Haute École Bruxelles–Brabant
École Supérieure d'Informatique
Bachelor en Informatique

Rue Royale, 67 – 1000 Bruxelles
02/219.15.46 – esi@he2b.be

ALG3

Algorithmique

2020-2021

Activité d'apprentissage enseignée par :

S. Drobisz *S. Rexhep* *N. Richard*

Document produit avec LaTeX.
Version du 21 octobre 2020.

Table des matières

Première partie

Introduction

Sur les notations

- Dans les notes, nous utilisons un pseudo-code dont la syntaxe ne devrait pas surprendre le lecteur. La sémantique du pseudo-code est semblable à celle de Java. Les différentes méthodes et différents objets introduits sont clairement expliquées.
- Attention : en `C++`, une variable peut désigner directement un objet ou bien être une référence à l'objet (ce dernier cas devant être indiqué explicitement). Il faut s'en rappeler et être prudent lors de la traduction du pseudo-code vers le `C++`.

Sur les paramètres

En Java les paramètres sont transmis par valeur. Pour les objets, cela correspond à en transmettre la référence, (ce qui rend possible une modification par effet de bord). Nous suivons généralement cette façon de procéder. Toutefois, lorsque cela s'avère nécessaire, nous utiliserons la convention suivante pour les paramètres (dans la déclaration de l'algorithme ou de la méthode)

- ▷ Suivi d'une flèche vers le bas (↓), la valeur initiale du paramètre est nécessaire à l'algorithme pour fonctionner.

  Le module va interroger l'état de l'objet sans le modifier.
- ▷ Suivi d'une flèche vers le haut (↑), l'algorithme va donner une valeur au paramètre à la fin de son déroulement.

  Le module va créer un objet. C'est un paramètre de retour.
- ▷ Suivi de la double flèche (↕), l'algorithme va utiliser la valeur de départ mais va aussi la modifier si elle change au cours de l'algorithme.

  Le module va interroger l'état de l'objet et s'autorise à le modifier.
- ▷ On admet également les notations IN, OUT et IN/OUT en lieu et place des flèches.
- ▷ Si rien n'est indiqué, il faut comprendre qu'il s'agit d'un paramètre « à la Java ».

Il s'agit d'une information importante pour celui qui doit comprendre l'algorithme. Lorsqu'il faut l'implémenter, on choisira la solution la plus proche en fonction de ce que permet le langage choisi, mais c'est une question qui n'est pas du ressort de ce cours d'algorithmique.

Dans certains langages, il se peut que

- ▷ si on écrit **méthode**(Classe obj IN) la méthode puisse modifier l'objet,
- ▷ si on écrit **méthode**(Classe obj IN/OUT) la méthode puisse modifier l'objet mais aussi en fournir un autre en sortie.

Deuxième partie

Les listes

Chapitre 1

La liste chainée

Tout comme un tableau, une liste chainée est une collection d'éléments. Mais, contrairement au tableau, les éléments ne sont pas contigus en mémoire mais « chainés ». Nous verrons ses avantages et inconvénients par rapport au tableau, comment l'implémenter en orienté-objet ainsi que les opérations de base sur les listes.

1.1 Définitions

Voici quelques définitions de la liste chainée (*linked list* en anglais) issues de différents livres sur le sujet. Elles nous indiquent à la fois à quoi cela ressemble et l'utilité d'une telle structure.

- *Une liste chainée désigne en informatique une structure de données représentant une collection ordonnée et de taille arbitraire d'éléments de même type, dont la représentation en mémoire de l'ordinateur est une succession de cellules faites d'un contenu et d'un pointeur vers une autre cellule. De façon imagée, l'ensemble des cellules ressemble à une chaine dont les maillons seraient les cellules. L'accès aux éléments d'une liste se fait de manière séquentielle : chaque élément permet l'accès au suivant (contrairement au tableau dans lequel l'accès se fait de manière directe, par adressage de chaque cellule dudit tableau).* (Wikipédia, janvier 2016)
- *Une liste est un conteneur séquentiel capable d'insérer et de supprimer des éléments localement de façon constante, c'est-à-dire indépendamment de la taille du conteneur.* (Structures de données en Java – Hubbard – ed. Schaum's)
- *Les listes sont des conteneurs destinés aux insertions s'effectuant en temps constant quelle que soit la position dans le conteneur.* (La bibliothèque standard STL du C++ – Fontaine – InterEditions)
- *Une liste chainée est une structure de données dans laquelle les objets sont arrangés linéairement. Toutefois, contrairement au tableau, pour lequel l'ordre linéaire est dé-*

terminé par les indices, l'ordre d'une liste chainée est déterminé par un pointeur dans chaque objet. (Introduction à l'algorithmique – Cormen, Leiserson, Rivest – Dunod)

1.2 Liste *versus* liste chainée

Dans la terminologie moderne, le terme *liste* est plutôt utilisé pour dénoter toute collection séquentielle d'éléments (il existe un premier élément, un deuxième, ...). Autrement dit, chaque élément a une position précise dans la collection. En ce sens, un tableau ou un fichier sont également des *listes* !

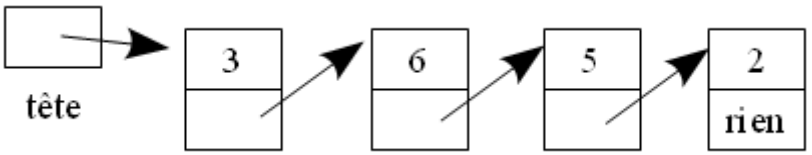
On peut opposer la *liste* à la structure d'*ensemble* où les éléments ne sont pas positionnés (un élément *est* ou *n'est pas* dans un ensemble mais sans que sa position puisse être donnée).

Le concept de liste ne doit pas être confondu avec la notion de *tri*. On peut parler de la *liste* contenant les éléments 3, 8 et 2 dans cet ordre (au sens de *séquence* et pas d'*ordre de tri*).

1.3 Représentation et terminologie

Pour écrire explicitement une liste chainée et son contenu sous forme compacte, on indique la liste des valeurs qu'elle contient entre parenthèses. Exemple : la liste (3, 6, 5, 2).

Schématiquement, on la représente plutôt ainsi :

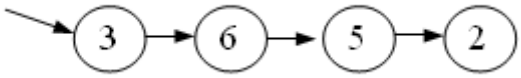


De ce schéma, il apparait que chaque **élément** d'une liste chainée est composé de 2 parties : la *valeur* proprement dite et un *accès* à l'élément suivant.

De plus, on accède au premier élément de la liste via une zone spéciale, la **tête** de la liste chainée.

Le dernier élément n'ayant pas de suivant, on indique *rien* dans la zone réservée à l'accès au suivant. Dans la littérature et selon les langages, on trouvera, à la place de *rien*, les notations *null*, *nil*, ... ou encore Ø.

Le schéma ci-dessus peut encore se compactifier davantage, en une forme qui ne fait pas apparaitre la tête de liste ni les accès aux éléments suivants :



Au contraire du tableau, donc, il n'y a pas d'accès direct à un élément (en donnant son indice) mais il est nécessaire de partir du premier élément et de suivre la « *chaine* ». Comment représenter ce *lien*, cet *accès* en mémoire ? Cela dépend du type de langage :

- ▷ en orienté-objet (OO), un élément est représenté par un objet dont un des attributs est une référence vers l'objet représentant l'élément suivant ;
- ▷ en programmation procédurale (non orienté-objet), on utilise la notion de pointeur (cf. le cours de C/C++). Il est même possible de s'en sortir avec un langage qui ne dispose pas de la notion de pointeur (comme en Cobol par exemple). Cette implémentation sera présentée dans ce syllabus à titre d'information.

1.4 Rappel : opérations sur les tableaux et complexité

Afin de mieux comprendre ce qu'apporte la liste chainée par rapport au tableau, revoyons ce que « coutent » les opérations courantes sur les tableaux. Le tableau ci-dessous donne la complexité (par l'algorithme le plus économique) des opérations élémentaires pour un tableau de taille n trié et non trié.

opération	**tableau non trié**	**tableau trié**
ajout d'un élément	$O(1)$	$O(n)$
suppression d'un élément d'indice connu	$O(1)$	$O(n)$
suppression d'un élément de valeur connue	$O(n)$	$O(n)$
recherche d'un élément	$O(n)$	$O(\log_2 n)$
parcours du tableau	$O(n)$	$O(n)$

Pour rappel, la notation $O(n)$ indique que l'opération prend un temps qui est proportionnel à n, la taille du tableau. La plupart des opérations sont de complexité $O(n)$, par exemple, lors d'une suppression dans un tableau trié, il faut décaler des éléments pour boucher le trou qui est apparu. Il en est de même pour un ajout dans un tableau trié.

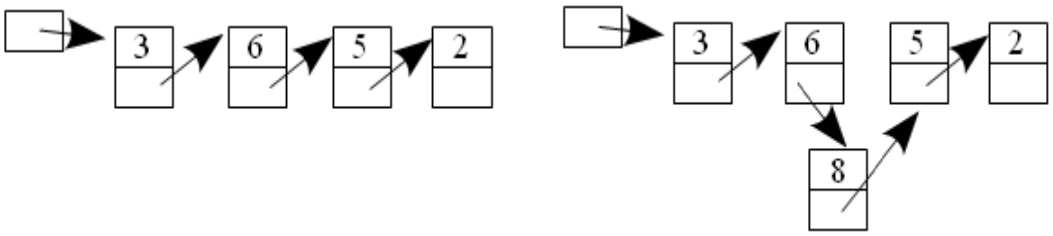
Si le tableau n'est pas trié, il est évident qu'on peut rajouter l'élément en fin de tableau, ce qui est immédiat (complexité $O(1)$), et l'élément supprimé peut être remplacé par le dernier élément du tableau pour boucher le trou.

Dans le cas d'un tableau trié, c'est l'algorithme de recherche dichotomique qui permet de trouver rapidement un élément avec une complexité $O(\log_2 n)$.

1.5 Ajout d'un élément dans une liste chainée

Dans une liste chainée (qu'elle soit ordonnée ou non), il n'est pas nécessaire de décaler des éléments pour un ajouter un. Reprenons la liste chainée de l'exemple ci-dessus et ajoutons l'élément 8 entre le 6 et le 5 :

avant : après :

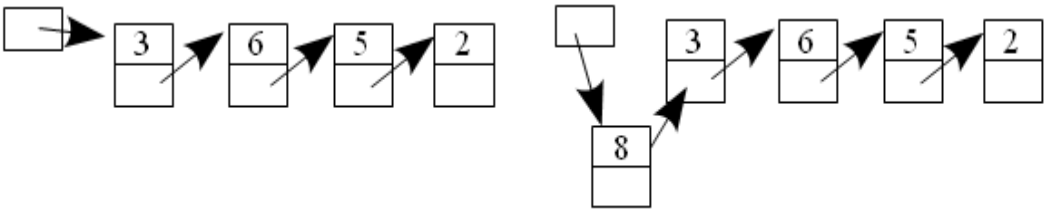


Il aura suffi de modifier des « flèches » (c'est-à-dire des « liens » ou des « accès »). L'opération prend donc le même temps quelle que soit la taille de la liste chainée (pour autant qu'on soit déjà positionné à l'endroit de l'ajout).

On voit que, si on connait l'endroit où l'élément doit être rajouté, la complexité de cette opération est $O(1)$. Il n'y a ici pas de décalage d'éléments comme pour un tableau, pour la simple raison que les éléments d'une liste chainée n'ont aucune raison d'être à des endroits contigus dans la mémoire de l'ordinateur.

Il y a un ajout qui fait figure de cas particulier, c'est l'ajout en tête de liste chainée, car il modifie obligatoirement l'accès au premier élément. Par exemple, si nous ajoutons 8 en première position, cela donne :

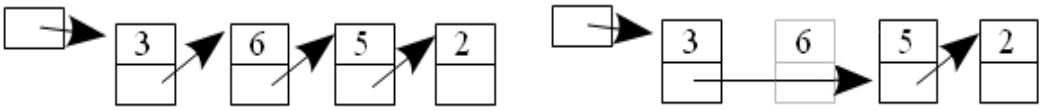
avant : après :



1.6 Suppression d'un élément dans une liste chainée

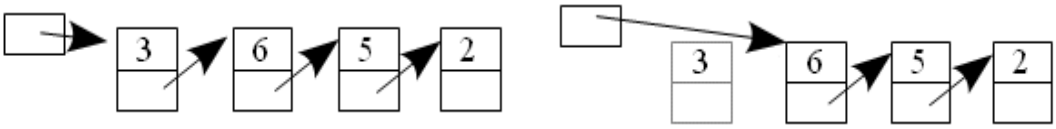
Lorsque la position de l'élément à supprimer est connue, la suppression d'un élément se fait tout aussi simplement. L'opération est également de complexité $O(1)$. Par exemple, supprimons l'élément de valeur 6 :

avant : après :



Ici aussi, le cas de la suppression du premier élément est particulier, car il modifie la tête de liste :

avant : après :



Remarquez que l'élément, quoique supprimé de la liste chainée, est encore présent sur les schémas. Bien que l'élément ne fasse plus logiquement partie de l'ensemble des éléments de la liste chainée, il est encore présent physiquement dans la mémoire. La suppression définitive de l'élément est gérée soit par le programmeur lui-même, soit c'est le système qui nettoie de la mémoire les éléments auxquels plus aucun accès ne mène. C'est le rôle du *garbage collector*, solution que nous avons adoptée en logique OO, mais qui n'est pas vraie pour tous les langages OO (cf. C++).

1.7 Recherche d'un élément

La recherche d'un élément dans une liste chainée non triée est similaire à celle d'un tableau, il faut obligatoirement tout parcourir pour savoir si l'élément est présent ou non.

Au contraire d'un tableau, le fait de savoir que la liste chainée est triée (sur les valeurs des éléments) ne permet pas de développer un algorithme rapide de type *recherche dichotomique*. Tout au plus on peut, comme avec un tableau, s'arrêter lorsqu'on dépasse la position où l'élément recherché aurait dû se trouver.

1.8 Liste chainée *versus* tableau

opération	tableau non trié	tableau trié	liste chainée non triée	liste chainée triée
ajout d'un élément	$O(1)$	$O(n)$	$O(1)$	$O(n)$
suppression d'un élément de valeur donnée	$O(n)$	$O(n)$	$O(n)$	$O(n)$
recherche d'un élément	$O(n)$	$O(\log_2 n)$	$O(n)$	$O(n)$
parcours	$O(n)$	$O(n)$	$O(n)$	$O(n)$

D'après ce tableau, la liste chainée ne semble pas beaucoup plus avantageuse qu'un tableau mais son intérêt réside dans le fait que l'ajout et la suppression d'éléments ne nécessite aucun décalage des éléments. Elle s'avère donc plus performante pour le type de problème nécessitant un grand nombre d'ajouts et de suppressions dans une collection de données.

Un autre avantage de la liste chainée est le gain de place en mémoire, en effet la liste n'occupe que la place requise pour les éléments qui la constituent. Il n'y a pas de problème de taille physique ou de débordement comme c'est le cas pour les tableaux. Théoriquement, sa taille est infinie.

Nous allons étudier l'implémentation de la liste chainée en OO, l'implémentation procédurale (ou *« C-like »*) est laissée à titre d'information.

1.9 Implémentation orienté-objet

1.9.1 Implémentation

Pour l'implémentation OO, nous allons définir deux classes, une pour un élément de liste chainée, et une autre pour la liste chainée elle-même.

Nous avons vu qu'un élément de liste est composé d'un attribut valeur (qui contient la valeur proprement dite de l'élément) et d'un attribut permettant d'accéder à l'élément suivant, que nous nommerons simplement suivant. Le type T de la valeur de l'élément est noté entre crochets.

```
classe ÉlémentListe<T>
 privé:
    valeur : T
    suivant : ÉlémentListe<T>
 public:
    constructeur ÉlémentListe<T>(val : T, elt : ÉlémentListe<T>)
    constructeur ÉlémentListe<T>(val : T)                          // suivant à rien.
    méthode getValeur() → T
    méthode setValeur(val : T)
    méthode getSuivant() → ÉlémentListe<T>          // renvoie rien si il n'y a pas de suivant.
    méthode setSuivant(elt : ÉlémentListe<T>)
fin classe
```

Remarquez la définition récursive de cette classe : l'attribut suivant est lui-même un objet de la classe dans laquelle il est défini.

```
classe ListeChainée<T>
 privé:
    premier : ÉlémentListe<T>
 public:
    constructeur ListeChainée<T>()                                    // crée une liste vide
    méthode getPremier() → ÉlémentListe<T>
    méthode setPremier(elt : ÉlémentListe<T>)
    méthode estVide() → booléen
    méthode insérerTête(val : T)
    //      insère en début de liste un nouvel élément de valeur val.
    méthode supprimerTête()
    //      supprime le premier élément de la liste chainée.
    méthode insérerAprès(elt : ÉlémentListe<T>, val : T)
    //      insère un nouvel élément de valeur val après elt.
    méthode supprimerAprès(elt : ÉlémentListe<T>)
    //      supprime l'élément qui suit elt de la liste chainée.
fin classe
```

1.9.2 Remarques :

- ▷ Pour la classe ListeChainée, on pourrait envisager l'ajout d'autres attributs pour accélérer certaines opérations. Par exemple, on pourrait définir comme attribut la taille de la liste chainée, ce qui serait redondant puisqu'on peut la calculer en parcourant les éléments. On parle d'attribut « calculable » pour indiquer un attribut qui contient une information qui pourrait être calculée à partir des autres attributs. La pratique montre qu'il ne faut pas abuser de ce type d'attribut.
- ▷ À votre avis, pourquoi n'y a t-il pas de méthode supprimer(elt : ÉlémentListe) qui supprime de la liste chainée l'élément donné en paramètre ?

1.9.3 Implémentation (suite)

Nous détaillons à présent le contenu des constructeurs et méthodes définies ci-dessus.

Pour la classe ÉlémentListe<T> :

```
constructeur ÉlémentListe<T>(val : T, elt : ÉlémentListe<T>)
    valeur ← val
    suivant ← elt
fin constructeur
```

```
constructeur ÉlémentListe<T>(val : T)
    valeur ← val
    suivant ← rien
fin constructeur
```

```
méthode getValeur() → T
    retourner valeur
fin méthode
```

```
méthode setValeur(val : T)
    valeur ← val
fin méthode
```

```
méthode getSuivant() → ÉlémentListe<T>
    retourner suivant
fin méthode
```

```
méthode setSuivant(elt : ÉlémentListe<T>)
    suivant ← elt
fin méthode
```

Passons à présent à la classe ListeChainée<T> :

```
constructeur ListeChainée<T>()
    premier ← rien
fin constructeur
```

```
méthode getPremier() → ÉlémentListe<T>
    retourner premier
fin méthode
```

```
méthode setPremier(elt : ÉlémentListe<T>)
    premier ← elt
fin méthode
```

```
méthode estVide() → booléen
    retourner premier = rien
fin méthode
```

```
méthode insérerTête(val : T)
    elt : ÉlémentListe<T>
    elt ← nouveau ÉlémentListe<T>(val, premier)
    premier ← elt
fin méthode
```

```
méthode supprimerTête()
    // les lignes en italique sont nécessaires si la libération de
    // l'espace inutilisé est à charge du programme (comme en C++)
    sauve : ÉlémentListe<T>
    si premier = rien alors
        erreur « la liste est vide »
    fin si
    sauve ← premier
    premier ← premier.getSuivant()

    libérer sauve
    // primitive qui permet de récupérer l'espace mémoire
    // qui était occupé par l'élément de liste sauve
fin méthode
```

```
méthode insérerAprès(elt : ÉlémentListe<T>, val : T)
    eltAInsérer : ÉlémentListe<T>
    si elt = rien alors
        erreur « elt est vide, pas d'insertion possible. »
    fin si
    eltAInsérer ← nouveau ÉlémentListe<T>(val, elt.getSuivant())
    elt.setSuivant(eltAInsérer)
fin méthode
```

```
méthode supprimerAprès(elt : ÉlémentListe<T>)
    // les lignes en italique sont nécessaires si la libération de
    // l'espace inutilisé est à charge du programme (comme en C++)
    sauve : ÉlémentListe<T>
    si elt = rien alors
        erreur « élément inexistant »
    sinon
        si elt.getSuivant( ) = rien alors
            erreur « pas de suivant »
        fin si
    fin si
    sauve ← elt.getSuivant()
    elt.setSuivant(elt.getSuivant().getSuivant())

    libérer sauve
fin méthode
```

1.9.4 Exemple : taille d'une liste chainée

Vous aurez remarqué que des méthodes « traditionnelles » des listes ne sont pas présentes dans la classe ListeChainée. Ces méthodes peuvent être écrites dans des algorithmes extérieurs à la classe.

Par exemple, écrivons l'algorithme calculant la taille d'une liste chainée.

```
algorithme tailleListeChainée(maListe : ListeChainée<T>) → entier
    courant : ÉlémentListe<T>
    cpt : entier
    cpt ← 0
    courant ← maListe.getPremier( )
    tant que courant ≠ rien faire
        cpt ← cpt + 1
        courant ← courant.getSuivant( )
    fin tant que
    retourner cpt
fin algorithme
```

Vous vous demandez peut-être pourquoi ne pas avoir écrit une méthode taille() directement dans la classe ListeChainée. C'est parce que cette méthode (comme d'autres) cache un parcours. En effet, imaginez avoir écrit la méthode taille() dans la classe ListeChainée. Vous allez probablement l'utiliser de la façon suivante :

```
i ← 1
tant que i ≤ maListeChainée.taille() faire
    // on fait quelque chose
    i ← i + 1
fin tant que
```

qui, à chaque appel de maListeChainée.taille(), va parcourir toute la liste chainée ! L'algorithme aura alors une complexité de n^2 alors qu'un parcours simple ne devrait avoir qu'une complexité de n. Il faut donc plutôt écrire :

```
courant ← maListeChainée.getPremier()
tant que courant ≠ rien faire
    // on fait quelque chose
    courant ← courant.getSuivant( )
fin tant que
```

comme on l'a fait ci-dessus.

1.10 Implémentation procédurale

1.10.1 Notations

Pour les besoins de cette implémentation, nous devons introduire une nouvelle notation : l'**accès** à une variable, ou encore **référence** ou **pointeur**.

La déclaration de cet accès se fera comme suit :

```
p : accès à T
```

ce qui signifie que p permet d'accéder à un élément de type T.

p désigne la variable qui contient l'adresse. Cette variable peut être utilisée en partie droite ou gauche d'une affectation concernant les adresses.

Dans le cadre de ce cours, comme les éléments d'une liste chainée peuvent être vus comme composés (structurés) d'un champ valeur et d'un champ accès, nous n'aurons que des accès à des structures mais les langages non objets permettent généralement des accès vers des éléments de type quelconque.

Dans le contexte des références, il faut pouvoir créer dynamiquement l'élément référencé. Cela s'indique par la primitive allouer, à utiliser comme suit :

```
p ← allouer (T)
```

où T est le type de l'élément référencé.

Exemple :

```
p ← allouer (Personne)
```

Cette instruction provoque l'allocation d'un espace mémoire pour une variable de type Personne et l'accès à cet espace est placé dans p. En C, p pourrait être un pointeur recevant l'adresse de l'espace alloué lors de cette demande d'allocation.

La variable accessible par p sera notée

```
p^
```

p^ désigne la variable dont l'adresse est rangée dans p. La variable p^ peut apparaitre en partie droite ou gauche d'une affectation concernant les variables de même type.

Si p^ est une variable structurée, pour accéder au champ truc de la variable accessible par p, on utilisera la notation suivante :

```
p^.truc
```

Pour récupérer l'espace mémoire inutilement occupé par une variable qui n'est plus utilisée, on a l'opération inverse de allouer qui se notera par la nouvelle primitive :

```
libérer p
```

qui supprime définitivement la variable d'accès p, ce qui a pour effet de rendre cet espace mémoire à nouveau allouable.

Comme nous l'avons déjà fait précédemment, un élément de liste est composé d'un champ valeur (qui contient la valeur proprement dite de l'élément) et d'un champ permettant d'accéder à l'élément suivant, que nous nommerons simplement suivant. Le type T de la valeur de l'élément est noté entre crochets :

```
structure ÉlémentListe<T>
    valeur : T
    suivant : accès à ÉlémentListe<T>
fin structure
```

Exemple : pour un élément de liste chainée d'entiers, on définirait la structure :

```
structure ÉlémentListe<entier>
    valeur : entier
    suivant : accès à ÉlémentListe<entier>
fin structure
```

En non orienté-objet, une liste chainée se reconnait simplement par le fait que des éléments semblables (de même type) sont chainés, ce qui signifie que chacun d'eux contient un accès à son suivant. Pour manipuler un tel ensemble de données, la seule condition est de pouvoir accéder au premier élément de cet ensemble par une variable de type accès, que l'on peut appeler premier, têteListe ou encore TL (et qui n'a nul besoin d'être structurée). La liste chainée n'existe donc pas physiquement, elle n'est donc rien d'autre qu'un accès, celui au premier élément. Si cette liste est vide, cet accès a pour valeur rien :

```
têteListe : accès à ÉlémentListe<T>                    // accès au premier élément, rien si vide
```

Exemple : pour déclarer une liste chainée d'entiers, on aurait :

```
têteListe : accès à ÉlémentListe<entier>
```

1.10.2 Exemple 1 : taille d'une liste chainée

Voici la version non OO d'un algorithme calculant le nombre d'éléments d'une liste chainée d'accès TL.

```
algorithme tailleListeChainée(TL : accès à ÉlémentListe<T>) → entier
    courant : accès à ÉlémentListe<T>
    cpt : entier
    cpt ← 0
    courant ← TL
    tant que courant ≠ rien faire
        cpt ← cpt + 1
        courant ← courant^.suivant
    fin tant que
    retourner cpt
fin algorithme
```

1.10.3 Exemple 2 : mise d'un fichier en liste chainée

Dans ce 2e exemple, on montre comment mettre le contenu d'un fichier de variables structurées Identité (nom, prénom : chaines, dateNais : Date) dans une liste chainée.

```
algorithme miseEnListe(Personnes : fichier de Identité) → accès à ÉlémentListe<Identité>
    courant, précédent : TL : accès à ÉlémentListe<Identité>
    enr : Identité
    TL ← rien                                                    // la liste est vide au départ
    ouvrir Personnes (IN)
    demander Personnes ; enr

    si NON EOF(Personnes) alors                  // traitement spécial pour le 1er enregistrement
        TL ← allouer (ÉlémentListe<Identité>)
        TL^.valeur ← enr
        TL^.suivant ← rien
        précédent ← TL
        demander Personnes ; enr
    fin si

    tant que NON EOF(Personnes) faire
        courant ← allouer (ÉlémentListe<Identité>)
        courant^.valeur ← enr
        courant^.suivant ← rien
        précédent^.suivant ← courant
        précédent ← courant
        demander Personnes ; enr
    fin tant que

    fermer Personnes
    retourner TL
fin algorithme
```

1.11 Représentation sans pointeur

La plupart des langages non orienté-objet possèdent la notion de pointeur mais il y a des exceptions (Cobol par exemple). Comment implémenter une liste chainée dans un tel langage ? En simulant par exemple la liste chainée via un tableau d'éléments contenant 2 champs (la valeur de l'élément de liste et l'indice de l'élément suivant), accompagné d'une variable tête (entier) qui indique l'indice du premier élément de la liste.

Exemple : la liste chainée (7, 6, 5, 2) pourrait être représentée ainsi :

tête

3

1	*2*	*3*	*4*	*5*
2	6	7	5	?
0	4	2	1	?

À condition de savoir aussi où commencer, c'est-à-dire connaitre l'indice du premier élément (le 3e dans notre exemple). L'indice 0 reprend le rôle de «rien» et indique la fin de la liste chainée. Pour ajouter un élément, il suffit de l'ajouter en fin de tableau et d'adapter les indices sur la deuxième ligne. La suppression est aussi aisée mais crée un « trou » qu'il faut pouvoir gérer (par exemple via une liste des places libres).

1.11.1 Commentaire

Dans l'implémentation procédurale, nous aurions pu plagier complètement la version OO où 2 classes ont été définies (ListeChainée et ÉlémentListe), il aurait fallu 2 structures correspondantes et les algorithmes correspondants aux constructeurs et méthodes. Ceci a été fait pour ElémentListe mais pas pour ListeChainée. À votre avis, pourquoi ?

1.12 Variantes : Liste bidirectionnelle et liste circulaire

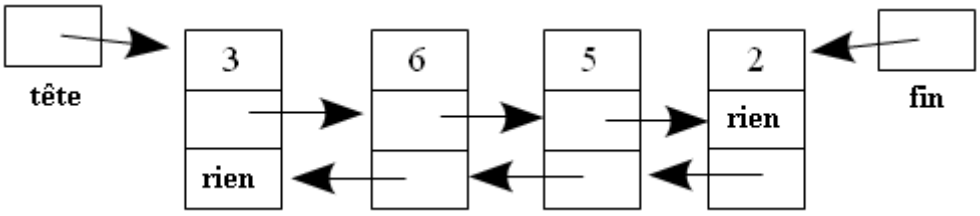
Il existe quelques variantes à la liste chainée, notamment la *liste bidirectionnelle* et la *liste circulaire*.

1.12.1 Liste bidirectionnelle

Dans une **liste bidirectionnelle**, chaque élément possède, en plus d'une référence à l'élément suivant, une référence à l'élément *précédent*. Un des avantages est de faciliter grandement la suppression car il n'est plus nécessaire de connaitre l'élément précédent pour pouvoir le faire puisqu'on peut le retrouver à partir de l'élément à supprimer.

Comme les liens entre les éléments permettent de se déplacer dans les deux sens, il est logique qu'à la tête de liste corresponde une fin de liste (ou queue de liste).

Schématiquement :



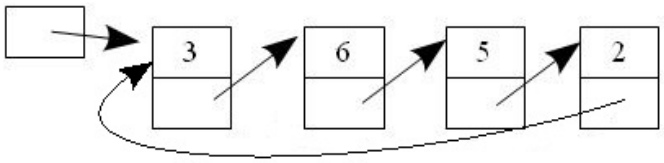
On peut facilement construire les classes ÉlémentListeBD et ListeBD en adaptant légèrement les classes pour la liste chainée simple. La classe ListeBD aura comme attributs les deux accès aux extrémités de la liste, nommés premier et dernier.

```
classe ÉlémentListeBD<T>
 privé:
    valeur : T
    suivant : ÉlémentListeBD<T>
    précédent : ÉlémentListeBD<T>
 public:
    constructeur ÉlémentListeBD<T>(val : T, suiv, prec : ÉlémentListeBD<T>)
    constructeur ÉlémentListeBD<T>(val : T)                    // précédent et suivant à rien
    méthode getValeur() → T
    méthode setValeur(val : T)
    méthode getSuivant() → ÉlémentListeBD<T>                   // renvoie rien si pas de suivant
    méthode setSuivant(elt : ÉlémentListeBD<T>)
    méthode getPrécédent() → ÉlémentListeBD<T>                 // renvoie rien si pas de précédent
    méthode setPrécédent(elt : ÉlémentListeBD<T>)
fin classe
```

```
classe ListeBD<T>
 privé:
    premier : ÉlémentListeBD<T>
    dernier : ÉlémentListeBD<T>
 public:
    constructeur ListeBD<T>()
    méthode getPremier() → ÉlémentListeBD<T>
    méthode setPremier(ÉlémentListeBD<T>)
    méthode getDernier() → ÉlémentListeBD<T>
    méthode setDernier(ÉlémentListeBD<T>)
    méthode estVide() → booléen
    méthode insérerTête(val : T)
    méthode insérerFin(val : T)
    méthode insérerAprès(elt : ÉlémentListeBD<T>, val : T)
    méthode insérerAvant(elt : ÉlémentListeBD<T>, val : T)
    méthode supprimerTête()
    méthode supprimerFin()
    méthode supprimer(elt : ÉlémentListeBD<T>)
fin classe
```

1.12.2 Liste circulaire

Dans une **liste circulaire**, le dernier élément ne référence pas « rien » mais indique le premier élément. Schématiquement :



Il n'est pas nécessaire de créer une classe nouvelle pour les éléments de la liste circulaire, puisqu'ils sont identiques à ceux d'une liste chainée simple. On peut faire fonctionner ce type de liste avec les outils développés pour la liste chainée, il faut simplement veiller ici à la cohérence de la liste circulaire, c'est-à-dire qu'à tout moment le dernier élément doit être lié au premier. Noter que premier a ici un sens différent du point de vue logique, dans une liste circulaire il n'y a pas réellement de *premier* élément, mais un accès à un élément quelconque de la liste, peu importe sa position.

1.13 Exercices

Remarques préliminaires

- ▷ *Dans les exercices qui suivent, vous serez souvent amenés à écrire des algorithmes similaires dont le code ne diffère qu'en un seul endroit. Dans cette situation, il faut toujours se demander s'il n'est pas possible de factoriser le code, c'est-à-dire de le rendre modulaire de façon à limiter l'écriture de code semblable.*
- ▷ *Dans les exercices sur les listes, on s'efforcera toujours de trouver la méthode la plus efficace (éviter de parcourir inutilement la liste) et aussi la plus économique (éviter si possible les nouvelles demandes d'allocation d'éléments si ce n'est pas nécessaire).*

1 Insertions, recherches et suppressions

Soit une liste chainée contenant des valeurs d'un type T quelconque. Écrire :

1. un algorithme qui ajoute dans la liste un nouvel élément dont la valeur val est donnée (à l'endroit qui permet le code le plus simple).

2. un algorithme qui recherche dans la liste la valeur val et retourne l'accès à l'élément qui le contient (ou *rien* si la valeur ne s'y trouve pas). Si la valeur s'y trouve plusieurs fois, l'algorithme en retourne la première occurrence.
3. un algorithme qui retourne un booléen indiquant si la liste contient la valeur val.
4. un algorithme qui supprime de la liste le premier élément contenant la valeur val. L'algorithme retournera un booléen indiquant si la suppression a été réalisée : vrai si val a été supprimé, et faux sinon (parce que la valeur ne s'y trouvait pas).
5. un algorithme qui supprime de la liste toutes les occurrences de la valeur val et retourne le nombre de suppressions réalisées.

2 Le grand nettoyage

Soit une liste chainée dont les valeurs (non triées) sont des entiers compris entre 0 et 9 inclus. On demande d'écrire l'algorithme qui supprime de cette liste toutes les occurrences de la valeur la plus fréquente. En cas d'*ex-æquo*, c'est la plus grande des valeurs qui est éliminée.

Exemple : la liste (8, 7, 6, 7, 7, 1, 7, 1, 5, 7) devient (8, 6, 1, 1, 5).

3 Liste chainée triée

Soit une liste chainée dont les éléments sont triés (les éléments sont d'un type T non précisé, mais on suppose que les opérateurs de comparaison (<, >, =, ...) sont autorisés pour ce type). Réécrire les algorithmes de l'exercice 1 en effectuant les changements optimisant le code et la performance.

4 Pas de doublons

Soit une liste chainée dont les éléments sont triés (idem que pour l'exercice précédent) mais **ne contenant pas de doublons**. Réécrire les algorithmes de l'exercice 3 qui nécessitent un changement ou une adaptation.

5 Liste bidirectionnelle

Développer le code des méthodes introduites dans la classe ListeBD.

6 Liste circulaire

Transformer la classe ListeChainée en ListeChainéeCirculaire, en ne changeant que les méthodes qui nécessitent une adaptation. Ajouter dans cette classe une méthode qui calcule la taille de la liste, et une méthode qui permet d'ajouter un élément entré en paramètre à l'emplacement qui permet le code le plus performant.

7 Tassement de liste

Soit une liste chainée d'entiers dont les éléments sont triés en ordre croissant sur les valeurs, et dans laquelle plusieurs éléments consécutifs peuvent avoir la même valeur. Écrire un algorithme qui supprime de cette liste toutes les valeurs redondantes.

Exemple : la liste (5, 5, 7, 7, 7, 11, 13, 13, 15) devient (5, 7, 11, 13, 15)

8 Inter-classement de listes

Soient deux listes chainées de même type d'éléments. On voudrait créer à partir de celles-ci une liste chainée issue de l'inter-classement des éléments de ces deux listes, c'est-à-dire contenant dans l'ordre le premier élément de la première liste, le premier élément de la seconde liste, le 2^{e} élément de la première liste, le 2^{e} élément de la seconde liste et ainsi de suite. Si une des deux listes est plus longue, on rattache le reste de celle-ci à la fin de la liste obtenue par l'inter-classement des premiers éléments.

Exemple : l'inter-classement des listes (2, 5, 9, 8, 10) et (7, 4, 3) donnera comme résultat la liste (2, 7, 5, 4, 9, 3, 8, 10)

Réaliser cet exercice :

1. En créant une 3^{e} liste (les deux listes de départ restent intactes)
2. Sans nouvelle demande d'allocation d'élément, c'est-à-dire en incorporant les éléments de la 2^{e} liste à la première.

9 Fusion de listes

Écrire un algorithme qui fusionne 2 listes chainées triées de même type d'éléments. Comme dans l'exercice précédent, on envisagera les cas ou (a) c'est une nouvelle liste qui est créée (b) la seconde liste est incorporée à la première sans nouvelle demande d'allocation d'élément.

10 Tri-fusion par éclatement des monotonies croissantes

Soit une liste chainée à valeurs entières. Une monotonie croissante est une suite d'éléments qui se suivent et dont les valeurs sont dans l'ordre croissant. Le tri-fusion consiste à extraire une monotonie, puis une autre, et ainsi de suite, et de fusionner chaque monotonie extraite avec le résultat des fusions précédentes. Après la dernière fusion, la liste obtenue sera triée, et la liste initiale aura donc été transformée en une liste triée. Réaliser l'algorithme.

11 Les ensembles

Définissons une classe Ensemble représentant un ensemble (au sens mathématique du terme) dont les éléments sont de type T. Développer le code des constructeurs et méthodes.

```
classe Ensemble<T>
 privé:
    v : aleurs : ListeChainée<T>
 public:
    constructeur Ensemble<T>()
    // crée un ensemble vide
    méthode ajouter(val : T)
    // ajoute une valeur à l'ensemble
    méthode contient(val : T) → booléen
    // indique si l'ensemble contient la valeur
    méthode union(autreEnsemble : Ensemble<T>) → Ensemble<T>
    // crée l'ensemble contenant tous les éléments des 2 premiers
    méthode intersection(autreEnsemble : Ensemble<T>) → Ensemble<T>
    // crée l'ensemble des éléments communs aux 2 premiers
    méthode différence(autreEnsemble : Ensemble<T>) → Ensemble<T>
    // retourne l'ensemble des éléments qui ne sont pas dans le 2eensemble
fin classe
```

12 L'algorithme du survivant

Le jeu du survivant se déroule comme suit : un nombre indéterminé de joueurs sont assis en cercle. Le meneur du jeu choisit un nombre entier au hasard (par ex. en lançant un dé) que nous appellerons le *pas de décompte*. Ensuite, il parcourt le cercle de joueurs en tournant dans le sens horloger derrière eux tout en comptant une unité à chaque joueur rencontré, et ce jusqu'à ce que pas de décompte est atteint. Le joueur derrière lequel il s'arrête est alors éliminé, et le meneur recommence son décompte à partir du joueur restant suivant. Il recommence ceci jusqu'à ce qu'il ne reste plus qu'un seul joueur, qui est le « survivant » de ce jeu.

Réaliser l'algorithme qui simule ce jeu. Les paramètres sont une liste chainée circulaire contenant les noms des joueurs, et le pas de décompte. On suppose que le premier élément auquel on accède dans la liste correspond au premier joueur compté par le meneur de jeu. L'algorithme retourne le nom du survivant.

13 Polynôme

Vous vous rappelez, par votre cours de mathématique, qu'un *monôme* est une expression de la forme ax^n, où a est le coefficient, x la variable et n l'exposant (ex. : $3x^2$). Un *polynôme* est alors une somme de monômes (ex. : $3x^4 - x^2 + 2$)

Représentation : convenons de représenter un monôme comme une simple structure avec 2 champs : le coefficient et l'exposant. Un polynôme sera alors représenté comme une liste chainée de monômes triés en ordre décroissant sur l'exposant.

Problème : écrire un algorithme permettant d'additionner 2 polynômes et un autre permettant de dériver un polynôme.

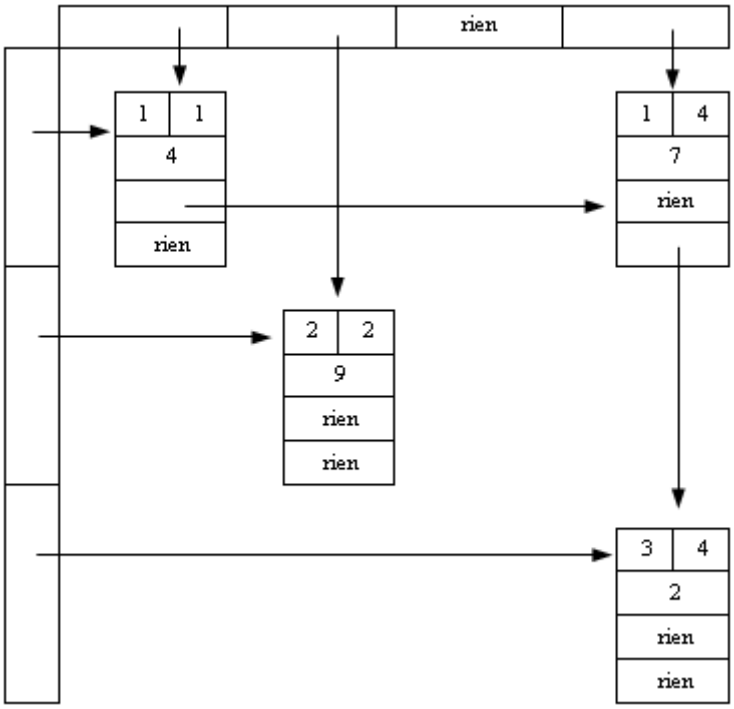
14 Matrice creuse

Une matrice creuse est une matrice où la plupart des éléments sont nuls. On les rencontre beaucoup dans des problèmes physiques impliquant des systèmes linéaires. Représenter une telle matrice par un tableau classique à 2 dimensions n'est efficace ni en terme d'espace mémoire (une matrice 1000 x 1000 va contenir un million de valeurs presque toutes nulles) ni en terme de temps de calcul (beaucoup de calculs avec des 0). C'est pourquoi on utilise souvent dans ce cas là un maillage de listes (une liste par ligne et une liste par colonne).

Exemple : La matrice :

4	0	0	7
0	9	0	0
0	0	0	2

sera représentée comme le suggère le schéma suivant :



Chaque élément de la liste contient le numéro de ligne et de colonne, la valeur, un lien vers l'élément non nul suivant dans la ligne et un lien vers l'élément non nul suivant dans la colonne. On dispose également d'un tableau pour les têtes de lignes et un tableau pour les têtes de colonnes.

On vous demande de :

1. Définir les attributs de la classe ÉlémentMatriceCreuse représentant un élément de la matrice.
2. Définir les attributs de la classe MatriceCreuse ainsi que

```
constructeur MatriceCreuse(nbLignes, nbColonnes : entiers)
// construit une matrice vide
méthode get(ligne, col : entiers) → entier
// donne la valeur de l'élément de la matrice en position (ligne, col)
méthode set(ligne, col, valeur : entiers)
// affecte une valeur à l'élément de la matrice en position (ligne, col) ;
// si la valeur est nulle il faudra peut-être supprimer un élément ;
// au contraire, si la valeur est non nulle, il faudra peut-être en ajouter.
```

Chapitre 2

L'interface

2.1 Définition

▷ *L'unité fondamentale utilisée dans le langage de programmation Java est la* ***classe****, mais l'unité fondamentale de la conception orientée objet est le* ***type****. Bien que des classes définissent des types, il est très utile et très puissant de pouvoir définir un type sans pour autant définir une classe. Une* ***interface*** *définit un type dans un format abstrait au travers de méthodes ou d'autres types qui forment le contrat pour ce type. Les interfaces ne contiennent aucune implémentation et* ***il n'est pas possible d'instancier une interface****. Ce sont les classes qui* ***implémentent*** *une ou plusieurs interfaces. Une interface est donc un élément de pure conception alors qu'une classe sert à la fois à la conception et à l'implémentation.* (Le langage Java 3e édition – Arnold, Gosling et Holmes – ed. Vuibert)

▷ *Une interface est une définition fonctionnelle d'un type de données.*

Les interfaces sont basées sur la distinction entre l'entête d'une méthode et l'implémentation de celle-ci. L'entête d'une méthode comprend toutes les informations nécessaires pour appeler cette méthode (le nom de la méthode, l'ensemble des paramètres qu'elle reçoit et le type de données qu'elle renvoie). L'implémentation d'une méthode comprend non seulement les informations de l'entête, mais aussi les instructions exécutables qui caractérisent le comportement de la méthode. La définition d'une interface ne contient que les entêtes de la méthode, et toute classe qui implémente l'interface doit donc définir les implémentations de la méthode.

Il est aussi possible de décrire une interface en disant qu'elle définit un type de données, au même titre qu'une classe. En conséquence, une interface peut être utilisée comme annotation de type, tout comme une classe. Mais à l'inverse d'une classe, il n'est pas possible d'instancier une interface. C'est en raison de cette distinction que l'on voit les interfaces comme des types de données abstraites et les classes comme des types de données concrètes. (adapté de http://www.bases-as3.fr/poo-heritage-polymorphisme-interface, août 2013)

2.2 Implémentation orienté-objet

2.2.1 Allure générale

En première année, nous avons défini la classe Moment. Toutefois, un doute planait sur le choix de l'implémentation : allions-nous définir un moment avec 3 attributs (heure, minute,

seconde), ce qui nous permettait d'écrire facilement les méthodes getHeure(), getMinute() et getSeconde() ; ou allions-nous n'utiliser qu'un seul attribut (le nombre total de secondes écoulées), ce qui nous simplifiait le calcul de la différence entre deux moments ?

Avant de choisir l'implémentation, nous nous sommes demandés ce que nous voulions comme méthodes dans cette classe :

```
méthode getJour() → entier                          // nb de jours dans une durée
méthode getHeure() → entier                         // entier entre 0 et 23 inclus
méthode getMinute() → entier                        // entier entre 0 et 59 inclus
méthode getSeconde() → entier                       // entier entre 0 et 59 inclus

méthode getTotalHeures() → entier                   // Le nombre total d'heures
méthode getTotalMinutes() → entier                  // Le nombre total de minutes
méthode getTotalSecondes() → entier                 // Le nombre total de secondes
```

Peu importe l'implémentation choisie, nous voulions qu'un Moment respecte ce contrat.

Ce contrat est ce qu'on appelle une **interface** :

```
interface Moment
    méthode getJour() → entier                      // nb de jours dans une durée
    méthode getHeure() → entier                     // entier entre 0 et 23 inclus
    méthode getMinute() → entier                    // entier entre 0 et 59 inclus
    méthode getSeconde() → entier                   // entier entre 0 et 59 inclus

    méthode getTotalHeures() → entier               // Le nombre total d'heures
    méthode getTotalMinutes() → entier              // Le nombre total de minutes
    méthode getTotalSecondes() → entier             // Le nombre total de secondes
fin interface
```

Vous voyez qu'on ne trouve ici que les entêtes des méthodes et aucun code à l'intérieur. Il est donc impossible de créer une instance de Moment.

À partir de là, nous pouvons écrire des classes qui vont implémenter cette interface.

```
classe Durée implémente Moment
 privé:
    totalSecondes : entier
 public:
    constructeur Durée(secondes : entier)
        si secondes < 0 alors
            erreur "paramètre négatif"
        fin si
        totalSecondes ← secondes
    fin constructeur
    constructeur Durée(heure, minute, seconde : entiers)
        si heure < 0 OU minute < 0 OU seconde < 0 alors
            erreur "un des paramètres est négatif"
        fin si
        totalSecondes ← 3600*heure + 60*minute + seconde
    fin constructeur

    méthode getJour() → entier                          // nb de jours dans une durée
        retourner totalSecondes DIV (3600*24)
    fin méthode

    méthode getHeure() → entier                         // entier entre 0 et 23 inclus
        // On doit enlever les jours éventuels
        retourner (totalSecondes DIV 3600) MOD 24
    fin méthode
    méthode getMinute() → entier                        // entier entre 0 et 59 inclus
        // On doit enlever les heures éventuelles
        retourner (totalSecondes DIV 60) MOD 60
    fin méthode
    méthode getSeconde() → entier                       // entier entre 0 et 59 inclus
        // On doit enlever les minutes éventuelles
        retourner totalSecondes MOD 60
    fin méthode

    méthode getTotalHeures() → entier                   // Le nombre total d'heures
        retourner totalSecondes DIV 3600
    fin méthode
    méthode getTotalMinutes() → entier                  // Le nombre total de minutes
        retourner totalSecondes DIV 60
    fin méthode
    méthode getTotalSecondes() → entier                 // Le nombre total de secondes
        retourner totalSecondes
    fin méthode
fin classe
```

```
classe MomentHMS implémente Moment
 privé:
    jour : entier
    heure : entier
    minute : entier
    seconde : entier
 public:
    constructeur MomentHMS(secondes : entier)
        jour ← secondes DIV (3600*24)
        heure ← (secondes DIV 3600) MOD 24
        minute ← (secondes DIV 60) MOD 60
        seconde ← secondes MOD 60
    fin constructeur
    constructeur MomentHMS(unJour, uneHeure, uneMinute, uneSeconde : entiers)
        jour ← unJour
        heure ← uneHeure
        minute ← uneMinute
        seconde ← uneSeconde
    fin constructeur

    méthode getJour() → entier
        retourner jour
    fin méthode
    méthode getHeure() → entier                          // entier entre 0 et 23 inclus
        retourner heure
    fin méthode
    méthode getMinute() → entier                         // entier entre 0 et 59 inclus
        retourner minute
    fin méthode
    méthode getSeconde() → entier                        // entier entre 0 et 59 inclus
        retourner secondes
    fin méthode

    méthode getTotalHeures() → entier                    // Le nombre total d'heures
        retourner jour * 24 + heure
    fin méthode
    méthode getTotalMinutes() → entier                   // Le nombre total de minutes
        retourner getTotalHeures() * 60 + minute
    fin méthode
    méthode getTotalSecondes() → entier                  // Le nombre total de secondes
        retourner getTotalMinutes() * 60 + seconde
    fin méthode
fin classe
```

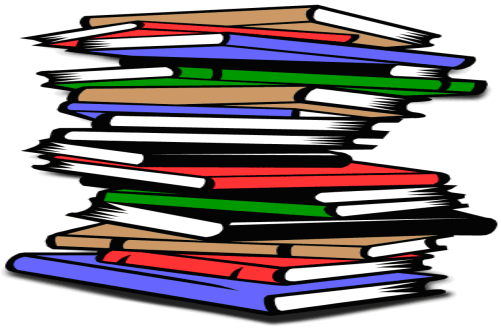
2.3 Exercices

1 Les listes

Quelle interface écririez-vous pour les listes ?

Chapitre 3

La pile



3.1 Définition

Une **pile** est une collection d'éléments admettant les fonctionnalités suivantes :

- ▷ on peut toujours ajouter un élément à la collection ;
- ▷ seul le dernier élément ajouté peut être consulté ou enlevé ;
- ▷ on peut savoir si la collection est vide.

La pile (en anglais *stack*) est donc une collection de données de type *dernier entré, premier sorti* (en anglais on dit « LIFO », c'est-à-dire *last in first out*). L'analogie avec la pile de dossiers sur un bureau est claire : les dossiers sont déposés et retirés du sommet et on ne peut jamais ajouter, retirer ou consulter un dossier qui se trouve ailleurs dans la pile.

On ne peut donc pas parcourir une pile, ou consulter directement le *n-ième* élément. Les opérations permises avec les piles sont donc peu nombreuses, mais c'est précisément là leur

spécificité : elles ne sont utilisées en informatique que dans des situations particulières où seules ces opérations sont requises et utilisées. Paradoxalement, on implémentera une pile en restreignant des structures plus riches aux seules opérations autorisées par les piles. Cette restriction permet de n'utiliser que les fonctionnalités nécessaires de la pile, simplifiant ainsi son utilisation.

Des exemples d'utilisations sont la gestion de la mémoire par les micro-processeurs, l'évaluation des expressions mathématiques en notation polonaise inverse, la fonction « ctrl-Z » dans un traitement de texte qui permet d'annuler les frappes précédentes, la mémorisation des pages web visitées par un navigateur, etc. Nous les utiliserons aussi plus loin dans ce cours pour parcourir les arbres et les graphes.

3.2 Implémentation orienté-objet

3.2.1 Allure générale

Nous allons d'abord décrire l'allure générale de l'interface Pile.

Il existe plusieurs possibilités de classes implémentant cette interface. Nous les détaillerons à titre d'exercice.

```
interface Pile<T>                                    // T est le type des éléments de la pile
    méthode empiler(élément : T)                     // ajoute un élément au sommet de la pile
    méthode sommet() → T
    //      retourne la valeur de l'élément au sommet de la pile, sans le retirer
    méthode dépiler() → T                            // enlève et retourne l'élément au sommet
    méthode estVide() → booléen                      // indique si la pile est vide
fin interface
```

3.2.2 Remarques :

- ▷ Théoriquement, et dans la majorité des utilisations, la pile est *infinie*, c'est-à-dire qu'on peut y ajouter un nombre indéterminé d'éléments (comme c'était le cas pour la classe Liste étudiée en 1ère année). Dans certaines situations, on peut cependant imposer une capacité maximale à la pile (en pratique, c'est le cas de la pile de dossiers dans un bureau, elle est forcément limitée par la hauteur du plafond !). Nous aborderons ce cas particulier dans les exercices.
- ▷ Lors de l'implémentation de la classe, il faudra songer à envoyer un message d'erreur lorsqu'on utilise les méthodes *sommet* et *dépiler* si la pile est vide. Si la pile possède une taille maximale, alors c'est *empiler* qui doit générer une erreur lorsque la pile est pleine.
- ▷ Nous avons utilisé ici des noms de méthodes neutres indépendants de tout langage de programmation. Dans la littérature anglaise, on trouvera souvent *push*, *top* et *pop* en lieu et place de *empiler*, *sommet* et *dépiler*.

3.3 Exemple d'utilisation

Afin d'illustrer l'utilisation d'une classe implémentant l'interface Pile, nous donnons pour exemple un algorithme qui lit une suite d'enregistrements d'un fichier **fileIn** (de type **Info**) et les reproduit en ordre inverse dans le fichier **fileOut**.

```
algorithme inverserOrdre(fileIn↓ : FichierEntrée d'Info, fileOut↑ : FichierSortie d'Info)
    enr : Info
    // on déclare la pile du type de l'interface Pile
    pile : Pile<Info>
    // on la crée du type de la classe qui implémente l'interface Pile
    pile ← nouveau ClassePile<Info>

    // 1ère étape : parcours du fichier et mise en pile
    fileIn.ouvrir( )
    enr ← fileIn.lire( )
    tant que NON fileIn.EOF( ) faire
        pile.empiler(enr)
        enr ← fileIn.lire( )
    fin tant que
    fileIn.fermer( )

    // 2e étape : vider la pile et écrire les éléments dans le fichier de sortie
    fileOut.ouvrir( )
    tant que NON pile.estVide( ) faire
        enr ← pile.dépiler( )
        fileOut.écrire(enr)
    fin tant que
    fileOut.fermer( )
fin algorithme
```

3.4 Exercices

1 Implémentation via une liste chainée

Détaillez l'implémentation de la classe PileListe en utilisant comme attribut privé une liste chainée. Veillez à utiliser les méthodes qui permettent la gestion la plus efficace.

2 La pile de taille finie

On envisage ici une pile dont la capacité est limitée : elle ne peut contenir au plus qu'un nombre donné d'éléments. On demande d'implémenter ce type de pile en utilisant un tableau. Les attributs privés de la classe seront les suivants :

```
classe PileTab<T> implémente Pile<T>
 privé:
    tab : tableau<T>                    // tableau (dynamique) contenant les éléments de la pile
    indSommet : entier                                  // indice du sommet de la pile
    tailleMax : entier                                    // taille maximale de la pile
fin classe
```

On remplira le tableau en ajoutant les éléments les uns à la suite des autres et en retenant la position du dernier élément, qui correspondra à la valeur de l'attribut *indSommet*. Cet attribut augmentera lorsqu'on ajoute un élément, et va décroître lorsqu'on en retire un.

Détaillez l'implémentation des méthodes de la classe PileTab correspondant à cette situation. Il faudra aussi adapter le constructeur, en lui donnant comme paramètre la taille maximale de la pile.

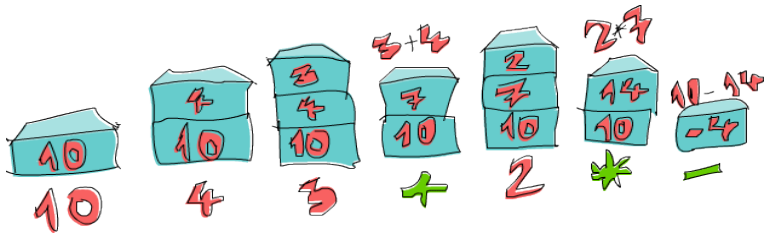
3 L'itinéraire retour

Un fichier **aller** contient la description d'un itinéraire. Chaque enregistrement du fichier est une structure **Étape** contenant les champs **ville** (chaine) et **km** (réel). Écrire un algorithme qui crée le fichier **retour** qui contiendra – au même format que **aller** – la description de l'itinéraire retour.

Exemple :

aller		*retour*	
Bruxelles	0	Amsterdam	0
Antwerpen	40	Utrecht	50
Breda	100	Breda	120
Utrecht	170	Antwerpen	180
Amsterdam	220	Bruxelles	220

4 Notation polonaise inverse



La notation polonaise inverse (encore appelée notation postfixée) consiste à donner tous les opérandes d'une opération avant de donner l'opérateur lui-même. Le résultat de l'opération devient un opérande pouvant à son tour être utilisé comme nouvel opérande d'une opération.

L'intérêt de cette notation est qu'il ne nécessite pas de parenthèse à condition d'écrire les opérandes dans le bon ordre.

Par exemple,

- 1 + 2 s'écrit « 1 2 + »
- (1 * 2) + 3 s'écrit « 1 2 * 3 + »
- 1 * (2 + 3) s'écrit « 1 2 3 + * »
- (4 + 3) * (3 + 2) s'écrit « 4 3 + 3 2 + * »
- 4 + (3 * 3) + 2 s'écrit « 4 3 3 * + 2 + »
- 32 - 2 * (12 - 3 * (7 - 2)) s'écrit « 32 2 12 3 7 2 - * - * - »

Reprenons l'exemple « 4 3 3 * + 2 + » : une pile contiendra successivement

		3			
	3	3	9	2	
4	4	4	4	13	15

À l'aide d'une Pile<entiers>, écrivez un algorithme qui interprète les opérations arithmétiques simplifiées aux entiers en notation polonaise inverse.

Votre algorithme recevra en paramètre un tableau de chaines et retournera un entier, le résultat de l'opération.

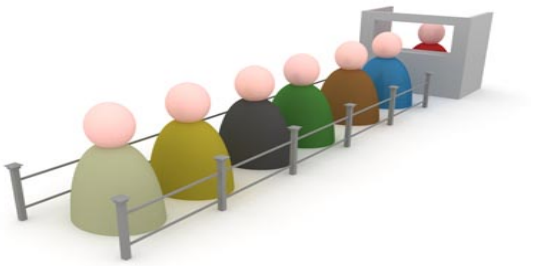
Vous pouvez utiliser les algorithmes

- ▷ **estNombre(c : chaine)** qui retourne vrai si c peut être converti en un nombre et faux sinon.
- ▷ **nombre(c :chaine)** qui convertit c en l'entier correspondant

Nous supposerons le tableau entré comme ne comportant pas d'erreur et n'utilisant que des opérateurs binaires +, -, * et / (pour la division entière).

Chapitre 4

La file



4.1 Définition

Une **file** est une collection d'éléments admettant les fonctionnalités suivantes : 

- ▷ on peut toujours ajouter un élément à la collection
- ▷ seul le premier élément ajouté peut être consulté ou enlevé
- ▷ on peut savoir si la collection est vide

La file (en anglais *queue*) est donc une collection de données de type *premier entré, premier sorti* (en anglais on dit « FIFO », c'est-à-dire *first in first out*). L'analogie avec une file de clients à un guichet (poste, caisse du supermarché, ...) est évidente : c'est le premier arrivé qui est le premier servi, et il est très malvenu d'essayer de doubler une personne dans une file ! Noter qu'une fois entré dans une *file* – au sens informatique du terme – on ne peut pas en sortir par l'arrière, le seul scénario possible pour en sortir est d'attendre patiemment son tour et d'arriver en tête de la file.

De même que pour la pile, on ne peut donc pas non plus parcourir une file, ou consulter directement le *n-ième* élément. Les files sont très utiles en informatique, citons par exemple la création de mémoire tampon (*buffer*) dans de nombreuses applications, les processeurs multitâches qui doivent accorder du temps-machine à chaque tâche, la file d'attente des impressions pour une imprimante, ...

4.2 Implémentation orienté-objet

4.2.1 Allure générale

Comme pour la pile, l'interface File ne contient qu'un nombre restreint de méthodes qui correspondent aux quelques opérations permises avec cette structure : ajouter un élément (*« enfiler »*), consulter l'élément de tête, et le retirer (*« défiler »*). Comme précédemment, le détail des classes implémentant l'interface sera laissé à titre d'exercices.

```
interface File<T>                                          // T est le type des éléments de la file
    méthode enfiler(élément : T)                           // ajoute un élément dans la file
    méthode tête() → T          // retourne la valeur de l'élément en tête de file, sans le retirer
    méthode défiler() → T                                  // enlève et retourne l'élément de tête
    méthode estVide() → booléen                            // indique si la file est vide
fin interface
```

4.2.2 Remarques :

- ▷ De même que dans le chapitre précédent, la file est supposée *infinie*, c'est-à-dire qu'on peut y ajouter un nombre indéterminé d'éléments. Le cas de la file limitée à une capacité maximale sera envisagé dans l'exercice 2.
- ▷ Dans l'implémentation, il faudra songer à envoyer un message d'erreur lorsqu'on utilise les méthodes *tête* et *défiler* si la file est vide. Si la file possède une taille maximale, alors c'est *enfiler* qui doit générer une erreur lorsque la file est pleine.

4.3 Exercices

1 Implémentation via une liste chainée bidirectionnelle

Détaillez l'implémentation de la classe file en utilisant comme représentation des données une liste chainée bidirectionnelle.

2 La file de taille limitée

La file de taille limitée ne peut contenir au plus qu'un nombre donné d'éléments. On demande d'implémenter ce type de file en utilisant un tableau. Les attributs privés de la classe seront les suivants :

```
classe FileTab<T> implémente File<T>
 privé:
    tab : tableau<T >                  // tableau (dynamique) contenant les éléments de la file
    premier : entier                          // indice du premier élément entré (tête de file)
    dernier : entier                            // indice du dernier élément entré (fin de file)
    nbMaxÉlément : entier                          // nombre maximum d'éléments de la file
fin classe
```

Nous proposons ici d'offrir un constructeur qui reçoit le nombre maximum d'éléments de la file et qui crée un tableau d'indices 0 à nbMaxÉlément (le tableau aura donc un élément de plus que le nombre maximum d'éléments de la file, nous allons expliquer pourquoi). On remplit le tableau en ajoutant les éléments les uns à la suite des autres en retenant la position du premier et du dernier via deux indices. Lorsqu'on ajoute un élément, la position du dernier va augmenter, et lorsqu'on enlève un élément, cela revient à augmenter la position du premier, afin d'éviter de devoir retasser l'ensemble de la file en début de tableau, ce qui serait une perte d'efficacité.

Exemple :

Supposons qu'après la création de la file, les éléments 4, 3, 6 et 2 ont été ajoutés.

L'indice du premier est 0 et celui du dernier est 3.

4	3	6	2						

On ajoute l'élément 7. L'indice du dernier est à présent 4.

4	3	6	2	7					

On enlève un élément de la file. L'indice du premier vaut maintenant 1

	3	6	2	7					

Ce mécanisme peut se faire de façon circulaire : arrivés en fin de tableau, les indices premier et dernier repassent à 0. Lorsque premier et dernier sont égaux, la file n'a qu'un seul élément. Si cet élément est supprimé, l'incrémentation de premier aura pour conséquence que l'indice premier sera d'une unité supérieur à l'indice dernier (en tenant compte de la rotation des valeurs), et ce sera donc la condition correspondant à une file vide. Le tableau ne sera jamais rempli au maximum pour pouvoir distinguer le cas d'une file vide et le cas d'une file pleine. Nous placerons donc dans le tableau au maximum nbMaxÉlément éléments (en laissant un élément « vide » entre les indices dernier et premier).

3 Le monte-charge

Un monte-charge de chantier assure le transport de personnes et de matériel entre deux niveaux. Lorsque le monte-charge arrive à un étage, tous ses passagers en sortent. La charge maximale admise a la valeur MAX (donnée en paramètre). Étant donné les risques liés à l'utilisation du monte-charge en surcharge, le poids de chacun des passagers et des charges qu'ils transportent (outils, brouette de sable, ...) est vérifié avant l'entrée dans le monte-charge.

Pour gérer l'embarquement dans le monte-charge, on souhaite faire appel à un algorithme « embarquement » qui détermine combien de personnes qui se trouvent actuellement en file devant le monte-charge peuvent y entrer.

Cet algorithme respectera l'ordre d'arrivée des personnes et mettra à jour la file des personnes en attente qu'il aura reçue en paramètre.

1. Choisissez une représentation de données adaptée à la solution du problème et justifiez votre choix.
2. Écrivez l'algorithme « embarquement »
3. Écrivez l'algorithme « arrivée » qui ajoute à la file une personne d'un poids donné en paramètre.
4. Quels cas doit-on envisager ?

4 À l'envers

Écrire un algorithme recevant en paramètre une file d'entiers. À l'issue de cet algorithme, les valeurs de la file seront en ordre inverse et débarrassées des éléments pairs.

Exemple : Si le contenu de la file est le suivant :

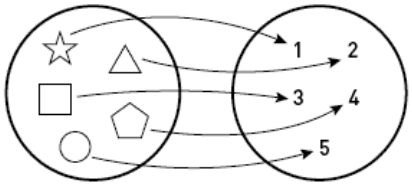
$$1 \to 0 \to 4 \to 25 \to 20 \to 11 \to 3 \to 7 \to 8 \to 73 \to 2 \to 4$$

son contenu sera après traitement :

$$73 \to 7 \to 3 \to 11 \to 25 \to 1$$

Chapitre 5

L'association



5.1 Introduction

Dans ce chapitre, nous considérons des ensembles de données possédant une **clé** (aussi appelée **identifiant**). Par définition, la clé est unique pour chaque élément de l'ensemble.

Vous connaissez déjà de nombreux exemples par le cours de bases de données, citons entre autres :

- ▷ le numéro d'étudiant, unique pour chaque étudiant de l'ESI ;
- ▷ le numéro de châssis d'un véhicule automobile ;
- ▷ le numéro de catalogue des produits vendus dans une grande surface ;
- ▷ le code postal des communes de Belgique ; etc.

Pour pouvoir stocker les éléments de ce type d'ensemble dans le cadre de processus qui requièrent des accès fréquents aux données, nous devons recourir – parmi les structures de données vues jusqu'ici – au tableau ou à la liste (la liste non chainée vue en 1ère année). L'inconvénient de ces structures réside dans le fait de devoir localiser les données par un indice, qui est dans la plupart des cas un numéro d'ordre arbitraire qui n'est pas directement lié à la clé ; il faut donc toujours accompagner la structure choisie d'un outil de recherche, ce qui implique un parcours des données pour chacune des opérations de bases telles que la modification, la consultation ou la suppression d'un élément.

Prenons l'exemple des étudiants de l'ESI dont l'identifiant est un entier de 5 chiffres (par ex. 35421). Si on stocke les données dans un tableau par ordre alphabétique, il est évident que l'indice de la case où se trouve un étudiant donné n'aura aucun rapport direct avec son numéro d'étudiant. De même si on trie les données sur les numéros d'étudiant, rien ne permet de le localiser directement, si ce n'est dans ce cas que la recherche pourrait être accélérée (par une recherche dichotomique par ex.). On pourrait aussi envisager de placer un étudiant directement dans la case correspondant à son numéro d'étudiant (ainsi, l'étudiant 35421 serait dans la case 35421) mais ce n'est pas une idée très heureuse car elle exigerait d'utiliser un tableau énorme dont la majorité des cases seraient inutilisées (il n'y a pas 35000

étudiants à l'ESI !) et de plus le tableau contiendrait de nombreux trous (il n'y a pas un étudiant pour chacun des numéros dans un intervalle donné, certains numéros disparaissent, par exemple suite à un abandon).

Nous allons, dans ce chapitre, construire une structure qui permet de faire la liaison directe entre un élément et sa clé, et qui nous dispensera du problème « technique » de devoir rechercher cet élément dans la structure. La connaissance de sa clé devrait permettre de le localiser immédiatement par un algorithme de complexité minimale (en $O(1)$ plutôt qu'en $O(n)$).

5.2 Définition

Une **association** (on dit aussi *dictionnaire* ou encore *map* selon la terminologie anglaise) lie des éléments à leur clé (ou identifiant). Il s'agit d'une structure de données qui contient des couples (clé, valeur) et qui autorise les opérations suivantes : stocker, retirer et modifier un élément à partir de sa clé, connaitre le nombre d'éléments de l'ensemble (c'est-à-dire le nombre de couples), et obtenir la liste des clés des éléments présents dans la structure.

5.3 Description orienté-objet

Nous allons définir une interface Map (pour des raisons pratiques, nous optons pour le nom anglais en raison de sa brièveté) dont les méthodes réalisent les opérations citées ci-dessus. Les éléments contenus dans cette « map » sont des couples (clé, valeur). Plus précisément ce sont des variables structurées dont les champs sont d'une part la clé (de type K, le plus souvent un entier ou une chaine de caractère) et d'autre part la valeur (de type T quelconque). En particulier, la clé peut être elle-même un champ de la valeur, ce qui peut paraitre redondant, mais nécessaire pour le fonctionnement des classes implémentant cette interface.

```
interface Map<K, T>
    // K est le type de la clé
    // et T celui de la valeur
    méthode setÉlément(clé : K, val : T)    // ajoute ou modifie le couple dont la clé est donnée
    méthode getValeur(clé : K) → T                          // retourne la valeur associée à la clé
    méthode supprimer(clé : K)                                 // retire le couple associé à la clé
    méthode contient(clé : K) → booléen        // indique si le couple dont la clé est donnée est
présent
    méthode taille() → entier                                          // retourne le nombre de couples
    méthode listeClés() → Liste<K>                          // retourne la liste des clés présentes
fin interface
```

La méthode setÉlément peut éventuellement écraser une valeur précédente si la clé était déjà présente. Les méthodes getValeur et supprimer génèrent un message d'erreur si la clé en paramètre est absente de l'ensemble. La dernière méthode est nécessaire pour pouvoir parcourir les éléments de l'ensemble. En effet, sa structure interne sera un attribut privé inconnu à l'extérieur de la classe, il est nécessaire d'avoir un outil donnant la liste des clés présentes. En parcourant la liste et en utilisant la méthode getValeur, on pourra ainsi passer en revue tous les éléments de l'association. Il est à noter que la liste retournée n'est pas forcément dans l'ordre des clés, c'est une limitation de cette structure.

5.4 Exemple d'utilisation

Illustrons par un exemple l'utilité d'une classe implémentant Map. Supposons qu'on veuille faire des statistiques sur les marques de voitures d'un grand parc automobile. On voudrait connaitre le nombre de véhicules pour chacune des marques présentes. Les données sont contenues dans un fichier **fichAuto** dont chaque enregistrement de type **Voiture** contient

les champs **marque** (chaine), **modèle** (chaine), **immatriculation** (chaine), et d'autres données qui ne sont pas utiles ici.

Écrivons une première version sans utiliser une classe implémentant Map. Nous avons besoin d'un compteur associé à chacune des marques de voitures. Il est possible d'utiliser un tableau, mais avec l'inconvénient de ne pas savoir à l'avance le nombre de marques différentes (ce qui pourrait se solutionner en déclarant le tableau avec une taille raisonnablement grande dans ce contexte, 500 par exemple). Nous allons plutôt opter pour une liste (classe Liste), dont les éléments seront une structure **eltListe** contenant les champs **marque** (chaine) et **cpt** (entier). La démarche est simple : lorsqu'une nouvelle marque est rencontrée, il faut ajouter un compteur dans la liste initialisé à 1 ; si la marque a déjà été rencontrée, il faut retrouver le compteur associé à cette marque et l'incrémenter. Un algorithme de recherche est donc indispensable pour le bon fonctionnement.

```
algorithme statVoitures(fichAuto : FichierEntrée de Voiture)
    i, ind : entier
    enr : Voiture
    elt : eltListe
    listeCpt : Liste<eltListe>
    listeCpt ← nouveau Liste<eltListe>( )
    fichAuto.ouvrir( )
    enr ← fichAuto.lire( )
    tant que NON fichAuto.EOF( ) faire
        ind ← recherche(listeCpt, enr.marque)
        si ind = 0 alors                                  // la marque n'a pas encore été comptée
            elt.marque ← enr.marque
            elt.cpt ← 1
            listeCpt.ajouter(elt)
        sinon                                             // la marque est présente dans la liste
            elt← listeCpt.get(ind)
            elt.cpt ← elt.cpt + 1
            listeCpt.set(ind, elt)
        fin si
        enr ← fichAuto.lire( )
    fin tant que
    fichAuto.fermer( )
    pour i de 1 à listeCpt.taille( ) faire
        afficher listeCpt.get(i).marque, listeCpt.get(i).cpt
    fin pour
fin algorithme
```

```
algorithme recherche(liste : Liste<eltListe>, marque : chaine) → entier
    // recherche dans la liste l'élément correspondant à la marque en paramètre
    // et retourne sa position dans la liste, 0 s'il ne s'y trouve pas.
    i : entier
    trouvé : booléen
    trouvé ← faux
    i ← 0
    tant que NON trouvé ET i < liste.taille( ) faire
        i ← i + 1
        trouvé ← liste.get(i).marque = marque
    fin tant que
    si trouvé alors
        retourner i
    sinon
        retourner 0
    fin si
fin algorithme
```

Avec une classe implémentant Map, le code se simplifie considérablement : nous sommes débarassés de l'algorithme de recherche, et du souci de connaitre à quel endroit se trouve le

compteur associé à une marque donnée. Les valeurs seront ici les compteurs, et les clés les marques de voitures. On utilise donc une Map<chaine, entier> :

```
algorithme statVoitures(fichAuto : FichierEntrée de Voiture)
    enr : Voiture
    i : entier
    liste : Liste<chaine>
    compteurs : Map <chaine, entier>
    compteurs ← nouveau ClasseMap<chaine, entier>( )
    fichAuto.ouvrir( )
    enr ← fichAuto.lire( )
    tant que NON fichAuto.EOF( ) faire
        si compteurs.contient(enr.marque) alors
            // la marque est déjà présente dans la Map
            compteurs.setÉlément(enr.marque, compteurs.getValeur(enr.marque) + 1)
        sinon
            // la marque n'est pas encore dans la Map
            compteurs.setÉlément(enr.marque, 1)
        fin si
        enr ← fichAuto.lire( )
    fin tant que
    fichAuto.fermer( )
    liste ← compteurs.listeClés( )
    pour i de 1 à liste.taille( ) faire
        afficher liste.get(i), compteurs.getValeur(liste.get(i))
    fin pour
fin algorithme
```

5.5 Implémentation

On voit, par l'exemple précédent, que l'association apparait de l'extérieur comme un ensemble non ordonné, une grande boite dans laquelle on introduit les couples (clé, valeur) et de laquelle on peut les récupérer très facilement. Comment cela fonctionne-t-il ?

Pour implémenter une association, il est bien sûr possible d'utiliser les structures déjà connues :

- ▷ un tableau (trié ou non) ou une liste dont les éléments sont les couples (clé, valeur) comme dans l'exemple des statistiques des marques de voitures
- ▷ une liste chainée (triée ou non)
- ▷ un arbre binaire ordonné sur les clés (voir chapitre sur les arbres)

Utiliser une de ces structures comme attribut de la classe reviendrait dès lors à camoufler à l'intérieur de la classe un algorithme de recherche tel que celui qui apparait explicitement dans l'exemple ci-dessus, ce qui ne serait pas un réel progrès quant à l'efficacité.

Il existe une structure particulière très efficace, nommée *table de hachage*, qui utilise un tableau pour stocker les éléments, et la position d'un élément dans ce tableau se retrouve très rapidement en utilisant une *fonction de hachage*.

5.6 La fonction de hachage

Une *fonction de hachage* transforme une clé en un « petit » entier. Le but est d'obtenir par cette fonction (que nous noterons $h(x)$) des valeurs entre 1 (ou 0) et une valeur maximale n, et qui seront réparties le plus uniformément possible dans cet intervalle. Le choix de la fonction de hachage dépend du type de clé et aussi de la taille des éléments à classer.

Cependant, il est courant que l'ensemble d'arrivée de la fonction soit plus petit que l'ensemble des clés, et il est donc inévitable que deux clés différentes donnent le même nombre après

hachage. Dans ce cas on parle de « collision ». Lors du choix de la fonction de hachage, il faut également chercher à minimiser ces collisions, en vue d'un fonctionnement performant de la classe.

Exemples :

- ▷ Prenons l'ensemble des étudiants de l'ESI, avec comme clé le numéro d'étudiant de 5 chiffres. La fonction $h(x) = x$ DIV 1000 ne serait pas un bon choix car elle donnerait un nombre réduit de valeurs (comprises actuellement entre 35 et 40), à partir d'un ensemble de plusieurs centaines d'étudiants. La fonction $h(x) = x$ MOD 100 est déjà bien meilleure, elle donnerait des valeurs entre 0 et 99. Il y aura bien sûr des collisions, vu que les étudiants de numéros 37156 et 38956 seraient « hachés » de la même façon.
- ▷ Pour les codes postaux des villes de Belgique, on pourrait prendre la fonction $h(x) = x$ DIV 10. Elle donnerait un ensemble de valeurs entre 100 et 999 où les collisions seraient peu nombreuses (vu que la plupart des codes se terminent par 0).

5.7 La table de hachage

La *table de hachage* est un tableau contenant les couples (clé, valeur) d'une association ; l'indice du tableau correspondant à un élément de clé donnée est déterminé par la fonction de hachage $h(x)$. Les valeurs de la fonction déterminent la taille du tableau (ou vice-versa). Si la fonction de hachage donne des valeurs entre 1 et n, il faudra un tableau de n éléments.

Pour les éléments de la table, nous définirons le type structuré *générique* suivant :

```
structure Élément<K, T>
    clé : K
    valeur : T
fin structure
```

Par facilité de notation, nous admettrons par la suite la notation (a, b) pour désigner une variable de ce type, avec a et b comme valeurs respectives des champs clé et valeur.

La table de hachage sera déclarée comme suit :

```
table : tableau [1 à n] de Élément<K, T>
```

Pour stocker le couple (clé, valeur), nous écrirons donc :

```
table[h(clé)] ← (clé, valeur)
```

et pour trouver la valeur associée à une clé :

```
val ← table[h(clé)].valeur
```

Exemple : Dans cet exemple, seules les clés apparaissent, nous faisons abstraction des valeurs. Prenons une table de capacité 10, pour y stocker des éléments déterminés par des clés à valeurs entières. La fonction de hachage est $h(x) = x$ MOD 10 + 1. Si les valeurs des clés à introduire dans la table sont 12, 17, 29 et 33 on aura la configuration suivante :

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
		12	33				17		29

Nous voyons aussi que l'occupation du tableau est irrégulière. Il faut dès lors pouvoir reconnaitre les cases vides des cases occupées (par exemple en y mettant une valeur aberrante de la clé). Que se passerait-il si on veut introduire la clé de valeur 42 ? Elle devrait occuper la même case que 12, ce qui n'est pas possible. Nous sommes dans le cas d'une « collision » pour lequel nous envisageons deux solutions.

5.8 Gestion des collisions

Il y a *collision* entre deux clés k_1 et k_2 lorsque $h(k_1) = h(k_2)$ Que faire dans ces cas là ? Nous décrivons deux techniques courantes : l'« adressage ouvert » et le « chainage ».

5.8.1 L'adressage ouvert

L'idée est la suivante : lorsqu'on veut utiliser un emplacement de la table et que celui-ci est occupé, on va voir ailleurs. Dans la technique la plus simple, on va simplement continuer le parcours du tableau, à partir de la position de départ, à la recherche d'un « trou ». Pour la recherche, on procède de même ; la fonction de hachage nous indique où commencer à chercher et on poursuit jusqu'à trouver l'élément ou un « trou ».

Poursuivons l'exemple ci-dessus : la clé 42 devrait occuper la case d'indice 3. Comme celle-ci est occupée, on parcourt le tableau jusqu'à la prochaine case libre, celle d'indice 5 et on y place l'élément :

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
		12	33	42			17		29

Ce parcours de recherche est « circulaire » : arrivé à la dernière case, on revient au début. Ainsi, la clé 39 qui ne peut pas être mise à sa place « normale » (occupée par 29) sera placée en première position :

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
39		12	33	42			17		29

L'exemple peut laisser perplexe vu la petite taille du tableau ; si le tableau a une taille suffisante, on peut imaginer que la clé cherchée ne sera jamais très loin de la position donnée par $h(x)$ de telle façon que la longueur du parcours requis peut être considérée comme négligeable.

Noter que la suppression est plus délicate à mettre en œuvre : il ne suffit pas de supprimer simplement la clé dans la case donnée par la fonction de hachage, car s'il y a eu une collision, on ne retrouverait plus l'autre valeur de clé. Il faut donc reboucher le trou avec la dernière clé donnant la même valeur de hachage.

Ainsi, dans l'exemple, si on supprime 12, il faut déménager 42 dans la case d'indice 3 :

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
39		42	33				17		29

Notons que la table a aussi une capacité maximale, on ne pourra pas y mettre plus de clés que sa taille. Cette technique est donc assez délicate mais intéressante lorsque les allocations dynamiques sont impossibles ou coûteuses. Si elles sont possibles, on utilisera plutôt la seconde technique :

5.8.2 Le chainage

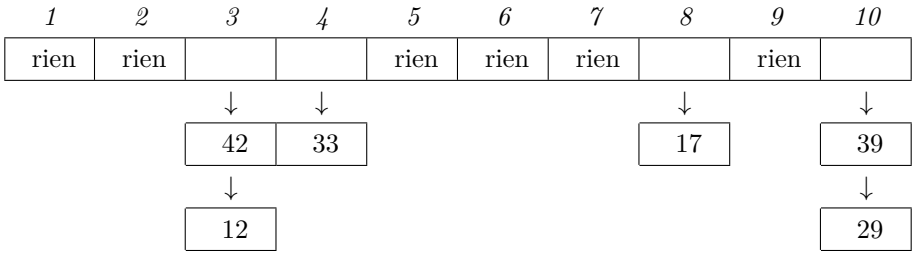
Avec cette technique, une entrée de la table ne contient pas un couple (clé, valeur) mais une liste chainée de couples (clé, valeur) : précisément tous les couples dont les clés ont la même valeur de hachage.

```
table : tableau [1 à n] de ListeChainée<Élément<K, T>>
```

Pour rechercher un élément, il suffit à présent de parcourir la liste chainée dont l'accès se trouve dans la case d'indice $h(x)$. L'avantage est l'aisance de l'implémentation, moins délicate que l'adressage ouvert, et de plus, il n'y a pas de limitation (théorique) au nombre

de clés. Toutefois, il faut veiller à prendre un tableau assez grand pour que la taille des listes reste réduite, le but étant de minimiser les parcours.

Avec la technique de chainage, l'introduction des clés 12, 17, 29, 33, 42 et 39 dans la table donnerait la configuration suivante :



(Il est évident que les ajouts se feront toujours en tête de liste).

5.9 Efficacité de la classe Map

Comparons l'efficacité de plusieurs implémentations de l'interface Map : le tableau (ordonné), la liste (non ordonnée), l'arbre binaire (ordonné) ou la table de hachage.

	recherche	**ajout**	**suppression**
tableau	$O(log_2 n)$	$O(n)$	$O(n)$
liste chainée	$O(n)$	$O(1)$	$O(n)$
arbre	$O(log_2 n)$	$O(log_2 n)$	$O(log_2 n)$
table de hachage	$O(1)$	$O(1)$	$O(1)$

Remarques

- ▷ La table de hachage est particulièrement efficace mais ses performances se dégradent si la table est fort remplie à cause du nombre croissant de collisions, inconvénient que n'a pas l'arbre.
- ▷ L'arbre sera étudié plus loin dans le cours ; signalons déjà que c'est une structure qui a aussi l'avantage de permettre de donner facilement les éléments dans l'ordre des clés ce que ne permet pas la table de hachage.

5.10 Exercices

1 Hachage de clés entières

On stocke des valeurs associées à des clés entières dans une table de hachage de capacité 15. La fonction de hachage est $h(x) = x$ MOD 15+1. Pour les deux techniques d'implémentations présentées au cours (adressage ouvert et chainage), représenter par un schéma l'état de la table après insertion dans l'ordre des clés suivantes :

17, 22, 36, 55, 21, 152, 64, 63, 10, 32

Donner ensuite l'état de la table après suppression dans l'ordre des clés 55, 36 et 152.

2 Hachage de chaines

On stocke des chaines de caractères dans une table de hachage par la technique de l'adressage ouvert. La table permet de stocker 13 éléments et la fonction de hachage est donnée par la formule $h(x) = (position(car(x, 1))+1)$ DIV 2 où position(car) est la position dans l'alphabet du caractère car, et car(maChaine, pos) donne le caractère en position pos (à partir de 1) dans la chaine maChaine.

Représenter le contenu de la table après exécution dans l'ordre des actions suivantes :

- ▷ on ajoute « André », « Edouard », « Francis », « Fabien », « Gilles » et « Geoffrey »

- ▷ on retire « Francis » et « Gilles »
- ▷ on rajoute « Arnaud » et « Francis ».

3 Les membres

Un fichier membres contient la liste des membres d'une association, classés par ordre alphabétique sur leur nom. Chaque enregistrement est un élément d'une structure Membre composée des champs nom, prénom, rue, numéro, ville, codePostal, dateNais.

Écrire un algorithme qui donne le nom de la ville où habite le maximum de membres de l'association.

4 Map avec chainage

Détailler l'implémentation de la classe ListeMap à l'aide d'une table de hachage fonctionnant avec la technique du chainage. On considère que la fonction de hachage $h(x)$ donne des valeurs toujours comprises entre 1 et 1000.

5 Gestion de stock

Soient la classe Article et la structure Achat définies comme suit :

```
classe Article
 privé:
    code : entier
    libellé : chaine
    prix : réel
    quantité : entier
 public:
    constructeur Article(unCode : entier, unLibellé : chaine, unPrix : entier, uneQuantité : entier)
    méthode getLibellé() → chaine
    méthode getCode() → entier
    méthode getPrix() → réel
    méthode getQuantité() → entier
    méthode addQuantité(q : entier)
    méthode setPrix(p : entier)
fin classe
```

```
structure Achat
    codeArticle : entier
    quantité : entier
fin structure
```

Définissez la classe **Stock** pour qu'elle n'offre que les méthodes suivantes que vous implémenterez. Veillez à choisir une implémentation des données assurant de bonnes performances.

```
méthode màjStock(articles : Liste<Article>)
// articles reprend les articles de réapprovisionnement du stock
// (nouveaux articles et/ou articles existants)
méthode getQuantité(code : entier) → entier
méthode getLibellé(code : entier) → chaine
méthode getPrix(code : entier) → réel
méthode getCodesArticles() → Liste<entier>
méthode évaluer(panier : Liste<Achat>) → réel          // retourne le prix total du panier
```

6 Cactus hôtel

La chaine CactusHotel possède 3 implantations en Europe (Paris, Londres et Amsterdam). Elle souhaite fidéliser ses meilleurs clients de l'année 2015 en leur offrant des bons de réduction à utiliser pendant l'année 2016.

Les clients concernés sont classés dans les catégories « silver » et « gold » :

- ▷ les clients « silver » sont ceux qui ont séjourné dans 2 hôtels parmi les 3 de la chaine ; ils recevront chacun un bon de réduction de 25%.
- ▷ les clients « gold » sont ceux qui ont séjourné dans les 3 hôtels de la chaine ; ceux-ci recevront un bon de réduction de 50%.

En vue de préparer l'envoi du courrier aux clients concernés par ces ristournes, on demande d'écrire un algorithme qui renverra une liste d'éléments de type **Data**, qui est une structure contenant les champs :

Data

- ▷ nom : chaine
- ▷ adresse : chaine
- ▷ taux : entier (contenant le taux de réduction, soit 25 ou 50)

Pour ce faire, vous avez à votre disposition 3 listes : listeP, listeL et listeA qui contiennent respectivement les données relatives aux clients ayant séjourné en 2015 dans les hôtels à Paris, Londres et Amsterdam. Chaque élément de ces listes est de type Personne.

N.B. : on peut considérer que les listes ne contiennent pas de doublons.

Personne

- ▷ nom : chaine
- ▷ clientID : chaine (code d'identification du client)
- ▷ adresse : chaine

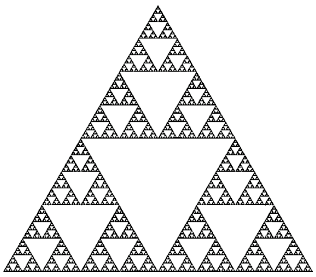
Remarques : les listes de ce problèmes sont des objets de la classe Liste telle que vue en première année.

Chapitre 6

La récursivité

« Pour comprendre le principe de récursivité,

il faut d'abord comprendre le principe de récursivité »

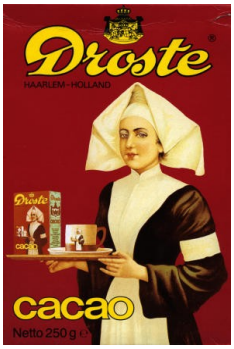


6.1 Introduction

Une procédure est dite **récursive** lorsque dans sa description, une des étapes de cette procédure est la procédure elle-même. Des termes synonymes de la récursivité sont la **récurrence** ou l'**auto-référence**.

La récursivité est une démarche qui consiste à faire référence à ce qui fait l'objet de la démarche, ainsi c'est le fait de décrire un processus dépendant de données en faisant appel à ce même processus sur d'autres données plus « simples », de montrer une image contenant des images similaires, de définir un concept en invoquant le même concept. (Wikipédia, août 2013)

Les illustrations ci-dessous sont des exemples célèbres de dessins récursifs :



La *vache qui rit* qui orne la fameuse boite de fromage porte des boucles d'oreilles en forme

de boites de fromage identiques au dessin entier. Il en est de même pour le dessin de la boite de cacao *Droste* : sur le plateau porté par la dame apparait en plus petit la même boite de cacao. Dans le troisième dessin, chaque carré contient un carré plus petit inscrit dans le carré précédent de telle sorte que l'image formée à partir du 3^{e} carré est identique au carré de départ, mais réduite de 50%. Chacun de ces dessins s'*auto-contient* donc à une échelle plus petite. Il en est de même pour le dessin fractal en haut de ce chapitre, il s'agit du *triangle de Sierpinski* formé dans chaque coin de triangles plus petits mais pareils au triangle entier.

6.2 Définitions récursives

Les dessins ci-dessus offrent des exemples de *récursivité infinie*. En mathématique et en informatique, la récursivité définit plutôt un processus dont la description se décompose en un (ou plusieurs) cas de base, et un ensemble de règles réduisant les autres cas aux cas de base par l'exécution d'un nombre fini d'étapes.

Certains concepts peuvent se définir récursivement, à condition que la définition contienne une clause d'évaluation immédiate sans appel récursif.

▷ Prenons par exemple la formule *récursive* de la factorielle :

$$n! = n * (n-1)!$$

Cette formule est incomplète si on ne définit pas le cas initial

$$0! = 1$$

À partir du cas de base et de la formule récursive, on peut calculer la factorielle de n'importe quel nombre naturel.

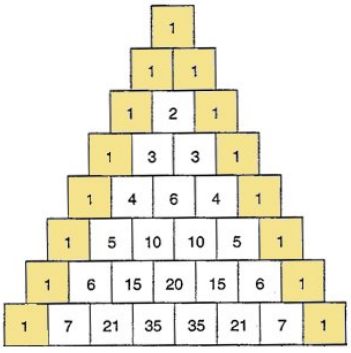
▷ Parmi les autres exemples rencontrés au cours de mathématique, citons la formule récursive des coefficients binomiaux :

$$C_n^p = C_{n-1}^{p-1} + C_{n-1}^p \quad \text{si } 0 < p < n$$

avec les cas de base :

$$C_n^0 = 1 \text{ et } C_n^n = 1$$

Cette définition permet de reconstruire facilement le triangle de Pascal :



Noter que le calcul récursif est plus commode d'utilisation que la formule « directe » faisant intervenir les factorielles :

$$C_n^p = \frac{n!}{(n-p)!p!}$$

▷ Un troisième exemple est la formule récursive pour calculer les nombres de Fibonacci :

$$F_n = F_{n-1} + F_{n-2}$$

avec les cas initiaux

$$F_0 = 1 \text{ et } F_1 = 1$$

Cette formule récursive est bien plus maniable que la formule donnant « directement » le *n-ième* nombre de Fibonacci :

$$F_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$$

▷ On peut aussi définir de façon récursive certaines structures courantes en informatique, comme le montre par exemple cette définition récursive de la liste chainée :

une liste chainée est un ensemble soit vide, soit composé d'un élément auquel est attaché une liste chainée.

Cette définition est intelligible par le fait qu'on sous-entend que la liste chainée à laquelle la définition fait référence n'est plus identique à la première, mais plus petite d'un élément, et éventuellement qu'elle peut se terminer (ensemble vide).

On peut, de façon semblable, définir une chaine de caractères :

une chaine de caractères est un ensemble soit vide, soit composé d'un caractère concaténé à une chaine de caractères.

6.3 Algorithme récursif

En informatique, un **algorithme récursif** est un algorithme qui, lors de son exécution, fait appel une ou plusieurs fois à lui-même. Ainsi, dans un algorithme récursif, nous pourrons écrire parmi les instructions l'appel à cet algorithme lui-même :

```
algorithme récursif(...)
    // instructions
    récursif(...)
    // instructions
fin algorithme
```

Si cette situation peut surprendre au premier abord, elle n'est pourtant pas très différente de la situation suivante, appelée *récursivité croisée* :

```
algorithme premier(...)
    // instructions
    second(...)
    // instructions
fin algorithme
```

```
algorithme second(...)
    // instructions
    premier(...)
    // instructions
fin algorithme
```

Chaque algorithme fait appel à l'autre, et si on recopie le code de l'algorithme second à la place de son appel dans premier, on obtient la même situation que dans l'algorithme récursif qui précède. Néanmoins, en absence de paramètres, ces deux situations génèrent un processus infini, et pire, une saturation de la mémoire ! En effet, pour pouvoir fonctionner, chaque appel de l'algorithme génère un nouvel ensemble des variables utilisées dans l'algorithme. La situation est donc très différente d'une boucle infinie du type *tant que* ou *faire jusqu'à ce que*.

Pour ne pas tomber dans ce processus sans fond, un algorithme récursif bien écrit doit forcément posséder un (ou plusieurs) paramètre(s), et contenir une structure alternative qui conduit à une clause d'évaluation immédiate de l'appel récursif pour une (ou plusieurs) valeur(s) initiale(s) du(des) paramètre(s).

La forme générale est la suivante :

```
algorithme récursif(paramètre)
    si paramètre = valeur initiale alors
        // il peut y avoir plusieurs cas initiaux
        // instructions sans appel récursif.
    sinon
        // instructions menant à des valeurs différentes du paramètre,
        // et après un nombre fini d'appels récursifs aux valeurs initiales.
    fin si
fin algorithme
```

6.4 Exemples classiques

Dans ce paragraphe, nous présentons quelques grands classiques de la programmation récursive.

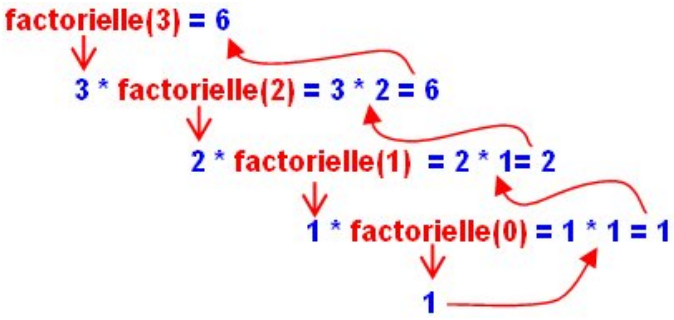
6.4.1 La factorielle

L'algorithme récursif calculant la factorielle d'un entier positif est issu directement de la définition récursive rappelée ci-dessus :

```
algorithme factorielle(n↓ : entier) → entier
    si n = 0 alors
        retourner 1
    sinon
        retourner n * factorielle(n - 1)
    fin si
fin algorithme
```

Comment cela fonctionne-t-il ?

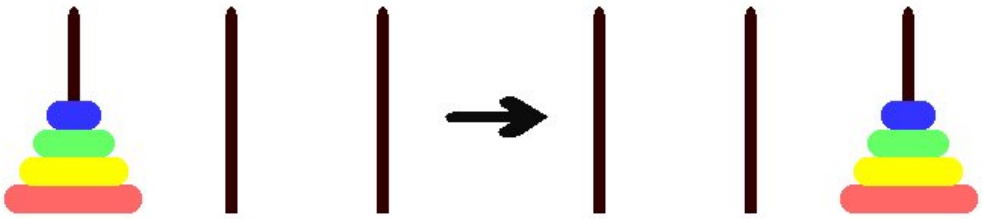
Par exemple, si l'algorithme est appelé avec la valeur 3 du paramètre, il doit calculer 3 fois la factorielle de 2. Le calcul est donc mis en attente jusqu'à ce que la factorielle de 2 soit connue ; et il en est de même pour le calcul de factorielle de 2, qui appelle la factorielle de 1 qui à son tour appelle la factorielle de 0. À ce moment, la valeur 1 est retournée, et les renvois se font en cascade jusqu'au premier appel qui peut enfin retourner la valeur 6.



6.4.2 Les tours de Hanoï

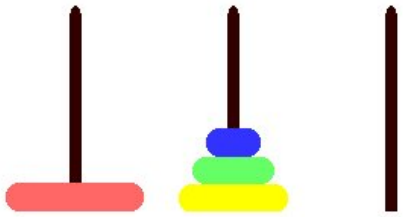
Il s'agit d'un jeu de réflexion consistant à déplacer une tour formée de n disques de diamètres différents. La tour qui se trouve sur un socle de départ doit être reconstruite sur un socle d'arrivée, en utilisant un socle intermédiaire, et en respectant les règles suivantes :

- ▷ on ne peut déplacer qu'un disque à la fois ;
- ▷ on ne peut jamais placer un disque sur un autre disque de diamètre plus petit.

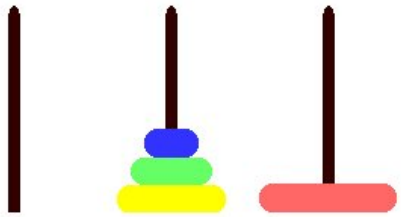


On voudrait écrire un algorithme qui affiche la liste des déplacements des disques qui conduisent à la reconstruction de la tour. La récursivité permet de décrire très simplement la marche à suivre : supposons qu'on sache comment reconstituer une tour de $n-1$ disques, la reconstitution d'une tour de n disques est alors un jeu d'enfant et s'exécute en 3 étapes :

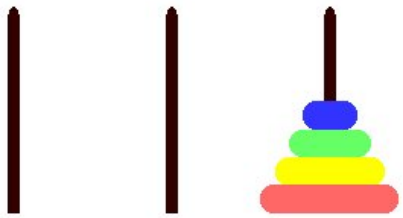
- ▷ reconstituer la tour formée par les $n-1$ disques les plus petits du socle de départ vers le socle intermédiaire (en utilisant le socle d'arrivée comme socle intermédiaire)



- ▷ déplacer le disque le plus grand du socle de départ vers le socle d'arrivée



- ▷ reconstituer la tour formée par les $n-1$ disques les plus petits du socle intermédiaire vers le socle d'arrivée (en utilisant cette fois-ci le socle de départ comme socle intermédiaire)



Cette démarche conduit au code suivant (les socles sont déclarés comme caractères, on peut leur donner des noms 'A', 'B' et 'C' par exemple) :

```
algorithme Hanoï(n↓ : entier, départ↓, arrivée↓, intermédiaire↓ : caractères)
    si n > 0 alors
        Hanoï(n − 1, départ, intermédiaire, arrivée)
        afficher « déplacer le disque de taille », n « du socle », départ, « vers le socle », arrivée
        Hanoï(n − 1, intermédiaire, arrivée, départ)
    fin si
fin algorithme
```

6.4.3 Le tri Quicksort

Il s'agit – comme son nom l'indique – d'une technique de *tri rapide* des éléments d'un tableau. L'idée est relativement simple : on choisit d'abord un élément pivot (aléatoirement ou de façon déterminée ; en pratique, c'est l'élément du milieu). On réarrange ensuite – par un nombre minimum de déplacements – le tableau de façon à le partager en deux sous-tableaux : à gauche les éléments inférieurs au pivot, et à droite les éléments supérieurs au pivot. Il suffit ensuite de réappliquer l'algorithme de façon récursive à chacun des sous-tableaux obtenus. Lorsque la taille d'un sous-tableau est inférieure à 2, il n'y a alors plus rien à trier.

Montrons le fonctionnement de l'algorithme sur un exemple. Pour le tableau suivant, le pivot sera le $5^{\text{ème}}$ élément, de valeur 6

4	3	8	9	6	3	7	1	5	3

Le réarrangement se fait en échangeant le premier élément supérieur ou égal au pivot (en partant de la gauche) avec le premier élément inférieur ou égal au pivot (en partant de la droite) :

4	3	8	9	6	3	7	1	5	3

ce qui donne :

4	3	3	9	6	3	7	1	5	8

On recommence ceci jusqu'à ce que les indices de recherche *gauche* et *droite* se rencontrent ; on obtient ainsi les étapes suivantes :

4	3	3	5	6	3	7	1	9	8

4	3	3	5	1	3	7	6	9	8

4	3	3	5	1	3	6	7	9	8

On recommence alors cette procédure dans chacun des 2 sous-tableaux obtenus, dans l'exemple celui formé par les 7 premiers éléments, et celui formé par les 3 derniers.

L'algorithme correspondant pour le tri d'un sous-tableau délimité par les indices bInf et bSup est le suivant :

```
algorithme quicksortRécursif(tab↓↑ : tableau[1 à n] d'entiers, bInf↓, bSup↓ : entiers)
    gauche, droite, pivot : entiers
    si bInf < bSup alors
        pivot ← tab[(bInf + bSup) DIV 2]
        gauche ← bInf-1
        droite ← bSup+1
        faire
            faire
                gauche ← gauche + 1
            tant que tab[gauche] < pivot
            faire
                droite ← droite – 1
            tant que tab[droite] > pivot
            si gauche < droite alors
                swap(tab[gauche], tab[droite])                    // échange des deux éléments
            fin si
        tant que gauche < droite
        quicksortRécursif(tab, bInf, gauche – 1)
        quicksortRécursif(tab, droite  + 1, bSup)
    fin si
fin algorithme
```

Cet algorithme est appelé la première fois par l'algorithme « façade » :

```
algorithme quicksort(tab↓↑ : tableau[1 à n] de T)
    quicksortRecursif(tab, 1, n)
fin algorithme
```

6.5 Exercices

Dans les exercices suivants, il faudra parfois écrire un « algorithme façade », c'est-à-dire un algorithme initial non récursif dans lequel les paramètres sont initialisés et qui appelle la première fois l'algorithme récursif. Il est intéressant aussi de comparer chaque problème avec sa solution itérative – lorsqu'elle existe – et de s'interroger sur la solution la plus efficace.

1 Les nombres de Fibonacci

Écrire l'algorithme récursif permettant de calculer le *n-ième* nombre de Fibonacci, issu de la définition récursive donnée dans ce chapitre. Combien de fois l'algorithme est-il exécuté pour le calcul de F_5 ? de F_6 ?

Adapter l'algorithme pour que le nombre total d'appels récursifs s'affiche à l'issue du calcul.

2 Les tours de Hanoï

Tracer l'algorithme des tours de Hanoï dans le cas $n = 3$, c'est-à-dire détailler l'affichage complet de cet algorithme. Combien d'affichages cet algorithme génère-t-il pour une valeur n quelconque ?

3 Taille d'une liste chainée

Écrire un algorithme récursif qui calcule la taille d'une liste chainée. Comparez les versions procédurale et orientée objet de cet algorithme.

4 Le tableau symétrique

Écrire l'algorithme qui vérifie si le contenu d'un tableau est symétrique (c'est-à-dire si les premier et dernier élément sont égaux, les second et avant-dernier, et ainsi de suite).

5 Division entière

Écrire un algorithme qui calcule la division entière de 2 entiers positifs de manière récursive ; les seuls opérateurs arithmétiques autorisés sont l'addition et la soustraction.

6 Recherche dichotomique

Écrire une version récursive de l'algorithme de recherche dichotomique dans un tableau ordonné d'éléments de type T. Cet algorithme reçoit en paramètre le tableau, la valeur recherchée, et retourne un booléen indiquant si la valeur se trouve dans le tableau. Dans l'affirmative, la position de la valeur recherchée est renvoyée dans un paramètre en sortie.

7 Le plus grand commun diviseur

L'algorithme d'Euclide permet de calculer rapidement le PGCD de 2 nombres. Il peut se définir de la façon suivante :

- ▷ PGCD(a, 0) = a
- ▷ PGCD(a, b) = PGCD(b, a MOD b) si b $\neq 0$

Écrire un algorithme récursif pour le calcul du PGCD. Cet algorithme générera une erreur si un des 2 nombres est négatif.

8 Calcul de puissance

Écrire un algorithme qui calcule la puissance entière d'un nombre réel en tenant compte du fait que :

- ▷ $x^n = (x^{nDIV2})^2$ si n est pair
- ▷ $x^n = (x^{(n-1)DIV2})^2 * x$ si n est impair

Vérifier que votre algorithme fonctionne si n est négatif ou nul. Combien de fois l'algorithme récursif est-il exécuté lors du calcul de x^{37} ?

9 Les coefficients binomiaux

Calculer les coefficients binomiaux à partir de la définition récursive :

- ▷ $C_n^0 = 1$
- ▷ $C_n^n = 1$
- ▷ $C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$ si $0 < p < n$

Veillez à vérifier les paramètres de départ (n et p ne peuvent pas être négatifs). Si $p > n$, alors le coefficient vaut 0.

10 Évaluation d'un nombre donné en chiffres romains

Les chiffres romains sont	Les nombres romains s'écrivent
	I = 1
M = 1000	II = 2
D = 500	III = 3
C = 100	IV = 4
L = 50	V = 5
X = 10	VI = 6
V = 5	VII = 7
I = 1	VIII = 8
	IX = 9
	X = 10

Tous les nombres finissant par 1, 2 ou 3 se terminent par I, II ou III mais c'est aussi le cas pour ceux finissant par 6, 7 ou 8.

Il en va du même principe pour les dizaines et les centaines. 20 s'écrit XX et 300 CCC.

Tous les nombres se finissant par 4 se terminent par IV. De même, 40 s'écrit XL ; et 44, XLIV.

Tous les nombres se finissant par 9 se terminent par IX. De même, 90 s'écrit XC ; et 99, XCIX.

Par exemples,

- ▷ 15 s'écrit XV ;
- ▷ 47 s'écrit XLVII ;
- ▷ 98 s'écrit XCVIII ;
- ▷ 14 s'écrit XIV ;
- ▷ 149 s'écrit CXLIX (100 + 40 + 9) ;
- ▷ 1490 s'écrit MCDXC ;
- ▷ 1900 s'écrit MCM ;
- ▷ 1990 s'écrit MCMXC ;
- ▷ 1999 s'écrit MCMXCIX.

Vous aurez remarqué que, si on regarde 2 chiffres romains consécutifs :

- ▷ si le premier chiffre romain a une valeur inférieure au deuxième, alors on le soustrait de la valeur de tout le reste ;
- ▷ sinon, on l'additionne à la valeur de tout le reste.

Par exemple :

MCMXCIX

On est sur le premier M.

Le chiffre qui le suit est C. Il est plus petit.

Donc, le résultat sera la valeur de M (1000) plus la valeur du reste (CMXCIX).

La valeur du reste est la suivante :

C est plus petit que M.

Donc, la valeur de CMXCIX est la valeur de MXCIX moins la valeur de C.

La valeur de MXCIX est la suivante :

M est plus grand que X, donc, on a la valeur de M plus la valeur de XCIX.

X est plus petit que C, donc,

la valeur de XCIX est la valeur de CIX moins la valeur de X.

C est plus grand que I, donc,

la valeur de CIX est la valeur de C plus la valeur de IX.

I est plus petit que X, donc,

la valeur de IX est la valeur de X moins la valeur de I,

c'est-à-dire 10 - 1 = 9

CIX = C + 9 = 109

XCIX = CIX - X = 109 - 10 = 99

MXCIX = M + XCIX = 1000 + 99 = 1099

CMXCIX = MXCIX - C = 1099 - 100 = 999

MCMXCIX = 1000 + 999 = 1999

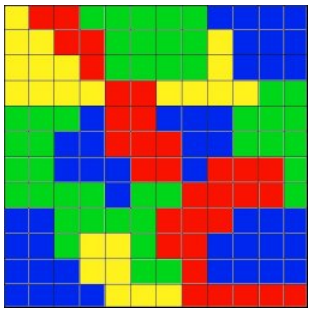
Donc, quand on a un nombre romain,

- ▷ si le premier chiffre de ce nombre a une valeur inférieure au deuxième, alors on le soustrait au reste, sinon, on l'additionne ;
- ▷ si le nombre romain a un seul chiffre, on prend la correspondance (M = 1000, D = 500, ...).

Écrivez l'algorithme de conversion d'un nombre romain de façon récursive. Il recevra en paramètre un tableau de caractères (le chiffre romain) et retournera la valeur correspondante.

11 Les taches de couleurs

Les couleurs des pavés d'un quadrillage sont stockées dans un tableau à 2 dimensions $n \times n$ (éléments de type Couleur). Les pavés adjacents (par les cotés, mais pas par les coins) de même couleur forment des « taches » de formes variées, comme le suggère le dessin suivant :



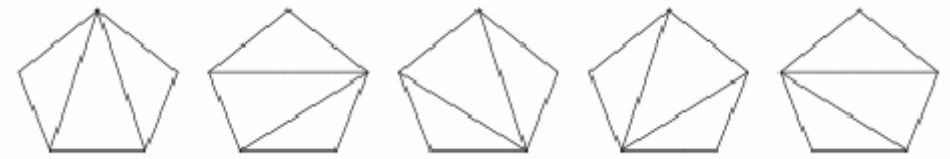
Résoudre les problèmes suivants :

1. vérifier si 2 pavés (i_1, j_1) et (i_2, j_2) sont dans une même tache ;
2. trouver l'aire de la tache contenant le pavé (i, j) ;
3. trouver le périmètre de la tache contenant le pavé (i, j) ;
4. trouver l'aire de la plus grande tache du quadrillage ;
5. changer la couleur de la tache contenant le pavé (i, j) avec une couleur entrée en paramètre.

Aide : à partir d'un pavé de départ, on visitera les pavés adjacents de même couleur, et on utilisera un tableau de booléens pour se rappeler si un pavé a déjà été traité.

12 Les nombres de Catalan

Le *n-ième nombre de Catalan*, noté C_n, correspond au nombre de différentes façons de partager un polygone de $n + 2$ côtés en triangles. Par exemple $C_3 = 5$, car il y a 5 façons de partager un pentagone en 3 triangles :



Trivialement, $C_0 = 1$, et on démontre que $C_n = \sum_{i=0}^{n-1} C_i * C_{n-1-i}$

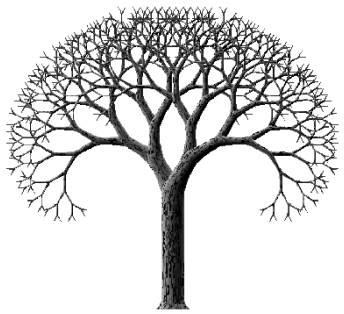
1. Calculer les nombres de Catalan C_i pour $i = 1, 2, 3, 4, 5$. Vérifiez que les valeurs obtenues correspondent bien à la définition géométrique.

2. Écrire l'algorithme récursif, issu directement de la formule mathématique ci-dessus, permettant de calculer C_n. Cette solution récursive vous semble-t-elle efficace ? Combien d'appels récursifs sont exécutés pour le calcul de C_5 ?

Chapitre 7

La structure d'arbre

La structure d'arbre trouve de nombreuses utilités en informatique : évaluation d'expressions algébriques, modélisation de l'inclusion d'ensembles, d'organisations hiérarchiques, de schémas en analyse, d'arbre généalogique, arborescence des dossiers dans un système d'exploitation, variables structurées C et en Cobol...

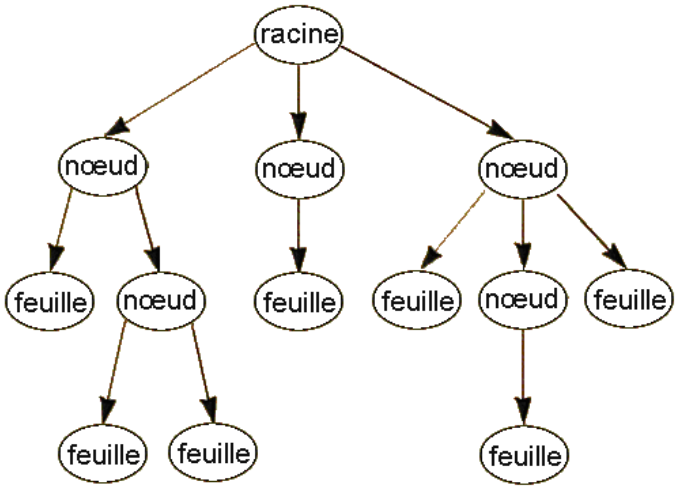


7.1 Définition et terminologie

7.1.1 Arbre

En informatique, un **arbre** est une structure de données constituée de **nœuds** assemblés par **niveaux**. C'est un graphe orienté connexe dont chaque arc relie un **nœud père** à un **nœud fils**. Chaque nœud peut posséder un nombre quelconque de fils ; un nœud sans fils est appelé **nœud terminal** ou **feuille**.

Chaque nœud possède un et un seul père, sauf la **racine**, située au sommet de l'arbre, qui ne possède pas de nœud père. Les nœuds autres que la racine ou les feuilles sont appelés **nœuds internes**. Deux nœuds ayant le même père sont des **frères**.

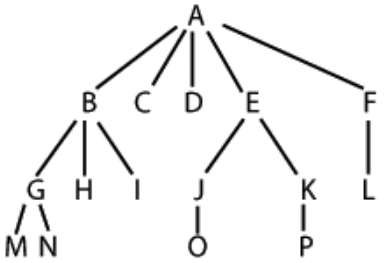


Attention : ne pas confondre l'arbre au sens informatique et celui étudié dans la théorie des graphes au cours de mathématique ; en théorie des graphes, un arbre est défini comme un graphe sans circuit et est dépourvu de toute structure hiérarchique. L'arbre au sens informatique est aussi appelé arbre enraciné.

On peut aussi définir l'arbre de manière récursive : c'est une structure soit vide, soit composée d'un nœud auquel est rattaché un ensemble fini d'arbres disjoints. Notez que cette définition récursive correspond bien à la réalité de ce qu'est un arbre dans la nature !

7.1.2 Niveau et hauteur

Le **niveau** (ou **profondeur**) d'un nœud est la distance qui le sépare de la racine. Il en découle que le niveau de la racine est 0, et le niveau de tout autre nœud est égal au niveau de son père augmenté de 1.



Exemple : dans le graphe ci-dessus, les niveaux sont les suivants : 0 pour A, 1 pour B, C, D, E, F, 2 pour G, H, I, J, K, L et 3 pour M, N, O, P.

Inversement, la **hauteur** d'un nœud est la distance entre ce nœud et sa descendance la plus éloignée : la hauteur d'une feuille est donc 0 et la hauteur de tout autre nœud est le maximum de la hauteur de ses fils augmenté de 1. La **hauteur d'un arbre** est la hauteur de sa racine, et correspond aussi à son niveau maximum. Elle n'est pas définie pour un arbre vide.

Pour le même graphe ci-dessus, les nœuds C, D, H, I, L, M, N, O et P ont une hauteur 0 (ce sont les feuilles de l'arbre). La hauteur de G, J, K et F est 1, celle de B et E est 2 et enfin A est à hauteur 3, qui correspond aussi à la hauteur de l'arbre.

Les nœuds d'un arbre contiennent de l'information, appelée parfois « étiquette ». L'étiquette peut être un simple entier ou une variable structurée plus complexe, l'instance d'un objet, un pointeur, etc. Le type de l'information détermine le type de l'arbre : on parlera d'un arbre d'entiers, de chaines et en général d'arbre de type T.

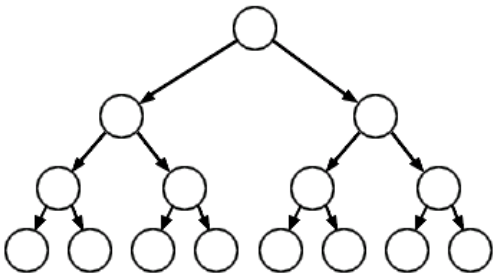
7.1.3 Arbres particuliers

Un arbre dont tout nœud possède au plus un descendant est appelé **arbre dégénéré**. Cet arbre est donc une liste chainée.

Un arbre dont tout nœud possède au plus un nombre déterminé n de descendants est appelé **arbre n-aire**.

Si dans un arbre n-aire, toutes les feuilles ont le même niveau, et si la racine et tous les nœuds internes ont le même nombre de fils, l'arbre est alors **complet**. En particulier, un arbre dégénéré est complet !

Le schéma ci-dessus montre un arbre « 2-aire » complet : chaque nœud non terminal a 2 fils, et toutes les feuilles sont au même niveau.



7.2 Parcours d'un arbre

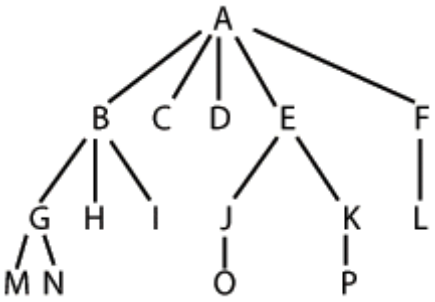
Il y a plusieurs façons de parcourir un arbre, le choix du parcours dépendant du type de traitement des informations de l'arbre.

7.2.1 Parcours en largeur

Le parcours en largeur, aussi nommé **parcours par niveau**, consiste à visiter les nœuds dans l'ordre de leur niveau : la racine, puis les nœuds de niveau 1, ceux de niveau 2 et ainsi de suite. L'ordre de parcours des nœuds pour un niveau donné est déterminé de manière récursive par l'ordre des nœuds parents.

Sur les schémas, l'ordre des nœuds correspond à leur disposition de gauche à droite.

Exemple : pour le graphe suivant, l'ordre de visite des nœuds par le parcours par niveau est A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P.

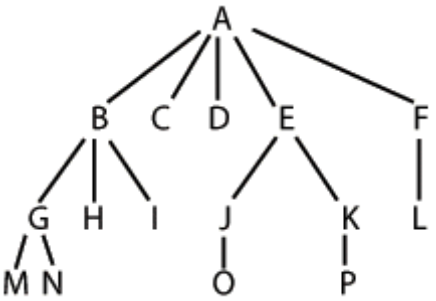


7.2.2 Parcours en profondeur

Il s'agit d'un parcours récursif sur les nœuds de l'arbre : partant de la racine, on visite tous ses fils, mais on ne passe à un nœud frère qu'après avoir visité tous les fils du nœud courant, et ceci récursivement.

Il y a deux possibilités de traitement des nœuds :

- ▷ si le nœud courant est traité avant ses fils, on parle de **parcours préfixé** ;
- ▷ Si le nœud courant est traité après ses fils, on parle de **parcours postfixé**.



Pour le graphe ci-dessus, le parcours en profondeur préfixé donne : A, B, G, M, N, H, I, C, D, E, J, O, K, P, F, L.

Le parcours en profondeur postfixé donne : M, N, G, H, I, B, C, D, O, J, P, K, E, L, F, A.

7.3 Implémentation orienté objet

Un arbre est déterminé par sa racine qui est un nœud de type T.

Chaque nœud contient de l'information (attribut valeur de type T) et doit permettre de passer à ses descendants (le 1er, le 2e, etc.) d'où également un attribut de type Liste qui permet d'accéder par position à chacun des fils (Liste d'éléments de type Nœud<T>).

L'implémentation de l'arbre est donc déterminée par la donnée des deux classes suivantes :

```
classe Nœud<T>
 privé:
    valeur : T
    ListeFils : Liste<Nœud<T>>
 public:
    constructeur Nœud<T>(val : T)
    // construit un nœud sans fils
    méthode getValeur() → T
    méthode setValeur(val : T)
    méthode getNbFils() → entier
    méthode getFils(i : entier) → Nœud<T>
    méthode setFils(i : entier, fils : Nœud<T>)
    méthode ajouterFils(fils : Nœud<T>)                    // ajout à la fin de la liste des fils
    méthode supprimerFils(i : entier)
fin classe
```

```
classe Arbre<T>
 privé:
    racine  : Nœud<T>
 public:
    constructeur Arbre<T>()
    // construit un arbre vide
    méthode getRacine() → Nœud<T>
    méthode setRacine(racine : Nœud<T>)
fin classe
```

Les contenus des méthodes de la classe Nœud<T> se laissent deviner aisément ; les cinq dernières méthodes consistent à appliquer respectivement sur l'attribut **ListeFils** les méthodes *taille*, *get*, *set*, *ajouter* et *supprimer* de la classe Liste.

Noter que – comme pour la liste chainée monodirectionnelle – on ne peut voyager dans un arbre que dans un seul sens, de la racine vers les feuilles, il n'y a pas de lien d'un nœud vers son nœud père. Le rajout de ce lien serait une possibilité d'implémentation (comme pour la liste bidirectionnelle) mais il n'est pas nécessaire pour la plupart des algorithmes de parcours qui fonctionnent essentiellement de manière récursive (pour les parcours en profondeur) ou à l'aide d'une file (parcours en largeur).

Nous donnons à titre d'exemple le détail de ces algorithmes de parcours.

7.4 Exemple : algorithmes de parcours

7.4.1 Parcours en largeur

Il n'existe pas de version récursive pour ce parcours.

La solution ci-dessous utilise une file où sont placés les fils des nœuds visités successifs.

```
algorithme parcoursLargeur(monArbre : Arbre<T>)
    maFile : File<Nœud<T>>
    nœudCourant : Nœud<T>
    i : entier
    maFile ← nouveau FileListe<Nœud<T>>()
    si monArbre.getRacine( ) ≠ rien alors
        maFile.enfiler(monArbre.getRacine( ))
    fin si
    tant que NON maFile.estVide( ) faire
        nœudCourant ← maFile.défiler( )
        // traitement du nœud courant
        pour i de 1 à nœudCourant.getNbFils( ) faire
            maFile.enfiler(nœudCourant.getFils(i))
        fin pour
    fin tant que
fin algorithme
```

7.4.2 Parcours en profondeur

Ce parcours est essentiellement récursif.

Il faut ici un algorithme façade, qui appelle l'algorithme récursif avec la racine de l'arbre comme première valeur du paramètre.

Noter l'emplacement du traitement du nœud courant selon le parcours préfixé ou postfixé.

```
algorithme parcoursProfondeur(monArbre : Arbre<T>)
    si monArbre.getRacine( ) ≠ rien alors
        parcoursProfondeurRécursif(monArbre.getRacine( ))
    fin si
fin algorithme
```

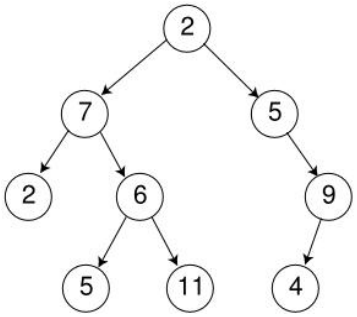
```
algorithme parcoursProfondeurRécursif(nœudCourant : Nœud<T>)
    i : entier
    // instructions de traitement du nœud courant si parcours préfixé
    pour i de 1 à nœudCourant.getNbFils( ) faire
        parcoursProfondeurRécursif(nœudCourant.getFils(i))
    fin pour
    // instructions de traitement du nœud courant si parcours postfixé
fin algorithme
```

7.5 Arbre binaire

7.5.1 L'arbre binaire

L'**arbre binaire** est un type d'arbre particulier : chaque nœud possède au plus deux descendants qui sont appelés **fils gauche** et **fils droit**.

Les deux sous-arbres attachés à un nœud donné sont appelés sous-arbre gauche et sous-arbre droit.



Attention, l'arbre binaire n'est pas un cas particulier d'arbre n-aire (avec la valeur 2 pour n), car les deux fils d'un nœud sont liés à une orientation (gauche ou droit).

Dans le cas où un nœud ne possède qu'un fils, il faut clairement indiquer dans le schéma d'un arbre binaire s'il s'agit du fils gauche ou droite, on ne peut donc jamais représenter un fils à la verticale du père.

Ainsi, dans l'exemple ci-dessus, le nœud 9 est le fils droit du nœud 5, et le nœud 4 est le fils gauche du nœud 9.

7.5.2 Implémentation

Elle se déduit de l'implémentation de l'arbre « quelconque ».

Pour le nœud d'arbre binaire, la liste des fils est remplacée par deux liens : un vers le fils gauche et un autre vers le fils droit.

```
classe NœudBinaire<T>
 privé:
    valeur : T
    gauche : NœudBinaire<T>
    droit : NœudBinaire<T>
 public:
    constructeur NœudBinaire<T>(val : T)                    // construit un nœud sans fils
    méthode getValeur() → T
    méthode setValeur(val : T)
    méthode getGauche() → NœudBinaire<T>
    méthode setGauche(fils : NœudBinaire<T>)
    méthode getDroit() → NœudBinaire<T>
    méthode setDroit(fils : NœudBinaire<T>)
fin classe
```

```
classe ArbreBinaire<T>
 privé:
    racine : NœudBinaire<T>
 public:
    constructeur ArbreBinaire<T>()                          // construit un arbre vide
    méthode getRacine() → NœudBinaire<T>
    méthode setRacine(r : NœudBinaire<T>)
fin classe
```

7.6 Parcours de l'arbre binaire

7.6.1 Parcours en largeur

La boucle « pour » du parcours de l'arbre « quelconque » est remplacée ici par la mise en file de chacun des fils du nœud courant.

Il faut toutefois prendre garde de ne pas mettre de lien vide dans la file.

```
algorithme parcoursLargeur(monArbre : ArbreBinaire<T>)
    maFile : File<NœudBinaire<T>>
    nœudCourant : NœudBinaire<T>
    i : entier
    maFile ← nouveau FileListe<NœudBinaire<T>>()
    si monArbre.getRacine( ) ≠ rien alors
        maFile.enfiler(monArbre.getRacine( ))
    fin si
    tant que NON maFile.estVide( ) faire
        nœudCourant ← maFile.défiler( )
        // traitement du nœud courant
        si nœudCourant.getGauche( ) ≠ rien alors
            maFile.enfiler(nœudCourant.getGauche( ))
        fin si
        si nœudCourant.getDroit( ) ≠ rien alors
            maFile.enfiler(nœudCourant.getDroit( ))
        fin si
    fin tant que
fin algorithme
```

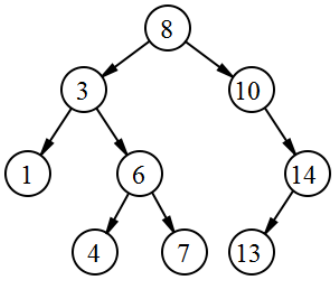
7.6.2 Parcours en profondeur

Pour les arbres binaires, aux parcours en profondeur préfixé et postfixé s'ajoute le **parcours infixé** : le traitement d'un nœud se fait entre les parcours de son sous-arbre gauche et de son sous-arbre droit.

```
algorithme parcoursProfondeur(monArbre : ArbreBinaire<T>)
    parcoursProfondeurRécursif(monArbre.getRacine( ))
fin algorithme
```

```
algorithme parcoursProfondeurRécursif(nœudCourant : NœudBinaire<T>)
    si nœudCourant ≠ rien alors
        // instructions de traitement du nœud courant si parcours préfixé (ou RGD)
        parcoursProfondeurRécursif(nœudCourant.getGauche( ))
        // instructions de traitement du nœud courant si parcours infixé (ou GRD)
        parcoursProfondeurRécursif(nœudCourant.getDroit( ))
        // instructions de traitement du nœud courant si parcours postfixé (ou GDR)
    fin si
fin algorithme
```

Par exemple, pour l'arbre binaire représenté ci-dessous :



les ordres de traitements sont les suivants :

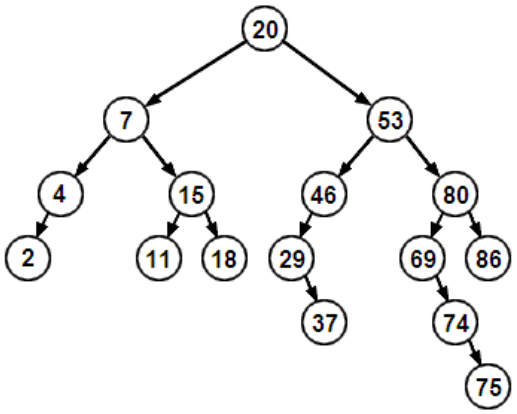
- ▷ *parcours en largeur : 8, 3, 10, 1, 6, 14, 4, 7, 13*
- ▷ *parcours en profondeur préfixé : 8, 3, 1, 6, 4, 7, 10, 14, 13*
- ▷ *parcours en profondeur infixé : 1, 3, 4, 6, 7, 8, 10, 13, 14*
- ▷ *parcours en profondeur postfixé : 1, 4, 7, 6, 3, 13, 14, 10, 8*

7.7 Arbre binaire ordonné

7.7.1 Définition

Un arbre binaire est ordonné si pour tout nœud, toutes les valeurs de son sous-arbre gauche sont inférieures ou égales à la valeur du nœud, et toutes les valeurs de son sous-arbre droit sont strictement supérieures à la valeur du nœud.

Exemple :



Cette structure se prête assez bien aux opérations de recherche, d'ajout, de suppression car elle permet de localiser rapidement le nœud où doit être exécuté l'opération, tout en offrant une souplesse dans ses modifications (comme pour une liste chainée).

La recherche s'apparente à la recherche dichotomique dans un tableau, et si l'arbre est *équilibré* (pour une définition précise de ce terme, voir énoncé de l'ex. 9), toutes les opérations citées ont une complexité en $O(log_2 n)$.

Pour obtenir la liste des valeurs d'un arbre binaire ordonné dans l'ordre, il suffit de parcourir ses nœuds par le parcours en profondeur infixé.

7.8 Exercices

7.8.1 arbre quelconque ou binaire

Les exercices 1 à 8 peuvent être résolus indifféremment dans le cas d'un arbre quelconque ou d'un arbre binaire.

1 Combien de nœuds ?
Écrire un algorithme qui compte le nombre total de nœuds contenus dans un arbre.

2 Et combien de feuilles ?
Écrire un algorithme qui compte le nombre de feuilles que possède un arbre.

3 Hauteur
Écrire un algorithme qui calcule la hauteur d'un arbre, c'est-à-dire la plus grande hauteur (ou la plus grande profondeur) de ses nœuds.

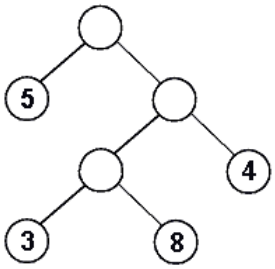
4 Problème de niveau
Écrire un algorithme qui retourne le nombre de nœuds d'un arbre situés à un niveau donné (valeur entière entrée en paramètre).

5 Tout le monde est là ?
Écrire un algorithme qui vérifie si un arbre est complet.

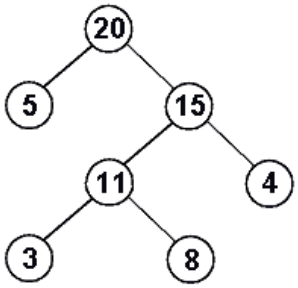
6 Arbre dégénéré
Écrire un algorithme qui indique si un arbre est dégénéré ou non.

7 Additions
Soit un arbre d'entiers dont seules les feuilles ont été affectées par des valeurs.

Compléter l'arbre de telle sorte que la valeur de chaque nœud soit égale à la somme des valeurs de ses fils.



devient après remplissage



8 Combien de valeurs ?
Écrire un algorithme qui indique le nombre de valeurs différentes que contient un arbre.

7.8.2 Arbres binaires

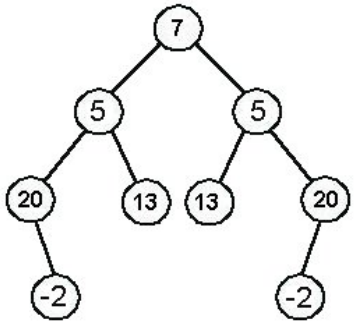
9 Arbre binaire équilibré
Dans un arbre binaire, on définit le **facteur d'équilibre** d'un nœud de la façon suivante : c'est la différence entre la hauteur de son sous-arbre gauche et celle de son sous-arbre droit. On dit que l'arbre est **équilibré** si, pour tout nœud de l'arbre, ce facteur d'équilibre est compris entre –1 et 1.

10 Arbre binaire pair
Un arbre binaire est **pair** si chacun de ses nœuds a soit 2 fils, soit aucun. Écrire un algorithme qui vérifie si un arbre binaire possède cette propriété.

N. B. : un arbre vide est pair.

11 OXO
Soit un arbre binaire de caractères. Vérifier s'il contient la configuration 'O'-'X'-'O' formée respectivement par un fils gauche, un père et un fils droit.

12 Arbre binaire symétrique
Écrire un algorithme qui vérifie si un arbre binaire est **symétrique**, c'est-à-dire si le sous-arbre gauche de la racine est identique en miroir à son sous-arbre droit.



7.8.3 Arbres binaires ordonnés

13 L'arbre binaire est-il ordonné ?
Écrire un algorithme qui vérifie si un arbre binaire est ordonné.

14 Le maximum

Écrire un algorithme qui retourne la valeur maximum d'un arbre binaire ordonné.

15 Recherche dichotomique

Écrire un algorithme qui recherche dans un arbre binaire la présence d'un nœud de valeur donnée, et en retourne l'accès. Si la valeur ne se trouve pas dans l'arbre, l'algorithme retourne *rien* dans ce cas.

16 Ajout de valeur

Écrire un algorithme qui ajoute un nœud dans un arbre binaire ordonné, dont la valeur est entrée en paramètre.

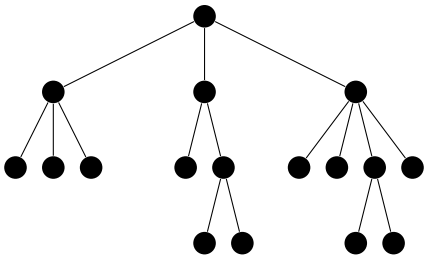
7.8.4 Arbres quelconques

17 Élagage

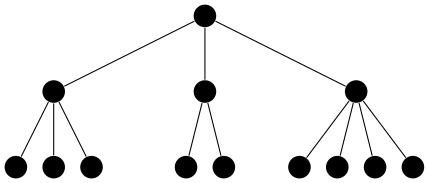
Écrire un algorithme qui supprime d'un arbre donné tous les nœuds dont le niveau est supérieur à un niveau maximal donné en paramètre.

Par exemple,

si l'arbre de départ est le suivant et le niveau donné est 2 :



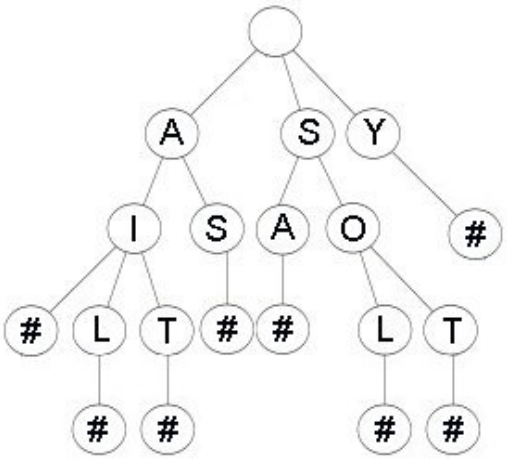
l'arbre deviendra :



18 Le dictionnaire

Utilisons un arbre pour stocker les mots d'un dictionnaire. Les lettres d'un mot sont représentées chacune par un nœud à un niveau différent et les mots ayant le même début partagent leurs nœuds. De plus, un caractère spécial indique la fin d'un mot. La racine ne contient rien. Les fils d'un nœud ne sont pas forcément dans l'ordre alphabétique.

Dans l'exemple ci-dessous, le dictionnaire contient les mots AI, AIL, AIT, AS, SA, SOL, SOT, Y.



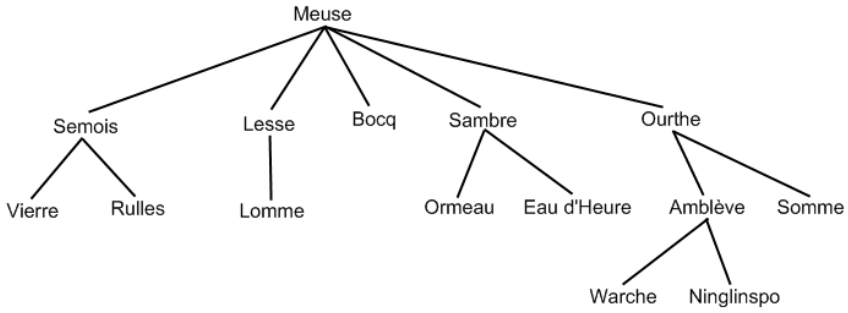
Écrire un algorithme :

1. qui affiche tous les mots du dictionnaire
2. qui ajoute un mot dans le dictionnaire

3. qui enlève un mot du dictionnaire

19 Au fil de l'eau

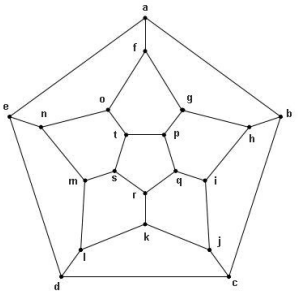
Les cours d'eau d'un bassin fluvial peuvent être schématisés avec un arbre : la racine contient le nom du fleuve, et les fils d'un nœud sont les affluents du fleuve ou des rivières faisant partie du bassin fluvial.



Écrire un algorithme qui reçoit le nom d'un cours d'eau en paramètre et affiche la suite des rivières qui le conduisent au fleuve principal. Par exemple, pour *« Somme »*, l'algorithme affiche *« Somme »*, *« Ourthe »* et *« Meuse »*.

Chapitre 8

Les graphes



8.1 Utilité

La théorie des graphes permet de répondre à différents problèmes se formulant en termes d'**objets** et de **liens** entre ceux-ci. De nombreux problèmes relatifs à l'étude des réseaux peuvent être résolus grâce à la théorie des graphes, que ce soit des problèmes de voyage (réseaux aériens, routiers, ferroviaires) ou des problèmes de communication entre personnes (réseaux téléphoniques) ou entre ordinateurs (réseaux informatiques, internet, ...)

Un problème classique de la théorie des graphes, et qui a prouvé toute son utilité depuis l'invention du GPS, est la détermination du chemin le plus rapide (ou le plus court ou encore le moins coûteux) entre deux localisations géographiques. Pour résoudre ce type de problème, il faut connaitre les différents relais ou points de passage (objets) et savoir lesquels sont reliés entre eux par des routes (les liens), ainsi que des informations supplémentaires sur ces liens (distance, type de route, etc.).

Un autre pilier de la théorie des graphes est l'étude de l'ordonnancement des tâches. Quelles sont les contraintes conditionnant l'avancement d'un projet ? Quelle tâche doit être finie avant que telle autre ne commence ? Comment tenir compte des tâches concurrentes ?

D'autres domaines peuvent également être abordés :

- ▷ analyse d'un programme, d'un algorithme
- ▷ élaboration de cartes géographiques
- ▷ représentation de relations entre individus (familiales, professionnelles, ...)
- ▷ représentation d'automates d'états finis, de tables de décisions, ...

Le but de cette partie du cours n'est pas d'aborder jusque dans les moindres détails la théorie des graphes. Celle-ci et ses développements sont d'une telle richesse qu'ils font l'objet de

nombreux ouvrages spécialisés. Nous nous limiterons à un aperçu synthétisé permettant de situer les problèmes et de servir de tremplin à une étude personnelle plus approfondie.

8.2 Terminologie

N.B. Les graphes vous sont déjà familiers, puisqu'ils ont déjà été étudiés au cours de mathématique de 1ère année. Nous rappelons brièvement ici les principales définitions, et renvoyons le lecteur au syllabus de mathématique de 1ère pour plus de précision.

8.2.1 Définition

Un **graphe** est un modèle mathématique représentant les liens existant entre différents objets de même type, appelés **nœuds** ou **sommets**. Les liens sont appelés **arcs** ou **arêtes**, selon que le graphe est orienté ou non. On peut donc voir un graphe comme l'association de l'ensemble S de ses nœuds et de l'ensemble A de ses arêtes :

$$G = (S, A)$$

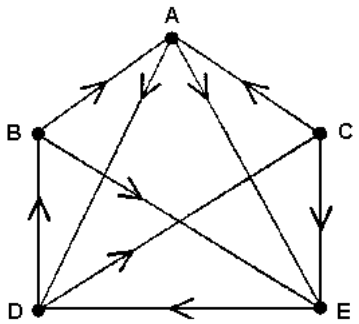
On représente schématiquement un graphe par un ensemble de points reliés entre eux par des traits ou des lignes représentant ces liens.

8.2.2 Types de graphes

8.2.2.1 Graphe orienté

Un graphe est **orienté** lorsque ses nœuds sont reliés par des **arcs**. Un arc est un **couple** (u, v) de nœuds, où u est l'**origine** du couple (ou encore le *prédécesseur*, le *départ* ou l'*extrémité initiale*) et v l'**extrémité** (ou encore le *successeur*, l'*arrivée* ou l'*extrémité terminale*). On dira de cet arc qu'il *part* du nœud u et qu'il *arrive* au nœud v. Dans la représentation schématique d'un graphe orienté, on représente habituellement les arcs par des flèches qui traduisent le sens de la relation entre deux nœuds.

Exemple : G = (S, A) avec S = {A, B, C, D, E} et A = {(A, D), (A, E), (B, A), (B, E), (C, A), (C, E), (D, B), (D, C), (E, D)}



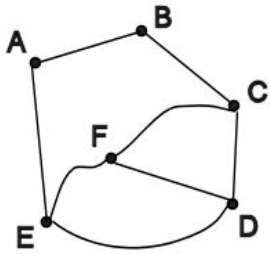
Un graphe orienté peut aussi contenir des boucles : une **boucle** est un arc dont l'origine et l'extrémité sont identiques, elle part et arrive au même nœud.

Un graphe orienté est dit **symétrique** si l'existence d'un arc allant de u vers v implique l'existence d'un arc « réciproque » de v vers u (avec $u \neq v$).

8.2.2.2 Graphe non orienté

Dans un graphe **non orienté**, les nœuds sont reliés par des **arêtes**. Une arête est une **paire** $\{u, v\}$ de nœuds, c'est-à-dire un ensemble non ordonné de deux nœuds (pour rappel, $\{u, v\} = \{v, u\}$)

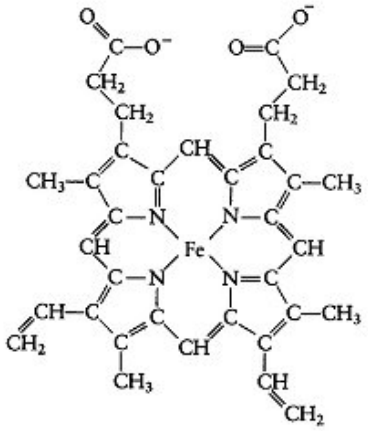
Exemple : G = (S, A) avec S = {A, B, C, D, E, F} et A = {{A, B}, {A, E}, {B, C}, {C, D}, {C, F}, {D, E}, {D, F}, {E, F}}



Remarquons qu'un graphe non orienté peut être représenté par un graphe orienté : il suffit de remplacer chaque arête reliant deux nœuds distincts par deux arcs orienté (ce qui donne un graphe symétrique). L'inverse n'est évidemment pas vrai !

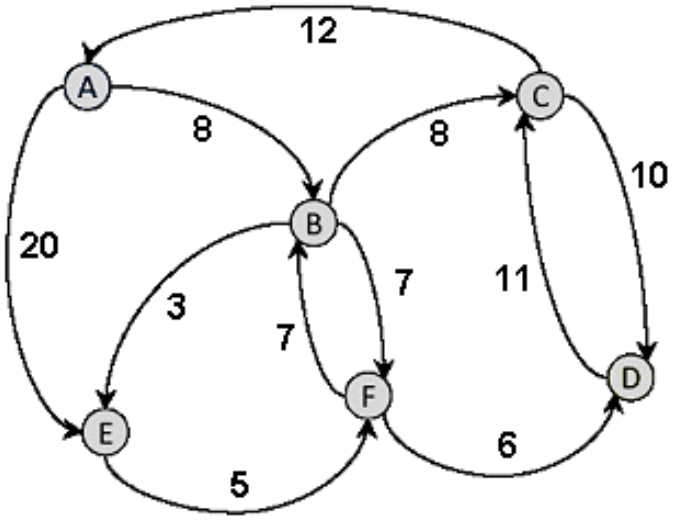
8.2.2.3 Multigraphe

Lorsqu'on admet que deux nœuds d'un graphe peuvent être rejoints par plusieurs arcs ou arêtes, on parle de **multigraphe**. Ce type de graphe est par exemple utilisé en chimie pour la représentation des molécules, un double lien représentant une liaison plus forte entre les atomes d'une molécule.



8.2.2.4 Graphe pondéré

Lorsqu'on associe une valeur numérique à chaque arc ou arête d'un graphe, on obtient un **graphe pondéré**. L'exemple le plus connu d'un tel graphe est la carte routière sur laquelle on indique les distances le long des routes joignant deux points de repère. D'autres informations possibles sont le temps, le coût, le poids, ...



8.2.2.5 Graphe étiqueté

Un graphe est étiqueté lorsque ce sont des informations de type texte qui sont attachées sur chaque arc ou arête (par exemple les numéros des routes sur une carte routière : « A19 » ou « E40 », ...)

8.2.3 Adjacence, incidence et degré

On dit que deux nœuds sont **adjacents** (ou **voisins**) s'il existe un arc (pour les graphes orientés) ou une arête (pour les graphes non orientés) reliant ces nœuds. L'adjacence s'applique aussi aux liens : deux arcs (ou deux arêtes) sont **adjacents** lorsqu'ils ont un nœud en commun. Par contre, la relation entre les nœuds et les liens s'exprime en terme d'**incidence** : on dit qu'un arc ou une arête est **incident** à un nœud.

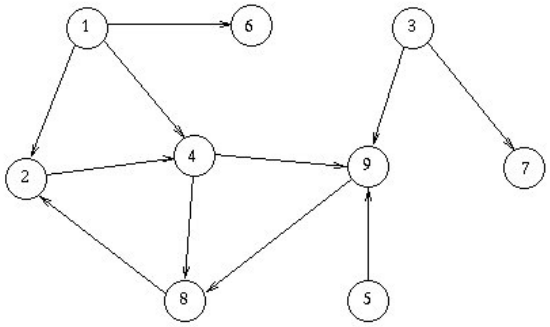
Un graphe non orienté dans lequel chaque nœud est adjacent à tous les autres nœuds est dit **complet**. Pour un graphe orienté, un graphe complet est tel qu'en chaque nœud partent des arcs vers tous les autres nœuds.

Le **degré** d'un nœud est le nombre d'arcs (ou d'arêtes) incidents à ce nœud. Dans le cas d'un graphe orienté, on définit encore :

- ▷ le **degré entrant** d'un nœud : c'est le nombre d'arcs qui arrivent en ce nœud
- ▷ le **degré sortant** d'un nœud : c'est le nombre d'arcs qui partent de ce nœud

Toujours dans le cas des graphes orientés, un **puits** est un nœud dont le degré sortant est nul : arrivé dans un puits, on ne peut plus en sortir ! Inversement, une **source** est un nœud dont le degré entrant est nul. On peut donc partir d'une source, mais on ne peut plus jamais y revenir ! Si les degrés entrant et sortant sont tous les deux nuls, on a alors un **nœud isolé**.

Exemple : dans le graphe ci-dessous, le nœud 8 est de degré 3 ; son degré entrant est 2 et son degré sortant est 1. Les nœuds 1, 3 et 5 sont des sources, et les nœuds 6 et 7 sont des puits.



Remarque : une boucle compte pour deux dans le calcul du degré d'un nœud, que ce soit dans un graphe orienté ou non orienté.

8.2.4 Chemin, cycle et connexité

8.2.4.1 Chemin

Dans un graphe orienté, un **chemin** est une suite d'arcs adjacents de la forme :

$$(u_0, u_1),\ (u_1, u_2),\ (u_2, u_3),\ ...,\ (u_{n-1}, u_n)$$

Dans cette suite, chaque arc part du nœud où arrive l'arc précédent. Dans un graphe non orienté, la définition est analogue : un chemin (aussi appelé **chaine** dans ce cas) est une suite d'arêtes adjacentes.

N.B. : on pourrait aussi décrire un chemin en donnant une suite de nœuds adjacents plutôt qu'une suite d'arcs ou d'arêtes.

Un chemin est **simple** si tous ses liens sont distincts.

N.B. : dans tout ce qui suit, nous ne considérerons uniquement que les chemins simples.

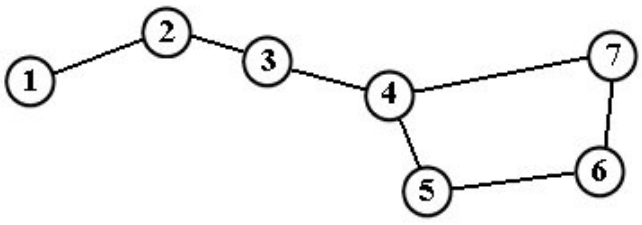
Un chemin est **élémentaire** si tous ses nœuds (sauf éventuellement le premier et le dernier) sont distincts.

La **longueur** d'un chemin est le nombre de liens qui composent ce chemin.

La **distance** entre deux nœuds est le minimum des longueurs parmi tous les chemins qui relient ces deux nœuds.

Enfin, le **diamètre** d'un graphe est la plus grande distance possible entre deux nœuds quelconques de ce graphe.

Exemple : La suite d'arêtes {2, 3}, {3, 4}, {4, 5}, {5, 6}, {6, 7} du graphe non orienté ci-dessous forme un chemin de longueur 5. La distance entre les nœuds 2 et 7 est 3, car la longueur du chemin {2, 3}, {3, 4}, {4, 7} est plus courte. Le diamètre de ce graphe est 5.



8.2.4.2 Circuits et cycles

Dans un graphe orienté, un **circuit** est un chemin fermé, c'est-à-dire une suite d'au minimum 2 arcs dont le nœud de départ est identique au nœud d'arrivée. Pour un graphe non orienté, on parle plutôt de **cycle** : c'est également un chemin fermé constitué d'au moins 3 arêtes adjacentes.

Un circuit (ou un cycle) est **simple** ou **élémentaire** selon que le chemin correspondant peut être qualifié de la même façon.

Dans l'exemple ci-dessus, la suite d'arêtes {4, 5}, {5, 6}, {6, 7}, {7, 4} forme un cycle élémentaire de longueur 4.

8.2.4.3 Connexité

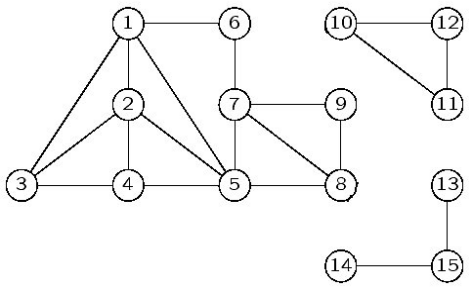
Un graphe **non orienté** est **connexe** (ou **connecté**) s'il existe toujours un chemin reliant deux nœuds quelconques de ce graphe. Autrement dit, un graphe connexe est « d'un seul tenant ».

Pour un graphe **orienté**, la définition de connexité est analogue, toutefois en considérant les arcs sans leur orientation.

Une **composante connexe** d'un graphe est un sous-ensemble de nœuds et d'arêtes de ce graphe formant lui-même un graphe connexe, et tel qu'il n'existe aucun chemin reliant un de ses nœuds à un nœud hors de ce sous-ensemble.

Un graphe connexe ne possédant aucun cycle ni boucle est un **arbre**, et un graphe dont toutes les composantes connexes sont des arbres est une **forêt**. Attention, ne pas confondre cette définition d'arbre avec les arbres binaires et n-aires étudiés précédemment. Dans le contexte d'un graphe, il n'y a pas de notion de racine, de niveau, de hauteur, de nœuds père et fils, etc. Par contre, les nœuds d'un arbre qui sont de degré 1 portent aussi le nom de **feuilles**.

Exemple : Dans le graphe suivant, il y a trois composantes connexes, respectivement formées par les sous-ensembles de nœuds {1, 2, 3, 4, 5, 6, 7, 8, 9}, {10, 11, 12} et {13, 14, 15}. La 3$^{\text{ème}}$ de ces composantes est un arbre.



8.3 Implémentation en mémoire

8.3.1 Représentation des nœuds

Dans les exemples précédents, les nœuds des graphes étaient nommés par des lettres ou des numéros, mais ils peuvent contenir de l'information diverse. Par ex. dans le graphe d'un réseau de métro, les nœuds seraient des chaines (noms des stations) ; en théorie des jeux, les nœuds peuvent contenir la configuration complète de l'état d'un jeu, les arêtes représentant alors les possibilités de passage d'une configuration à une autre. Le type des nœuds détermine le type du graphe : on parlera d'un graphe d'entiers, de chaines,... et de façon générique, d'un graphe de type T.

Le contenu des nœuds peut être stocké dans un tableau ou une liste, selon le contexte. Un tableau convient bien pour implémenter un problème de graphe « statique » (où le nombre de nœuds reste fixe) et la liste pour la modélisation d'un graphe qui subirait des transformations (ajout ou suppression de nœuds). Dans tous les cas, on considère qu'il est toujours possible de numéroter les nœuds de 1 à n.

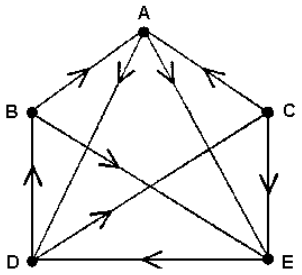
8.3.2 Représentation des arcs/arêtes

8.3.2.1 Représentation matricielle

La représentation matricielle est la façon la plus évidente pour représenter les arêtes d'un graphe. Si seule l'existence des liens doit être représentée, on utilise une matrice de booléens appelée **matrice d'adjacence** (ou **matrice de contigüité**). Chaque ligne et chaque colonne de cette matrice correspond à un nœud du graphe, conformément à la numérotation des nœuds choisie.

Pour un graphe orienté, les lignes correspondent à l'origine des arcs et les colonnes à leur extrémité. Ainsi, l'élément d'indices (i, j) sera **vrai** si un arc part de i et arrive en j.

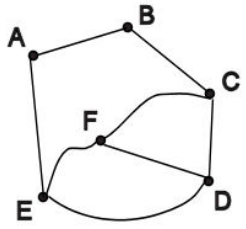
Exemple : Le graphe orienté ci-dessous à gauche peut être représenté par la matrice d'adjacence à droite.



	A	B	C	D	E
A	F	F	F	V	V
B	V	F	F	F	V
C	V	F	F	F	V
D	F	V	V	F	F
E	F	F	F	V	F

Dans le cas d'un graphe non orienté, l'élément d'indices (i, j) est mis à **vrai** si une arête rejoint les nœuds i et j. Comme cette relation est symétrique, la matrice d'adjacence qui en résulte est une **matrice symétrique**, c'est-à-dire telle que $m_{ij} = m_{ji} \forall i, j$. Par économie, on peut aussi se limiter dans ce cas à une **matrice triangulaire**, où seule une moitié de la matrice est utilisée (par exemple les éléments m_{ij} tels que $j \geq i$ s'il y a des boucles et $j > i$ sinon).

Exemple : Le graphe non orienté de la figure ci-dessous peut être représenté par une matrice symétrique (à gauche) ou triangulaire supérieure (à droite).



	A	B	C	D	E	F
A	F	V	F	F	V	F

	A	B	C	D	E	F
A	F	V	F	F	V	F

B	V	F	V	F	F	F
C	F	V	F	V	F	V
D	F	F	V	F	V	V
E	V	F	F	V	F	V
F	F	F	V	V	V	F

B	F	F	V	F	F	F
C	F	F	F	V	F	V
D	F	F	F	F	V	V
E	F	F	F	F	F	V
F	F	F	F	F	F	F

La représentation matricielle peut s'étendre aux cas des graphes pondérés (en utilisant une matrice d'entiers contenant les poids des arêtes) ou d'un graphe étiqueté (matrice de chaines). L'absence d'arc entre deux nœuds peut alors être signalée par des valeurs aberrantes (par exemple –1 ou « infini » pour des distances, « rien » pour le graphe étiqueté, ...).

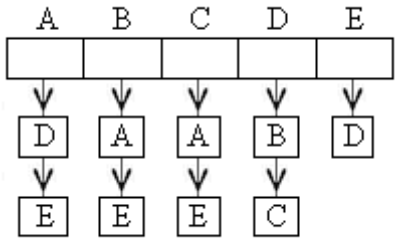
N.B. : Quelle que soit la configuration choisie, la représentation matricielle est cependant coûteuse en mémoire, car beaucoup d'éléments restent en général inoccupés. La matrice creuse (voir ex. dans le chapitre sur les listes chainées) est plus appropriée à ce type de contenu.

8.3.2.2 Représentation par un tableau de listes

L'idée est de faire correspondre à chaque nœud la liste des nœuds qui lui sont adjacents (pour un graphe orienté, ce sera la liste des nœuds pour lesquels un arc part du nœud considéré). Ces listes (qui peuvent éventuellement être chainées) sont contenues dans un tableau, le **tableau des listes d'adjacence**, dont les indices correspondent à la numérotation des nœuds. Les variantes de ce type de représentation sont nombreuses.

L'avantage principal de cette représentation est le gain de place en mémoire. Le désavantage est la perte de l'accès direct à l'information : dans la matrice, on voit directement si i et j sont adjacents, dans le cas du tableau, il faut faire une recherche pour voir si j se trouve dans la liste d'indice i.

Exemple : voici une représentation du graphe orienté de la page précédente sous forme de tableau de listes.



Au lieu de ce schéma, le contenu de ce tableau peut aussi être décrit par l'écriture plus condensée suivante : (D, E), (A, E), (A, E), (B, C), (D).

8.4 Problèmes divers

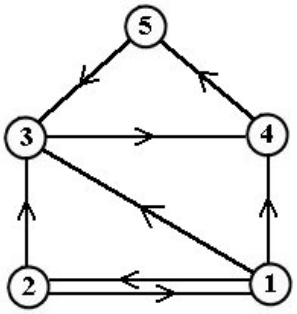
8.4.1 Accessibilité d'un nœud à partir d'un autre

Un nœud j est **accessible** à partir d'un nœud i s'il existe un chemin partant de i et arrivant à j. L'accessibilité peut donc exister pour deux nœuds non adjacents. Dans un graphe non orienté, l'accessibilité est assez évidente à établir : tous les nœuds d'une composante connexe du graphe sont forcément accessibles. Le problème est moins immédiat et plus intéressant pour les graphes orientés, et nous ne considérerons que ceux-ci dans cette section.

8.4.1.1 Matrice d'accessibilité

Similairement à la matrice d'adjacence (qui indique si un nœud i est en contact direct avec un nœud j), la **matrice d'accessibilité** est une matrice booléenne qui indique si à partir d'un nœud i, il existe un chemin qui mène au nœud j. Comme précédemment, l'indice ligne i est associé à l'origine et l'indice colonne j à l'arrivée du chemin.

Exemple : Voici un graphe et sa matrice d'accessibilité. On voit facilement que tous les nœuds sont accessibles à partir des nœuds 1 et 2. Par contre, les nœuds 3, 4 et 5 forment un cycle de longueur 3 duquel il n'est plus possible de retourner en 1 ou en 2.



$A =$

V	V	V	V	V
V	V	V	V	V
F	F	V	V	V
F	F	V	V	V
F	F	V	V	V

Nous allons voir deux façons de générer la matrice d'accessibilité en partant de la matrice d'adjacence d'un graphe.

8.4.1.2 Petit rappel de calcul matriciel

La somme de deux matrices de mêmes dimensions s'obtient en additionnant les éléments de mêmes indices dans les deux matrices.

Par exemple :

1	0	4
–2	2	1
3	0	–1

$+$

3	2	1
2	0	3
–2	2	0

$=$

4	2	5
0	2	4
1	2	–1

La multiplication est un peu plus complexe : l'élément d'indices (i, j) du produit de deux matrices est le *produit scalaire* de la $i^{\text{ème}}$ ligne de la première matrice par la $j^{\text{ème}}$ colonne de la seconde. Le produit scalaire de deux vecteurs (ou tableaux) de même taille est la somme des produits des éléments de mêmes indices dans ces vecteurs.

En formule cela donne :

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Attention, le produit matriciel n'est pas commutatif, donc $AB \neq BA$ en général.

Exemple : pour obtenir l'élément d'indices $(2, 1)$ du produit, on calcule le produit scalaire de la $2^{\text{ème}}$ ligne de la $1^{\text{ère}}$ matrice et de la $1^{\text{ère}}$ colonne de la $2^{\text{ème}}$, soit $-2*3+2*2+1*(-2) = -4$.

1	0	4
–2	2	1
3	0	–1

$*$

3	2	1
2	0	3
–2	2	0

$=$

–5	10	1
–4	–2	4
11	4	3

Dans ce qui suit, nous serons amenés à calculer la somme et le produit de matrices booléennes. Le calcul est analogue, en remplaçant toutefois les opérateurs + et * respectivement par les opérateurs logiques OU et ET.

Exemple : somme et produit de deux matrices booléennes.

V	F	V
V	V	F
F	V	F

\+

V	V	F
V	F	V
F	F	F

=

V	V	V
V	V	V
F	V	F

V	F	V
V	V	F
F	V	F

*

V	V	F
V	F	V
F	F	F

=

V	V	F
V	V	V
V	F	V

Le premier élément du produit est obtenu en calculant (V ET V) OU (F ET V) OU (V ET F) ≡ V. On voit donc que l'élément d'indices (i, j) de ce produit matriciel est vrai dès que la ligne i de la 1ère matrice et la colonne j de la 2e contiennent un élément vrai en même position.

À partir du produit matriciel, on définit encore l'élévation à une puissance d'une matrice carrée (c'est-à-dire une matrice ayant autant de lignes que de colonnes) : $A^2 = A * A$, $A^3 = A * A^2$, et récursivement, $A^n = A * A^{n-1}$.

8.4.1.3 Puissances de la matrice d'adjacence

Si M est la matrice d'adjacence d'un graphe orienté, alors la *k-ième* puissance de M possède la propriété suivante : $(M^k)_{ij}$ est vrai si et seulement s'il existe un chemin de longueur k partant de i et arrivant à j.

En effet, comme $(M^2)_{ij} = \sum_{k=1}^{n} M_{ik} * M_{kj}$, cette somme donne vrai s'il existe au moins un k pour lequel M_{ik} et M_{kj} sont vrais simultanément, autrement dit s'il existe un nœud k où arrive un arc partant de i, et d'où part un arc allant vers j, et donc s'il existe un chemin de longueur 2 entre i et$j$. Par récurrence, on prouve facilement que la propriété est valide pour les puissances suivantes.

Supposons maintenant qu'au moins un élément (i, j) parmi les matrices M, M^2, M^3, ..., M^n est vrai (où n est le nombre de nœuds du graphe), alors il existe au moins un chemin (de longueur maximale n) entre les nœuds i et j. On en déduit la formule suivante pour la matrice d'accessibilité : $A = M + M^2 + M^3 + \cdots + M^n = \sum_{k=1}^{n} M^k$

Exemple : calculons les puissances successives de la matrice d'adjacence du graphe précédent :

$M =$

F	V	V	V	F
V	F	V	F	F
F	F	F	V	F
F	F	F	F	V
F	F	V	F	F

$M^2 =$

V	F	V	V	V
F	V	V	V	F
F	F	F	F	V
F	F	V	F	F
F	F	F	V	F

$M^3 =$

F	V	V	V	V
V	F	V	V	V
F	F	V	F	F

$M^4 =$

V	F	V	V	V
F	V	V	V	V
F	F	F	V	F

F	F	F	V	F
F	F	F	F	V

F	F	F	F	V
F	F	V	F	F

$M^5 =$

F	V	V	V	V
V	F	V	V	V
F	F	F	F	V
F	F	V	F	F
F	F	F	V	F

On vérifie facilement qu'en additionnant (logiquement) les 5 matrices, on obtient bien la matrice d'accessibilité.

Remarques.

1. Si on travaille avec des matrices à valeurs entières plutôt que booléennes (c'est-à-dire avec 1 pour **vrai** et 0 pour **faux**), $(M^k)_{ij}$ représente alors le nombre de chemins de longueur k allant de i à j. La somme des puissances de la matrice d'adjacence donne alors le nombre de chemins allant de i à j de longueur inférieure ou égale à n (nombre de nœuds).
2. L'algorithme qui découle de cette formule ne présente aucune difficulté d'élaboration. Il est cependant moins performant que celui de Roy-Warshall que nous présentons ci-dessous.
3. Le graphe obtenu à partir de la matrice d'accessibilité s'appelle la **fermeture transitive** du graphe de départ ou encore **graphe d'accessibilité**.

8.4.1.4 Algorithme de Roy-Warshall

Soit M_k la matrice booléenne définie de la façon suivante : $M_k[i,j]$ = **vrai** s'il existe un chemin du nœud i au nœud j dont les indices des autres nœuds valent au maximum k (autrement dit, qui n'utilise aucun autre nœud intermédiaire que ceux d'indices 1 à k). Cette matrice est définie pour les valeurs de k entre 0 et n inclus (n étant le nombre de nœuds).

Il résulte de cette définition que M_0 est la matrice d'adjacence et que M_n est la matrice d'accessibilité. L'algorithme de Roy-Warshall permet de calculer facilement M_k à partir de M_{k-1}, et donc d'accéder en n étapes à la matrice d'accessibilité. La règle de récurrence est la suivante :

$$M_k[i,j] \text{ est vrai} \Leftrightarrow M_{k-1}[i,j] \text{ est vrai OU } (M_{k-1}[i,k] \text{ ET } M_{k-1}[k,j] \text{ sont vrais})$$

En effet, si $M_{k-1}[i,j]$ est vrai, $M_k[i,j]$ l'est forcément. Si $M_{k-1}[i,k]$ et $M_{k-1}[k,j]$ sont tous deux vrais, cela veut dire qu'il existe un chemin allant de i à k, et un autre allant de k à j, chacun n'utilisant que les nœuds d'indices valant au maximum $k-1$. En mettant ces deux chemins bout à bout, on obtient bien un chemin de i à j dont les indices des nœuds sont au maximum k.

Un des avantages de l'algorithme est de pouvoir calculer les matrices M_k dans un seul tableau. Une fois qu'un élément [i, j] est vrai, il reste vrai à toutes les étapes restantes. Le code de l'algorithme est le suivant :

```
algorithme RoyWarshall(M↓, A↑ : tableaux[1 à n, 1 à n] de booléens)
    i, j, k : entiers
    A ← M                          // initialisation de A avec une copie de la matrice d'adjacence
    pour k de 1 à n faire
        // calcul des éléments de A_k
        pour i de 1 à n faire
            pour j de 1 à n faire
                A[i, j] ← A[i, j] ou (A[i, k] et A[k, j])
            fin pour
        fin pour
    fin pour
    // A contient à présent la matrice d'accessibilité
fin algorithme
```

Exemple : Toujours pour le même graphe, voici les états successifs de la matrice obtenue par l'algorithme de Roy-Warshall. Les nouveaux éléments devenus à vrai sont indiqués en grisé.

$M_0 =$

F	V	V	V	F
V	F	V	F	F
F	F	F	V	F
F	F	F	F	V
F	F	V	F	F

$M_1 =$

F	V	V	V	F
V	V	V	V	F
F	F	F	V	F
F	F	F	F	V
F	F	V	F	F

$M_2 =$

V	V	V	V	F
V	V	V	V	F
F	F	F	V	F
F	F	F	F	V
F	F	V	F	F

$M_3 =$

V	V	V	V	F
V	V	V	V	F
F	F	F	V	F
F	F	F	F	V
F	F	V	V	F

$M_4 =$

V	V	V	V	V
V	V	V	V	V
F	F	F	V	V
F	F	F	F	V
F	F	V	V	V

$M_5 =$

V	V	V	V	V
V	V	V	V	V
F	F	V	V	V
F	F	V	V	V
F	F	V	V	V

8.4.2 Problèmes d'optimisation

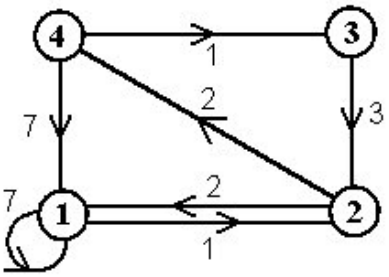
Ce type de problèmes s'applique aux graphes pondérés, orientés ou non. Il en existe beaucoup de variantes : détermination du chemin de poids ou de coût minimum, de la distance minimale ou du chemin le plus court entre deux nœuds d'un graphe, du chemin de poids minimum passant par tous les nœuds d'un graphe (problème du voyageur de commerce), la recherche de l'arbre de poids minimum inclus dans un graphe (algorithme de Kruskal), etc.

Pour représenter un graphe pondéré, on utilise une matrice des poids (entiers ou réels) W, où W_{ij} est une valeur positive représentant le poids du lien existant entre les nœuds i et j. L'absence de lien doit être signalée par une valeur aberrante, qui selon le contexte peut être négative, nulle ou infinie. Dans le cas de la recherche du minimum, les liens inexistants doivent être initialisés à une valeur « infinie » (plus techniquement une constante du type *highest value*).

Une légère modification de l'algorithme de Roy-Warshall permet de construire la matrice M des poids minimum entre les différents nœuds d'un graphe. À l'étape k, on compare la valeur M_{ij} connue jusque là avec le poids d'un chemin passant par le nœud k. Si ce détour s'avère plus coûteux, on garde M_{ij} ; si par contre le détour par le nœud k allège le poids, M_{ij} est remplacé alors par le poids de ce nouveau chemin, soit $M_{ik} + M_{kj}$.

```
algorithme poidsMinimum(W↓, M↑ : tableaux[1 à n, 1 à n] d'entiers)                    // réels
    i, j, k : entiers
    M ← W                                  // initialisation de M avec une copie de la matrice des poids
    pour k de 1 à n faire
        pour i de 1 à n faire
            pour j de 1 à n faire
                M[i, j] ← min(M[i, j], M[i, k] + M[k, j])
            fin pour
        fin pour
    fin pour
    // M contient à présent la matrice des poids minimum
fin algorithme
```

Exemple : Voici pour le graphe suivant les contenus des matrices obtenues à chaque étape de l'algorithme de recherche des poids minimum.



état initial

7	1	∞	∞
2	∞	∞	2
∞	3	∞	∞
7	∞	1	∞

k=1

7	1	∞	∞
2	3	∞	2
∞	3	∞	∞
7	8	1	∞

k=2

3	1	∞	3
2	3	∞	2
5	3	∞	5
7	8	1	10

k=3

3	1	∞	3
2	3	∞	2
5	3	∞	5
6	4	1	6

k=4

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

Remarque. L'algorithme ci-dessus détermine les poids minimum entre toutes les paires de nœuds du graphe. Il existe de nombreuses variantes, dont l'**algorithme de Dijkstra** qui recherche les poids uniquement entre un nœud donné et les autres nœuds du graphe. Cet algorithme utilise seulement un tableau de n valeurs (au lieu d'une matrice n x n).

8.4.3 Parcours d'un graphe

Comment parcourir les nœuds d'un graphe ? Plusieurs solutions sont possibles. On pourrait bien sûr les parcourir dans l'ordre de leur énumération, mais ce parcours ne tiendrait pas compte des liens entre les nœuds. Deux parcours particuliers apparaissent dans la théorie des graphes, selon le type de problème à résoudre : le parcours *par contagion* et le parcours *par sondage*.

Au cours de l'exécution des algorithmes correspondant à ces parcours, chaque nœud sera dans un des 3 états suivants : **Prêt** (état initial), **Attente** (en attente d'être traité) ou **Traité**.

Ces états peuvent être stockés dans un tableau de chaines, ou éventuellement de booléens (en assimilant les états **Attente** et **Traité** en un seul état « **Visité** »). Dans les deux algorithmes ci-dessous, la détermination des nœuds voisins est très aisée, que ce soit par la

matrice d'adjacence ou par le tableau de listes. Les algorithmes sont rédigés en « macro-logique », afin de pouvoir les adapter facilement selon le choix de la représentation choisie.

8.4.3.1 Parcours par contagion

À partir d'un nœud de départ, on visite d'abord ses voisins, puis les voisins des voisins et ainsi de suite, ... On visite donc le graphe par « couches », d'abord les voisins dans l'entourage immédiat, puis on s'éloigne petit à petit de plus en plus loin...

```
algorithme ParcoursContagion(départ, ...)    // la représentation du graphe n'est pas précisée ici
    // initialiser tous les états des nœuds à prêt
    état de départ ← Attente
    file.enfiler(départ)
    tant que NON file.estVide( ) faire
        nœud ← file.défiler( )
        // traitement du nœud
        état de nœud ← Traité
        pour tous les voisins de nœuds de
            si alorsà état de voisin = Prêt faire
                état de voisin ← Attente
                file.enfiler(voisin)
            fin si
        fin pour
    fin tant que
fin algorithme
```

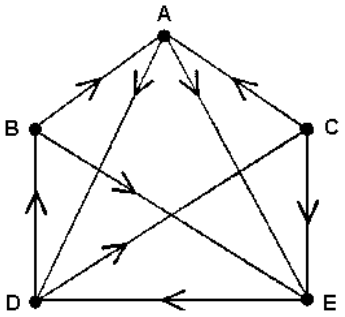
8.4.3.2 Parcours par sondage

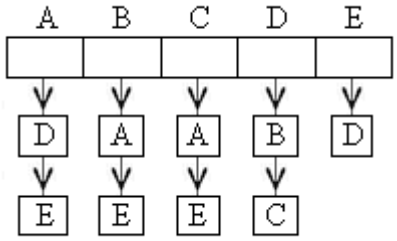
L'algorithme est semblable au précédent, à part que l'on utilise ici une pile au lieu d'une file. Il en résulte le parcours suivant : à partir du nœud de départ, on visite un voisin, puis immédiatement un voisin de ce voisin et ainsi de suite, on s'éloigne le plus loin possible jusqu'à un premier cul-de-sac. On revient alors en arrière et on recommence dans la première voie non encore visitée.

```
algorithme ParcoursSondage(départ, ...)      // la représentation du graphe n'est pas précisée ici
    // initialiser tous les états des nœuds à prêt
    état de départ ← Attente
    pile.empiler(départ)
    tant que NON pile.estVide( ) faire
        nœud ← pile.dépiler( )
        // traitement du nœud
        état de nœud ← Traité
        pour tous les voisins de nœuds de
            si alorsà état de voisin = Prêt faire
                état de voisin ← Attente
                pile.empiler(voisin)
            fin si
        fin pour
    fin tant que
fin algorithme
```

Remarque. Noter que dans ces algorithmes, il n'y a pas de contrainte sur l'ordre de visite des voisins, celui-ci dépendra en partie de l'énumération des nœuds choisie ou du parcours des listes d'adjacence.

Exemple : Dans le graphe suivant, en supposant que les voisins soient parcourus dans l'ordre des listes du tableau, le parcours par contagion à partir de A donnera A, D, E, B, C et le parcours par sondage A, E, D, C, B.





8.5 Exercices

1

Soit un graphe orienté dont la matrice d'adjacence est la suivante :

	A	B	C	D	E
A	F	V	F	V	F
B	F	F	V	F	F
C	V	F	F	F	F
D	F	F	F	F	V
E	V	F	F	F	F

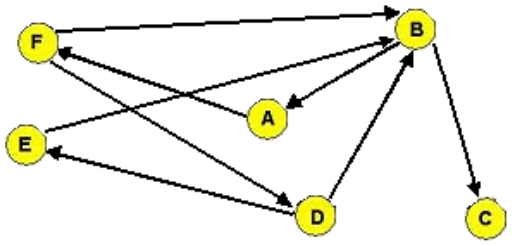
1. Dessiner le graphe correspondant
2. Établir la matrice d'accessibilité par l'algorithme de Roy-Warshall
3. Que représente la matrice obtenue à la $3^{\text{ème}}$ étape de cet algorithme ?

2

Un graphe orienté est représenté par un tableau de listes d'adjacence. Le contenu de ce tableau est (2), ( ), (1,3), (2, 4), (3, 5, 7), (4, 6), (5, 7), (4, 6). Dessiner un graphe correspondant.

3

Soit le graphe orienté suivant :



- ▷ Donner sa matrice d'adjacence, en faisant correspondre les lettres prises dans l'ordre alphabétique aux indices pris dans l'ordre croissant.
- ▷ Ce graphe est-il complet ? Expliquer.
- ▷ Ce graphe contient-il des nœuds particuliers ? Si oui, lesquels.
- ▷ Existe-t-il un chemin reliant B à E ? Si oui lequel. Dans quelle étape de l'algorithme de Roy-Warshall ce chemin apparait-il ?
- ▷ Que donne l'énumération des valeurs des nœuds donnée par contagion à partir du nœud F ?
- ▷ Même question par sondage.

4

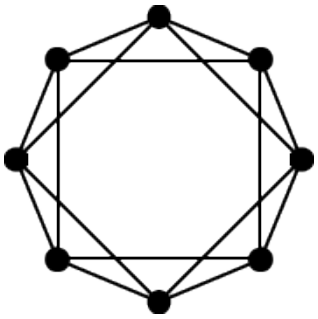
Soit un graphe orienté à n nœuds (avec $n > 1$). Comment relier ces nœuds avec le nombre minimum d'arcs pour obtenir un graphe dont la matrice d'accessibilité ne contienne que des éléments vrais ? Quel est le nombre minimum d'arcs requis ?

5

Est-il possible que la matrice d'accessibilité d'un graphe soit identique à sa matrice d'adjacence ?

6 **Le n-gone**

Écrire un algorithme qui reçoit un entier n en paramètre ($n \geq 5$) et retourne la matrice d'adjacence d'un graphe ayant la forme suivante : le contour est un n-gone, et chaque nœud est relié à ses 4 voisins les plus proches, comme le suggère le dessin suivant dans le cas $n = 8$.



7 **Suppression d'arêtes**

Soit un graphe dont les valeurs attachées aux nœuds sont contenues dans un tableau valNœuds de n entiers, et soit M sa matrice d'adjacence (tableau $n \times n$ de booléens). Écrire un algorithme qui supprime de ce graphe toutes les arêtes reliant deux nœuds de valeurs paires.

8 **De la matrice d'adjacence au tableau de listes**

Soit un graphe orienté donné par sa matrice d'adjacence ($n \times n$). Écrire un algorithme qui établit la représentation de ce graphe par un tableau de listes d'entiers (a) de la classe Liste (b) de la classe ListeChainée.

9 **Du tableau de listes à la matrice d'adjacence**

Problème inverse : soit un graphe orienté donné par un tableau de listes chainées d'entiers, écrire un algorithme qui détermine la matrice d'adjacence de ce graphe.

10 **Le degré maximal (version graphe non orienté)**

Soit un graphe non orienté donné par sa matrice d'adjacence (a) symétrique (b) triangulaire supérieure. Écrire un algorithme qui détermine le nœud de plus grand degré de ce graphe. En cas d'ex-æquo, on retournera le nœud de plus petit indice.

11 **Le degré maximal (version graphe orienté)**

Pour un graphe orienté donné par un tableau de listes d'adjacence, écrire un algorithme qui détermine le nœud de degré maximum. En cas d'ex-æquo, on retournera le nœud de plus petit indice. Les listes du tableau sont de type Liste<entier>.

12 **Puits et sources**

Soit un graphe orienté donné par sa matrice d'adjacence. Écrire un algorithme qui détermine le nombre de puits du graphe.
Même question avec les sources.

13 **Accessibilité avec puissances**

Écrire l'algorithme qui calcule la matrice d'accessibilité d'un graphe orienté par la formule de la somme des puissances de la matrice d'adjacence. Quelle est la complexité de cet algorithme ? Comparer avec la complexité de l'algorithme de Roy-Warshall.

14 **Accessibilité via tableau de listes**

Soit un graphe orienté donné par un tableau de listes chainées. Écrire l'algorithme qui donne sous la même forme le graphe issu de la matrice d'accessibilité du graphe de départ.

15 Parcours par contagion

Soit un graphe orienté donné par un tableau de listes (de la classe Liste). Détailler l'algorithme qui permet de parcourir le graphe par contagion à partir d'un nœud de départ entré en paramètre, en vous basant sur la « macro-logique » donnée en 8.4.3.1.

16 Les racines

Dans un graphe orienté, une **racine** est un nœud à partir duquel on peut atteindre tous les autres nœuds. Écrire un algorithme qui reçoit en paramètre la matrice d'adjacence du graphe et un numéro de nœud, et retourne un booléen indiquant si ce nœud est une racine.

17 Les sous-composantes connexes

Soit un graphe non orienté donné par sa matrice d'adjacence. Écrire un algorithme qui détermine le nombre de sous-composantes connexes de ce graphe.
Même question pour un graphe orienté.

Pour vous aider, l'idée est la suivante : il faut travailler avec un algorithme de parcours fonctionnant avec un tableau d'état. On lance l'algorithme avec le nœud 1, et on initialise un cpt à 1. À la fin du parcours, tous les nœuds dans la même composante connexe que 1 sont traités. On cherche s'il reste un nœud non traité, et on relance la recherche à partir de ce nœud (et on incrément le cpt). Et on réitère le processus jusqu'à ce qu'il ne reste plus de nœuds non traités. À la fin, le cpt donne le nombre de composantes connexes.

18 Le métro

Un réseau de métro est composé de plusieurs lignes qui peuvent se croiser en certaines stations. Les lignes sont implémentées dans un tableau **lignes** de listes : chaque élément du tableau est une liste chainée bidirectionnelle contenant les noms des stations dans l'ordre de parcours d'une ligne donnée. Deux stations distinctes ont nécessairement des noms différents et une station n'apparait qu'une seule fois pour chaque ligne.

On demande d'implémenter une classe Réseau, en complétant le code du constructeur et des 3 méthodes décrites ci-dessous :

```
classe Réseau
 privé:                                          // à vous de donner les éléments privés
 public:
    constructeur Réseau(lignes : tableau [1 à n] de ListeBD<chaine>)
    méthode getStations() → tableau [1 à m] de chaines
    // retourne le tableau des noms de toutes les stations du réseau
    méthode getAdjacence() → tableau [1 à m, 1 à m] de booléens
    // retourne la matrice d'adjacence symétrique du graphe représentant le réseau. Chacune des
stations est représentée par l'indice que son nom a dans le tableau retourné par getStations().
    méthode getDistance(station1, station2 : chaines) → entier
    // retourne la matrice des longueurs des chemins les plus courts entre les différentes stations.
Ici aussi chacune des stations est représentée par l'indice que son nom a dans le tableau retourné
par getStations().
fin classe
```

Remarque : on peut supposer que le réseau comporte moins de 500 stations et que le réseau est connexe.

19 Écriture de classes

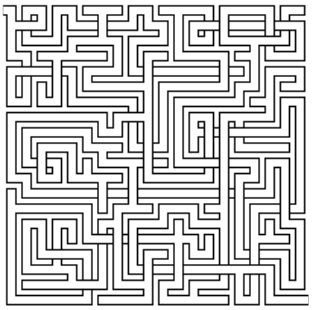
Implémenter l'interface graphe dont les méthodes sont décrites ci-dessous dans une classe (a) GrapheOrienté (b) GrapheNonOrienté.

```
interface Graphe<T>
    méthode getNbNœuds( )() → entier
    méthode ajouteNœud(nœud : T)
    méthode retireNœud(indNœud : entier) → T
    méthode setNœud(indNœud : entier, nœud : T)
    méthode getNœud(indNœud : entier) → T
    méthode sontAdjacents(indNœud1 : entier, indNœud2 : entier) → booléen
    méthode getDistance(indDepart : entier, indArrivee : entier) → entier
    méthode getAccessibles(indDepart : entier, indArrivee : entier) → booléen
    méthode getSondage(indDepart : entier) → Liste<entier>
    // retourne la liste des nœuds dans l'ordre d'un parcours par sondage
    méthode getContagion(indDepart : entier) → Liste<entier>
    // retourne la liste des nœuds dans l'ordre d'un parcours par contagion
fin interface
```

Chapitre 9

Le backtracking

Le backtracking est une technique de résolution de problème où on construit petit à petit la solution par essais/erreurs.



9.1 Définition

*Le **backtracking** (appelé aussi **retour sur trace**) est un algorithme qui consiste à revenir légèrement en arrière sur des décisions prises afin de sortir d'un blocage. La méthode des essais et erreurs constitue un exemple simple de backtracking. Le terme est surtout utilisé en programmation, où il désigne une stratégie pour trouver des solutions à des problèmes de satisfaction de contraintes.* (Wikipédia, août 2013)

Le **backtracking** s'applique lorsque

- ▷ La **solution** du problème à résoudre est faite d'une suite d'étapes ;
- ▷ à chaque étape, plusieurs voies de recherche sont possibles sans qu'il soit possible de déterminer à priori laquelle est la bonne.

Pour **résoudre un tel problème**, il faudra procéder par essais : si, à une certaine étape, on arrive à une impasse, il faut revenir en arrière et opter pour une autre voie au niveau de l'étape précédente. L'organisation de l'algorithme doit être telle qu'à chaque étape toutes les voies sont envisagées une et une seule fois (afin de ne rater aucune possibilité et de ne pas essayer plusieurs fois une voie menant à une impasse).

Un exemple typique est celui du **labyrinthe**, dans lequel il faut trouver un chemin menant de l'entrée à la sortie. Ce labyrinthe est formé de couloirs et de carrefours. À chaque carrefour il faut faire un choix parmi les directions possibles. Si on arrive à une impasse ou si on revient à un endroit déjà visité, le chemin n'est pas le bon. Il faut alors revenir sur ses pas et faire un autre choix au niveau du carrefour précédent.

Dans ce problème, la *solution*, si elle existe, est une suite de directions qu'il faut prendre à chaque carrefour (par exemple : 1er carrefour à gauche, 2e carrefour à droite, etc). Chaque carrefour de ce chemin constitue une *étape* de la solution, et les *voies de recherche* à chaque étape sont les directions possibles à un carrefour donné.

9.2 Structure générale de l'algorithme

Nous distinguerons 2 types d'algorithmes de backtracking : celui qui recherche *une* solution (si il en existe au moins une !) et celui qui recherche *toutes* les solutions.

Dans les deux cas, il est essentiel de se poser les questions suivantes avant de se lancer dans l'écriture de l'algorithme :

- ▷ tout d'abord, le problème posé est-il bien du type « backtracking » ? Peut-il être résolu par la démarche d'essai-erreur ?
- ▷ cherche-t-on toutes les solutions ou une seule ?
- ▷ qu'est-ce qui donne exactement la (ou une) *solution* au problème ? quelle représentation choisir ?
- ▷ que représente une *étape* de la solution au problème ? Le nombre d'étapes est-il connu à l'avance ?
- ▷ quelles sont les *voies de recherches* possibles à chaque étape de l'algorithme ? Sont-elle pareilles à chaque étape ?

L'algorithme de backtracking est essentiellement récursif : il doit posséder comme paramètre la *solution en construction* du problème et aussi souvent un paramètre d'étape. Au départ la solution est « vide » (ou ne contient que l'étape de départ) et est complétée au fur et à mesure que l'on progresse dans les étapes du problème. Elle peut aussi être « démontée » en cas de marche arrière. Si la dernière étape est atteinte, on dit alors que la solution est *complète*, et fournit donc une solution au problème posé. À chaque étape, il est aussi nécessaire d'examiner les voies possibles, et ne choisir que celles qui sont *acceptables*, c'est-à-dire celles qui peuvent compléter la solution en construction dans le respect des contraintes du problème.

L'algorithme backtracking sera donc enclenché par un algorithme « façade » qui traitera de l'initialisation des données ainsi que de l'affichage de la solution du problème. La vérification de l'acceptabilité d'une voie de recherche sera aussi souvent traitée dans un algorithme « acceptable » afin de ne pas surcharger l'algorithme principal.

9.2.1 Trouver une solution

Le code ci-dessous est plus un canevas qu'un algorithme. C'est un modèle très basique d'algorithme pour résoudre un problème de backtraking. Les parties en italiques gras devront être complétées et adaptées en fonction du problème précis.

```
algorithme trouver_une_solution(paramètres du problème)
    trouvé : booléen
    déclarer et initialiser la solution
    trouvé ← backtracking(solution, [paramètre d'étape, autres paramètres utiles])
    si trouvé alors
        afficher solution
    sinon
        afficher « pas de solution au problème ! »
    fin si
fin algorithme
```

```
algorithme backtracking(solution en construction↓↑, [paramètre d'étape↓, ...]) → booléen
    réussite : booléen
    réussite ← faux
    Initialiser les voies de recherche
    faire
        Choisir une voie non encore choisie
        si la voie est acceptable alors
            ajouter la voie à la solution en cours
            si la solution est complète alors
                réussite ← vrai
            sinon
                réussite ← backtracking(solution en construction, [étape suivante, ...])
            fin si
            si NON réussite alors
                Enlever la voie de la solution
            fin si
        fin si
    until réussite OU toutes les voies ont été explorées
        retourner réussite
    fin algorithme
```

9.2.2 Trouver toutes les solutions ?

Dans ce 2e cas, la variable réussite n'est plus utile, l'algorithme ne s'arrête que lorsque toutes les voies ont été explorées à toutes les étapes du problème. Lorsqu'une solution est complète, on peut soit l'afficher, soit l'ajouter dans une liste de solution ; si on se contente de savoir combien de solutions existent, on peut aussi simplement incrémenter un compteur (qui sera un paramètre entrant et sortant de l'algorithme backtracking). Noter que dans ce cas, l'algorithme backtracking est un algorithme sans renvoi de valeur.

```
algorithme trouver_toutes_les_solutions(paramètres du problème)
    déclarer et initialiser la solution
    backtracking(solution, [paramètre d'étape, autres paramètres utiles])
fin algorithme
```

```
algorithme backtracking(solution en construction↓↑, [paramètre d'étape↓, ...])
    Initialiser les voies de recherche
    faire
        Choisir une voie non encore choisie
        si la voie est acceptable alors
            ajouter la voie à la solution en cours
            si la solution est complète alors
                afficher la solution
            sinon
                backtracking(solution en construction, [étape suivante, ...])
            fin si
            Enlever la voie de la solution
        fin si
    until toutes les voies ont été explorées
    fin algorithme
```

N.B. : le code consistant à enlever *la voie de la solution* peut souvent être omis, surtout lorsqu'on construit la solution dans un tableau. En effet, l'aller et retour qui se produit lors du backtracking a pour effet d'écraser les portions de solutions abandonnées aux étapes précédentes par des nouvelles valeurs. Il est aussi courant d'*ajouter la voie à la solution en cours* avant de vérifier si elle est acceptable, pour la même raison.

9.3 Un exemple : les 8 reines

Le problème des 8 reines est un exemple classique pour présenter le backtracking. Le but est de disposer 8 reines sur un échiquier de telle manière que 2 quelconques d'entre elles ne soient pas mutuellement en position de prise, ce qui revient à dire que 2 de ces 8 reines ne peuvent pas se trouver sur une même ligne, sur une même colonne ou sur une même diagonale.

9.3.1 Comment représenter la solution ?

La solution, si elle existe, est l'ensemble des 8 cases sur lesquelles peuvent se trouver les 8 reines. Il serait possible d'imaginer un algorithme qui va investiguer tous les sous-ensembles de 8 cases possibles sur un échiquier (soit $C_{64}^{8} = 4426165368$ possibilités !), ce qui serait assez lent. On peut faire beaucoup mieux en partant du fait qu'il ne peut y avoir 2 reines sur une même ligne et donc que chaque reine est placée dans une ligne différente. La solution sera alors stockée dans un simple tableau `reines` de 8 entiers, où `reines[i]` contiendra l'indice de la colonne correspondant à la reine dans la ligne d'indice `i`. L'algorithme testera donc plutôt toutes les positions possibles des 8 reines dans leur ligne, soit $8^8 = 16777216$ possibilités (soit 264 fois moins).

Attention, les chiffres donnés ne donnent qu'un ordre de grandeur pour la complexité des algorithmes : il ne faut pas oublier que le backtracking court-circuite la recherche en faisant marche arrière lorsque qu'une portion de solution non conforme est rencontrée.

9.3.2 Étapes et voies de recherche

À l'étape `i` de la recherche d'une solution, on va placer une reine dans la ligne `i`. Il y a donc 8 étapes, et les voies de recherche lors d'une étape donnée sont les 8 cases de cette ligne. Une reine pourra être placée dans une case donnée si elle n'entre pas en conflit avec les reines déjà placées jusqu'ici : elle ne peut être ni dans une même colonne ni dans une même diagonale qu'une reine précédemment placée. Par la représentation choisie, il n'est plus nécessaire de vérifier si 2 reines sont dans une même ligne.

Voici une version de l'algorithme qui affichera la 1ère solution trouvée.

```
algorithme huitReines()
    reines : tableau [1 à 8] d'entiers
    si backtracking(reines, 1) alors
        afficher reines
    sinon
        afficher « pas de solution »
    fin si
fin algorithme
```

```
algorithme backtracking(reines↓↑ : tableau [1 à 8] d'entiers, ligne : entier) → booléen
    réussite : booléen
    colonne : entier
    réussite ← faux
    colonne ← 0
    tant que NON réussite ET colonne < 8 faire
        colonne ← colonne + 1
        reine[ligne] ← colonne
        si estAcceptable(reines, ligne) alors
            si ligne=8 alors                                    // solution complète
                réussite ← vrai
            sinon
                réussite ← backtracking(reines, ligne+1)
            fin si
        fin si
    fin tant que
    retourner réussite
fin algorithme
```

```
algorithme estAcceptable(reines : tableau [1 à 8] d'entiers, ligne : entier) → booléen
    // il faut vérifier que la dernière reine ajoutée dans le tableau
    // (de coordonnées (ligne, reines[ligne]))
    // n'est pas dans une même colonne ou une même diagonale
    // que les autres reines déjà dans la solution
    // (de coordonnées (i, reines[i]), où i est compris entre 1 et ligne–1)
    i : entier
    ok : booléen
    i ← 1
    ok ← vrai
    tant que ok ET i < ligne faire
        ok ← reines[i] ≠ reines[ligne] ET ligne-i ≠ |reines[i] - reines[ligne]|
        i ← i+1
    fin tant que
    retourner ok
fin algorithme
```

9.4 Le parcours minimum

Soit un tableau à deux dimensions d'entiers positifs. On veut trouver dans ce tableau le parcours menant de la case en haut à gauche vers la case en bas à droite donnant le minimum de la somme des valeurs des cases parcourues. Par parcours, on entend une suite de cases adjacentes par un coté (dont toutes les cases sont logiquement distinctes).

Exemple :

1	3	2	2	4	7	6
8	9	4	8	3	3	3
6	1	2	6	10	8	5
8	2	7	5	7	9	3
5	2	1	2	6	5	2
4	3	4	3	2	3	2
2	2	5	6	7	2	1

Le parcours grisé donne la solution du problème, avec une somme de 31. Tout autre parcours donnerait une valeur supérieure. Remarquez que le parcours donné par la solution n'est pas

le plus court en nombre de cases ; notez aussi que l'algorithme qui consisterait à choisir à chaque étape la case de valeur minimale parmi les voisines de la précédente ne donnerait pas le résultat correct (on arriverait à un total de 34 avec l'exemple donné).

Nous sommes donc dans un type de problème où le nombre d'étapes (qui sont ici les cases du parcours) est inconnu à l'avance. Plutôt que de travailler avec un tableau, nous utiliserons une liste de cases pour stocker la solution optimale. Les voies de recherches à chaque étape sont les 4 cases voisines. Pour accepter une nouvelle case dans la solution en construction, on doit vérifier qu'on ne sort pas du tableau (ceci concerne les cases situées sur le bord), qu'on ne repasse pas par une case déjà visitée (ce qui fait que seulement 3 cases voisines de la case précédente seront acceptables à priori), que la somme temporaire ne soit pas plus grande qu'une précédente valeur de la somme optimale.

```
structure Case
    ligne : entier
    colonne : entier
fin structure
```

```
algorithme parcoursMinimum(tab : tableau[1 à n, 1 à m] d'entiers) → Liste<Case>
    parcoursOptimal : Liste<Case>                    // cette liste stockera la meilleure solution
    parcours : Liste<Case>                  // cette liste stockera la solution en construction
    case : Case                              // contient les coordonnées de la case courante
    somme : entier
    sommeMin : entier
    parcoursOptimal ← nouveau Liste<Case>
    init(tab, parcoursOptimal, sommeMin)                                    // voir ci-dessous
    case.ligne ← 1
    case.colonne ← 1
    parcours ← nouveau Liste<Case>
    parcours.ajouter(case)                        // on initialise le parcours avec la 1ère case
    somme ← tab[1, 1]
    backtracking(tab, parcours, parcoursOptimal, somme, sommeMin)
    retourner parcoursOptimal
fin algorithme
```

```
algorithme backtracking(tab : tableau[1 à n, 1 à m] d'entiers, parcours : Liste<Case>, parcoursOptimal : Liste<Case>, somme↓ : entier, sommeMin↓↑ : entier)
    i : entier
    case : Case                                                    // nouvelle case visitée
    dernièreCase : Case                                            // dernière case visitée
    dernièreCase ← parcours.get(parcours.taille())
    pour i de 1 à 4 faire                                          // pour explorer dans les 4 directions
        case.ligne ← dernièreCase.ligne
        case.colonne ← dernièreCase.colonne
        selon que i  vaut vaut
         1:
            // vers la droite
            case.colonne ← dernièreCase.colonne + 1
         2:
            // vers le bas
            case.ligne ← dernièreCase.ligne + 1
         3:
            // vers la gauche
            case.colonne ← dernièreCase.colonne – 1
         4:
            // vers le haut
            case.ligne ← dernièreCase.ligne – 1
        fin selon que
        si estAcceptable(tab, parcours, somme, sommeMin, case) alors
            // on ajoute la nouvelle case dans la solution en construction
            parcours.ajouter(case)
            somme ← somme + tab[case.ligne, case.colonne]
            si case.ligne = n ET case.colonne = m alors             // solution complète
                si somme < sommeMin alors
                    // on remplace le parcours optimal par le nouveau
                    copierListe(parcoursOptimal, parcours)         // voir ci-dessous
                    sommeMin ← somme
                fin si
            sinon
                backtracking(tab, parcours, parcoursOptimal, somme, sommeMin)
            fin si
            // on supprime la dernière case pour laisser
            // la place à la voie suivante
            parcours.supprimer()
            // on retire aussi la valeur de la case de la somme
            somme ← somme - tab[case.ligne, case.colonne]
        fin si
    fin pour
fin algorithme
```

```
algorithme estAcceptable(tab : tableau[1 à n, 1 à m] d'entiers, parcours : Liste<Case>, somme↓ :
entier, sommeMin↓ : entier, case↓ : Case) → booléen
    ok : booléen
    i : entier
    // on vérifie d'abord si la nouvelle case est bien dans le tableau
    ok ← case.ligne ≥ 1 ET case.ligne ≤ n ET case.colonne ≥ 1 ET case.colonne ≤ m
    // ensuite on vérifie que la case est différente des cases précédentes du parcours
    i ← 1
    tant que ok ET i ≤ parcours.taille() faire
        ok ← case.ligne ≠ parcours.get(i).ligne OU case.colonne ≠ parcours.get(i).colonne
        i ← i + 1
    fin tant que
    // enfin, on vérifie si la somme partielle obtenue avec la nouvelle case
    // ne dépasse pas le minimum déjà obtenu avec un parcours complet
    si ok alors
        ok ← somme + tab[case.ligne, case.colonne] < sommeMin
    fin si
    retourner ok
fin algorithme
```

```
// la somme minimum doit être initialisée ; pour cela, on choisit un parcours arbitraire
// (celui longeant les bords supérieur et droit) avec lequel on initialise aussi le parcours optimal.
algorithme init(tab : tableau[1 à n, 1 à m] d'entiers, parcoursOptimal : Liste<Case>, sommeMin↑ :
entier)
    case : Case
    i : entier
    sommeMin ← 0
    pour i de 1 à m faire
        sommeMin ← sommeMin + tab[1, i]
        case.ligne ← 1
        case.colonne ← i
        parcoursOptimal.ajouter(case)
    fin pour
    pour i de 2 à n faire
        sommeMin ← sommeMin + tab[i, m]
        case.ligne ← i
        case.colonne ← m
        parcoursOptimal.ajouter(case)
    fin pour
fin algorithme
```

```
algorithme copierListe(liste1, liste2 : Liste<Case>)
    // remplace la liste 1 par une copie de la liste 2
    i : entier
    liste1.vider()
    pour i de 1 à liste2.taille() faire
        liste1.ajouter(liste2.get(i))
    fin pour
fin algorithme
```

9.5 Exercices

1 Les reines

Adapter l'exemple des reines de façon à ce que l'algorithme affiche toutes les solutions. Ajouter aussi un paramètre afin de résoudre le problème pour un nombre quelconque n de reines placées sur un quadrillage n x n.

2 La chaine

Construire une chaine de 100 occurrences des caractères 'A', 'B' et 'C' de manière à ce qu'il n'y ait jamais de sous-chaines adjacentes de mêmes longueurs qui soient identiques.

Exemples :

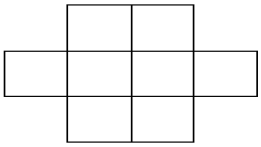
- ▷ AABC est incorrect
- ▷ ABCBC est incorrect
- ▷ ABCBACAB est correct (mais pas de longueur 100)

3 Le carré magique

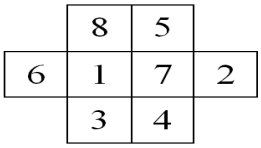
Construire un carré magique de 9 cases (3x3) dans lesquelles apparaissent chacun des nombres de 1 à 9. Pour rappel, la somme des éléments de chaque ligne, de chaque colonne et des deux diagonales d'un carré magique doit être toujours la même. Écrire un algorithme qui recherche toutes les solutions.

4 Le petit casse-tête

Le but est de placer chacun des 8 nombres entiers de 1 à 8 (donc 1, 2, 3, 4, 5, 6, 7, 8) dans la grille ci-dessous, sans que deux nombres consécutifs soient voisins (c'est-à-dire qu'ils ne peuvent occuper des cases adjacentes par un coté ou par un coin).



Pour fixer les choses, voici une solution fausse du petit casse-tête :



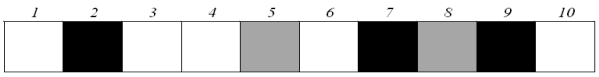
Il y a 2 erreurs : 3 et 4 sont voisins (par un coté) et 7 et 8 sont aussi voisins (par un coin). Écrire un algorithme qui recherche toutes les solutions de ce petit casse-tête.

5 La fresque tricolore

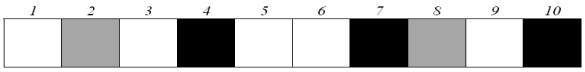
Un artiste veut réaliser une fresque constituée d'une série de carrés disposés côte à côte. Chaque carré sera peint soit en noir, en blanc ou en gris, avec la condition que 3 carrés espacés d'une même distance ne soient pas peints dans la même couleur.

Par « 3 carrés espacés d'une même distance » on entend que la distance entre les deux premiers carrés est identique à celle entre les deux derniers, ou encore que les indices correspondants à la position des 3 carrés sont en progression arithmétique.

Exemple de solution correcte :



Exemple de solution incorrecte :



Cette solution contient 4 triples de carrés incorrects :

- ▷ les carrés 1, 3 et 5 sont de la même couleur (blanc)
- ▷ les carrés 1, 5 et 9 sont de la même couleur (blanc)
- ▷ les carrés 3, 6 et 9 sont de la même couleur (blanc)
- ▷ les carrés 4, 7 et 10 sont de la même couleur (noir)

Écrire un algorithme qui reçoit en paramètre le nombre de carrés de la fresque (soit un entier n) et affiche une possibilité de remplissage (s'il en existe).

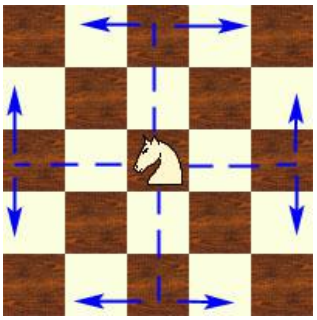
Astuce : construire la solution dans un tableau d'entiers, et ne faire apparaitre les couleurs dans l'algorithme que lors de l'affichage de la solution.

6 Les dérangements

Écrire un algorithme qui génère les dérangements des n premiers entiers. Un dérangement est une permutation telle que l'entier i n'est jamais en position i.

7 Les cavaliers

Le problème consiste à trouver la suite de déplacements que doit effectuer un cavalier sur un échiquier de manière à passer sur toutes les cases une et une seule fois. Pour rappel, un cavalier se déplace en « L ». Il avance de 2 cases en ligne droite puis tourne d'une case à droite ou à gauche.



Aide : pour effectuer les 8 déplacements possibles du cavalier, utiliser un tableau move de 8 éléments de type vecteur (structure contenant les champs entiers dl et dc). Le contenu (fixe) de ce tableau donne les 8 différences de coordonnées (ligne et colonne) possibles lors du déplacement du cavalier :

(-2, 1)	(-1, 2)	(1, 2)	(2, 1)	(2, -1)	(1, -2)	(-1, -2)	(-1, -2)

8 Saute-mouton

Un champ est partitionné en n^2 parcelles, dans lesquelles se trouvent des moutons (un mouton maximum par parcelle). Le but est de partir du coin en haut à gauche, et d'arriver en bas à droite, en rencontrant un maximum de moutons. Les seules directions autorisées sont vers la droite et vers le bas.

Pour l'implémentation du problème, le champ est donné par un tableau nxn de booléens (vrai s'il y a un mouton, faux sinon). Écrire l'algorithme qui renvoie le nombre maximum, et affiche un parcours donnant ce maximum (plusieurs parcours peuvent en effet correspondre au maximum). Le parcours sera affiché sous la forme « x parcelles vers la droite », « y parcelles vers le bas », etc.

Exemple : pour l'exemple ci-dessous, la réponse est 8 ; le parcours choisi est :

2 parcelles vers la droite

2 parcelles vers le bas

2 parcelles vers la droite

2 parcelles vers le bas

1 parcelle vers la droite

2 parcelles vers le bas

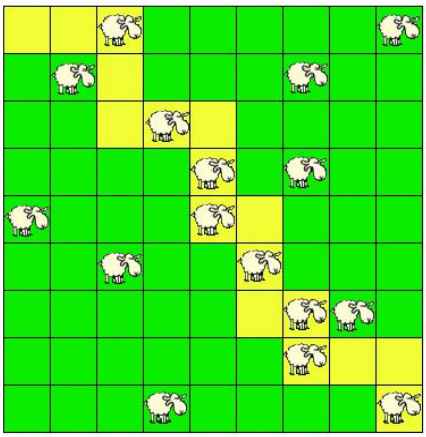
1 parcelle vers la droite

1 parcelle vers le bas

2 parcelles vers la droite

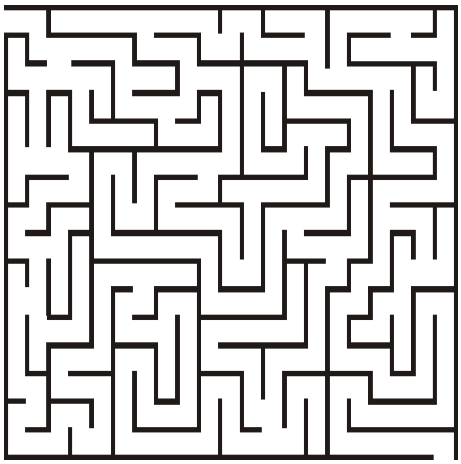
1 parcelle vers le bas

N.B. : les moutons ne se déplacent pas pendant le parcours !



9 Le labyrinthe

On considère un labyrinthe 10 x 10 formé de cellules carrées. L'entrée est en haut à gauche, et la sortie en bas à droite, comme dans le dessin ci-dessous. Écrire l'algorithme qui affiche le chemin menant à la sortie (sous forme d'une succession de directions : « à gauche », « à droite », « vers le bas », etc.)



Le labyrinthe est représenté par un tableau lab [10 x 10] dont les éléments sont du type structuré accès. Les champs de cette structure (gauche, droite, haut, bas) sont des booléens exprimant s'il y a un passage vers les cases voisines (ou vers l'extérieur du labyrinthe).

10 Le Sudoku

Écrire un algorithme qui résout une grille de Sudoku. La grille contient déjà des chiffres au départ, et il faut trouver la solution (on peut supposer que cette solution existe et qu'elle est unique).

11 Le compte juste

Écrire un algorithme retournant un booléen exprimant le fait qu'une somme à payer peut être fournie exactement avec les billets et la monnaie dont on dispose. Ceux-ci sont stockés dans un tableau d'entiers disponible, dont chaque élément représente la valeur d'une pièce de monnaie ou d'un billet du portefeuille.

Ex : Le contenu de disponible est [10, 20, 50, 50, 5, 20, 2, 10, 10, 2]. Peut-on payer 123 € ?

12 Le mot le plus long

Écrire un programme qui trouve tous les mots les plus longs qu'on peut former à partir d'un ensemble de lettres (donné par un tableau [1 à n] de caractères).

On peut utiliser une classe Dictionnaire qui connait tous les mots valides.

```
classe Dictionnaire
 public:
    constructeur Dictionnaire()
    // crée un dictionnaire avec tous les mots valides.
    méthode contientMot(mot : chaine) → booléen
    // vrai si le mot est présent dans le dictionnaire.
    méthode contientMotCommencantPar(mot : chaine) → booléen
    // vrai si le dictionnaire contient un mot commençant par la chaine donnée.
fin classe
```

Troisième partie

Les annexes

```
classe Liste <T>
 public:
    constructeur Liste <T>()
    méthode get(pos : entier) → T                          // retourne l'élément en position pos
    méthode set(pos : entier, valeur : T)                  // modifie l'élément en position pos
    méthode taille() → entier                              // retourne la taille de la liste
    méthode ajouter(valeur : T)                            // ajoute une valeur en fin de liste
    méthode insérer(pos : entier, valeur : T)              // insère un élément en position pos
    méthode supprimer()                                    // supprime le dernier élément
    méthode supprimerPos(pos : entier)                     // supprime l'élément en position pos
    méthode supprimer(valeur : T)                          // supprime l'élément de valeur donnée
    méthode vider()                                        // vide la liste
    méthode estVide() → booléen                            // indique si la liste est vide
    méthode existe(valeur↓ : T, pos↑ : entier) → booléen   // recherche un élément
fin classe
```

```
classe ÉlémentListe<T>
 privé:
    valeur : T
    suivant : ÉlémentListe<T>
 public:
    constructeur ÉlémentListe<T>(val : T, elt : ÉlémentListe<T>)
    constructeur ÉlémentListe<T>(val : T)                  // suivant à rien.
    méthode getValeur() → T
    méthode setValeur(val : T)
    méthode getSuivant() → ÉlémentListe<T>                 // renvoie rien si pas de suivant.
    méthode setSuivant(elt : ÉlémentListe<T>)
fin classe
```

```
classe ListeChainée<T>
 privé:
    premier : ÉlémentListe<T>
 public:
    constructeur ListeChainée<T>()                         // crée une liste vide
    méthode getPremier() → ÉlémentListe<T>
    méthode setPremier(elt :ÉlémentListe<T>)
    méthode estVide() → booléen
    méthode insérerTête(val : T)
    //      insère en début de liste un nouvel élément de valeur val.
    méthode supprimerTête()
    //      supprime le premier élément de la liste chainée.
    méthode insérerAprès(elt : ÉlémentListe<T>, val : T)
    //      insère un nouvel élément de valeur val après elt.
    méthode supprimerAprès(elt : ÉlémentListe<T>)
    //      supprime l'élément qui suit elt de la liste chainée.
fin classe
```

```
classe ListeCirculaire<T>
 privé:
    premier : ÉlémentListe<T>
 public:
    constructeur ListeCirculaire<T>()                                    // crée une liste vide
    méthode getPremier() → ÉlémentListe<T>          // récupère un élément de la liste (pour
pouvoir la parcourir)
    méthode estVide() → booléen
    méthode insérer(val : T)
    //      insère dans la liste un nouvel élément de valeur val.
    méthode insérerAprès(elt : ÉlémentListe<T>, val : T)
    //      insère un nouvel élément de valeur val après elt.
    méthode supprimerAprès(elt : ÉlémentListe<T>)
    //      supprime l'élément qui suit elt de la liste chainée.
fin classe
```

```
classe ÉlémentListeBD<T>
 privé:
    valeur : T
    suivant : ÉlémentListeBD<T>
    précédent : ÉlémentListeBD<T>
 public:
    constructeur ÉlémentListeBD<T>(val : T, suiv, prec : ÉlémentListeBD<T>)
    constructeur ÉlémentListeBD<T>(val : T)                  // précédent et suivant à rien
    méthode getValeur() → T
    méthode setValeur(val : T)
    méthode getSuivant() → ÉlémentListeBD<T>                  // vaut rien si pas de suivant
    méthode setSuivant(elt : ÉlémentListeBD<T>)
    méthode getPrécédent() → ÉlémentListeBD<T>              // vaut rien si pas de précédent
    méthode setPrécédent(elt : ÉlémentListeBD<T>)
fin classe
```

```
classe ListeBD<T>
 privé:
    premier : ÉlémentListeBD<T>
    dernier : ÉlémentListeBD<T>
 public:
    constructeur ListeBD<T>()
    méthode getPremier() → ÉlémentListeBD<T>
    méthode setPremier(ÉlémentListeBD<T>)
    méthode getDernier() → ÉlémentListeBD<T>
    méthode setDernier(ÉlémentListeBD<T>)
    méthode estVide() → booléen
    méthode insérerTête(val : T)
    méthode insérerFin(val : T)
    méthode insérerAprès(elt : ÉlémentListeBD<T>, val : T)
    méthode insérerAvant(elt : ÉlémentListeBD<T>, val : T)
    méthode supprimerTête()
    méthode supprimerFin()
    méthode supprimer(elt : ÉlémentListeBD<T>))
fin classe
```

```
interface Pile<T>                                    // T est le type des éléments de la pile
    méthode empiler(élément : T)                  // ajoute un élément au sommet de la pile
    méthode sommet() → T
    //      retourne la valeur de l'élément au sommet de la pile, sans le retirer
    méthode dépiler() → T                          // enlève et retourne l'élément au sommet
    méthode estVide() → booléen                                  // indique si la pile est vide
fin interface
```

Les classes qui implémentent l'interface Pile<T> sont

PileTab<T> et PileListe<T>

```
interface File<T>                                          // T est le type des éléments de la file
    méthode enfiler(élément : T)                                  // ajoute un élément dans la file
    méthode tête() → T
    // retourne la valeur de l'élément en tête de file, sans le retirer
    méthode défiler() → T                                  // enlève et retourne l'élément de tête
    méthode estVide() → booléen                                   // indique si la file est vide
fin interface
```

Les classes qui implémentent l'interface File<T> sont

FileTab<T> et FileListe<T>

```
interface Map<K, T>
    // K est le type de la clé
    // et T celui de la valeur
    méthode setÉlément(clé : K, val : T)    // ajoute ou modifie le couple dont la clé est donnée
    méthode getValeur(clé : K) → T                        // retourne la valeur associée à la clé
    méthode supprimer(clé : K)                               // retire le couple associé à la clé
    méthode contient(clé : K) → booléen       // indique si le couple dont la clé est donnée est
présent
    méthode taille() → entier                                  // retourne le nombre de couples
    méthode listeClés() → Liste<K>                        // retourne la liste des clés présentes
fin interface
```

```
classe Nœud<T>
 privé:
    valeur : T
    ListeFils : Liste<Nœud<T>>
 public:
    constructeur Nœud<T>(val : T)
    // construit un nœud sans fils
    méthode getValeur() → T
    méthode setValeur(val : T)
    méthode getNbFils() → entier
    méthode getFils(i : entier) → Nœud<T>
    méthode setFils(i : entier, fils : Nœud<T>)
    méthode ajouterFils(fils : Nœud<T>)                  // ajout à la fin de la liste des fils
    méthode supprimerFils(i : entier)
fin classe
```

```
classe Arbre<T>
 privé:
    racine  : Nœud<T>
 public:
    constructeur Arbre<T>()
    // construit un arbre vide
    méthode getRacine() → Nœud<T>
    méthode setRacine(racine : Nœud<T>)
fin classe
```

```
classe NœudBinaire<T>
 privé:
    valeur : T
    gauche : <NœudBinaire<T>>
    droit : <NœudBinaire<T>>
 public:
    constructeur NœudBinaire<T>(val : T)                    // construit un nœud sans fils
    méthode getValeur() → T
    méthode setValeur(val : T)
    méthode getGauche() → NœudBinaire<T>
    méthode setGauche(fils : NœudBinaire<T>)
    méthode getDroit() → NœudBinaire<T>
    méthode setDroit(fils : NœudBinaire<T>)
fin classe
```

```
classe ArbreBinaire<T>
 privé:
    racine : NœudBinaire<T>
 public:
    constructeur ArbreBinaire<T>()                          // construit un arbre vide
    méthode getRacine() → NœudBinaire<T>
    méthode setRacine(r : NœudBinaire<T>)
fin classe
```