Fondements de l'informatique
Structures de données et algorithmes

Frédéric Boulanger

11 mars 2014

Table des matières

Chapitre 1

Introduction

Qu'est-ce que l'informatique ?

L'importance du numérique : son, image, texte, video sont traités par les même appareils (par opposition aux films argentiques, disques vinyle et livres papier).

L'informatique à Supélec.

Programme du cours.

Utilisation de Java

Notion d'algorithme, distinction algorithme/programme.

1.1 Abstraction des données

Utilisation de services (invocation de méthodes)

Inutile de connaître la représentation exacte des données.

Exemples. Différence entre un tableau de dates et trois tableaux parallèles de jours, mois et années. Classe String de Java. Différence entre une String et un tableau de caractères.

Chapitre 2

Types abstraits algébriques

2.1 Introduction

Lors de la conception d'une application, on a souvent besoin de définir de nouveaux types de données (ou de nouvelles classes) correspondant aux objets spécifiques manipulés par cette application.

Il importe de définir ces nouveaux types de données par leur comportement (leurs fonctionnalités) et non par leur représentation en mémoire.

On définit donc un nouveau type en spécifiant les opérations que l'on souhaite effectuer sur les données de ce type.

Si on utilise un langage à objets pour écrire l'application, le type de données sera associé à une classe, et chaque opération à une méthode de la classe.

Si on utilise un langage ne possédant pas la notion d'objet, comme C ou PASCAL, le type de données sera associé à un type du langage, et chaque opération sera associée à une fonction travaillant sur des données de ce type.

Lors de la conception de l'application, le type ne sera connu dans un premier temps que par son nom et le nom des opérations associées.

Il faudra bien sûr aussi indiquer pour chaque opération sur quelles données elle travaille et le résultat qu'elle rend (par exemple, l'opération + sur les entiers reçoit deux entiers et rend un entier).

Mais il faut aussi préciser pour chaque opération ce qu'elle fait en indiquant quelles doivent être les propriétés vérifiées par le résultat qu'elle rend.

Dans un premier temps, on ne s'intéresse ni à la représentation des données en mémoire, ni à la manière de réaliser les opérations. On se situe donc à un niveau pratiquement indépendant du langage de programmation qu'on utilisera par la suite.

Spécifier les propriétés du résultat d'une opération peut se faire « en bon français »(par exemple, pour l'opération + sur les entiers, on peut indiquer que le résultat est la somme des paramètres).

Toutefois, une formulation en langage naturel est dans de nombreux cas une source d'ambiguïté. C'est pourquoi nous allons utiliser une méthode mathématique rigoureuse pour spécifier un type de données : les types abstraits algébriques.

Cette méthode nous fournira aussi des outils permettant de vérifier que l'implémentation est conforme à la spécification.

2.2 Écriture d'une spécification

2.2.1 Exemple des booléens

Nous allons présenter cette méthode de spécification sur un exemple simple : le type de données booléen.

spec BOOL_0
  sorte Bool
  operations
    vrai : $\rightarrow$ Bool
    faux : $\rightarrow$ Bool
    non : Bool $\rightarrow$ Bool
    et : Bool Bool $\rightarrow$ Bool
    ou : Bool Bool $\rightarrow$ Bool
  axiomes a, b, c : Bool

```
    (b1) non(vrai) = faux
    (b2) non(non(a)) = a
    (b3) et(a, vrai) = a
    (b4) et(a,faux) = faux
    (b5) et(et(a,b),c) = et(a,et(b,c))
    (b6) et(a,b) = et(b,a)
    (b7) ou(a,b) = non(et(non(a),non(b)))
fspec
```

Une spécification commence par le mot clé *spec* suivi du nom que l'on souhaite donner à la spécification, et se termine par le mot clé *fspec*.

Derrière le mot clé *sorte*, on donne la liste des types de données définis par cette spécification (nommés dans une telle spécification les sortes).

Si cela est possible, on essaiera de ne définir qu'une seule sorte dans une spécification. Une spécification multi-sortes est toujours plus compliquée, et ne doit être envisagée que si on ne peut faire autrement (par exemple dans le cas de types fortement imbriqués).

On trouvera ensuite, derrière le mot clé *operations*, la liste des opérations que l'on souhaite pouvoir effectuer sur notre type de données, accompagnées de ce que l'on appelle leur profil : il s'agit de donner pour chaque opération le nombre et le type des opérandes, ainsi que le type de la valeur rendue (devant la flèche, la liste des opérandes, derrière la flèche la sorte du résultat).

Les opérations sans opérande (ici *vrai* et *faux*) sont nommées (à juste titre !) des constantes.

Les lignes qui suivent le mot clé *axiomes* donnent les propriétés que doivent vérifier les opérations.

Nous verrons par la suite la signification formelle de ces axiomes. De manière plus pragmatique, sur un exemple :

(b3) et(a, vrai) = a

b_3 est simplement une étiquette (facultative) permettant de référencer par la suite cet axiome. On trouve ensuite, de part et d'autre du signe =, deux expressions, chacune d'entre elles pouvant utiliser des variables (celles déclarées derrière le mot clé *axiomes*). L'axiome indique que, quelles que soient les valeurs de ces variables, l'expression de gauche doit avoir même valeur que celle de droite.

Cet ensemble d'axiomes est l'ensemble des propriétés qui doivent être vérifiées pour que les opérateurs donnent le résultat escompté. L'ensemble d'axiomes définit le comportement des opérations. En vérifiant les axiomes pour une implémentation, on vérifie ainsi qu'elle a le comportement spécifié.

Pour des raisons de commodité d'usage, il est aussi possible d'utiliser une notation plus classique pour certains opérateurs que la notation fonctionnelle. Dans cette notation, dite in-fixée, les opérandes figurent de part et d'autre de l'opérateur. On prendra alors soin de placer dans le nom de l'opérateur (au niveau du profil) un certains nombres de caractères _, le nième caractère _ symbolisant la place du nième opérande. Par exemple :

_∧_ : Bool Bool → Bool

indique que le *et* de a et b s'écrit $a \wedge b$.

La spécification du type booléen devient alors :

```
spec BOOL1
  sorte Bool
  operations
    vrai :              → Bool
    faux :              → Bool
    ¬_   : Bool         → Bool
    _∧_  : Bool Bool → Bool
    _∨_  : Bool Bool → Bool
  axiomes a, b, c : Bool
    (b1) ¬vrai = faux
    (b2) ¬¬a = a
    (b3) a∧vrai = a
    (b4) a∧faux = faux
    (b5) (a∧b)∧c = a∧(b∧c)
    (b6) a∧b = b∧a
    (b7) a∨b = ¬((¬a)∧(¬b))
fspec
```

2.2.2 Signature

Définition 1 (Signature) *La signature d'une spécification est l'ensemble des sortes de cette spécification, associé à l'ensemble des noms d'opérations et de leur profil.*

Il s'agit donc des paragraphes *sorte(s)* et *operations* de la spécification.

2.2.3 Termes

Nous allons maintenant présenter de manière plus rigoureuse que nous l'avons fait dans le paragraphe précédent la manière de spécifier le résultat de chaque opération, autrement dit la manière d'écrire les axiomes.

De manière informelle, on appelle terme toute expression que l'on peut construire à partir des opérateurs et des variables en respectant le profil des opérations. Un terme sans variable est appelé un terme clos.

Définissons maintenant de manière plus formelle cette notion de terme.

Définition 2 (Alphabet) *Un alphabet est un ensemble de symboles. Un mot construit sur un alphabet A est une suite finie d'éléments de A, éventuellement vide.*

Soit une spécification *SPEC* et un ensemble X de variables. On considère l'alphabet A constitué des opérateurs de *SPEC*, des variables de X, des parenthèses ouvrante et fermante et de la virgule.

Définition 3 (Termes) *L'ensemble $T(X)_S$ des termes de sorte S de la spécification SPEC avec variables dans X est le plus petit ensemble de mots construits sur A contenant :*

- *Les symboles de constantes de SPEC de sorte S.*
- *Les variables de X de sorte S*
- *Les suites de symboles de la forme $f(t_1,\ldots,t_n)$ où f est un opérateur de profil $f : S_1 \ldots S_n \to S$ et $t_1 \ldots t_n$ des termes de $T(X)_{S_1} \ldots T(X)_{S_n}$*
- *Les suites de symboles de la forme $t_1\ f\ t_2$ où f est un opérateur de profil $\_f\_ : S_1\ S_2 \to S$ et t_1 un terme de sorte $T(X)_{S_1}$ et t_2 un terme de sorte $T(X)_{S_2}$.*
- *Les suites de symboles de la forme $f\ t_1$ où f est un opérateur de profil $f\_ : S_1 \to S$ et t_1 un terme de sorte $T(X)_{S_1}$.*
- *Les suites de symboles de la forme $t_1\ f$ où f est un opérateur de profil $\_f : S_1 \to S$ et t_1 un terme de sorte $T(X)_{S_1}$.*

L'ensemble $T(X)$ des termes de SPEC avec variables dans X est l'union de tous les $T(X)_S$ pour toutes les sortes S de SPEC.

L'ensemble $T(\emptyset)_S$ encore noté T_S est l'ensemble des termes clos de sorte S de SPEC.

2.2.4 Axiomes

Définition 4 (Axiome) *Un axiome est une expression de la forme :*

$t_1 = t_2$

où t_1 et t_2 sont des termes de SPEC de la même sorte (termes avec ou sans variables).

Intuitivement, les axiomes permettent d'indiquer quels sont les termes clos qui doivent être « égaux », donnant ainsi les propriétés que doivent posséder les opérations pour qu'elles fassent ce que l'on attend d'elles. Ces termes clos sont obtenus en remplaçant les variables par des termes clos de la même sorte que les variables remplacées, une même variable devant bien sûr être remplacée partout par le même terme.

Toutes les autres propriétés que les résultats des opérations doivent posséder doivent pouvoir être démontrées à l'aide des seuls axiomes de la spécification (ou encore à l'aide de propriétés précédemment démontrées), en utilisant une méthode que nous verrons par la suite.

Cette notion « d'égalité »de termes clos peut être définie de manière plus formelle. Les axiomes induisent une relation $\mathcal{R}$ entre termes clos définie par :

Définition 5 *Soient t_1 et t_2 deux termes clos. On a $t_1 \mathcal{R} t_2$ si et seulement si il existe un axiome tel qu'en remplaçant les variables par des termes clos on obtienne t_1 à gauche du signe égal et t_2 à droite.*

Cette relation $\mathcal{R}$ induit une autre relation sur les termes clos notée $\cong$ et nommée *congruence* définie par les règles suivantes :

- $\forall (t_1, t_2), t_1 \mathcal{R} t_2 \Rightarrow t_1 \cong t_2$
- $\forall t, t \cong t$
- $\forall (t_1, t_2), t_1 \cong t_2 \Rightarrow t_2 \cong t_1$
- $\forall (t_1, t_2, t_3), t_1 \cong t_2$ et $t_2 \cong t_3 \Rightarrow t_1 \cong t_3$

— Pour tout opérateur f de profil $f : S_1 \dots S_n \to S$
Pour tous termes $t_1 \dots t_n$ de sortes $S_1 \dots S_n$ et $t'_1 \dots t'_n$ de sortes $S_1 \dots S_n$
$t_1 \cong t'_1, \dots, t_n \cong t'_n \Rightarrow f(t_1, \dots, t_n) \cong f(t'_1, \dots, t'_n)$

Cette relation est, par construction, une relation d'équivalence. Elle respecte de plus les opérateurs. On appelle une telle relation une congruence. Les termes que nous avons appelés « égaux »sont en fait des termes appartenant à la même classe d'équivalence (de congruence).

La relation $\cong$ des spécifications $BOOL_0$ et $BOOL_1$ possède deux classes de congruence, le terme *vrai* étant un représentant de l'une d'entre elle et le terme *faux* un représentant de l'autre (ceci restant bien sûr à démontrer).

2.3 Théorèmes

Un théorème est une propriété que l'on peut démontrer à l'aide des axiomes de la spécification et des théorèmes précédemment démontrés.

Un théorème possède la même forme qu'un axiome :

$t_1 = t_2$

où t_1 et t_2 sont des termes (avec ou sans variables) de la même sorte.

Un théorème possède la même signification intuitive qu'un axiome : il indique que si on remplace dans t_1 et t_2 les variables par des termes clos de la même sorte, les termes obtenus sont alors « égaux »(en fait ils sont congrus au sens de la relation $\cong$).

La démonstration de théorèmes doit permettre de confirmer que les résultats des opérations de la spécification sont ceux qui sont escomptés. Si un théorème jugé utile ne peut être démontré, il est alors légitime de se demander si le jeu d'axiomes de la spécification est suffisant, et s'il n'est pas opportun de le compléter. A l'opposé, si on arrive à démontrer une propriété non souhaitée (i.e. manifestement fausse), il est alors indispensable de vérifier les axiomes afin d'isoler celui ou ceux qui sont erronés.

A titre d'exemple, en considérant la spécification $BOOL_1$, on doit être capable de démontrer $vrai \wedge a = a$ et $a \vee vrai = vrai$.

Par contre, si on arrive à démontrer $vrai = faux$, cela indique clairement qu'un ou plusieurs axiomes sont erronés.

On disposera de deux méthodes pour démontrer un théorème, basées sur la définition de la relation $\cong$.

Définissons auparavant une notation :

Soit t un terme avec variables utilisant la variable x de sorte S. Soit u un terme de sorte S (avec ou sans variables). On note $t[u/x]$ le terme obtenu en remplaçant partout la variable x par le terme u.

2.3.1 Théorèmes déductifs

Soit à démontrer le théorème $t_1 = t_2$.

A partir d'un des deux termes (t_1 ou t_2), on applique une succession de transformations jusqu'à obtenir le deuxième.

Un terme t peut être remplacé par un terme t' si et seulement si $t = t'$ est un axiome ou un théorème.

Les cinq règles suivantes permettent de construire directement des théorèmes à partir des axiomes ou de théorèmes précédemment démontrés :

— R1 : réflexivité
Pour tout terme t, $t = t$ est un théorème

— R2 : symétrie
Pour tous termes t_1 et t_2 de même sorte, si $t_1 = t_2$ est un théorème ou un axiome, alors $t_2 = t_1$ est un théorème.
Par exemple, selon $BOOL_1$, $a = a \wedge vrai$ est un théorème puisque $a \wedge vrai = a$ est un axiome.

— R3 : transitivité
Pour tous termes t_1, t_2 et t_3 de même sorte, si $t_1 = t_2$ et $t_2 = t_3$ sont des théorèmes ou des axiomes, alors $t_1 = t_3$ est un théorème.

— R4 : substitution
Pour tous termes t_1 et t_2 de même sorte avec variables dans X,
pour tout $x \in X$,
pour tout terme u de même sorte que x,
si $t_1 = t_2$ est un théorème ou un axiome, alors $t_1[u/x] = t_2[u/x]$ est un théorème.

— R5 : congruence
Pour tous termes $t_1, \dots, t_n$ de sortes $S_1, \dots, S_n$,
Pour tous termes $t'_1, \dots, t'_n$ de sortes $S_1, \dots, S_n$,
Pour toute opération de profil $f : S_1 \dots S_n \to S$,

Si $t_1 = t'_1, \ldots, t_n = t'_n$ sont des théorèmes ou des axiomes, alors $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ est un théorème.

Exemple 1

Avec la spécification $BOOL_1$, montrons le théorème suivant :

(th₁) $\neg faux = vrai$

On a $faux = \neg vrai$ par b_1 et $R2$

donc $\neg faux = \neg\neg vrai$ par $R5$

$\neg\neg vrai = vrai$ par b_2 et $R4$

donc $\neg faux = vrai$ par $R3$

Exemple 2

Montrons, avec la spécification $BOOL_1$, que $a \vee vrai = vrai$ est un théorème.

$a \vee vrai = \neg((\neg a) \wedge (\neg vrai))$ par b_7 et $R4$

$\neg((\neg a) \wedge (\neg vrai)) = \neg((\neg a) \wedge faux)$ par b_1 et $R5$

or $(\neg a) \wedge faux = faux$ par b_4 et $R4$

donc :

$\neg((\neg a) \wedge faux) = \neg(faux)$ par $R5$ et le théorème précédent

$\neg faux = vrai$ par le théorème de l'exemple 1

donc :

$a \vee vrai = vrai$ par $R3$

Exemple 3

Montrons, avec la spécification $BOOL_1$, que $a \wedge b = \neg((\neg a) \vee (\neg b))$ est un théorème.

$\neg((\neg a) \vee (\neg b)) = \neg(\neg(\neg(\neg a) \wedge \neg(\neg b)))$ par b_7 et $R5$

$\neg(\neg(\neg(\neg a) \wedge \neg(\neg b))) = \neg(\neg(a \wedge b))$ par b_2 et $R5$

$\neg(\neg(a \wedge b)) = a \wedge b$ par b_2 et $R4$

donc $\neg((\neg a) \vee (\neg b)) = a \wedge b$ par $R3$

et donc $a \wedge b = \neg((\neg a) \vee (\neg b))$ par $R2$

2.3.2 Induction structurelle

Soit une spécification *SPEC*, X un ensemble de variables et $T(X)$ l'ensemble des termes avec variables dans X.

Soit P un prédicat sur les éléments de $T(X)$, c'est-à-dire une propriété telle que pour tout élément t de $T(X)$, $P(t)$ soit vérifiée ou non.

La propriété P est vérifiée pour tout terme t de $T(X)$ si :

1. $P(t)$ est vérifiée pour toutes les constantes de *SPEC* et toutes les variables de X.
2. Pour tout terme t de la forme $f(t_1, \ldots, t_n)$, si $P(t_1), \ldots, P(t_n)$ sont vérifiées, alors $P(t)$ est vérifiée.

Exemple

Considérons la spécification $BOOL_1$, un ensemble X de variables vide (donc $T(X)$ est l'ensemble des termes clos de $BOOL_1$).

Montrons que tout terme clos t est congru à *vrai* ou à *faux* (selon la relation $\cong$). Cela revient à dire que pour tout terme clos t, t=*vrai* est un théorème ou t=*faux* est un théorème. Notons cette propriété P.

1. La propriété est vérifiée pour *vrai* et *faux* car *vrai*=*vrai* et *faux*=*faux* sont des théorèmes.
2. Les termes sont de trois formes différentes : $\neg t$, $t_1 \wedge t_2$ ou $t_1 \vee t_2$.
   (a) forme $\neg t$ avec $P(t)$
   — si t=*vrai*,
   $\neg t = \neg vrai$ par $R5$
   $\neg vrai = faux$ par b_1
   donc $\neg t = faux$ par $R3$
   — si t=*faux*,
   $\neg t = \neg faux$ par $R5$
   $\neg faux = vrai$ par le théorème th1 de l'exemple 1
   donc $\neg t = vrai$ par $R3$
   (b) forme $t_1 \wedge t_2$ avec $P(t_1)$ et $P(t_2)$
   — si t_1=*faux*
   $t_1 \wedge t_2 = faux \wedge t_2$ par $R5$
   $faux \wedge t_2 = t_2 \wedge faux$ par b_6
   $t_2 \wedge faux = faux$ par b_4 et $R4$
   donc $t_1 \wedge t_2 = faux$ par $R3$

— si t_1=*vrai* et t_2=*vrai*,
$t_1 \wedge t_2 = vrai \wedge vrai$ par $R5$
$vrai \wedge vrai = vrai$ par b_3 et $R4$
donc $t_1 \wedge t_2 = vrai$ par $R3$
— si t_1=*vrai* et t_2=*faux*, $t_1 \wedge t_2 = vrai \wedge faux$ par $R5$
$vrai \wedge faux = faux$ par b_4 et $R4$
donc $t_1 \wedge t_2 = faux$ par $R3$

(c) forme $t_1 \vee t_2$ avec $P(t_1)$ et $P(t_2)$
— si t_1=*vrai*, $t_1 \vee t_2 = vrai \vee t_2$ par $R5$
$vrai \vee t_2 = t_2 \vee vrai$ par un théorème à démontrer et $R4$
$t_2 \vee vrai = vrai$ par le théorème de l'exemple 2 et $R4$
donc $t_1 \vee t_2 = vrai$ par $R3$
— si t_1=*faux* et t_2=*vrai*, $t_1 \vee t_2 = faux \vee vrai$ par $R5$
$faux \vee vrai = vrai$ par le théorème de l'exemple 2 et $R4$
donc $t_1 \vee t_2 = vrai$ par $R3$
— si $t_1 = faux$ et $t_2 = faux$, $t_1 \vee t_2 = faux \vee faux$ par $R5$
$faux \vee faux = faux$ par un théorème à démontrer et $R4$
donc $t_1 \vee t_2 = faux$ par $R3$

2.3.3 Théorèmes inductifs

Tous les théorèmes ne peuvent pas être démontrés de manière déductive, c'est-à-dire en appliquant uniquement une suite de transformations.

Par exemple, on ne saurait montrer de cette manière le théorème $t \wedge t = t$ avec la spécification $BOOL_1$, aucun axiome ne permettant de transformer le terme $t \wedge t$.

Il est toutefois possible de monter ce théorème par une autre méthode dite par induction.

Définition 6 (théorème inductif) *$t_1 = t_2$ est un théorème inductif si toute équation obtenue à partir de $t_1 = t_2$ en remplaçant les variables par des termes clos de la même sorte que ces variables est un théorème déductif.*

Il est possible de démontrer le théorème $t \wedge t = t$ de cette manière, en utilisant le résultat vu dans le paragraphe précédent qui permet d'affirmer que pour tout terme clos t, t=*vrai* ou t=*faux* est un théorème.

— si t=*vrai* est un théorème, alors $t \wedge t = vrai \wedge vrai = vrai = t$
— si t=*faux* est un théorème, alors $t \wedge t = faux \wedge faux = faux = t$

2.4 Extension d'une spécification

Le procédé d'extension de spécification permet de construire de nouvelles spécifications en incluant d'autres.

Donnons pour commencer la spécification d'un type « entier naturel »extrêmement simplifié.

```
spec NAT0
  sorte Nat
  operations
    0    :       → Nat
    succ : Nat → Nat
  axiomes
fspec
```

Cette spécification est munie d'un opérateur 0 (constante), et d'un opérateur *succ* donnant le successeur d'un entier. Elle ne possède pas d'axiome.

Tout terme clos de NAT_0 possède donc la forme :

— soit de la constante 0
— soit d'une suite d'utilisations de *succ* sur cette même constante.

L'opérateur *succ* possède une propriété bien particulière : il permet de construire tous les termes clos à partir de la constante 0. Un tel opérateur, permettant de « fabriquer »tous les termes clos à partir des constantes est qualifié de *constructeur*.

Nous souhaitons maintenant ajouter des opérations arithmétiques classiques à ce type « entier naturel ». Nous pouvons le faire sans avoir à réécrire complètement la spécification, mais en en définissant une nouvelle qui « étend »NAT_0.

spec NAT_1
etend NAT_0
operations
_+_ : Nat Nat → Nat
axiomes n,m : Nat
0 + n = n
succ(m) + n = succ(m + n)
fspec

On peut remarquer que cette spécification n'introduit pas de nouvelle sorte. Cela reste toutefois possible dans le cas général.

Lorsque l'on a réussi à isoler un constructeur (ici *succ*), la méthode à employer pour écrire les axiomes permettant de spécifier les autres opérateurs est, dans la plupart des cas, extrêmement simple et systématique : il suffit de donner des axiomes donnant le résultat de l'opération pour les constantes et pour les termes construits.

Une spécification peut en étendre plusieurs. Si on souhaite ajouter à NAT_1 un opérateur de comparaison noté ==, on peut le faire par une nouvelle spécification NAT_2 étendant à la fois $BOOL_1$ et NAT_1.

spec NAT_2
etend NAT_1,$BOOL_1$
operations
_==_ : Nat Nat → Bool
axiomes n,m : Nat
0 == 0 = vrai
succ(n) == 0 = faux
0 == succ(n) = faux
succ(n) == succ(m) = n == m
fspec

2.5 Termes conditionnels

Dans de nombreux cas, il est difficile d'exprimer le résultat d'une opération par un terme simple. Il est souvent utile de pouvoir dire que le résultat doit être un terme t_1 ou un autre terme t_2 en fonction d'une condition exprimée sur les paramètres.

A cet effet, toute spécification étendant $BOOL$ peut être munie, pour chaque sorte S, d'un opérateur que l'on peut noter $\boldsymbol{si}_S$:

si$_S$: Bool S S → S

défini par les 2 axiomes :

si$_S$(vrai ,x,y) = x
si$_S$(faux,x,y) = y

Pour des raisons de commodité d'écriture, on introduit un opérateur *si_alors_sinon_fsi* de profil :

si _alors_sinon_fsi : Bool S S → S

répondant aux mêmes spécifications, mais valable pour toute sorte S. Cet opérateur est dit polymorphe.

si vrai **alors** x **sinon** y **fsi** = x
si faux **alors** x **sinon** y **fsi** = y

Cet opérateur est supposé implicitement défini pour toute spécification étendant $BOOL$

Exemple : Spécification d'un type ensemble d'entiers

spec ENS
etend NAT_1, $BOOL_1$
sorte Ens
operations
∅ : → Ens
i : Ens Nat → Ens *ajoute un élément à un ensemble*
_∈_ : Nat Ens →Bool
card : Ens → Nat
s : Ens Nat →Ens *supprime un élément d'un ensemble*
axiomes x,y:Nat;E:Ens

```
    x ∈ ∅ = faux
    x ∈ i(E,y) = x==y ∨ x∈E
    card(∅) = 0
    card(i(E,x)) = si x ∈ E alors card(E) sinon succ(card(E)) fsi
    s(∅,x) = ∅
    s(i(E,x),y) = si x∈E
                  alors s(E,y)
       sinon
                    si x==y alors E sinon i(s(E,y),x) fsi
     fsi
fspec
```

Signalons que l'opérateur i est un constructeur du type ensemble.

2.6 Domaine de définition des opérateurs

Il est possible de restreindre le domaine de définition d'un ou plusieurs opérateurs, en indiquant une condition devant être remplie par ses opérandes pour que le terme résultant ait un sens (i.e. soit défini).

Par exemple, si on souhaite indiquer pour la spécification *ENS* que $i(E, x)$ n'a de sens que s'il n'y a pas déjà dans E un élément de même valeur que x, on écrira :

pre i(E,x) = ¬(x ∈ E)

De même, si $s(E, x)$ n'a de sens que s'il y a dans E un élément de même valeur que x, on écrira :

pre s(E,x) = x ∈ E

La spécification devient alors :

```
spec ENS1
  etend NAT1,BOOL1
  sorte Ens
  operations
    ∅    :            → Ens
    i    : Ens Nat → Ens        ajoute un élément à un ensemble
    _∈_  : Nat Ens →Bool
    card : Ens      → Nat
    s    : Ens Nat →Ens         supprime un élément d'un ensemble
  preconditions
    pre i(E,x) = ¬(x ∈ E)
    pre s(E,x) = x ∈ E
  axiomes x,y:Nat;E:Ens
    x∈∅ = faux
    x∈i(E,y) = x==y ∨ x∈E
    card(∅) = 0
    card(i(E,x)) = succ(card(E))
    s(i(E,x),y) = si x==y alors E sinon i(s(E,y),x) fsi
fspec
```

On peut remarquer que les axiomes ne prévoient pas les cas où on est en dehors du domaine de définition. Il n'est donc plus utile, pour $card(i(E, x))$, de tester si x appartient à E.

Pour toute spécification, on considérera que l'opérateur polymorphe Def est implicitement défini par :

— pour tout opérateur f, $Def(f(x_1, \ldots, x_n)) = Def(x_1) \wedge \ldots \wedge Def(x_n) \wedge \text{pre}f(x_1, \ldots, x_n)$

— pour les termes conditionnels, $Def(\text{si } x_1 \text{ alors } x_2 \text{ sinon } x_3 \text{ fsi}) = Def(x_1) \wedge ((x_1 \wedge Def(x_2)) \vee (\neg x_1 \wedge Def(x_3))$

Remarque

Un opérateur sans pré-condition est supposé implicitement muni d'une pré-condition toujours vraie.

2.7 Suffisante complétude et consistance

Une spécification est dite « suffisamment complète »si elle n'introduit pas de nouvelle classe de congruence dans les sortes prédéfinies. Par exemple, dans la spécification NAT_1, si on arrive à isoler des termes de sorte *Bool* qui ne sont congrus ni à *vrai*, ni à *faux*, la spécification n'est pas suffisamment complète.

Une spécification est dite « consistante »si elle ne réunit pas des classes de congruences de sorte prédéfinies. Par exemple, dans la spécification NAT_1, si on arrive à montrer que *vrai* est congru à *faux*, la spécification est inconsistante.

Malheureusement ces propriétés sont indécidables dans le cas général. Autrement dit, s'il est souvent possible de montrer qu'une spécification n'est pas suffisamment complète ou qu'elle est inconsistante en exhibant des exemples, il est souvent impossible de montrer qu'une spécification est consistante ou suffisamment complète.

Chapitre 3

Implémentation objet des types abstraits

Schéma d'instance (private), encapsulation
Constructeurs
Méthodes
Choix de l'API (Application Programming Interface) = choix des opérations du TAA
Utilisation de l'héritage, sous-typage (inclusion des protocoles) et héritage (du schéma d'instance et des méthodes).
Identité (==) et égalité (.equals())

3.1 Points

```
package example;
//* A point in the plane
public abstract class Point {
  //* Abscissa
  public abstract double x();
  //* Ordinate
  public abstract double y();
  //* Radius
  public abstract double r();
  //* Azimuth
  public abstract double theta();
  //* Distance from an other point p
  public double distanceFrom(Point p) {
    return Math.hypot(this.x() - p.x(), this.y() - p.y());
  }
}
```

```
package example;
import java.text.DecimalFormat;
import java.util.Locale;


public class CartesianPoint extends Point {
  private double x;
  private double y;

  public CartesianPoint(double x, double y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public double x() {
    return this.x;
  }
```

```
  @Override
  public double y() {
    return this.y;
  }

  @Override
  public double r() {
    return Math.hypot(this.x, this.y);
  }

  @Override
  public double theta() {
    return Math.atan2(this.y, this.x);
  }

  @Override
  public String toString() {
    // Use a DecimalFormat to print only two fractional digits.
    DecimalFormat fmt = new DecimalFormat("#.##");
    return "(" + fmt.format(this.x) + "," + fmt.format(this.y) +")";
  }

  public static void main(String[] args) {
    // Use ROOT locale so that numbers are displayed like Java literals
    Locale.setDefault(Locale.ROOT);
    DecimalFormat fmt = new DecimalFormat("#.##");
    Point p = new CartesianPoint(0, 1);
    System.out.println(
        "p = " + p.toString() + " = "
          + fmt.format(p.r()) + ":" + fmt.format(Math.toDegrees(p.theta()))
    );
  }

}
```

```
package example;
import java.text.DecimalFormat;
import java.util.Locale;


public class PolarPoint extends Point {
  private double r;
  private double theta;

  public PolarPoint(double r, double theta) {
    this.r = r;
    this.theta = theta;
  }

  @Override
  public double x() {
    return this.r * Math.cos(this.theta);
  }

  @Override
  public double y() {
    return this.r * Math.sin(this.theta);
  }

  @Override
  public double r() {
    return this.r;
  }

  @Override
```

```
  public double theta() {
    return this.theta;
  }

  @Override
  public String toString() {
    // Use a DecimalFormat to print only two fractional digits.
    DecimalFormat fmt = new DecimalFormat("#.##");
    return "(" + fmt.format(Math.toDegrees(this.theta)) + ":" + fmt.format(this.r) +")";
  }

  public static void main(String[] args) {
    // Use ROOT locale so that numbers are displayed like Java literals
    Locale.setDefault(Locale.ROOT);
    DecimalFormat fmt = new DecimalFormat("#.##");
    Point p = new PolarPoint(1, Math.toRadians(45));
    System.out.println(
        "p = (" + fmt.format(p.x()) + "," + fmt.format(p.y()) + ") = "
          + p.toString()
    );
  }

}
```

3.2 Dates

```
package example;

public abstract class Date implements Comparable<Date> {
  public abstract int year();
  public abstract int month();
  public abstract int day();
}
```

```
package example;

public class StructDate extends Date {
  private int year;
  private int month;
  private int day;

  public StructDate(int day, int month, int year) {
    assert((day >= 1) && (day <= 31) && (month >= 1) && (month <= 12) && (year > 0));
    this.year = year;
    this.month = month;
    this.day = day;
  }

  @Override
  public int year() {
    return this.year;
  }

  @Override
  public int month() {
    return this.month;
  }

  @Override
  public int day() {
    return this.day;
  }
```

```
  @Override
  public int compareTo(Date d) {
    if (this.year < d.year()) {
      return -1;
    } else if (this.year > d.year()) {
      return 1;
    } else {
      if (this.month < d.month()) {
        return -1;
      } else if (this.month > d.month()) {
        return 1;
      } else {
        if (this.day < d.day()) {
          return -1;
        } else if (this.day > d.day()) {
          return 1;
        } else {
          return 0;
        }
      }
    }
  }

  @Override
  public String toString() {
    return this.day + "/" + this.month + "/" + this.year;
  }

  public static void main(String[] args) {
    Date era = new StructDate(1,1,1970);
    Date d = new StructDate(13,2,2012);

    System.out.print(d);
    int t = d.compareTo(era);
    if (t < 0) {
      System.out.print(" is before ");
    } else if (t > 0) {
      System.out.print(" is after ");
    } else {
      System.out.print(" is same as ");
    }
    System.out.println(era);

    d = new StructDate(45,-1,13);
    System.out.println(d);
  }

}
```

```
package example;

public class FieldDate extends Date {
  private int value;

  public FieldDate(int day, int month, int year) {
    assert((day >= 1) && (day <= 31) && (month >= 1) && (month <= 12) && (year >= 1));
    this.value = computeValue(day, month, year);
  }

  private static int computeValue(int day, int month, int year) {
//    return 512 * year + 32 * month + day;
    return year << 9 | month << 5 | day;
  }
```

```
  @Override
  public int year() {
//    return this.value / 512;
    return this.value >> 9;
  }

  @Override
  public int month() {
//    return (this.value % 512) / 32;
    return (this.value >> 5) & 0xF;
  }

  @Override
  public int day() {
//    return this.value % 32;
    return this.value & 0x1F;
  }

  @Override
  public int compareTo(Date d) {
    return this.value - computeValue(d.day(), d.month(), d.year());
  }

  @Override
  public String toString() {
    return this.day() + "/" + this.month() + "/" + this.year();
  }

  public static void main(String[] args) {
    Date era = new FieldDate(1,1,1970);
    Date d = new FieldDate(13,2,2012);

    System.out.print(d);
    int t = d.compareTo(era);
    if (t < 0) {
      System.out.print(" is before ");
    } else if (t > 0) {
      System.out.print(" is after ");
    } else {
      System.out.print(" is same as ");
    }
    System.out.println(era);
  }

}
```

Exceptions (domaine de définition des opérations)

```
=> throw new RuntimeException("message d'erreur");
```

Assertions (axiomes, pré et post conditions)

```
=> assert (a > 0) && (b > 0) : "Both integers must be positive" ;
```

Chapitre 4

Types élémentaires

4.1 Bag

Un *Bag* est une collection très simple, à laquelle on ne peut qu'ajouter des éléments.

spec BAG
  etends BOOL, NAT, ELEMENT
  sorte Bag
  operations
    emptyBag : → Bag
    add : Bag Element → Bag
    contains : Bag Element → Bool
    size : Bag → Nat

  axiomes : b:Bag ; e, e_1, e_2:Element
    contains(emptyBag, e) = faux
    contains(add(b, e_1), e_2) = **si** (e_1 = e_2) **alors** vrai **sinon** contains(b, e_2) **fsi**
    size(emptyBag) = 0
    size(add(b,e)) = succ(size(b))
fspec

```
package basictype;
/**
 * A bag is just a collection of data. You can add items to it and get its size.
 * You can also get an iterator on its elements.
 *
 * @author Frederic Boulanger frederic.boulanger@supelec.fr
 *
 * @param <T> the type of elements in the bag
 *
 * @spec
 * emptyBag : -> Bag
 * add : Bag, Item -> Bag
 * contains : Bag, Item -> Bool
 * size : Bag -> Nat
 *
 * contains(emptyBag, e) = false
 * contains(add(b, e_1), e_2) = if (e_1 = e_2) then true else contains(b, e_2) endif
 * size(Bag) = 0
 * size(add(b,i)) = succ(size(b))
 *
 */
public abstract class Bag<T> implements Iterable<T> {
  // operations from the specification (excepted
  // constants, which will be handled by class constructors)
  public abstract void add(T item);
  public abstract int size();
  // Simple default implementation of contains.
  // Subclasses where a more efficient implementation is possible may override this
  public boolean contains(T item) {
```

```
    for (T element : this) {
      if (item.equals(element)) {
        return true;
      }
    }
    return false;
  }
  // operations linked to the Java framework
  @Override
  public String toString() {
    // Use a StringBuffer instead of String concatenation for efficiency
    StringBuffer result = new StringBuffer("[");
    boolean first = true;
    for (T item : this) {
      // prepend a comma ans a space before all items but the first one
      if (first) {
        first = false;
      } else {
        result.append(",␣");
      }
      result.append(item.toString());
    }
    result.append("]");
    return result.toString();
  }
}
```

4.2 Queue

Une *Queue* est une file d'attente (FIFO, *First In, First Out*)

spec QUEUE
  etends NAT, ELEMENT
  sorte Queue
  operations
    emptyQueue : → Queue
    enqueue : Queue Element → Queue
    dequeue : Queue → Queue
    head : Queue → Element
    size : Queue → Nat

  preconditions q:Queue
    pre first (q) = size(q) > 0
    pre dequeue(q) = size(q) > 0

  axiomes : q:Queue ; e, e_1, e_2:Element
    dequeue(enqueue(emptyQueue, e)) = emptyQueue
    dequeue(enqueue(enqueue(q, e_1), e_2)) = enqueue(dequeue(enqueue(q, e_1)), e_2)
    head(enqueue(emptyQueue, e)) = e
    head(enqueue(enqueue(q, e_1), e_2)) = head(enqueue(q, e_1))
    size (emptyQueue) = 0
    size (enqueue(q,e)) = succ(size(q))
fspec

```
package basictype;
/**
 * A queue is an ordered collection of items.
 * You can add items at the end of the queue,
 * remove items from the head of the queue, and get its size.
 *
 * You can also get an iterator on its elements.
 *
 * @author Frederic Boulanger frederic.boulanger@supelec.fr
 *
```

```
 * @param <T> the type of elements in the queue
 *
 * In the specification, we split the dequeue method into a head()
 * operation which yields the first item in the queue, and a dequeue
 * operation which removes it from the queue
 *
 * @spec
 * emptyQueue : -> Queue
 * enqueue : Queue, Item -> Queue
 * dequeue : Queue -> Queue
 * head : Queue -> Item
 * size : Queue -> Nat
 *
 * pre(dequeue(q)) = size(q) > 0
 * pre(head(q)) = size(q) > 0
 *
 * dequeue(enqueue(emptyQueue, e)) = emptyQueue
 * dequeue(enqueue(enqueue(q, e_1), e_2) = enqueue(dequeue(enqueue(q, e_1)), e_2)
 * head(enqueue(emptyQueue, e)) = e
 * head(enqueue(enqueue(q, e_1), e_2) = head(enqueue(q, e_1))
 * size(Queue) = 0
 * size(enqueue(q,i)) = succ(size(q))
 *
 */
public abstract class Queue<T> implements Iterable<T> {
  public abstract void enqueue(T item);
  public abstract T dequeue();
  public abstract int size();
  @Override
  public String toString() {
    StringBuffer buf = new StringBuffer("<");
    boolean first = true;
    for (T item : this) {
      if (first) {
        first = false;
      } else {
        buf.append(", ");
      }
      buf.append(item.toString());
    }
    buf.append("<");
    return buf.toString();
  }
}
```

4.3 Stack

Une *Stack* est une pile (LIFO, *Last In, First Out*)

spec STACK
  etends NAT, ELEMENT
  sorte Stack
  operations
    emptyStack : → Stack
    push : Stack Element → Stack
    pop : Stack → Stack
    top : Stack → Element
    size : Stack → Nat

  preconditions s:Stack
    pre top(s) = size(s) > 0
    pre pop(s) = size(s) > 0

  axiomes : s:Stack ; e:Element

pop(push(s, e)) = s
top(push(s, e)) = e
size(emptyStack) = 0
size(push(s,e)) = succ(size(s))

fspec

```
package basictype;
/**
 * A stack is an ordered collection of items.
 * You can push items onto the stack,
 * pop items from the top of the stack, and get its size.
 *
 * You can also get an iterator on its elements.
 *
 * @author Frederic Boulanger frederic.boulanger@supelec.fr
 *
 * @param <T> the type of elements in the stack
 *
 * In the specification, we split the pop method into a top()
 * operation which yields the top of the stack, and a pop()
 * operation which removes it from the stack.
 *
 * @spec
 * Stack : -> Stack
 * push : Stack, Item -> Stack
 * pop : Stack -> Stack
 * top : Stack -> Item
 * size : Stack -> Nat
 *
 * pre(pop(s)) = size(s) > 0
 * pre(top(s)) = size(s) > 0
 *
 * size(Stack) = 0
 * size(push(s,i)) = succ(size(s))
 * pop(push(s,i)) = s
 * top(push(s,i)) = i
 *
 */
public abstract class Stack<T> implements Iterable<T> {
  public abstract void push(T item);
  public abstract T pop();
  public abstract T top();
  public abstract int size();
  @Override
  public String toString() {
    StringBuffer buf = new StringBuffer(":");
    boolean first = true;
    for (T item : this) {
      if (first) {
        first = false;
      } else {
        buf.append(",␣");
      }
      buf.append(item.toString());
    }
    buf.append("|");
    return buf.toString();
  }
}
```

4.3.1 Algorithme de Dijkstra pour analyser une expression

```
package basictype;
```

```
import java.util.HashMap;
import java.util.Map;

public class DijkstraEval {
  // Give priority to / over * over -  over +
  private static final Map<String, Integer> priority;
  static {
    priority = new HashMap<String, Integer>();
    priority.put("(", -1); // opening parenthesis are processed only when a closing parenthesis is
    priority.put("+", 0);
    priority.put("-", 1);
    priority.put("*", 2);
    priority.put("/", 3);
  }

  //@ Process an operator: get its arguments and push the result on the operands stack.
  private static void processOperator(Stack<Double> values, String op) {
    Double res = values.pop();
    switch (op) {
      case "+" :
  res = values.pop() + res;
  break;
      case "-" :
  res = values.pop() - res;
  break;
      case "*" :
  res = values.pop() * res;
  break;
      case "/" :
  res = values.pop() / res;
  break;
      default :
  throw new RuntimeException("Unknown operator \""+op+"\"");
    }
    values.push(res);
  }

  // With args = ( 2.1 + 3.2 - 1.5 ) * 5.7
  // we get (2.1+3.2-1.5)*5.7 = 21.660000000000004
  public static void main(String[] args) {
    Stack<String> operators = new LinkedStack<String>();
    Stack<Double> operands = new LinkedStack<Double>();
    for(String s : args) {
      System.out.print(s);
      switch (s) {
  case "(" :
    // Push an opening parenthesis as a marker in the operators stack
    operators.push(s);
    break;
  case "+" :
  case "-" :
  case "*" :
  case "/" :
    // It is an operator, push it on the operators stack
    // but first, process any operator with a higher priority
    int my_priority = priority.get(s);
    while ((operators.size() > 0) && (priority.get(operators.top()) > my_priority)) {
      processOperator(operands, operators.pop());
    }
    operators.push(s);
    break;
  case ")" :
    // A closing parenthesis: pop operators until an opening parenthesis is found
    String op = operators.pop();
```

```
    while (!op.equals("(")) {
      processOperator(operands, op);
      op = operators.pop();
    }
    break;
  default :  // Not a parenthesis nor an operator => it's a number
    operands.push(Double.parseDouble(s));
      }
    }
    // At the end of the expression, process the operators to compute the value
    while (operators.size() > 0) {
      processOperator(operands, operators.pop());
    }
    if (operands.size() == 0) {
      // If the operands stack is empty, we did not compute anything
      System.out.println("No expression");
    } else if (operands.size() > 1) {
      // If the operands stack has more than one item, an operator is missing.
      System.out.println("Missing operator, or extra operand");
    } else {
      // Else, display the value of the expression
      System.out.println(" = " + operands.pop());
    }
  }
}
```

4.4 Implémentations par tableau

Accès, insertion, suppression
Problème du redimensionnement

```
package basictype;

import java.util.Iterator;

/**
 * A bag which uses an array for storing its elements.
 * The array is resized dynamically when it becomes
 * too small for holding the elements of the bag.
 *
 * @author Frederic Boulanger frederic.boulanger@supelec.fr
 *
 * @param <T> the type of elements in the bag
 */
public class ArrayBag<T> extends Bag<T> {
  private static final int initCapacity = 5;
  private T[] elements;
  private int size;

  //* Create an empty ArrayBag
  public ArrayBag() {
    this.size = 0;
    resize(initCapacity);
  }
  //* Add an item to the bag
  @Override
  public void add(T item) {
    if (this.size == this.elements.length) {
      grow();
    }
    this.elements[this.size++] = item;
  }
```

```
  //* Get the number of items in the bag
  @Override
  public int size() {
    return this.size;
  }

  //* Get an iterator on the bag
  @Override
  public Iterator<T> iterator() {
    return new ArrayBagIterator();
  }

  /**
   * An iterator on an ArrayBag
   */
  private class ArrayBagIterator implements Iterator <T> {
    private int currentIndex = 0;

    @Override
    public boolean hasNext() {
      return this.currentIndex < ArrayBag.this.size ;
    }

    @Override
    public T next() {
      return ArrayBag.this.elements[this.currentIndex++];
    }

    @Override
    public void remove() {
      // Don't do anything, removing items from a bag is not supported
    }
  }

  //* Enlarge the array for storing the elements of the bag
  private void grow() {
    resize(this.elements.length * 2);
  }

  //* Resize the array for storing the elements of the bag
  private void resize(int newCapacity) {
    // Create a new array with the new capacity
    // Required annotation because Java erases parameter
    // types at runtime and can't check the following cast.
    @SuppressWarnings("unchecked")
    T[] newElements = (T[]) new Object[newCapacity];
    // Copy the elements from the old array to the new one
    for (int i = 0; i < this.size; i++) {
      newElements[i] = this.elements[i];
    }
    // Use the new array as storage
    this.elements = newElements;
  }

  private static final String[] defaultArgs = {
    "un", "deux", "trois", "quatre", "cinq",
    "six", "sept", "huit", "neuf", "dix",
    "onze", "douze", "treize", "quatorze", "quinze"
  };
  /**
   * Test function
   */
  public static void main(String[] args) {
    String[] items = args;
    if (items.length == 0) { // No arguments -> use default sample of items
```

```
      items = defaultArgs;
    }
    Bag<String> b = new ArrayBag<String>();
    for(String item: items) {
      b.add(item);
    }

    System.out.println("# Iterate");
    for(String item : b) {
      System.out.println(item);
    }
  }

}
```

```
package basictype;

import java.util.Iterator;

/**
 * A Queue which uses a rotating buffer for storing its elements.
 * The array is resized dynamically when it becomes too small
 * for holding the elements of the queue.
 *
 * @author Frederic Boulanger frederic.boulanger@supelec.fr
 *
 * @param <T> the type of elements in the queue
 */
public class BufferQueue<T> extends Queue<T> {
  private static final int initialCapacity = 5;

  private T[] buffer;
  private int queue_index;
  private int dequeue_index;
  private int size;
  private int capacity;

  public BufferQueue() {
    this(initialCapacity);
  }

  public BufferQueue(int capacity) {
    this.size = 0;
    this.queue_index = 0;
    this.dequeue_index = 0;
    this.capacity = 0;
    resize(capacity);
  }

  @Override
  public void enqueue(T item) {
    if (this.size == this.capacity) {
      grow();
    }
    this.buffer[this.queue_index]= item;
    this.queue_index = (this.queue_index + 1) % this.capacity;
    this.size++;
  }

  @Override
  public T dequeue() {
    T result = null;
    if (this.size > 0) {
      result = this.buffer[this.dequeue_index];
      this.dequeue_index = (this.dequeue_index + 1) % this.capacity;
```

```
      this.size--;
      shrink();
      return result;
    } else {
      throw new RuntimeException("Error: dequeue on empty queue.");
    }
  }

  @Override
  public int size() {
    return this.size;
  }

  @Override
  public Iterator<T> iterator() {
    return new BufferQueueIterator();
  }

  private void grow() {
    resize(this.capacity * 2);
  }

  private void shrink() {
    if ((this.size >= initialCapacity) && (this.capacity >= 2 * this.size)) {
      resize(this.size);
    }
  }

  @SuppressWarnings("unchecked")
  private void resize(int newCapacity) {
    T[] newBuffer = (T[]) new Object[newCapacity];
    for (int i = 0; i < this.size; i++) {
      newBuffer[i] = this.buffer[(this.dequeue_index + i) % this.capacity];
    }
    this.buffer = newBuffer;
    this.capacity = newCapacity;
    this.dequeue_index = 0;
    this.queue_index = this.size;
  }

  private class BufferQueueIterator implements Iterator<T> {
    private int currentIndex = BufferQueue.this.dequeue_index;
    private int items_to_go = BufferQueue.this.size;

    @Override
    public boolean hasNext() {
      return this.items_to_go > 0;
    }

    @Override
    public T next() {
      T result = BufferQueue.this.buffer[this.currentIndex];
      this.currentIndex = (this.currentIndex + 1) % BufferQueue.this.capacity;
      this.items_to_go--;
      return result;
    }

    @Override
    public void remove() {
      // Don't do anything, removing items from a queue is for dequeue() only!
    }
  }


  private static final String[] defaultArgs = {
```

```
    "un", "deux", "trois", "quatre", "cinq",
    "six", "sept", "huit", "neuf", "dix",
    "onze", "douze", "treize", "quatorze", "quinze"
  };

  public static void main(String[] args) {
    String[] test = args;
    if (test.length == 0) {
      test = defaultArgs;
    }

    Queue<String> q = new BufferQueue<String>();
    for (int i = 0; i < 5; i++) {
      q.enqueue(test[i]);
    }
    System.out.println(q.dequeue());
    for (int i = 5; i < 10; i++) {
      q.enqueue(test[i]);
    }
    System.out.println(q.dequeue());
    System.out.println(q.dequeue());
    for (int i = 10; i < 15; i++) {
      q.enqueue(test[i]);
    }

    System.out.println("# Iterate");
    System.out.println(q);

    System.out.println("# Dequeue");
    while (q.size() > 0) {
      System.out.println(q.dequeue());
    }

    // Should throw an exception
    // q.dequeue();
  }

}
```

```
package basictype;

import java.util.Iterator;


/**
 * A stack which uses an array for storing its elements.
 * The array is resized dynamically when it becomes
 * too small for holding the elements of the stack.
 *
 * @author Frederic Boulanger frederic.boulanger@supelec.fr
 *
 * @param <T> the type of elements in the stack
 */
public class ArrayStack<T> extends Stack<T> {
  private static final int initCapacity = 5;
  private T[] elements;
  private int size;

  //* Create an empty stack
  public ArrayStack() {
    this.size = 0;
    resize(initCapacity);
  }

  //* Push an item onto the stack
```

```
@Override
public void push(T item) {
  if (this.size == this.elements.length) {
    grow();
  }
  this.elements[this.size++] = item;
}

@Override
public T top() {
  return this.elements[this.size - 1];
}

//* Remove the item at the top of the stack and return it
@Override
public T pop() {
  T result = null;
  if (this.size > 0) {
    result = this.elements[--this.size];
    this.elements[this.size]= null;
    shrink();
    return result;
  } else {
    throw new RuntimeException("Error: pop on empty stack.");
  }
}

//* Get the number of items on the stack
@Override
public int size() {
  return this.size;
}

//* Get an iterator on the stack
@Override
public Iterator<T> iterator() {
  return new ArrayStackIterator();
}

//* Private class of the iterators on this kind of stacks
private class ArrayStackIterator implements Iterator <T> {
  private int currentIndex = ArrayStack.this.size;

  @Override
  public boolean hasNext() {
    return this.currentIndex > 0 ;
  }

  @Override
  public T next() {
    return ArrayStack.this.elements[--this.currentIndex];
  }

  @Override
  public void remove() {
    // Don't do anything, removing items from a stack is for pop() only!
  }
}

//* Grow the array for storing the items on the stack
private void grow() {
  resize(this.elements.length * 2);
}

//* Shrink the array for storing the items if its is wasting too much memory
```

```
  private void shrink() {
    if ((this.size >= initCapacity) && (this.elements.length >= (2*this.size))) {
      resize(this.size);
    }
  }

  //* Resize the array for storing the items on the stack
  private void resize(int newCapacity) {
    // Create a new array with the new capacity
    @SuppressWarnings("unchecked")
    T[] newElements = (T[])new Object[newCapacity];
    // Copy the elements from the old array to the new one
    for (int i = 0; i < this.size; i++) {
      newElements[i] = this.elements[i];
    }
    // Use the new array as storage
    this.elements = newElements;
  }

  private static final String[] defaultArgs = {
    "un", "deux", "trois", "quatre", "cinq",
    "six", "sept", "huit", "neuf", "dix",
    "onze", "douze", "treize", "quatorze", "quinze"
  };
  /**
   * Test program
   */
  public static void main(String[] args) {
    String[] test = args;
    if (test.length == 0) {
      test = defaultArgs;
    }
    Stack<String> s = new ArrayStack<String>();
    for(String arg: test) {
      s.push(arg);
    }

    System.out.println("Taille = " + s.size());
    System.out.println("# Iterate");
    System.out.println(s);

    System.out.println("# Pop");
    while (s.size() > 0) {
      System.out.println(s.pop());
    }
  }

}
```

4.5 Implémentations par chaînage

Accès, insertion, suppression
Problème du redimensionnement
Complexité quadratique du parcours

```
package basictype;
import java.util.Iterator;

public class LinkedBag<T> extends Bag<T> {
  private class Link {
    T element;
    Link next;
```

```
    Link(T element, Link next) {
      this.element = element;
      this.next = next;
    }
  }

  private Link last_added;
  private int size;

  public LinkedBag() {
    this.size = 0;
    this.last_added = null;
  }

  @Override
  public void add(T item) {
    this.last_added = new Link(item, this.last_added);
    this.size++;
  }

  @Override
  public int size() {
    return this.size;
  }

  @Override
  public Iterator<T> iterator() {
    return new LinkedBagIterator();
  }

  private class LinkedBagIterator implements Iterator <T> {
    private Link currentLink = LinkedBag.this.last_added;

    @Override
    public boolean hasNext() {
      return this.currentLink != null ;
    }

    @Override
    public T next() {
      T result = this.currentLink.element;
      this.currentLink = this.currentLink.next;
      return result;
    }

    @Override
    public void remove() {
      // Don't do anything, removing items from a bag is not allowed!
    }
  }

  private static final String[] defaultArgs = {
    "un", "deux", "trois", "quatre", "cinq",
    "six", "sept", "huit", "neuf", "dix",
    "onze", "douze", "treize", "quatorze", "quinze"
  };

  public static void main(String[] args) {
    String[] items = args;
    if (items.length == 0) { // No arguments -> use default sample of items
      items = defaultArgs;
    }
    Bag<String> b = new LinkedBag<String>();
    for(String item: items) {
      b.add(item);
```

```
    }

    System.out.println("# Iterate");
    for(String item : b) {
      System.out.println(item);
    }
  }

}
```

```
package basictype;
import java.util.Iterator;

public class LinkedQueue<T> extends Queue<T> {
  private class Link {
    T element;
    Link next;

    Link(T element, Link next) {
      this.element = element;
      this.next = next;
    }
  }

  private Link first;
  private Link last;
  private int size;

  public LinkedQueue() {
    this.size = 0;
    this.first = null;
    this.last = null;
  }

  @Override
  public void enqueue(T item) {
    Link previousLast = this.last;
    this.last = new Link(item, null);
    if (previousLast == null) {
      this.first = this.last;
    } else {
      previousLast.next = this.last;
    }
    this.size++;
  }

  @Override
  public T dequeue() {
    T result = null;
    if (this.first != null) {
      result = this.first.element;
      this.first = this.first.next;
      if (this.first == null) {
        this.last = null;
      }
      this.size--;
      return result;
    } else {
      throw new RuntimeException("Error: dequeue on empty queue.");
    }
  }

  @Override
  public int size() {
    return this.size;
```

```
  }

  @Override
  public Iterator<T> iterator() {
    return new LinkedQueueIterator();
  }

  private class LinkedQueueIterator implements Iterator <T> {
    private Link currentLink = LinkedQueue.this.first;

    @Override
    public boolean hasNext() {
      return this.currentLink != null ;
    }

    @Override
    public T next() {
      T result = this.currentLink.element;
      this.currentLink = this.currentLink.next;
      return result;
    }

    @Override
    public void remove() {
      // Don't do anything, removing items from a queue is for dequeue() only!
    }
  }


  private static final String[] defaultArgs = {
    "un", "deux", "trois", "quatre", "cinq",
    "six", "sept", "huit", "neuf", "dix",
    "onze", "douze", "treize", "quatorze", "quinze"
  };

  public static void main(String[] args) {
    String[] items = args;
    if (items.length == 0) { // No arguments -> use default sample of items
      items = defaultArgs;
    }
    Queue<String> q = new LinkedQueue<String>();
    for(String arg: items) {
      q.enqueue(arg);
    }

    System.out.println("# Iterate");
    System.out.println(q);

    System.out.println("# Dequeue");
    while (q.size() > 0) {
      System.out.println(q.dequeue());
    }
  }

}
```

```
package basictype;
import java.util.Iterator;


public class LinkedStack<T> extends Stack<T> {
  private class Link {
    T element;
    Link next;
```

```
    Link(T element, Link next) {
      this.element = element;
      this.next = next;
    }
  }

  private Link top;
  private int size;

  public LinkedStack() {
    this.size = 0;
    this.top = null;
  }

  @Override
  public void push(T item) {
    this.top = new Link(item, this.top);
    this.size++;
  }

  @Override
  public T top() {
    return this.top.element;
  }

  @Override
  public T pop() {
    T result = null;
    if (this.top != null) {
      result = this.top.element;
      this.top = this.top.next;
      this.size--;
      return result;
    } else {
      throw new RuntimeException("Error: pop on empty stack.");
    }
  }

  @Override
  public int size() {
    return this.size;
  }

  @Override
  public Iterator<T> iterator() {
    return new LinkedStackIterator();
  }

  private class LinkedStackIterator implements Iterator <T> {
    private Link currentLink = LinkedStack.this.top;

    @Override
    public boolean hasNext() {
      return this.currentLink != null ;
    }

    @Override
    public T next() {
      T result = this.currentLink.element;
      this.currentLink = this.currentLink.next;
      return result;
    }

    @Override
    public void remove() {
```

```
      // Don't do anything, removing items from a stack is for pop() only!
    }
  }


  private static final String[] defaultArgs = {
    "un", "deux", "trois", "quatre", "cinq",
    "six", "sept", "huit", "neuf", "dix",
    "onze", "douze", "treize", "quatorze", "quinze"
  };

  public static void main(String[] args) {
    String[] test = args;
    if (test.length == 0) {
      test = defaultArgs;
    }
    Stack<String> s = new LinkedStack<String>();
    for(String arg: test) {
      s.push(arg);
    }

    System.out.println("# Iterate");
    System.out.println(s);

    System.out.println("# Pop");
    while (s.size() > 0) {
      System.out.println(s.pop());
    }
  }

}
```

4.6 Itérateurs

Notion de collection
Parcours d'une collection
Complexité linéaire du parcours quelque soit la représentation.

Chapitre 5

Complexité des algorithmes

5.1 Objectifs

On a souvent le choix entre plusieurs algorithmes lorsque l'on cherche à résoudre un problème donné.

Considérons, à titre d'exemple, le problème consistant à classer les éléments d'un tableau en ordre croissant. Il existe de nombreux algorithmes permettant de le résoudre. On peut citer, parmi les plus connus :

— le tri par sélection,
— le tri par insertion,
— le tri par tas,
— le tri rapide,
— le tri par fusion.

Cette liste n'est bien sûr pas exhaustive.

Il convient donc, face à une situation donnée, de choisir le bon algorithme de classement. Les critères de choix sont principalement de deux natures :

— la rapidité d'exécution du programme résultant,
— l'encombrement en mémoire de ce programme (plus précisément la taille des données temporaires qu'il utilise).

Dans le cadre de ce document, nous nous intéresserons uniquement au critère de rapidité d'exécution des programmes.

Ce critère de rapidité mérite cependant d'être précisé. Il est en effet souvent impossible d'affirmer qu'un algorithme est meilleur qu'un autre dans toutes les situations. La qualité d'un algorithme dépend de paramètres propres au problème traité qu'il importe de préciser pour effectuer un choix judicieux. On peut citer :

— La taille des données manipulées par l'algorithme
  Dans le cas d'un algorithme de classement de tableau, il s'agit du nombre d'éléments du tableau. D'une manière générale, on peut dire que la différence de rapidité entre plusieurs algorithmes permettant de résoudre un problème n'est sensible que lorsque les données manipulées sont de taille importante.
— La nature et l'organisation des données
  Les données à manipuler dans un problème ne sont pas toujours quelconques. On peut, par exemple, être amené à choisir un algorithme pour classer un tableau que l'on sait parfaitement être peu désordonné (presque classé). Le meilleur algorithme n'est alors sans doute pas celui qui va le plus vite dans un cas « moyen ».
— Des contraintes de temps maximal d'exécution.
  Dans certaines applications (typiquement les applications de contrôle de processus physiques critiques dans le domaine des transports, de la santé ou du nucléaire), le temps d'exécution ne doit pas dépasser une limite donnée. On peut alors être amené à proscrire un algorithme très bon dans la majorité des cas, mais très mauvais dans certains cas particuliers.

Il importe donc de posséder des outils permettant d'évaluer le temps d'exécution d'un algorithme, avant même d'écrire le programme correspondant.

Cette évaluation, effectuée pendant la phase d'analyse du problème, peut permettre d'éviter de s'engager dans des impasses et de gagner un temps considérable pendant la phase de réalisation d'un projet.

5.2 Complexité en temps

5.2.1 Définitions et exemples

Nous allons donc tenter d'évaluer le temps d'exécution d'un programme utilisant un algorithme donné. Une évaluation exacte de ce temps est totalement illusoire, car elle dépend de paramètres trop nombreux et trop difficiles à maîtriser. Parmi ces paramètres, nous pouvons citer :

— le langage utilisé pour coder l'algorithme.
— le programme lui-même : on désire en effet évaluer un algorithme avant d'écrire le programme correspondant. Il n'est pas possible de connaître le temps exact d'exécution d'un programme avant qu'il soit écrit.
— le compilateur utilisé : le temps d'exécution exact d'un programme ne dépend pas que de ce programme, mais aussi de la capacité du compilateur (le programme qui traduit le texte du programme en des instructions compréhensibles par l'ordinateur) à générer du code efficace.
— l'ordinateur sur lequel le programme va s'exécuter (en particulier de sa rapidité).

On désire en fait obtenir une valeur caractéristique du temps d'exécution du programme qui ne dépende que de l'algorithme utilisé, et pas de l'implémentation de ce dernier.

L'objectif recherché dans cette approche est de pouvoir comparer plusieurs algorithmes permettant de résoudre un même problème, et ceci en particulier lorsque la taille ou le nombre de données manipulées devient grand.

En effet, pour un problème donné, bien des algorithmes se comportent de manière similaire pour des données de petite dimension. Le temps d'exécution du programme correspondant est suffisamment faible pour qu'il soit relativement inutile de se préoccuper de l'efficacité de l'algorithme utilisé (par exemple quelques millisecondes).

Par contre, lorsque la taille des données manipulées devient importante, il devient capital de choisir un algorithme qui peut donner un temps d'exécution de quelques secondes, plutôt qu'un autre qui le ferait passer à quelques heures ou même quelques jours.

Dans certains cas, on cherchera plutôt à évaluer le temps d'exécution en fonction de la valeur des données. Par exemple, pour le problème consistant à élever un nombre k à une puissance entière n, la représentation informatique des données manipulées est de taille fixe. La donnée intéressante dans ce problème est l'évolution du temps d'exécution du programme quand n varie. Dans la suite de ce chapitre, nous parlerons toujours de « paramètre reflétant la taille des données », sachant que dans certains cas particuliers ce paramètre sera plutôt associé à la valeur de la donnée.

Nous allons donc chercher, pour réaliser cette mesure, une fonction dépendant de la taille des données.

Pour réaliser l'analyse d'un algorithme, nous allons mettre en évidence :

— Une opération élémentaire, choisie de manière à ce que le temps d'exécution du programme soit toujours asymptotiquement proportionnel, quand la taille des données tend vers l'infini, au nombre de fois que l'on exécute cette opération.
— Un paramètre reflétant la taille des données.

Définition 7 (complexité d'un algorithme) *Soit A un algorithme et d une donnée d'entrée de A.*

On appelle complexité de A pour d, et on note $C_A(d)$ le nombre de fois que l'opération élémentaire est exécutée lors de l'exécution de A sur d.

Nous allons illustrer ces notions sur un exemple : le tri par sélection.

Les algorithmes de tri ont tous un objectif commun : étant donné un ensemble sur lequel on dispose d'une relation d'ordre total que l'on nommera par la suite $\geq$, classer un tableau dont les éléments appartiennent à cet ensemble en ordre croissant (ou décroissant).

La méthode employée par le tri par sélection se fait en *tailleTab* itérations, où *tailleTab* est le nombre d'éléments du tableau à classer. Lors de la ^{e}i itération, on place, classés en ordre croissant, les i plus petits éléments du tableau dans les i premières positions de ce tableau. Au début de cette itération, les $i-1$ plus petits éléments du tableau sont donc classés et occupent les $i-1$ premières positions du tableau. Il reste donc à rechercher le plus petit élément des $tailleTab - i$ éléments restant, et à l'échanger avec le ^{e}i élément.

L'algorithme de classement du tableau *tab* de taille *tailleTab* par sélection peut donc s'écrire, en supposant que les indices du tableau varient entre 1 et *tailleTab* :

```
1  pour i variant de 1 à tailleTab
2    indiceDuPlusPetit ←i
3    pour j variant de i+1 à tailleTab
4      si tab[j] < tab[indiceDuPlusPetit]
5        indiceDuPlusPetit←j
6      fin si
7    fin pour
```

```
8    echanger tab[i] et tab[indiceDuPlusPetit]
9   fin pour
```

Exemple 5.1 – Algorithme du tri par sélection

Le paramètre reflétant la taille des données est ici le nombre d'éléments du tableau, c'est à dire *tailleTab*.

D'une manière générale, ce paramètre doit être choisi de manière à représenter quelque chose de significatif pour le problème traité. Si ce dernier est un problème concernant des matrices carrées, on pourra alors choisir la dimension d'une matrice ou encore son nombre d'éléments en fonction de ce qui est effectivement significatif pour la sémantique du problème. Il est clair que la valeur de la complexité dépendra de ce choix. Il est donc essentiel de préciser clairement le paramètre choisi avant de donner une complexité. Il va de soi que l'on ne pourra comparer la complexité de deux algorithmes résolvant le même problème que si on a choisi le même paramètre dans les deux calculs de complexité.

Le choix de l'opération élémentaire est moins évident. On pourrait être tenté de choisir celle qui prend le plus de temps à s'exécuter, en pensant que c'est elle qui va prédominer sur le temps d'exécution total du programme. S'il est plus long d'échanger deux éléments du tableau que de les comparer, ce qui est probablement le cas, un individu peu attentif déciderait alors que cet échange est l'opération élémentaire.

Cependant, si on dénombre le nombre d'échanges et le nombre de comparaisons d'éléments du tableau effectuées pendant le tri, on obtient :

Nombre d'échanges : $tailleTab$

Nombre de comparaisons : $(tailleTab \times (tailleTab - 1))/2$

Si te est la durée d'une opération d'échange et tc celle d'une opération de comparaison, le temps passé à faire des comparaisons sera supérieur à celui passé à faire des échanges pour toute valeur de $tailleTab$ supérieure à $2(te/tc) + 1$.

A titre d'exemple, si $te = 10 \times tc$, l'opération de comparaison devient prédominante dès que $tailleTab$ dépasse 21.

Ce simple exemple montre bien qu'il importe de choisir, non pas l'opération la plus longue, mais celle qui est effectuée le plus souvent. Dans le cas d'un algorithme itératif, ce sera souvent une opération située dans la boucle la plus interne.

La complexité de notre algorithme de tri par sélection, en ayant choisi $tailleTab$ comme paramètre reflétant la taille des données et la comparaison comme opération élémentaire, est donc de $(tailleTab \times (tailleTab - 1))/2$. Il est ici possible d'exprimer cette complexité en fonction du paramètre reflétant la taille des données car elle est identique pour toutes les données de même taille. Dans ce calcul, nous avons négligé certaines opérations car elles s'exécutent moins souvent que la comparaison. Il aurait été possible de les prendre en compte, mais l'ordre de grandeur du résultat, qui est en fait le seul résultat intéressant, n'aurait pas été modifié.

Cependant, pour que le calcul de complexité possède un intérêt, il est impératif que la durée d'exécution de l'opération élémentaire choisie soit indépendante du paramètre reflétant la taille des données. C'est effectivement le cas des deux opérations envisagées pour l'algorithme précédent. Imaginons cependant un langage de programmation qui offre une instruction permettant de trouver l'indice du plus petit élément d'un tableau. Il ne faudrait pas alors choisir cette instruction comme opération élémentaire, et affirmer que la complexité de l'algorithme est égale à $tailleTab$, car le temps d'exécution de cette instruction ne peut que dépendre de la taille du tableau.

5.2.2 Complexité dans le cas moyen, le meilleur et le pire

Il nous a été possible d'exprimer la complexité de l'algorithme de tri par sélection uniquement en fonction de la taille des données, sans avoir à se préoccuper de la valeur de ces données, c'est à dire de la valeur des éléments contenus dans le tableau. Le temps d'exécution d'un programme de tri par sélection ne dépend effectivement que de la taille du tableau à classer.

Cependant, pour la majorité des algorithmes, le temps d'exécution dépend aussi de la valeur ou de l'organisation des données.

Définition 8 (complexité dans le pire des cas) *Soit A un algorithme et $D(n)$ l'ensemble de toutes les données d'entrée possibles de A de taille n.*

On appelle complexité de A dans le pire des cas et on note $CA_{pire}(n)$ la complexité de cet algorithme dans le cas le plus défavorable, autrement dit, la valeur maximale de cette complexité pour tous les éléments de $D(n)$:

$$CA_{pire}(n) = \max_{d \in D(n)} CA(d)$$

La complexité dans le pire des cas est particulièrement importante, car elle permet d'évaluer les performances d'un algorithme quand les circonstances sont défavorables. Dans certains types d'applications, par exemple pour

la commande de dispositifs critiques (contrôle d'attitude d'un avion, freinage d'un train ou flux de neutrons dans une centrale nucléaire), il existe des contraintes impératives imposant qu'une action soit réalisée dans un délai maximum. On peut être alors amené à rejeter un algorithme qui serait très rapide dans la majorité des cas, mais très lent dans certains cas particuliers.

Définition 9 (complexité dans le meilleur des cas) *Soit A un algorithme et $D(n)$ l'ensemble de toutes les données d'entrée possibles de A de taille n.*

On appelle complexité de A dans le meilleur des cas et on note $CA_{meilleur}(n)$ la complexité de cet algorithme dans le cas le plus favorable, autrement dit, la valeur minimale de cette complexité pour tous les éléments de $D(n)$.

$CA_{meilleur}(n) = \min_{d \in D(n)} CA(d)$

La complexité dans le meilleur des cas nous permet d'isoler des situations dans lesquelles un algorithme est particulièrement rapide. Si, dans le problème à traiter, on se trouve dans ces situations, on peut alors être amené à choisir un algorithme qui n'est pas très rapide dans le cas moyen, mais excellent dans les situations auxquelles on est confronté.

Par exemple, l'algorithme de tri par insertion que nous allons étudier par la suite se comporte mieux que beaucoup d'autres algorithmes de classement lorsque le tableau à classer est déjà « presque »classé. Si on se trouve dans cette situation (et ce sera souvent le cas), on aura alors sans doute intérêt à choisir cet algorithme.

Définition 10 (complexité en moyenne) *Soit A un algorithme et $D(n)$ l'ensemble de toutes les données d'entrée possibles de A de taille n. On appelle complexité de A en moyenne et on note $CA_{moyen}(n)$ la complexité de cet algorithme dans le cas moyen, autrement dit, la valeur moyenne de cette complexité pour tous les éléments de $D(n)$.*

$CA_{moyen}(n) = \frac{\sum_{d \in D(n)} CA(d)}{card(D(n))}$

On remarquera que l'expression précédente n'est valable que si $D(n)$ est un ensemble fini. Un ordinateur étant un dispositif fini, ce sera toujours le cas dans les problèmes traités. La complexité en moyenne est l'outil de mesure qu'il faudra utiliser lorsque la configuration et la valeur des données peut être quelconque, et qu'il n'existe pas de contrainte sur le temps maximum d'exécution du programme.

Considérons à titre d'exemple l'algorithme de classement connu sous le nom de tri par insertion.

Le tri par insertion se fait lui aussi en $tailleTab$ itérations, où $tailleTab$ est le nombre d'éléments du tableau à classer. Au début de la i^{e} itération, les $i-1$ premiers éléments du tableau sont classés en ordre croissant, mais ce ne sont pas nécessairement les $i-1$ plus petits éléments du tableau. On place alors le i^{e} élément du tableau à sa place dans ce sous-tableau, afin d'obtenir un sous-tableau classé de i éléments. Pour ce faire, tous les éléments du sous-tableau plus grands que l'élément à placer sont décalés d'une position vers la fin du tableau, et ce dernier est mis dans le trou ainsi laissé. On constate donc qu'à chaque itération, les i premiers éléments ne sont pas les i plus petits du tableau original (comme c'est le cas pour le tri par sélection) mais ceux qui occupaient effectivement les i premières positions avant le début du classement.

Cette méthode est utilisée dans l'algorithme de l'exemple 5.2.

On peut ici choisir indifféremment la copie ou la comparaison d'éléments comme opération élémentaire, car les nombres de fois où elles sont exécutées ne diffèrent que d'une unité, ce qui ne change pas l'ordre de grandeur de la complexité. Nous verrons par la suite que seul cet ordre de grandeur présente en réalité un intérêt. Choisissons la copie d'éléments.

Le paramètre reflétant la taille des données est toujours le nombre d'éléments du tableau.

On constate ici que la complexité de cet algorithme dépend de l'ordre initial des éléments dans le tableau.

Si l'élément à déplacer est plus petit que tous ses prédécesseurs, la boucle *tant que* va s'exécuter $i-1$ fois, entraînant donc $i-1$ copies d'éléments. Si on ajoute la copie réalisée au début de la boucle *pour* et celle réalisée à la fin, cette itération de cette boucle réalise $i+1$ copies. Si pour toutes les itérations de la boucle *pour* l'élément à déplacer est plus petit que ceux qui le précèdent, le nombre de copies est donc égal à la somme des entiers compris entre 3 et $tailleTab+1$, c'est à dire $((tailleTab+1)(tailleTab+2))/2-3$. Il s'agit de la complexité dans le pire des cas de l'algorithme de tri par insertion. Cette situation se produit en particulier lorsque le tableau est initialement classé en ordre décroissant.

$C_{pire}(tailleTab) = ((tailleTab+1)(tailleTab+2))/2-3$

A l'opposé, si le tableau est initialement classé en ordre croissant, on n'exécutera jamais le corps de la boucle *tant que*, chaque itération de la boucle *pour* faisant donc 2 copies. La complexité dans le meilleur des cas de l'algorithme est donc de $(tailleTab-1) \times 2$

$C_{meilleur}(tailleTab) = (tailleTab-1) \times 2$

```
1  pour i variant de 2 à tailleTab
2    elementADeplacer ←tab[i]
3    j ←i − 1
4    tant que j ≥1 et que tab[j] > elementADeplacer
5      tab[j+1] ←tab[j]
6      j ←j−1
7    fin tant que
8    tab[j+1] ←elementADeplacer
9  fin pour
```

Exemple 5.2 – Algorithme du tri par insertion

Nous allons maintenant étudier la complexité en moyenne de cet algorithme.

A chaque itération de la boucle *pour*, on insère un élément dans le sous-tableau de *tab* composé des éléments dont les indices sont compris entre 1 et $i-1$. L'insertion se fait à une position k comprise entre 1 et i, entraînant $i-k$ copies d'éléments dans la boucle *tant que*. L'itération de la boucle pour réalise donc $i-k+2$ copies.

Intuitivement, on peut dire que le nombre de configurations du sous-tableau entraînant une insertion à l'indice k est identique pour toute valeur de k (cela peut se démontrer rigoureusement). Si on nomme N ce nombre de configurations, le nombre moyens de copies pour une itération de la boucle *pour* est donc égal à :

$$\frac{\sum_{k=1}^{i} N \times (i-k+2)}{N \times i} = \frac{\sum_{k=1}^{i} i-k+2}{i} = \frac{i+3}{2}$$

Le nombre de copies réalisées en moyenne pendant le tri, et donc sa complexité en moyenne, est de :

$$C_{moyen}(tailleTab) = \sum_{i=2}^{tailleTab} \frac{i+3}{2} = \frac{tailleTab^2}{4} + \frac{7 \times tailleTab}{4} - 2$$

5.2.3 Ordres de grandeur

Un des objectifs du calcul de complexité d'un algorithme est de permettre la comparaison de plusieurs algorithmes résolvant le même problème. Nous avons pu constater que les calculs menés ne permettent pas de déterminer de manière exacte les temps d'exécution des programmes. Seule une opération élémentaire est prise en compte, et cette opération peut être différente pour deux algorithmes résolvant le même problème. Clairement, les facteurs constants additifs ou même multiplicatifs ne doivent pas être pris en compte lors de la comparaison des performances de plusieurs algorithmes. Par contre, l'ordre de grandeur de la complexité va nous donner des indications très intéressantes.

Considérons à nouveau les problèmes de classement de tableau. Les deux algorithmes étudiés précédemment possèdent une complexité en moyenne de la forme $an^2 + bn + c$, où n est le nombre d'éléments du tableau à classer. Nous verrons par ailleurs un algorithme, connu sous le nom de tri rapide, dont la complexité en moyenne est de la forme $a'n\log(n) + b'$.

On constate que, quelles que soient les valeurs de a, b, c, a' et b', il existe une valeur de n pour laquelle la première expression devient supérieure à celle de la deuxième. Lorsque l'on travaille sur des tableaux de grande dimension, on aura donc intérêt à choisir un algorithme de la deuxième catégorie. On dira que le premier type d'algorithme est en n^2 et le deuxième en $n\log(n)$.

Notations en O et Θ

Définition 11 (domination asymptotique) *Une fonction f de $\mathbb{N}^*$ dans $\mathbb{R}$ est dominée asymptotiquement par une fonction g, elle aussi de $\mathbb{N}^*$ dans $\mathbb{R}$ si, et seulement si :*

$\exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}^*$ tels que $\forall n \in \mathbb{N}^*, n > n_0 \Rightarrow f(n) \leq c \times g(n)$

On utilisera alors la notation $f = \mathrm{O}(g)$, et dira que « f est en grand O de g ».

Les complexités dans le meilleur des cas, le pire des cas et en moyenne du tri par sélection sont donc en $\mathrm{O}(n^2)$. Les complexités dans le pire des cas et en moyenne du tri par insertion sont en $\mathrm{O}(n^2)$, et la complexité dans le meilleur des cas est en $\mathrm{O}(n)$.

Cependant, ces complexités sont toutes dominées asymptotiquement par des fonctions d'ordre de grandeur supérieur, et on peut donc en particulier aussi dire qu'elle sont toutes en $\mathrm{O}(n^3)$ et en $\mathrm{O}(e^n)$.

Définition 12 (ordre de grandeur) *Deux fonctions f et g de $\mathbb{N}^*$ dans $\mathbb{R}$ ont même ordre de grandeur asymptotique si, et seulement si :*

$\exists c \in \mathbb{R}^{+*}, \exists d \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}^*$ tels que $\forall n \in \mathbb{N}^*, n > n_0 \Rightarrow d \times g(n) \leq f(n) \leq c \times g(n)$

On utilisera alors la notation $f = \Theta(g)$, et on dira que « f est en grand thêta de g ».

Les complexités dans le meilleur des cas, le pire des cas et en moyenne du tri par sélection sont donc en $\Theta(n^2)$.

Les complexités dans le pire des cas et en moyenne du tri par insertion sont en $\Theta(n^2)$), et la complexité dans le meilleur des cas est en $\Theta(n)$).

Échelle de comparaison

Nous allons envisager ici un ensemble de fonctions représentant les ordres de grandeur de complexité les plus classiques, formant ainsi une échelle de comparaison pour les algorithmes. Pour toutes ces fonctions, n est le paramètre reflétant la taille des données. Chaque fonction envisagée est d'un ordre de grandeur supérieur à la précédente (elle la domine asymptotiquement).

— $f(n) = 1$
On parle dans ce cas de complexité constante. Toute fonction constante est en $\Theta(1)$.

— $f(n) = \log(n)$
On parle dans ce cas de complexité logarithmique. En particulier, toute fonction f telle que $f(n) = a\log(n) + b$, où a et b sont des constantes est en $\Theta(\log(n))$. La propriété $1 = \mathrm{O}(\log(n))$ est vérifiée.

— $f(n) = n$
On parle dans ce cas de complexité linéaire. En particulier, toute fonction f telle que $f(n) = an + b$, où a et b sont des constantes est en $\Theta(n)$. La propriété $\log(n) = \mathrm{O}(n)$ est vérifiée.

— $f(n) = n\log(n)$
En particulier, toute fonction f telle que $f(n) = an\log(n) + bn + c$, où a, b et c sont des constantes est en $\Theta(n\log(n))$. La propriété $n = \mathrm{O}(n\log(n))$ est vérifiée.

— $f(n) = n^2$
En particulier, toute fonction f telle que $f(n) = an^2 + bn + c$, où a, b et c sont des constantes est en $\Theta(n^2)$. La propriété $n\log(n) = O(n^2)$ est vérifiée.

— $f(n) = n^k$
On parle dans ce cas de complexité polynomiale. Pour toute valeur de k supérieure à 2, cette fonction domine asymptotiquement toutes les précédentes.

— $f(n) = 2^n$
On parle dans ce cas de complexité exponentielle. Cette fonction domine asymptotiquement toutes les précédentes.

A titre d'exemple, et afin mettre en évidence toute l'importance de ces calculs de complexité, nous allons donner les évolutions du temps d'exécution d'un programme utilisant l'algorithme de tri par insertion et d'un autre utilisant l'algorithme du tri rapide (qui est étudié par ailleurs) en fonction de la taille du tableau à classer. Nous ajouterons à ces programmes un hypothétique programme utilisant un algorithme exponentiel, afin de monter à quel point un tel algorithme devient vite inutilisable. Les temps indiqués sont ceux observés sur un ordinateur exécutant environ cent millions d'instructions par secondes, c'est-à-dire un ordinateur dont les performances sont très raisonnables au jour où ce chapitre est écrit.

	Complexité	Taille du tableau		
		100	10000	100000
Tri par insertion dans cas le meilleur	$\Theta(n)$	0,09 ms	9 ms	90 ms
Tri par insertion dans le cas moyen	$\Theta(n^2)$	1,75 ms	17 s	29 mn
Tri rapide dans le cas moyen	$\Theta(n\log(n))$	0,5 ms	100 ms	1,2 s
Tri rapide dans le cas le pire	$\Theta(n^2)$	2 ms	20 s	33 mn
Algorithme exponentiel	$\Theta(2^n)$	1 ms	5×10^{2969} jours	énorme!!!

On constate de manière évidente qu'un algorithme exponentiel devient très vite inutilisable lorsque la taille des données croît. L'amélioration des performances des ordinateurs ne saurait représenter une solution à ce problème (une multiplication par 1024 de la vitesse de l'ordinateur ne permettrait que d'ajouter 10 au nombre de données traitées dans le même temps). Il convient donc d'éviter de tels algorithmes dès que le problème posé le permet (ce qui n'est pas toujours le cas).

On remarquera que l'algorithme de tri par insertion est loin d'être ridicule devant un tri rapide lorsque la configuration des données est appropriée.

Une dernière remarque concerne le tri rapide qui, s'il est probablement le meilleur algorithme de classement connu à ce jour pour le cas moyen, devient assez mauvais si par malheur les données sont dans une configuration défavorable. On lui préférera alors un autre algorithme, comme le tri pas tas (heap sort), dont la complexité est en $\Theta(n\log(n))$ dans le cas moyen comme dans le cas le pire.

Chapitre 6

Algorithmes de classement

Les algorithmes de classement, aussi appelés *tris* bien que cette appellation soit impropre, permettent de classer les éléments dans un certain ordre.

Ces algorithmes sont particulièrement importants car ils sont utilisés pour améliorer l'efficacité de la recherche d'un élément dans une liste et la comparaison de deux listes. En effet, lorsqu'une liste est classée, on peut arrêter une recherche séquentielle dès qu'on rencontre un élément plus grand que celui qu'on recherche. On peut aussi utiliser une méthode de recherche dichotomique.

Le classement des éléments d'une liste s'appuie sur une propriété de ces éléments. Par exemple, une liste de personnes pourra être classée par ordre alphabétique croissant des noms des personnes. La propriété des éléments utilisée pour le classement est appelée « clef » du classement. Deux éléments ayant la même clef ne sont pas forcément identiques, ce qui fait que la relation utilisée pour le classement n'est pas anti-symétrique : $(a \leq b) \wedge (b \leq a) \not\Rightarrow (a == b)$. Cette relation n'est donc pas une relation d'ordre, mais un *pré-ordre*. Deux éléments distincts qui ne peuvent pas être distingués dans la relation de pré-ordre sont dits *équivalents*.

Un tri (ou algorithme de classement) est dit *stable* s'il conserve l'ordre dans la liste des éléments équivalents. Dans le cas contraire, le tri est dit *instable*.

Soit une liste $l = (e_i)_{1 \leq i \leq n}$, et $(c_i)_{1 \leq i \leq n}$ les clefs associées aux éléments de l. Soit $\leq$ une relation d'ordre total sur les clefs, et $\preceq$ le pré-ordre total induit par $\leq$ sur les éléments de l. Classer la liste l selon $\preceq$ consiste à déterminer une permutation σ de $[1, n]$ telle que :

$$\forall i \in [1, n-1], c_{\sigma(i)} \leq c_{\sigma(i+1)}$$

Le classement est stable si de plus :

$$\forall i, j \in [1, n], (i < j) \wedge (c_i = c_j) \Rightarrow \sigma(i) < \sigma(j)$$

6.1 Tris par sélection

Les tris par sélection consistent à rechercher un élément minimal de la liste, à le placer dans la liste classée, puis à réitérer l'opération sur la liste initiale privée de cet élément minimal.

Pour simplifier, on distingue ici la liste à classer de la liste classée. Dans la pratique, il est assez fréquent de déplacer les éléments dans la liste au lieu de les placer dans une nouvelle liste résultat. Lorsque c'est le cas, on parle de tri *sur place*.

6.1.1 Sélection ordinaire

Le tri par sélection le plus simple consiste à parcourir la liste du début à la fin pour rechercher un élément minimal. On échange alors cet élément minimal avec l'élément situé en tête de la liste, et on recommence avec la liste privée de son premier élément.

En utilisant un tableau pour représenter la liste à classer, le tri par sélection ordinaire peut être codé comme suit :

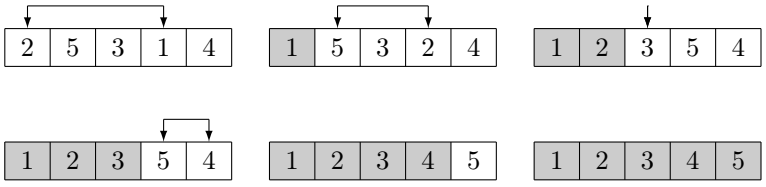
```
1  static void classer(int[] tab) {
2    // On effectue un classement "sur place", en considérant des
3    // tableaux de plus en plus courts.
4    for (int premier = 0; premier < tab.length; premier++) {
5      // On recherche un élément minimal
6      int indice_mini = premier;
```

```
7       for (int i = premier + 1; i < tab.length; i++) {
8         if (tab[i] < tab[indice_mini]) {
9           indice_mini = i;
10        }
11      }
12      // Si l'élément minimal n'est pas le premier élément du
13      // sous-tableau considéré, on l'échange avec le premier.
14      if (indice_mini != premier) {
15        int tmp = tab[premier];
16        tab[premier] = tab[indice_mini];
17        tab[indice_mini] = tmp;
18      }
19    }
20  }
```

La boucle qui commence à la ligne 4 applique l'algorithme à la liste complète, puis à la liste privée de son premier élément etc.

L'algorithme consiste à rechercher un élément minimal grâce à la boucle qui débute à la ligne 7, puis à permuter cet élément minimal avec le premier élément de la liste (lignes 15 à 17) dans le cas où l'élément minimal n'est pas en tête de liste.

La figure suivante montre l'évolution d'une liste de 5 éléments au cours de ce classement, la zone grisée indiquant la partie classée de la liste, et les flèches indiquant la permutation de l'élément minimal avec le premier élément de la partie non-classée de la liste :



Ce tri est stable puisque, si plusieurs éléments équivalents se trouvent dans la liste, ils seront traités dans leur ordre d'apparition dans la liste. Le premier sera placé en tête de liste avant que le second soit placé en tête de la liste privée du premier etc.

Complexité

La boucle qui débute à la ligne 4 est effectuée n fois si n est la longueur de la liste. Chaque exécution du corps de cette boucle s'applique à une sous-liste de longueur k. Dans la boucle qui débute à la ligne 7, on compare le premier élément de la liste à tous les autres, ce qui fait $k-1$ comparaisons. L'algorithme complet effectue donc $\sum_{k=n}^{1}(k-1)$, soit $\sum_{k=n-1}^{0} k = \frac{n(n-1)}{2}$ comparaisons.

Dans tous les cas, la complexité en nombre de comparaisons du tri par sélection est donc en $\mathrm{O}(n^2)$.

En ce qui concerne le nombre de permutations, tout dépend de la disposition des éléments dans la liste :

— si la liste est déjà classée, il n'y aura aucune permutation ;

— si la liste est presque classée, mais que son plus grand élément est en tête, il y aura une permutation à chaque tour, soit $n-1$ permutations.

La complexité en nombre de permutations est donc au pire en $\mathrm{O}(n)$, alors que la complexité en nombre de comparaisons est toujours en $\mathrm{O}(n^2)$. Asymptotiquement, c'est la donc la complexité en nombre de comparaisons qui prime.

6.1.2 Sélection par transposition, ou « tri à bulles »

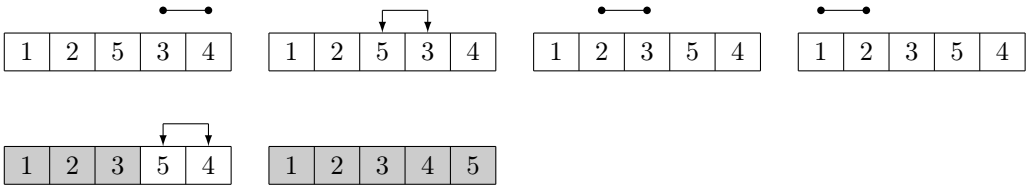
Cet algorithme consiste à effectuer les permutations en même temps que la recherche d'un élément minimal : en partant de la fin de la liste à classer, on permute un élément avec le précédent s'ils ne sont pas dans l'ordre. L'élément minimal se comporte donc comme une « bulle » qui remonte vers le début de la liste.

```
1  static void classer(int[] tab) {
2    int premier = 0;  // indice du premier élément de la liste
3    int dernier = tab.length - 1; // indice du dernier élément
4
5    // On effectue un classement "sur place", en considérant des
6    // tableaux de plus en plus courts.
7    while (premier < dernier) {
```

```
8       int derniere_perm = dernier; // rang de la dernière permutation
9       for (int i = dernier; i > premier; i--) {
10        if (tab[i] < tab[i-1]) {
11          derniere_perm = i;
12          int tmp = tab[i];
13          tab[i] = tab[i-1];
14          tab[i-1] = tmp;
15        }
16      }
17      premier = derniere_perm;
18    }
19  }
```

Tous les éléments situés avant les deux derniers éléments permutés sont correctement classés, il est donc inutile de les comparer de nouveau. On mémorise donc le rang de la dernière permutation dans la variable derniere_perm, et on l'utilise pour mettre à jour premier afin de ne pas faire de comparaisons inutiles par la suite.

La figure suivante montre l'évolution d'une liste de 5 éléments au cours de ce classement, la zone grisée indiquant la partie que l'on sait être classée puisqu'elle se trouve avant la dernière permutation effectuée. Les flèches indiquent la comparaison suivie de la permutation de deux éléments, alors que les points indiquent une comparaison sans permutation :



La dernière permutation se fait entre les 3eet 4éléments lors de la sélection du premier élément minimal. Au tour suivant, on sait donc que les 3 premiers éléments sont bien classés, il ne reste donc que les deux derniers à comparer.

Complexité

Comme pour le tri par sélection ordinaire, on doit au pire faire remonter $n-1$ fois l'élément minimal en début d'une liste de longueur k. La complexité au pire de cet algorithme en nombre de comparaisons est donc aussi en $O(n^2)$.

Dans le pire des cas, on est aussi amené à permuter deux éléments à chaque tour de boucle interne, ce qui donne une complexité au pire en nombre de permutations en $O(n^2)$. On peut montrer que la complexité en moyenne est du même ordre de grandeur.

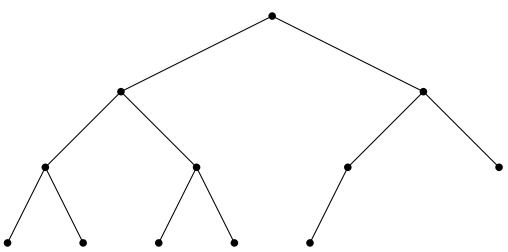
Dans le cas général, le tri à bulle n'est donc pas intéressant comparé au tri par sélection ordinaire. Il n'est intéressant que lorsque les listes sont déjà classées à quelques permutations d'éléments consécutifs près.

Ce tri est stable puisque deux éléments équivalents ne sont jamais permutés.

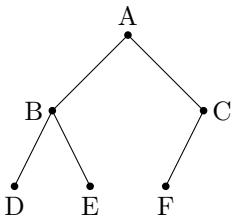
6.1.3 Le tri par tas

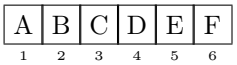
Comme son nom l'indique, cet algorithme utilise un *tas* pour sélectionner un élément minimal de la liste. Un tas est la représentation sous forme de liste d'un arbre binaire parfait partiellement ordonné.

Un arbre binaire est un arbre dont les nœuds ont au plus deux fils. Un arbre binaire de est dit *parfait* lorsque tous ses niveaux sauf le dernier sont pleins, et que le dernier niveau est rempli par la gauche :



Un tel arbre se représente facilement sous la forme d'une liste, en plaçant la racine en tête de liste, suivie des ses fils (le gauche, puis le droit), suivis de leurs fils etc. Le fils gauche de l'élément i est l'élément $2i$, son fils droit étant l'élément $2i+1$. Le père de l'élément i est l'élément $\lfloor \frac{i}{2} \rfloor$, comme indiqué ci-dessous :





Lorsque les étiquettes des nœuds de l'arbre sont munies d'un pré-ordre total, on dit que l'arbre parfait est *partiellement ordonné* lorsque tout nœud a une étiquette supérieure ou égale à celle de son père s'il existe. La liste représentant un tel arbre est alors un tas.

La racine d'un arbre parfait partiellement ordonné étant un élément minimal de l'arbre, la tête du tas correspondant est un élément minimal de ce tas. C'est cette propriété qui est utilisée pour sélectionner un élément minimal dans l'algorithme de classement par tas.

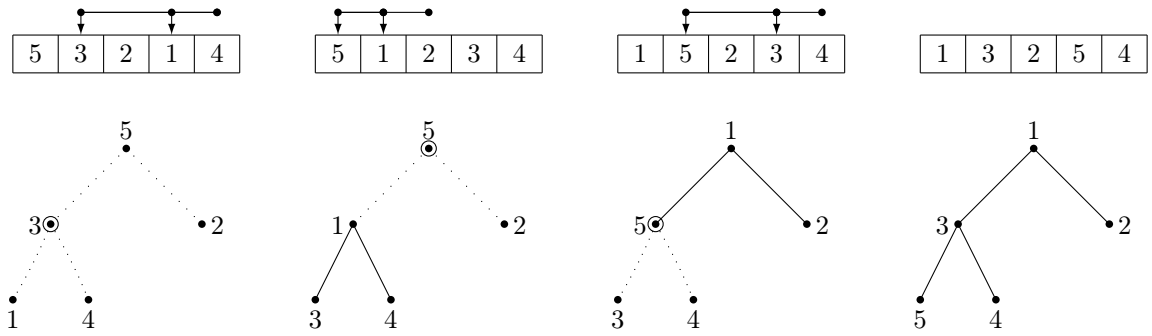
On commence donc par construire un tas à partir de la liste à classer. On parcourt pour cela les nœuds internes de l'arbre (ceux qui ne sont pas des feuilles) en partant de celui qui se trouve le plus bas à droite (le nœud C dans l'exemple précédent). Pour chaque nœud, si un des fils a une étiquette supérieure à celle de son père, on permute le nœud avec celui des fils qui a l'étiquette la plus petite, et on recommence jusqu'à ce que le sous-arbre considéré soit partiellement ordonné ou qu'on atteigne une feuille.

Il est facile de trouver le dernier nœud interne : c'est le père de la dernière feuille, donc l'élément $\lfloor \frac{n}{2} \rfloor$ du tas si n est la taille de la liste à classer. Lorsqu'on utilise un tableau pour représenter la liste, les éléments sont indicés à partir de 0. Le dernier nœud interne a donc pour indice $\lfloor \frac{n}{2} \rfloor - 1$, le fils gauche de l'élément d'indice i a pour indice $2i + 1$ et son fils droit $2i + 2$. L'algorithme de construction du tas devient :

```
1  static int construireTas(int [] tab) {
2    int courant = (tab.length / 2) - 1;
3    while (courant ≥0) {
4      inserer(tab, courant, tab.length);
5      courant--;
6    }
7    return tab.length;
8  }
9
10 static void inserer(int [] tab, int courant, int taille ) {
11   int gauche = 2 * courant + 1;
12   int droit = 2 * courant + 2;
13   if (gauche ≥ taille ) {
14     return; // le noeud courant est une feuille
15   }
16   int filsmin = gauche;
17   if ((droit < taille ) && (tab[droit] < tab[gauche])) {
18     filsmin = droit;
19   }
20   if (tab[courant] > tab[filsmin]) {
21     echanger(tab, courant, filsmin );
22     inserer(tab, filsmin , taille );
23   }
24 }
```

La méthode `inserer` réorganise l'arbre de façon à ce que le sous-arbre de racine l'élément d'indice `courant` soit partiellement ordonné. La méthode `construireTas` applique `inserer` à chacun des nœuds internes, ce qui rend l'arbre global partiellement ordonné et transforme donc la liste en tas.

La méthode `inserer` commence par rechercher le fils du nœud courant qui a l'étiquette la plus petite, puis, si cette étiquette est supérieure à celle du nœud courant, elle permute les deux nœuds et se rappelle pour former un tas avec le sous-arbre ayant pour racine le fils permuté. La figure suivante montre le déroulement de la construction du tas pour la liste `{5, 3, 2, 1, 4}`. Le nœud inséré est indiqué par un cercle. Les traits en pointillé indiquent les filiations pour lesquelles l'ordre n'est pas forcément respecté, alors que les traits pleins indiquent le respect de l'ordre :

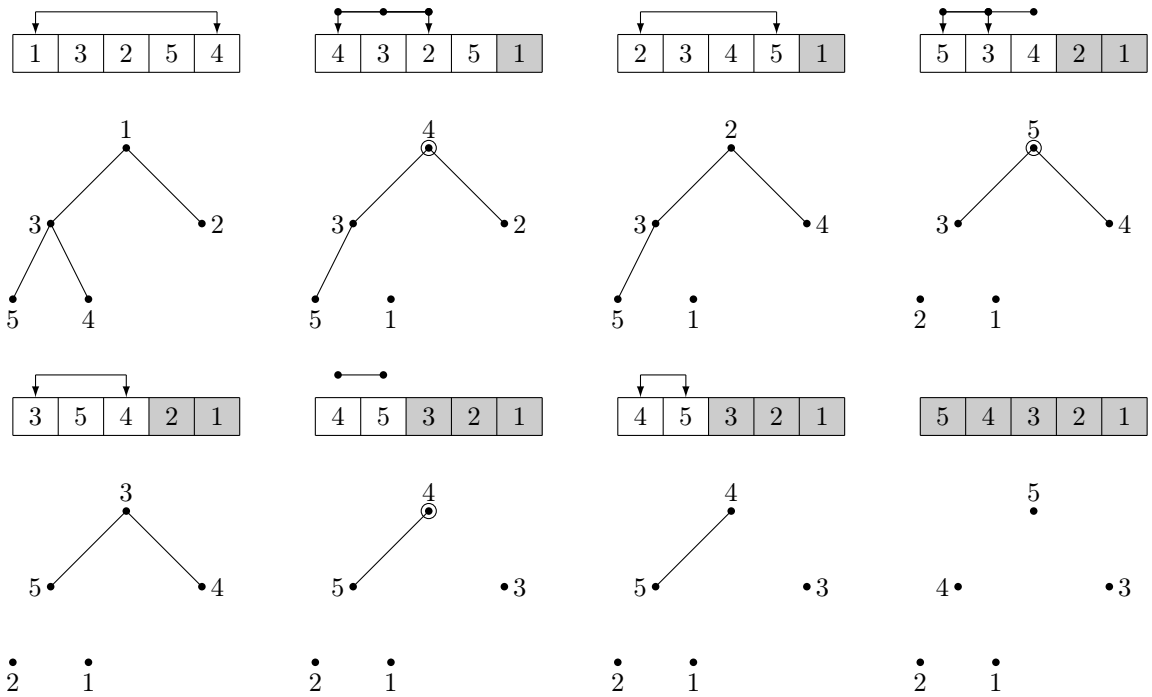


Une fois le tas construit, la sélection d'un élément minimal consiste à prendre l'élément en tête du tas, qui est la racine de l'arbre parfait partiellement ordonné.

Il faut ensuite appliquer de nouveau l'algorithme au tas privé de son premier élément. Mais il serait stupide de reconstruire complètement un tas à partir de la liste privée de son premier élément. On va donc conserver la structure de notre tas en permutant son premier et son dernier élément. La taille du tas va être réduite d'une unité afin que l'élément minimal ainsi placé en fin de liste ne fasse plus partie du tas. L'élément placé en tête de la liste (donc à la racine de l'arbre parfait) sera ensuite inséré pour restaurer la structure de tas. La méthode oterMin implémente cette opération :

```
1  static int oterMin(int [] tab, int taille ) {
2    // on permute l'élément minimal avec la dernière feuille
3    echanger(tab, 0, taille - 1)
4     taille --; // la taille est réduite
5     inserer (tab, 0, taille ); // reconstitution d'un tas
6    return taille ;
7  }
```

La figure suivante indique l'évolution du tas lorsque l'on retire ses éléments un à un, en insérant la nouvelle racine pour restaurer la structure de tas. La zone grisée correspond à la partie de la liste qui ne fait plus partie du tas :



Finalement, l'algorithme de classement par tas se code :

```
1  static void classer (int [] tab) {
2    int taille = construireTas(tab);
3    while ( taille > 0) {
4       taille = oterMin(tab, taille );
5    }
6  }
```

Le tri par tas classe les éléments de la liste en ordre inverse de l'ordre sur les clefs puisque l'élément minimal est toujours placé en fin de liste. Pour obtenir un classement usuel, il suffit d'utiliser la relation d'ordre inverse lors de la construction du tas, ou d'inverser la liste finale.

Le tri par tas n'est pas stable. Une liste contenant deux éléments équivalents est un contre-exemple de sa stabilité.

Complexité

Au pire, inserer effectue 2 comparaisons et une permutation avant de se rappeler sur un des fils du nœud courant. Ce fils est l'élément $2i$ ou l'élément $2i+1$ du tas, donc la position du nœud courant est au moins doublée à chaque appel. Comme cette position est limitée à la taille n du tas, il y a au plus $\log_2(\frac{n}{i}) = \log_2(n) - \log_2(i)$ appels récursifs pour insérer l'élément i dans un tas de taille n. La complexité au pire de inserer est donc en $O(log_2(n))$, aussi bien en nombre de comparaisons qu'en nombre de permutations.

Pour construire le tas, on appelle inserer pour chacun des $\lfloor \frac{n}{2} \rfloor$ nœuds internes de l'arbre. Considérons k tel que $2^{k-1} \leq i < 2^k$. k est le niveau du nœud i dans l'arbre, la racine étant au niveau 1 et les feuilles les plus basses au niveau h tel que $h - 1 \leq \log_2(n) < h$. Le nombre de nœuds situés au niveau k est $2^k - 2^{k-1} = 2^{k-1}$. Pour ces nœuds, la complexité au pire de inserer est inférieure à $\log_2(n) - k - 1$. Donc pour chaque niveau k de l'arbre, la complexité au pire de la construction du tas est $2^{k-1}(\log_2(n) - k - 1)$. Le nombre de niveaux est $h \leq \log_2(n) + 1$, mais comme on ne traite que les nœuds internes, seuls les $h - 1$ premiers niveaux sont à considérer. La complexité au pire de la construction du tas est donc :

$$C = \sum_{k=1}^{h-1} 2^{k-1}(\log_2(n) - k - 1) < \sum_{k=1}^{h-1} 2^{k-1}(h - k - 1)$$

En posant $i = h - k - 1$, il vient :

$$C < \sum_{i=h}^{2} 2^{h-i} i = \sum_{i=2}^{h} 2^{h-1} 2^{1-i} i = \sum_{i=2}^{h} 2^{h-1} (\frac{1}{2})^{i-1} i = 2^{h-1} \sum_{i=2}^{h} i(\frac{1}{2})^{i-1} < 2^{h-1} \sum_{i=1}^{\infty} i(\frac{1}{2})^{i-1}$$

Or $\sum_{i=1}^{\infty} i x^{i-1}$ est la dérivée par rapport à x de $\sum_{i=1}^{\infty} x^i = \frac{1}{1-x} - 1$ quand $|x| < 1$. La dérivée par rapport à x de $\frac{1}{1-x} - 1$ étant $\frac{1}{(1-x)^2}$, on a donc :

$$C < 2^{h-1} \frac{1}{(1 - \frac{1}{2})^2} = 2^{h-1} \times 4 \leq 4n$$

La complexité au pire en nombre de comparaisons ou de permutations de la construction du tas est donc en $O(n)$.

Pour classer la liste, on appelle ensuite n fois otermin sur des tas dont la taille varie de n à 1. La complexité de otermin étant celle de inserer plus une permutation, la complexité du classement proprement dit est donc :

$$\sum_{i=n}^{1} \log_2(i) = \log_2(\prod_{i=1}^{n} i) = \log_2 n!$$

D'après la formule de Stirling, $n!$ se comporte comme $n^n e^{-n} \sqrt{2\pi n}$ quand $n \to \infty$. Asymptotiquement, la complexité du classement est donc en $O(n \log_2(n))$. La complexité du tri par tas est donc en $O(n \log_2(n))$.

6.2 Classements par insertion

Ces algorithmes de classement consistent à insérer un nouvel élément à sa place dans une liste déjà classée.

6.2.1 Insertion séquentielle

La recherche de la position à laquelle le nouvel élément doit être inséré se fait en parcourant la liste de la fin au début, et en décalant vers la droite les éléments supérieurs à l'élément à insérer :

```
1  static void classer(int[] tab) {
2    for (int n = 1; n < tab.length; n++) {
3      int elem = tab[n];    // élément à insérer
4      int i = n;
5      while ((i > 0) && (tab[i-1] > elem)) {
```

```
6         tab[i] = tab[i-1]; // décalage vers la droite
7         i--;
8       }
9       if (i != n) {
10        tab[i] = elem;      // insertion .
11      }
12    }
13  }
```

Cet algorithme de classement est stable car lors de la recherche de la position d'insertion, on s'arrête au premier élément inférieur ou équivalent à l'élément à insérer, et on insère après (on compare avec l'élément $i-1$, mais on insère en i).

Complexité

Au pire, la recherche de la position d'insertion dans une liste de longueur k se fait en k comparaisons et k permutations si tous les éléments sont supérieurs à celui que l'on insère. Le classement consiste à insérer un élément dans des sous-listes de longueur 1 à $n-1$ de la liste à classer. La complexité au pire, tant en nombre de comparaisons qu'en nombre de permutations est donc :

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \mathrm{O}(n^2)$$

Lorsqu'on insère un élément supérieur ou équivalent à tous les éléments de la liste, l'insertion se fait en une seule comparaison. Cet algorithme est donc très efficace pour des listes presque classées.

6.2.2 Insertion dichotomique

Dans cette version du classement par insertion, on utilise une recherche dichotomique pour trouver la position d'insertion : on compare l'élément à insérer avec un élément situé vers le milieu de la liste. La liste étant classée, si l'élément à insérer est plus grand que l'élément milieu, la position d'insertion se trouve dans la deuxième partie de la liste, sinon, elle se trouve dans la première partie.

```
1  static void classer(int[] tab) {
2    for (int n = 1; n < tab.length; n++) {
3      int elem = tab[n];       // élément à insérer
4      int bas = 0;             // bas de l'intervalle de recherche
5      int haut = n;            // haut de l'intervalle de recherche
6      while (bas <= haut) {
7        int milieu = bas + (haut - bas)/2;
8        if (tab[milieu] <= elem) {
9          bas = milieu + 1;
10       } else {
11         haut = milieu - 1;
12       }
13     }
14     if (bas < n) {           // si l'élément à insérer n'est pas en place
15       for (int i = n; i > bas; i--) {
16         tab[i] = tab[i-1]; // décaler les éléments vers la droite
17       }
18       tab[bas] = elem;       // et insérer.
19     }
20   }
21 }
```

Les éléments situés à gauche de bas sont inférieurs ou équivalents à l'élément à insérer. Ceux situés à droite de haut lui sont strictement supérieurs. À la fin de la boucle **while**, haut est situé juste avant bas, donc haut indique un élément maximal inférieur ou équivalent à l'élément à insérer, et bas indique un élément minimal strictement supérieur à l'élément à insérer, si ces éléments existent. L'élément à insérer doit donc l'être entre haut et bas. Cet algorithme de classement est stable puisqu'un élément est inséré après les éléments équivalents déjà présents dans la liste.

Complexité

La recherche dichotomique divise l'intervalle de recherche en 2 à chaque étape jusqu'à ce qu'il soit vide. L'intervalle initial étant de longueur n, le nombre d'étapes est de l'ordre de $\log_2(n)$. Chaque étape compare l'élément à insérer à l'élément milieu, la recherche est donc en $\log_2(n)$ comparaisons. Cette recherche est faite dans les sous-listes de longueur 1 à $n-1$ de la liste à classer. La complexité du classement en nombre de comparaisons est donc :

$$\sum_{k=1}^{n-1} \log_2(k) = \log_2(\prod_{k=1}^{n-1} k) = \log_2((n-1)!) \underset{\text{(formule de Stirling)}}{\approx} \log_2((n-1)^{n-1}) = \mathrm{O}(n \log_2(n))$$

La complexité en nombre de transferts est en $\mathrm{O}(n^2)$, comme pour l'insertion séquentielle car le nombre d'éléments à décaler est le même.

6.3 Le tri rapide

Les tris par insertion insèrent un élément à sa place dans une sous-liste déjà classée de la liste complète, et répètent cette opération sur des sous-listes classées de longueur croissante jusqu'à ce que toute la liste soit classée.

L'algorithme du tri rapide consiste à placer un élément p à sa place dans la liste complète. Il faut pour cela déterminer quels sont les élément de la liste inférieurs à p et ceux supérieurs à p. On en profite pour placer les éléments supérieurs à droite de p et les éléments inférieurs à gauche de p.

Lorsque l'élément p est bien placé dans la liste, les sous-listes situées à gauche et à droite de p peuvent donc être classées indépendamment l'une de l'autre en appliquant le même algorithme.

La phase cruciale du tri rapide est donc le placement de l'élément p — appelé *pivot* — dans la liste, et la constitution des sous-listes gauche et droite contenant respectivement les éléments inférieurs au pivot et les élément supérieurs au pivot. C'est ce qu'on appelle la *partition* de la liste.

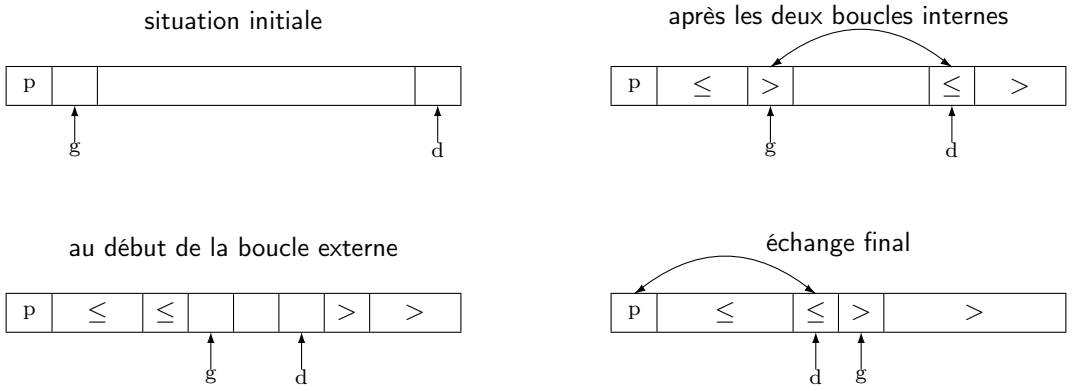
Le principe de partition est, partant de chaque extrémité de la liste, de rechercher le dernier élément inférieur ou équivalent au pivot et le premier élément supérieur au pivot, puis de les échanger. L'opération est répétée jusqu'à ce qu'il n'y ait plus d'éléments de la liste entre ces deux éléments.

Le dernier élément inférieur ou équivalent au pivot sont alors échangés de façon à ce que tous les éléments situés à gauche du pivot lui soient inférieurs ou égaux, et tous ceux situés à sa droite lui soient supérieurs.

On choisit en général le premier élément de la liste pour pivot, ce qui conduit à l'implémentation suivante de partition :

```
1  static int partition (int [] tab, int bas, int haut) {
2    int p = tab[bas];       // on prend le 1er élément pour pivot
3    int g = bas + 1;        // les éléments à gauche de g sont <= p
4    int d = haut;           // les éléments à droite de d sont > p
5
6    while (g <= d) {        // tant qu'il y a des éléments indéterminés
7      while (tab[d] > p) { // si l'élément d est > p ...
8        d--;                //     il est à droite de d.
9      }
10     while ((g <= d) && (tab[g] <= p)) {
11       g++;                // tab[g] <= p, donc àgauche de g
12     }
13     if (g < d) {          // tab[g] > p et tab[d] <= p
14       int tmp = tab[g]; // on les échange
15       tab[g] = tab[d];
16       tab[d] = tmp;
17       g++;                // et on met à jour g et d
18       d--;
19     }
20   }
21   if (bas != d) {         // tab[d] est le dernier élément <= p
22     tab[bas] = tab[d];    // donc on l'échange avec p qui est
23     tab[d] = p;           // maintenant à sa place.
24   }
25   return d;
26 }
```

Le schéma suivant montre le déroulement de partition , du début où les indices g et d sont placés à chaque extrémité de la liste, à la fin où d indique la position du pivot. À la fin de la première ligne du schéma, les deux boucles internes ont été exécutées, et les éléments indiqués par g et d vont être permutés. Au début de la deuxième ligne, ils ont été permutés, et g et d ont été mis à jour. La boucle externe s'exécute jusqu'à se que g et d se croisent. On arrive alors à la situation finale, et il ne reste qu'à permuter le pivot avec l'élément indiqué par d, et à rendre d :



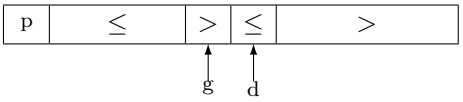
L'algorithme du tri rapide consiste à classer de la même façon les sous-listes créées par partition . On a donc :

```
1  static void triRapide(int[] tab, int bas, int haut) {
2    if (bas < haut) {
3      int pivot;
4      pivot = partition(tab, bas, haut);
5      triRapide(tab, bas, pivot - 1);
6      triRapide(tab, pivot + 1, haut);
7    }
8  }
```

6.3.1 Complexité

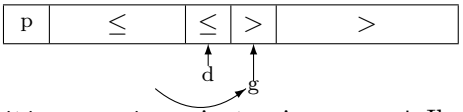
L'algorithme de partition s'arrête lorsque g et d se croisent. Ceci peut se produire de trois façons :

— les boucles internes s'arrêtent avec $d = g + 1$, puis les éléments sont permutés et les deux indices sont mis à jour, rendant d inférieur à g :



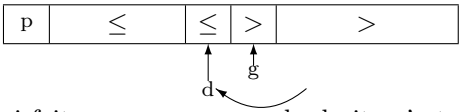
On a alors comparé au pivot les éléments compris entre le pivot et g inclus, soit $g - 1$ comparaisons. On a aussi comparé au pivot les éléments compris entre d inclus et la fin de la liste. Si n est la longueur de la liste, cela fait $n - d + 1$ comparaisons. Comme $d = g + 1$, cela fait un total de $n - 1$ comparaisons.

— les boucles internes s'arrêtent après que g a dépassé d vers la droite :



L'élément indiqué par g n'a pas été comparé au pivot puisque $g < d$. Il y a donc eu $(n - d + 1) + (g - 2)$, avec $g = d + 1$, ce qui donne n comparaisons.

— enfin, les boucles internes peuvent s'arrêter après que d a dépassé g vers la gauche :



Dans ce cas, la boucle interne qui fait progresser g vers la droite n'est pas exécutée. g se trouve donc à l'endroit où il a été placé lors de la dernière permutation, et l'élément indiqué par g n'a pas été comparé au pivot. Le nombre de comparaisons est donc le même que dans le cas précédent, soit n.

La complexité de partition en nombre de comparaisons est donc de l'ordre de n dans tous les cas.

En ce qui concerne le nombre de permutations d'éléments, dans le meilleur des cas (la liste est classée), il n'y en a aucune. Dans le pire des cas, il peut y avoir une permutation à chaque fois qu'un des indices g

ou d progresse. Après une permutation, g est incrémenté et d est décrémenté, le nombre d'éléments restant à comparer est donc diminué de 2. Comme il y a au départ $n-1$ éléments à comparer au pivot, on aura au maximum $\lfloor \frac{n-1}{2} \rfloor = \lfloor \frac{n}{2} \rfloor - 1$ permutations.

La complexité de partition en nombre de permutations d'éléments est donc de l'ordre de n dans le pire des cas.

triRapide se rappelle récursivement sur chacune des sous-listes déterminées par partition . À chaque niveau k de la récursion, il y a au plus $n-k$ éléments à classer puisqu'on élimine le pivot. Les deux appels à triRapide travaillent donc sur des sous-listes dont la somme des longueurs est au plus $n-k$. La complexité de partition à ce niveau de la récursion est donc de l'ordre de $n-k$. Comme la récursion s'arrête quand la sous-liste est de longueur inférieure ou égale à 1, il y a au plus $n-1$ niveaux de récursion. La complexité au pire du tri rapide est donc :

$$\sum_{k=0}^{n-1} n-k = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \mathrm{O}(n^2)$$

Au mieux, le niveau de récursion k détermine 2^k pivots : l'appel principal, au niveau 0, détermine un pivot et appelle triRapide sur deux sous-listes de plus d'un élément, ce qui permet de déterminer 2 pivots au niveau 1 etc. Le nombre d'éléments à comparer au niveau $k>0$ est donc au mieux $n-2^{k-1}$ (on retire à chaque fois les pivots placés au niveau précédent). La récursion s'arrêtant lorsque la sous-liste est de longueur inférieure ou égale à 1, on aura au plus $1+\log_2(n-1)$ niveaux de récursion. À chaque niveau de récursion k, partition est appliquée à des sous-listes dont la somme des longueurs vaut $n-2^{k-1}$ pour $k>0$ (la liste étant de longueur n au niveau 0). La complexité du tri rapide dans le meilleur des cas est donc :

$$\begin{aligned}
& n + \sum_{k=1}^{1+\log_2(n-1)} n - 2^{k-1} \\
= \;& n + (1+\log_2(n-1))n - \sum_{k=1}^{1+\log_2(n-1)} 2^{k-1} \\
= \;& n(2+\log_2(n-1)) - \sum_{i=2}^{\log_2(n-1)} 2^k \\
= \;& n(2+\log_2(n-1)) - (2^{1+\log_2(n-1)} - 2^2) \\
= \;& n(2+\log_2(n-1)) - (2(n-1)-4) = n\log_2(n-1) + 5 \\
= \;& \mathrm{O}(n\log_2(n))
\end{aligned}$$

La complexité en moyenne du tri rapide pour un tableau de n éléments est la complexité en moyenne du placement pour n élément plus la complexité en moyenne des tris rapides des deux partitions du tableau. Le pivot peut se trouver à n'importe quelle position dans le tableau, on a donc n cas possibles. La complexité en moyenne est donc la moyenne de la complexité de tous ces cas ($n>1$). Le sous-tableau de droite comporte les éléments d'indice 1 à $p-1$, soit $p-1-1+1 = p-1$ éléments. Le sous-tableau de gauche comporte les éléments d'indice $p+1$ à n, soit $n-(p+1)+1 = n-p-1+1 = n-p$ éléments, d'où la formule suivante :

$$M(n) = M_{\text{placement}}(n) + \frac{1}{n}\sum_{p=1}^{n} M(p-1) + M(n-p)$$

Comme la complexité au pire du placement pour n éléments est n, sa complexité en moyenne est $\leq n$. On a donc, en notant $M(n)$ la complexité en moyenne du tri rapide pour un tableau de n éléments :

$$M(n) \leq n + \frac{1}{n}\sum_{p=1}^{n} M(p-1) + M(n-p)$$

On a de plus $M(0) = M(1) = 0$.
On pose $S_n = n + \frac{1}{n}\sum_{p=1}^{n}(S_{p-1} + S_{n-p})$ pour $n \geq 2$, avec $S_0 = S_1 = 0$.
On a alors :

$$S_n \quad = \quad n + \frac{1}{n}\left(\sum_{k=0}^{n-1} S_k + \sum_{l=n-1}^{0} S_l\right) \quad \text{avec} \quad k = p-1 \quad \text{et} \quad l = n-p$$

$$\begin{aligned} &= n + \frac{1}{n}\left(\sum_{k=0}^{n-1} S_k + \sum_{l=0}^{n-1} S_l\right) \\ &= n + \frac{2}{n}\sum_{k=0}^{n-1} S_k \quad \text{pour} \quad n \geq 2 \end{aligned}$$

$$S_{n-1} = n - 1 + \frac{2}{n-1}\sum_{k=0}^{n-2} S_k \quad \text{pour} \quad n \geq 3$$

On veut soustraire les deux séries pour faire apparaître S_{n-1}.
On va pour cela supprimer le facteur $\frac{2}{n}$ devant la somme :

$$\begin{aligned} nS_n &= n^2 + 2\sum_{k=0}^{n-1} S_k \quad \text{pour} \quad n \geq 2 \\ (n-1)S_{n-1} &= (n-1)^2 + 2\sum_{k=0}^{n-2} S_k \quad \text{pour} \quad n \geq 3 \end{aligned}$$

d'où :

$$\begin{aligned} nS_n - (n-1)S_{n-1} &= n^2 - (n-1)^2 + 2\left(\sum_{k=0}^{n-1} S_k - \sum_{k=0}^{n-2} S_k\right) \quad \text{pour} \quad n \geq 3 \\ &= n^2 - (n-1)^2 + 2S_{n-1} \\ nS_n &= n^2 - (n^2 - 2n + 1) + (n+1)S_{n-1} \\ &= 2n - 1 + (n+1)S_{n-1} \\ S_n &= 2 - \frac{1}{n} + (n+1)\frac{S_{n-1}}{n} \\ \frac{S_n}{n+1} &= \frac{2}{n+1} - \frac{1}{n(n+1)} + \frac{S_{n-1}}{n} \\ \text{or} \quad & \frac{1}{n(n+1)} = \frac{1}{n} - \frac{1}{n+1} \\ \frac{S_n}{n+1} &= \frac{2}{n+1} - \frac{1}{n} + \frac{1}{n+1} + \frac{S_{n-1}}{n} \\ \frac{S_n - 1}{n+1} &= \frac{2}{n+1} + \frac{S_{n-1} - 1}{n} \quad \text{pour} \quad n \geq 3 \end{aligned}$$

On pose alors $C_n = \frac{S_n - 1}{n+1}$ et on obtient :

$$\begin{aligned} C_n &= C_{n-1} + \frac{2}{n+1} \quad \text{pour} \quad n \geq 3 \\ C_0 &= \frac{S_0 - 1}{1} = 0 - 1 = -1 \\ C_1 &= \frac{S_1 - 1}{2} = \frac{0-1}{2} = -\frac{1}{2} \\ C_2 &= \frac{S_2 - 1}{3} = \frac{2-1}{3} = \frac{1}{3} \end{aligned}$$

$$\begin{aligned} C_n &= C_2 + \sum_{k=3}^{n} \frac{2}{k+1} \\ &= \frac{1}{3} + 2\sum_{l=4}^{n+1} \frac{1}{l} \\ &= \frac{1}{3} + 2\left(\sum_{l=1}^{n+1} \frac{1}{l} - \sum_{k=1}^{3} \frac{1}{k}\right) \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{3} + 2\left(\sum_{l=1}^{n+1}\frac{1}{l} - (1+\frac{1}{2}+\frac{1}{3})\right)\\
&= \frac{1}{3} - 2\frac{11}{6} + 2\sum_{l=1}^{n+1}\frac{1}{l}\\
C_n &= 2\mathcal{H}_{n+1} - \frac{10}{3}\\
\text{or}\quad & S_n = (n+1)C_n + 1\\
S_n &= (n+1)\left(2\mathcal{H}_{n+1} - \frac{10}{3}\right) + 1\\
&= 2(n+1)\mathcal{H}_{n+1} - \frac{10n}{3} - \frac{7}{3}\\
S_n &= \Theta(n\log(n))
\end{aligned}
$$

où $\mathcal{H}_n$ désigne la série harmonique $\sum_{k=1}^{n}\frac{1}{n}$ qui se comporte comme $\mathcal{C} + \log(n)$ quand $n \to \infty$, $\mathcal{C} = 0,577215$ étant la constante d'Euler.

De tous les algorithmes de classement connus, le tri rapide est le plus efficace en moyenne lorsque la taille des listes est suffisamment grande. Pour de petites listes, un classement par insertion sera plus efficace.

Le principal inconvénient du tri rapide est que sa complexité au pire est en $O(n^2)$. S'il est nécessaire d'avoir un classement en $O(n\log(n))$ dans tous les cas, on devra utiliser un tri par tas. Bien que la complexité en moyenne du tri par tas et celle du tri rapide soient toutes deux en $O(n\log(n))$, le tri par tas est plus lent.

6.4 Borne inférieure de la complexité d'un tri

Le tri par tas et le tri rapide ayant une complexité en $O(n\log(n))$, on peut se demander s'il est possible de faire mieux : existe-t-il un algorithme permettant de classer une liste avec une complexité d'un ordre de grandeur inférieur à $O(n\log(n))$?

Nous avons vu au début de ce chapitre qu'effectuer un classement revient à déterminer une permutation des positions des éléments de la liste. Pour une liste de longueur n, le nombre de permutations possibles est $n!$. Si on s'autorise pour seule opération la comparaison de deux éléments l_i et l_j de la liste, le résultat de chaque opération ne peut prendre que deux valeurs : $l_i \leq l_j$ ou $l_i \not\leq l_j$. Si on numérote les permutations de 0 à $n!-1$, chaque comparaison de deux éléments détermine au mieux un bit de la représentation binaire du numéro de la permutation recherchée. Sur k bits, on peut représenter les entiers de l'intervalle $[0..2^k-1]$, donc pour représenter les numéros des permutations, il faut un nombre de bits k tel que $2^k - 1 \geq n! - 1$, soit $k \geq \log_2(n!)$. D'après la formule de Stirling, $n! \approx n^n e^{-n}\sqrt{2\pi n}$ quand $n \to \infty$, donc k est au mieux de l'ordre de $n\log(n)$.

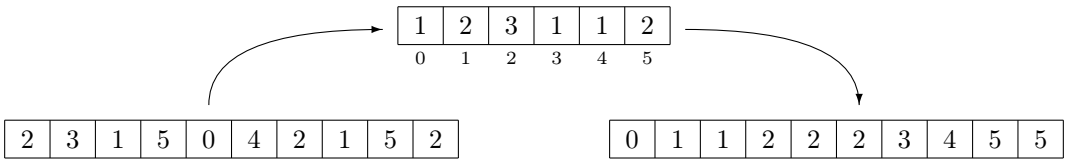
Il est donc impossible de classer une liste avec une complexité d'un ordre inférieur à $n\log(n)$ si on n'utilise que des comparaisons deux à deux d'éléments de la liste.

6.5 Tri par compteurs

Cet algorithme est un exemple de tri qui atteint une complexité inférieure à $n\log(n)$ en exploitant une information supplémentaire sur les éléments à classer.

Si tous les éléments de la liste sont des entiers de l'intervalle $[0..m-1]$, il suffit de compter le nombre d'occurrences de chaque élément dans la liste pour pour les classer : on construit la liste classée en lui ajoutant chaque élément autant de fois que l'indique le compteur correspondant.

La figure suivante montre le classement d'une liste d'entiers de l'intervalle [0..5] à l'aide d'un tableau de 6 compteurs :



Un tel classement s'effectue en n comparaisons (pour calculer les compteurs) et en n déplacements d'éléments (pour remplir la liste classée). Le classement peut se faire en place, mais il nécessite m compteurs. Sa complexité

en espace peut donc devenir très importante : pour classer des entiers codés sur 32 bits, il faudrait un tableau de 2^{32} compteurs !

Le tri par compteur fait partie de la famille des tris basés. Ces tris permettent de classer des éléments d'après une clef appartenant à une base finie. Pour des éléments quelconques, les compteurs sont remplacés par des listes d'éléments équivalents, le résultat étant obtenu par concaténation de ces listes.

6.6 Conclusion

Cet éventail d'algorithmes de classement montre qu'il n'y a pas d'algorithme idéal adapté à toutes les situations. Il ne faut pas oublier que les ordres de grandeur de complexité indiquent un comportement asymptotique pour des tailles de données très grandes. Il ne faut donc pas hésiter à utiliser un simple tri par insertion si on a souvent à classer des listes de quelques dizaines d'éléments, ou si on ajoute des éléments à une liste déjà classée, au fur et à mesure qu'ils sont disponibles. Par contre, ne pas utiliser le tri rapide ou le tri par tas lorsque c'est nécessaire rendra un programme inutilisable.

Chapitre 7

Files de priorité

Implémentation à l'aide d'une pile :

— insert = push
— delMax = boucle interne du tri par sélection pour placer un élément maximal au sommet, puis pop

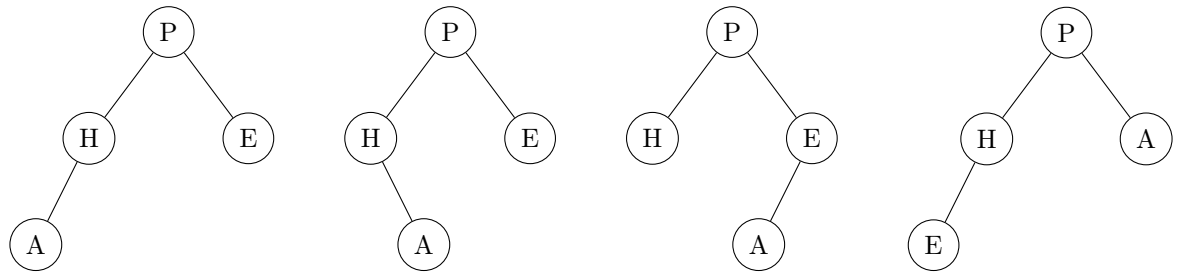
Complexité de insert = $O(1)$ (au pb d'allocation mémoire près pour une implementation par tableau), complexité de delMax = $O(n)$.

Autre possibilité : maintenir les éléments en ordre dans la pile. push insère un élément en laissant un élément maximal au sommet de la pile, delMax devient pop.

Complexité de insert = $O(n)$, complexité de delMax = $O(1)$ (au pb d'allocation mémoire près pour une implementation par tableau).

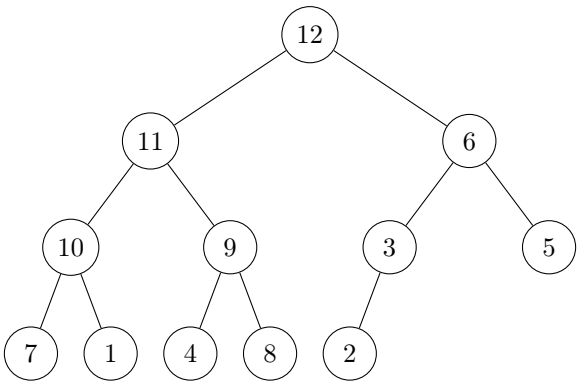
7.1 Tas

Un tas binaire (*binary heap*) est une structure de données dans laquelle chaque élement est supérieur ou égal à deux autres éléments (chacun étant lui-même supérieur ou égal à deux autres éléments). Cette structure correspond à celle d'un arbre binaire dont chaque nœud porte une valeur supérieure ou égale à celle de ses fils. Les trois arbres binaires ci-dessous sont ordonnés en tas et contiennent les éléments H, E, A, P :



L'intérêt de cette structure est que la racine d'un tas binaire est toujours un élément maximal. Toutefois, il est plus facile de représenter un arbre binaire en mémoire lorsqu'il est *complet*. Un arbre binaire complet est un arbre binaire dont tous les étages sont complets, sauf éventuellement le dernier, et dont les feuilles manquantes sont toutes situées à droite du dernier étage. Dans l'exemple ci-dessus, seuls le premier et le dernier arbre sont complets.

Un tas est donc un arbre binaire complet ordonné en tas. Un tas peut être représenté par un tableau en rangeant les éléments de l'arbre étage par étage, de la gauche vers la droite. Ainsi, la racine du tas se trouve à l'indice 0, son fils gauche à l'indice 1, son fils droit à l'indice 2 etc. :



12	11	6	10	9	3	5	7	1	4	8	2
0	1	2	3	4	5	6	7	8	9	10	11

Dans cette représentation d'un arbre binaire dans un tableau, la racine se trouve à l'indice 0, le fils gauche du nœud se trouvant à l'indice i est à l'indice $2i + 1$, sont fils droit est à l'indice $2(i + 1)$, et sa racine est à l'indice $(i - 1) \div 2$.

La navigation vers le père ou vers un des fils d'un nœud se fait donc en tant constant dans un arbre binaire représenté dans un tableau. D'autre part, un tas étant un arbre binaire complet, sa hauteur est la partie entière de $\log_2 n$ si n est son nombre de nœuds (le nombre de nœuds d'un arbre binaire complet de hauteur h est compris entre 2^h et $2^{h+1} - 1$). Le parcours d'un tas de la racine vers une feuille se fait donc avec une complexité en $O(\log n)$.

```
1  package priorityqueue;
2  import java.util.Iterator;
3
4  // A heap is a left-complete tree in which the value of a node is greater than the value of its sons.
5  // Hence the root holds the greatest value.
6  public abstract class Heap<T extends Comparable<T>> extends PriorityQueue<T> implements Iterable<T> {
7    private T[] tree_;
8    private int size_;
9
10   //@ Compare two elements of the heap (redefined in MaxHeap and MinHeap)
11   protected abstract boolean less(T k1, T k2);
12
13   public Heap() {
14     this(5);
15   }
16
17   public Heap(int capacity) {
18     this.size_ = 0;
19     resize(capacity);
20   }
21
22   protected Heap(T[] array) {
23     this.tree_ = array;
24     this.size_ = array.length;
25     for (int i = 1; i < this.size_; i++) {
26       swim(i);
27     }
28   }
29
30   // Heap sort of array T[] a:
31   //   MaxHeap<T> heap(a);
32   //   a = heap.sort();
33   public T[] sort() {
34     int n = this.size_;
35     for(int i = 0; i < n; i++) {
```

```
36        removeMax();
37      }
38      return this.tree_;
39    }
40
41    @Override
42    public void insert(T item) {
43      if ((this.size_ + 1) <= this.tree_.length) {
44        grow();
45      }
46      int ins_index = this.size_++;
47      this.tree_[ins_index] = item;
48      if ( ! less (this.tree_[ins_index], this.tree_[parent(ins_index)]) ) {
49        this.swim(ins_index);
50      }
51    }
52
53    @Override
54    public T removeMax() {
55      T res = this.tree_[0];
56      exchange(0, --this.size_);
57      sink(0);
58      return res;
59    }
60
61    public T remove(T item) {
62      for (int i = 0; i < this.size_; i++) {
63        if (this.tree_[i].equals(item)) {
64    exchange(i, --this.size_);
65    if (this.swim(i) == i) {
66      this.sink(i);
67    }
68    return this.tree_[this.size_];
69        }
70      }
71      return null;
72    }
73
74    private int swim(int start) {
75      if ((start > 0) && ( ! less(this.tree_[start], this.tree_[parent(start)])) ) {
76        exchange(start, parent(start));
77        return swim(parent(start));
78      }
79      return start;
80    }
81
82    private int sink(int start) {
83      if (start < this.size_) {
84        int left = left_son(start);
85        int right = right_son(start);
86        int max_son = left;
87        if ((right < this.size_) && less(this.tree_[left], this.tree_[right])) {
88    max_son = right;
89        }
90        if ((left < this.size_) && ! less(this.tree_[max_son], this.tree_[start])) {
91    exchange(max_son, start);
92    return sink(max_son);
93        }
94      }
95      return start;
96    }
97
98    @Override
99    public int size() {
100     return this.size_;
```

```
101   }
102
103   private void exchange(int i, int j) {
104     T z = this.tree_[i];
105     this.tree_[i] = this.tree_[j];
106     this.tree_[j] = z;
107   }
108
109   private static int parent(int i) {
110     return (i-1)/2;
111   }
112
113   private static int left_son(int i) {
114     return 2*i+1;
115   }
116
117   private static int right_son(int i) {
118     return 2*(i+1);
119   }
120
121   private void grow() {
122     resize(2*this.tree_.length);
123   }
124
125   private void resize(int capacity) {
126     @SuppressWarnings("unchecked")
127     T[] newtree = (T[]) new Comparable[capacity];
128     for (int i = 0; i < this.size_; i++) {
129       newtree[i] = this.tree_[i];
130     }
131     this.tree_ = newtree;
132   }
133
134   private class HeapIterator implements Iterator<T> {
135     private int next = 0;
136     @Override
137     public boolean hasNext() {
138       return (this.next < Heap.this.size_);
139     }
140     @Override
141     public T next() {
142       return Heap.this.tree_[this.next++];
143     }
144     @Override
145     public void remove() {
146       // TODO Auto-generated method stub
147     }
148   }
149
150   @Override
151   public Iterator<T> iterator() {
152     return new HeapIterator();
153   }
154
155   private void printSubTree(int root, StringBuffer buf, String indent) {
156     if (root < this.size_) {
157       buf.append(indent);
158       buf.append(this.tree_[root].toString());
159       buf.append(System.getProperty("line.separator"));
160       printSubTree(left_son(root), buf, indent+"␣␣");
161       printSubTree(right_son(root), buf, indent+"␣␣");
162     }
163   }
164
165   @Override
```

```
166     public String toString() {
167       StringBuffer buf = new StringBuffer();
168       printSubTree(0,buf,"");
169       return buf.toString();
170     }
171   }
```

```
1   package priorityqueue;
2
3   public class MaxHeap<T extends Comparable<T>> extends Heap<T> {
4     public MaxHeap() {
5       super();
6     }
7
8     public MaxHeap(int capacity) {
9       super(capacity);
10    }
11
12    public MaxHeap(T[] array) {
13      super(array);
14    }
15
16    @Override
17    protected boolean less(T k1, T k2) {
18      return (k1.compareTo(k2) < 0);
19    }
20
21    public static void main(String [] args) {
22      Heap<String> h = new MaxHeap<String>();
23      h.insert("Cecile");
24      h.insert("Frederic");
25      h.insert("Christophe");
26      h.insert("Dominique");
27      h.insert("Yolaine");
28      h.insert("Marc-Antoine");
29      h.insert("Safouan");
30
31      System.out.println(h.toString());
32      System.out.println();
33      while (h.size() > 0) {
34        System.out.println("## " + h.removeMax());
35        System.out.println(h.toString());
36      }
37
38      Integer [] data = {9, 6, 1, 5, 2, 4, 3, 7, 8};
39      MaxHeap<Integer> mh = new MaxHeap<Integer>(data);
40      data = mh.sort();
41      for(Integer i : data) {
42        System.out.print(i);
43        System.out.print(" ");
44      }
45    }
46  }
```

```
1   package priorityqueue;
2
3   public class MinHeap<T extends Comparable<T>> extends Heap<T> {
4     public MinHeap() {
5       super();
6     }
7
8     public MinHeap(int capacity) {
9       super(capacity);
10    }
```

```
11
12    public MinHeap(T[] array) {
13      super(array);
14    }
15
16    @Override
17    protected boolean less(T k1, T k2) {
18      return (k1.compareTo(k2) > 0);
19    }
20
21    public static void main(String [] args) {
22      Heap<String> h = new MinHeap<String>();
23      h.insert("Cecile");
24      h.insert("Frederic");
25      h.insert("Christophe");
26      h.insert("Dominique");
27      h.insert("Yolaine");
28      h.insert("Marc-Antoine");
29      h.insert("Safouan");
30
31      System.out.println(h.toString());
32      System.out.println();
33      while (h.size() > 0) {
34        System.out.println("## " + h.removeMax());
35        System.out.println(h.toString());
36      }
37
38      Integer [] data = {9, 6, 1, 5, 2, 4, 3, 7, 8};
39      MinHeap<Integer> mh = new MinHeap<Integer>(data);
40      data = mh.sort();
41      for(Integer i : data) {
42        System.out.print(i);
43        System.out.print(" ");
44      }
45    }
46  }
```

Chapitre 8

Algorithmes de recherche

Problème de la recherche d'une information dans des collections comme des bases de données, le web, les documents présents sur un disque dur etc.

La recherche d'une donnée particulière (*value*) se fait à partir d'une clef (*key*). La structure qui permet de retrouver la valeur associée à une clef est appelée *table de symboles*, *dictionnaire* ou encore *index*. Nous examinerons deux techniques classiques pour réaliser des tables de symboles : les arbres binaires de recherche et les tables de hachage.

8.1 Tables de symboles

Une table de symboles permet de mémoriser des associations entre clefs et valeurs. L'opération d'insertion ajoute une paire clef-valeur à la table. L'opération de recherche permet de retrouver la valeur associée à une clef. On cherche à structurer les données de manière à rendre efficaces les opérations d'insertion et de recherche.

8.1.1 Exemples d'utilisation

Application	Fonction	Clef	Valeur
dictionnaire	trouver une définition	mot	définition
index	trouver des pages	mot	liste de pages
système de fichier	trouver un fichier	chemin d'accès	données du fichier
téléphonie	trouver la cellule	numéro de téléphone	cellule dans laquelle se trouve le téléphone

8.2 Tableaux associatifs

Un tableau associatif est une table de symboles dans laquelle toutes les clefs sont distinctes. Si on place dans la table une valeur à une clef qui était déjà présente, la nouvelle valeur remplace l'ancienne. Une telle table se comporte donc comme un tableau, les indices entiers étant remplacés par les clefs.

```
1  package search;
2  import java.util.Iterator;
3
4  public interface AssociativeArray<K,V> {
5    public abstract void put(K key, V value);
6    public abstract V get(K key);
7    public int size();
8    public Iterator<K> keys();
9    public abstract long nbComparison();
10 }
```

8.3 Implémentation par liste

```
1  package search;
2  import java.util.Iterator;
```

```
3
4  public class ListAssocArray<K, V> implements AssociativeArray<K, V> {
5
6    private class Node {
7      public K key;
8      public V value;
9      public Node next;
10     public Node(K key, V value, Node next) {
11       this.key = key;
12       this.value = value;
13       this.next = next;
14     }
15   }
16
17   private Node first_;
18   private int size_;
19   private long compares_;
20
21   public ListAssocArray() {
22     this.first_ = null;
23     this.size_ = 0;
24     this.compares_ = 0;
25   }
26
27   @Override
28   public void put(K key, V value) {
29     Node current = search(key);
30     if (current == null) {
31       this.first_ = new Node(key, value, this.first_);
32       this.size_ ++;
33     } else {
34       current.value = value;
35     }
36   }
37
38   @Override
39   public V get(K key) {
40     Node current = search(key);
41     if (current != null) {
42       return current.value;
43     }
44     return null;
45   }
46
47   private Node search(K key) {
48     for (Node current = this.first_; current != null; current = current.next) {
49       this.compares_++;
50       if (key.equals(current.key)) {
51         return current;
52       }
53     }
54     return null;
55   }
56
57   @Override
58   public Iterator<K> keys() {
59     return new KeyIterator();
60   }
61
62   @Override
63   public int size() {
64     return this.size_;
65   }
66
67   private class KeyIterator implements Iterator<K> {
```

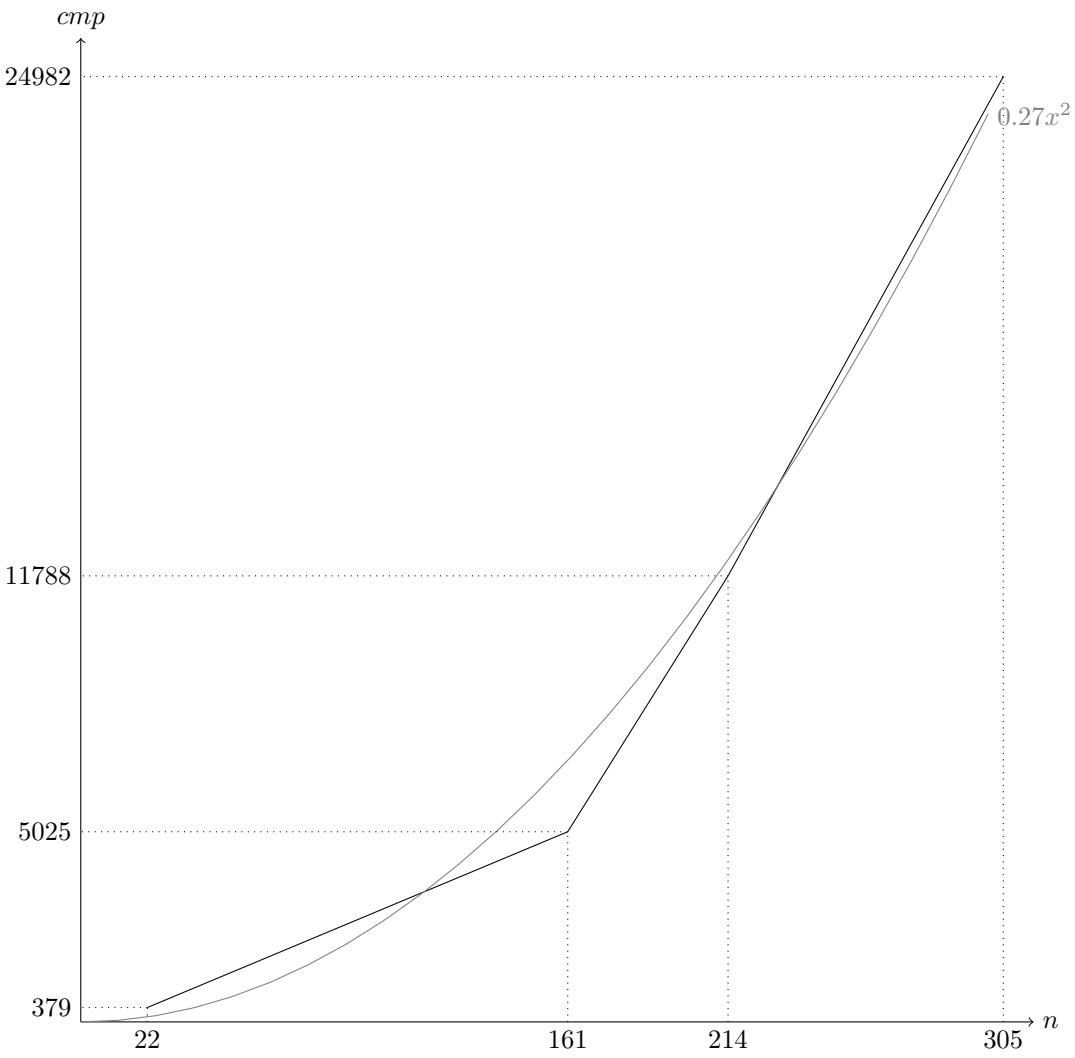
```
68      private Node next_;
69
70      public KeyIterator() {
71        this.next_ = ListAssocArray.this.first_;
72      }
73
74      @Override
75      public boolean hasNext() {
76        return this.next_ != null;
77      }
78
79      @Override
80      public K next() {
81        K key = this.next_.key;
82        this.next_ = this.next_.next;
83        return key;
84      }
85
86      @Override
87      public void remove() {
88        // No remove
89      }
90    }
91
92    @Override
93    public long nbComparison() {
94      return this.compares_;
95    }
96  }
```

8.3.1 Complexité

Lors de l'insertion d'un symbole dans la table, il faut le comparer aux k symboles déjà présents. Le nombre de comparaisons pendant l'insertion de n symboles dans une table initialement vide est donc :

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$$

Lors de la recherche des n symboles, on fait en moyenne $\frac{n}{2}$ comparaisons pour chaque recherche, ce qui donne en moyenne $n\frac{n}{2}$ comparaisons. L'insertion de n symboles suivie de leur recherche a donc une complexité en $\Theta(n^2)$ avec une implémentation par liste.



8.4 Clefs ordonnées, arbre binaires de recherche

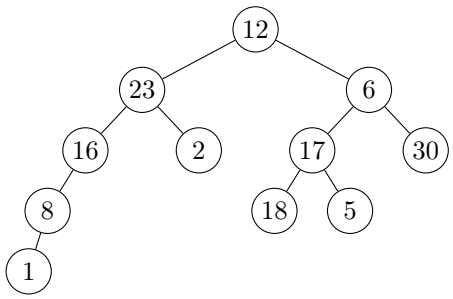
Lorsqu'il existe un ordre sur les clefs, on peut utiliser cet ordre pour diminuer le nombre de comparaisons nécessaires. Une première solution est de conserver les paires clef-valeur dans une liste ordonnée par clefs croissantes. Si la liste est dans un tableau, lors de la recherche, on compare la clef avec celle de l'élément qui se trouve au milieu de la liste (on peut y accéder directement par un calcul sur les indices), puis, si la clef est inférieure, on recommence avec la première moitié de la liste, sinon, on recommence avec la deuxième moitié. La recherche d'une clef se fait donc en $\Theta(\log n)$. Par contre, l'utilisation d'un tableau rend l'insertion coûteuse puisqu'il faut déplacer les éléments du tableau pour faire la place au nouvel élément. L'insertion se fait en moyenne en $\Theta(n)$.

Il est possible d'obtenir une complexité en $\Theta(\log n)$ pour la recherche et pour l'insertion en utilisant des arbres binaires de recherche.

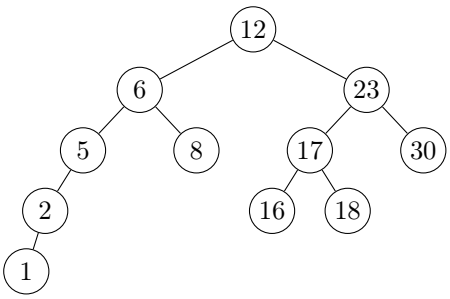
Un arbre binaire de recherche est un arbre binaire tel que pour tout nœud, la valeur de tous les nœuds du sous-arbre gauche est inférieure à celle du nœud, et celle de tous les nœuds du sous-arbre droit est supérieure.

La recherche d'un clef dans un arbre binaire de recherche consiste, en partant de la racine, à descendre à gauche si la clef cherchée est inférieure à la valeur du nœud courant, ou à descendre à droite dans le cas contraire. On s'arrête quand on trouve un nœud portant la clef cherchée ou quand on atteint une feuille. La complexité est donc de l'ordre de grandeur de la hauteur de l'arbre, soit $\Theta(\log n)$ en moyenne.

De même, pour l'insertion, on recherche la clef en partant de la racine, et si on ne la trouve pas, on ajoute un nouveau nœud comme fils gauche ou fils droit de la racine selon que la clef est inférieure ou supérieure à celle de la feuille atteinte. La complexité de l'insertion est donc en $\Theta(\log n)$ en moyenne.



Un arbre binaire

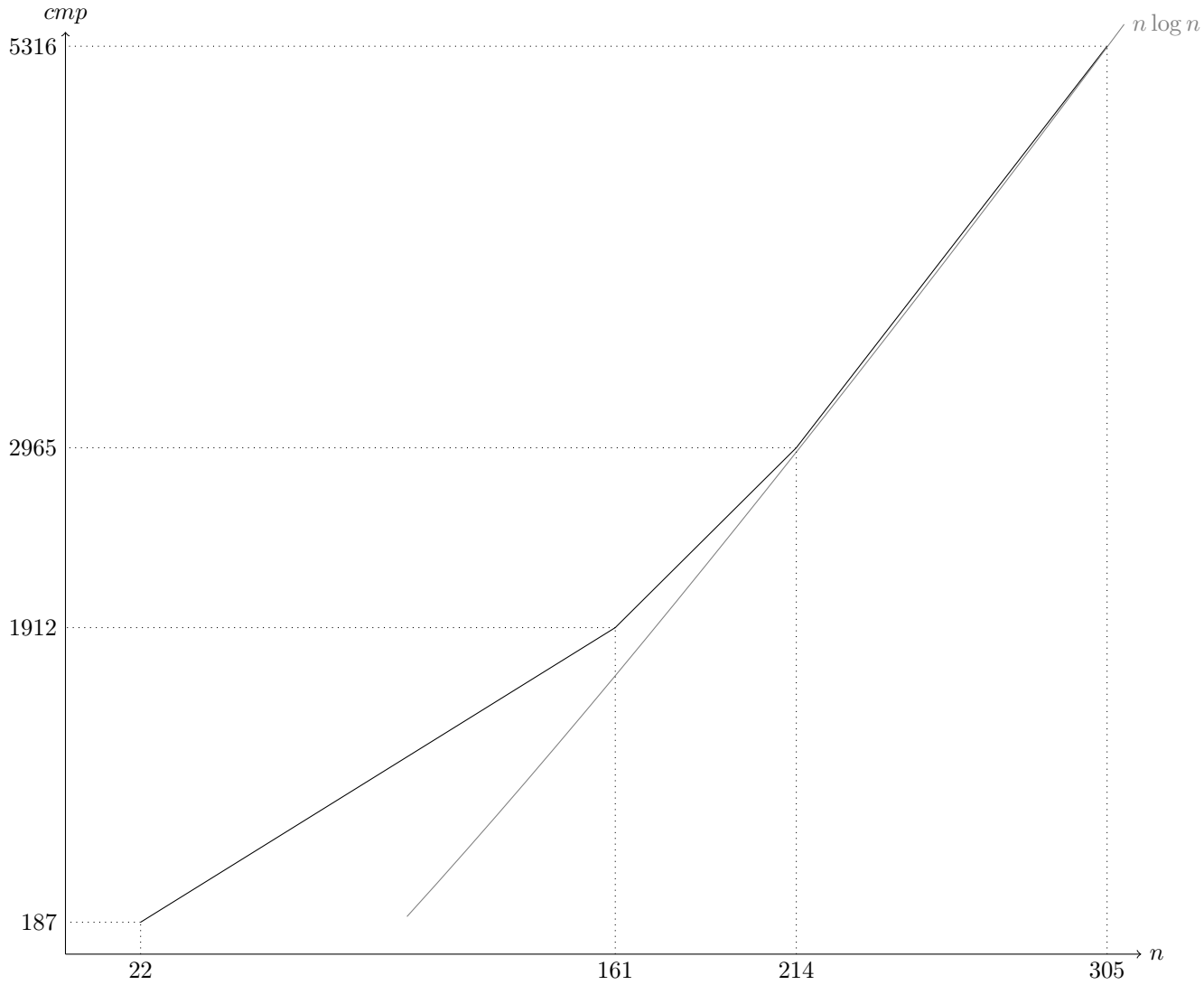


Un arbre binaire de recherche

```
1  package search;
2
3  import java.util.Iterator;
4
5  import tree.BinaryTree;
6  import tree.TreeCursor;
7
8
9  public class BTreeAssocArray<K extends Comparable<K>, V> extends BinaryTree<K>
10                                  implements AssociativeArray<K, V> {
11    private class Node implements TreeCursor<K> {
12      public K key;
13      public V value;
14      public Node lesser;
15      public Node greater;
16
17      public Node(K key, V value, Node lesser, Node greater) {
18        this.key = key;
19        this.value = value;
20        this.lesser = lesser;
21        this.greater = greater;
22      }
23
24      public Node(K key, V value) {
25        this(key, value, null, null);
26      }
27    }
28
29    private Node root_;
30    private int size_;
31    private long compares_ = 0;
32
33    public BTreeAssocArray() {
34      this.root_ = null;
35      this.size_ = 0;
36      this.compares_ = 0;
37    }
38
39    @Override
40    public void put(K key, V value) {
41      Node current = this.root_;
42      Node previous = null;
43      int lastCompare = 0;
44      while ( (current != null) ) {
45        lastCompare = key.compareTo(current.key);
46        this.compares_ ++;
47        if (lastCompare == 0) {
48          current.value = value;
49          return;
50        }
51        previous = current;
52        if (lastCompare < 0) {
53          current = current.lesser;
```

```
54        } else {
55          current = current.greater;
56        }
57      }
58      this.size_ ++;
59      if (previous == null) {
60        this.root_ = new Node(key, value);
61      } else if (lastCompare < 0) {
62        previous.lesser = new Node(key, value);
63      } else {
64        previous.greater = new Node(key, value);
65      }
66    }
67
68    @Override
69    public V get(K key) {
70      Node current = this.root_;
71      while ( (current != null) ) {
72        int res = key.compareTo(current.key);
73        this.compares_ ++;
74        if (res == 0) {
75          return current.value;
76        }
77        if (res < 0) {
78          current = current.lesser;
79        } else {
80          current = current.greater;
81        }
82      }
83      return null;
84    }
85
86    @Override
87    public Iterator<K> keys() {
88      return iterator();
89    }
90
91    @Override
92    public int size() {
93      return this.size_;
94    }
95
96    @Override
97    public TreeCursor<K> root() {
98      return this.root_;
99    }
100
101   @Override
102   public TreeCursor<K> toLeft(TreeCursor<K> c) {
103     return ((Node)c).lesser;
104   }
105
106   @Override
107   public TreeCursor<K> toRight(TreeCursor<K> c) {
108     return ((Node)c).greater;
109   }
110
111   @Override
112   public K getElement(TreeCursor<K> c) {
113     return ((Node)c).key;
114   }
115
116   @Override
117   public long nbComparison() {
118     return this.compares_;
```

```
119   }
120 }
```



Toutefois, lorsque l'arbre n'est pas équilibré, sa hauteur peut être bien supérieure à $\log_2 n$ et atteindre n dans le cas le pire. Nous verrons en TD qu'il est possible de rééquilibrer un arbre binaire grâce à des opérations de rotation, et de considérer des arbres binaires de recherche dont le déséquilibre est borné, ce qui permet à la hauteur d'être de l'ordre de $\log_2 n$ dans tous les cas.

8.5 Tables de hachage

Lorsqu'il n'est pas possible de comparer les clefs, on peut essayer d'obtenir directement l'indice d'un élément dans la table en fonction de sa clef. Si cela était possible, on aurait une recherche et une insertion en $\Theta(1)$. Pour un tableau de taille n, le problème consiste à calculer un entier compris entre 0 et $n-1$ à partir d'une clef, en assurant que deux clefs différentes donnent des indices différents.

Dans la pratique, il suffit de savoir calculer un entier de $[0..n[$ à partir d'une clef, de façon à ce que la probabilité d'une collision, c'est-à-dire que deux clefs donnent le même indice, soit faible. Il faut aussi savoir traiter les collisions.

Une table de hachage est une structure de donnée qui utilise une fonction de hachage pour obtenir l'indice dans un tableau d'un élément à partir de sa clef, et qui permet ainsi la recherche et l'insertion d'éléments avec une complexité en $\Theta(1)$ sous certaines conditions.

8.5.1 Fonctions de hachage

Pour limiter les collisions, la fonction de hachage doit distribuer les indices de façon la plus uniforme possible dans $[0..n[$ pour l'ensemble des clefs. La meilleure technique pour atteindre cet objectif consiste à utiliser chaque

bit d'information de la clef pour calculer le code de hachage. Pour ramener le code dans l'intervalle $[0..n[$, il est commode de prendre le reste modulo n d'un entier calculé à partir de la clef (hachage modulaire). Toutefois, si n n'est pas premier, ceci conduit à ignorer une partie des informations. Par exemple, si $n = 10^4$ et que les clefs sont des dates au format "JJMMAAAA", on ne va utiliser que les quatre derniers chiffres du numéro : toutes les dates d'une même année correspondront au même indice dans le tableau !.

Il est toutefois fréquent que la taille du tableau soit une puissance de 2. Dans ce cas, seuls les k derniers bits du code de hachage seront utilisés, et il est nécessaire de les faire dépendre de tous les bits de la clef pour éviter les collisions.

Exemple : chaînes de caractères

En Java, un caractère est codé sur 16 bits. Pour obtenir un entier sur 32 bits à partir d'une chaîne de caractères, il faut prendre en compte tous ses caractères pour générer les 32 bits du code. Ceci peut se faire en utilisant alternativement les caractères pour modifier les 16 bits de poids faible et les 16 bits de poids fort du code. L'utilisation du ou exclusif permet de ne pas détruire d'information (le OU a tendance à supprimer les bits à 0, le ET à supprimer les bits à 1) :

```
1  package search;
2
3  public class Hash {
4    public static int hashString1(String s) {
5      int h = 0;
6      for (int i = 0; i < s.length(); i++) {
7        h = (h << 16) | ((h >> 16) ^ s.charAt(i));
8      }
9      return h & Integer.MAX_VALUE;
10   }
11
12   public static int hashString2(String s) {
13     int h = 0;
14     for (int i = 0; i < s.length(); i++) {
15       h = (h << 15) ^ ((h >> 16) ^ s.charAt(i));
16     }
17     return h & Integer.MAX_VALUE;
18   }
19
20   public static void main(String [] args) {
21     System.out.println("Max integer = 0x" + Integer.toHexString(Integer.MAX_VALUE));
22     System.out.println("Min integer = 0x" + Integer.toHexString(Integer.MIN_VALUE));
23     System.out.println("Math.abs(Min integer) = " + Math.abs(Integer.MIN_VALUE));
24     System.out.println();
25
26     String [] samples = {
27         "Albert", "Alan", "Alberd",
28         "The brown fox jumps over the lazy dog",
29         "The brown fox jumps over the lady dog"
30     };
31     for (String s : samples) {
32       int h = hashString1(s);
33       System.out.println(s + " : 0x" + Integer.toHexString(h)
34                          + " = " + h % 31 + " mod 31" + " = " + h % 128 + " mod 128");
35     }
36     System.out.println();
37     for (String s : samples) {
38       int h = hashString2(s);
39       System.out.println(s + " : 0x" + Integer.toHexString(h)
40                          + " = " + h % 31 + " mod 31" + " = " + h % 128 + " mod 128");
41     }
42     System.out.println();
43     for (String s : samples) {
44       int h = s.hashCode();
45       System.out.println(s + " : 0x" + Integer.toHexString(h)
46                          + " = " + h % 31 + " mod 31" + " = " + h % 128 + " mod 128");
47     }
```

```
48   }
49 }
```

```
Max integer = 0x7fffffff
Min integer = 0x80000000
Math.abs(Min integer) = -2147483648

Albert : 0x51007d = 8 mod 31 = 125 mod 128
Alan : 0x200002 = 4 mod 31 = 2 mod 128
Alberd : 0x51006d = 23 mod 31 = 109 mod 128
The brown fox jumps over the lazy dog : 0x4b0041 = 29 mod 31 = 65 mod 128
The brown fox jumps over the lady dog : 0x550041 = 18 mod 31 = 65 mod 128

Albert : 0x3fd6205d = 18 mod 31 = 93 mod 128
Alan : 0x20c058 = 14 mod 31 = 88 mod 128
Alberd : 0x3fd6204d = 2 mod 31 = 77 mod 128
The brown fox jumps over the lazy dog : 0x29ff4132 = 0 mod 31 = 50 mod 128
The brown fox jumps over the lady dog : 0x29fcbecd = 2 mod 31 = 77 mod 128

Albert : 0x750b0090 = 23 mod 31 = 16 mod 128
Alan : 0x1f2db8 = 17 mod 31 = 56 mod 128
Alberd : 0x750b0080 = 7 mod 31 = 0 mod 128
The brown fox jumps over the lazy dog : 0x4d5c6786 = 18 mod 31 = 6 mod 128
The brown fox jumps over the lady dog : 0x27d1c9dc = 18 mod 31 = 92 mod 128
```

En java, la classe Object définit une méthode hashCode(). Il est donc possible d'obtenir un code de hachage pour tout objet. Lorsqu'on redéfinit cette méthode dans une classe, il est important de redéfinir equals et compareTo de façon à ce que :

— a.equals(b) soit égal à a.compareTo(b) == 0

— a.equals(b) ⇒ a.hashCode() == b.hashCode()

Par contre, il n'est pas indispensable que des objets différents (au sens de equals) aient des hash codes différents (même si cela est souhaitable pour éviter les collisions).

La méthode hashCode() de Java rend un int qui n'est pas nécessairement positif. Utiliser k.hashCode() % N comme indice dans un tableau de taille N pour mener à une exception si le résultat est négatif. D'autre part, Math.abs(k.hashCode()) peut rendre un résultat négatif. Pour obtenir un indice dans un tableau de taille N à partir d'une clef k, il vaut donc mieux utiliser (k.hashCode() & 0x7FFFFFFF) % N.

8.5.2 Gestion des collisions

Pour des clefs comportants plus de bits que le code de hachage, il est inévitable que plusieurs clefs donnent le même code de hachage et donc le même indice dans le tableau. Il faut donc être capable de gérer ces collisions, c'est-à-dire de stocker et de retrouver des éléments de clefs différentes qui donnent le même indice.

Il existe deux techniques pour traiter ce problème :

— le sondage, qui consiste à rechercher une entrée libre dans le tableau. Par exemple, le sondage linéaire recherche une entrée libre en examinant les entrées suivantes (modulo la taille du tableau) ;

— le chaînage, qui consiste à stocker tous les éléments dont les clefs sont en collision dans une liste.

Dans les deux cas, en supposant que la fonction de hachage répartit les clefs uniformément sur les entrées du tableau, l'accès à un élément se fera avec une complexité constante plus une recherche parmi en moyenne M/N éléments si la table contient M éléments et possède N entrées. On peut donc considérer que ce temps est constant.

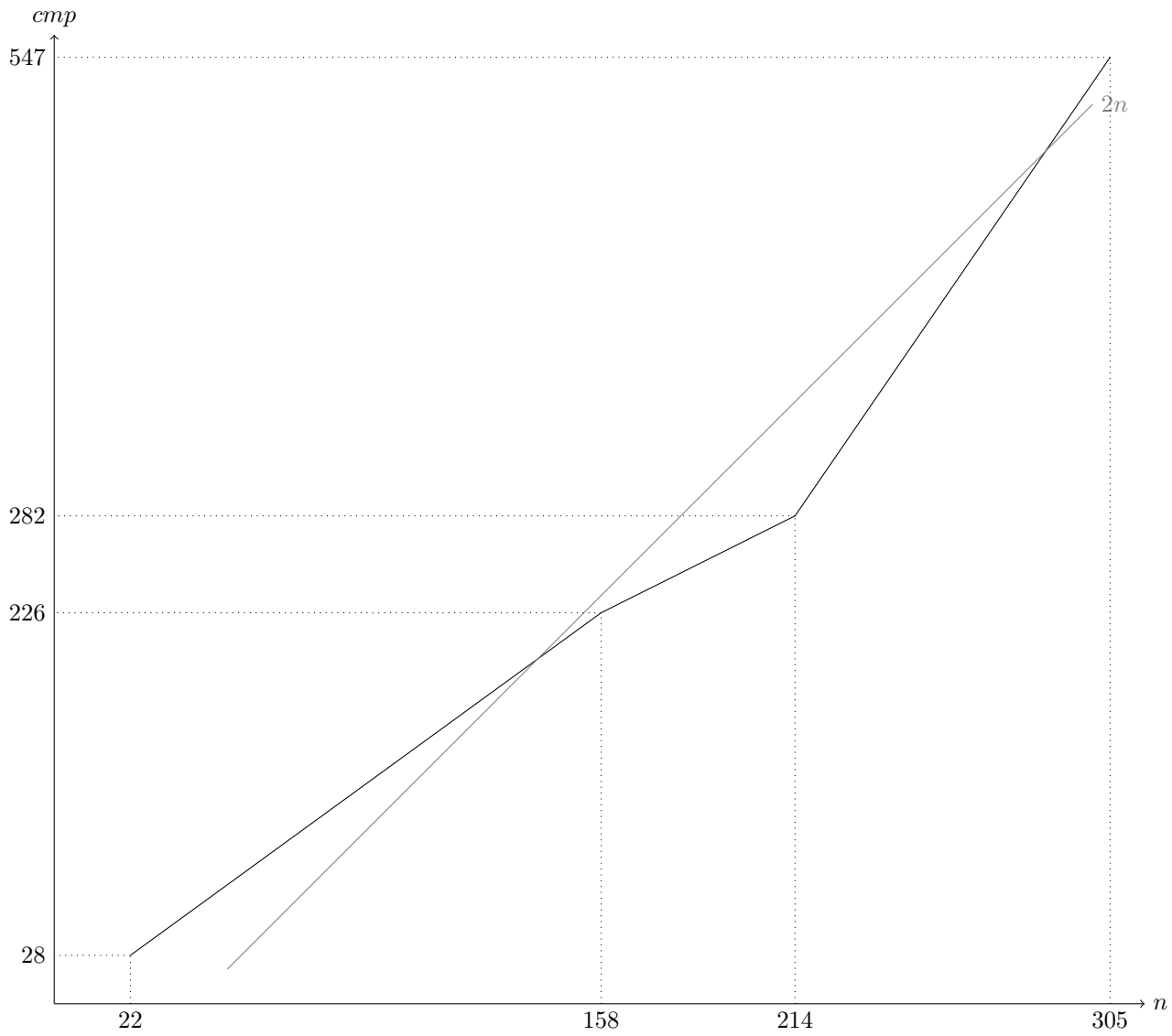
Pour diminuer le temps d'accès, il faut diminuer M/N donc augmenter N. Il y a donc un compromis entre l'espace occupé par la table et le temps mis pour accéder aux éléments. Lorsque la table devient trop pleine (par exemple si $M > 2N$), on aura intérêt à la redimensionner pour conserver de bonnes performances.

```
1  package search;
2  import java.lang.reflect.Array;
3  import java.util.Iterator;
4
5
6  public class HashTable<K, V> implements AssociativeArray<K, V> {
7    private static final int init_size_ = 16;
```

```
8
9   private class Pair {
10    K key;
11    V value;
12    Pair next;
13    Pair(K key, V value, Pair next) {
14      this.key = key;
15      this.value = value;
16      this.next = next;
17    }
18  }
19
20  private Pair [] table_;
21  private int size_;
22  private int nbComparison_;
23
24  public HashTable() {
25    this(init_size_);
26  }
27
28  @SuppressWarnings("unchecked")
29  public HashTable(int tableSize) {
30    this.size_ = 0;
31    this.table_ = (Pair[]) Array.newInstance(Pair.class, tableSize);
32    this.nbComparison_ = 0;
33  }
34
35  @Override
36  public void put(K key, V value) {
37    int idx = (key.hashCode() & Integer.MAX_VALUE) % this.table_.length;
38    Pair p = find(key, idx);
39    if ((p == null) && (value != null)) {
40      this.table_[idx] = new Pair(key, value, this.table_[idx]);
41      this.size_ ++;
42      if (this.size_ > 2*this.table_.length) {
43        grow(2*this.table_.length);
44      }
45    } else {
46      if (value == null) { // put(key, null) = remove(key)
47        p = this.table_[idx];
48        Pair previous = null;
49        while (!key.equals(p.key)) {
50          previous = p;
51          p = p.next;
52        }
53        if (previous == null) {
54          this.table_[idx] = p;
55        } else {
56          previous.next = p.next;
57        }
58      } else {
59        p.value = value;
60      }
61    }
62  }
63
64  private Pair find(K key, int idx) {
65    Pair p = this.table_[idx];
66    while ((p != null) && !key.equals(p.key)) {
67      this.nbComparison_ ++;
68      p = p.next;
69    }
70    return p;
71  }
72
```

```
73    @Override
74    public V get(K key) {
75      int idx = (key.hashCode() & Integer.MAX_VALUE) % this.table_.length;
76      Pair p = find(key, idx);
77      if (p == null) {
78        return null;
79      }
80      return p.value;
81    }
82
83    @Override
84    public int size() {
85      return this.size_;
86    }
87
88    public double averageLength() {
89      double l = 0;
90      for (int i = 0; i < this.table_.length; i++) {
91        Pair p = this.table_[i];
92        while (p != null) {
93          l++;
94          p = p.next;
95        }
96      }
97      return l/this.table_.length;
98    }
99    private void grow(int arraySize) {
100     HashTable<K,V> newTable = new HashTable<K,V>(arraySize);
101     for (int i = 0; i < this.table_.length; i++) {
102       Pair p = this.table_[i];
103       while (p != null) {
104         newTable.putPair(p);
105         p = p.next;
106       }
107     }
108     this.table_ = newTable.table_;
109     this.size_ = newTable.size_;
110   }
111   private void putPair(Pair p) {
112     int idx = (p.key.hashCode() & Integer.MAX_VALUE) % this.table_.length;
113     this.table_[idx] = new Pair(p.key, p.value, this.table_[idx]);
114     this.size_ ++;
115   }
116
117   private class HTIterator implements Iterator<K> {
118     private int curIdx_;
119     private Pair curP_;
120
121     public HTIterator() {
122       this.curIdx_ = 0;
123       this.curP_ = null;
124       nextEntry();
125     }
126     private void nextEntry() {
127       while ((this.curIdx_ < HashTable.this.table_.length)
128         && (HashTable.this.table_[this.curIdx_] == null)) {
129         this.curIdx_ ++;
130       }
131       if (this.curIdx_ < HashTable.this.table_.length) {
132         this.curP_ = HashTable.this.table_[this.curIdx_];
133       }
134     }
135     @Override
136     public boolean hasNext() {
137       return (this.curP_ != null);
```

```
138       }
139       @Override
140       public K next() {
141         if (this.curP_ == null) {
142           return null;
143         }
144         K key = this.curP_.key;
145         this.curP_ = this.curP_.next;
146         if (this.curP_ == null) {
147           this.curIdx_++;
148           nextEntry();
149         }
150         return key;
151       }
152       @Override
153       public void remove() { // Unsupported
154         throw new UnsupportedOperationException();
155       }
156     }
157     @Override
158     public Iterator<K> keys() {
159       return new HTIterator();
160     }
161
162     @Override
163     public long nbComparison() {
164       return this.nbComparison_;
165     }
166
167   }
```


cmp
547
282
226
28
2n
n
22
158
214
305

Chapitre 9

Graphes

Un graphe est une structure de donnée qui comporte des sommets reliés par des arcs ou des arêtes. Lorsque les connexions entre les sommets sont orientées, on parle d'arcs et de graphe orienté. Dans le cas contraire, on parle d'arêtes et de graphe non orienté.

On note $G = (V, E)$ un graphe G ayant les sommets (*vertex/vertices*) V et les arcs ou arêtes (*edges*) E.

9.1 Terminologie

boucle arc ou arête ayant le même sommet à ses deux extrémités

successeur un sommet s' d'un graphe orienté est dit successeur d'un sommet s du même graphe s'il existe un arc reliant s à s' dans ce graphe.

sommets adjacents sommets reliés par une arête ou par un arc.

arcs ou arêtes adjacents arcs ou arêtes possédant une extrémité commune.

degré nombre d'arcs ou d'arêtes ayant ce sommet pour extrémité.

chaîne suite $(s_i)_{1 \leqslant i \leqslant n}$ de sommets telle que pour $\forall i \in [1, n[$, s_i est adjacent à s_{i+1}.

cycle chaîne fermée ($s_1 = s_n$).

chemin chaîne d'un graphe orienté dans laquelle tous les arcs sont orientés de s_i vers s_{i+1}.

circuit chemin fermé.

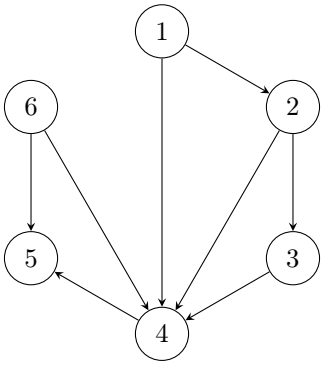
partie connexe d'un graphe ensemble de sommets d'un graphe tel que pour toute paire de sommets de cet ensemble, il existe une chaîne les reliant.

partie fortement connexe d'un graphe ensemble de sommets d'un graphe orienté tel que pour toute paire (u, v) de sommets de cet ensemble, il existe un chemin reliant u à v et un chemin reliant v à u.

Les graphes sont très utilisés dans divers domaines, soit parce qu'ils représentent directement le système (réseaux de transport (métro, routes), réseaux de distribution (électricité, internet)), soit parce qu'ils représentent une relation (dépendances, liens entre pages web etc.).

9.2 Parcours d'un graphe

Selon la structure de donnée file ou pile utilisée, on obtient deux parcours de graphes : le parcours en profondeur d'abord (pré-ordre pour les arbres) et le parcours en largeur d'abord.



```java
package graph;
import java.lang. reflect .Array;

import basictype.Bag;
import basictype.LinkedBag;


public class Graph {
  private Bag<Integer> [] nodes_;

  @SuppressWarnings("unchecked")
  public Graph(int size) {
    this.nodes_ = (Bag<Integer>[]) Array.newInstance(Bag.class, size+1);
    for(int i = 1; i < this.nodes_.length; i++) {
      this.nodes_[i] = new LinkedBag<Integer>();
    }
  }

  public Iterable<Integer> getAdjacents(int n) {
    return this.nodes_[n];
  }

  public void connect(int from, int to) {
    this.nodes_[from].add(to);
  }

  public int size() {
    return this.nodes_.length - 1;
  }

  public Graph invert() {
    Graph inv = new Graph(size());
    for (int i = 1; i <= size(); i++) {
      for (int adj : getAdjacents(i)) {
        inv.connect(adj, i);
      }
    }
    return inv;
  }

  @Override
  public String toString() {
    StringBuffer buf = new StringBuffer();
    for (int i = 1; i < this.nodes_.length; i++) {
      buf.append(i + ":␣");
      for(int adj : this.nodes_[i]) {
        buf.append(adj + "␣");
      }
      buf.append(System.getProperty("line.separator"));
    }
    return buf.toString();
  }
}
```

```java
package graph;
import basictype.LinkedQueue;
import basictype.LinkedStack;
import basictype.Queue;
import basictype.Stack;


public class GraphWalk {
  public static int [] getComponents(Graph g) {
    return new Component(g).components();
```

```
11    }
12    private static class Component {
13      private Graph g;
14      private boolean [] marked;
15      private int [] component;
16      private int num_components;
17
18      public Component(Graph g) {
19        this.g = g;
20        this.marked = new boolean[g.size()+1];
21        this.component = new int[g.size()+1];
22        this.num_components = 0;
23      }
24      public int [] components() {
25        for (int i = 1; i <= this.g.size(); i++) {
26          if (!this.marked[i]) {
27            this.num_components++;
28            findComponents(i);
29          }
30        }
31        return this.component;
32      }
33      private void findComponents(int i) {
34        this.marked[i] = true;
35        this.component[i] = this.num_components;
36        for (int adj : this.g.getAdjacents(i)) {
37          if (!this.marked[adj]) {
38            findComponents(adj);
39          }
40        }
41      }
42    }
43    public static Iterable<Integer> depthFirst(Graph g, int startNode) {
44      return new DFS(g, startNode).dfs();
45    }
46    private static class DFS {
47      private Graph g;
48      private int start;
49      private Queue<Integer> q;
50      private boolean [] marked;
51      public DFS(Graph g, int start) {
52        this.g = g;
53        this.start = start;
54        this.q = new LinkedQueue<Integer>();
55        this.marked = new boolean[g.size()+1];
56      }
57      public Iterable<Integer> dfs() {
58        dfs(this.start);
59        return this.q;
60      }
61      private void dfs(int s) {
62        this.q.enqueue(s);
63        for (int adj : this.g.getAdjacents(s)) {
64          if (!this.marked[adj]) {
65            this.marked[adj] = true;
66            dfs(adj);
67          }
68        }
69      }
70    }
71
72    public static Queue<Integer> breadthFirst(Graph g, int startNode) {
73      boolean [] marked = new boolean[g.size()+1];
74      Queue<Integer> nodes = new LinkedQueue<Integer>();
75      Queue<Integer> order = new LinkedQueue<Integer>();
```

```
76      marked[startNode] = true;
77      nodes.enqueue(startNode);
78      while (nodes.size() > 0) {
79        int current = nodes.dequeue();
80        order.enqueue(current);
81        for (int adj : g.getAdjacents(current)) {
82          if (!marked[adj]) {
83            marked[adj] = true;
84            nodes.enqueue(adj);
85          }
86        }
87      }
88      return order;
89    }
90
91    public static Iterable<Integer> findCycle(Graph g) {
92      return new Cycle(g).findCycle();
93    }
94    private static class Cycle {
95      private Graph g;
96      private boolean[] marked;
97      private boolean[] on_path;
98      private int cycleStart;
99
100     public Cycle(Graph g) {
101       this.g = g;
102       this.marked = new boolean[g.size()+1];
103       this.on_path = new boolean[g.size()+1];
104       this.cycleStart = 0;
105     }
106
107     public Iterable<Integer> findCycle() {
108       Iterable<Integer> cycle = null;
109       for (int i = 1; i <= this.g.size(); i++) {
110         cycle = findCycle(i);
111         if (cycle != null) {
112           break;
113         }
114       }
115       return cycle;
116     }
117
118     private Stack<Integer> findCycle(int from) {
119       Stack<Integer> c = null;
120       this.marked[from] = true;
121       this.on_path[from] = true;
122       for(int adj : this.g.getAdjacents(from)) {
123         if (this.on_path[adj]) {
124           c = new LinkedStack<Integer>();
125           c.push(adj);
126           c.push(from);
127           this.cycleStart = adj;
128           return c;
129         }
130         if (!this.marked[adj]) {
131           c = findCycle(adj);
132           if (c != null) {
133             if (this.cycleStart > 0) {
134               c.push(from);
135               if (this.cycleStart == from) {
136                 this.cycleStart = 0;
137               }
138             }
139             return c;
140           }
```

```
141         }
142       }
143       this.on_path[from] = false;
144       return c;
145     }
146   }
147
148   public static Iterable<Integer> topological(Graph g) {
149     return new Topo(g).topological();
150   }
151
152   private static class Topo {
153     private Graph g;
154     private boolean[] marked;
155     private Stack<Integer> topo;
156
157     public Topo(Graph g) {
158       this.g = g;
159       this.marked = new boolean[g.size()+1];
160       this.topo = new LinkedStack<Integer>();
161     }
162
163     public Iterable<Integer> topological() {
164       for (int i = 1; i <= this.g.size(); i++) {
165         if (!this.marked[i]) {
166           topoSort(i);
167         }
168       }
169       return this.topo;
170     }
171     private void topoSort(int i) {
172       this.marked[i] = true;
173       for (int adj : this.g.getAdjacents(i)) {
174         if (!this.marked[adj]) {
175           topoSort(adj);
176         }
177       }
178       this.topo.push(i); // push node once all its successors have been pushed
179     }
180   }
181 }
```

```
1  package graph;
2  import utils.Operation;
3
4  public class GraphTest {
5    static class NodePrinter implements Operation<Object> {
6      @Override
7      public void performOn(Object item) {
8        System.out.print(item.toString() + "␣");
9
10     }
11
12   }
13
14   public static void main(String[] args) {
15     Graph g = new Graph(6);
16     g.connect(1, 2);
17     g.connect(1, 4);
18     g.connect(2, 3);
19     g.connect(2, 4);
20     g.connect(3, 4);
21     g.connect(4, 5);
22     g.connect(6, 4);
23     g.connect(6, 5);
```

```
24      System.out. println (g. toString ());
25
26      for (int i = 1; i <= g.size(); i++) {
27        System.out. println ("Depth␣first␣from␣" + i);
28        System.out. println (GraphWalk.depthFirst(g, i));
29      }
30
31      for (int i = 1; i <= g.size(); i++) {
32        System.out. println ("Breadth␣first␣from␣" + i);
33        System.out. println (GraphWalk.breadthFirst(g, i));
34      }
35
36      System.out. println ();
37       Iterable <Integer> cycle = GraphWalk.findCycle(g);
38      if (cycle == null) {
39        System.out. println ("No␣cycle␣in␣graph");
40      } else {
41        System.out. println ("Found␣cycle:␣" + cycle. toString ());
42      }
43
44      System.out. println ("Topological␣sort :␣" + GraphWalk.topological(g));
45
46    }
47
48  }
```

```
1: 4 2
2: 4 3
3: 4
4: 5
5:
6: 5 4

Depth first from 1
<1, 4, 5, 2, 3<
Depth first from 2
<2, 4, 5, 3<
Depth first from 3
<3, 4, 5<
Depth first from 4
<4, 5<
Depth first from 5
<5<
Depth first from 6
<6, 5, 4<
Breadth first from 1
<1, 4, 2, 5, 3<
Breadth first from 2
<2, 4, 3, 5<
Breadth first from 3
<3, 4, 5<
Breadth first from 4
<4, 5<
Breadth first from 5
<5<
Breadth first from 6
<6, 5, 4<

No cycle in graph
Topological sort: :6, 1, 2, 3, 4, 5|
```

Les parcours permettent de trouver les composantes connexes d'un graphe. Dans cet exemple, le graphe est connexe (le parcours à partir du sommet 6 passe par tous les sommets), mais il n'est pas fortement connexe (il n'y a pas de chemin de 1 vers 6 par exemple).

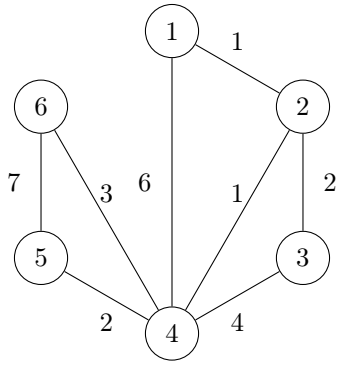
Le parcours en profondeur d'abord correspond à la recherche d'un chemin dans un labyrinthe. On avance le plus loin possible à partir d'une intersection en marquant chaque intersection où l'on passe, et en déroulant un fil derrière soit. Ici, le fil est la pile des sommets déjà visités.

Le parcours en largeur permet de trouver les plus courts chemins (en nombre d'arcs ou d'arête) à partir d'un sommet.

9.3 Graphes valués

Dans un graphe valué, chaque arc ou arête possède un poids ou valuation.

On note $G = (V, E, \omega)$ un graphe valué G ayant les sommets (*vertex/vertices*) V, les arcs ou arêtes (*edges*) E et la fonctions de valuation $\omega : E \to D$, D étant le domaine des valeurs des arcs ou arêtes.




```
1  package graph;
2
3  import java.lang. reflect .Array;
4
5  import search.HashTable;
6
7
8  public class VGraph extends Graph {
9    private HashTable<Integer, Integer> [] arc_values_;
10
11   @SuppressWarnings("unchecked")
12   public VGraph(int size) {
13     super( size );
14     this.arc_values_ = (HashTable<Integer, Integer>[]) Array.newInstance(
15                                 HashTable.class, size+1);
16     for(int i = 1; i < this.arc_values_.length; i++) {
17       this.arc_values_[i] = new HashTable<Integer, Integer>();
18     }
19   }
20   @Override
21   public void connect(int from, int to) {
22     this.connect(from, to, 0);
23   }
24
25   public void connect(int from, int to, int value) {
26     super.connect(from, to);
27     this.arc_values_[from].put(to, value);
28   }
29
30   public int getValue(int from, int to) {
31     return this.arc_values_[from].get(to);
32   }
33
34   public void setValue(int from, int to, int value) {
```

```
35      this.arc_values_[from].put(to, value);
36    }
37
38    @Override
39    public VGraph invert() {
40      VGraph inv = new VGraph(size());
41      for (int i = 1; i <= size(); i++) {
42        for (int adj : getAdjacents(i)) {
43          inv.connect(adj, i, getValue(i, adj));
44        }
45      }
46      return inv;
47    }
48
49    @Override
50    public String toString() {
51      StringBuffer buf = new StringBuffer();
52      for (int i = 1; i <= size(); i++) {
53        buf.append(i + ":␣");
54        for(int adj : getAdjacents(i)) {
55          buf.append(adj + "(" + getValue(i, adj) + ")␣");
56        }
57        buf.append(System.getProperty("line.separator"));
58      }
59      return buf.toString();
60    }
61  }
```

9.3.1 Arbre recouvrant minimal

Pour un graphe connexe, un arbre recouvrant minimal (*Minimum Spanning Tree*) est un arbre (graphe sans cycle) contenant tous les sommets du graphe et dont la somme des coûts des arêtes est minimale.

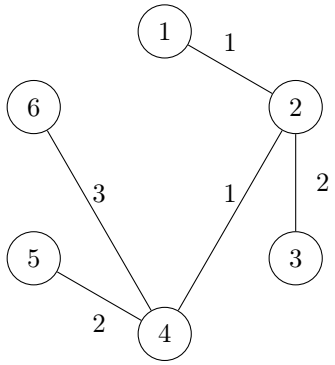
La recherche d'un arbre recouvrant de coût minimal est utilisée pour optimiser ceratines structures : distribution de puissance électrique à un ensemble d'abonnés, nombre minimal de lignes aériennes permettant de relier des aéroports, ressemblances les plus fortes dans l'ensemble des visages reconnus sur des photos...

L'algorithme de Kruskal permet de trouver un arbre recouvrant minimal d'un graphe G à n sommets comme suit :

1. on crée un graphe valué contenant tous les sommets de G, mais sans aucune arête ;
2. on ajoute une à une $n-1$ arêtes de G à ce graphe en prenant à chaque fois l'arête de poids minimal qui connecte deux parties connexes distinctes (on n'introduit pas de cycle).

$n-1$ arêtes suffisent puisqu'on construit un arbre ayant n nœuds. S'il y a moins de $n-1$ arêtes dans G, c'est qu'il n'est pas connexe, et on ne peut alors pas trouver d'arbre recouvrant pour G.

La complexité de cet algorithme est en $\Theta(E \log E)$ où E est le nombre d'arêtes (*edges*) du graphe. En effet, obtenir les E arêtes en ordre croissant se fait en $E \log E$ en utilisant un tas.




```
1  package graph;
2  import priorityqueue.MinHeap;
3
```

```
4  public class Kruskal {
5    private class Edge implements Comparable<Edge> {
6      public int from;
7      public int to;
8      public int weight;
9
10     public Edge(int from, int to, int weight) {
11       this.from = from;
12       this.to = to;
13       this.weight = weight;
14     }
15
16     @Override
17     public int compareTo(Edge e) {
18       return this.weight - e.weight;
19     }
20   }
21   private VGraph g_;
22   private MinHeap<Edge> edges_;
23   private int [] components_;
24
25   public Kruskal(VGraph g) {
26     this.g_ = g;
27     this.components_ = new int[g.size()+1];
28     for (int i = 0; i <= g.size(); i++) {
29       this.components_[i] = i;
30     }
31     this.edges_ = new MinHeap<Edge>();
32     for (int i = 1; i <= g.size(); i++) {
33       for (int adj : g.getAdjacents(i)) {
34         this.edges_.insert(new Edge(i, adj, g.getValue(i, adj)));
35       }
36     }
37   }
38   private boolean connected(int u, int v) {
39     return this.components_[u] == this.components_[v];
40   }
41   private void merge(int u, int v) {
42     int min = u;
43     int max = v;
44     if (v < min) {
45       min = v;
46       max = u;
47     }
48     for (int i = 0; i <= this.g_.size(); i++) {
49       if (this.components_[i] == max) {
50         this.components_[i] = min;
51       }
52     }
53   }
54
55   public VGraph mst() {
56     VGraph mst = new VGraph(this.g_.size()) ;
57     // A tree has at most n-1 edges for n vertices
58     for (int i = 1; i < this.g_.size(); i++) {
59       if (this.edges_.size() > 0) {
60         Edge e = this.edges_.removeMax();
61         if (!connected(e.from, e.to)) {
62           mst.connect(e.from, e.to, e.weight);
63           merge(e.from, e.to);
64         }
65       } else {
66         throw new Error("No␣MST␣in␣a␣disconnected␣graph!");
67       }
68     }
```

```
69      return mst;
70    }
71  }
```

9.4 Chemins les plus courts

Dans un graphe valué, on peut s'intéresser à la recherche des chemin les plus courts à partir d'un sommet, non pas en nombre d'arêtes, mais en somme des coûts des arêtes. Une application de ce problème est la recherche du trajet le plus rapide, le plus court, ou ayant le moins de correspondances dans un réseau de transport.

Trouver les chemins les plus courts dans un graphe à partir d'un sommet consiste à construire un arbre dont ce sommet est la racine et tel que seul le chemin le plus court du graphe est conservé dans l'arbre.

```
1   package graph;
2   import priorityqueue.Heap;
3   import priorityqueue.MinHeap;
4
5   //import java.util.PriorityQueue;
6
7   public class Dijkstra {
8     private class Node implements Comparable<Node> {
9       public int num;
10
11      public Node(int num) {
12        this.num = num;
13      }
14      @Override
15      public boolean equals(Object o) {
16        if (Node.class.isInstance(o)) {
17          return this.num == ((Node)o).num;
18        } else {
19          return false;
20        }
21      }
22      @Override
23      public int compareTo(Node o) {
24        return Dijkstra.this.distTo[this.num] - Dijkstra.this.distTo[o.num];
25      }
26      @Override
27      public int hashCode() {
28        return this.num;
29      }
30    }
31    private VGraph g;
32    private int origin;
33    private int[] distTo;
34    private int[] nodeFrom;
35  //  private PriorityQueue<Node> heap;
36    private Heap<Node> heap;
37
38    public Dijkstra(VGraph g, int origin) {
39      this.g = g;
40      this.origin = origin;
41      this.distTo = new int[g.size()+1];
42      this.nodeFrom = new int[g.size()+1];
43      for (int i = 0; i <= g.size(); i++) {
44        this.distTo[i] = Integer.MAX_VALUE;
45        this.nodeFrom[i] = 0;
46      }
47      this.distTo[origin] = 0;
48  //    this.heap = new PriorityQueue<Node>();
49  //    this.heap.add(new Node(origin));
50      this.heap = new MinHeap<Node>();
51      this.heap.insert(new Node(origin));
```

```
52    }
53
54    public VGraph shortestPath() {
55      VGraph path = new VGraph(this.g.size());
56      while (this.heap.size() > 0) {
57  //      Node n = this.heap.poll();
58        Node n = this.heap.removeMax();
59        relax(n.num);
60      }
61      for (int i = 1; i <= this.g.size(); i++) {
62        if (i == this.origin){
63          continue;
64        }
65        if (this.distTo[i] < Integer.MAX_VALUE) {
66          path.connect(this.nodeFrom[i], i, this.distTo[i]);
67        }
68      }
69      return path;
70    }
71
72    private void relax(int vertex) {
73      for(int adj : this.g.getAdjacents(vertex)) {
74        int w = this.g.getValue(vertex, adj);
75        if (this.distTo[adj] > this.distTo[vertex] + w) {
76          this.distTo[adj] = this.distTo[vertex] + w;
77          this.nodeFrom[adj] = vertex;
78          Node n = new Node(adj);
79  //          if (this.heap.contains(n)) {
80  //            this.heap.remove(n);
81  //          }
82  //          this.heap.add(n);
83          this.heap.remove(n);
84          this.heap.insert(n);
85        }
86      }
87    }
88  }
```

Chapitre 10

Conclusion