Sommaire

RAPPELS **2**
1. Description et implémentation d'un algorithme 2
2. Récursivité 3
3. Structures de données 5

CHAPITRE 1 : COMPLEXITE ALGORITHMIQUE **6**
1. Généralités 6
2. Modèle d'analyse de complexité 6
3. Mise en œuvre 7
4. Différentes formes de complexité 10
5. Complexité polynomiale et complexité exponentielle 11

CHAPITRE 2 : ALGORITHMES DE TRI **12**
1. Présentation 12
2. Tri par sélection 12
3. Tri par insertion 13
4. Tri à bulles 14
5. Tri fusion 14
6. Tri rapide 16

CHAPITRE 3 : LES ARBRES **18**
1. Introduction 18
2. Arbre n-aire 18
3. Arbre binaire 22
4. Arbres binaires particuliers 24

CHAPITRE 4 : LES GRAPHES **30**
1. Définition 30
2. Représentation des graphes 31
3. Parcours des graphes 32
4. Problème des chemins optimaux 33

ANNEXE : LE LANGAGE JAVA **36**

REFERENCES

- Thomas H. Cormen, Algorithmes Notions de base *Collection : Sciences Sup, Dunod,* 2013.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest Algorithmique - 3ème édition - Cours avec 957 exercices et 158 problèmes Broché, Dunod, 2010.
- Rémy Malgouyres, Rita Zrour et Fabien Feschet. *Initiation à l'algorithmique et à la programmation en C : cours avec 129 exercices corrigés*. 2ième Edition. Dunod, Paris, 2011. ISBN : 978-2-10-055703-5.
- Damien Berthet et Vincent Labatut. *Algorithmique & programmation en langage C - vol.1 : Supports de cours*. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.232.
- Damien Berthet et Vincent Labatut. *Algorithmique & programmation en langage C - vol.2 : Sujets de travaux pratiques*. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.258.
- Damien Berthet et Vincent Labatut. *Algorithmique & programmation en langage C - vol.3 : Corrigés de travaux pratiques*. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.217.
- Claude Delannoy. *Apprendre à programmer en Turbo C*. Chihab- EYROLLES, 1994.

RAPPELS

Plan

1. Description et implémentation d'un algorithme
2. Récursivité
3. Structures de données

Intuitivement un algorithme est n'importe quelle procédure de calcul non ambiguë pouvant prendre une ou plusieurs valeurs en entrée et produisant une ou plusieurs valeurs en sortie. C'est un outil pour la résolution d'un problème bien spécifié dont l'énoncé indique la relation entre les entrées et les sorties. L'algorithme peut aussi être défini plus simplement comme une suite finie d'instructions non ambiguës pouvant être exécutées de façon automatique.

Parmi les exemples classiques d'algorithmes, on cite :

- Calcul du produit de deux matrices
- Calcul de la solution d'un système d'équations linéaire
- Calcul du plus court chemin (algorithme de Dijkstra)
- Tri d'une liste
- Recherche d'un mot dans un dictionnaire
- Recherche sur le Web

A titre d'exemple, on défini formellement, l'algorithme de tri comme suit :

Entrée : Une suite de n nombres $\{a_1, a_2, \dots, a_n\}$.
Sortie: Une permutation (ordonnancement) $\{a'_1, a'_2, \dots, a'_n\}$ telle que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
Par exemple, la séquence {1,18,5,4,33} doit être transformée en {1,4,5,18,33} par l'algorithme de tri. La séquence d'entrée est appelée **instance** du problème de tri.

1. DESCRIPTION ET IMPLEMENTATION D'UN ALGORITHME

• Description d'un algorithme

Il y a trois façons de décrire un algorithme :

a- Principe de fonctionnement
Exemple : recherche séquentielle d'un élément dans une liste non triée. Il s'agit de comparer successivement tous les éléments de la liste avec l'élément recherché jusqu'à rencontrer celui-ci ou bien arriver à la fin de la liste.

b- Pseudo code ou formalisme permettant d'exprimer :
- les affectations : ←
- les instructions conditionnelles : **Si** …, **Alors** …, **Sinon**… **Finsi**
- les boucles : **Tant que** condition …..**Fin boucle**

Exemple : Recherche séquentielle dans un tableau

```
// Entrée : n données se trouvant dans un tableau T.
// La donnée recherchée notée clé. On utilise une variable booléenne
// notée trouvé et un entier i.
i ← 1;
trouvé ← faux;
Tant que ((non trouvé) et (i <= n))
          Si (T[i] = clé) Alors trouvé ← vrai;
          Sinon i ← i+1;
Fin boucle
Renvoyer trouvé
```

c- Langage de programmation tel que Python, C, C++ ou Java.

Exemple : Tri par insertion en Java :

```
public void insertionSort()
{ int i, j;
  for (j=1; j<nElems; j++)
  { long temp = A[j];
    i = j;
    while (i>0 && A[i-1]>=temp)
    { A[i] = A[i-1];
      --i;
    }
    A[i] = temp;
  }
}
```

en Python :

```
def insertionSort():
   for j in range(1,nElems):
      temp = A[j]
      i = j
      while i>0 and A[i-1]>=temp:
         A[i] = A[i-1]
         i -= 1
      A[i] = temp
```

Remarques

- Les trois méthodes précédentes de description d'un algorithme sont de plus en plus précises et donc de moins en moins ambiguës.
- Une bonne manière de programmer un algorithme consiste à enchaîner les trois méthodes c'est-à-dire : avant de passer à la programmation il faut donner le principe puis l'exprimer en pseudo code.
- Il est généralement nécessaire de prouver l'algorithme en question c'est-à-dire s'assurer qu'il est correct dans tous les cas. Il existe des techniques de preuve d'algorithmes mais dépassent le niveau de ce cours.

• Implémentation d'un algorithme

L'implémentation d'un algorithme consiste à le traduire dans un langage de programmation et ceci dans le but de l'exécuter sur ordinateur. L'implémentation nécessite le choix de la représentation des données ou la **structure de données**. La résolution d'un problème peut être souvent effectuée par plusieurs algorithmes et pour chaque algorithme il existe plusieurs structures de données. L'algorithme ainsi que la structure de donnée associée à son implémentation est caractérisé par :

- Sa **simplicité** : un algorithme simple est facile à comprendre, à implémenter et généralement à prouver.

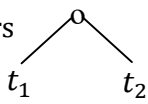
- Son **temps d'exécution** (complexité en temps ou complexité temporelle) : On préfère un algorithme qui s'exécute en une seconde à un autre qui prend une heure sur une même machine. De même, on rejette tout algorithme dont le temps d'exécution dépasse un siècle !!

- Son **requis mémoire** (complexité en mémoire ou complexité spatiale) : cette mémoire dépend à la fois de l'algorithme et des structures des données choisies. Le requis mémoire ne doit jamais dépasser une certaine limite qui dépend de la machine utilisée. Ceci a pour conséquence de mettre une limite sur la taille des problèmes pouvant être résolus.

2. RECURSIVITE

Un objet est dit récursif s'il est constitué en partie ou est défini en terme de lui-même. La récursivité est un moyen particulièrement puissant dans les définitions mathématiques. Quelques exemples bien connus sont :

- La fonction Factorielle $n!$ (définie pour les entiers non négatifs) :
  (a) $0! = 1$
  (b) Si $n > 0$, alors $n! = n \times (n-1)!$

- Les structures d'arbres :
  (a) o est un arbre (dit arbre vide)
  (b) si t_1 et t_2 sont des arbres, alors  est un arbre.

Les algorithmes récursifs sont principalement appropriés lorsque le problème à résoudre ou la fonction à calculer, ou la structure de données à traiter sont déjà définis en termes récursifs. Ils sont à la base du célèbre paradigme de programmation « Divide and Conquer ».

En général, un algorithme récursif $\mathcal{A}$ peut être exprimé comme une composition C d'instructions de base S_i (ne contenant pas $\mathcal{A}$) et de $\mathcal{A}$ lui-même.

$$\mathcal{A} \equiv \mathrm{C}[S_i, \mathcal{A}]$$

Récursivité directe et indirecte

Si une procédure $\mathcal{P}$ contient une référence explicite à elle-même, on parle alors de récursivité directe. Si une procédure $\mathcal{P}$ contient une référence explicite à une autre procédure $\mathcal{T}$ qui contient une référence (directe ou indirecte) à $\mathcal{P}$, on parle dans ce cas de récursivité indirecte.

Récursivité et variables locales

Si une procédure récursive contient des variables locales, à chaque appel récursif de cette procédure, un nouvel ensemble de ces variables locales est créé ; même si elles ont le même nom, leurs adresses mémoire et leurs valeurs sont différentes.

Terminaison des algorithmes récursifs

Comme pour les instructions répétitives (Boucles), la possibilité de calculs interminables (boucle infinie) existe dans les procédures récursives. C'est pour cette raison qu'une procédure récursive $\mathcal{P}$ est soumise généralement à une condition $\mathcal{B}$, qui doit être à un moment donné non satisfaite. Le schéma général des algorithmes récursifs peut donc être exprimé plus précisément comme suit :

$$\mathcal{A} = if\ (\mathcal{B})\ \mathrm{C}[S_i, \mathcal{A}]$$

Quand ne pas utiliser la récursivité ?

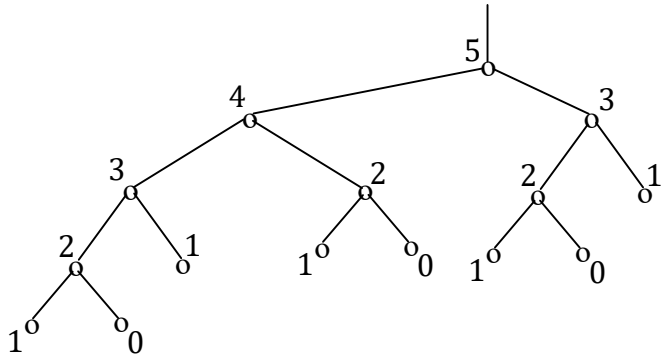
Les algorithmes récursifs sont particulièrement appropriés lorsque le problème sous-jacent où les données à traiter sont définis en termes récursifs. Cela ne signifie pas, toutefois, qu'une définition récursive du problème et/ou des données qui conduisent à un algorithme récursif, est la meilleure façon de résoudre le problème. Un exemple classique est le calcul de la suite de Fibonacci qui est définie comme suit :

$$Fib_0 = 0, Fib_1 = 1 \quad et \quad Fib_n = Fib_{n-1} + Fib_{n-2} \quad pour\ n \geq 2$$

Une approche récursive directe et naïve conduit à la procédure suivante :

```
public static int  Fib(int n)
{ if (n==0) return 0;
  else if (n==1) return 1;
       else return Fib(n-1)+Fib(n-2); }
```

Le Calcul de Fib_n par un appel Fib(n) provoque une activation récursive de la fonction. Mais combien de fois ? Nous remarquons que chaque appel avec $n > 1$ mène à 2 autres appels, le nombre total d'appels augmentera ainsi de façon exponentielle. Un tel programme est clairement impraticable. La figure suivante montre qu'il faut 15 appels de la fonction récursive Fib(n) pour calculer Fib_5.



Cependant, il est clair que les nombres de Fibonacci peuvent être calculés par une méthode itérative qui permet d'éviter le recalcul des mêmes valeurs par l'utilisation de variables auxiliaires tels que $x = Fib_i$ et $y = Fib_{i-1}$.

```
i=1; x=1; y=0;
while (i<n) {
 z=x;
 i++;
 x=x+y;
 y=z;}
```

Cet exemple ne devrait pas cependant conduire les programmeurs à se dérober de la récursivité à n'importe quel prix. Il existe plusieurs applications intéressantes à la récursivité comme vont le montrer les sections et chapitres suivants.

D'un autre coté, chaque algorithme récursif peut être transformé en un algorithme itératif. Toutefois, cela implique la manipulation explicite d'une pile, qui peut rendre plus difficile la compréhension du programme.

Règles de base pour la conception d'algorithmes récursifs

Lors de l'écriture de routines récursives, il est essentiel de garder à l'esprit les trois règles de base suivantes :

1. **Cas de base** : vous devez toujours avoir certains cas, de base, qui peuvent être résolus sans récursivité.
2. **Progression** : les appels récursifs doivent toujours progresser vers les cas de base.
3. **Ne jamais refaire le travail**, en résolvant la même instance d'un problème dans des appels récursifs séparés.

3. STRUCTURES DE DONNEES

Les structures de données spécifient la manière de **représenter les données** d'un problème qui peut être résolu par ordinateur à l'aide d'un **algorithme**. Le choix d'une structure de données doit prendre en considération la taille mémoire nécessaire à son implémentation ainsi que sa facilité d'accès. Une structure de données peut être choisie indépendamment du langage de programmation qui est utilisé pour l'écriture du programme manipulant ses données. Ce langage est supposé offrir, d'une façon ou d'une autre, les mécanismes nécessaires pour définir et manipuler ces structures de données. Les langages de programmation modernes permettent d'attribuer et manipuler la mémoire disponible de la machine sous forme de variables isolées les unes des autres, sous forme de tableaux, de pointeurs indiquant la localisation dans la mémoire de l'objet qui nous intéresse ou à l'aide de structures prédéfinies plus complexes.
Dans le cas des variables isolées ou des tableaux statiques, la place en mémoire est attribuée par le compilateur et ne peut faire l'objet de modification pendant l'exécution du programme. Cependant dans le cas de tableaux dynamiques ou listes chaînées, l'allocation de la mémoire nécessaire est effectuée pendant l'exécution et par conséquent, elle peut augmenter ou diminuer selon le besoin.

Les structures de données classiques peuvent être classées en trois catégories distinctes :

- les **structures linéaires** ou **séquentielles** : appelées ainsi parce que les données sont organisées sous forme d'une liste les unes derrières les autres. Ce sont des structures qui peuvent être représentées par des listes linéaires telles que les tableaux ou des listes chaînées ayant une seule dimension. Dans cette catégorie, il y a lieu de citer les **piles**, pour lesquelles les données peuvent être **ajoutées ou supprimées** à partir d'une même extrémité ; et les **files** où les données peuvent être **ajoutées à partir d'une extrémité** tandis qu'elles sont **supprimées de l'autre extrémité**.
- les **structures arborescentes** et en particulier les arbres binaires.
- les **structures relationnelles** qui prennent en compte des relations existant ou non entre les entités qu'elles décrivent.

CHAPITRE 1 : COMPLEXITE ALGORITHMIQUE

Plan

1. Généralités
2. Modèle d'analyse de complexité
3. Mise en œuvre
4. Différentes formes de complexité
5. Complexité polynomiale et complexité exponentielle

1. GENERALITES

L'étude de la complexité des algorithmes a pour objectif l'estimation du coût d'un algorithme (assorti d'une structure de donnée). Cette mesure permet la comparaison de deux algorithmes sans avoir à les programmer. La complexité peut être étudiée sous deux aspects : temporelle et spatiale.

• Complexité temporelle

Si l'on prend en compte pour l'estimation de la complexité les ressources de la machine telles que la fréquence d'horloge, le nombre de processeurs, le temps d'accès disque etc., on se rend compte immédiatement de la complication voir l'impossibilité d'une telle tâche. Pour cela, on se contente souvent d'estimer la **relation entre la taille des données et le temps d'exécution**, et ceci **indépendamment de l'architecture utilisée**.

On définit la complexité d'un algorithme comme étant la mesure du nombre d'opérations élémentaires qu'il effectue sur un jeu de données. La complexité est exprimée comme une fonction de la taille du jeu de données.

L'analyse de la complexité consiste généralement à mesurer **asymptotiquement** le temps requis pour l'exécution de l'algorithme sur une machine selon un modèle de calcul. Cette mesure permet juste d'avoir une idée imprécise mais **très utile** sur le temps d'exécution de l'algorithme en question. Elle nous permet, entre autres, d'estimer la **taille des instances** pouvant être traitées sur une machine donnée en un temps raisonnable. Cependant, l'analyse de la complexité peut s'avérer très difficile même pour des algorithmes relativement simples nécessitant parfois des outils mathématiques très poussés.

D'une façon formelle, on peut aussi définir la complexité d'un algorithme $\mathcal{A}$ comme étant tout **ordre de grandeur** du nombre d'opérations élémentaires effectuées pendant le déroulement de $\mathcal{A}$. Ces notions seront définies au fur et à mesure.

• Complexité spatiale

La complexité spatiale représente, quant à elle, l'utilisation mémoire que va nécessiter l'algorithme. Celle-ci peut aussi dépendre comme pour la complexité temporelle de la taille de l'instance à traiter par l'algorithme.

On abordera essentiellement dans ce chapitre la complexité temporelle ou en temps.

2. MODELE D'ANALYSE DE COMPLEXITE

Il s'agit d'un modèle simplifié qui tient compte des ressources technologiques ainsi que leurs coûts associés. On prendra comme référence un modèle de machine à accès aléatoire et à processeur unique où les opérations sont exécutées l'une après l'autre sans opérations simultanées. Suivant le type d'instruction, la complexité est évaluée comme suit :

a.Séquence

$\mathcal{A}$: | $\mathcal{J}$;
| $\mathcal{K}$; $\Rightarrow T_{\mathcal{A}}(N) = T_{\mathcal{J}}(N) + T_{\mathcal{K}}(N)$

b. Alternative

$\mathcal{A}$: | **Si** $\mathcal{C}$ **Alors** $\mathcal{J}$ **Sinon** $\mathcal{K}$ $\Rightarrow T_{\mathcal{A}}(N) = T_{\mathcal{C}}(N) + max\{T_{\mathcal{J}}(N), T_{\mathcal{K}}(N)\}$: Pire cas

$\Rightarrow T_{\mathcal{A}}(N) = T_{\mathcal{C}}(N) + min\{T_{\mathcal{J}}(N), T_{\mathcal{K}}(N)\}$: Meilleur cas

c. Boucles

$\mathcal{A}$: nb ↻ $\mathcal{B}$ $\Rightarrow T_{\mathcal{A}}(N) = nb \times T_{\mathcal{B}}(N)$

d. Récursivité

Etablir la formule de récurrence et en déduire la complexité.

Opérations élémentaires

On appelle **opérations élémentaires** les opérations suivantes :

- un accès mémoire pour lire ou écrire la valeur d'une variable ou d'une case d'un tableau ;
- une opération arithmétique entre entiers ou réels telle que l'addition, soustraction, multiplication, division ou calcul du reste d'une division entière ;
- une comparaison entre deux entiers ou réels.

Exemple : L'instruction « c ←a + b ; » nécessite les quatre opérations élémentaires suivantes :

1- un accès mémoire pour la lecture de la valeur de a,
2- un accès mémoire pour la lecture de la valeur de b,
3- une addition de a et b,
4- un accès mémoire pour l'écriture de la nouvelle valeur de c.

Remarque

- Il est recommandé de repérer les opérations fondamentales d'un algorithme. Leur nombre intervient principalement dans l'étude de la complexité. Voici quelques exemples :

Algorithme	Opérations fondamentales
Recherche d'un élément	Comparaisons
Tri	Comparaisons et déplacements (permutations)
Multiplication de matrices	Addition et multiplications

- Faire attention aux boucles et à la récursivité ; la complexité finale d'un algorithme provient généralement de ces deux types d'instructions.

3. MISE EN ŒUVRE

3.1. Exemple d'illustration : Le Tri par insertion

Considérons l'algorithme de tri par insertion permettant de trier un tableau A de taille N. Considérons le modèle de complexité qui associe à chaque ligne i (du code) son coût (ou temps d'exécution) C_i. L'analyse peut se faire donc comme suit :

```
public static void triInsertion (int[]A,int N) // coût      Nombre de fois
    {                                          // ==========================
    int j, temp, i;
    for (j=2 ; j<=N ; j++)                     // coût C1 ........ N
       {
         temp = A[j];                          // coût C2 ........ N-1
         i = j-1;                              // coût C3 ........ N-1
         while (i>0 && A[i]>temp)              // coût C4 ........ 2+3+...+N
           {
             A[i+1]=A[i];                      // coût C5 ........ 1+2+...N-1
             i=i-1;                            // coût C6 ........ 1+2+...N-1
           }
         A[i+1]=temp;                          // coût C7 ........ N-1
      }
    }
```

3.2. Rappel des principales suites

- **Suite arithmétique :**

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

- **Suite géométrique :**

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

- **Suite harmonique :**

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = Ln(n)$$

La complexité de l'algorithme de tri par insertion est donc :

$$T(N) = (\frac{C_4 + C_5 + C_6}{2})N^2 + (C_1 + C_2 + C_3 + \frac{C_4 - C_5 - C_6}{2} + C_7)N - (C_2 + C_3 + C_4 + C_7)$$

Et puisqu'on s'intéresse aux valeurs de N très grandes (comportement asymptotique de la complexité), nous pouvons noter la complexité en utilisant les ordres de grandeurs (voir rappel ci-après) comme suit :

$$\mathbf{T(N) = O(N^2)}$$

Note :
T(N) représente la complexité dans le **pire des cas** car on n'a pas tenu compte dans notre calcul de la valeur de l'expression logique de la condition (A[i] > temp) qui détermine le nombre d'exécutions de la boucle interne. L'algorithme de tri par insertion peut donc prendre moins de temps pour des tableaux ayant une structure particulière et par conséquent, la complexité est une variable aléatoire. C'est pourquoi on parle de **complexité en moyenne** d'un algorithme qu'on peut calculer comme étant l'espérance de cette variable aléatoire.

3.3. Généralités sur les ordres de grandeurs

- Notation Θ (thêta) : soit g une fonction positive d'une variable entière n. **Θ(g(n))** désigne l'ensemble des fonctions positives de la variable n, pour lesquelles il existe deux constantes c_1, c_2 et un entier n_0, satisfaisant la relation :

$$\mathbf{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \forall n \geq n_0}$$

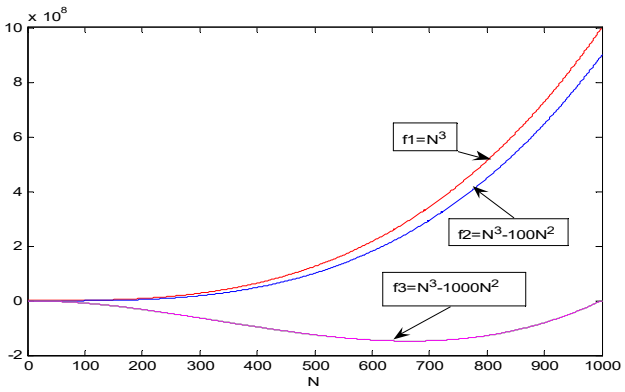
Par abus de notation on écrit f(n)=Θ(g(n)) pour exprimer que f ∈ Θ(g(n)) (malgré que Θ(g(n)) est un ensemble et f est un élément de celui-ci).

Exemple :
Soit $g(n) = n^2$ et $f(n) = 50n^2+10n$.
Il s'agit de trouver c_1,c_2 et n_0 tels que : $c_1n^2 \leq 50n^2+10n \Rightarrow c_1 \leq 50$,
$50n^2+10n \leq c_2n^2 \Rightarrow c_2 \geq 50$.

donc on a bien $f(n) = \Theta(g(n)) = \Theta(n^2)$

La figure suivante montre les graphes de trois fonctions f1, f2 et f3. On peut facilement vérifier que f2 = Θ(f1(N)) et aussi f3 = Θ(f1(N)).

On peut interpréter la relation f(n)=Θ(g(n)) comme suit : pour n assez grand, la fonction f est **bornée à la fois supérieurement et inférieurement** par la fonction g, c'est-à-dire que les fonctions f et g sont égales, à une constante prét.



- Notation O (grand o) : Lorsque la fonction f est bornée **uniquement supérieurement** par la fonction g on utilise la notation **O**. **O(g(n))** désigne donc l'ensemble des fonctions positives de la variable n, pour lesquelles il existe une constante c et un entier n_0, satisfaisant la relation : $\mathbf{0 \leq f(n) \leq c\, g(n) \quad \forall\, n \geq n_0}$

La relation f(n) = O(g(n)) indique que la fonction f est bornée supérieurement par la fonction g pour des valeurs suffisamment grandes de l'argument n. Donc f(n) = Θ(g(n)) implique que f(n) = O(g(n)) (par abus de notation). Formellement, on peut écrire $\Theta(g(n)) \subseteq O(g(n))$.

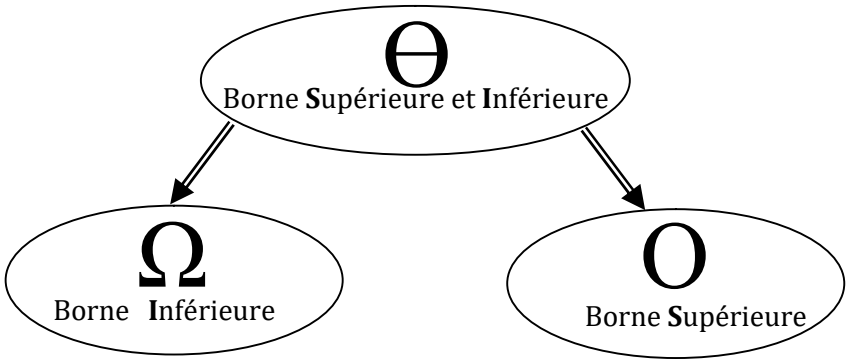
Exemple : $100n^2 + 10n = \mathbf{O}(n^2)$ mais on peut aussi écrire $\mathbf{100n = O(n^2)}$ **!** Car ceci est équivalent à dire que 100n est asymptotiquement bornée supérieurement par n^2.

- Notation Ω (grand oméga): Ω(g(n)) désigne l'ensemble des fonctions positives de la variable n, pour lesquelles il existe une constante c et un entier n_0, satisfaisant la relation : $\mathbf{0 \leq cg(n) \leq f(n) \quad \forall\, n \geq n_0}$

La relation f(n) = Ω(g(n)) indique que la fonction f est bornée inférieurement par la fonction g pour des valeurs suffisamment grandes de l'argument n. Par conséquent, **f(n) = Θ(g(n)) implique que f(n) = Ω(g(n))** (par abus de notation). Formellement, on peut écrire : **$\Theta(g(n)) \subseteq \Omega(g(n))$**. Le théorème suivant découle immédiatement des définitions données :

Théorème : f(n) = **Θ**(g(n)) si et seulement si : f(n) =**Ω**(g(n)) et f(n) = **O**(g(n))

Les trois notations précédentes peuvent être récapitulées par le schéma suivant :



Quelques propriétés de la notation O

1. Les facteurs constants peuvent être ignorés (Ex : $3x^2 = O(x^2)$)
2. Une grande puissance de n croît plus vite qu'une puissance inférieure ($n^r = O(n^s)$ si $0 \leq r \leq s$)
3. Le taux de croissance d'une somme de termes est le taux de croissance du terme le plus rapide en croissance. (Ex : $an^3 + bn^2 = O(n^3)$)
4. Les fonctions exponenielles sont plus rapides en croissance que les puissances (polynômes)
5. Tous les logarithmes ont un même taux de croissance.

3.4. Autres méthodes d'analyse de complexité

La complexité d'un algorithme peut être dans certains cas facilement trouvée si on arrive à analyser et comprendre l'algorithme et repérer ses opérations fondamentales. Ainsi pour le tri par insertion, on peut s'intéresser aux comparaisons comme suit :

Le nombre de comparaisons exécutées à l'itération i est au plus i. Le nombre total de comparaisons est :

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

Pour une fonction récursive, il y a lieu d'écrire la formule de récurrence correspondante puis de déduire la complexité. Ainsi pour la fonction récursive de calcul de n!

```
public static int  Fact(int n)
{ if (n==0) return 1;
  else return n*Fact(n-1); }
```

On obtient la formule de recurrence : $T(0) = O(1)\ \ et\ \ T(n) = T(n-1) + O(1)$

Le développement de cette formule se fait comme suit : $T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(n-p) + p$. Puis on pose $n - p = 0$ et on obtient : $T(n) = n + 1 = O(n)$

3.5. Exercices

1. Analyser la complexité de la recherche séquentielle puis de la recherche dichotomique.

2. Montrer que la routine Tour de Hanoi a une complexité de $O(2^N)$

```
Procedure Hanoï(N, A, C, B)
{ si N≠0 alors
    Hanoï(N-1,A,B,C);
    Déplacer le disque de A vers C
    Hanoï(N-1,B,C,A);
  finsi
}
```

3. Développer la formule de récurrence suivante : $T(n) = T(n/2) + an + b$ et $T(1) = 1$

4. DIFFERENTES FORMES DE COMPLEXITE

Il est évident que la complexité d'un algorithme peut ne pas être la même pour deux jeux de données différents. Ceci peut avoir des conséquences sur le choix du meilleur algorithme pour un problème donné. En pratique, on s'intéresse aux formes suivantes de la complexité :

4.1. Complexité dans le pire des cas (Worst case)

Il s'agit de considérer le plus grand nombre d'opérations élémentaires effectuées sur l'**ensemble de toutes les instances** du problème ; on cherche une borne supérieure qui est atteinte même pour des instances ayant une très faible, voire zéro, probabilité.

Cette forme de complexité est plus simple à calculer mais peut conduire à un choix erroné d'un algorithme. En effet le pire cas peut être un cas très rare !

4.2. Complexité dans le meilleur des cas (Best case)

Il s'agit, cette fois-ci, de considérer le plus petit nombre d'opérations élémentaires ; c'est-à-dire on cherche un minorant de la complexité au lieu d'un majorant.

Cette forme de complexité peut être considérée comme complément de la complexité dans le pire des cas mais n'offre aucune garantie à l'utilisateur.

4.3. Complexité en moyenne (Average case)

Il s'agit de calculer la moyenne (espérance mathématique) des nombres d'opérations élémentaires effectuées sur la totalité des instances. Ce calcul est généralement très difficile et souvent même délicat à mettre en œuvre car il faut connaître la probabilité de chacun des jeux de données pour pouvoir calculer la complexité en moyenne.

Cette forme d'analyse fait actuellement l'objet de nombreux travaux de recherche et permet d'expliquer, d'une part le comportement de certains algorithmes en pratique ; et d'autre part, le choix d'algorithmes pour des problèmes ayant des tailles considérables tels que les problèmes d'apprentissage en intelligence artificielle.

5. COMPLEXITE POLYNOMIALE ET COMPLEXITE EXPONENTIELLE

Un algorithme est considéré pratique, relativement à une classe de problèmes, s'il est capable de résoudre n'importe qu'elle instance (dans le pire des cas) ou en moyenne (complexité moyenne) en temps polynomial c'est-à-dire, sa complexité est $O(N^k)$, où k est un entier quelconque.

En réalité une complexité est dite polynomiale si elle peut être bornée par un polynome, on retrouve donc :

$O(1)$	(Complexité constante)
$O(log(N))$	(Complexité logarithmique)
$O(\sqrt{N})$	(Complexité racinaire)
$O(N)$	(Complexité linéaire)
$O(NlogN)$	(Complexité quasi-linéaire)
$O(N^2)$	(Complexité quadratique)
$O(N^3)$	(Complexité cubique)
…	

D'autre part une complexité est dite exponentielle ($O(k^N)$ en général) si elle ne peut pas être bornée par un polynome, on retrouve par exemple :

$O(N^{logN})$	(Complexité sous exponentielle)
$O(2^N), O(3^N)$	(Complexité exponentielle)
$O(N!)$	(Complexité factorielle)
$O(2^{2^N})$	(Complexité double exponentielle)

CHAPITRE 2 : ALGORITHMES DE TRI

Plan

1. Présentation
2. Tri par sélection
3. Tri par insertion
4. Tri à bulles
5. Tri par fusion
6. Tri rapide

1. PRESENTATION

Le tri est sans doute un des problèmes les plus fondamentaux de l'algorithmique. Après le tri beaucoup d'autres problèmes deviennent facile à résoudre tels que l'unicité ou la recherche.

Le tri consiste à réarranger une liste de n objets de telle manière :

$X_1 \leq X_2 \leq \cdots \leq X_n$: Tri par ordre croissant

$X_1 \geq X_2 \geq \cdots \geq X_n$: Tri par ordre décroissant

Il existe plusieurs méthodes de tri, nous citons :

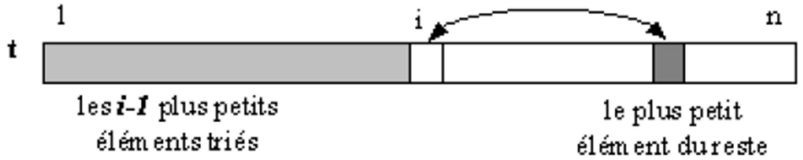
- Tri par sélection
- Tri par insertion
- Tri à bulles
- Tri par fusion
- Tri rapide
- Tri shell
- …

Dans ce qui suit, on décrit les principaux algorithmes de tri puis on analysera leur complexité temporelle.

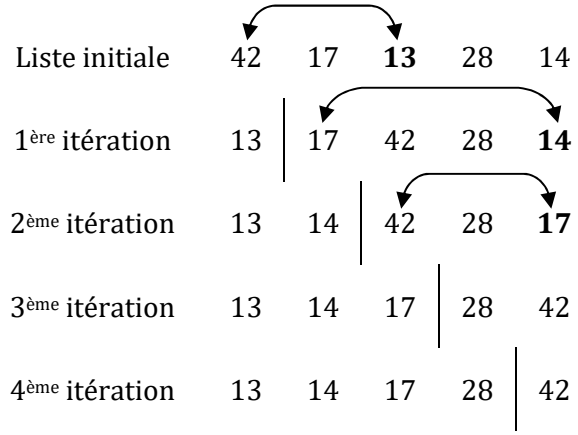
2. TRI PAR SELECTION (SELECTION SORT)

2.1. Principe

Itérativement, le tri par sélection consiste à chercher le plus petit élément puis de le mettre au début.



Exemple



2.2. Implémentation

```
public void selectionSort(int[] A, int n)
{
for (int i=0;i<n-1;i++)
 { int imin=i;
   for (int j=i+1;j<n;j++)
       if (A[j]<A[imin]) imin=j;
   swap(A,i,imin)
 }
}
```

2.3. Complexité

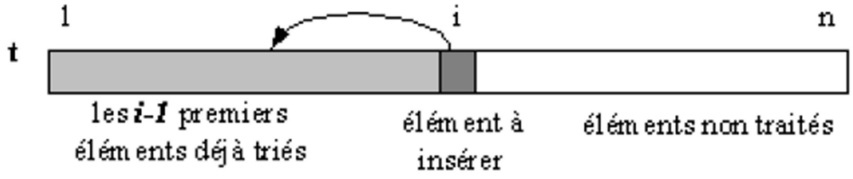
Le pire et le meilleur cas sont pareils, puisque pour trouver le plus petit élément, $(n-1)$ itérations sont nécessaires, pour le 2ème plus petit élément, $(n-2)$ itérations sont effectuées… jusqu'à l'avant dernier plus petit élément qui nécessite 1 itération. Le nombre total d'itérations est donc :

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

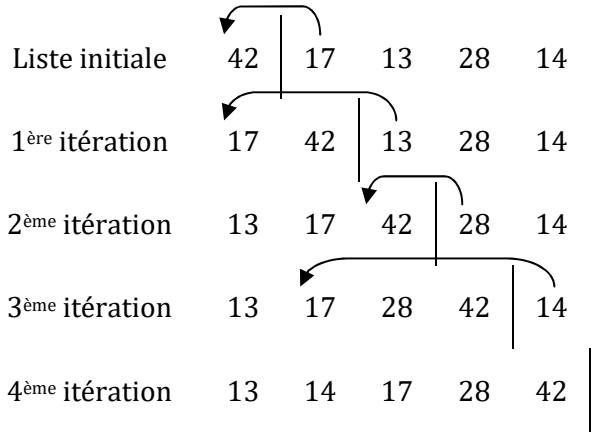
3. TRI PAR INSERTION (INSERTION SORT)

3.1. Principe

Itérativement, on insère le prochain élément dans la partie qui a été déjà triée précédemment. La partie de départ qui est triée est le premier élément.



Exemple



3.2. Implémentation

```
public void insertionSort(int[] A, int n)
{
 for (int i=1;i<n;i++)
 { int temp=A[i],j=i;
   while (j>0 && A[j-1]>temp)
   { A[j]=A[j-1];j--;}
   A[j]=temp;
 }
}
```

3.3. Complexité

Comme nous n'avons pas nécessairement à scanner toute la partie déjà triée, le pire et le meilleur cas sont différents.

Meilleur cas : si le tableau est déjà trié, chaque élément est toujours inséré à la fin de la partie triée ; nous n'avons à déplacer aucun élément. Comme nous avons à insérer $(n-1)$ éléments, chacun générant seulement une comparaison, la complexité est $O(n)$.

Pire cas : si le tableau est inversement trié, chaque élément est inséré au début de la partie triée. Dans ce cas, tous les éléments de la partie triée doivent être déplacés à chaque itération. La ième itération génère (i-1) comparaisons et échanges de valeurs :

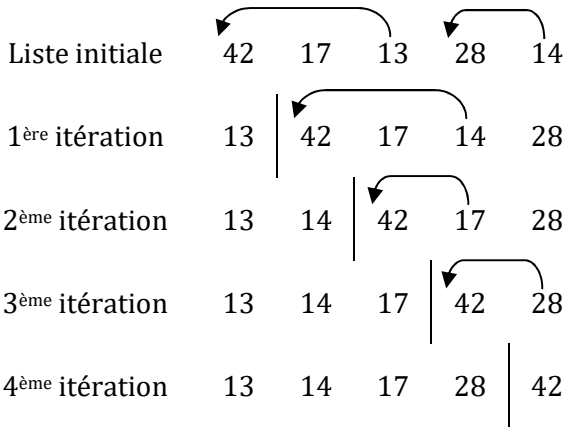
$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2} = O(n^2)$$

4. TRI A BULLES (BUBBLE SORT)

4.1. Principe

Parcourir le tableau en comparant deux à deux les éléments successifs et permuter s'ils ne sont pas dans l'ordre.

Exemple



4.2. Implémentation

```
public void bubbleSort(int[] A, int n)
{
for (int i=0;i<n-1;i++)
  for (int j=n-1;j>i;j--)
    if (A[j]<A[j-1]) swap(A,j,j-1);
}
```

4.3. Complexité

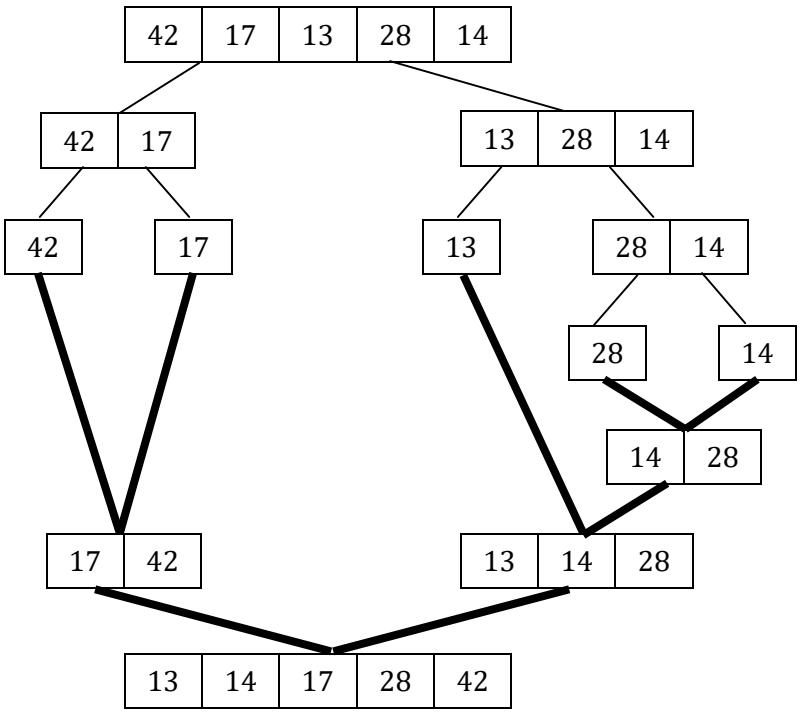
Globalement, c'est la même complexité que le tri par sélection. Le meilleur et le pire cas sont pareils avec une complexité de $O(n^2)$.

5. TRI PAR FUSION (MERGE SORT)

5.1. Principe

Cet algorithme divise en deux parties égales le tableau. Après que ces deux parties soient triées (de manière généralement récursive), elles sont fusionnées pour l'ensemble des données.

Exemple



5.2. Implémentation

```
public void mergeSort(int[] A, int deb , int fin)
{
 int mil;
 if (deb<fin) {
   mil=(deb+fin)/2;        (* DIVIDE *) ..... O(1)
   mergeSort(A,deb,mil);   (* CONQUER *) ..... T(n/2)
   mergeSort(A,mil+1,fin);(* CONQUER *) ..... T(n/2)
   fusion(A,deb,mil,fin); (* COMBINE *) ..... O(n)
  }
}

public void fusion(int[] A, int deb, int mil, int fin)
{
  int n1,n2,i,j; int[] R = new int[50]; int[] L = new int[50];
  n1=mil-deb+1;
  n2=fin-mil;
  for (i=1;i<=n1;i++) L[i]=A[deb+i-1];
  for (j=1;j<=n2;j++) R[j]=A[mil+j];
  L[n1+1]=9999;  //Nombre très grand
  R[n2+1]=9999;
  i=j=1;
  for (int k=deb;k<=fin;k++) {
  if (L[i]<=R[j]) {A[k]=L[i];i++;}
  else {A[k]=R[j];j++;}
 }
}
```

5.3. Complexité

La complexité peut être exprimée par récurrence :

$$T(n) = \begin{cases} O(1) & \text{si } n = 1 \\ 2T(n/2) + O(n) & \text{si } n > 1 \end{cases} \Rightarrow T(n) = O(n log_2 n)$$

Le tableau A sera divisé par 2 jusqu'à obtention de tableaux de taille 1, ainsi :
Taille de A : n , $/2$, $n/4$, ..., $n/2^p$ avec $n/2^p = 1 \Rightarrow p = log_2(n)$

Puisqu'à chaque étape, une (ou plusieurs) opération(s) de fusion de l'ordre $O(n)$ est exécutée sur les sous tableaux obtenus, on obtient donc $O(nlog_2 n)$.

6. TRI RAPIDE (QUICK SORT)

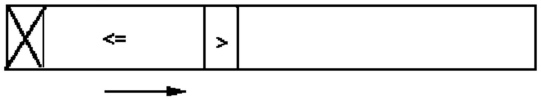
6.1. Principe

L'idée de cet algorithme est de diviser le tableau en deux parties séparées par un élément appelé pivot de telle manière que les éléments de la partie gauche soient tous inférieurs ou égaux à cet élément et ceux de la partie droite soient tous supérieurs à ce pivot. Cette étape fondamentale du tri rapide s'appelle le partitionnement.

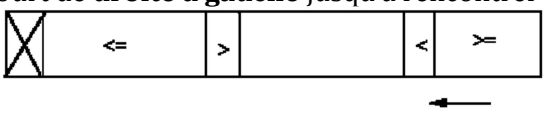
Choix du pivot : Le choix idéal serait que ça coupe le tableau exactement en deux parties égales, mais cela n'est pas toujours possible. On peut prendre le premier ou le dernier ou de manière aléatoire,…

Partitionnement :

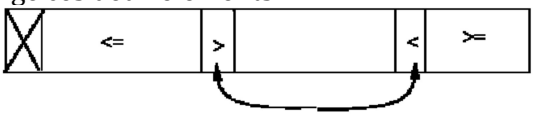
- On parcourt de **gauche à droite** jusqu'à rencontrer un élément **supérieur** au pivot.



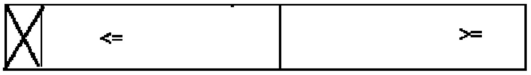
- On parcourt de **droite à gauche** jusqu'à rencontrer un élément **inférieur** au pivot.



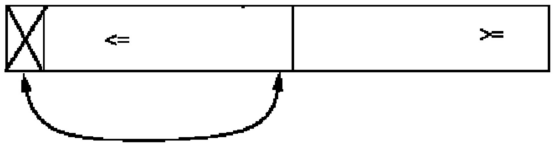
- On échange ces deux éléments.



- On recommence les parcours gauche-droite et droite-gauche jusqu'à avoir :

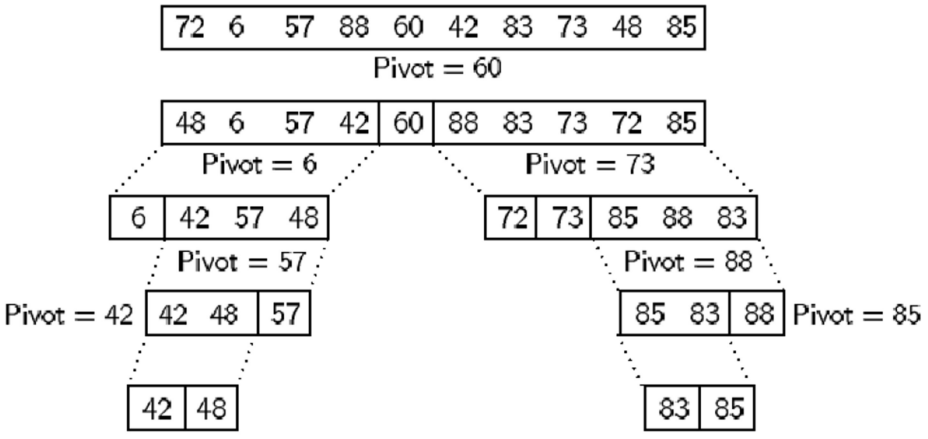


- Il suffit alors de mettre le pivot à la frontière par un échange



Dans ce qui suit (Exemple & implémentation) on choisit l'élément se trouvant au milieu du tableau comme pivot.

Exemple



6.2. Implémentation

```
public void quickSort(int[] A, int deb, int fin)
{
 int ipivot;
 if (deb<fin)
  {
   ipivot=partition(A,deb,fin);
   quickSort(A,deb,ipivot);
   quickSort(A,ipivot+1,fin);
  }
}

public int partition(int[] A, int deb, int fin)
{
 int pivot,aux,i,j;
 pivot=A[(deb+fin)/2];
 i=deb;j=fin;
 while (i<j)
  {
   while (pivot>A[i]) i++;
   while (pivot<A[j]) --j;
   if (i<j) swap(A,i,j);
  }
 return j;
}
```

6.3. Complexité

- **Cas favorable :**

La meilleure chose qui puisse arriver, c'est qu'à chaque fois que la fonction Partition() est appelée, elle divise exactement le tableau (ou le sous-tableau) en 2 parties égales.

À la première passe, les n éléments du tableau sont comparés avec la valeur pivot pour les balancer à la droite ou à la gauche. Il y a donc n comparaisons. À la seconde passe il y a 2 fonctions Partition() qui effectuent leur rôle chacune sur leur moitié de tableau. Chaque fonction doit comparer les $n/2$ éléments du sous-tableau pour effectuer le balancement. Donc de fait, il y a encore n comparaisons pour cette passe. Il en sera de même pour les autres itérations.

Nous pouvons exprimer la complexité sous forme de récurrence :

$$T(n) = \begin{cases} O(1) \; si \; n = 1 \\ 2T\left(\frac{n}{2}\right) + \; O(n) \; si \; n > 1 \end{cases} \Rightarrow T(n) = O(nlog_2 n)$$

- **Cas défavorable :**

La pire chose qui puisse arriver, c'est qu'à chaque appel à la fonction Partition(), à cause des circonstances de la disposition des données, celle-ci place la totalité du sous-tableau à droite ou à gauche (excluant bien sûr l'élément pivot qui est alors à sa place définitive). Dès lors le tri rapide se transforme en un tri à bulles.

À la première passe, il y aura donc n comparaisons. À la seconde passe il y a déjà une valeur ordonnée et un sous-tableau de $n-1$ éléments, il y aura donc $n-1$ comparaisons effectuées par la fonction Partition(). À la 3ème passe, il y aura $n-2$ comparaisons, etc.

Pour effectuer le tri au complet, il aura donc fallu en tout n passes. D'où la complexité:

$$n + (n-1) + \ldots + 2 + 1 \; = \; n(n+1)/2$$

Soit donc O(n^2) = pire cas de Quicksort. Mais il reste que la complexité en moyenne est O($n\log_2 n$).

CHAPITRE 3 : LES ARBRES

Plan

1. Introduction
2. Arbres (n-aires)
3. Arbres binaires
4. Arbres binaires de recherche
5. Les tas

1. INTRODUCTION

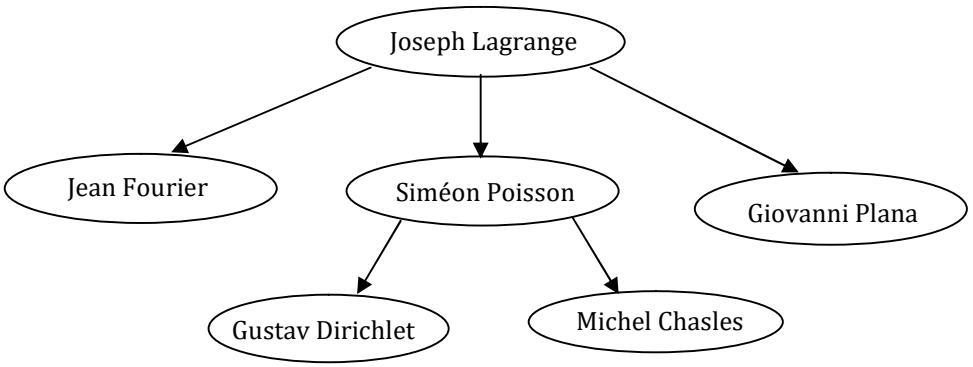
Pour de grandes quantités de données, la complexité linéaire des opérations sur les listes est prohibitive. Nous étudions dans ce chapitre, des structure de données dites hierarchiques ou arborescentes pour qui le temps d'exécution de la plupart des opérations est de l'ordre de $O(logN)$. Nous passons en revue les arbres n-aires et les arbres binaires, puis nous montrons comment utiliser les arbres binaires pour développer un algorithme de recherche avec une complexité en moyenne de $O(logN)$. Nous terminons avec une autre application des arbres binaires à savoir les tas ainsi qu'un nouvel algorithme efficace de tri dit tri par tas.

2. ARBRE N-AIRE

2.1. Définitions

Un arbre est défini **récursivement** comme étant une structure composée d'éléments appelés **nœuds** liés par une relation « Parent/Fils » ; il contient un nœud particulier appelé **racine** (le seul qui n'a pas de nœud parent)**,** ainsi qu'une suite ordonnée (qui peut être vide) **d'arbres** disjoints $A_1, A_2, \ldots, A_m$ appelés **sous-arbres.** Un **nœud** peut prendre n'importe quel type de donnée : simple (numérique, caractère,…) ou composé (structure, classe) etc.

Un exemple typique d'arbre est celui de l'arbre généalogique où les nœuds représentent les personnes et la relation entre nœuds est la relation classique de « parenté ». La figure suivante illustre la schématisation d'une partie de l'arbre généalogique mathématique (http://www.genealogy.ams.org). La relation « parenté » est interprétée comme « étudiant ou doctorant de » :



Il y a lieu de retenir les définitions suivantes relatives aux arbres :

- Les **fils** d'un nœud sont les racines de ses sous arbres, par exemple les « doctorants » de Lagrange sont Fourier, Poisson et Plana.
- Le **père** d'un nœud (excepté la racine) est l'unique nœud dont il est fils.
- Un **nœud interne** est un nœud qui a au moins un fils, par exemple Poisson est un nœud interne.
- Une **feuille** d'un arbre est un nœud qui n'a pas de fils.
- Les **ancêtres** d'un nœud sont les nœuds qui le relient à la racine y compris celle-ci et le nœud lui-même.
- Les **descendants** d'un nœud sont les nœuds qui appartiennent au sous-arbre ayant ce nœud comme racine.
- La **profondeur** d'un nœud est la longueur du chemin reliant celui-ci à la racine.
- La **hauteur** d'un arbre est la profondeur max. de ses nœuds.
- La **taille** d'un arbre est le nombre total de ses nœuds.

2.2. Parcours d'un arbre

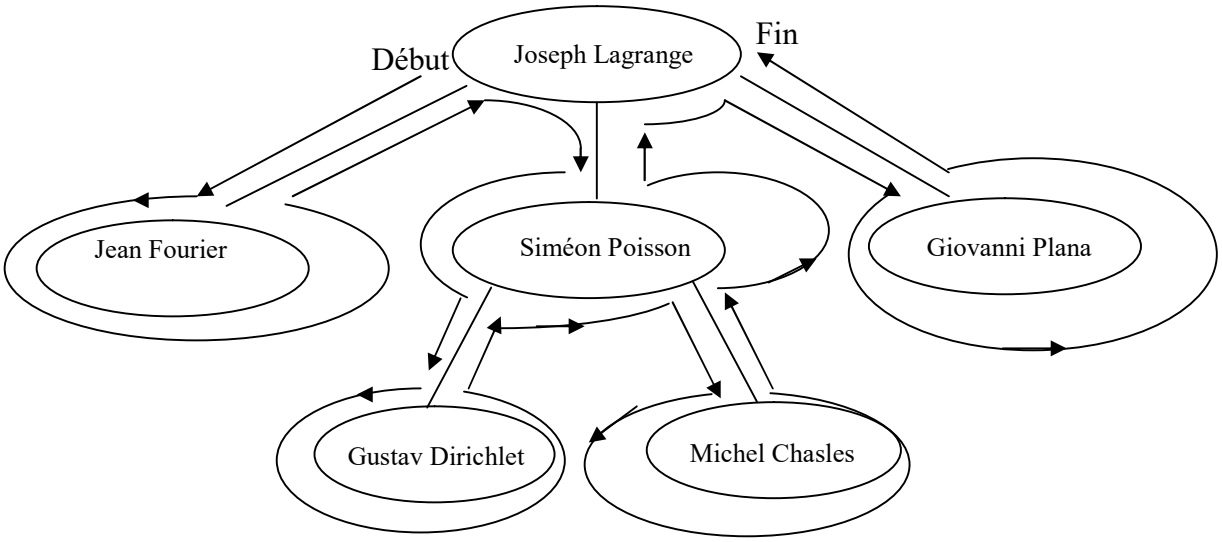
La procédure permettant d'examiner une fois et une seule fois tous les nœuds d'un arbre est appelée parcours d'un arbre. Il y a lieu de citer :

Parcours en préordre (préfixe) « RGD = Racine Gauche Droite »

Il s'agit d'implémenter récursivement les étapes suivantes :

- Examen de la racine
- Parcourir en **préordre** le premier sous-arbre (le plus à gauche)
- …
- …
- Parcourir en **préordre** le dernier sous-arbre (le plus à droite).

Dans l'exemple précédent, on parcourt les nœuds *Lagrange, Fourier, Poisson, Dirichlet, Chasles, Plana*.



Le schéma ci-dessus illustre le parcours en préordre comme étant un examen des nœuds dés la première rencontre uniquement.

Parcours en postordre (postfixe) « GDR = Gauche Droite Racine »

Dans ce type de parcours, il s'agit d'implémenter récursivement les étapes suivantes :

- Parcourir en postordre le premier sous-arbre
- …
- …
- Parcourir en postordre le dernier sous-arbre
- Examiner la racine.

Le parcours en postordre de l'arbre précèdent se fait comme suit : *Fourier, Dirichlet, Chasles, Poisson, Plana, Lagrange*. Cette opération est similaire à la précédente sauf que cette fois-ci l'examen des nœuds se fait à la dernière rencontre.

Parcours par niveaux

Consiste à parcourir l'arbre à partir de la racine (niveau 0), puis les nœuds du niveau 1 de gauche à droite et ainsi de suite.

Le parcours par niveau de l'arbre cité en exemple se fait donc comme suit : *Lagrange, Fourier, Poisson, Plana, Dirichlet, Chasles*.

2.3. Quelques opérations sur les arbres

On peut citer parmi les opérations sur les arbres :

- **initialiserArbre()** : retourne un arbre vide ou nul.
- **parent (n,A)** : fonction qui renvoie le parent d'un nœud n dans l'arbre A. Dans le cas de la racine un nœud nul est renvoyé par la fonction. Ceci servira comme un signal d'arrêt lors de la navigation.
- **filsLePlusAGauche(n,A)** : renvoie le fils le plus à gauche du nœud n dans l'arbre A. Dans le cas d'un nœud qui n'est pas interne, c'est-à-dire n'ayant pas de fils, cette fonction renvoie le nœud nul.
- **filsLePlusADroite(n,A)** : Idem mais renvoie le fils le plus à droite.
- **racine(A)** renvoie le nœud racine. Dans le cas où l'arbre est vide, elle retourne le nœud nul.
- **estFeuille(n,A)** : permet de tester si le nœud n est une feuille.
- **nbFeuilles(A)** : Retourne le nombre de feuilles de l'arbre A.

2.4. Implémentation des arbres

Comme les structures étudiées dans le chapitre précédent, les arbres peuvent être implémentés à l'aide des tableaux et aussi à l'aide de listes chaînées. Chaque implémentation a ses propres avantages ainsi que ses inconvénients quant à sa capacité et son efficacité à implémenter les opérations classiques sur les arbres décrites précédemment.

A l'aide de tableaux

Considérons un arbre A dont les *n* nœuds sont désignés par 0,1,…,n-1. La meilleure structure « en termes d'efficacité » qui supporte l'opération ***Parent (n,A)*** est un tableau A dont l'élément A[i] pointe sur le parent du nœud i. La racine est pointée par le symbole ou pointeur nul. Par conséquent on peut implémenter un arbre quelconque par un tableau A définit comme suit :

A[i] = j si le nœud j est parent du nœud i
A[i] = -1 dans le cas où i est la racine de l'arbre.

Il est clair qu'avec cette représentation l'opération parent est de complexité constante indépendamment de la taille de l'arbre. Cependant, cette représentation complique d'avantage les opérations utilisant l'information sur les fils d'un nœud. En outre, on ne peut pas imposer un ordre sur les fils d'un nœud, en particulier les opérations **FilsLePlusAGauche**, **FilsLePlusADroite** ne peuvent être définies.

Exemple : Si on fait correspondre à chaque nœud de l'arbre précédent un indice ; on obtient :

Nœud	Indice
J. Lagrange	0
J. Fourier	1
S. Poisson	2
G.Plana	3
G. Dirichlet	4
M. Chasles	5

$\Rightarrow$

0	1	2	3	4	5
-1	0	0	0	2	2

A l'aide de tableaux de pointeurs

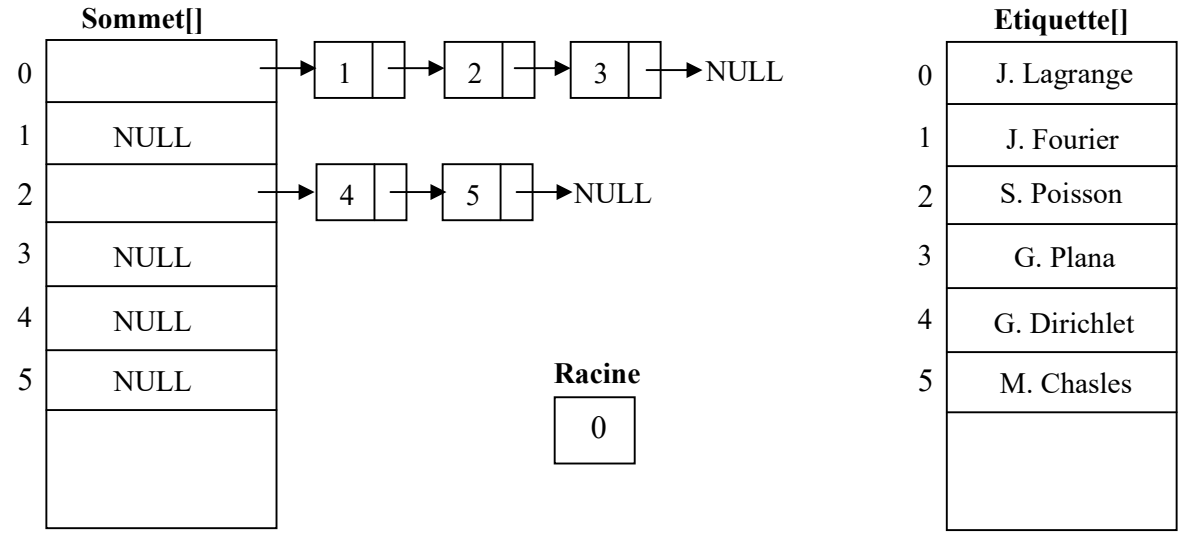
Une façon efficace de représenter un arbre consiste à construire une liste des fils de chaque nœud qui peut être implémentée avec une liste chaînée, car le nombre de fils varie d'un nœud à un autre :

```
const int MaxNoeuds=1000;

struct Bloc {
  int element;
  Bloc *suiv;
};

struct Arbre {
  Bloc *Sommet[MaxNoeuds];
  string Etiquette[MaxNoeuds];
  int Racine;
};
```

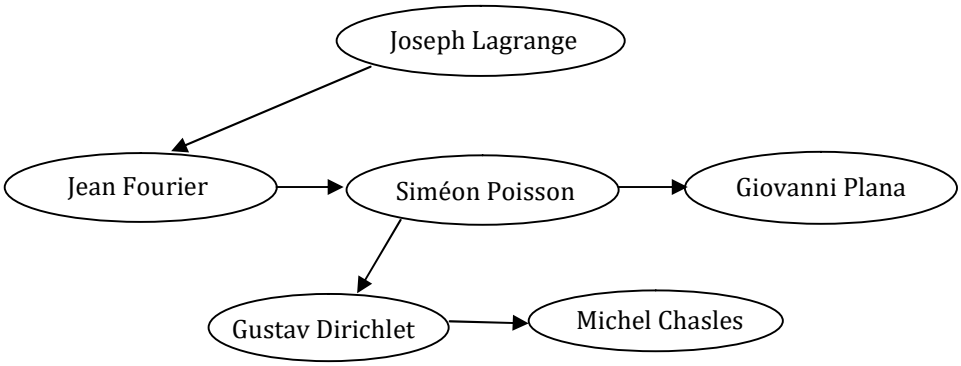
Ainsi l'arbre cité en illustration est représenté comme suit :



Exercice : Proposer une implémentation des routines citées en I.3 en se basant sur cette représentation.

Représentation dynamique

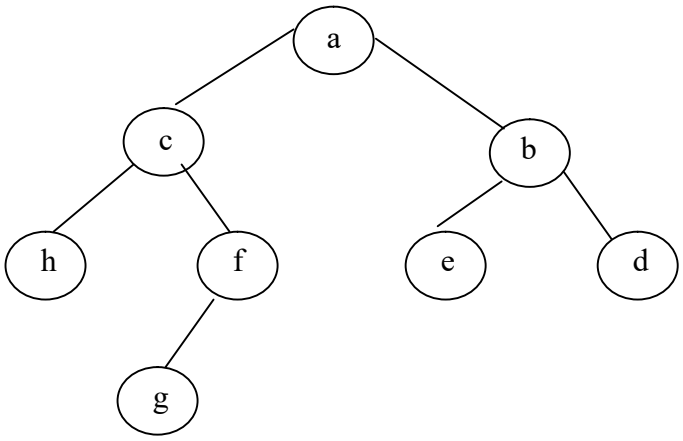
Vu que le nombre de fils peut varier d'un nœud à un autre et de sucroit n'est pas connu à l'avance, une représentation dynamique basée sur un bloc associant la donnée du nœud avec des pointeurs vers tous ses fils serait irréalisable. La solution est simple : Il suffit de garder pour chaque nœud un lien vers le fils le plus à gauche et un lien vers le nœud représentant le prochain « frère » comme suit :




```
struct Noeud {
 int element;
 Noeud *premierFils;
 Noeud *frere;
};
```

3. ARBRE BINAIRE

Un arbre binaire peut être soit vide, soit constitué du nœud racine qui pointe éventuellement vers deux sous arbres. A chaque niveau de l'arbre binaire, les deux sous arbres binaires disjoints sont appelés sous arbre gauche et sous arbre droit. Un arbre binaire désigne souvent un arbre non vide. Les procédures de parcours des arbres n-aires restent valables pour les arbres binaires et la terminologie utilisée est légèrement remaniée pour tenir compte de certaines spécificités des arbres binaires. En effet, lorsqu'on désigne les fils d'un nœud on parle du fils gauche et fils droit dans le cas où l'un deux ou bien les deux existent.



Remarque : Un arbre binaire est dit :

- **homogène** lorsque tous les nœuds ont deux ou zéro fils.
- **dégénéré** si tous ses nœuds n'ont qu'un seul fils.
- **complet** si chaque niveau de l'arbre est complètement rempli.
- **parfait** lorsque tous les niveaux sauf éventuellement le dernier sont remplis, et dans ce cas les feuilles du dernier niveau sont groupées à gauche.

Pour implémenter un arbre binaire, on associe à chaque nœud une structure de donnée contenant la donnée et deux adresses qui pointent vers les deux nœuds fils. Par convention, une adresse nulle indique un arbre (ou sous arbre) vide. De cette façon, il suffit donc de mémoriser l'adresse de la racine.

3.1. Structures de données

A l'aide de tableaux (Statique)

Les arbres binaires peuvent être représentés par la structure suivante :

```
const int MaxNoeuds=1000;

struct Noeud{
 int element;
 int filsGauche;
 int filsDroit;
};

Noeud Arbre[MaxNoeuds];
```

A l'aide de pointeurs (Dynamique)

Au lieu d'utiliser des champs entiers pour pointer vers les fils gauche et droite, on peut faire usage des pointeurs comme suit :

```
struct Noeud {
 int element;
 Noeud *filsGauche, *filsDroit, *parent;
};
```

Ainsi un arbre binaire est représenté par une structure chaînée dans laquelle chaque nœud est un objet. En plus des champs Element (clé) et adresse de la donnée, chaque nœud contient trois champs : FilsGauche, FilsDroit et Parent. Ces champs pointent vers le sous arbre gauche, sous arbre droit et le parent du nœud respectivement dans le cas où ils existent. Dans le cas contraire, un ou plusieurs de ces champs contient la valeur NULL. La racine est le seul nœud dont le champ parent est NULL.

Le pointeur Parent permet de faciliter le parcours dans le sens « feuilles → racine ». Une autre variante consiste à supprimer ce pointeur pour un gain d'espace mémoire et pour simplifier les procédures de mise à jour de l'arbre binaire, mais au détriment du parcours de l'arbre. Ainsi on peut représenter plus simplement l'arbre binaire comme suit :

```
struct Noeud {
 int element;
 Noeud *filsGauche, *filsDroit; };
```

Note : L'arbre binaire est déclaré et initialisé avec l'instruction : **`Noeud *racine=NULL;`**

3.2. Parcours d'un arbre binaire

En plus des parcours en préordre et postordre dèjà étudiés pour les arbres en général, un troisième type de parcours peut être utilisé dans le cas des arbres binaires : Le parcours en ordre (infixe).

⇒ **PARCOURS EN PREORDRE (PREFIXE) : « RGD = RACINE GAUCHE DROITE »**

Il s'agit d'implémenter récursivement les étapes suivantes :

- Examiner la racine
- Parcourir en préordre le sous-arbre gauche
- Parcourir en préordre le sous-arbre droit

Ainsi le parcours en préordre du même arbre binaire donne : a, c, h, f, g, b, e, d.

⇒ **PARCOURS EN ORDRE (INFIXE) : « GRD = GAUCHE RACINE DROITE »**

Il s'agit d'implémenter récursivement les étapes suivantes :

- Parcourir en ordre le sous-arbre gauche
- Examiner la racine
- Parcourir en ordre le sous-arbre droit

Ainsi le parcours en ordre de l'arbre binaire cité en exemple donne : h, c, g, f, a, e, b, d.

⇒ **PARCOURS EN POSTORDRE (POSTFIXE) : « GDR = GAUCHE DROITE RACINE »**

Il s'agit d'implémenter récursivement les étapes suivantes :

- Parcourir en postordre le sous-arbre gauche
- Parcourir en postordre le sous-arbre droit
- Examiner la racine

Le parcours en postordre du même arbre binaire donne : h, g, f, c, e, d, b, a.

Ces parcours peuvent être implementés aisément en utilisant des routines récursives comme suit :

```
void parcoursPrefixe(Noeud *racine)
{ if (racine!=NULL) {
    cout<<racine->element<<endl;
    parcoursPrefixe(racine->filsGauche);
    parcoursPrefixe(racine->filsDroit); }
}

void parcoursInfixe(Noeud *racine)
{ if (racine!=NULL) {
    parcoursInfixe(racine->filsGauche);
    cout<<racine->element<<endl;
    parcoursInfixe(racine->filsDroit); }
}

void parcoursPostfixe(Noeud *racine)
{ if (racine!=NULL) {
    parcoursPostfixe(racine->filsGauche);
    parcoursPostfixe(racine->filsDroit);
    cout<<racine->element<<endl; }
}
```

Exercice : Proposer une implementation itérative de ces 3 parcours.

3.3. Opérations sur les arbres binaires

Voici des détails d'implémentation de quelques opérations sur les arbres binaires en se basant sur la représentation chaînée :

- Tester si un nœud est une feuille :

```
bool estFeuille(Noeud *p)
{ return (p->filsGauche==NULL && p->filsDroit==NULL); }
```

- Taille d'un arbre (nombre de nœuds) :

```
int taille(Noeud *racine)
{ if (racine==NULL) return 0;
  else return 1+taille(racine->filsGauche)+taille(racine->filsDroit);
}
```

- Nombre de feuilles d'un arbre :

```
int nbFeuilles(Noeud *racine)
{ if (racine==NULL) return 0;
  else if (estFeuille(racine)) return 1;
       else return nbFeuilles(racine->filsGauche)+nbFeuilles(racine->filsDroit);
}
```

4. ARBRES BINAIRES PARTICULIERS

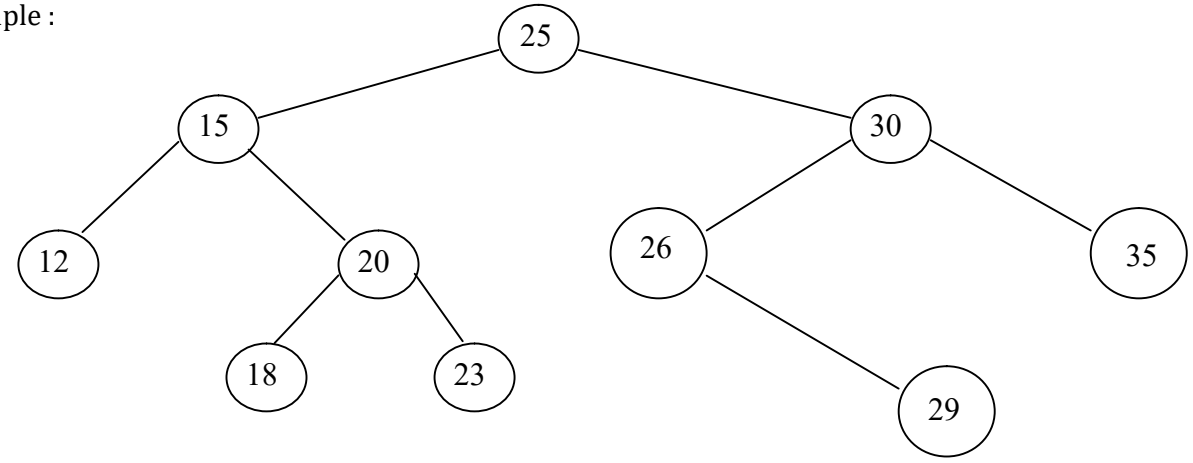
Nous passons en revue quelques arbres binaires particuliers : les arbres ordonnés ou arbres de recherche ainsi que les tas.

4.1. Arbre binaire de recherche

Un Arbre Binaire de Recherche (ABR) ou arbre ordonné est un arbre où les données, présentes au niveau des nœuds, sont régies par une relation d'ordre totale. En plus, quelque soit le nœud interne, les données présentes dans le sous arbre à gauche sont inférieures ou égales à la donnée présente dans ce nœud, qui elle-même est inférieure ou égale à celles présentes au niveau du sous arbre droit.

Cette manière d'organiser les données permet de rechercher les éléments avec une complexité en moyenne de l'ordre de O(logN). L'avantage de cette méthode de recherche par rapport à la recherche dichotomique dans une liste triée réside notamment dans l'opération d'insertion. En effet pour maintenir une liste triée, l'insertion se fait dans le pire des cas (et aussi en moyenne) en O(N), alors que dans les arbres binaires de recherche, la complexité en moyenne de l'insertion est de l'ordre de O(logN).

Exemple :



De par sa structure, le parcours **en ordre (infixe)** d'un arbre binaire de recherche produit une liste triée des données qu'il contient. Les procédures classiques de recherche, insertion, suppression, minimum ou maximum peuvent être aisément effectuées à l'aide de procédures récursives ou itératives.

- **Recherche**

```
bool rechercher(Noeud *racine , int val)
{ if (racine==null) return false;
  else if (racine->element==val) return true;
       else if (racine->element>val)
                 return rechercher(racine->filsGauche,val);
            else return rechercher(racine->filsDroit,val);
}
```

- **Insertion**

```
void inserer(Noeud *&racine , intval)
{ Noeud *p=racine, *prec=NULL ;
  while (p!=NULL) {
  prec=p;
  if (p->element>val) p=p->filsGauche;
  else p=p->filsDroit;
  }
  Noeud *nouv=new Noeud;
  nouv->element=val;
  nouv->filsGauche=nouv->filsDroit=NULL;
  if (prec==NULL) racine=nouv;
  else if  (prec->element>val) prec->filsGauche=nouv;
       else prec->filsDroit=nouv;
}
```

Il est facile de constater que la complexité des opérations insertion et recherche dans un arbre binaire de recherche (de taille n) est égale, **dans le pire des cas**, à la hauteur de l'arbre soit :

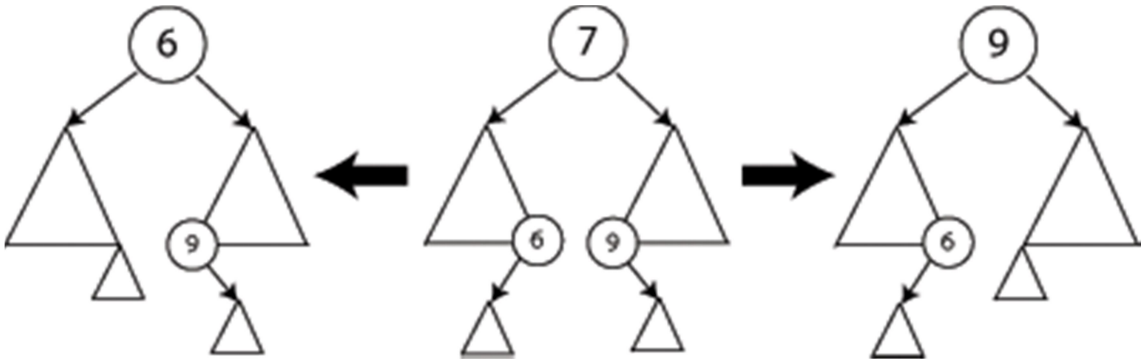
- $O(\log_2(n))$ dans le cas d'un arbre équilibré.
- $O(n)$ dans le cas d'un arbre dégénéré.

Il reste que la complexité en moyenne est de l'ordre de $O(\log_2(n))$.

- **Suppression**

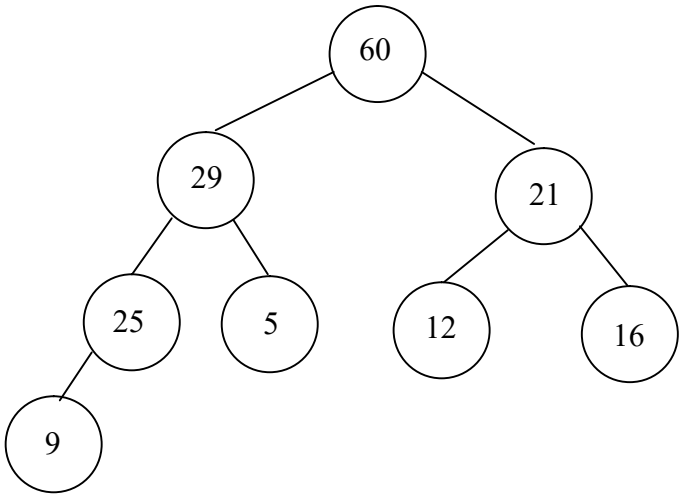
Plusieurs cas sont à considérer, une fois que le nœud à supprimer a été trouvé à partir de sa clé :

- *Suppression d'une feuille* : Il suffit de l'enlever de l'arbre vu qu'elle n'a pas de fils.
- *Suppression d'un nœud avec un enfant* : Il faut l'enlever de l'arbre en le remplaçant par son fils.
- *Suppression d'un nœud avec deux enfants* : Supposons que le nœud à supprimer soit appelé T (le nœud de valeur 7 dans le graphique ci-dessous). On échange le nœud T avec son successeur le plus proche (le nœud le plus à gauche du sous-arbre droit - ci-dessous, le nœud de valeur 9) ou son plus proche prédécesseur (le nœud le plus à droite du sous-arbre gauche - ci-dessous, le nœud de valeur 6). Cela permet de garder une structure d'arbre binaire de recherche. Puis on applique à nouveau la procédure de suppression à T, qui est maintenant une feuille ou un nœud avec un seul fils.



4.2. Les tas (Heaps)

Définition
Un tas est représenté sous forme d'un arbre binaire parfait (rempli de la gauche vers la droite). Les données dans les sommets doivent appartenir à un ensemble totalement ordonné et la donnée présente au niveau d'un nœud doit être plus grande que celles présentes au niveau de ses deux nœuds fils dans le cas où ce nœud possède deux fils ou bien au niveau de son nœud fils, dans le cas où il n'y a qu'un seul. On parle alors de max-tas (ou tas décroissant) comme l'illustre la figure suivante. Noter la différence avec un arbre binaire de recherche.



Le tas peut aussi être ordonné de manière croissante, on parle alors de min-tas (tas croissant).

Implémentation d'un tas à l'aide d'une structure linéaire

Pour chaque structure de tas correspond une structure linéaire (tableau, liste) particulière obtenue par la numérotation des nœuds ligne par ligne et de gauche à droite. La racine occupe le premier élément de la structure. Il est facile de vérifier (par récurrence) que les nœuds du niveau *i* occupent les rangs entre $[2^i, 2^{i+1}[$. Le parent d'un nœud qui se trouve en position j, autre que la racine, occupe la position $\lfloor j/2 \rfloor$, c'est-à-dire la partie entière de j divisé par 2.

Le tableau suivant correspond au codage de la structure de tas décrite précédemment :

1	2	3	4	5	6	7	8
60	29	21	25	5	12	16	9

La declaration d'un tas peut donc être :

```
const int MaxTas=999;

struct Tas {
 int Donnees[MaxTas];    //La case indice 0 ne sera pas utilisée
 int taille;
};
```

Quelques opérations sur le tas

⇒ Insertion dans un tas

L'insertion dans un tas doit respecter la structure d'arbre parfait. Pour cela, l'insertion se fait au niveau de la dernière rangée si celle-ci n'est pas pleine. Dans le cas contraire, on doit créer une nouvelle rangée et l'élément à insérer doit occuper la première place dans cette nouvelle rangée. Dans les deux cas l'insertion se fait toujours à la fin de la structure linéaire, en l'occurrence le tableau correspondant. D'autre part, pour respecter le fait que la donnée présente au niveau du père doit être supérieure à celles présentes dans les nœuds fils, le nouvel élément inséré doit être échangé avec son père si ce principe n'est pas respecté autant de fois qu'il est nécessaire.

La procédure d'insertion peut être implémentée comme suit :

```
void insertionTas(Tas &T, int cle)
{ int i=T.taille+1;
  while (i>=2 && cle>T.Donnees[i/2])
  { T.Donnees[i]=T.Donnees[i/2]; i=i/2;}
  T.Donnees[i]=cle;
  T.taille++;
}
```

Complexité de l'insertion dans un tas

L'insertion de la k^{ème} donnée se fait toujours, en premier lieu à la fin du tableau. Ceci correspond donc au niveau $\log_2(k)$ de l'arbre qui représente le tas. Par conséquent, dans le pire des cas et lors de l'insertion, la donnée peut remonter jusqu'au sommet de l'arbre. Par conséquent, la complexité de l'insertion est $O(\log_2(k))$ et ceci bien sûr dans le pire des cas.

Exercice : Proposer une implémentation de la suppression dans un tas et mesurer sa complexité.

Le Tri par Tas (HeapSort)

De par sa structure, le sommet d'un tas contient toujours la donnée ayant la plus grande valeur de la clé (max-Tas) ou la plus petite (min-Tas). Cette remarque importante peut être utilisée pour le développement d'un des algorithmes les plus efficaces pour le tri qui a une complexité dans le pire des cas, et aussi en moyenne, égale à $O(n\log_2(n))$.

Dans le cas d'un max-tas, le principe du tri tas consiste d'abord à échanger la donnée contenue dans la racine avec celle contenue dans la dernière feuille de l'arbre qui correspond à la dernière case du tableau représentant le tas. Par conséquent, après cette opération d'échange la donnée contenue dans la dernière case du tableau est la plus grande et reste inchangée dans le cas du tri par ordre croissant des données d'un tas. Pour compléter le tri, l'algorithme doit reconstituer les données dont les indices se situent entre 1 et n-1 en un sous tas pour pouvoir réutiliser la remarque précédente. Pour respecter la structure d'un tas, la nouvelle donnée qui vient d'être insérée dans la racine doit être éventuellement déplacée vers le bas pour être plus grande que les données contenues dans ses fils. Cette opération doit être répétée autant de fois qu'il est nécessaire jusqu'à atteinte d'une feuille du tas et que la donnée qui descend est plus petite que celles contenues dans ses deux fils.

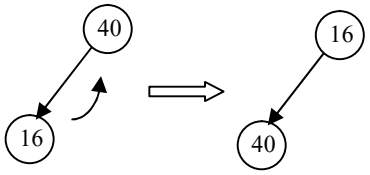
Exemple : Tri Tas (cas d'un min-tas) appliqué au tableau suivant :

40	16	20	23	28

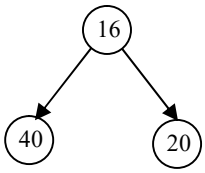
a) Construction du Tas (Insertion)

- Insertion de 40

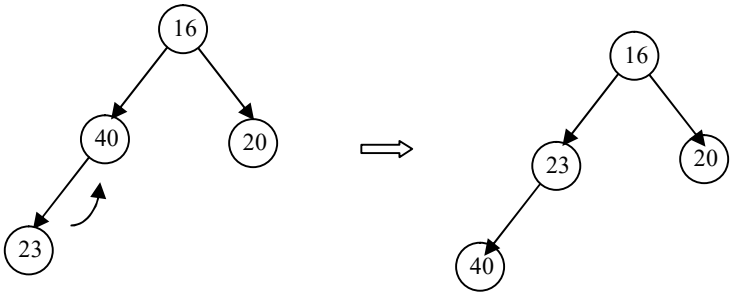
- Insertion de 16



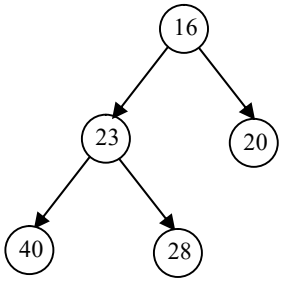
- Insertion de 20



- Insertion de 23



- Insertion de 28



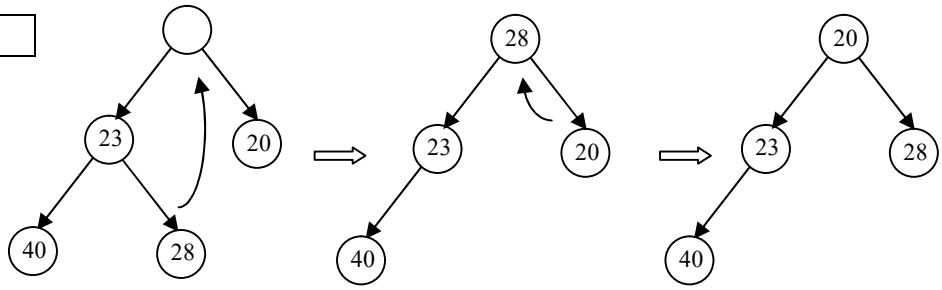
Tas Obtenu après cette étape :

16	23	20	40	28

b) Tri Tas (Suppression)

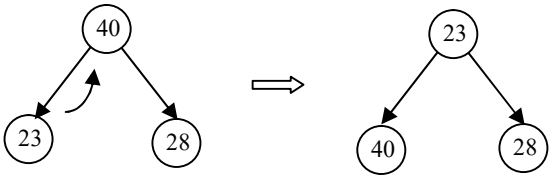
- Suppression de la racine 16

16				



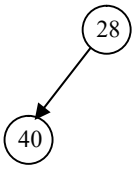
- Suppression de 20

16	20			



- Suppression de 23

16	20	23		



- Suppression de 28

16	20	23	28	

- Suppression de 40 (Tas vide)

16	20	23	28	40

Les détails d'implémentation du tri tas figurent ci-dessous.

```
void DeplaceVersLeBas(Tas &A, int premier, int dernier)
{
int r,temp;
r=premier;
while (r<=dernier/2)
    if (dernier==2*r)
     {
      if (A.Donnees[r]>A.Donnees[2*r])
          {
            temp=A.Donnees[r];
            A.Donnees[r]=A.Donnees[2*r];
            A.Donnees[2*r]=temp;
          }
      r=dernier;
     }
    else
     if (A.Donnees[r]>A.Donnees[2*r] &&A.Donnees[2*r]<=A.Donnees[2*r+1])
      {
       temp=A.Donnees[r];
       A.Donnees[r]=A.Donnees[2*r];
       A.Donnees[2*r]=temp;
       r=2*r;
      }
     else
      if (A.Donnees[r]>A.Donnees[2*r+1] &&A.Donnees[2*r+1]<=A.Donnees[2*r])
       {
        temp=A.Donnees[r];
        A.Donnees[r]=A.Donnees[2*r+1];
        A.Donnees[2*r+1]=temp;
        r=2*r+1;
       }
      else r=dernier;
}

void TriTas(Tas& A)
{
int i,temp;
for (i=A.taille/2;i>=1;i--) DeplaceVersLeBas(A,i,A.taille);
for (i=A.taille;i>=2;i--)
{ temp=A.Donnees[1];
  A.Donnees[1]=A.Donnees[i];
  A.Donnees[i]=temp;
  DeplaceVersLeBas(A,1,i-1);
}
}
```

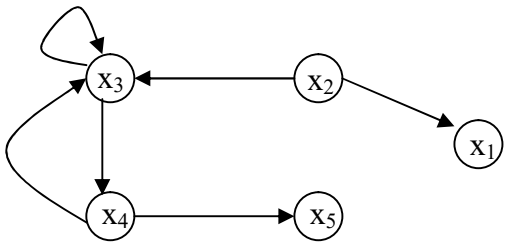
La complexité du tri tas peut être évaluée comme suit : chaque opération de déplacement prend dans le pire des cas $\log_2(p)$ où p étant le nombre de nœuds concernés par cette opération. Par conséquent, la complexité totale est $O(\sum_{p=1}^{n} \log_2(p)\ ) = O(n\log_2(n))$.

CHAPITRE 4 : LES GRAPHES

Plan

1. Définition
2. Représentation des graphes
3. Parcours des graphes
4. Le problème des chemins optimaux

1. DEFINITION



1.1 Graphes orientés

Définitions :

- Un graphe $\boldsymbol{G}$ est un couple $(\boldsymbol{X}, \boldsymbol{U})$ où :
  - X est un ensemble $\{x_1, \dots, x_N\}$ de ***nœuds ou sommets***.
  - $U = \{u_1, u_2, \dots, u_M\}$ est une famille de couples ordonnées de sommets appelées ***arcs***.
- Un graphe est dit **valué** s'il ∃ une application $C : U \rightarrow R$, associant à chaque arc u un réel C_u.
- Une **boucle** est un arc reliant un nœud à lui-même.
- Chaque arc $u = (x, y)$ a deux extrémités appelées **initiale** (x) et **terminale** (y). u est dit **incident intérieurement** à x et **incident extérieurement** à y.
- Dans un arc de la forme (x, y), x est appelé **prédécesseur** de y et y est dit **successeur** de x. x et y sont appelés sommets voisins (ou adjacents).
- L'ensemble des successeurs de x est noté $\Gamma(x)$ et celui de ses prédécesseurs est noté $\Gamma^-(x)$.
- Le nombre de successurs de x est appelé **demi degré extérieur** et est noté $d_G^+(x) = |\Gamma(x)|$. Le **demi degré intérieur** de x est défini comme étant $d_G^-(x) = |\Gamma^-(x)|$. Le **degré** de x est défini comme suit : $d_G(x) = d_G^+(x) + d_G^-(x)$.
- La **densité** d'un graphe est définie par le rapport $\boldsymbol{m/n^2}$ c'est-à-dire le nombre actuel d'arc de G divisé par le nombre maximum d'arcs que peut avoir G. La plupart des graphes rencontrés en pratique ne sont pas très dense (à faible densité). Ils sont appelés creux (en Anglais sparse).

1.2 Graphes non orientés

Dans les applications où on s'intéresse uniquement aux paires de sommets reliées et non à leurs orientations, on parle de graphes non orientés. Dans ce type de graphe, on parle d'une arête (edge en Anglais) $e = [x, y]$ au lieu de l'arc (x, y). Un graphe non orienté est noté $G = (X, E)$ où E désigne une famille d'arêtes. Les termes qui s'appliquent aux graphes orientés et sont indépendants de l'orientation restent valables pour les graphes non orientés.

1.3 Parcours

- On appelle **chemin** μ de longueur p toute suite de p arcs $(u_1, \dots, u_p)$ telle que l'extrémité initiale de u_i est égale à l'extrémité terminale de u_{i-1} $\forall\, i > 1$, et l'extrémité terminale de u_i est égale à l'extrémité initiale de u_{i+1} $\forall\, i < p$.
- On appelle **circuit** tout chemin fermé, c'est-à-dire un chemin tel que $u_1 = u_p$.
- Un arc est un chemin de longueur 1 et une boucle est un circuit de longueur 1 aussi.
- Si le graphe est non orienté, on parle de **chaine** au lieu de chemin, et de **cycle** au lieu de circuit.
- Un **parcours** est un élément de l'ensemble des **chemins**, **circuits**, **chaînes** et **cycle**s.

1.4 Graphes particuliers

- Un graphe orienté $G = (X, U)$ est dit **symétrique** si $(x, y) \in U \Longrightarrow (y, x) \in U$. Les graphes non orientés peuvent être représentés par des graphes orientés symétriques.
- Le graphe **complémentaire** de $G = (X, U)$ est le graphe $H = (X, X^2 - U)$.
- Un graphe est dit **complet** si toute paire de sommets est connectée par un arc ou une arête.

2. REPRESENTATION DES GRAPHES

2.1 Matrice d'adjacence

Considérons un graphe orienté $G = (X, U)$ dont le nombre de sommets est n et celui des arcs étant m. Une **matrice d'adjacence** est une matrice M $(n \times n)$ dont les éléments sont booléens indiquant si les sommets correspondants sont reliés.

Un graphe valué $G = (X, U, W)$ peut être représenté par une matrice W $(n \times n)$ qui donne à la fois les coûts des arcs et fait fonction de la matrice d'adjacence. Les matrices d'adjacence permettent de détecter les boucles, la symétrie ainsi que la connectivité. On peut facilement à l'aide de cette structure de données obtenir la liste des successeurs d'un sommet ainsi que celle de ses prédécesseurs. Cependant, ces structures ne sont pas adéquates pour les graphes dont la densité est faible.

Exemple : Le graphe donné en exemple à la section 1.1. est représenté par la matrice d'adjacence M :

	x_1	x_2	x_3	x_4	x_5
x_1	0	0	0	0	0
x_2	1	0	1	0	0
x_3	0	0	1	1	0
x_4	0	0	1	0	1
x_5	0	0	0	0	0

2.2 Listes d'adjacence

a. Listes Contigües

Les listes d'adjacence implémentent la structure $G = (X, \Gamma, \Gamma)$ où les sommets sont rangés consécutivement dans des tableaux ou bien dans des listes chaînées. Dans le cas des tableaux, on note par **Succ** le tableau dont les éléments sont les listes des successeurs des sommets $0,1, \ldots, n-1$ dans l'ordre, c'est-à-dire $\Gamma(0), \Gamma(1), \ldots, \Gamma(n-1)$. Le nombre d'éléments de **Succ** est donc le nombre d'arcs du graphe à savoir m. Pour délimiter ces listes successives des successeurs, on fait usage d'une structure **Tête** sous forme de tableau à n éléments, qui donne pour chaque sommet l'indice dans le tableau **Succ** où commence ses successeurs. En effet, les successeurs d'un sommet y sont rangés dans **Succ** de **Tete[y]** à **Tete[y+1]-1**. Dans le cas où un sommet y n'a pas de successeurs, on pose **Tete[y]=Tete[y+1]**.

Les graphes non orientés sont codés comme des graphes orientés symétriques. Si $[x, y]$ est une arête, x apparaît dans la liste des successeurs de y, et y dans celle de x. Dans le cas d'un graphe valué $G = (X, U, W)$, on fait usage d'un tableau de poids **W** en regard du tableau **Succ**.

L'avantage majeur de ce codage étant sa compacité. En effet, un graphe consomme $n + m + 1$ mots mémoire qui est bien moins que n^2 mots d'une matrice dans le cas où G n'est pas dense.

Exemple : (Graphe section 1.1)

1	2	3	4	5	6
1	1	3	5	7	7

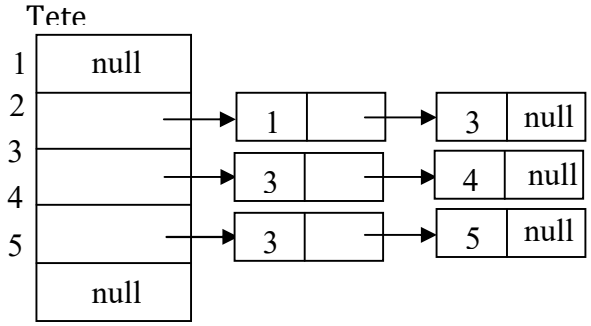
Tete

1	3	3	4	3	5
1	2	3	4	5	6

Succ

b. Listes Chaînées

Le même principe peut être appliqué en utilisant des listes chaînées comme suit :



Là aussi, il y a lieu d'ajouter pour chaque bloc une case supplémentaire dans le cas de graphe valué pour stocker le poids de l'arc.

3. PARCOURS DES GRAPHES

3.1 Construction des Listes de prédécesseurs

a. Représentation en matrice d'adjacence

Soit M la matrice d'adjacence d'un graphe G d'ordre n. Les successeurs d'un sommet quelconque i peuvent être obtenus en parcourant la ligne i de la matrice M en $O(n)$ opérations. Les prédécesseurs sont obtenus en parcourant la colonne i également en $O(n)$.

b. Représentation en listes d'adjacence

Pour les graphes de faible densité, il est avantageux de coder le graphe par des structures linéaires représentant les listes d'adjacence. En effet, la structure nécessite seulement $O(m+n)$ mots-mémoire et la liste des successeurs ne nécessite que $O(d^+(i))$ opérations. Cependant, les listes d'adjacence ne peuvent pas être manipulées efficacement pour retrouver la liste des prédécesseurs d'un sommet i. Pour retrouver la liste des prédécesseurs, il faut balayer tout le graphe pour détecter les sommets dont i est successeur et ceci nécessite $O(m)$ opérations et $O(nm)$ opérations pour construire les listes des prédécesseurs de tous les sommets. Dans le cas où la liste des prédécesseurs est requise fréquemment, il est plus avantageux, en termes de complexité temps, de construire le graphe inverse G^- de G où les listes des successeurs ne sont autres que les listes de prédécesseurs dans G. Dans ce cas on consomme $2(m+n)$ mots pour les deux graphes, mais c'est nettement inférieur à n^2 pour les graphes de faible densité.

3.2. Exploration des Graphes

L'exploration consiste à déterminer l'ensemble des descendants d'un sommet s, c'est-à-dire l'ensemble des sommets situés sur des chemins d'origine s. Le principe de l'exploration d'un graphe peut être décrit comme suit : Au début on marque le sommet s, ensuite à chaque fois qu'on rencontre un arc (x, y) avec x non marqué et y marqué, on marque alors y.

a. Exploration en largeur (BFS : Breadth-First Search)

Dans ce type d'exploration, on implémente l'ensemble des sommets marqués Z comme étant une file. Les sommets sont marqués par ordre de nombre d'arcs croissant à s : on commence par les successeurs de s, puis les successeurs des successeurs de s, etc. Un sommet x en tête de la file Z reste tant que ses successeurs ne sont pas examinés. Pendant ce temps, tout successeur de x non marqué est marqué et rangé en fin de Z.

Algorithme :

```
Marquer s ; Enfiler(F,s)
Repeter  ……N
  x=Defiler(F)
  pour tout successeur y non marqué de x ……∑d+(i) = M
   Enfiler(F,y)
   Marquer y
  Finpour
Jusqu'à FileVide(F)
```

Complexité : O(N+M)≈ O(M)

b. Exploration en Profondeur (DFS : Depth-First Search)

Il s'agit cette fois ci d'implémenter Z comme étant une pile et l'exploration progresse en se déplaçant le plus loin possible le long d'un chemin dont s est l'origine avant de rebrousser chemin. La pile Z contient les sommets explorés et permet à la procédure de rebrousser chemin.

Algorithme :

```
Marquer s ; Empiler(P,s)
Repeter
  Tant qu'il y a un successeur y à TetePile(P) et non marqué faire
    Marquer y
    Empiler(P,y)
  Fintq
  x=Depiler(P)
Jusqu'à PileVide(P)
```

Idem pour la complexité que BFS

c. Exemple

Soit un graphe G orienté dont les sommets sont (s_1, s_2, s_3, s_4, s_5, s_6) représenté par la matrice d'adjacence suivante. Donner les ordres de parcours en largeur BFS et en profondeur DFS, à partir du sommet s1.

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

BFS :

File	s1	s3 s6 s2	s5 s4 s3 s6	s5 s4 s3	s5 s4	s5	File Vide
Sommets Marqués	s1	s2,s6,s3	s4,s5				

Ordre de défilement : s1,s2,s6,s3,s4,s5

DFS :

Pile	s1	s2 s1	s4 s2 s1	s3 s4 s2 s1	s5 s3 s4 s2 s1	s6 s5 s3 s4 s2 s1	s5 s3 s4 s2 s1	s3 s4 s2 s1	s4 s2 s1	s2 s1	s1	Pile Vide
Sommets Marqués	s1	s2	s4	s3	s5	s6						

Ordre de dépilement : s6,s5,s3,s4,s2,s1

4. LE PROBLEME DES CHEMINS OPTIMAUX

4.1. Typologie, algorithmes et applications des problèmes de Chemins Optimaux

Soit $G = (X, A, W)$ un graphe orienté valué et considérons le coût d'un chemin comme étant la somme des coûts de ses arcs. Les principaux problèmes consistent à trouver des chemins de coût minimal. Dans ce contexte, il y a lieu de distinguer les trois problèmes suivants :

1. Soient s, t deux sommets, trouver un plus court chemin de s à t.
2. Trouver un plus court chemin de s vers tout autre sommet de G.
3. Trouver un plus court chemin entre tout couple de sommets.

Dans cette section, on s'intéressera au second problème en calculant pour chaque sommet x la valeur du plus court chemin du sommet de départ vers x ; V[x], appelée aussi étiquette ou label.

Il existe une famille d'algorithmes qui calculent V[x] d'une manière définitive pour chaque sommet x. Ces algorithmes sont appelés **à fixation d'étiquettes** et le plus répandu de cette classe est l'algorithme de Dijkstra. D'autres algorithmes affinent jusqu'à la dernière itération l'étiquette de chaque sommet x. Cette classe est appelée **à correction d'étiquettes**. Il y a lieu de distinguer les cas suivants :

- Cas W constante. Le problème se réduit à celui de la recherche des chemins contenant le plus petit nombre d'arcs qui peut être résolu par une exploration en largeur.

- Cas $W \geq 0$. Le problème peut être résolu par l'algorithme de Dijkstra qui est du type à fixation d'étiquettes et dont la complexité est $O(n^2)$. Il existe une implémentation en structure de tas dont la complexité est $O(m\ log\ n)$.

- Cas W quelconque. Il existe un algorithme dû à Bellman du type à correction d'étiquettes et dont le principe repose sur la programmation dynamique. La complexité étant $O(nm)$ qui peut être réduite à $O(m)$ lorsque le graphe G ne contient pas de circuits.

Les applications des problèmes des chemins optimaux sont nombreuses et diverses. Dans le domaine des transports, par exemple, on s'intéresse aux chemins optimaux d'une ville x à une autre ville y. Dans le routage du trafic réseau, on parle des protocoles OSPF (open **shortest path** first).

4.2. Algorithme de DIJKSTRA

Il fait partie des algorithmes à fixation d'étiquettes et ne peut être appliqué que pour les graphes à valuations **positives**. L'algorithme fixe l'étiquette de chaque sommet x à chaque itération en mettant à jour un tableau de booléens. Lors de sa phase d'initialisation, l'algorithme initialise un tableau V à ∞, P à zéros et D à faux. Pour un sommet donné s, on initialise V[s] à zéro et P[s] à s. L'itération principale de l'algorithme est constituée de deux boucles. La première consiste à trouver un sommet non encore fixé de l'ensemble V qui est minimal. Pour cela, la valeur ∞est d'abord affectée à Vmin, ensuite pour tout y allant de 1 à n dont D[y] est faux et V[y] strictement inférieure à la valeur Vmin, l'algorithme conserve y dans une variable x et affecte V[y] à Vmin. Dans le cas où Vmin est toujours ∞, D[x] est remplacé par vrai. La seconde boucle consiste à parcourir tous les sommets k successeurs de x pour mettre à jour la liste des successeurs. Si V[x] +W[k] est inférieur à V[y] alors V[x] + W[k] est affectée à V[y]. Finalement, x est affecté à P[y].

Algorithme de Dijkstra

```
s : sommet de départ
Initialiser le tableau V à +∞      //Valeur des plus court chemins
Initialiser le tableau P à 0       //Détail des chemins (Path)
V[s]=O
P[s]=s
Répéter
  // Chercher sommet non fixé de V minimal
  Vmin=+∞
  Pour i=1 à n faire
    Si (i non fixé  et  V[i]<Vmin) alors  x=i; Vmin=V[i]  fsi
  Finpour
  Si Vmin<+∞
    Fixer x
    // Mise à jour des successeurs
    Pour chaque successeur y de x faire
      Si V[x]+W[x][y]<V[y]
          V[y]= V[x]+W[x][y];
          P[y]=x
      Fsi
    Finpour
  Fsi
Jusqu'à Vmin=+∞
```

La complexité de l'algorithme de Dijkstra dans le pire des cas étant $O(n^2)$.

Exemple : Soit G = (X,U,C) un graphe orienté valué représenté par la figure suivante. Dérouler l'algorithme de Dijkstra en prenant comme sommet de départ 1.

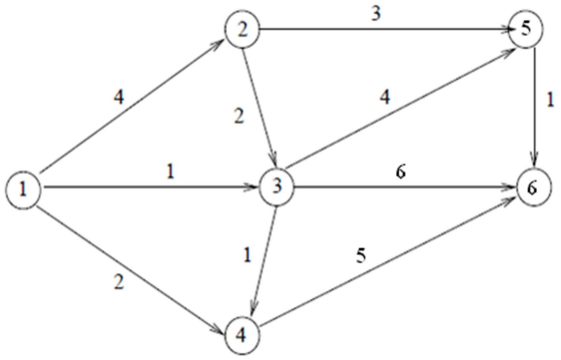


Tableau V

1	2	3	4	5	6	Sommets Marqués
0	∞	∞	∞	∞	∞	1
	4	1	2	∞	∞	3
	4		2	5	7	4
	4			5	7	2
				5	7	5
					6	6
0	**4**	**1**	**2**	**5**	**6**	**Résultat Final**

Tableau P

1	2	3	4	5	6
1	0 1	0 1	0 1	0 3	0 3 5

On en déduit que les chemins optimaux à partir du sommet 1 sont :

1→1	0
1→2	4
1→3	1
1→4	2
1→3→5	5
1→3→5→6	6

ANNEXE : LE LANGAGE JAVA

Plan

1. Structure générale
2. Déclarations et expressions
3. Instructions fondamentales
4. Fonctions (ou méthodes)
5. Classes et objets
6. Encapsulation
7. Héritage
8. Transtypage et polymorphisme
9. Généricité
10. Gestion des exceptions

Le langage **Java** est un langage de programmation informatique orienté objet créé par James Gosling et Patrick Naughton, employés de Sun Microsystems, avec le soutien de Bill Joy (cofondateur de Sun Microsystems), présenté officiellement le 23 mai 1995. La société Sun a été ensuite rachetée en 2009 par la société Oracle qui détient et maintient désormais Java. La particularité et l'objectif central de Java est que les logiciels écrits dans ce langage doivent être très facilement portables sur plusieurs systèmes d'exploitation tels que UNIX, Windows, Mac OS ou GNU/Linux, avec peu ou pas de modifications.

1. STRUCTURE GENERALE

Un programme Java est constitué :

- d'un ensemble de directives de compilation (ou d'importation) comme
  ```
  import java.io.*
  import java.util.*
  ```
- d'un ensemble de classes ; généralement chaque classe est stockée dans un fichier portant le même nom de la classe avec l'extension .java

Chaque classe contient des membres qui sont

- des attributs
- des fonctions (ou méthodes)

Les membres d'une classe sont

- des membres de classe (static)
- des membres d'objet (ou d'instance)

Une fonction spéciale est appelée à l'exécution et doit se trouver dans l'une des classes formant le programme :

```
public static void main(String[] args) {. . .}
```

Voici un exemple d'un programme typique écrit en Java :

```
public classHelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Le fichier source porte presque toujours le nom de la classe avec l'extension ".java" (ici "HelloWorld.java", ce serait même obligatoire si la classe avait l'attribut **public** dans sa déclaration — la rendant alors accessible à tout autre programme).

2. DECLARATIONS & EXPRESSIONS

Types primitifs

byte : un octet
short, int, long : entier sur 2, 4 et 8 octets respectivement
float, double : réel sur 4, 8 octets
char : caractère sur 2 octets
boolean : booléens, de valeur true et false seulement.

Déclaration

- Une variable se déclare en donnant d'abord son type.

```
int i,j;
float re,im;
boolean trouve;
short x=0; // Une variable peut être initialisée lors de sa déclaration
```

- Une variable **static** est une donnée de classe.
- Une variable **final static** est une constante.

Opérateurs (Par ordre décroissant de priorité)

* Arithmétiques :

Plus unaire	+		Moins unaire	−	
Incrément	++		Décrément	−−	
Multiplication	*		Division	/	Modulo %
Addition	+		Soustraction	−	
Affectation	=	+= *= /= −= %=			

* Logiques :

Non	!					
Comparaisons	==	!=	<	<=	>	>=
Ou	||					
Et	&&					

Remarques :

* L'opérateur **?** permet d'exécuter l'équivalent d'une instruction if else;

Ex : `max = (a>b) ? a : b;`

* L'affectation retourne la valeur affectée ce qui permet d'écrire par exemple a = b = 5 ;

Expressions

```
radians = (degres/180) * Math.PI;
s = "Bonjour" + "Monsieur";
resultat = (i!=0) && (i%2==0);
j += 2; // j = j + 2
```

- Le transtypage (conversion de type ou cast) se fait avec la syntaxe : `(type) variable`
  Exemple : `float x; int d =(int)x;`

- Incrémentations

```
      ++i i++ --j j--
Exemple :
      i = 1;
      x = ++i;
      j = 3;
      y = j--;

On obtient x = i = 2 et y = 3, j = 2.
```

Tableaux

Les indices d'un tableau de taille n vont de 0 à n-1 (comme dans le langage C/C++). Une variable de type tableau est une référence. Elle contient l'adresse du tableau.

- Déclaration et construction

```
// Tableau de 5 entiers
int[] tab=new int[5]; ou int tab[]=new int[5]; ou int tab[]={0,0,0,0,0};
double[][] mat = new double[N][M]; // Matrice de doubles
```

- Utilisation

```
x = m[i][j];
for (i = 0; i < a.length; i++)
  System.out.print(a[i]);
```

La propriété **length** fournit le nombre d'éléments du tableau.

3. INSTRUCTIONS FONDAMENTALES

Une instruction atomique est toujours terminée par point virgule (;). Un bloc est une suite de déclarations et d'instructions encadrée par des accolades {……}. Voici la syntaxe des principales instructions de contrôle :

Boucles

Bien qu'elles aient toutes un rôle similaire, chaque boucle est pourtant adaptée à une situation :

- Structure **tant que** (adaptée pour effectuer des opérations tant qu'une condition est remplie) :

```
while (<expression booléenne>) {
       instruction(s)
}
```

- Structure **faire … tant que** (comme la structure **tant que** mais la première itération est exécutée quelle que soit la condition, pour les autres itérations la condition doit être remplie) :

```
do {
  instruction(s)
}
while (<expression booléenne>);
```

- Structure **pour** (adaptée lorsqu'une collection doit être parcourue en totalité pour traitement) :

```
for(<initialisation>;<condition de poursuite>;<expression d'incrémentation>){
  instruction(s)
}
```

- Structure **pour chaque** (simplification du for en for each, dans laquelle l'expression doit être un tableau ou une collection) :

```
for (type variable : <expression>) {
    instruction(s)
}
```

Structures conditionnelles

- Structure **si … sinon**

```
if (<expression booléenne>)
{ instruction(s) }
else { instruction(s) }
```

- Structure **atteindre … cas x … cas y …** : embranchement vers un bloc d'instructions énuméré.

```
switch(<expression de type numérique>){
case <constante de type numérique>:
        instruction(s)
break;
case <constante de type numérique>:
        instruction(s)
break;
[...]
default:
        instruction(s)
}
```

Remarque :

- Le switch ne fonctionne pas sur toutes les constantes de type numérique mais seulement sur les entiers. Switch fonctionne également avec des variables de type char.
- La commande **break** sort immédiatement de la boucle en cours (for, while, do), et permet de sortir d'une clause contenue dans un switch. Si le break est omis, l'exécution du switch se poursuit de 'case' en 'case'.
- Une expression **continue** termine l'itération en cours et continue à la prochaine. Son usage est néanmoins à éviter puisqu'elle tend à favoriser un type de programmation non structurée (programmation spaghetti).

4. FONCTIONS (OU METHODES)

Chaque classe contient une suite de fonctions ou méthodes. Voici quelques exemples :

```
static int next(int n) {
if (n % 2 == 1)
   return 3 * n + 1;
return n / 2;
}

static int pgcd(int a, int b) {
return (b == 0) ? a : pgcd( b, a % b );
}
```

La **signature** est la suite des types des arguments. Une fonction qui ne retourne pas de valeur a pour type de retour le type **void** (équivalent des procédures en Pascal).

Surcharge (Overloading)

Un même identificateur peut désigner des fonctions différentes pourvu que leurs signatures soient différentes.

```
static int produit(int n, int p) {
return n*p;
}

static int produit(int n) {
return n*n;
}
```

5. CLASSES & OBJETS

Une classe est un ensemble de déclarations d'attributs et de définitions de fonctions. Elle permet aussi la déclaration d'un type non primitif. Pour créer des objets d'une classe, on utilise l'opérateur **new**.

```
class Complex {
      int re;
      int im;

      public static void main(String[] args) {
            Complex a = new Complex();
            a.re = 3; a.im = 5;
            ...

      }
}
```

Constructeurs

La fonction Complex() est le constructeur par défaut. On peut déclarer (par surcharge) des constructeurs appropriés :

```
class Complex {
...
Complex (int x, int y) { re = x; im = y; }
...
}
```

et écrire

```
public static void main(String[] args) {
Complex a = new Complex(3,5);
...
}
```

Méthodes d'objet

Les méthodes d'objet s'appliquent à l'objet appelant.

```
class Complex {
...
void imprimer() {
System.out.println(re + " " + im);
}
...
}
```

On s'en sert avec

```
a.imprimer();
```

Méthodes de classe

De manière équivalente :

```
static void simprimer(Complex a) {
System.out.println(a.re + " " + a.im);
}
```

s'utilise par

```
simprimer(a);
```

Objets et types primitifs

Une variable d'un type primitif désigne une valeur de ce type alors qu'une variable d'un type non primitif désigne l'emplacement où se trouve l'objet après sa création. On dit que c'est une référence. C'est pourquoi la déclaration d'une référence, et la création de l'objet sont deux actes distincts.

```
// Declaration de variables
Complex a;
int[] m;
// Creation de l'objet
a = new Complex();
m = new int[10];
```

Après sa déclaration, une variable d'objet contient **null**.

Copie de références et copie d'objets

```
Complex a, b;
int x = 4 , y;
y = x; // Pour les types primitifs, copie de valeur; x et y 2 objets différents
a = new Complex(3,4);
b = a ; // Copie de reference : un seul objet
b.re = 5;
```

Ici, a et b désignent le même objet. En particulier, a.re vaut 5. Une copie "véritable" doit être programmée. Cette différence entre types primitifs et non primitifs peut conduire à des confusions ; c'est pour cela des types non primitifs équivalents aux types primitifs sont prédéfinies dans le langage Java tels que :

Integer	≡	int
Float	≡	float
Boolean	≡	boolean

6. ENCAPSULATION

En Java, comme dans beaucoup de langages orientés objet, les classes, les attributs et les méthodes bénéficient de niveaux d'accessibilité, qui indiquent dans quelles circonstances on peut accéder à ces éléments. Ces niveaux sont au nombre de 4, correspondant à 3 mots-clés utilisés comme modificateurs : *private*, *protected* (voir héritage) et *public*. La quatrième possibilité est de ne pas spécifier de modificateur (comportement par défaut). Dans ce cas, l'élément n'est accessible que depuis les classes faisant partie du même *package*. Voici un exemple :

```
package com.moimeme.temps;

class Horloge { … }

public class Calendrier
{  void ajouteJour() { … }
   int mois;
        …
}
```

La classe *Horloge*, la méthode *ajouteJour* et l'attribut *mois* ne sont accessibles que depuis les classes faisant partie du *package com.moimeme.temps*.

Modificateur "private" :

Un attribut ou une méthode déclarée "private" n'est accessible que depuis l'intérieur même de la classe. C'est le principe de l'encapsulation.

Modificateur "public" :

Une classe, un attribut ou une méthode déclarée "public" est visible par toutes les classes et les méthodes.

7. HERITAGE

• Le mot réservé « extends »

Pour signifier qu'une classe fille hérite d'une classe mère, on utilise le mot clé **extends**

```
class fille extends mère
```

En java, toutes les classes sont dérivées de la classe **Object**. La classe Object est la classe de base de toutes les autres. C'est la seule classe de Java qui ne possède pas de classe mère. Tous les objets en Java, quelle que soit leur classe, sont du type Object. Cela implique que tous les objets possèdent déjà à leur naissance un certain nombre d'attributs et de méthodes dérivées d'Object.

• Les membres protégés : protected

Les sous-classes n'ont pas accès aux membres private de leur classe mère. Si un attribut de la classe mère est déclaré private, cet attribut n'est pas accessible par les sous-classes (même si elles sont bien héritées) et ne sont accessibles qu'à travers les accesseurs et/ou les méthodes de la classe mère qui les utilise.

```
public class Ville { private String nom; private int nbHab; … }

public class Capitale extends Ville
{ private String pays;
  public afficheCapitale( )
  { System.out.println(nom + " est la Capitale de "+ pays); }
…
}
```

Dans ce cas, il y a erreur, l'attribut nom est privé. Si on veut qu'un attribut ou une méthode soit encapsulé pour l'extérieur mais qu'il soit accessible par les sous-classes, il faut le déclarer **protected** à la place de **private**.

```
public class Ville { protected String nom; protected int nbHab; … }
```

• Redéfinition des méthodes héritées

Une méthode de la classe mère peut être implémentée différemment dans une classe fille : la méthode est dite redéfinie. Aucun signe n'indique en java qu'une méthode est redéfinie (contrairement à Pascal ou à C++). La redéfinition d'une méthode dans une classe fille cache la méthode d'origine de la classe mère. Pour utiliser la méthode redéfinie de la classe et non celle qui a été implémentée dans la classe fille, on utilise le mot-clé **super** suivi du nom de la méthode.

Exemple : super.machin( ) fait appel à la méthode machin( ) implémentée dans la classe mère et non à l'implémentation de la classe fille.

• Utilisation d'un constructeur de la classe mère

On peut aussi faire appel au constructeur de la classe mère en utilisant le mot-clé super( ) avec les paramètres requis. **super** remplace le nom du constructeur de la classe mère. L'instruction super( ) doit être obligatoirement la première instruction des constructeurs de la classe fille. Tout constructeur d'une classe fille appelle forcément l'un des constructeurs de la classe mère : si cet appel n'est pas explicite, l'appel du constructeur par défaut (sans paramètre) est effectué implicitement. Ainsi, un objet dérivé est toujours construit sur la base d'un objet de sa classe mère.

Voici un exemple d'une classe Ville, qui sera étendue par la suite par une classe Capitale :

```
class Ville {
private String nom; //le nom ne sera accessible que par Ville, et pas par Capitale
protected int nbHab; //le nombre d'habitants sera accessible par la classe Capitale

public Ville(String leNom)
{ nom = leNom.toUpperCase( ); nbHab = -1;}
```

```
public Ville (String leNom, int leNbHab)
{ nom = leNom.toUpperCase( );
  if (leNbHab < 0) { System.out.println("Un nombre d'habitant doit être positif.");
                     nbHab = -1;
                    } else nbHab = leNbHab;
}

public String getNom() { return nom; }

public int getNbHab( ) { return nbHab; }

public void setNbHab(int nvNbHab) {
  if (nvNbHab < 0)
   System.out.println("Un nombre d'habitant doit être positif. La modification n'a pas été
prise en compte"); else nbHab = nvNbHab;
}

public String presenteToi() {
String presente = "Ville "+ nom +" nombre d'habitants "; if (nbHab == -1) presente = presente
+ "inconnu"; else presente = presente + " = " + nbHab;
return presente;
} }

class Capitale extends Ville
{ private String pays;
  //constructeurs
  public Capitale(String leNom, String lePays)
  { super(leNom); //appel du constructeur de Ville.
    pays = lePays;
  }


  public Capitale(String leNom, String lePays, int leNbHab) {
   super(leNom, leNbHab); pays = lePays; }

//accesseurs supplémentaires

public String getPays( ) { return pays; }
public void setPays(String nomPays) { pays = nomPays; }

//méthode presenteToi( ) redéfinie
public String presenteToi( )
{ String presente = super.presenteToi( );
  presente = presente + " Capitale de "+ pays; return presente; } }

//Classe de test public

class testVille {
public static void main(String args[])
{ Ville v1 = new Ville("Oran", 1500000);
  Ville v2 = new Ville("Tlemcen");
  Capitale c1 = new Capitale("Alger", "Algerie", 3000000);
  Capitale c2 = new Capitale("Tunis", "Tunisie");
  System.out.println(v1.presenteToi( ));
  System.out.println(v2.presenteToi( ));
  System.out.println(c1.presenteToi( ));
  System.out.println(c2.presenteToi( ));
}
}
```

Exécution:

```
Ville Oran nombre d'habitants = 1500000
Ville Tlemcen nombre d'habitants inconnu
Ville Alger nombre d'habitants 3000000 Capitale de Algerie
Ville Tunis nombre d'habitants inconnu Capitale de Tunisie
```

8. TRANSTYPAGE ET POLYMORPHISME

Le transtypage (conversion de type ou cast en anglais) consiste à modifier le type d'une variable ou d'une expression. Par exemple, il est possible de transtyper un int en double.

• Le transtypage de références d'objets

Il est possible de convertir un objet d'une classe en un objet d'une autre classe si les classes ont un lien d'héritage (encore une fois, on utilise le mot objet par abus de langage : ce sont les références aux objets et non les objets eux-mêmes qui sont transtypés). Le transtypage d'un objet dans le sens fille → mère est implicite. En revanche, le transtypage dans le sens mère → fille doit être explicite et n'est pas toujours possible.

• Transtypage implicite (sens fille → mère)

Il est toujours possible d'utiliser une référence de la classe mère pour désigner un objet d'une classe dérivée (fille, petite-fille et toute la descendance). Il y a alors transtypage implicite de la classe fille vers la classe mère. Cela est logique si on se dit qu'un objet dérivé EST un objet de base.

Exemple : une Capitale EST une Ville, donc on peut utiliser une référence de type Ville pour désigner une Capitale ; il y a transtypage implicite d'une Capitale en Ville. Par contre, une Ville n'est pas forcément une Capitale : donc on ne peut pas transtyper une Ville en Capitale, autrement utiliser une référence de Capitale pour désigner une Ville.

Remarquez que contrairement au transtypage des types primitifs, le transtypage implicite de références entraîne une perte de précision (le type Ville étant moins précis que le type Capitale)

Exemple d'utilisation du transtypage implicite :

```
Ville v; Capitale c = new Capitale("Paris", "France");
v = c; //transtypage implicite d'une Ville en Capitale
```

L'instruction d'affectation v = c implique un transtypage implicite. La référence v du type Ville est alors utilisée pour désigner un objet de type Capitale. L'objet désigné par c, qui est du type Capitale est donc transtypé en Ville. Donc on ne pourra pas utiliser v pour appeler une méthode spécifique de la classe Capitale (car v est une référence sur une Ville, qui ne peut pas appeler des méthodes de la classe Capitale). Il faudrait pour cela effectuer un transtypage explicite de v en Capitale.

• Transtypage explicite (sens mère → fille)

Le transtypage explicite des références est utilisé pour convertir le type d'une référence dans un type dérivé. C'est logique si l'on se dit qu'un objet d'une classe mère n'est pas un objet de ses classes filles (une Ville n'est pas une Capitale). Par exemple, si l'on voulait utiliser la référence v (qui est du type Ville mais qui désigne une Capitale) pour invoquer une méthode spécifique de la classe Capitale, il faudrait réaliser un transtypage de v en Capitale (sinon, le compilateur refuserait : le transtypage dans le sens mère → fille n'étant pas implicite).

```
String lePays; lePays = (Capitale) v.getPays( );
```

Si on avait écrit lePays = v.getPays( ) , le compilateur aurait généré une erreur car getPays( ) n'est pas une méthode de la classe Ville.

Un transtypage explicite n'est pas toujours possible, même si les classes ont un lien de parenté. C'est au programmeur de vérifier que son transtypage n'engendrera pas d'erreur, c'est-à-dire que l'objet est bien effectivement du type dans lequel on transtype la référence. v peut être transtypé en Capitale car v désigne effectivement un objet de type Capitale.

Exemple de transtypage explicite impossible :
créer un objet Ville à partir d'une référence de type Capitale car dans ce cas, c ne peut pas désigner une capitale (il n'y a pas de pays) Capitale c = new Ville("Bruxelles");

- **Le polymorphisme comme choix dynamique du code à l'exécution**

Et si on voulait invoquer la méthode presenteToi( ) avec la référence v ? v.presenteToi( ); (On supposera ici pour alléger l'écriture, que la méthode presenteToi est une procédure).
Quelle implémentation de presenteToi( ) s'exécuterait alors ? Celle de Ville … qui afficherait : Ville Alger nombre d'habitants inconnu ou celle de Capitale ? qui afficherait Ville Alger nombre d'habitants inconnu Capitale de Algerie.
Réponse : c'est l'implémentation de Capitale qui est exécutée, même si v est de type Ville. Pourquoi ? A l'invocation d'une méthode, le choix de l'implémentation à exécuter ne se fait pas en fonction du type déclaré de la référence à l'objet, mais en fonction du type réel de l'objet. v désigne un objet de type Capitale, même si c'est une référence de type Ville : donc c'est la méthode implémentée dans Capitale qui est exécutée. Ce mécanisme montre un aspect important du polymorphisme : le choix du code à exécuter (pour une méthode polymorphe) ne se fait pas statiquement à la compilation mais dynamiquement à l'exécution.

9. GENERICITE

Un des objectifs principaux de la programmation orientée objet est la réutilisation du code. Un mécanisme important qui facilite la réutilisation du code est la généricité. Ce mécanisme consiste principalement à rendre plus général un code. Si par exemple, une implémentation est toujours identique à l'exception du type de base de l'objet manipulé, une implémentation générique peut être utilisée pour décrire la fonctionnalité de base. Prenons une routine de tri d'un tableau d'items ; la logique de tri est indépendante du type des objets à trier (entier, réels, chaines, …) ; une routine de tri générique peut donc être conçue dans ce cas.
Depuis la version 1.5 de Java, il est possible d'implémenter des classes génériques de manière simple. Nous allons montrer dans cette section comment créer des classes et méthodes génériques.

Classes génériques

Considérons la classe (très simple) suivante :

```
1 public class CaseMemoire
2 {
3 // méthodes publiques
4 public int lire( ) { return valeur; }
5 public void ecrire(int x) { valeur  = x; }
6
7 // attributs privés
8 private int valeur;
9 }
```

Une version générique de cette classe peut être comme suit :

```
1  public class CaseMemoire<MonType>
2 {
3 // méthodes publiques
4 public MonType lire( ) { return valeur; }
5 public void ecrire( MonType x ) { valeur  = x; }
6
7 // attributs privés
8 private MonType valeur;
9 }
```

Une classe générique inclut un ou plusieurs types paramètres entourés par <> après le nom de la classe. Ainsi l'utilisateur peut créer des objets comme CaseMemoire<String> ou encore CaseMemoire<Integer> mais pas CaseMemoire<int>. Un exemple d'utilisation d'une telle classe peut être comme suit :

```
1 class Demo
2 {
3 public static void main( String [] args )
4 {
5 CaseMemoire<Integer> m = new CaseMemoire<Integer>();
6
7 m.ecrire( 37 );
8 int val = m.lire( );
9 System.out.println( "Le contenu de la case mémoire est : " + val );
10 }
11 }
```

La ligne 5 peut néanmoins être réécrite de manière plus simple en utilisant l'opérateur <> (dit opérateur diamant) :

```
5 CaseMemoire<Integer> m = new CaseMemoire<>();
```

puisqu'il est evident que l'objet créé sera aussi de type `CaseMemoire<Integer>`

Méthodes génériques

De manière similaire, il est possible de créer des méthodes génériques comme dans l'exemple de la méthode contains suivante :

```
1 public static <MonType> boolean contains( MonType [] tab, MonType x )
2 {
3 for ( MonTypeval : tab )
4    if (x.equals(val))
5        return true;
6
7 return false;
8 }
```

Cette méthode générique permet de faire une recherche (séquentielle) d'un élement x dans le tableau tab de type générique.

10. TRAITEMENT DES EXCEPTIONS

Une *exception* est l'interruption de l'exécution du programme à la suite d'un événement particulier (division par zéro, manque d'espace mémoire, …) et requiert une gestion spéciale.

Si vous vouliez tester les erreurs attentivement dans un langage de programmation qui ne supporte pas la gestion des exceptions vous devriez entourer l'appel de chaque fonction avec le code de vérification des erreurs, même si vous appeliez la même fonction plusieurs fois. En plus, dans ce type de langage une exception arrête complètement l'exécution de tout le programme, ce qui peut avoir des conséquences désastreuses pour des applications temps réel critiques.

Avec la Gestion des exceptions vous mettez tout dans un bloc appelée **région surveillée** indiquée par le mot réservé **try** et capturer toutes les exceptions en un seul endroit. Cela signifie que votre code est beaucoup plus simple à lire et à écrire car le bon code est séparé de celui de la gestion des erreurs.
Bien sûr, l'exception générée doit être traitée quelque part. Cet endroit est le *gestionnaire d'exceptions*, indiqué par le mot réservé **catch**, et il y en a un pour chaque type d'exception. Mais plus important encore, après le traitement d'une exception l'exécution du programme se poursuit.

- **Syntaxe générale**

```
try                                    // Surveiller une région
{
// Code susceptible de générer des exceptions…
   ……..
}
catch (déclaration-exception)        // Attraper une exception
{
// Code permettant de traiter une exception…
instructions
}
catch (déclaration-exception)        // Attraper une exception
{
// Code permettant de traiter une exception…
instructions
}
…
finally {
traitement_pour_terminer_proprement;
  }
```

Le bloc de code **finally** sera exécuté quelque soit le résultat lorsque le programme sortira du bloc **try-catch**.

Voici un exemple de capture d'une exception :

```
FileOutputStreamfos=null;

try{
//Chacune de ces deux instructions peut générer une exception
// création d'un flux pour écrire dans un fichier
fos=newFileOutputStream(...);
// écriture de données dans ce flux
fos.write(...);
}
catch(IOException e){
//Gestion de l'erreur de création ou d'écriture dans le flux
e.printStackTrace();
}
finally{
//Cette section de code est toujours exécutée, qu'il y ait une exception ou pas
// fermeture du flux s'il a été ouvert
if (fos!=null) fos.close();
}
```

Cet exemple permet d'illustrer le mécanisme des exceptions en Java. Dans le cas d'une erreur d'entrée/sortie dans le bloc **try**, l'exécution reprend dans le bloc **catch** correspondant à cette situation (exception de type **IOException**). Dans ce bloc **catch**, la variable **e** référence l'exception qui s'est produite. Ici, nous invoquons la méthode **printStackTrace()** qui affiche dans la console des informations sur l'exception qui s'est produite : nom, motif, état de la pile d'appels au moment de la levée de l'exception et, éventuellement, numéro de ligne auquel l'erreur s'est produite.
Le bloc **finally** est ensuite exécuté (ici pour refermer les ressources utilisées). Il ne s'agit ici que d'un exemple, l'action à mettre en œuvre lorsqu'une exception survient dépend du fonctionnement général de l'application et de la nature de l'exception.