O(f(n))$.

Souvent on écrira aussi $\Theta(f(n))$ pour dénoter une fonction non-spécifiée, qui est du même ordre que $f(n)$.

2.3 Exemples

On considère $f(n) = 2 * n^2 + 3 * n + 5$ et $g(n) = n^2$. On observe que $f(n)/g(n) = 2 + 3/n + 5/n^2 < 10$ pour tout n , tandis que $g(n)/f(n) = 1/(2 + 3/n + 5/n^2) < 1$ pour tout n , donc $f(n) \in \Theta(g(n))$. On considère $f(n) = n^2$ et $g(n) = n^4$. On observe que $f(n)/g(n) = 1/n^2 < 1$ pour tout n , donc $f(n) \in O(g(n))$ tandis que $g(n)/f(n) = n^2$, qui n'est pas borné. Donc $f(n) \notin O(g(n))$.

On donne ici une liste de termes relatifs à l'ordre de grandeur :

- Les fonctions du même ordre de grandeur de n s'appellent *linéaires*. Exemples : $4n - 3$, $2n + 1$.
- Les fonctions du même ordre de grandeur de n^2 s'appellent *quadratiques*. Exemples : $3n^2 + 2n - 4$, $1000n^2 - 2$.

1. on suppose ici que $f(n)$ et $g(n)$ sont strictement positives - cela n'est pas très restrictif est simplifie la présentation.

- Les fonctions qui ont le même ordre de grandeur de n^k , pour quelque $k \geq 0$, s'appellent *polynomiales*.
- Les fonctions du même ordre de grandeur de $2^{f(n)}$, pour quelque fonction polynomiale f , s'appellent *exponentielles*.

Toutes les fonctions exponentielles ne sont pas du même ordre : par exemple 2^n n'est pas dans $\Theta(2^{n^2})$. Exercice : de quel ordre sont $10^n, n * 2^n$?

Un ordre de grandeur que aura une certaine importance dans le prochaine chapitre est celui de $n \log_2 n$.

Pourtant, il faut un peu se méfier de la complexité asymptotique. Parfois des algorithmes dont la complexité au pire des cas est assez grande (exponentielle) sont utilisé avec succès en pratique, car, par exemple, le pire des cas ne se présente pas souvent.

2.4 Les pas

On veut compter le nombre des pas, mais on a observé que la notion de pas n'est pas définie à priori. De plus, vu qu'on s'intéresse premièrement à l'ordre de grandeur de la complexité d'un algorithme, une définition exacte de pas n'est pas non plus nécessaire. Il suffira que le temps d'exécution de tout pas soit borné par une constante. Donc, on peut considérer comme pas, toute opération dont le temps d'exécution ne dépend pas de la taille des données. Ici on conviendra que chaque ligne du pseudocode correspond à un pas. Cela est différent de la façon de compter les pas du premier chapitre, mais il ne change en rien le calcul de la complexité asymptotique des algorithmes.

Chapitre 3

Récursion

3.1 Nommer un algorithme

Donner un nom à un algorithme est essentiel si on veut pouvoir l'utiliser à l'intérieur d'un autre algorithme. Si par exemple on veut décrire un algorithme qui utilise le tri par bulles, on pourrait réécrire le tri par bulles, ou bien juste utiliser son nom.

TRIBULLE(C) :

1. $n \leftarrow \text{longueur}(C)$
2. $\text{fini} \leftarrow 0$
3. **tant que** $\text{fini} = 0$ **faire** :
4. $\text{fini} \leftarrow 1$
5. **pour** $i \leftarrow 1$ **à** n **faire** :
6. **si** $C[i] > C[i + 1]$ **faire** :
7. $\text{temp} \leftarrow C[i]$
8. $C[i] \leftarrow C[i + 1]$
9. $C[i + 1] \leftarrow \text{temp}$
10. $\text{fini} \leftarrow 0$

Dans le nom de l'algorithme on indique aussi quelles sont les données d'entrée, ou paramètres, de l'algorithme. Dans ce cas est le tableau à trier. Ici C est le *paramètre formel*. Pour symboliser l'exécution de l'algorithme, on écrit son nom, et on remplace les paramètres formels avec la donnée sur la quelle on veut exécuter l'algorithme. Si on veut lancer le tri par bulle sur un tableau spécifique D on écrit TRIBULLE(D). Le nombre de pas exécutés par cet appel correspond au nombre de pas de l'algorithme correspondant.

On pourrait aussi être plus flexible et spécifier quelles parties du tableau on veut trier. On peut vouloir trier le tableau entre la case j et la case k .

TRIBULLE(C, j, k) :

1. $\text{fini} \leftarrow 0$
2. **tant que** $\text{fini} = 0$ **faire** :
3. $\text{fini} \leftarrow 1$
4. **pour** $i \leftarrow j$ **à** k **faire** :
5. **si** $C[i] > C[i + 1]$ **faire** :
6. $\text{temp} \leftarrow C[i]$
7. $C[i] \leftarrow C[i + 1]$
8. $C[i + 1] \leftarrow \text{temp}$
9. $\text{fini} \leftarrow 0$

Attention : cet algorithme est correct sous la condition que $k \leq \text{longueur}(C)$, car sinon $C[i]$ n'a pas de sens pour $i > n$. Pour trier entièrement le tableau D , il faudrait exécuter TRIBULLE($D, 1, \text{longueur}(D)$).

Parfois un algorithme est censé renvoyer un résultat, sous forme d'un nombre. Dans ce cas on peut affecter ce nombre à une variable, ou le renvoyer ultérieurement. Considérons l'algorithme pour trouver la maximum d'un tableau. On ajoute le même niveau de flexibilité que pour le tri par bulles, c'est à dire on met les limites parmi les paramètres.

MAX(C, j, k) :

1. $temp = 0$
2. **pour** $i \leftarrow j$ **à** k **faire** :
3. **si** $C[i] > temp$ **faire** :
4. $temp \leftarrow C[i]$
5. **renvoie** $temp$

Maintenant on peut affecter la valeur renvoyée par cet algorithme :

$$m \leftarrow \text{MAX}(D, 1, \text{longueur}(D))$$

On peut utiliser les noms même pour des algorithmes très simples, pour faciliter la description d'algorithmes plus complexes. Par exemple, on peut décrire un simple algorithme pour calculer le maximum entre deux nombres. Cela nous sera utile par la suite.

MAX2(n, m)

1. **si** $n > m$ **renvoie** n
2. **sinon renvoie** m

On note que ici ce n'est pas question de temps d'exécution : pour tous paramètres le temps d'exécution est le même.

3.2 Récursion

Dans les exemples ci-dessus, donner un nom à un algorithme est quelque chose de pratique, mais pas strictement nécessaire. On pourrait toujours remplacer l'appel à un algorithme par sa description.

Pourtant les noms nous permettent aussi de faire appel à un algorithme *dans sa propre définition*. Un algorithme qui fait appel à soi-même est dit *récuratif*.

Voyons un exemple :

SANSFIN(n)

1. $m \leftarrow \text{SANSFIN}(n + 1)$
2. **renvoie** m

Ici l'algorithme SANSFIN(0), pour renvoyer son résultat, appelle l'algorithme SANSFIN(1) qui, pour renvoyer son résultat, appelle l'algorithme SANSFIN(2) qui, pour renvoyer son résultat, appelle l'algorithme SANSFIN(3) qui, pour renvoyer son résultat, appelle l'algorithme SANSFIN(4) qui, ... C'est clair que l'algorithme ne se termine pas.

Quel est donc l'intérêt d'un algorithme récursif s'il ne termine pas ? Le fait que un algorithme *peut* s'appeler ne veut pas dire qu'il *doit* s'appeler. La terminaison d'un algorithme récursif dépend des *cas de base* : les cas où un algorithme ne fait pas appel à soi-même mais termine directement.

Pour mieux comprendre, on considère un algorithme récursif pour chercher le maximum dans un tableau entre les positions i et j . Cet algorithme compare l'élément à la position i avec le maximum des éléments suivants, et renvoie le plus grand des deux. Mais s'il n'y a plus d'éléments suivants, l'algorithme s'arrête, en renvoyant l'élément même.

MAXREC(C, i, j)

1. **si** $i \geq j$
2. **renvoie** $C[j]$
3. **sinon**
4. $maxtmp \leftarrow \text{MAXREC}(C, i + 1, j)$
5. **renvoie** MAX2($C[i], maxtmp$)

Pour trouver le max du tableau, on doit exécuter MAXREC($C, 1, longueur(C)$).

Quel est le temps d'exécution de cet algorithme ? On suppose que le temps soit une fonction $T(n)$, où n est la taille de la partie du tableau dans la quelle on cherche, c'est-à-dire $n = longueur(C) - i + 1$. Pour exécuter MAXREC, on exécute un nombre fixé de pas disons k_1 , puis on exécute MAXREC sur une partie plus petite du tableau, puis on exécute MAX2, qui consiste en un nombre fixé de pas, disons k_2 . Sauf dans le cas où $n = 1$, en ce cas on exécute un nombre fixé de pas, disons h . On a donc :

$$T(n) = \begin{cases} k_1 + k_2 + T(n - 1) & \text{si } n > 1 \\ h & \text{sinon} \end{cases}$$

Quelle est la solution de cette équation ? On verra plus tard une méthode générale pour résoudre ce genre d'équations. Ici on observe que on pourrait montrer par *induction* que $T(n) = h + (n - 1)(k_1 + k_2)$. Il s'agit donc d'un temps linéaire : $T(n) \in \Theta(n)$.

3.3 Diviser pour Régner : Tri par fusion

Dans l'exemple précédent du calcul du Max, le temps d'exécution de l'algorithme récursif n'est pas asymptotiquement meilleur de l'algorithme itératif. Cependant, l'utilisation de la technique de la récursion permet souvent d'obtenir des algorithmes plus efficaces.

On va maintenant présenter une forme générale partagée par plusieurs algorithmes récursifs : *Diviser pour Régner*. Un algorithme de ce type consiste en trois phases :

1. (Diviser) On divise le problème en plusieurs sous-problèmes de plus petite taille
2. (Régner) On résout récursivement ces problèmes. Quand la taille du sous-problème est suffisamment petite (en général 1 ou 2), on donne la réponse directement.
3. (Recombinaison) On combine les solutions des sous-problèmes, pour obtenir une solution du problème principal.

Le premier exemple de cette technique qu'on présente ici est un algorithme de tri qui s'appelle Tri par Fusion ("Mergesort" en Anglais). Le schéma général s'applique de la façon suivante :

1. (Diviser) On divise le tableau en deux tableaux de la même taille (ou presque si le tableau original a une taille impaire)
2. (Régner) On trie récursivement ces tableaux. Quand la taille du tableau est 1, il n'y a rien à faire et donc on donne la réponse directement.
3. (Recombinaison) On fusionne les deux tableaux triés, de façon que le tableau qu'on obtient est aussi trié.

La seule chose qui reste à détailler est cette fusion. On va décrire donc d'abord cet "sous-algorithme". Il y a plusieurs façons de formuler le problème. On pourrait avoir en entrée les deux tableaux à fusionner et demander de renvoyer le tableau fusionné en sortie. Pour l'application à notre algorithme de tri, cela nous convient de utiliser la représentation suivante : un seul tableau, C , et trois positions dans ce tableau : i, k, j , avec l'hypothèse que $1 \leq i \leq k < j \leq \text{longueur}(C)$. Les deux tableaux à fusionner sont la partie du tableau C comprise entre i et k , dénoté par $C[i..k]$, et la partie du tableau C comprise entre $k + 1$ et j , dénoté par $C[k + 1..j]$. On suppose que $C[i..k]$ et $C[k + 1..j]$ sont triés. La fusion consiste à réorganiser le contenu de ces deux tableaux de façon que le tableau $C[i..j]$ soit trié. L'algorithme utilise un tableau auxiliaire D . À chaque pas on regarde les premiers nombres de chaque tableau, et on insère le plus petit des deux dans D .

FUSION(C, i, k, j)

```

1.  $h1 \leftarrow i$ 
2.  $h2 \leftarrow k + 1$ 
3.  $l \leftarrow 1$ 
4. tant que  $h1 \leq k$  et  $h2 \leq j$  faire :
5.     si  $C[h1] < C[h2]$ 
6.     // si le plus petit est dans le premier tableau
7.          $D[l] \leftarrow C[h1]$ 
8.          $h1 \leftarrow h1 + 1$ 
9.     sinon
10.    // si le plus petit est dans le deuxième tableau,
11.         $D[l] \leftarrow C[h2]$ 
12.         $h2 \leftarrow h2 + 1$ 
13.     $l \leftarrow l + 1$ 
14.    // un des deux tableaux est terminé, il faut copier celui qui reste
15.    si  $h1 > k$ 
16.        tant que  $h2 \leq j$  faire :
17.             $D[l] \leftarrow C[h2]$ 
18.             $h2 \leftarrow h2 + 1$ 
19.             $l \leftarrow l + 1$ 
20.    sinon
21.        tant que  $h1 \leq k$  faire :
22.             $D[l] \leftarrow C[h1]$ 
23.             $h1 \leftarrow h1 + 1$ 
24.             $l \leftarrow l + 1$ 
25.    // finalement on recopie tout dans  $C$ 
26.    pour  $l \leftarrow 1$  à  $j - i + 1$  faire :
27.         $C[l + i - 1] \leftarrow D[l]$ 
```

Quelle est la complexité de cette fusion ? On a des affectations, puis une boucle. Combien de tours fait cette boucle ? Jusqu'à ce qu'on termine un des deux tableaux. Disons que ce nombre est n_1 . Après on exécute une deuxième boucle pour terminer de remplir d . On fait donc un certain nombre de tours, n_2 . On peut pas a priori connaître ces deux nombres, mais on sait que $n_1 + n_2 = k - i + 1$ c'est-à-dire la somme des longueurs des deux tableaux à fusionner. Finalement on recopie le contenu de D dans C , et encore une fois on fait $k - i + 1$ tours. On a pas besoin de compter exactement le nombre de pas dans chaque boucle : il nous suffit de dire que la complexité de l'algorithme de fusion est $\Theta(n)$, où n est la longueur totale de deux tableaux à fusionner. La Fusion est donc linéaire.

On peut maintenant écrire l'algorithme de tri par fusion comme suit. Ici les paramètres sont : le tableau c et deux positions dans ce tableau : i, j , avec l'hypothèse que $1 \leq i \leq j \leq \text{longueur}(C)$.

TRIFUSION(C, i, j)

1. **si** $i < j$
2. // diviser
3. $k \leftarrow i + ((j - i) \text{ div } 2)$
4. // régner
5. TRIFUSION(C, i, k)
6. TRIFUSION($C, k + 1, j$)
7. // recombinaison
8. FUSION(C, i, k, j)

Pour trier un tableau C il suffit d'appeler donc TRIFUSION($C, 1, \text{longueur}(C)$).

3.4 Complexité du Tri par fusion

On a vu que la complexité de la fusion est linéaire, mais quelle est la complexité de l'algorithme entier ? Ce n'est pas évident de compter combien de pas on fait. Cherchons de voir cela sous un autre point de vue. Le nombre de pas pour le tri sur un tableau de taille n est égal à deux fois le nombre de pas pour le tri d'un tableau de taille $n/2$ plus le nombre de pas pour la fusion. Sauf le cas où $n = 1$, dans le quel cas, on fait qu'un pas (vérifier la condition). Disons que le temps d'exécution pour un tableau de taille n est une certaine fonction $T(n)$. Donc $T(n)$ satisfait l'équation suivante :

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

On rappelle que ici on écrit $\Theta(n)$ pour indiquer une fonction $f(n) \in \Theta(n)$, qu'on ne connaît pas exactement.

Pour calculer $T(n)$ il nous suffit de résoudre cette équation. Bien sur la solution ne peut pas être exacte, car dans l'équation on n'a pas spécifié la complexité exacte de la fusion. La solution sera seulement un ordre de grandeur. Mais le quel ? Y a-t-il une solution unique ? Est-ce $T(n) = n^2$? Si $n > 1$ devrait être : $n^2 = 2(n/2)^2 + \Theta(n)$. Donc $n^2 = n^2/2 + \Theta(n)$. Ce qui impliquerait $\Theta(n) = n^2/2$, qui est faux. Est-ce $T(n) = n$? Si $n > 1$ devrait être : $n = 2(n/2) + \Theta(n)$. Donc $n = n + \Theta(n)$. Ce qui impliquerait $\Theta(n) = 0$, qui est faux.

Il faut trouver une fonction entre ces deux. Pour cela on utilisera un théorème général sur les équations récursives.

On observe que le temps d'exécution du tri par fusion est toujours le même pour tout tableau de taille n . Il n'y a pas de sens ici de parler de pire des cas.

En passant, il faut aussi observer que l'équation récursive correcte serait

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{si } n > 1 \end{cases}$$

car, si le tableau est de longueur impair, on ne peut pas le diviser exactement en deux. Toutefois, cette précision n'est pas nécessaire, car ne changerait pas la valeur asymptotique de la solution. Pour plus de détails, voir [CLRS02].

3.5 Le Théorème du coût récursif

Considerons d'abord un cas idéal, où la phase de recombinaison ne prend pas de temps. On aurait donc

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ aT(n/b) & \text{si } n > 1 \end{cases}$$

L'idée est que on divise le problème en a sous-problèmes, chacun de taille n/b . On observe que

$$T(n) = aT(n/b) = a^2T(n/b^2) = a^3T(n/b^3) = \dots = a^kT(n/b^k)$$

(comme d'habitude on ignore les approximations dues à $\lceil - \rceil$ et $\lfloor - \rfloor$). Pour quelle valeur de k faut-il s'arrêter ? On s'arrête quand $b^k > n$ c'est à dire quand $k = \log_b n$. On a donc que $T(n) = a^{\log_b n} T(1) = a^{\log_b n}$. Pour les propriétés des logarithmes on a : $a^{\log_b n} = n^{\log_b a}$. On conclut que $T(n) \in \Theta(n^{\log_b a})$.

Pour le cas général, supposons d'avoir une équation récursive :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq h \\ aT(n/b) + f(n) & \text{si } n > h \end{cases} \quad (3.1)$$

L'idée est que, pour des données de petite taille (plus petite que une constante h) le temps d'exécution est une constante. Pour de données des grande taille, on divise le problème en a sous problèmes, chacun de taille n/b (plus précisément $\lceil n/b \rceil$, ou $\lfloor n/b \rfloor$). Finalement le temps employé pour recombinaison des solutions des sous-problèmes est une fonction $f(n)$.

Théorème 1 (Théorème du coût récursif) *Sous l'hypothèse que $a \geq 1, b > 1$ et $h \geq 1$, soit $T(n)$ une fonction qui satisfait l'équation 3.1. On considère trois cas :*

1. si $f(n) \in O(n^{\log_b a - \epsilon})$ pour un $\epsilon > 0$, alors $T(n) \in \Theta(n^{\log_b a})$
2. si $f(n) \in \Theta(n^{\log_b a})$ alors $T(n) \in \Theta(n^{\log_b a} \log_2 n)$
3. si $n^{\log_b a + \epsilon} \in O(f(n))$ pour un $\epsilon > 0$, et si $af(n/b) < cf(n)$ pour un $c < 1$ et n suffisamment grand, alors $T(n) \in \Theta(f(n))$

On voit que l'énoncé exact est assez complexe. Il faut aussi observer que les trois cas ne sont pas les seuls possibles, mais ils sont ceux pour les quels on peut résoudre l'équation. Moralement on a que si la phase de recombinaison est plus rapide que la résolution des sous-problèmes, alors la recombinaison ne compte pas. Si le temps pour recombinaison est du même ordre que la résolution des sous-problèmes, alors on ajoute un facteur logarithmique. Si la recombinaison est plus lente que la résolution des sous-problèmes, alors est le temps de recombinaison qui l'emporte.

Finalement on applique ce théorème à l'algorithme de tri par fusion. L'équation récursive était :

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

On observe que $\log_2 2 = 1$, et que donc on est dans le deuxième cas : $f(n) \in \Theta(n)$. Grâce au théorème du coût récursif, on conclut que $T(n) \in \Theta(n \log_2 n)$. L'algorithme de tri fusion est plus efficace que les autres algorithmes vu précédemment.

3.6 Tri rapide

Un autre algorithme récursif, proposé par Tony Hoare, est le *Tri rapide* (Quicksort en Anglais), qu'on verra en TD.

Chapitre 4

Listes, Files et Piles

<i>val</i>	<i>cle</i>	<i>nom</i>
1	42	eric

FIGURE 4.1: Un objet

Pour introduire les structures de données, imaginons que nos données soient des livres d'une grande bibliothèque. Quelles sont les opérations qu'on veut faire avec cette bibliothèque ? Peut être on veut savoir si un livre se trouve dans la bibliothèque. Ou bien on veut emprunter un livre. Ou encore on veut ajouter un nouveau livre à la collection, ou rendre un livre qu'on a emprunté. On peut se demander quel est le livre le plus ancien ou le plus récent.

Chacune de ces opérations peut être plus ou moins efficace à exécuter. Si les livres se trouvent empilés en vrac dans un entrepôt, c'est très difficile de trouver un livre en particulier, mais c'est très facile d'ajouter un livre, il suffit de le jeter dans l'entrepôt. Si les livres sont bien rangés dans des casiers en ordre alphabétique, c'est relativement facile de trouver un livre, mais on pourrait avoir de soucis à insérer un nouveau livre, car l'endroit où il est censé aller n'est pas libre.

Dans ce chapitre on commencera à voir de différentes façon d'organiser les données, et on discutera les opérations qu'on peut faire avec chacune d'elles.

Avant de commencer, on aura aussi besoin d'une nouvelle notion, la notion de *pointeur*.

4.1 Les objets, et les pointeurs

Les objets Un objet est une simple généralisation d'un tableau. Un tableau contient plusieurs cases, numérotées de 1 à n . Un objet a aussi plusieurs cases, sauf qu'au lieu d'avoir un numéro, chaque case a une *étiquette*. Par exemple ici on a un objet O :

Les cases d'un objet s'appellent *champs*. La notation en pseudocode est aussi différente que la notation pour les tableau : pour indiquer le champ *cle* de l'objet O , on n'écrira pas $O[cle]$ comme on faisait pour les tableaux, mais on écrira $O.cle$.

(On observe, en passant, que dans des langages comme C, les objets permettent de rassembler des données de types différents, comme chaînes de caractères et nombres, tandis que en général dans un tableau on peut mettre seulement des données de même type.)

On considère que les tableaux sont un cas particulier d'objets. Dans la suite, quand on parlera d'objets, on inclura donc aussi les tableaux. On indiquera les objets avec des lettres majuscules.

Les pointeurs Jusqu'à maintenant, dans une variable ou bien dans un champ d'un objet on a stocké seulement des nombres. On voudrait pouvoir aussi stocker des objets dans le champ d'un autre objet. Cela est difficile à représenter dans un dessin. Cette difficulté de représentation correspond à une situation réelle dans les langages de programmation. Pour garder l'image des casiers, on peut bien stocker un livre dans une case, mais on ne peut pas stocker un casier entier. Ce qu'on peut faire, toutefois, c'est de stocker un catalogue, qui nous indique où on peut trouver le casier en question.

Pour sortir de la métaphore, on stockera donc dans un champ ou dans une variable non pas un objet, mais un *pointeur* vers un objet, représenté par une jolie petite flèche :

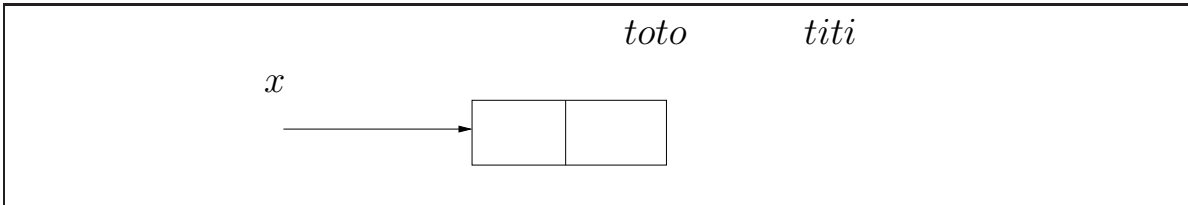


FIGURE 4.2: Un pointeur vers un objet

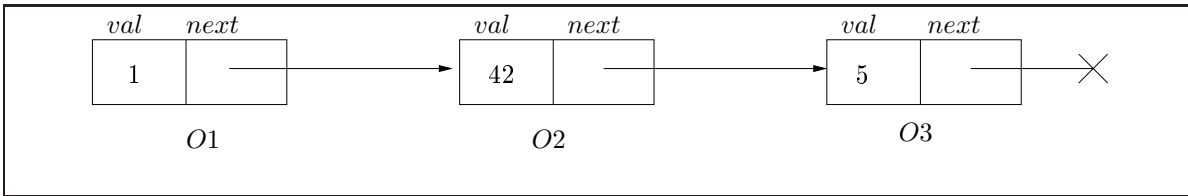


FIGURE 4.3: Trois objets

Pour l'utilisation des pointeurs, on cherchera à rester proche des langages impératifs tels que C ou Java. En C, la manipulation explicite des pointeurs est faite à l'aide des symboles `*` et `&`. Dans notre pseudocode, on fera la convention suivante (qui est celle de Java) : un objet est toujours représenté par un pointeur vers l'objet. Le pseudocode tiendra compte de cette convention. Par exemple si on a que *O1* et *O2* sont deux objets, on peut écrire :

$$O1.next \leftarrow O2$$

cela signifie que le champ *next* de l'objet *O1* est affecté avec un pointeur vers l'objet *O2*. Aussi, si *x* est un pointeur et *Obj* un objet on écrira $x = Obj$ pour indiquer que *x* est un pointeur vers *Obj* (en C on écrirait `*x==Obj` ou `x==&Obj`).

On écrira aussi :

$$O1.next.titi \leftarrow 5$$

cela signifie que le champ *titi* de l'objet pointé par le champ *next* de l'objet *O1* est affecté avec la valeur 5.

On peut aussi enchaîner les pointeurs, ci-dessus *O1.next.next.val* contient la valeur 5. Un champ qui est censé contenir un pointeur, mais qui pointe nulle part (comme dans l'objet *O3* ci-dessous), est dit pointer vers **null**.

Pointeurs comme paramètres On a vu dans les chapitres précédents que quand on applique un algorithme à un tableau (et plus généralement à un objet) on n'a pas besoin de renvoyer le résultat : c'est l'objet même qui est modifié. Par exemple c'est le tableau qui a changé, qui a été trié. On ne construit pas un nouveau tableau trié qu'on renvoie comme résultat.

En pratique, quand on appelle un algorithme avec des paramètres, on crée des copies de ces paramètres, et on travaille avec ces copies. Mais quand on passe en paramètre un objet, la convention sera qu'on ne crée pas une copie de l'objet même, mais seulement une copie du pointeur vers l'objet.

Si donc on appelle un algorithme avec un objet *Obj* en paramètre, et si on affecte des valeurs à des champs de *Obj*, c'est l'objet même qui change. Par contre si on passe en paramètre un nombre ou un pointeur, on fait bien des copies, et le nombre ou le pointeur originaux ne sont pas modifiés.

4.2 Listes chaînées

Un élément d'une liste chaînée est un objet avec un champ *val*, qui contient une valeur, et un champ *next* qui contient un pointeur vers un autre élément, ou bien vers **null**. Une *liste chaînée* est un pointeur vers un élément. Une description *récursive* d'une liste chaînée est la suivante : une liste chaînée est soit un pointeur vers **null**, soit un pointeur vers un objet qui a un champ *val*, qui contient une valeur, et une champ *next* qui contient une liste.

Une liste peut aussi être *circulaire* : cela arrive quand le champ *next* du dernier objet pointe vers le premier objet de la liste.

Imaginons qu'on veuille représenter notre bibliothèque avec une liste. Les champs *val* seront les numéros ISBN des livres. Chaque objet pourrait aussi contenir d'autres informations sur le livre.

Pour chercher un livre dans notre bibliothèque, l'algorithme est le suivant. On parcourt la liste tant qu'on n'a pas trouvé le numéro du livre. Si on le trouve, on renvoie un pointeur vers l'objet qui contient le livre, si on ne le trouve pas, on renvoie un pointeur vers **null**.

Dans l'algorithme suivant, on cherche dans la liste *l* le livre numéro *k* :

RECHERCHELISTE(*l*, *k*)

1. **tant que** *l* \neq **null** et *l.val* \neq *k* **faire** :
2. *l* \leftarrow *l.next*
3. **renvoie** *l*

On peut aussi écrire l'algorithme de façon récursive :

RECHERCHELISTEREC(*l*, *k*)

1. **si** *l* = **null** ou *l.val* = *k* **renvoie** *l*
2. **sinon renvoie** RECHERCHELISTEREC(*l.next*, *k*)

On remarque les deux cas possibles : une liste est soit un pointeur vers **null**, soit un pointeur vers un objet qui a un champ *val* et un champ *next* qui contient une liste.

La taille d'une liste est le nombre d'objets qu'elle contient. Quand le livre qu'on recherche n'est pas dans la liste, l'algorithme doit parcourir toute la liste avant de renvoyer **null**. Donc, rechercher un élément d'une liste chaînée coûte $\Theta(n)$.

Ajouter un livre à la liste est très facile, il suffit de le mettre en tête. On imagine vouloir insérer un objet *Obj* dont le champ *val* a déjà été rempli avec le numéro du livre.

AJOUTELISTE(*l*, *Obj*)

1. *Obj.next* \leftarrow *l*
2. *l* \leftarrow *Obj*
3. **renvoie** *l*

Cela est évidemment en temps constant, c'est-à-dire indépendant de la taille de la liste *l*.

Pour effacer un livre de la liste, il faut chercher ce livre, et s'arrêter juste avant l'objet correspondant, car il faut modifier le champ *next* de son prédécesseur.

ENLEVELISTE(l, k)

1. **si** $l = \text{null}$
2. **renvoie** l
3. **sinon si** $l.val = k$
4. **renvoie** $l.next$
5. **sinon**
6. **tant que** $l.next \neq \text{null}$ **et** $l.next.val \neq k$ **faire** :
7. $l \leftarrow l.next$
8. **si** $l.next \neq \text{null}$
9. $l.next \leftarrow l.next.next$
10. **renvoie** l

On peut aussi l'écrire de façon récursive :

ENLEVELISTEREC(l, k)

1. **si** $l = \text{null}$
2. **renvoie** l
3. **sinon si** $l.val = k$
4. **renvoie** $l.next$
5. **sinon**
6. $l.next \leftarrow \text{ENLEVELISTEREC}(l.next, k)$
7. **renvoie** l

4.3 Objet Liste

Les algorithmes ci-dessus doivent renvoyer le résultat, car quand on passe en paramètre un pointeur, on ne modifie pas le pointeur original. Pour pouvoir écrire des algorithmes sur les listes qui se comportent comme les algorithmes de tableaux, c'est-à-dire qui modifient la liste passée en paramètre, il faut utiliser une autre notion de liste.

Un *objet liste chaînée* L est un objet avec un champ *tete* qui contient un pointeur vers un élément. Autrement dit, un objet liste chaînée est un objet avec un champ *tete* qui contient une liste chaînée. Un objet liste peut éventuellement contenir d'autres champs. L'objet pointé par $L.tete$ est appelé la *tête* de la liste, tandis que le dernier objet, celui dont le champ *next* contient **null**, est la *queue*. Si $L.tete$ est lui même **null**, on dit que l'objet liste est *vide*.

Bien que subtile, la distinction entre liste et objet liste est très importante, et il faut bien la retenir. En particulier, une liste est un pointeur, donc si on appelle un algorithme sur une liste x , l'algorithme travaille sur une copie du pointeur x . Tandis que si on appelle un algorithme sur un objet liste L , on travaille sur une copie du pointeur vers L , mais on ne fait pas une copie de l'objet.

Certains algorithmes ne changent pas : quand on cherche un élément il faut le renvoyer :

RECHERCHEOBJETLISTE(L, k)

1. **renvoie** RECHERCHELISTE($L.tete, k$)

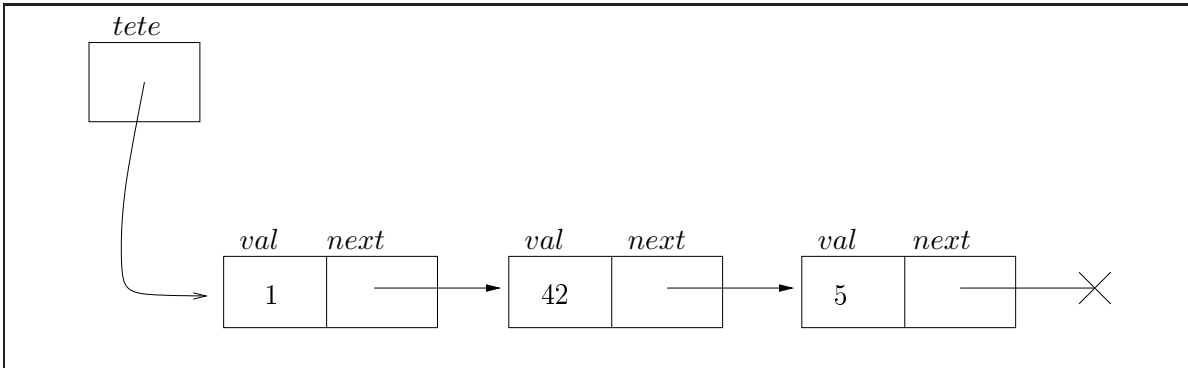


FIGURE 4.4: Un objet liste

Par contre on peut ajouter un livre à un objet liste, en modifiant l'objet liste, sans renvoyer le résultat :

AJOUTEOBJETLISTE(L, Obj)

1. $Obj.next \leftarrow L.tete$
2. $L.tete \leftarrow Obj$

De façon similaire on efface un livre d'un objet liste :

ENLEVEOBJETLISTE(L, k)

1. **si** $L.tete.val = k$
2. $L.tete \leftarrow L.tete.next$
3. **sinon**
4. $l \leftarrow L.tete$
5. **tant que** $l.next \neq \text{null}$ **et** $l.next.val \neq k$ **faire** :
6. $l \leftarrow l.next$
7. **si** $l.next \neq \text{null}$
8. $l.next \leftarrow l.next.next$

On peut aussi le récrire de la façon suivante :

ENLEVEOBJETLISTE2(L, k)

1. $l \leftarrow L.tete$
2. **si** $l.val = k$
3. $L.tete \leftarrow l.next$
4. **sinon**
5. ENLEVELISTE($l.next, k$)

Ou encore :

ENLEVEOBJETLISTE3(L, k)

1. $L.tete \leftarrow \text{ENLEVELISTE}(L.tete, k)$

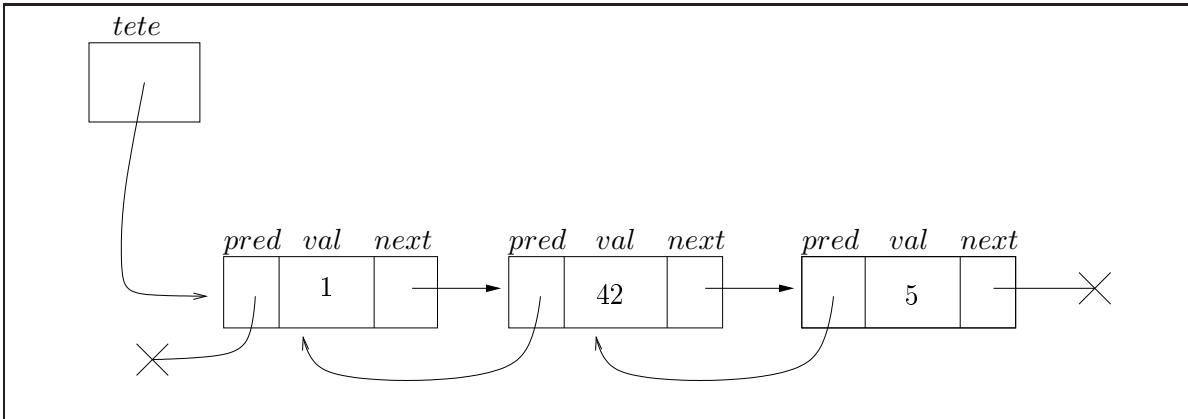


FIGURE 4.5: Un objet liste doublement chaînée

4.4 Liste doublement chaînée

On a vu que pour effacer un objet d'une liste, il faut connaître son prédécesseur. L'algorithme est un peu compliqué de ce fait. Ce serait plus facile si chaque élément d'une liste connaissait son prédécesseur. Cela nous amène à la définition suivante : Un élément d'une liste doublement chaînée est un objet avec un champ *val*, qui contient une valeur, et deux champs *next* et *pred* qui contiennent des pointeurs vers d'autres éléments, ou bien vers **null**. Pour tout pointeur x vers un élément d'une liste doublement chaînée il faut également que si $x.next$ n'est pas **null**, $x.next.pred = x$ et si $x.pred$ n'est pas **null**, $x.pred.next = x$. Une *liste doublement chaînée* est un pointeur vers un élément. (Une description *réursive* d'une liste doublement chaînée est aussi possible, mais nous n'allons pas la présenter.) Un *objet liste doublement chaînée* L est un objet avec un champ *tete* qui contient une liste doublement chaînée. Le champ $L.tete.pred$ doit être **null**.

L'algorithme pour chercher un livre ne change pas. On parcourt la liste tant qu'on n'a pas trouvé le numéro du livre. Si on le trouve, on renvoie un pointeur vers l'objet qui contient le livre, si on ne le trouve pas, on renvoie un pointeur vers **null**.

Pour ajouter un livre à la liste il faut aussi mettre à jour le champ *pred* :

AJOUTEOBJETLISTEDC(L, Obj)

1. $Obj.next \leftarrow L.tete$
2. $L.tete \leftarrow Obj$
3. $Obj.next.pred \leftarrow Obj$

Pour effacer un livre de la liste, il faut chercher ce livre, mais cette fois, le champ *pred* nous permet d'écrire un algorithme plus simple :

ENLEVEOBJETLISTEDC(L, k)

1. $x \leftarrow \text{RECHERCHEOBJETLISTEDC}(L, k)$
2. **si** $x.pred \neq \text{null}$
3. $x.pred.next \leftarrow x.next$
4. **sinon**
5. $L.tete \leftarrow x.next$
6. **si** $x.next \neq \text{null}$
7. $x.next.pred \leftarrow x.pred$

Exercice : écrire l'algorithme AJOUTEAPRESOBJETLISTEDC(L, Obj, x) qui ajoute l'objet Obj après l'élément pointé par x dans un objet liste L .

4.5 Files et Piles

On a décrit une structure où on peut enlever n'importe quel élément. Cependant le coût de cette opération est linéaire en la taille de la structure. Souvent on ne veut pas enlever n'importe quel élément. Si on veut toujours enlever l'élément le plus ancien (c'est à dire le premier qui a été inséré dans la liste), on parlera d'une structure FIFO ("First In First Out"). Si on veut toujours enlever l'élément le plus récent (c'est à dire le dernier qui a été inséré dans la liste), on parlera d'une structure LIFO ("Last In First Out"). Les structures FIFO s'appellent également *files* ("queue" en anglais). Les structures LIFO s'appellent également *pile* ("stack" en anglais).

Piles La pile s'appelle comme ça car cela rappelle l'ordre dans lequel on peut retirer les livres empilés sur le sol. Le dernier livre qu'on a posé sera le premier qu'on prend. Les opérations d'une pile sont si fréquentes qu'elles méritent des noms spéciaux. Traditionnellement on associe à une pile un test qui regarde si la pile est vide. On a donc les opérations suivantes :

- PILEVIDE(P)
- EMPILE(P, Obj)
- DEPILE(P)

On peut voir un objet liste chaînée comme une pile. On peut facilement écrire les algorithmes qui correspondent aux opérations de pile. On laisse ces algorithmes comme exercice.

Un autre implémentation possible de la pile est fait à l'aide des tableaux : la file est toujours représentée par un objet avec un champ *tete*. Les éléments de la file sont stockés dans un tableau pointé du champ *tab*. Le champ *tete* contient l'indice de la case du tableau qui contient le premier élément. Si le champ *tete* est à 0, cela signifie que la pile est vide.

PILEVIDETAB(P)

1. **si** $P.tete = 0$
2. **renvoie true**
3. **sinon**
4. **renvoie false**

Pour empiler un élément (que ce soit un entier ou un objet), on l'ajoute à la prochaine case du tableau et on incrémente le champ *tete*. Si on n'a plus de cases, on a plusieurs possibilités - on pourrait renvoyer une erreur ou bien ne rien faire, ou encore agrandir le tableau - cela sont des détails d'implémentation qu'on ne discutera pas ici.

EMPILETAB($P, Elem$)

1. $P.tete \leftarrow P.tete + 1$
2. $P.tab[tete] \leftarrow Elem$

Pour dépiler, il suffit de décrémenter le champ *tete*. Il n'est pas nécessaire d'effacer l'élément, car il sera effacé au prochain empilage.

DEPILETAB(P)

1. $x \leftarrow P.tete$
2. $P.tete \leftarrow P.tete - 1$
3. **renvoie** $P.tab[x]$

Files La file s'appelle comme ça car cela rappelle l'ordre dans lequel on sert les clients au bureau de poste. Le premier qui entre est le premier à être servi, et les autres se mettent, justement, en file. Les opérations d'une file sont si fréquentes qu'elles méritent des noms spéciaux :

- ENFILE(F, Obj)
- DEFILE(F)

On peut implémenter une file à l'aide d'une liste doublement chaînée comme une file, à condition d'ajouter un pointeur vers le dernier élément de la file, la *queue* de la file. Un objet file sera donc un objet avec deux champs : un champ *queue* qui pointe vers le premier élément (le dernier à être entré dans le bureau de poste), et un champ *tete* qui pointe vers le dernier élément (le premier à être entré dans le bureau).

On peut maintenant écrire les algorithmes qui correspondent aux opérations de file. On laisse ces algorithmes comme exercice. On laisse aussi comme exercice d'implémenter un file à l'aide d'une liste chaînée simple.

Un autre implémentation possible de la file est fait à l'aide des tableaux : la file est représentée par un objet avec deux champs *tete* et *queue*, et un troisième champ *tab* qui pointe vers un tableau. Les éléments de la file sont stockés dans *tab*. Le champ *queue* contient l'indice de la case du tableau qui contient le premier élément, tandis que *tete* contient l'indice de la case du tableau qui contient le dernier élément.

Pour enfiler un élément, on l'ajoute à la prochaine case du tableau et on incrémente le champ *queue*. Si on n'a plus de cases, on recommence du début du tableau - c'est comme si le tableau était circulaire. Bien sûr il faut faire attention à ne pas effacer la queue, c'est à dire il faut éviter qu'en incrémentant *queue* on ait $tete = queue$. Par simplicité on n'écrit pas ces vérifications dans notre pseudocode.

Ici on indique par *max* la longueur du tableau $F.tab$

ENFILETAB($F, Elem$)

1. **si** $F.queue = max$
2. $F.queue \leftarrow 1$
3. **sinon**
4. $F.queue \leftarrow F.queue + 1$
5. $F.tab[queue] \leftarrow Elem$

Pour défiler, il suffit d'incrémenter le champ *tete* de façon similaire. Il faut faire attention aux cas où la file est vide : dans ce cas on ne doit rien renvoyer. Ici aussi on omet cette vérification.

DEFILETAB(F)

1. $x \leftarrow F.tete$
2. **si** $F.tete = max$
3. $F.tete \leftarrow 1$
4. **sinon**
5. $F.tete \leftarrow F.tete + 1$
6. **renvoie** $F.tab[x]$

Chapitre 5

Arbres

Cette section s'inspire en partie de [FGS94].

5.1 Arbres Binaires Simples

Un *nœud* d'un arbre binaire est un objet avec trois champs *val*, *gauche*, *droit*. Le champ *val* contient une valeur, tandis que les champs *gauche*, *droit* contiennent des pointeurs vers des nœuds. Si N est un nœud, les nœuds pointés par *gauche*, *droit* (quand ils ne sont pas **null**) sont les *fil*s de N . Un nœud est le *parent* de ses fils. Un nœud sans fils s'appelle une *feuille*.

Un *arbre binaire* est un pointeur vers un nœud, qui s'appelle la *racine*.

Une définition récursive d'arbre binaire est la suivante : un arbre binaire est soit un pointeur vers **null**, soit un pointeur vers un objet qui a un champ *val*, qui contient une valeur, et deux champs *gauche*, *droit* qui contiennent des arbres. Ces deux arbres s'appellent *sous-arbre gauche* et *sous-arbre droit*.

Un *objet arbre binaire* est un objet A avec un champ *racine* qui pointe vers un arbre binaire.

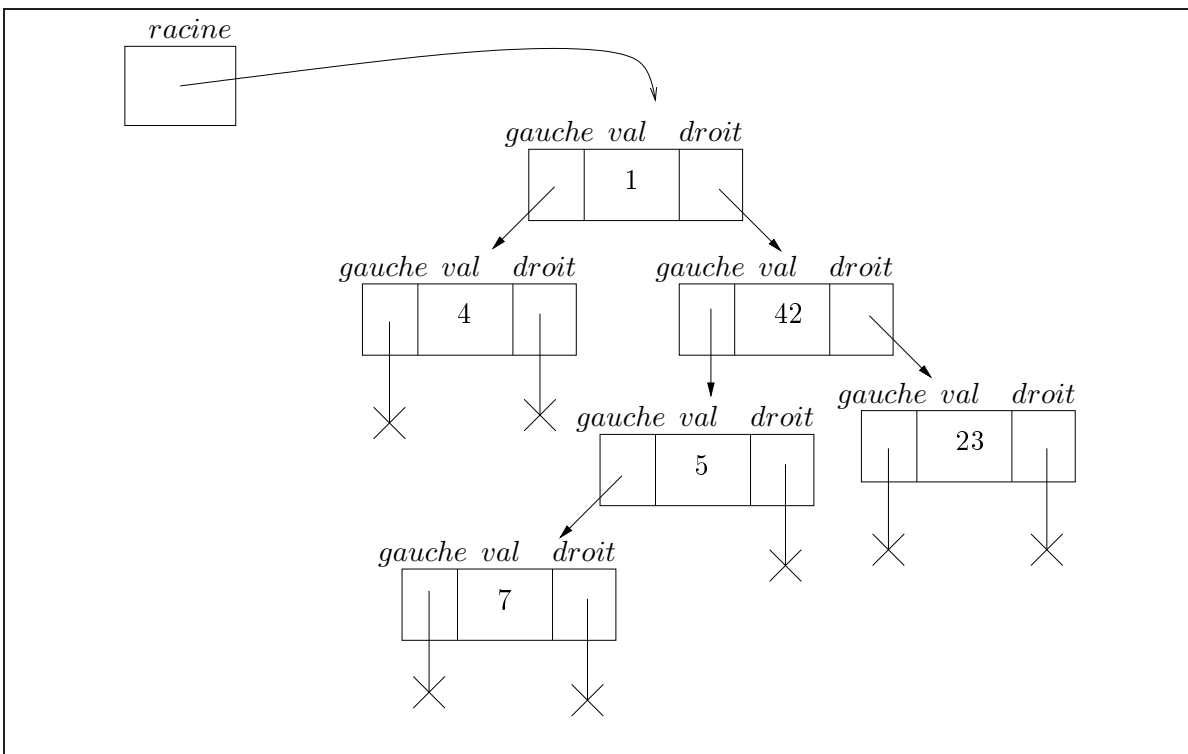


FIGURE 5.1: Un objet arbre binaire

Dans un arbre, les pointeurs vers **null** s'appellent les *fleurs*. Chaque feuille a deux fleurs, mais toutes les fleurs n'appartiennent pas à des feuilles.

Propriétés des arbres La *taille* d'un arbre binaire est le nombre de nœuds qu'il contient. La *hauteur* d'un nœud dans un arbre est le nombre de nœuds qui le séparent de la racine. La racine a donc hauteur 0, et le fils d'un nœud de hauteur h a hauteur $h + 1$. La hauteur d'un arbre est le maximum parmi les hauteurs des tous ses nœuds. La hauteur d'une fleur est définie comme la hauteur du nœud auquel elle appartient.

Un arbre binaire est *complet* si toutes les fleurs ont la même hauteur. Il est facile de prouver qu'un arbre binaire de hauteur m est complet si pour tout $h \leq m$, l'arbre contient 2^h nœud de hauteur h . Un simple calcul montre qu'un arbre binaire complet de hauteur m contient $2^{m+1} - 1$ nœuds. Donc la hauteur d'un arbre complet de taille n est $\Theta(\log_2 n)$. Les arbres qui ont cette même propriété sont dits *équilibrés*¹.

Un arbre binaire de hauteur m est *parfait* si toutes les fleurs ont hauteur m ou $m - 1$ et si toutes les feuilles de hauteur m sont "groupées à gauche".

5.2 Parcours d'un arbre binaire

On veut maintenant énumérer tous les éléments contenus dans un arbre. Ici on choisit de les afficher à l'écran, mais on pourrait également les stocker dans une pile P , ou dans un tableau.

Il y a plusieurs façon de *visiter* ou *parcourir* un arbre. On peut visiter un nœud avant de visiter ses fils, entre ses fils, ou après ses fils. Cela donne lieu à trois types de parcours : préfixe, infixe, et postfixe. On présent ici le parcours préfixe d'un arbre a , et on laisse les deux autres comme exercice.

PREPARCOURS(a)

1. si $a \neq \text{null}$
2. AFFICHE($a.val$)
3. PREPARCOURS($a.gauche$)
4. PREPARCOURS($a.droite$)

5.3 Arbres Binaires Avec Prédécesseur

Les arbres décrits ci-dessous ont les mêmes limites que les listes simplement chaînées. Si on veut enlever un élément d'un arbre il faut connaître son parent. Cela est plus facile quand chaque nœud contient un pointeur vers son parent, comme dans une liste doublement chaînée on a les pointeurs vers le prédécesseurs. Les nœuds de ces arbres ont un champ *par* qui pointe vers le parent, ou qui est **null** pour la racine.

5.4 Arbres Binaires de Recherche

Un *arbre binaire de recherche* est un arbre binaire a qui satisfait la propriété *dichotomique* suivante : pour tout nœud N contenu dans a

- les valeurs des nœuds du sous-arbre gauche de N sont tous plus petites ou égales à la valeur de N
- les valeurs des nœuds du sous-arbre droit de N sont tous plus grandes ou égales à la valeur de N

On considérera des arbres binaires de recherche avec prédécesseur.

L'opération principale sur les arbres binaires de recherche est... la recherche. Pour chercher un nœud avec une certaine valeur k on commence de la racine, et on suit le principe de la

1. Celle ci n'est pas une définition très formelle, car la notation Θ cache toujours une constante.

recherche dichotomique : si l'élément qu'on cherche est plus petit que la racine, on descend vers la gauche, sinon on descend vers la droite :

RECHERCHEABR(a, k)

1. **si** $a = \text{null}$ **renvoie** **null**
2. **si** $k = a.val$ **renvoie** a
3. **si** $k < a.val$
4. **renvoie** RECHERCHEABR($a.gauche, k$)
5. **sinon**
6. **renvoie** RECHERCHEABR($a.droite, k$)

Dans le pire des cas, il faut suivre la branche la plus longue de l'arbre. Donc si h est la hauteur de a , la recherche se fait en temps $\Theta(h)$. Si l'arbre est équilibré, on sait que $h \in \Theta(\log_2 n)$. Dans ce cas la recherche se fait en temps $\Theta(\log_2 n)$.

Pour insérer un nouvel élément dans l'arbre, on suit le même principe : par recherche dichotomique on cherche l'endroit où l'élément est censé se trouver et si cet endroit est libre on y insère l'élément. On donne ici une version récursive, et on laisse comme exercice d'écrire la version itérative. On suppose que N soit le nœud à insérer, avec les champs *gauche*, *droit*, *par* qui pointent vers **null**.

AJOUTEABR(a, N)

1. **si** $a = \text{null}$
2. $a \leftarrow N$
3. **sinon**
4. $N.par \leftarrow a$
5. **si** $N.val < a.val$
6. $a.gauche \leftarrow \text{AJOUTEABR}(a.gauche, N)$
7. **sinon**
8. $a.droit \leftarrow \text{AJOUTEABR}(a.droit, N)$
9. **renvoie** a

Effacer un élément est un peu plus difficile, car il faut à la fois garder la structure d'arbre, et la propriété dichotomique. On a besoin d'une discussion préliminaire.

D'abord on observe que pour rechercher l'élément minimum, il faut juste suivre la branche plus à gauche.

RECHERCHEMINABR(a)

1. **si** $a = \text{null}$ **renvoie** **null**
2. **si** $a.gauche = \text{null}$ **renvoie** $a.val$
3. **sinon** **renvoie** RECHERCHEMINABR($a.gauche$)

L'algorithme pour trouver le max suit la branche plus à droite. Cet algorithme a la même complexité que la recherche : $\Theta(h)$, ce qui signifie $\Theta(\log_2 n)$ pour des arbres équilibrés.

Une conséquence de la propriété dichotomique est que si on fait une visite infixe de l'arbre, on affiche les valeurs en ordre croissant. Cela peut se prouver facilement par induction (exercice pour ceux qui connaissent l'induction).

Un concept intéressant est celui de *successeur* d'un nœud N : le nœud qui contient la valeur plus petite, parmi tous les nœuds qui ont une valeur plus grande de N . Ce sera le nœud qui est visité juste après N au cours d'une visite infixe. On appellera ce nœud $S(N)$. Quand N a le fils droit, le nœud $S(N)$ sera toujours un descendant de N : il sera le minimum du sous-arbre droit de N . Si N n'a pas de fils droit le nœud $S(N)$ sera un ancêtre de N : il sera le plus proche ancêtre N' de N tel que N appartient au sous-arbre gauche de N' .

SUCCESEURABR(N)

1. **si** $N.droit \neq \text{null}$ **renvoie** RECHERCHEMINABR($N.droit$)
2. **sinon**
3. $y \leftarrow N$
4. **tant que** $y.par \neq \text{null}$ **et** $y = y.par.droit$ **faire**
5. $y \leftarrow y.par$
6. **renvoie** $y.par$

Vu qu'on parcourt au plus une branche, cet algorithme a complexité $\Theta(h)$.

On a maintenant tous les outils pour effacer un élément. Comme pour les listes, on trouvera plus facile d'écrire l'algorithme qui efface un nœud d'un objet arbre binaire de recherche A .

L'algorithme considère trois cas :

- Si le nœud N à effacer est une feuille, on l'enlève simplement.
- Si le nœud N à effacer a un seul fils, on le remplace par ce fils.
- Si le nœud N a deux fils, peut-on le remplacer par un des deux ? Non, car en général cela violerait les propriétés dichotomiques. En ce cas, on doit chercher $S(N)$, on échange sa valeur avec la valeur de N (et on échange les autres champs, s'il y en a) et finalement on élimine $S(N)$. Éliminer $S(N)$ est facile, car si N a deux fils, alors $S(N)$ a un seul fils (exercice). Donc on s'est réduit à un des deux premiers cas.

EFFACEABR(A, N)

1. // le pointeur y pointe vers le nœud à effacer
2. // le pointeur x pointe vers le nœud qui va prendre la place de y
3. // si N a au plus un fils, on va l'effacer : y est N
4. // si non on va effacer le successeur de N : y est $S(N)$
5. // dans les deux cas, y a au plus un fils
6. **si** $N.gauche = \text{null}$ **ou** $N.droit = \text{null}$
7. $y \leftarrow N$
8. **sinon**
9. $y \leftarrow \text{SUCESSEURABR}(N)$
10. // si le nœud à effacer a un fils, ce fils va prendre la place de y
11. // sinon c'est **null** qui va prendre la place de y
12. **si** $y.gauche \neq \text{null}$
13. $x \leftarrow y.gauche$
14. **sinon**
15. $x \leftarrow y.droit$
16. // si on doit effacer la racine, x est la nouvelle racine
17. // sinon on "passe autour" du nœud qu'on efface
18. **si** $x \neq \text{null}$
19. $x.par \leftarrow y.par$
20. **si** $y.par = \text{null}$
21. $A.racine \leftarrow x$
22. **sinon**
23. **si** $y = y.par.gauche$
24. $y.par.gauche \leftarrow x$
25. **si** $y = y.par.droit$
26. $y.par.droit \leftarrow x$
27. // si le nœud à éliminer est le successeur
28. // il faut copier la valeur du successeur dans N
29. **si** $y \neq N$
30. $N.val \leftarrow y.val$

Remarquons que toutes ces opérations sont en temps constant, sauf éventuellement la recherche du successeur qui a complexité $\Theta(h)$. Donc tout l'algorithme d'effacement a complexité $\Theta(h)$.

Chapitre 6

Le Tas

6.1 Arbres parfaits et tableaux

Les arbres parfaits peuvent également être représentés avec des tableaux. L'idée est que la relation fils-parent, en lieu d'être définie à l'aide de pointeurs, est définie par des relations arithmétiques entre les cases du tableau.

On imagine que les noeuds de l'arbre sont stockés dans le tableau *par niveaux* : La première case contient la racine, les deux cases suivantes contiennent les fils gauche et droit de la racine, le deux case suivantes les fils gauche et droit du fils gauche, et après les fils gauche et droit du fils droit, etc. Le fait que, dans le dernier niveau, les feuilles sont groupées à gauche rend cette représentation non-ambigüe.

Pour une case i , c'est facile de voir que son fils gauche se trouve dans la case $2 * i$, son fils droit dans la case $2 * i + 1$, tandis que son père se trouve dans la case $\lfloor i/2 \rfloor$. On aura donc les fonctions $gauche(i) = 2 * i$, $droit(i) = 2 * i + 1$, $pere(i) = \lfloor i/2 \rfloor$.

Finalement pour représenter un arbre binaire parfait on utilisera une variable *taille*, qui indique quel est, dans le tableau, la dernière case qui représente une feuille. Cela nous permettra d'enlever des feuilles à l'arbre, sans devoir réduire le tableau - il suffira de modifier la variable *taille*.

Pour résumer, un arbre binaire parfait T est un objet avec un champ $T.tab$, qui contient le tableau, et un champ $T.taille$.

6.2 Les Tas

Un *tas* ("Heap" en anglais) est un arbre binaire parfait T qui satisfait la *propriété du tas* : pour tout noeud i , on a que $T.tab[i] \geq T.tab[gauche(i)]$ et $T.tab[i] \geq T.tab[droit(i)]$. C'est à dire la valeur de chaque noeud est plus grande que la valeur de ses enfants.

On veut maintenant manipuler des tas : ajouter un noeud, enlever un noeud, construire un tas à partir d'un tableau quelconque.

Le premier algorithme qu'on voit consiste en l'ajout d'un noeud de valeur k au tas. L'idée est d'ajouter une feuille et de faire remonter le noeud jusqu'à ce que la propriété du tas soit satisfaite.

AJOUTETAS (T, k) :

1. $T.taille \leftarrow T.taille + 1$
2. $f \leftarrow T.taille$
3. $p \leftarrow pere(f)$
4. // on cherche le bon endroit où mettre k
5. **tant que** $k > T.tab[p]$ **et** $f > 1$
6. $T.tab[f] \leftarrow T.tab[p]$
7. $f \leftarrow p$
8. $p \leftarrow pere(p)$
9. // on echange les valeurs et on continue
10. $T.tab[f] \leftarrow k$

On rappelle l'algorithme qui échange le contenu de deux cases dans un tableau :

$\text{ECHANGE}(A, h, k)$

1. $\text{temp} \leftarrow A[h]$
2. $A[h] \leftarrow A[k]$
3. $A[k] \leftarrow \text{temp}$

En bref, on fait descendre la racine du bon côté, jusqu'à qu'elle se retrouve au bon endroit. Vu qu'on parcourt au pire une branche d'un arbre équilibré (les arbres parfaits sont équilibrés), la complexité de cet algorithme est logarithmique en la taille de l'arbre enraciné en i .

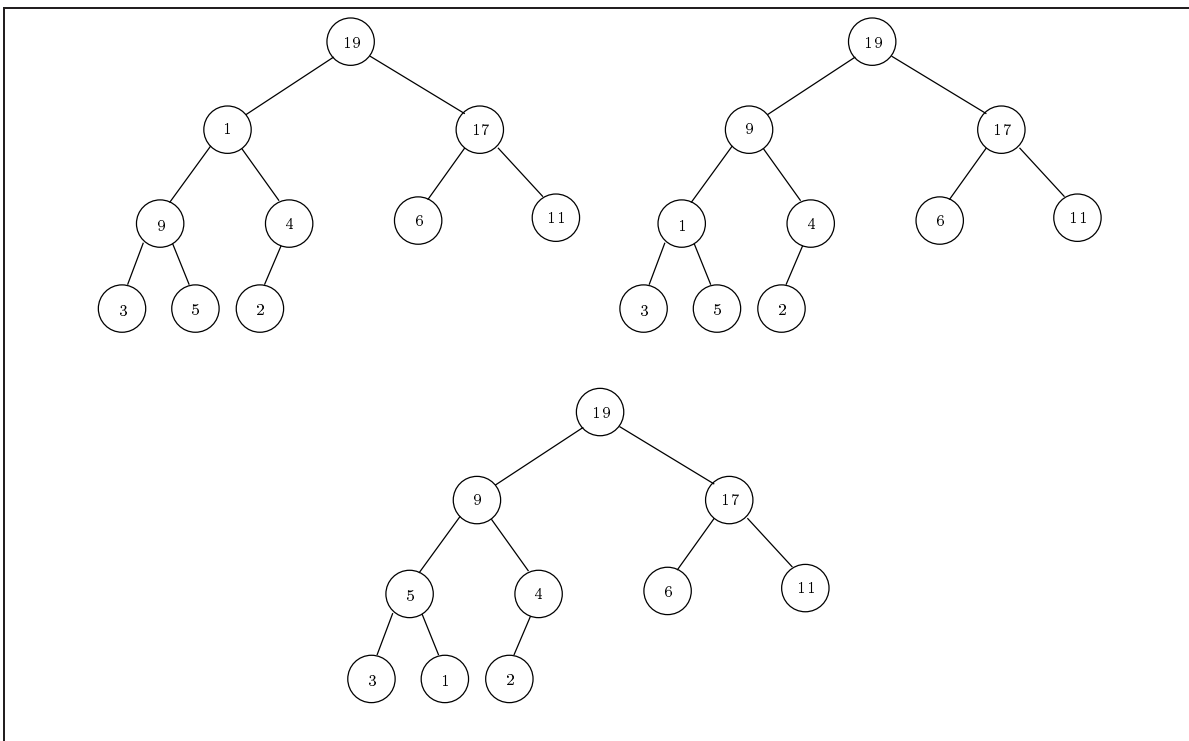


FIGURE 6.2: Un exemple de ENTASSER

Pour construire un tas on a deux stratégies : du haut vers le bas, et du bas vers le haut. Dans le premier cas on ajoute une feuille à la fois.

$\text{CREETASHAUTBAS}(A)$:

1. $T.\text{tab} \leftarrow A$
2. $T.\text{taille} \leftarrow 1$
3. **pour** $i \leftarrow 1$ **à** $\text{longueur}(A)$ **faire** :
4. $\text{AJOUTETAS}(T, A[i])$
5. **renvoie** T

Si n est la taille du tableau A , cet algorithme a une complexité de $n \log n$ (pourquoi ?)

L'intérêt de l'autre algorithme de création de tas est qu'il est linéaire.

CREETASBASHAUT(A) :

1. $T.tab \leftarrow A$
2. $T.taille \leftarrow longueur(A)$
3. $n \leftarrow \lfloor longueur(A)/2 \rfloor$
4. **pour** $i \leftarrow n$ **en descendant à 1 faire** :
5. ENTASSER (T, i)
6. **renvoie** T

Pour une explication formelle voir [CLRS02]. Intuitivement l'algorithme du haut vers le bas est cher car pour environ la moitié des noeuds, le coût d'insertion est logarithmique, tandis que dans l'algorithme du bas vers le haut, pour très peu de noeuds l'insertion coûte cher, et en effet pour environ la moitié l'insertion est en temps constant.

6.3 Tri par Tas

On peut utiliser un tas pour trier un tableau. D'abord on prend le tableau et on construit le tas. Le maximum du tableau se trouve à la racine. On "extraît" ce maximum, on le remplace par la feuille la plus à droite du tas, et on entasse. On obtient à nouveau un tas, où le deuxième plus grand élément du tableau est la nouvelle racine. On continue jusqu'à avoir vidé le tas. Les éléments ont été extraits dans l'ordre, et on les a mis dans l'ordre dans un tableau. L'astuce consiste à les mettre, à fur et à mesure, dans le tableau de départ. On n'utilise donc qu'une partie constante de mémoire (quelques variables pour les échanges). On parle donc d'un algorithme "en place".

La complexité de cet algorithme est de $n \log n$ (pourquoi?), la même que le tri par fusion. Mais le tri par fusion nécessite d'un tableau auxiliaire pour la fusion, et donc il n'est pas "en place". Si l'utilisation de la mémoire est un facteur critique, on préférera le tri par tas.

TRITAS (A) :

1. $T \leftarrow \text{CREETAS}(A)$
2. **pour** $i \leftarrow longueur(A)$ **en descendant à 2 faire** :
3. ECHANGE ($T.tab, 1, i$)
4. $T.taille \leftarrow T.taille - 1$
5. ENTASSER ($T, 1$)

Pour une analyse plus détaillée, voir [CLRS02], Chapitre 6.

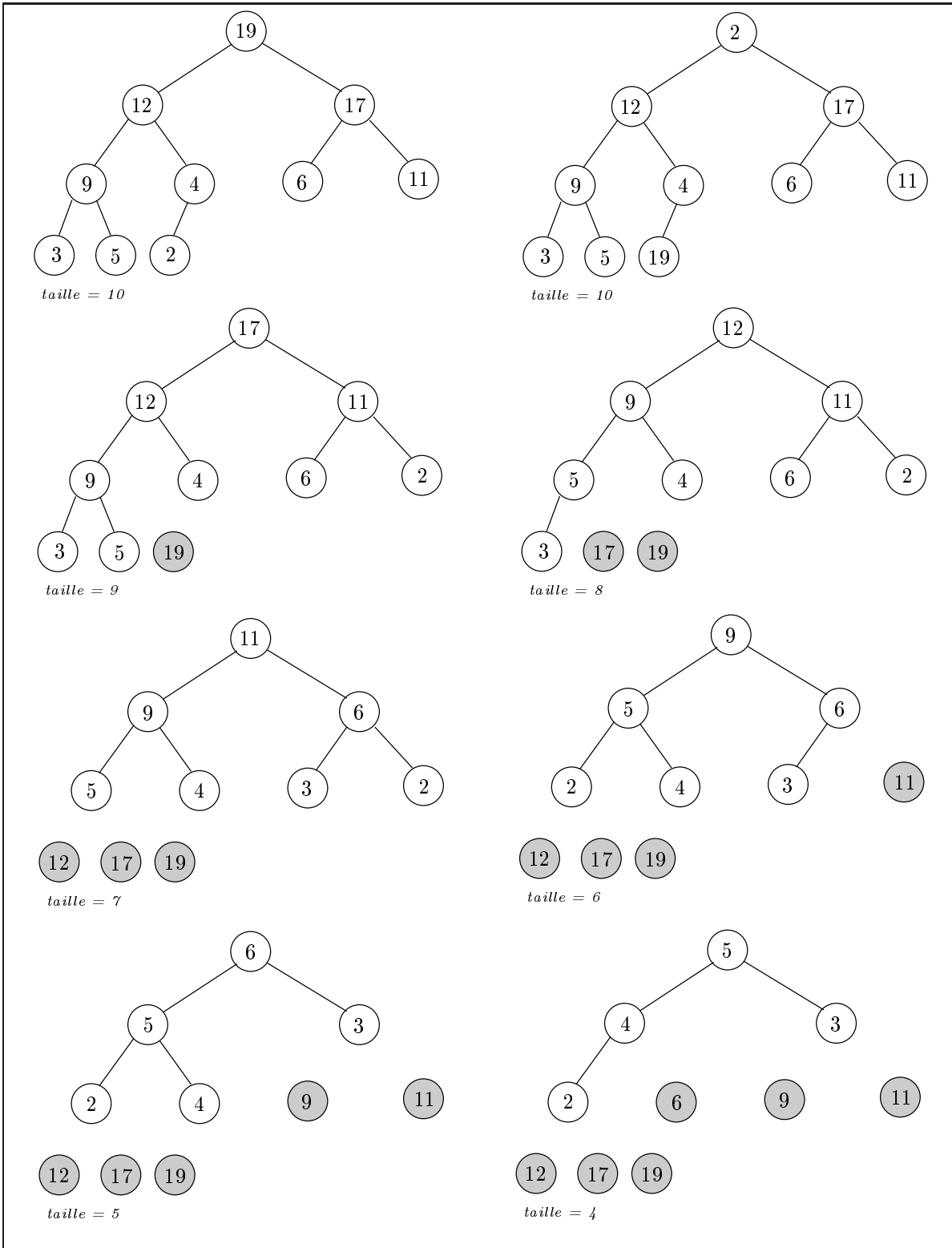


FIGURE 6.3: Les premiers pas d'un tri par tas

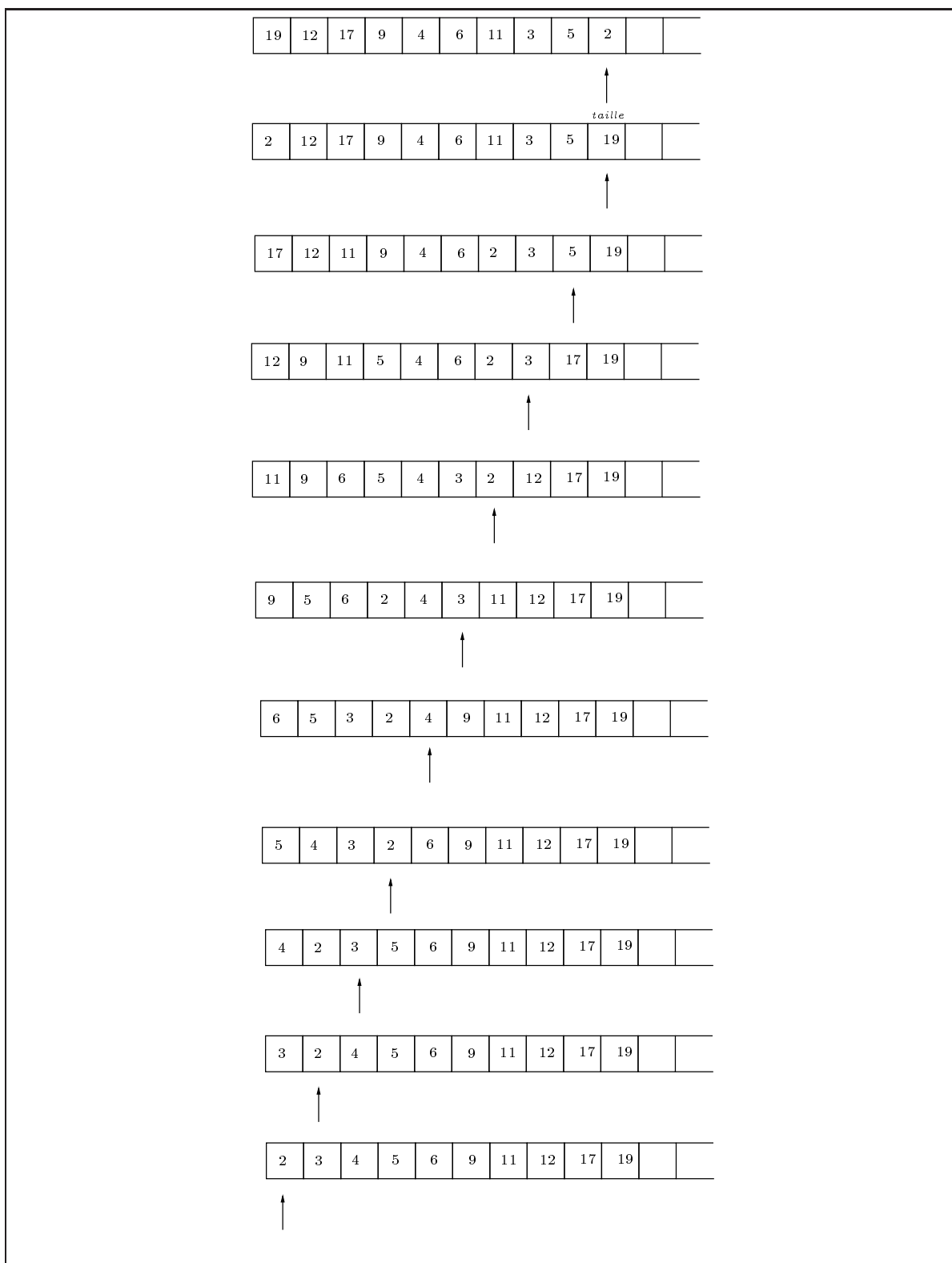


FIGURE 6.4: Le même tri par tas vu sur le tableau

Chapitre 7

Tables de hachage

Imaginons donc notre bibliothèque. Pour trouver facilement un livre (son numéro ISBN), il vaudrait mieux que les livres soient triés. On pourrait donc stocker les numéros dans un tableau trié. Cela permettrait un temps de recherche logarithmique en la taille de la bibliothèque. Mais qu'est-ce qui se passe quand on veut ajouter un nouveau livre à notre collection ? En moyenne, il faudra déplacer la moitié de nos livres, et dans le pire des cas il faudra les déplacer tous (quand on achète un livre avec un petit numéro ISBN), pour un temps d'insertion linéaire.

Pour éviter de déplacer tant de livres, il faudrait laisser plusieurs places libres dans notre bibliothèque. Idéalement, pour chaque livre possible, il faudrait laisser une place, au cas où on déciderait de l'acheter. Cela permettrait également un temps de recherche constant. Il suffirait de voir si la place du livre qu'on cherche est remplie ou pas. Mais il y a énormément de livres possibles, et on en achètera seulement une petite partie. Notre bibliothèque sera donc pour la plupart vide : un gaspillage de ressources.

Une solution serait de réserver une case pour plusieurs livres - le numéro de la case étant choisi en fonction du numéro ISBN. Encore une fois, la recherche serait en principe en temps constant. Mais en ayant moins de cases que de livres possibles, qu'est-ce qu'on ferait au cas où on achèterait un livre dont la case correspondante est déjà occupée ?

On présente dans la suite une réponse à ces questions en terme de *tables de hachage*. C'est une des solutions possibles pour obtenir une recherche et une insertion efficaces dans la bibliothèque. Pour d'autres solutions (arbres AVL, arbres rouges et noirs, etc.) on renvoie le lecteur intéressé à la littérature.

7.1 Résoudre les conflits

Le principe d'une table de hachage est très simple : on a un ensemble K de valeurs possibles (les livres), mais on sait qu'on en stockera seulement une petite partie. On alloue donc un tableau T , dont la taille m est de l'ordre de grandeur du nombre des livres qu'on prévoit d'avoir au maximum. Ensuite, on choisit une fonction $f : K \rightarrow m$ (qu'on appelle *fonction de hachage*). L'idée est qu'on stockera la valeur k dans la case $f(k)$. Vu que m est beaucoup plus petit que le nombre d'éléments de K , la fonction f enverra plusieurs valeurs dans la même case. À fur et à mesure qu'on remplit la bibliothèque, tôt ou tard, la fonction f cherchera à mettre un livre dans une case pleine. Il s'agit d'une situation de *conflit*. Que faire donc en cas de conflit ?

Le système le plus simple est de stocker dans chaque case du tableau un pointeur vers une liste chaînée. Chaque fois qu'on achète un livre, on l'insère en tête de la liste de la case correspondante. L'insertion est donc en temps constant. Cependant, pour la recherche, il faut, dans le pire des cas, parcourir toute la liste de la case correspondante au livre qu'on cherche. Dans le cas, très malheureux, mais théoriquement possible, où tous les livres correspondent à une seule et même case on pourrait avoir donc un temps de recherche linéaire en le nombre de livres. Mais en général il est peu probable que cela arrive. En pratique on peut penser que les livres sont bien distribués dans les cases, et que la longueur de chaque liste est bornée par une constante. En pratique donc on peut considérer le temps de recherche comme constant.

Pour une analyse plus fine, et pour d'autres stratégies pour la résolution des conflits, on renvoie au Chapitre 11 de [CLRS02].

7.2 Réduire les conflit

L'efficacité de la recherche dans une table de hachage dépend donc crucialement du fait que les données soient bien distribuées. Imaginons que notre fonction de hachage consiste en calculer la case en prenant seulement la partie "éditeur" du code ISBN. Si on achète pour notre bibliothèque surtout des livres de Hachette, on aura une case avec une liste très longue, et les autres presque vides. On veut choisir la case où mettre un livre de façon qu'il y ait le moins de probabilité que les livres s'accumulent tous dans le même endroit. Pour faire ça on choisira une fonction qui calcule la case à partir du code ISBN d'une façon plus complexe. L'idée est de bien "hacher" le code du livre pour en faire sortir la case correspondante. D'où le nom de fonction de *hachage*. Il n'y a pas une fonction "parfaite". Pour chaque fonction, on ne pourra pas éviter le pire des cas. Mais certaines propriétés sont du moins nécessaires pour une bonne fonction de hachage.

D'abord la fonction $f : K \rightarrow m$ doit être *surjective*, c'est-à-dire, toute case du tableau peut être utilisée. Aucune fonction constante, par exemple, ne sera une bonne fonction de hachage. Plus que cela, on veut que chaque case contienne au plus le même nombre de livres. C'est-à-dire, si K a N éléments, on veut que pour chaque $i \leq m$, le nombre d'éléments k de K tels que $f(k) = i$ soit N/m . Cela pourtant ne suffit pas. On voudrait également que, si on a n livres dans notre bibliothèque, en moyenne chaque case contienne n/m livres. Pour obtenir cela, il faut faire des hypothèses sur la distribution des données. Pour une analyse détaillée, on renvoie à la littérature. On se contente ici de montrer un exemple.

7.3 Exemple

La maison d'édition française "Musique d'antan" veut stocker les identificateurs des CD musicaux, représentés par des valeurs comprises entre 0 et $10^{12} - 1$, dans une table de hachage avec 10^4 cases. On suppose que les identificateurs des CD utilisent les conventions suivantes :

- Les deux premiers chiffres indiquent le pays de production.
- Les deux chiffres suivants indiquent la maison d'édition.
- Les six chiffres suivants sont spécifiques au CD.
- Les deux derniers chiffres indiquent le genre de la musique.

On considère les fonctions de hachage suivantes. Soit n la valeur à stocker :

- $f(n) = n \bmod (10^4)$
- $g(n) = \lfloor n/(10^8) \rfloor$
- $h(n) = \lfloor (n \bmod (10^8))/(10^4) \rfloor$
- $k(n) = \lfloor (n \bmod (10^8))/(10^6) \rfloor$

Quelle fonction devrait être utilisée en ces cas ? La fonction f garde les quatre derniers chiffres du code. Le nom de la maison d'édition suggère qu'elle se concentre sur un genre de musique. Donc les deux derniers chiffres du code de ses CD seront les mêmes. On utilisera seulement 100 cases du tableau sur les 10000 qu'on a à disposition. Donc f est un mauvais choix. La fonction g garde les quatre premiers chiffres du code. Cette fonction est encore pire, pour une maison d'édition donnée, elle est constante. On utilisera donc seulement 1 case du tableau. La fonction h garde les quatre chiffres du milieu du code. Avec les informations à notre disposition, on n'a pas raison de penser que les valeurs vont s'accumuler dans certaines cases. A priori donc on peut utiliser h comme fonction de hachage. La fonction k ne garde que deux chiffres du code. Elle n'utilise qu'une petite partie des cases disponibles, et elle est donc un mauvais choix.

Chapitre 8

Autres sujets

Autres arguments d'intérêt pour ce cours se trouvent dans [CLRS02], chapitres 15 (programmation dynamique), 32 (recherche de sous-chaines), 34 (les problèmes intrinsèquement difficiles), et seront traités en fonction du temps disponible.

Bibliographie

- [CLRS02] T. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction à l'Algorithmique - 2ème édition*. Dunod, 2002. Partiellement disponible en ligne à : <http://www.numilog.com> Version anglaise disponible sur Google Books.
- [FGS94] Ch. Froidevaux, M.-C. Gaudel, and M. Soria. *Types de Données et Algorithmes*. Ediscience, 1994.