Les listes.

Exercice 1

Ecrire une fonction qui lit des nombres entiers positifs au clavier et les place dans une liste simplement liée dans l'ordre inverse de leur lecture. La fin de la liste est signalée par la saisie d'une valeur négative.

Exercice 2

Ecrire une fonction qui lit des nombres entiers positifs au clavier et les place dans une liste simplement liée dans l'ordre de leur lecture. La fin de la liste est signalée par la saisie d'une valeur négative.

Exercice 3

Ecrire une fonction qui ajoute, à la bonne position, un entier à une liste ordonnée d'entiers. Si l'entier est déjà présent dans la liste, la fonction ne l'ajoute pas et retourne la valeur false, sinon elle retourne la valeur true.

Exercice 4.

Ecrire une fonction non récursive qui reçoit comme paramètre le pointeur de tête d'une liste d'entiers simplement liée et qui inverse cette liste. La fonction doit modifier la liste qui lui est passée en paramètre.

Exercice 5.

Ecrire une fonction récursive qui supprime toutes les occurrences d'un nombre donné d'une liste simplement liée de nombres entiers. La liste sera donc altérée par la fonction.

Exercice 6.

Ecrire une fonction récursive qui supprime d'une liste ordonnée d'entiers distincts les nombres présents dans une autre liste ordonnée d'entiers distincts. La première liste sera donc altérée par la fonction.

Exercice 7.

Ecrire une fonction recevant en paramètres deux listes triées, et créant une nouvelle liste triée comme étant la fusion de ces deux listes.
Les listes originales ne doivent pas être modifiées, et la nouvelle liste sera retournée par la fonction.

Exercice 8.

Un nombre entier peut être représenté sous la forme d'une liste liée de ses chiffres de sa représentation décimale rangés du plus significatif au moins significatif (ordre naturel d'écriture).

Ecrire une fonction récursive qui incrémente un nombre représenté de cette façon. La fonction devra altérer la liste représentant le nombre.

Exercice 9.

Ecrire une fonction récursive qui inverse une liste simplement liée d'entiers. La fonction devra donc altérer la liste donnée.

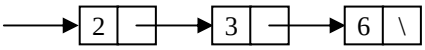
Exercice 10.

On peut transformer une liste d'entiers strictement positifs simplement liée non vide pour en faire une liste croissante d'entiers distincts entre 1 et le maximum de cette liste en procédant de la façon suivante.

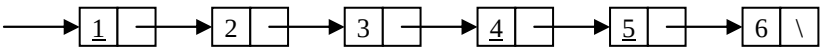
1. Si un entier manque dans la liste à une position donnée, on crée un nœud contenant l'entier et on le place dans la liste à cette position.

Exemple.

Si la liste de départ est



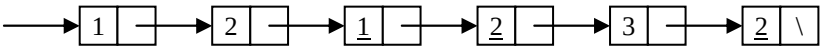
On ajoute des nœuds contenant 1, 4 et 5 pour obtenir la liste ci-dessous où les valeurs des nœuds ajoutés sont soulignées.



2. Si un entier n'apparaît pas dans l'ordre croissant par rapport aux éléments qui le précèdent, on le supprime de la liste.

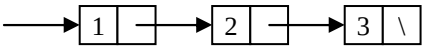
Exemple.

Si la liste de départ est

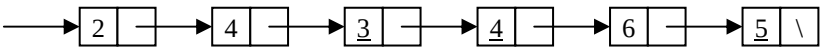


On supprime les nœuds contenant 1 et 2 soulignés dans la figure ci-dessus pour obtenir la liste ci-dessous.

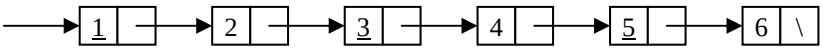
On peut combiner ajouts et suppressions.



Exemple.



On ajoutera un nœud contenant 1 au début de la liste, et un nœud contenant 3 après le nœud contenant 2. On supprimera les nœuds contenant 3 et 4 (soulignés dans la figure ci-dessus) et on ajoutera un nœud contenant 5 avant le nœud contenant 6. On supprimera le nœud contenant 5 (souligné) en fin de liste pour obtenir finalement la liste suivante :

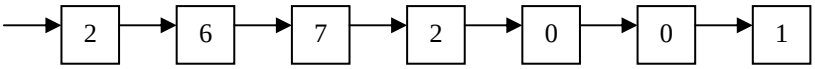


où les valeurs soulignées correspondent aux nœuds ajoutés.

On demande d'écrire une fonction qui reçoit comme paramètre une liste simplement liée non vide d'entiers strictement positifs et la transforme pour en faire une liste croissante d'entiers entre 1 et le maximum de cette liste en ajoutant et/ou supprimant des nœuds de la façon indiquée ci-dessus. La fonction n'effectuera qu'un seul parcours de la liste et n'ajoutera ou ne supprimera des nœuds que dans les circonstances décrites ci-dessus. La fonction ne pourra en aucun cas modifier la valeur d'un nœud de la liste.

Exercice 11.

On peut représenter un entier positif à l'aide d'une liste simplement liée d'entiers compris entre 0 et 9 constituée des chiffres de l'entier placé dans l'ordre inverse de l'écriture décimale. Le nombre 1002762 pourra être représenté par la liste suivante.

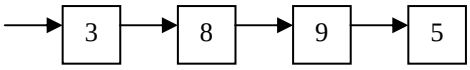


On supposera qu'une telle liste ne peut jamais se terminer par un 0. En particulier, le nombre 0 sera représenté par la liste vide.

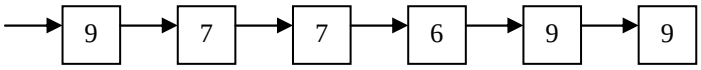
On demande d'écrire une fonction qui reçoit comme paramètres deux listes représentant des entiers comme décrit ci-dessus. La fonction devra soustraire le second entier du premier. On supposera que le second entier est toujours inférieur au premier. Au retour de la fonction, la première liste représentera un nombre qui correspond au résultat de la soustraction. La fonction doit donc modifier la première liste et pas créer une nouvelle liste.

Exemple.

Si le premier nombre est 1002762 représenté par la liste de l'exemple ci-dessus et le second nombre 5983 représenté par la liste suivante,



la fonction doit modifier la première liste pour qu'elle devienne :



qui correspond au nombre 996779 = 1002762 – 5983.

Attention, la première liste ne peut pas se terminer par un ou plusieurs 0, en particulier si les deux nombres sont les mêmes, au retour de la fonction, la première liste doit être vide.

La fonction demandée ne pourra créer aucun nouveau nœud. Elle pourra modifier ou supprimer des nœuds existants. La seconde liste ne sera pas modifiée par la fonction.

Exercice 12.

Ecrire une fonction récursive qui applique à tous les éléments d'une liste simplement liée d'entiers une fonction donnée.

En utilisant cette fonction, réaliser les opérations suivantes:

- affichage des nombres de la liste,
- affichage les nombres positifs de la liste,
- remplacement des nombres par leur valeur absolue,
- calcul de la somme des nombres de la liste (utiliser une variable globale pour la somme),
- calcul du produit des nombres de la liste (utiliser une variable globale pour le produit).

Exercice 13.

Définir une classe permettant de représenter des ensembles d'entiers. Les éléments d'un ensemble seront stockés dans la classe sous la forme d'une liste simplement liée croissante d'entiers. On demande d'implémenter, outre le constructeur de copie et le destructeur, les méthode suivantes:

- un constructeur sans paramètre qui crée un ensemble vide,
- une méthode permettant d'ajouter un entier à l'ensemble,
- une méthode permettant de supprimer un entier de l'ensemble,
- une méthode testant l'appartenance d'un entier à l'ensemble,
- une méthode qui donne le nombre d'éléments de l'ensemble,
- une méthode qui retourne l'ensemble obtenu en faisant l'intersection de l'ensemble avec un autre,
- une méthode qui retourne l'ensemble obtenu en faisant l'union de l'ensemble avec un autre,
- Une méthode qui retourne un tableau dynamique contenant les éléments de l'ensemble.

Les méthodes réalisant l'intersection et l'union doivent créer un nouvel ensemble et non modifier l'ensemble auquel elles sont appliquées.

Exercice 14.

Un polynôme peut être représenté par une liste dont les éléments contiennent le degré d'un terme (entier positif) et son coefficient (float). On supposera en outre que les éléments sont placés dans la liste par ordre croissant de leur degré. Les termes de coefficient nul ne sont pas représentés dans la liste.
En utilisant cette représentation, écrire une fonction qui retourne la somme de deux polynômes passés en paramètres.

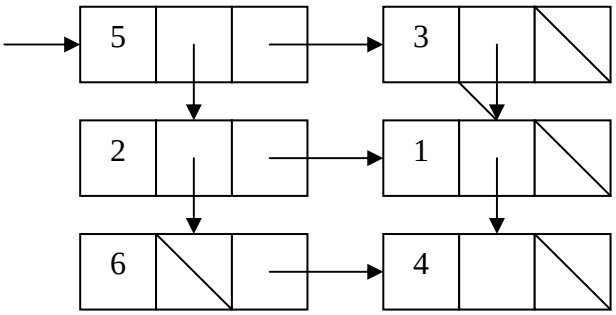
Exercice 15.

Une matrice peut être représentée par une structure dynamique dont chaque nœud représente un élément de la matrice, chaque nœud étant pourvu de deux pointeurs: le premier pointant vers le nœud correspondant à l'élément suivant sur la même ligne, le second pointant vers l'élément suivant dans la même colonne.

Par exemple, la matrice

$$\begin{pmatrix} 5 & 3 \\ 2 & 1 \\ 6 & 4 \end{pmatrix}$$

sera représentée par la structure dynamique suivante



En utilisant cette représentation, écrire une fonction récursive qui transpose une matrice. La fonction devra altérer la matrice qui lui est passée en paramètre.

Exercice 16.

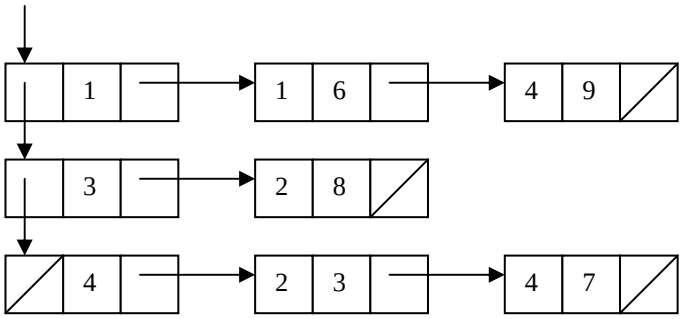
On appelle *matrice creuse* une matrice (de grande taille) dont la plupart des éléments ont la même valeur appelée la valeur par défaut. On peut implémenter de manière efficace du point de vue de l'utilisation de la mémoire une matrice creuse à l'aide d'une liste dont chaque élément contient un numéro de ligne et un pointeur vers la liste des valeurs différentes de la valeur par défaut de cette ligne associées à leur numéro de colonne. Les lignes qui ne contiennent que des valeurs par défaut ne sont pas représentées dans la liste des lignes. Celle-ci est ordonnée par rapport au numéro de ligne. De la même façon, la liste qui représente une ligne est ordonnée par rapport au numéro de colonne.

Exemple.

Soit la matrice entière

$$\begin{pmatrix} 6 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 3 & 0 & 7 \end{pmatrix}$$

Cette matrice peut être représentée par la structure dynamique suivante.



On demande d'écrire une classe *MatriceCreuse* permettant de représenter des matrice d'entiers en utilisant la structure décrite ci-dessus. La classe devra disposer d'un constructeur à trois paramètres entiers donnant les dimensions de la matrice et la valeur par défaut. Tous les éléments de la matrice construite par ce constructeur seront égaux à la valeur par défaut. La classe disposera de méthodes permettant de réaliser les opérations les plus élémentaires que l'on peut effectuer sur les matrices.

Exercice 17.

Soit la classe suivante qui permet de représenter des polynômes à coefficients réels (type C++ double).

```
class Polynome
{
  public:
    Polynome (double *c, int n);  // Construit un polynôme dont les coefficients sont dans le
                                  // tableau pointé par "c" de taille "n".
    Polynome (const Polynome &p);
    ~Polynome ();
    double val (double x);       // Retourne la valeur en "x" du polynôme.
  private:
    ...
}
```

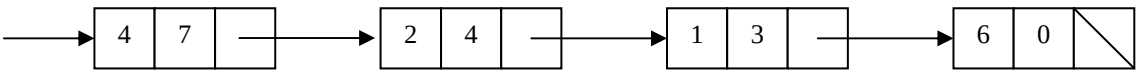
On impose la représentation (partie private) suivante : un polynôme sera représenté par une liste simplement liée de nœuds contenant les couples (coefficient, degré) des termes du polynôme pour lesquels le coefficient est non nul. Cette liste sera ordonnée par ordre décroissant du degré.

Exemple.

Le polynôme

$$4x^7 + 2x^4 + x^3 + 6$$

sera représenté par la liste



En particulier, le polynôme nul sera représenté par une liste vide.

On demande :

- de compléter la partie `private` de la classe `Polynome` (en définissant la/les classe(s) auxilliaire(s) nécessaire(s) pour la représentation imposée)
- de donner le code des constructeurs, du destructeur et de la méthode `val`.

Remarques.

1. Les coefficients du polynôme passés en paramètres au constructeur `Polynome (double *c, int n)` via le tableau de taille n pointé par c seront rangés par ordre croissant de degré. Ce tableau contiendra tous les coefficients du degré 0 au degré n – 1 et pourra se terminer par des 0.
Exemple:

Le tableau

6	0	0	1	2	0	0	4	0	0

représentera le polynôme

$$0x^9 + 0x^8 + 4x^7 + 0x^6 + 0x^5 + 2x^4 + x^3 + 0x^2 + 0x + 6$$

qui est identique au polynôme de l'exemple précédent.

2. On veillera à optimiser le calcul des puissances de x dans la méthode `val`.

Exercice 18.

Considérons des permutations sur l'ensemble des entiers de 1 à *n*.

Un cycle d'une telle permutation est la suite de nombres obtenue en partant d'un nombre et en regardant ses images successives par la permutation jusqu'à ce qu'on retombe sur ce nombre.

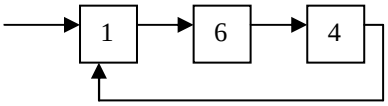
Exemple .

Soit la permutation suivante de l'ensemble des entiers de 1 à 6.

Entier	1	2	3	4	5	6
Image	6	5	3	1	2	4

La suite d'entiers (1 6 4) est un cycle de la permutation de même que la suite (5 2). La suite constituée du seul nombre 3 est également un cycle de la permutation.

On peut représenter un cycle à l'aide d'une liste circulaire simplement liée. Le cycle 1 6 4 peut être représenté par la liste suivante :



En mathématique on représente souvent les permutations en donnant une liste de cycles telle que tout élément apparaît dans un et un seul cycle de la liste. La permutation de l'exemple sera représentée par la notation

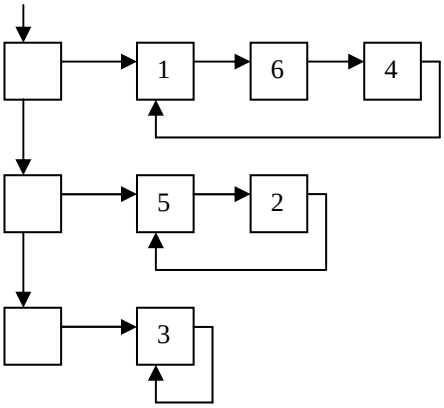
(1 6 4) (5 2) (3)

La représentation d'une permutation à l'aide d'une liste de cycles n'est pas unique. La liste de cycles

(2 5) (6 4 1) (3)

est une autre représentation de la permutation de l'exemple.

Une liste de cycles peut être représentée par une liste de listes circulaires comme l'illustre la figure ci-dessous pour la liste de cycles (1 6 4) (5 2) (3) :



On demande de décrire les différents types qui permettent de représenter une liste de cycles comme décrit ci-dessus.

En utilisant cette représentation, on demande d'écrire une fonction qui reçoit comme paramètre une liste de cycles d'une permutation et un entier x et qui retourne l'image de cet entier par la permutation. Pour la permutation de l'exemple, si $x = 2$, la fonction doit retourner 5. On suppose que l'entier x appartient à un des cycles de la liste.

Exercice 19.

Une matrice est dite *creuse* lorsque la plupart des ses éléments ont la même valeur appelée la valeur par défaut. Pour réaliser des économies de mémoire lorsqu'on travaille avec des matrices creuses, on opte pour une représentation dans laquelle on ne mémorise que les éléments dont la valeur diffère de la valeur par défaut. On peut, par exemple, utiliser une classe dont les objets représentent une matrice et contiennent ses dimensions, la valeur par défaut et un tableau de pointeurs dont l'élément i contiendra la tête d'une liste contenant les paires (colonne, valeur) des éléments de la ligne i de la matrice dont la valeur est différente de la valeur par défaut. Cette liste sera triée par ordre croissant des numéros de colonne. On supposera que les lignes et les colonnes sont numérotées à partir de 0.

Cette représentation nécessitera donc la définition de deux classes: la classe "Matrice" proprement dite et la classe "Nœud" permettant de représenter les nœuds des listes.

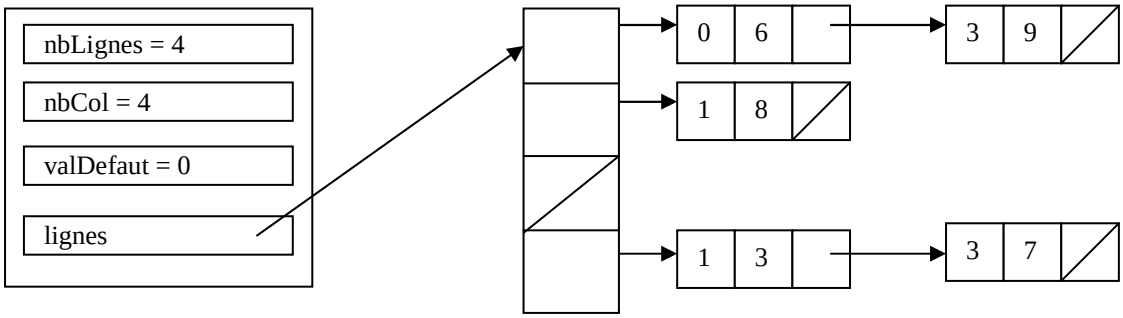
```
class Noeud
{
  public:
    int col;
    double val;
    Noeud *svt;
    Noeud(int c, double v, Noeud *s = NULL)
      :col(c), val(v), svt(s) {};
};

class Matrice
{
  public:
    Matrice(int n, int m, double vd);  //Construit une matrice n X m dont
                                       //tous les éléments ont la valeur
                                       //par défaut vd.
    Matrice(const Matrice &m);
    ~Matrice();
    int obtenirNbLignes();  //Retourne le nombre de lignes.
    int obtenirNbCol();     //Retourne le nombre de colonnes.
    void fixerVal(int i, int j, double v);  //Attribue la valeur v à
                                            //l'élément (i, j).
    double obtenirVal(int i, int j);  //Retourne la valeur de l'élément (i, j).
    Matrice *transposee();  //Retourne un pointeur vers la transposée
                            //de la matrice.
  private:
    int nbLignes, nbCol;  //Nombre de lignes et de colonnes de la matrice.
    double valDefaut;     //Valeur par défaut.
    Noeud **lignes;       //Tableau des pointeurs de tête des listes
                          // représentant les lignes.
};
```

Exemple.

La matrice suivante (dont la valeur par défaut est supposée être 0) :

$$\begin{pmatrix} 6 & 0 & 0 & 9 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 7 \end{pmatrix}$$

sera représentée par la structure ci-dessous :



On vous demande d'écrire le code complet du constructeur `Matrice(int n, int m, double vd)` et de la méthode `transposee`. Attention la méthode `transposee` doit créer une nouvelle matrice et retourner un pointeur vers celle-ci. Elle ne doit pas modifier la matrice d'origine. Le code des autres méthodes ne doit pas être donné sauf si on les utilise dans la méthode `transposee`.

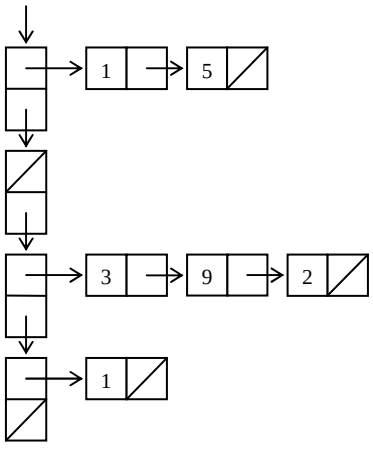
Suggestion: pour un code efficace et concis : parcourir la matrice de départ de la dernière ligne à la première.

Exercice 20.

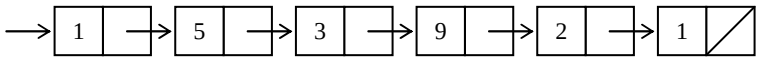
Ecrire une fonction *BoutABout* qui reçoit comme paramètre *une liste de listes d'entiers* et retourne *la liste d'entiers* obtenue en mettant bout à bout chacune de ces listes.
La fonction ne doit pas créer de nouveaux nœuds mais récupérer les nœuds des listes d'entiers. La fonction doit également supprimer (*delete*) les nœuds de la liste de listes passée en paramètre.

Exemple.

Si la fonction reçoit comme paramètre la liste de listes suivante :



la fonction devra retourner le pointeur de tête de la liste ci-dessous :



Remarque.

On demande de définir tous les types utilisés par la fonction, en particulier, le type permettant de représenter les nœuds d'une liste de listes d'entiers et le type permettant de représenter les nœuds d'une liste d'entiers.

Exercice 21.

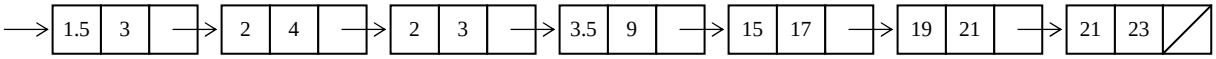
Ecrire une fonction qui reçoit comme paramètre un ensemble d'intervalles fermés sous la forme d'une liste simplement liée dont les nœuds contiennent les bornes inférieure et supérieure des intervalles (de type *double*). Cette liste sera ordonnée par ordre croissant de la borne inférieure. Si plusieurs intervalles consécutifs se chevauchent, la fonction les remplacera dans la liste par un seul de sorte qu'au retour de la fonction tous les intervalles seront disjoints.

Exemple.

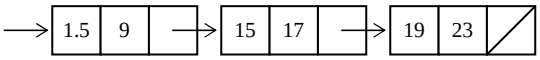
Soit l'ensemble d'intervalles :

{ [1.5, 3] ; [2, 4] ; [2, 3] ; [3.5, 9] ; [15, 17] ; [19, 21] ; [21, 23] }

représenté par la liste :



Au retour de la fonction la liste contiendra les valeurs suivantes :



Remarques.

- La fonction doit modifier la liste qui lui est passée comme argument.
- La fonction ne doit pas créer de nouveaux nœuds mais elle doit éventuellement en effacer.
- On demande de définir le type permettant de représenter les nœuds de la liste.

Exercice 22.

Une table est une structure de données contenant un ensemble de couples (clé, valeur) telle que deux couples ne peuvent avoir la même valeur de clé. Nous considérerons ici des tables dont les clés et les valeurs sont des entiers.

Les opérations de base que l'on peut effectuer sur une table sont les suivantes :

Ajouter un couple (clé, valeur) à une table.
Supprimer d'une table un couple dont la clé est donnée.
Indiquer si un couple de la table contient une clé donnée et retourner la valeur correspondante.

En C++, on peut représenter une table à l'aide d'une classe de la façon suivante.

```
class Table
{
  public:
    Table();
    Table(const Table &t);
    ~Table();
    void ajouter(int cle, int valeur);
    void supprimer(int cle);
    bool chercher(int cle, int &valeur);
  private:
    …
}
```

Remarques.

- Si on tente d'ajouter un couple dont la clé existe déjà dans la table le couple n'est pas ajouté mais la valeur du couple existant est remplacé par la valeur du couple à ajouter.
- Si aucun couple de la table ne contient la clé à supprimer, la table n'est pas modifiée.
- Si la table contient la clé indiquée, la méthode `chercher` retourne `true` et le paramètre `valeur` reçoit la valeur correspondant à la clé. Sinon la méthode retourne `false`.
- On demande de donner la partie privée de la classe `Table` et d'écrire le code de toutes ses méthodes, y compris les constructeurs et destructeur, en imposant comme représentation une liste simplement liée de couples (clé, valeur) triés par ordre croissant de la clé.

Exercice 23.

Ecrire une fonction booléenne qui reçoit comme paramètres :

un ensemble d'intervalles fermés sous la forme d'une liste simplement liée dont les nœuds contiennent les bornes inférieure et supérieure des intervalles (de type `double`) Cette liste est ordonnée par ordre croissant de la borne inférieure.
deux doubles qui sont les bornes d'un intervalle fermé.

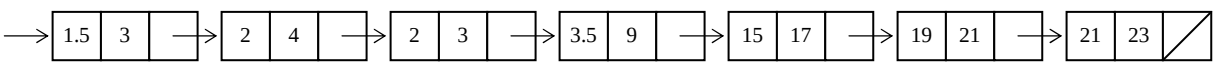
Cette fonction devra retourner `true` si et seulement si l'intervalle défini par les deux doubles est couvert par les intervalles de la liste c-à-d si cet intervalle est inclus dans la réunion des intervalles de la liste.

Exemple.

Soit l'ensemble d'intervalles :

{ [1.5, 3] ; [2, 4] ; [2, 3] ; [3.5, 9] ; [15, 17] ; [19, 21] ; [21, 23] }

représenté par la liste :



L'intervalle [2, 7] est couvert par l'ensemble mais l'intervalle [16, 20] ne l'est pas.

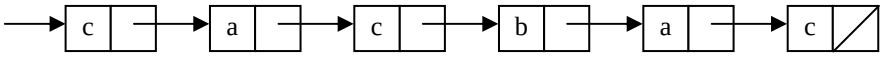
Exercice 24.

Ecrire une fonction qui reçoit comme paramètre une liste simplement liée dont les noeuds contiennent des caractères et retourne le pointeur de tête d'une liste simplement liée dont les noeuds contiennent les caractères de la liste passée en paramètre ainsi que leur nombre d'apparitions dans celle-ci. Cette liste devra être triée en ordre croissant des caractères.

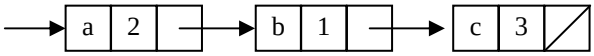
L'utilisation d'un tableau de travail est interdite.

Exemple.

Si la fonction reçoit comme paramètre la liste suivante.



La fonction retournera le pointeur de tête de la liste suivante.

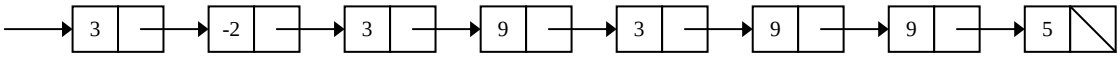


Exercice 25.

Ecrire une fonction qui reçoit comme paramètre une liste simplement liée d'entiers (non vide) et retourne un pointeur vers le nœud contenant la dernière occurrence de la plus grande valeur de la liste.

Exemple.

Si la fonction reçoit la liste suivante.



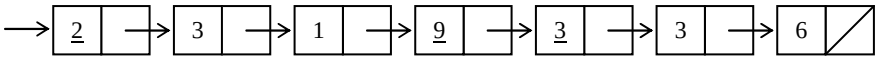
Elle devra retourner un pointeur vers le 7ème nœud de la liste qui contient la dernière occurrence de la valeur 9 qui est la plus grande valeur de la liste.

Exercice 26.

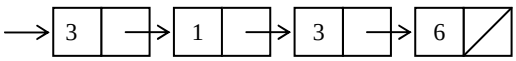
Ecrire une fonction qui reçoit comme paramètres une liste simplement liée d'entiers (non vide) et un entier. Cette fonction devra supprimer (`delete`) de la liste les nœuds immédiatement suivis par un nœud contenant cet entier.

Exemple.

Si la fonction reçoit comme paramètres la liste



et l'entier 3, la fonction supprimera de la liste les noeuds dont la valeur est soulignée dans la figure ci-dessus, de sorte qu'au retour de la fonction la liste aura la forme suivante.



Les piles.

Exercice 1.

Ecrire une fonction non récursive qui affiche les éléments d'une liste d'entiers dans l'ordre inverse de leur apparition dans la liste sans modifier la liste initiale mais en utilisant une pile.

Exercice 2.

Ecrire un programme qui simule le fonctionnement d'une calculatrice en notation polonaise inverse ou notation postfixe. Dans une telle calculatrice, les opérateurs sont entrés après les nombres, par exemple pour réaliser l'opération ((20+10)*4) on introduit dans la calculatrice

20 10 + 4 *

On supposera que la calculatrice ne peut traiter que des nombres entiers et les opérateurs "+", "-", "*" et "/" (division entière). L'utilisateur n'introduira que des entiers positifs. Les nombres et les opérateurs sont lus au clavier. On suppose que l'utilisateur presse "enter" après chaque nombre ou opérateur et qu'il ne tape pas d'espace.

Pour réaliser une telle calculatrice, il suffit d'utiliser une pile sur laquelle on place les nombres que l'utilisateur introduit. Quand l'utilisateur introduit un opérateur, on enlève les deux nombres au sommet de la pile et on les remplace par le résultat de l'opération appliquée à ces deux nombres. On demande qu'un message d'erreur soit affiché si l'utilisateur introduit un opérateur alors qu'il n'y a pas deux nombres sur la pile. On demande que le nombre au sommet de la pile soit affiché à l'écran si l'utilisateur entre le signe "=" (0 sera affiché s'il n'y a aucun nombre au sommet de la pile). Si l'utilisateur entre le caractère "v", la pile sera vidée pour réinitialiser la calculatrice.

Exercice 3.

Ecrire une fonction non récursive qui convertit une expression de la notation infixe (notation habituelle où l'opérateur et placé entre les opérandes) vers la notation postfixe. Les expressions considérées ne feront intervenir que les opérateurs "+", "-", "*","/" et des variables représentées par des lettres minuscules. Les expressions infixes seront supposées entièrement parenthésées.

Exemples1

Notation infixe	Notation postfixe
((a+b)*(a-d))	ab+ad-*
(a+(b*(a-d)))	abad-*+
(a+((b*a)-d))	aba*d-+

Pour réaliser cette conversion, on peut utiliser une pile de caractères. On parcours l'expression infixe de gauche à droite. Quand on rencontre une variable, on l'écrit dans l'expression résultat. Quand on rencontre un opérateur, on l'empile sur la pile. Quand on rencontre une parenthèse fermante, on enlève l'opérateur au sommet de la pile et on l'écrit dans l'expression résultat. On ne fait rien si on rencontre une parenthèse ouvrante.

Exercice 4.

Dans une expression en notation postfixe (ou notation polonaise inverse NPI), les opérateurs sont placés après les opérandes rendant les parenthèses superflues. On demande d'écrire une fonction qui retourne la valeur d'une expression en NPI passée en paramètre dans une chaîne de caractères. Les opérandes de l'expression seront des variables de type int représentées par une lettre minuscule, la fonction recevra donc également en paramètre un tableau contenant les valeurs des variables de l'expression: l'élément 0 contenant la valeur de a, l'élément 0 celle de b, …
On considère que l'expression ne fait intervenir que les opérateurs +, -, * et / (division entière) et qu'elle est syntaxiquement correcte.

Exemple.

La valeur de l'expression NPI ab+ca*- avec a=2, b=7 et c=3 vaut 3.

Pour évaluer une expression en NPI, il faut lire l'expression de gauche à droite. Quand on rencontre une variable, on empile sa valeur sur une pile initialement vide. Quand on rencontre un opérateur, on enlève les deux valeurs au sommet de la pile, on leur applique l'opérateur et on empile le résultat. A la fin du parcours de l'expression, il ne reste qu'une valeur dans la pile qui est la valeur de l'expression.

Exercice 5.

Ecrire une fonction qui évalue une expression en notation infixe passée en paramètre dans une chaîne de caractères et ne faisant intervenir que les opérateurs +, -, * et / et des variables entières représentées par des lettres minuscules. On supposera que toutes les sous-expressions de l'expression sont toujours placées entre parenthèses. La fonction recevra comme paramètre supplémentaire un tableau contenant les valeurs des variables de l'expression: l'élément 0 contenant la valeur de a, l'élément 0 celle de b, …

Pour évaluer une telle expression, il faut utiliser deux piles: une pile de valeurs et une pile d'opérateurs (on peut coder les opérateurs +=1, -=2, *=3, /=4 pour pouvoir utiliser une pile d'entiers). On parcourt l'expression de gauche à droite. Quand on rencontre une variable on empile sa valeur sur la pile des valeurs. Quand on rencontre un opérateur, on l'empile sur la pile des opérateurs. Quand on recontre une parenthèse fermante, on enlève les deux valeurs au sommet de la pile des valeurs et l'opérateur au sommet de la pile des opérateurs. On applique l'opérateur aux deux valeurs et on empile le résultat sur la pile des valeurs. Une fois l'expression parcourue, sa valeur est l'unique élément de la pile.

Les files.

Exercice 1.

On demande d'écrire une fonction qui affiche l'évolution du nombre de personnes dans une file d'attente à un guichet. Les personne sont supposées arriver dans la file à des instants donnés dans une liste passée en paramètre à la fonction, les personnes attendent dans la file avant d'être servies. Quand c'est leur tour, les personnes occupent le guichetier pendant un temps qui est également passé en paramètre via la liste. Une fois servies, les personnes quittent la file.

On suppose que les temps d'arrivée et les durées de service sont des entiers. La fonction recevra en paramètre une liste de couples: instant d'arrivée, durée de service qui sera triée par ordre croissant d'instant d'arrivée.

La fonction affichera à l'écran l'évolution de la file en affichant les instants où ont lieu les changements dans la file, la nature du changement: arrivée ou départ ainsi que le nombre de personnes dans la file.

Si un départ a lieu en même temps qu'une arrivée, deux lignes seront affichées correspondant au même instant.
On pourra également supposer que plusieurs arrivées se produisent simultanément, une ligne sera alors affichée par arrivée.

Exemple:

Soit la liste d'arrivée suivante:

Instants d'arrivée	Durées de service
5	2
6	3
10	5
12	3
12	1
17	1
25	3

La fonction devra afficher les lignes suivantes:

```
instant 5 arrivée longueur file 1
instant 6 arrivée longueur file 2
instant 7 départ longueur file 1
instant 10 arrivée longueur file 2
instant 10 départ longeur file 1
instant 12 arrivée longeur file 2
instant 12 arrivée longeur file 3
instant 15 départ longeur file 2
instant 17 arrivée longeur file 3
instant 18 départ longeur file 2
instant 19 départ longeur file 1
instant 20 départ longeur file 0
instant 25 arrivée longeur file 1
instant 28 départ longeur file 0
```

Exercice 2

Dans cet exercice on suppose données les deux classes suivantes (on ne doit donc pas donner les définitions des différentes méthodes).

1) la classe Stack représente une pile de couples d’entiers :

```
class Stack
{
  public:
    Stack();
    Stack(const Stack &s);
    ~Stack();
    bool empty();
    bool push(int x, int y);
    bool pop(int &x, int &y);
    bool top(int &x, int &y) ;
  private:
    ...
};
```

2) la classe Queue représente une file de pointeurs vers des objets de la classe Stack décrite ci-dessus :

```
class Queue
{
  public:
    Queue();
    Queue(const Queue &q);
    ~Queue();
    bool empty();
    bool insert(Stack *p);
    bool get(Stack *&p);
  private:
    ...
};
```

On peut supposer que le destructeur de la classe Queue appelle le destructeur de la classe Stack pour tous les objets Stack pointés par les pointeurs de la file.

Un labyrinthe peut être représenté par une matrice dynamique de booléens, les "cases" à *true* représentant les murs du labyrinthe. Les éventuelles sorties du labyrinthe sont les cases sur ses côtés qui contiennent *false*.

On peut trouver le plus court chemin qui mène à une sortie du labyrinthe à partir d'une position initiale en utilisant l'algorithme suivant.

On utilise une variable de la classe *Queue* décrite ci-dessus, appelons-la *q*.

1 On ajoute à *q* un pointeur vers une pile (classe *Stack*) dans laquelle on a empilé au préalable les coordonnées de la position initiale.
2 Tant que *q* n'est pas vide et tant qu'on a pas trouvé la sortie on effectue les opérations suivantes.
- On extrait le pointeur au début de *q*, appelons-le *p*.
- On récupère, sans les enlever (méthode *top*), les coordonnées *x* et *y* au sommet de la pile pointée par *p*.
- Si l'élément (*x, y*) de la matrice représentant le labyrinthe contient *false*,
  - ◊ si de plus il est sur un bord, alors on a trouvé la sortie
  - ◊ sinon
    - ∗ on place la valeur *true* dans cette case
    - ∗ puis pour chacune des quatre directions, on effectue les opérations suivantes.
      (a) On se déplace dans cette direction à partir de la position *(x, y)*, appelons les nouvelles coordonnées (*x', y'*).
      (b) On fait pointer un pointeur vers une copie de la pile pointée par *p* (n'oubliez pas que la classe *Stack* dispose d'un constructeur de copie).
      (c) On empile sur cette copie les coordonnées (*x', y'*).
      (d) On ajoute le pointeur vers cette copie à la file pointée par *q*.
    - ∗ On supprime la pile pointée par *p* et on attribue à *p* la valeur NULL.

- Sinon (si l'élément (*x, y*) contient *true*) on supprime la pile pointée par *p* et on attribue à *p* la valeur NULL.

3 Si *p* n'est pas NULL alors il pointe vers une pile qui contient un parmi les chemins les plus courts de la position initiale à une sortie. La position initiale est au fond de cette pile et la position de la sortie au sommet. Si *p* est NULL, il n'y a pas de chemin qui mène de la position initiale à la sortie.

On demande d'écrire une fonction qui reçoit comme paramètres un pointeur vers une matrice dynamique de booléens, ses dimensions, les coordonnées de la position initiale et met en œuvre l'algorithme décrit ci-dessus pour retourner un pointeur vers une pile contenant un chemin le plus court possible de la position initiale vers la sortie ou NULL si un tel chemin n'existe pas.

Exercice 3.

Ecrire une fonction qui inverse une file d'entiers qui lui est passée comme paramètre.

On demande de ne pas utiliser de tableau ou de liste de travail pour effectuer l'inversion mais d'utiliser plutôt une pile. Il existe en effet une méthode très simple pour inverser un file en utilisant une pile.

On suppose disponibles les classes Queue et Stack dont les parties publiques sont données ci-dessous et pour lesquelles vous ne devez donner ni la partie privée ni le code des méthodes.

```
class Queue
{
  public:
    Queue();
    Queue(const Queue &q);
    ~Queue();
    bool empty();
    void insert(int n);
    int get();
};

class Stack
{
  public:
    Stack();
    Stack(const Stack &s);
    ~Stack();
    bool empty();
    void push(int n);
    int pop();
};
```

Les arbres.

Exercice 1.

Ecrire une fonction qui affiche les éléments d'un arbre binaire contenant des caractères.

Exercice 2.

Ecrire une fonction qui compte le nombre d'occurrences d'un caractère donné dans un arbre dont les nœuds contiennent des caractères.

Exercice 3.

Ecrire une fonction qui construit l'image miroir d'un arbre dont chaque nœud contient un caractère.
Par exemple, l'arbre de droite ci-dessous est l'image miroir de l'arbre de gauche.

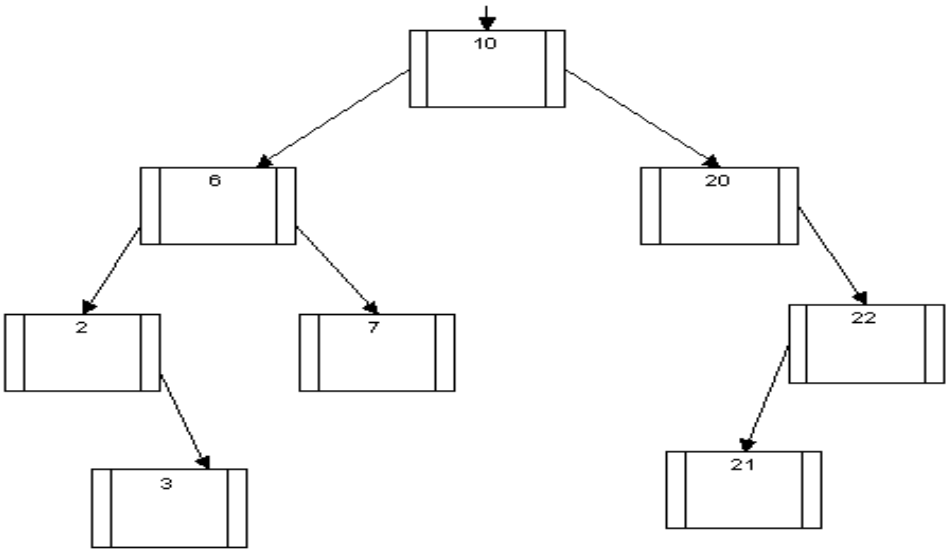
Exercice 4.

Ecrire une fonction booléenne qui vérifie si deux arbres contenant des entiers sont égaux, c'est-à-dire ont la même structure et contiennent les mêmes entiers aux mêmes positions.

Exercice 5.

Ecrire une fonction booléenne qui reçoit comme paramètres un arbre dont les nœuds contiennent des entiers et une chaîne de caractères composée uniquement de lettre "G" et "D". Cette chaîne représente un chemin descendant dans l'arbre à partir de la racine. Le caractère "G" indique qu'on doit descendre par le fils gauche d'un noeud, le caractère "D" par le fils droit. La fonction retournera *true* si le chemin correspond bien à un nœud de l'arbre *false* sinon. Si la fonction retourne *true*, elle retournera via un paramètre supplémentaire la valeur du nœud indiqué par le chemin.

Exemple.



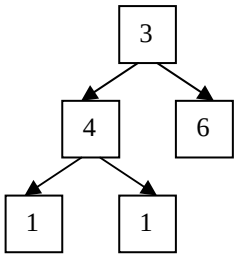
la chaîne "GD" indique le chemin qui mène au nœud contenant la valeur 7,
la chaîne "DDG" indique le chemin qui mène au nœud contenant la valeur 21,
la chaîne "" (chaîne vide) indique le chemin qui mène à la racine, soit le nœud contenant la valeur 10,
la chaîne "DL" ne correspond à aucun nœud de l'arbre.

Exercice 6.

Un arbre binaire dont les nœuds contiennent des entiers est dit équilibré en poids si, pour chacun de ses nœuds, soit la somme des valeurs dans son sous-arbre gauche est égale à la somme des valeurs dans son sous-arbre droit soit ses deux sous-arbres sont vides.

Exemple.

L'arbre binaire ci-dessous est équilibré en poids : en effet, les sommes des valeurs dans les 2 sous-arbres du nœud contenant 4 sont égales et valent 1 et les sommes des valeurs dans les deux sous-arbres du nœud contenant 3 sont égales et valent 6. Les autres nœuds de l'arbre n'ont pas de sous-arbre.



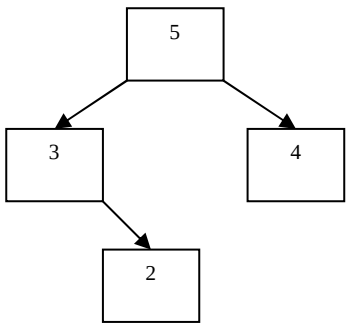
On demande d'écrire une fonction à valeur booléenne recevant comme paramètre un pointeur vers le nœud racine d'un arbre binaire d'entiers et qui indique si cet arbre est équilibré en poids.

Exercice 7.

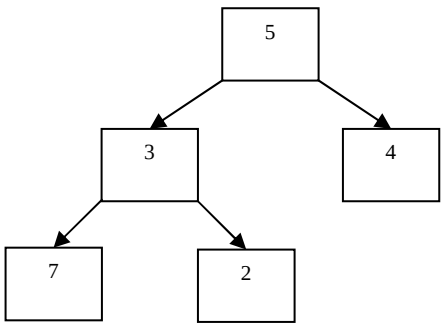
Ecrire une fonction à valeur booléenne qui indique si l'arbre binaire contenant des entiers qui lui est passé en paramètre vérifie la condition suivante : tout nœud de l'arbre est soit une feuille, soit a deux sous-arbres non vides. La fonction devra renvoyer `true` si la condition est vérifiée et `false` sinon.

Exemple.

a1



a2



La fonction retournera la valeur `false` pour l'arbre *a1* et la valeur `true` pour l'arbre *a2*.

Exercice 8.

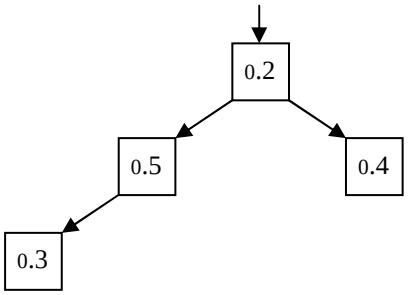
Soient des arbres binaires dont les nœuds contiennent des float compris entre 0 et 1. On interprétera ces valeurs comme représentant la probabilité de descendre par le fils gauche du nœud. La probabilité de descendre par le fils droit vaut donc 1 moins la valeur dans le nœud de l'arbre.

On demande d'écrire une fonction qui reçoit comme paramètre un tel arbre binaire et lui ajoute des nœuds de la façon suivante. Pour chaque nœud de l'arbre de départ qui n'a pas de fils gauche et/ou pas de fils droit, on ajoute un fils gauche et/ou fils droit. Dans ce nouveau nœud on place la probabilité d'arriver jusqu'à lui en partant de la

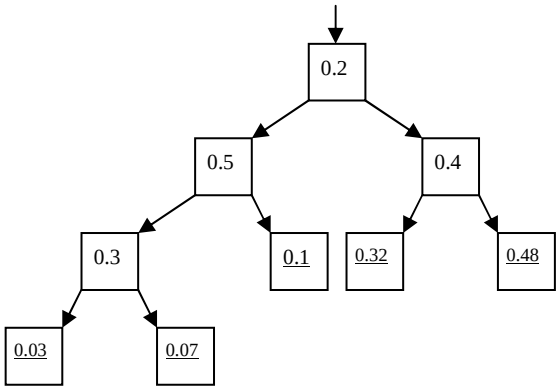
racine de l'arbre et en descendant par la gauche ou la droite des nœuds selon les probabilités indiquées dans les nœuds.

Exemple.

Soit l'arbre suivant passé en paramètre à la fonction :



La fonction ajoutera à l'arbre les nœuds suivants (dont les valeurs sont soulignées dans la figure ci-dessous) :



La valeur 0.07, par exemple, est obtenue en effectuant 0.2 x 0.5 x 0.7 car on arrive au nœud qui la contient en descendant par la gauche à partir du nœud racine, puis encore par la gauche du nœud contenant 0.5 et enfin par la droite du nœud contenant 0.3 (avec la probabilité 1 - 0.3 = 0.7).

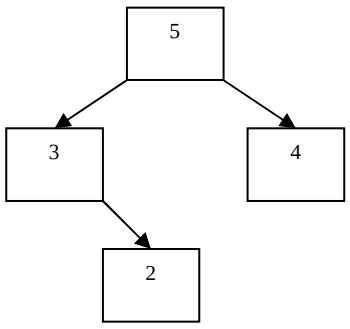
Exercice 9.

Ecrire une fonction qui ajoute aux valeurs des nœuds d'un arbre binaire *a1* contenant des entiers les valeurs des nœuds d'un autre arbre binaire *a2* contenant des entiers de la façon suivante. Pour tout nœud de *a1* qui possède un correspondant dans *a2* (c-à-d un nœud qui occupe la même position dans l'arbre) on ajoute la valeur du nœud correspondant de *a2* à la valeur du nœud de *a1*. Si un nœud de *a1* n'a pas de correspondant dans *a2* sa valeur reste inchangée. Pour tout nœud de *a2* qui n'a pas de correspondant dans *a1*, on ajoute un nœud dans *a1* à la même position et contenant la même valeur.

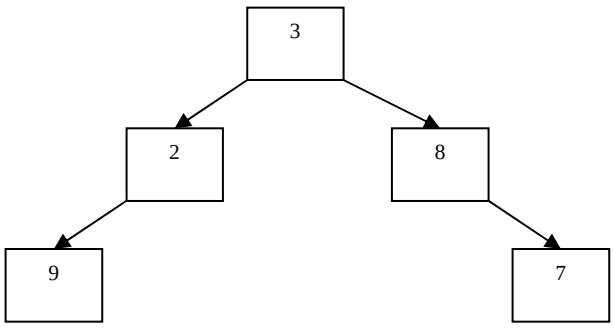
Exemple.

Soient *a1* et *a2* les arbres suivants :

a1

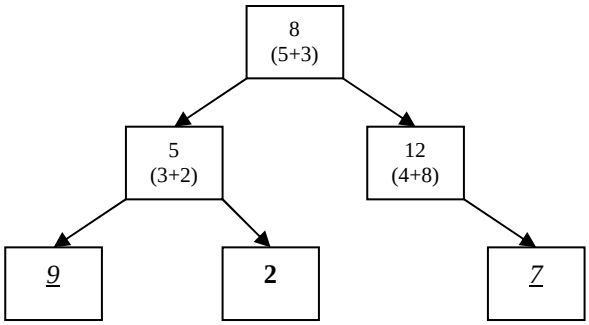


a2



Au retour de la fonction, l'arbre *a1* contiendra les valeurs suivantes :

a1



On a indiqué entre parenthèses les sommes de valeurs de nœuds de *a1* et *a2*, en **gras**, les valeurs des nœuds de *a1* qui n'ont pas été modifiées et en *italique souligné* les valeurs des nœuds ajoutés à *a1*.

Remarques.

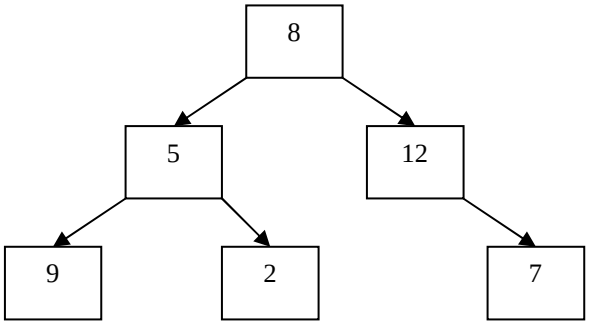
L'arbre *a2* ne peut pas être modifié par la fonction.

Les nœuds ajoutés à *a1* doivent être alloués par la fonction et en aucun cas être des nœuds de l'arbre *a2*.

Exercice 10.

Dans le parcours par niveaux d'un arbre binaire, comme son nom l'indique, on parcourt les nœuds niveau par niveau en descendant dans l'arbre.

Exemple.



Si on affiche les valeurs des nœuds de l'arbre binaire ci-dessus lors d'un parcours par niveaux de la gauche vers la droite, on obtient :

```
8 5 12 9 2 7
```

La façon la plus simple de réaliser un tel parcours est d'utiliser une file de pointeurs vers le type représentant les nœuds de l'arbre (on pourra supposer ici que les nœuds contiennent des entiers). On commence par ajouter à la file initialement vide le pointeur vers le nœud racine de l'arbre puis tant que la file n'est pas vide, on enlève un pointeur de la file et s'il n'est pas NULL, on affiche la valeur du nœud pointé par ce pointeur puis on ajoute à la file les pointeurs vers ses fils gauche et droit.

On pourra supposer qu'on dispose de la classe file (*Queue*) suivante dont on ne doit pas donner la définition des méthodes :

```
class Queue
{
  public:
    Queue();
```

```
    Queue(const Queue &q);
    ~Queue();
    bool empty();
    void insert(Noeud *p);
    bool get(Noeud *&p);
};
```

Remarques.

La méthode *get* retournera *false* si la file est vide. Sinon le pointeur passé comme argument recevra la valeur du pointeur au début de la file et celui-ci sera supprimé de la file.

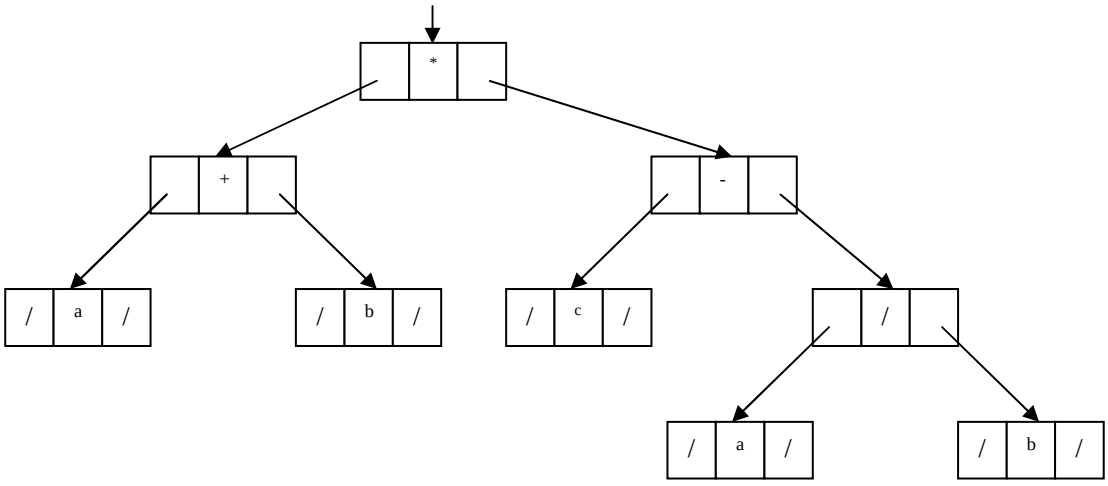
On demande d'écrire une fonction qui affiche les valeur d'un arbre binaire contenant des entiers dans l'ordre du parcours par niveaux.

Exercice 11.

Une expression arithmétique ne contenant que des variables représentées par des lettres minuscules et les opérateurs "+", "-", "*", "/" peut être représentée par un arbre binaire dont les nœuds contiennent un caractère. Une expression constituée d'une variable sera représentée par un arbre d'un seul nœud contenant la variable. Une expression composée sera représentée par un arbre dont la racine contient l'opérateur et dont les sous-arbres de gauche et droite contiennent respectivement les représentations des opérandes gauche et droit de l'expression.

Exemple:

L'expression ((a+b)*(c-(a/b))) peut être représentée par l'arbre



Ecrire une fonction qui reçoit comme paramètre un arbre représentant une expression et retourne une chaîne de caractères contenant l'expression en notation infixe.

Ecrire une fonction qui reçoit comme paramètre un arbre représentant une expression et retourne une chaîne de caractères contenant l'expression en notation postfixe.

Ecrire une fonction qui reçoit une expression en notation infixe dans une chaîne de caractères et retourne l'arbre représentant l'expression.

Ecrire une fonction qui reçoit comme paramètre un arbre représentant une expression et un tableau de 26 floats dont l'élément 0 contient la valeur de la variable "a", l'élément 1 la valeur de "b" … Cette fonction devra retourner la valeur de l'expression.

Exercice 12.

On peut représenter un arbre dont les nœuds contiennent des caractères par une chaîne de caractères de la façon suivante. Si l'arbre est vide la chaîne contiendra uniquement une parenthèse ouvrante et une parenthèse

fermante: `"()"`. Si *sg* et *sd* sont respectivement les chaînes représentant les sous arbres de gauche et de droite d'un arbre dont la racine contient le caractère *c*, la chaîne représentant l'arbre est obtenue en plaçant entre parenthèses le caractère *c* suivi des chaînes *sg* et *sd*.
Par exemple, la chaîne `(a(b(d()())(e()()))(c(f()())()))` est la représentation de l'arbre ci-dessous.

Ecrire une fonction qui construit la représentation sous forme de chaîne d'un arbre passé en paramètre.

Ecrire une fonction qui construit un arbre à partir de sa représentation donnée sous forme d'une chaîne de caractères.

Exercice 13.

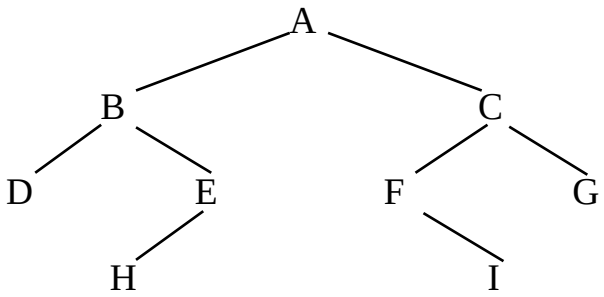
Considérons la structure *Node* suivante qui représente les nœuds d'un arbre binaire contenant des caractères.

```
struct Node
{
  char data;
  Node *left, *right;
  Node(char d, Node *l = NULL, Node *r = NULL)
    :data(d), left(l), right(r) {}
};
```

La fonction *affiche* suivante affiche les valeurs des nœuds d'un arbre binaire dont le pointeur vers le nœud racine lui est passé en paramètre en effectuant un parcours en ordre préfixe de cet arbre et en affichant le caractère '*' lorsque son paramètre vaut NULL.

```
void affiche(Node *t)
{
  if(t == NULL)
    cout << '*';
  else
  {
    cout << t->data;
    affiche(t->left);
    affiche(t->right);
  }
}
```

Pour l'arbre suivant :



la fonction affichera les caractères suivants à l'écran :

```
ABD**EH***CF*I**G**
```

Sous la condition que les nœuds ne peuvent pas contenir le caractère '*', la fonction affichera des chaînes de caractères différentes pour deux arbres différents. Il est donc possible de reconstruire l'arbre si on connaît la suite des caractères affichés par la fonction.

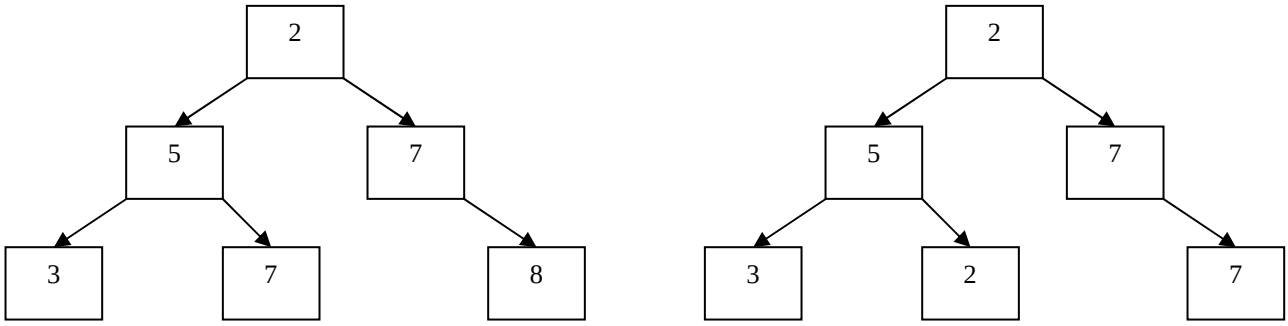
On demande d'écrire une fonction qui reçoit comme paramètre une chaîne de caractères (qui pourrait être obtenue en affichant le contenu d'un arbre par la fonction *affiche*) et construit et retourne l'arbre binaire qui correspond à cette chaîne. On supposera que cet arbre ne contient pas le caractère '*'.

Si la fonction reçoit, par exemple, la chaîne de caractères `"ABD**EH***CF*I**G**"` elle construira et retournera l'arbre de l'exemple ci-dessus.

Exercice 14.

Ecrire une fonction booléenne qui indique si un arbre binaire contenant des entiers possède un chemin qui descend de la racine jusqu'à une feuille tel que les valeurs des noeuds apparaissent en ordre strictement croissant.

Exemple.



La fonction devra retourner *true* pour l'arbre de gauche car le long du chemin 2-7-8, par exemple, les valeurs apparaissent en ordre strictement croissant.

La fonction retournera *false* pour l'arbre de droite car pour aucun des chemins de la racine à une feuille les entiers n'apparaissent en ordre strictement croissant. Ces chemins sont 2-5-3, 2-5-2 et 2-7-7.

On pourra supposer que l'arbre n'est pas vide.

Exercice 15.

Ecrire une fonction qui reçoit comme paramètre un arbre binaire dont les nœuds contiennent le pointeur de tête d'une liste simplement liée d'entiers. Cette fonction devra supprimer (delete) les nœuds feuilles de cet arbre et supprimer (delete) les nœuds des listes dont les pointeurs de tête sont dans ces feuilles.

Exercice 16.

Ecrire une fonction qui reçoit deux paramètres. Le premier sera un arbre binaire de recherche non vide contenant des entiers tous distincts et le second un entier. La fonction modifiera l'arbre de la façon suivante.

- Si l'entier passé comme second paramètre n'est pas présent dans l'arbre, la fonction ajoutera à l'arbre une feuille contenant cet entier. L'arbre résultant de cet ajout devra être un arbre binaire de recherche.

- Si l'entier passé comme second paramètre est présent dans l'arbre et si le noeud qui le contient est une feuille, alors la fonction supprimera cette feuille.

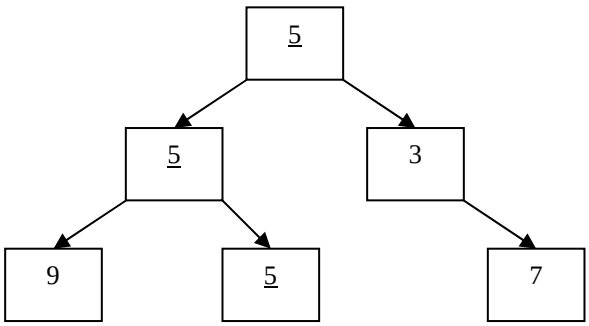
- Si l'entier passé comme second paramètre est présent dans l'arbre et si le nœud qui le contient n'est pas une feuille alors l'arbre ne sera pas modifié.

Exercice 17.

Ecrire une fonction booléenne qui reçoit comme paramètre un arbre binaire non vide contenant des entiers et retourne `true` si et seulement s'il existe dans l'arbre au moins un chemin descendant de la racine à une feuille le long duquel tous les noeuds contiennent la même valeur.

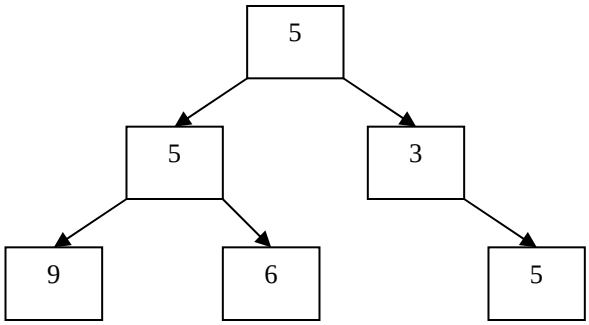
Exemples.

Pour l'arbre suivant



la fonction retournera `true` car il existe un chemin descendant de la racine à une feuille le long duquel on ne rencontre que la valeur 5. Les valeurs des noeuds d'un tel chemin sont soulignées dans la figure ci-dessus.

Pour l'arbre suivant



la fonction retournera `false` car il n'y a pas de chemin descendant de la racine à une feuille pour lequel les valeurs contenues dans les noeuds sont identiques. Il y a trois chemins de la racine à une feuille et les valeurs des noeuds le long de ces trois chemins sont 5-5-9, 5-5-6 et 5-3-5.

Les arbres binaires de recherche.

Exercice 1.

Dessiner tous les arbres binaires de recherche contenant les nombres entiers de 0 à 5.

Exercice 2.

Ecrire deux fonctions qui affichent les entiers contenus dans un arbre binaire de recherche, l'une en ordre croissant, l'autre en ordre décroissant.

Exercice 3.

Ecrire une fonction qui crée une liste ordonnée d'entiers contenant les valeurs d'un arbre binaire de recherche.

Exercice 4.

Ecrire une fonction booléenne qui teste si un arbre binaire contenant des entiers est un arbre binaire de recherche. On suppose que l'arbre passé à la fonction ne contient que des entiers positifs distincts.

Exercice 5.

Ecrire une fonction booléenne qui teste si un arbre binaire contenant des entiers est un arbre binaire de recherche sans faire l'hypothèse que les entiers sont positifs et distincts.

Exercice 6.

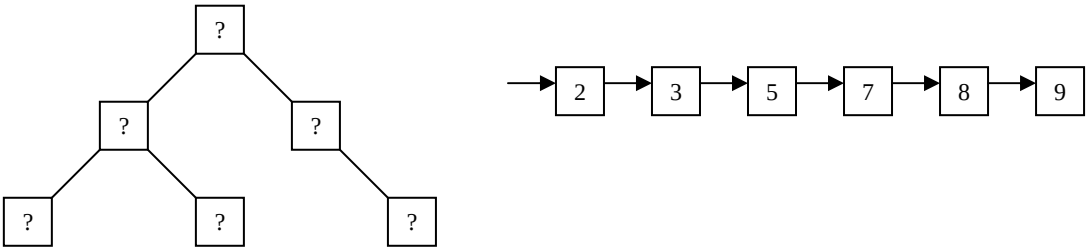
Écrire une fonction récursive qui, étant donné un arbre binaire de recherche et deux valeurs *min* et *max*, affiche dans l'ordre toutes les valeurs des nœuds de l'arbre qui sont comprises entre *min* et *max*.

Exercice 7.

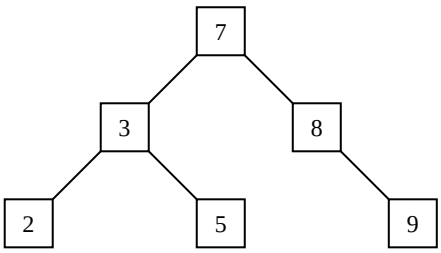
Ecrire une fonction qui remplit un arbre binaire avec des entiers contenus dans une liste strictement croissante simplement liée. La fonction reçoit comme paramètres un pointeur vers le nœud racine d'un arbre binaire contenant des entiers et le pointeur de tête de la liste d'entiers. On suppose que la liste contient autant d'éléments qu'il y a de nœuds dans l'arbre. Les valeurs présentes dans l'arbre lors de l'appel de la fonction sont sans importance. La fonction doit placer dans l'arbre les valeurs de la liste de sorte qu'au retour de la fonction l'arbre soit un arbre binaire de recherche. La fonction ne doit pas modifier la liste d'entiers.

Exemple.

Soient l'arbre et la liste passés comme paramètres à la fonction :



Au retour de la fonction l'arbre devra contenir les valeurs suivantes :



Exercice 8.

Un arbre est presque en équilibre si le nombres de nœuds dans les sous-arbres gauche et droit de chacun de ses nœuds ne diffèrent pas de plus de 1. Ecrire une fonction qui crée un arbre binaire de recherche presque en équilibre et contenant les entiers 1, 2, …, *n* où *n* est le paramètre de la fonction.

Exercice 9.

Les ADT « arbre binaire de recherche » et « liste doublement liée circulaire» sont bien deux concepts fondamentalement différents. Toutefois les deux ADT peuvent partager une structure de données commune. En effet, chaque élément possède deux pointeurs vers les éléments auxquels il est relié. Dans le cas de la liste doublement liée, on a un pointeur vers le prédécesseur et un autre vers le successeur. Et pour l'arbre binaire de recherche, on a un pointeur vers le fils droit et un autre vers le fils gauche. De plus nous devons disposer d'un pointeur dans un cas vers la tête de liste et dans l'autre vers la racine de l'arbre.

Considérons les déclarations suivantes :

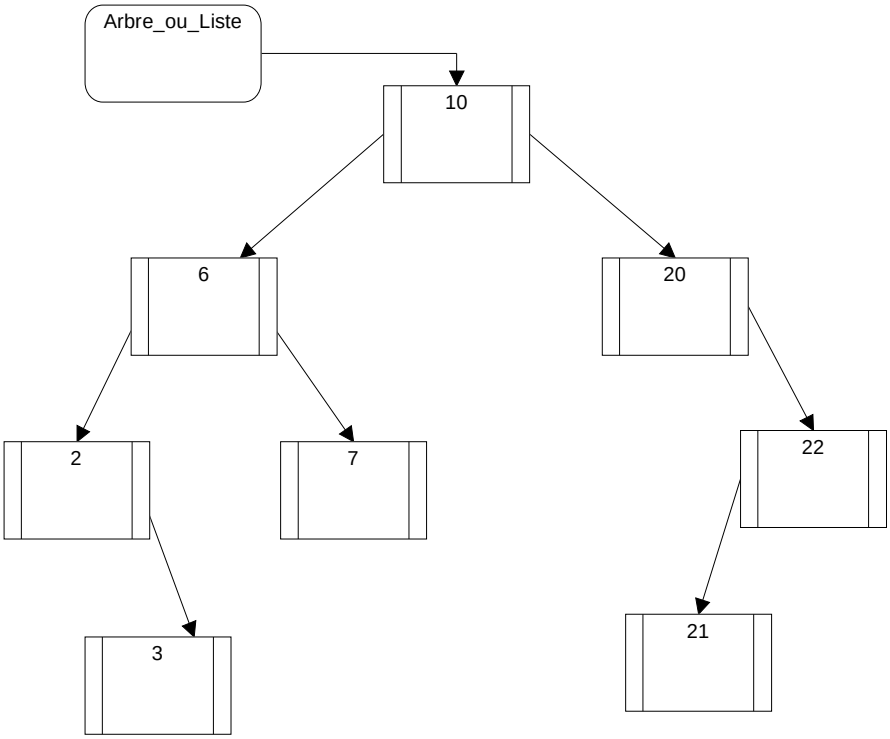
```
class Arbre_ou_Liste
{
public :
      Element *acces ;  // pointe vers la tête de liste ou la racine
      bool etat_arbre;  // true si l'objet représente un arbre
                        // false si l'objet représente une liste

      Arbre_ou_Liste(bool etat_init=true);
      ~Arbre_ou_Liste();
      bool Ajoute_dans_Arbre(int d);
      Transforme_en_Liste();
};

class Element
{
public :
      Element *ptr1;    // pointeur vers fils gauche ou vers suivant
      Element *ptr2;    // pointeur vers fils droit ou précédent
      int donnee;

      Element(int d=0);
};
```

Ecrivez la fonction membre récursive « Transforme_en_Liste » permettant de convertir l'arbre binaire de recherche initialement créé en une liste doublement liée circulaire triée. Remarquez bien qu'il s'agit de conversion et non de création. Les instances « d'Element » déjà utilisées pour l'arbre doivent être réutilisées pour constituer la liste, on ne fera donc pas de nouvelle création d'élément.

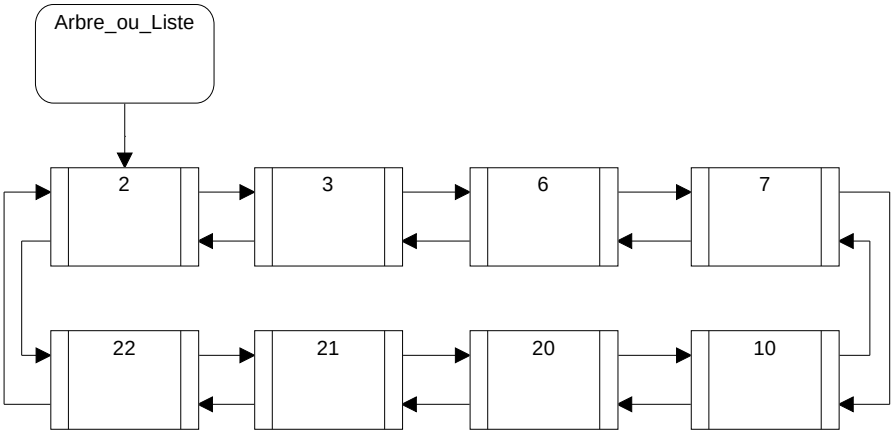
Les fonctions « Arbre_ou_Liste(bool) », « ~Arbre_ou_Liste() » et « bool Ajoute_dans_Arbre(int) » sont supposées connues et ne doivent pas être réécrites. Il vous est loisible d'ajouter des fonctions membres à l'une ou l'autre classe, mais dans ce cas vous devrez en donner le code complet et expliquer leur rôle.

Exemple.

L'arbre binaire de recherche suivant :



sera converti en la liste doublement liée circulaire suivante :

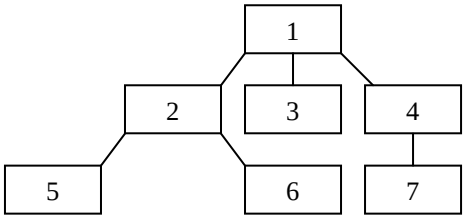


Les arbres n-aires.

Exercice 1.

Un arbre n-aire est un arbre dont les nœuds peuvent avoir zéro ou plusieurs fils.

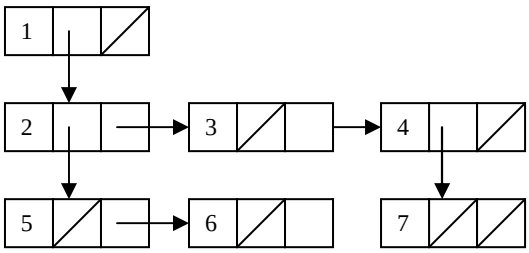
Exemple.



Pour représenter les nœuds d'un arbre n-aire en C++, on pourrait définir une classe dont les objets contiendraient : la valeur du nœud, un pointeur vers son premier fils, un pointeur vers son « frère » suivant.

```
class Noeud
{
  public:
    int val;
    Noeud *premFils;
    Noeud *frereSvt;
};
```

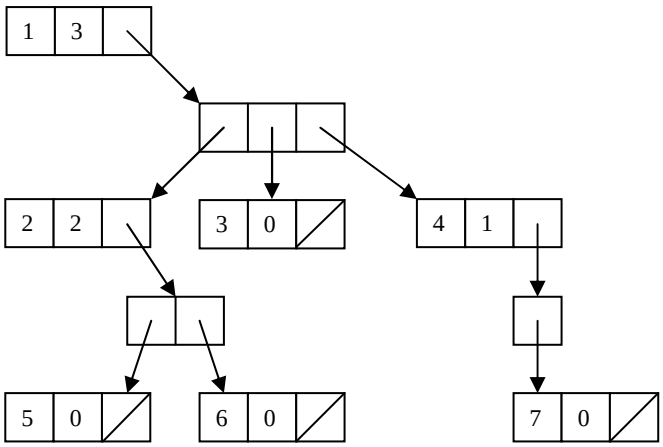
Exemple.



On pourrait opter pour une autre représentation où l'objet représentant un nœud contiendrait la valeur du nœud, le nombre de ses fils et un tableau dynamique de pointeurs vers ses fils.

```
class NoeudBis
{
  public:
    int val;
    int nbFils;
    NoeudBis **fils;
};
```

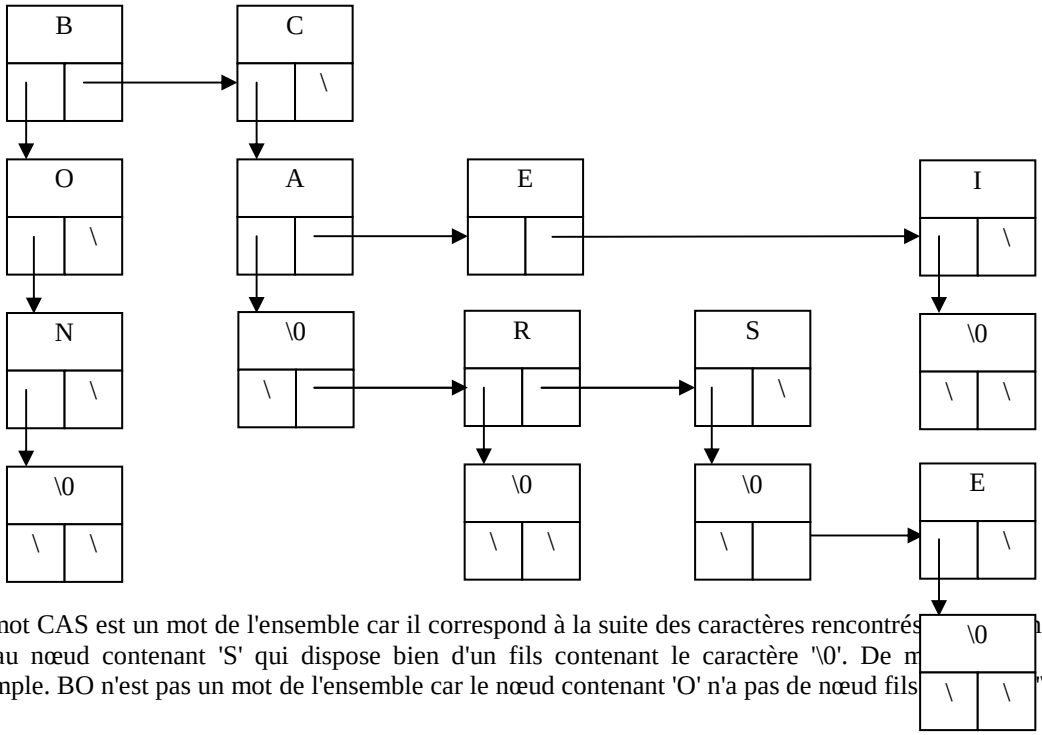
Exemple.



Ecrire deux fonctions qui effectuent la conversion d'une représentation vers l'autre.

Exercice 2.

Un ensemble de mots peut-être représenté à l'aide d'un arbre n-aire dont les nœuds contiennent un caractère. Un mot de l'ensemble sera obtenu par la suite des caractères rencontrés d'une racine à un nœud de l'arbre dont un nœud fils contient le caractère sentinelle '\0'. Soit par exemple l'ensemble de mots {BON, CA, CAR, CAS, CASE, CE, CI}, cet ensemble pourra être représenté par l'arbre suivant:



Le mot CAS est un mot de l'ensemble car il correspond à la suite des caractères rencontrés [illegible]ne contenant 'C' au nœud contenant 'S' qui dispose bien d'un fils contenant le caractère '\0'. De m[illegible] CASE, par exemple. BO n'est pas un mot de l'ensemble car le nœud contenant 'O' n'a pas de nœud fils [illegible] '\0'.

Dans l'exemple, les fils d'un nœud ont été rangés par ordre croissant du caractère qu'ils contiennent, cela permet d'optimiser la recherche d'un mot dans l'arbre.

On demande de définir une classe "Dictionnaire" permettant de représenter un ensemble de mot en utilisant la méthode vue ci-dessus. La classe devra contenir un constructeur sans paramètre permettant de créer un ensemble de mots vide, un constructeur de copie, un destructeur, un fonction permettant d'ajouter un mot à l'ensemble, une fonction qui indique si un mot appartient à l'ensemble et une fonction qui indique si une suite de caractères est le début d'un mot de l'ensemble (c-à-d si on peut compléter cette suite par un ou plusieurs caractères pour obtenir un mot de l'ensemble).

Ajouter à la classe "Dictionnaire" une méthode qui affiche les mots de l'ensemble qui correspondent à un mot donnée à une différence près. Deux mots ont une différence si une seule de leurs lettres diffèrent ou si un des mots a une lettre de plus que l'autre à une position quelconque.
Par exemple, si le mot passé en paramètre à la fonction est CAP et l'ensemble de mots celui de l'exemple plus haut, la méthode affichera les mots

CA	Lettre P en plus dans CAP
CAS	Dernière lettre différente

Avec la chaîne CAS, la méthode affichera

CA	Lettre S en plus dans CAS
CAR	Dernière lettre différente
CAS	Correspondance exacte
CASE	Lettre E en plus dans CASE

Ajouter à la classe "Dictionnaire" une méthode qui affiche le mot le plus proche d'un mot donné du point de vue du nombre de différences. Par exemple si le mot passé en paramètre à la fonction est BAS, le mot le plus proche de l'ensemble de l'exemple si dessus est CAS avec une seule différence, le mot BON a, par exemple, deux différences de même que les mots CA, CAR et CASE .