

Structures de données linéaires

I. Liste, Pile et file.

Une liste linéaire est la forme la plus simple et la plus courante d'organisation des données. On l'utilise pour stocker des données qui doivent être traitées de manière séquentielle.

Exemple

On désire, pour les besoins de la programmation d'un jeu de cartes, représenter le paquet de cartes. Parmi toutes les informations que l'on possède sur les cartes, la plupart sont inutiles (couleur du dos, taille, matière, propriétaire). Seuls nous intéressent, la couleur (trèfle, carreau, coeur ou pique), la hauteur (valeur de la carte), et le côté visible de la carte. Pour chacune des cartes ces informations utiles seront codées et rangées dans un article.

Ainsi on effectuera le codage suivant:

- pour le côté visible:
 - Face = 1 signifie que la carte est "cachée" (dos visible)
 - Face = 0 signifie que la carte est "visible" (face visible)
- pour la couleur:
 - couleur = 1 pour trèfle
 - couleur = 2 pour carreau
 - couleur = 3 pour coeur
 - couleur = 4 pour pique
- pour la hauteur:
 - hauteur = 1 pour l'as
 - hauteur = 2 pour le deux
 - |
 - hauteur = 12 pour la dame
 - hauteur = 13 pour le roi

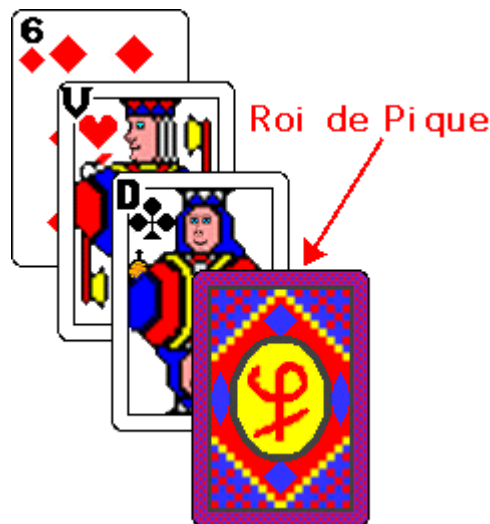
Un article aura alors la forme suivante :

Face	Couleur	Hauteur
------	---------	---------

De plus, on pourra éventuellement adjoindre à cet article une zone "suivant" servant à indiquer l'adresse de l'élément suivant. Ainsi on aura la représentation de l'article sous la forme :

Face	Couleur	Hauteur	Suivant
------	---------	---------	---------

Le paquet de quatre cartes suivant :



pourra alors être représenté par :



Définition

Une liste linéaire est une suite finie (éventuellement vide) d'articles de même type $X[1], X[2], \dots, X[N]$ avec $N > 0$, dont la propriété est qu'il est linéaire (une dimension), ordonné et :

si $N > 0$ alors $X[1]$ est le premier article

si $1 < k < N$ alors $X[k]$ est précédé de $X[k-1]$ et est suivi de $X[k+1]$.

Les articles sont repérés selon leur rang dans la liste mais il ne faut pas confondre le rang et la position!

Les opérations que l'on peut désirer réaliser sur une telle structure sont entre autres :

- examiner le k ème élément de la liste
- insérer un élément avant le k ème élément de la liste
- supprimer le k ème élément de la liste
- réunir deux listes
- rechercher les articles dont une zone contient une certaine valeur
- etc

Il existe différents type de listes linéaires. Un exemple tiré de la vie courante peut être le rangement de vêtements et son organisation comportemental.

Dans certains types de listes les opérations d'insertion et de suppression se déroulent toujours aux extrémités de la liste. Ces types de listes se rencontrant très fréquemment, on leur a donné un nom.

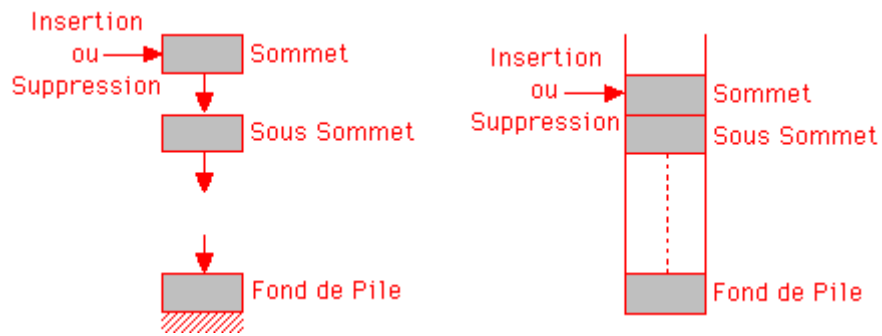
Définition

Une Pile est une structure de données linéaire dans laquelle les opérations d'insertion et de suppression se font à une extrémité appelée sommet. L'autre extrémité étant désignée par le terme "fond de pile". On rencontre également le terme LIFO (Last In First Out) pour désigner ce type de structure.

Exemples

- pile d'assiettes
- pile d'un microprocesseur
- une pile est utilisée pour le calcul des expressions postfixées

Représentations



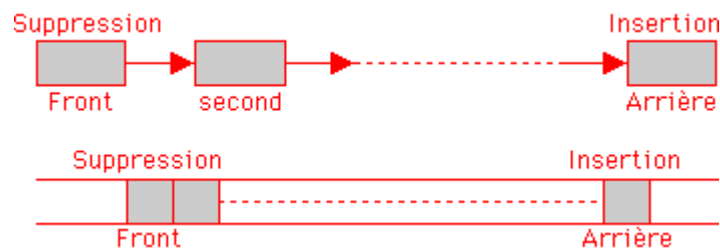
Définition

Une File est une structure de données linéaire dans laquelle les opérations d'insertion se font à une extrémité appelée arrière et les opérations de suppression se font à l'autre extrémité appelée front. On rencontre également le terme FIFO (Fist In Fist Out) pour désigner ce type de structure.

Exemples

- file de voitures
- de manière générale : Les files d'attentes

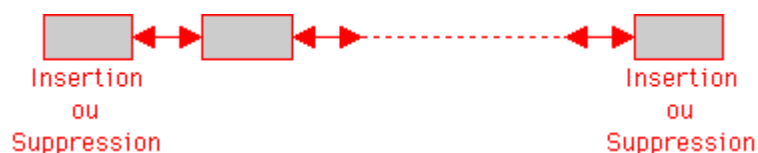
Représentations



Définition

Une queue ou double file (ou deque) est une structure de données linéaire dans laquelle les opérations d'insertion et de suppression se font aux deux extrémités.

Représentation



Remarque

On distingue également les Queues à entrée restreinte (QER) dans lesquelles on ne peut insérer qu'à une extrémité, mais sortir aux deux extrémités; et les Queues à sortie restreinte (QSR) dans lesquelles on ne peut supprimer qu'à une extrémité, mais insérer à l'une des deux extrémités.

II. Implantation en mémoire

Nous représenterons la mémoire comme une série de cases numérotées de 1 à M. M étant le nombre de cases mémoires disponible permettant le stockage des informations.



Ainsi, les articles $X[i]$ sont rangés dans ces cases. L'emplacement, l'adresse d'un article $X[i]$ est alors le numéro de la case où il débute.

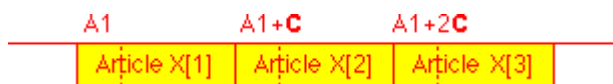
1. La Méthode séquentielle

C'est la façon la plus naturelle de procéder. En effet les articles $X[i]$ de la liste linéaire sont rangés, les uns à la suite des autres, dans des cases mémoires consécutives.

Ainsi, si chaque article a besoin de C cases mémoires pour son stockage, on aura:

Adresse de $X[j + 1] = \text{Adresse de } X[j] + C = A1 + j * C$ (où A1 est l'adresse de $X[1]$).

On peut donc, sans difficulté, accéder à un article quelconque de la liste linéaire, en connaissant l'adresse de base A1.

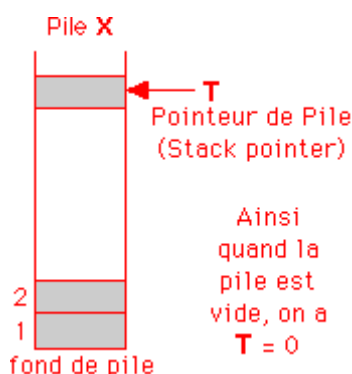


Par la suite nous supposons, pour plus de commodité, que $C = 1$ et $A1 = 1$.

Etudions, à présent les cas particuliers de la pile et de la file.

a. Cas d'une pile

Les opérations d'insertion et de suppression se font à une extrémité (sommet). Il faut donc connaître en permanence l'adresse de celle-ci. Pour cela, on utilisera un pointeur T qui nous indiquera où se trouve le sommet.



Primitives

Les opérations de base sur les piles sont simples :

empiler(p,e) : empile l'élément e dans la pile p.

dépiler(p) : dépile un élément de la pile p et renvoie cette valeur.

Les opérations d'insertion et de suppression pourront alors se réaliser avec les instructions suivantes :

Insertion de Y	Suppression
$T = T + 1;$ $X [T] = Y;$	$Y = X [T];$ $T = T - 1;$

Mais ici se pose deux problèmes:

- vouloir supprimer un élément alors que la pile est vide
- vouloir insérer un élément alors que la pile est pleine (les M cases mémoires réservées à la pile sont occupées).

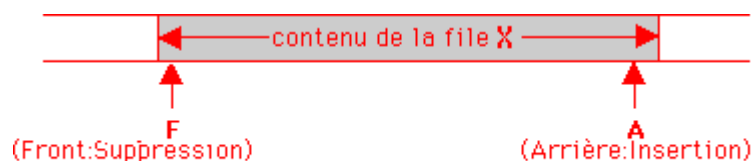
Il va donc falloir gérer ces exceptions et on écrira alors:

Insertion de Y	Suppression
<code>si (T == M) alors</code> <code> PLEIN();</code> <code>sinon</code> <code> {</code> <code> T = T + 1;</code> <code> X [T] = Y;</code> <code> }</code>	<code>si (T == 0) alors</code> <code> VIDE();</code> <code>sinon</code> <code> {</code> <code> Y = X [T];</code> <code> T = T - 1;</code> <code> }</code>

Où VIDE () et PLEIN () simulent les exceptions (appel à une procédure d'erreur, sortie du traitement, etc).

b. Cas d'une file

Les opérations d'insertion se font à une extrémité (arrière) et les opérations de suppression se font à l'autre extrémité (front). Il faut donc connaître en permanence l'adresse de celles-ci. Pour cela, on utilisera un pointeur F qui nous indiquera où se trouve le front et un pointeur A qui nous indiquera où se trouve l'arrière.



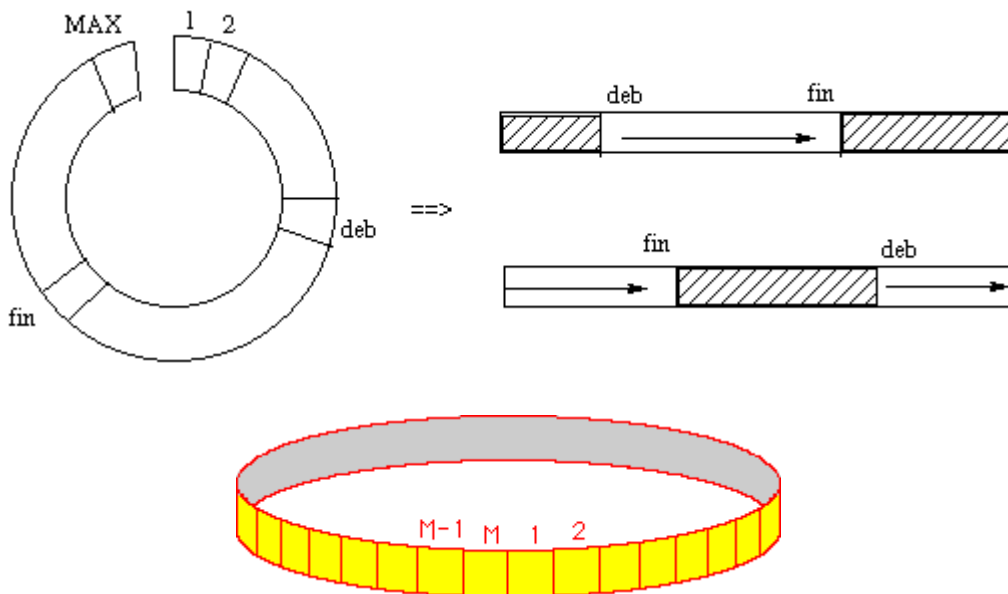
Les opérations d'insertion et de suppression pourront alors se réaliser avec les instructions suivantes :

Insertion de Y	Suppression
$A = A + 1;$ $X [A] = Y;$	$Y = X [F];$ $F = F + 1;$

A nouveau ici se pose deux problèmes:

- vouloir supprimer un élément alors que la file est vide
- vouloir insérer un élément alors que la file est pleine (les M cases mémoires réservées à la file sont occupées).

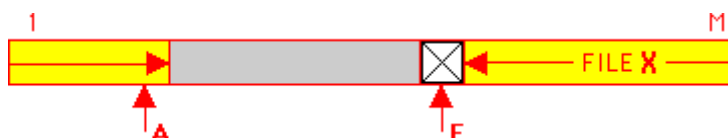
De plus, le fait que les deux pointeurs soient incrémentés lors de chaque opération signifie que l'on risque de se retrouver dans le cas où le début de mémoire est inoccupé, le pointeur A étant en fin de mémoire ($A = M$) et interdire ainsi les opérations d'insertions. Pour ne pas interrompre le traitement, dans un tel cas de figure, on va supposer la mémoire comme étant circulaire.



La file X pourra alors prendre la forme suivante:



Il reste encore un problème à résoudre. En effet, dans une telle configuration, les tests de file vide et de file pleine sont identiques ($A = F - 1$). Pour pouvoir différencier les deux cas, on va supposer que la case pointée par F est "interdite" pour le stockage d'un élément de la file.



Les opérations d'insertion et de suppression pourront alors se réaliser avec les instructions suivantes:

Insertion de Y	Suppression
<pre> si ((A == F-1) ou (A == M et F == 1)) PLEIN(); sinon { si (A == M) alors A = 1; sinon A = A + 1; X [A] = Y; } </pre>	<pre> si (A == F) alors VIDE(); sinon { si (F == M) alors F = 1; sinon F = F + 1; Y = X [F]; } </pre>

Le problème de la différenciation entre une liste pleine et une liste vide aurait pu aussi se résoudre en ajoutant une variable booléenne qui passerai de vrai à faux dès la première insertion d'un élément.

Remarquons aussi que l'on peut à tout moment connaître le nombre d'éléments de la file :

- Si $A > F$, il s'agit de $A - F$.
- Si $A \leq F$, il s'agit de $M - F + A$.

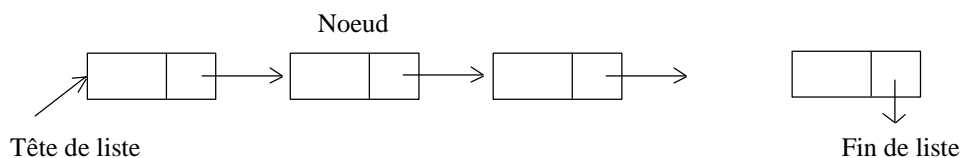
2. Les listes «chaînées»

Une autre approche traditionnelle pour implanter une liste est l'utilisation de pointeurs. De cette façon, la mémoire est allouée dynamiquement : la réservation de l'espace se fait en fonction des données insérées ou supprimées.

a. Listes simplement chaînées

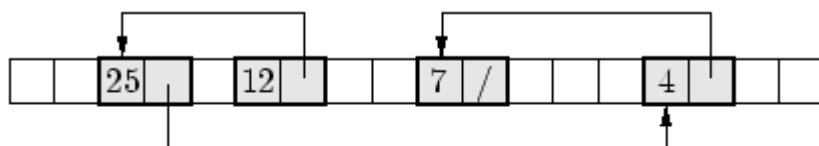
On ajoute à chaque article une information complémentaire qui pointe vers l'article suivant. L'ensemble sera appelé un maillon ou un noeud.

On peut représenter ce type de liste de la façon suivante :



La tête de liste est une variable qui va «pointer» le premier élément de la liste. Chaque article (noeud) est composé de son information et d'un pointeur vers le noeud suivant. Le dernier article n'ayant pas de suivant pointe vers vide, nil ou null représenté dans l'exemple suivant par /.

La linéarité de cette représentation est purement virtuelle. En effet, les éléments n'ont aucune raison d'être ordonnés ou contigus en mémoire.



Chaque élément est lié à son successeur. Il n'est donc pas possible d'accéder directement à un élément quelconque de la liste.

On peut simuler ce type de liste dans un tableau :

rang	valeur	suivant
0	4	1
1	7	/
2	12	3
3	25	0

Début de liste → ← Fin de chaînage

Mais en utilisant un tableau, on perd évidemment le côté dynamique de l'allocation de la mémoire.

Primitives

`teteDeListe(lst)` : permet d'accéder au pointeur désignant la tête de liste de la liste `lst`.

`valeur(p, lst)` ou plus simplement `valeur(p)` : permet d'accéder à la valeur du noeud pointé par `p` de la liste `lst`.

`suisvant(p, lst)` ou plus simplement `suisvant(p)` : permet d'accéder au suivant du noeud pointé par `p` de la liste `lst`.

Si on s'intéresse un peu plus à l'allocation mémoire, on peut aussi considérer des fonctions qui initialise une liste, crée un noeud pointé par `p` ou libère l'espace mémoire d'un noeud pointé par `p`.

Insertion d'un élément dans une liste chaînée

Il y a trois cas différents que nous devons traiter : l'insertion en début de liste, au sein de la liste et en fin de liste.

Insertion en début de liste :

```
p = creerNoeud();           // Ces deux lignes sont inutiles
valeur(p) = information;    // si le noeud existe déjà.
suisvant(p) = teteDeListe(lst);
teteDeListe(lst) = p;
```

Remarquons que cette suite d'instructions convient pour une insertion dans une liste vide.

Insertion après un noeud pointé par q :

```
p = creerNoeud();
valeur(p) = information;
suisvant(p) = suisvant(q);
suisvant(q) = p;
```

Insertion en fin de liste :

Il faut en premier lieu parcourir toute la liste et ensuite insérer après le dernier élément de celle-ci.

```
p = creerNoeud();
valeur(p) = information;
si (teteDeListe(lst) == NIL)
{
    suisvant(p) = NIL;
    teteDeListe(lst) = p;
}
sinon
{
    q = teteDeListe(lst);
    tant que (suisvant(q) != NIL)
        {
            q = suisvant(q);
        }
    suisvant(p) = NIL;
    suisvant(q) = p;
}
```

Suppression d'un élément dans une liste chaînée

Nous nous intéressons à la suppression du noeud pointé par `p` dans une liste `lst`. Le problème consiste à retrouver le prédécesseur de `p`.

```

si (teteDeListe(lst) == p)
    teteDeListe(lst) = suivant(p);
sinon
    {    tant que (suivant(q) != p)
        q = suivant(q);
        suivant(q) = suivant(p);
    }

```

Exercice

Trouver un algorithme qui donne le nombre d'éléments d'une liste chaînée `lst`.

Correction

```

si (teteDeListe(lst) == NIL)
    nbr = 0;
sinon
    {    q = teteDeListe(lst);
        nbr = 1;
        tant que (suivant(q) != NIL)
            {    nbr = nbr + 1;
                q = suivant(q);
            }
    }

```

Exercice

Trouver un algorithme donnant le maximum des valeurs d'une liste chaînée non vide d'entiers `lst`.

Correction

```

q = teteDeListe(lst);
max = valeur(q);
tant que (suivant(q) != NIL)
{    q = suivant(q);
    si (valeur(q) > max)
        max = valeur(q);
}

```

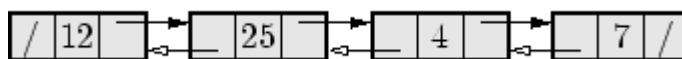
Exercices supplémentaires

- Ecrire une fonction qui indique si une valeur fait partie d'une liste.
- Ecrire une fonction qui renvoie la n -ième valeur d'une liste (traiter les cas où il n'y a pas de n -ième élément).
- Ecrire une fonction qui concatène deux listes `lst1` et `lst2`.
- Ecrire une fonction qui renverse une liste `lst` pour obtenir une liste `lstInverse`.
- Ecrire une fonction qui insère un élément `p` dans une liste `lst` (non vide) triée par ordre croissant.
- Ecrire une fonction qui fusionne deux listes non vides `lst1` et `lst2` triées pour obtenir de nouveau une liste triée `lst`.

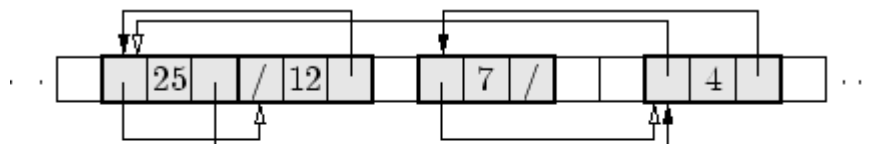
b. Listes doublement chaînées

Le parcours d'une liste chaînée se fait en sens unique. Cela complique bien des problèmes. Une solution apportée à ce parcours et de faire appel à une liste doublement chaînée. Ce type de liste est similaire à une liste chaînée à ceci près que l'on adjoint à chaque noeud une information qui est l'adresse de son prédécesseur.

Exemple



De nouveau, cette linéarité est purement virtuelle. Tout comme pour la liste chaînée, les noeuds ne sont pas nécessairement adjacents et/ou ordonnés en mémoire.



Primitives

En plus des primitives des listes simplement chaînées, c'est-à-dire : `teteDeListe(lst)`, `valeur(p)` et `suisvant(p)`, nous avons :

`finDeListe(lst)` qui permet d'accéder au pointeur désignant la fin de liste de la liste `lst`.
`precedent(p)` qui permet d'accéder au précédent du noeud pointé par `p`.

Insertion d'un élément dans une liste doublement chaînée

De même que pour les listes simplement chaînées, il y a trois cas différents que nous devons traiter : l'insertion en début de liste, au sein de la liste et en fin de liste. On gèrera en même temps les exceptions.

Insertion en début de liste :

```
p = creerNoeud(); // Ces deux lignes sont inutiles
valeur(p) = information; // si le noeud existe déjà.
si teteDeListe(lst) == NIL)
{
    finDeListe(lst) = p;
    suisvant(p) = NIL; /* idem a */
    precedent(p) = NIL; /* idem b */
    teteDeListe(lst) = p; /* idem c */
}
sinon
{
    precedent(teteDeListe(lst)) = p;
    suisvant(p) = teteDeListe(lst); /* idem a */
    precedent(p) = NIL; /* idem b */
    teteDeListe(lst) = p; /* idem c */
}
```

Remarquons que les lignes */* idem a */*, */* idem b */* et */* idem c */* des deux cas effectuent les mêmes opérations et peuvent donc être exclues du test et placées en fin d'instructions.

Insertion en fin de liste :

```
p = creerNoeud();
valeur(p) = information;
si (finDeListe(lst) == NIL)
    debutDeListe = p;
sinon
    suivant(finDeListe(lst)) = p;
precedent(p) = finDeListe(lst);
suivant(p) = NIL;
finDeListe(lst) = p;
```

Insertion après un noeud pointé par q :

```
si (q == finDeListe(lst))
    {    insérer en fin de liste
    }
sinon
    {    p = creerNoeud();
        valeur(p) = information;
        suivant(p) = suivant(q);
        precedent(p) = q;
        precedent(suivant(q)) = p;
        suivant(q) = p;
    }
```

Suppression d'un élément dans une liste doublement chaînée

```
si (teteDeListe(lst) == p)
    {    si (finDeListe(lst) == p)
        {    teteDeListe(lst) = NIL;
            finDeListe(lst) = NIL;
        }
        sinon
            {    teteDeListe(lst) = suivant(p);
                precedent(suivant(p)) = NIL;
            }
    }
sinon
    si (finDeListe(lst) == p)
        {    finDeListe(lst) = precedent(p);
            suivant(precedent(p)) = NIL;
        }
    sinon
        {    suivant(precedent(p)) = suivant(p);
            precedent(suivant(p)) = precedent(p);
        }
```

Exercice

Ecrire une fonction qui insère un noeud p dans une liste lst (non vide) triée par ordre croissant.

Correction

```
si (valeur(p) < valeur(teteDeListe(lst))
    insérer p en début de liste;
sinon
{   q = teteDeListe(lst);
    tant que (valeur(p) < valeur(q))
        q = suivant(q);
    si (q != NIL)
        insérer p après le precedent de q;
    sinon
        insérer p à la fin de la liste;
}
```

Exercice

Ecrire une fonction qui fusionne deux listes non vides `lst1` et `lst2` triées pour obtenir de nouveau une liste triée `lst`.

Remarque 1: on n'utilisera pas la fonction de l'exercice précédent.

Remarque 2 : c'est évidemment beaucoup plus simple en récursif en considérant qu'une liste est un premier élément et une liste.

Correction

```
p = teteDeListe(lst1);
q = teteDeListe(lst2);
teteDeListe(lst) = NIL;
tant que ((p!=NIL) && (q!=NIL))
{   si ((p!=NIL) && (q!=NIL))
        si (valeur(p) < valeur(q))
            {   insérer le noeud p à la fin de la liste lst
                p = suivant(p);
            }
        sinon
            {   insérer le noeud q à la fin de la liste lst
                q = suivant(q);
            }
    sinon
        si (p==NIL)
            {   insérer le noeud q à la fin de la liste lst
                q = suivant(q);
            }
        sinon
            {   insérer le noeud p à la fin de la liste lst
                p = suivant(p);
            }
}
```

c. Listes circulaires

Une liste circulaire peut être simplement chaînée ou doublement chaînée. Elle s'obtient en remplaçant le ou les pointeurs NIL par le pointeur de tête de liste et, dans le cas d'une liste doublement chaînée par le pointeur de début de liste.

On obtient une liste qui n'a ni premier ni dernier élément. Tous les noeuds sont accessibles à partir de n'importe quel autre.

III. Un exemple d'utilisation de pile : expressions arithmétiques

Les piles ont de nombreuses applications : les navigateurs web (précédent), les annulateurs d'actions dans un traitement de texte ou autres logiciels, les algorithmes de recherches en profondeur d'un arbre ou implicitement les algorithmes récursifs de certains langages.

Une des applications des piles nécessite de s'y attarder un peu. Il s'agit des expressions arithmétiques et de la méthode postfixée de calcul qui est utilisée par exemple dans certaines calculatrices HP.

Les expressions arithmétiques sont habituellement écrites de manière infixée, c'est à dire qu'un opérateur binaire est placé entre ses deux opérandes. Mais une autre façon d'évaluer les expressions mathématiques consiste à placer l'opérateur après ses opérandes. C'est ce qu'on appelle la notation postfixée, appelée aussi notation polonaise inversée car introduite en 1920 par le polonais Jan Lukasiewicz : Cela permet d'éviter les parenthèses et pas conséquent d'optimiser le compilateur.

Exemple

$3 + 4$ s'écrit en postfixé : $3 4 +$
 $3 + 5 - 9$ s'écrit en postfixé : $3 5 + 9 -$

Un intérêt de cette notation est qu'il n'y a plus besoin de connaître les priorités des opérateurs, et donc plus besoin de parenthèses (qui servent à contrer les priorités).

$3 + 2 * 5$ c'est-à-dire $3 + (2 * 5)$ s'écrit en postfixé : $3 2 5 * +$
 $(3 + 2) * 5$ s'écrit en postfixé : $3 2 + 5 *$
 $4 + 7 * (8 + 2)$ s'écrit en postfixé : $4 7 8 2 + * +$
 $4 7 + 8 2 * +$ donne en infixé : $4 + 7 + 8 * 2$

Pour effectuer un calcul en postfixé, on utilise une pile. On lit de gauche à droite et les règles sont les suivantes :

Si on rencontre :

- un nombre :
 - on l'empile
- un opérateur binaire:
 - on dépile le sommet et le sous sommet
 - on effectue le calcul sous-sommet "opération" sommet
 - on empile le résultat
- un opérateur unaire (une fonction ou le moins unaire) :
 - on dépile le sommet
 - on calcule la fonction pour la valeur du sommet
 - on empile le résultat

Remarque : Il faut bien différencier le moins unaire de l'opérande de soustraction. Pour cela, on utilisera plutôt NEG ou +/- pour l'opérateur unaire.

Un autre avantage du calcul en postfixé dans une calculatrice est qu'il n'y a pas d'état caché. L'utilisateur n'a pas besoin de se demander s'il a bien frappé la touche + - * ou / d'une calculatrice. En effet, une frappe entraîne immédiatement un calcul. D'un autre côté, il ne faut pas malencontreusement frapper deux fois une touche d'opération.

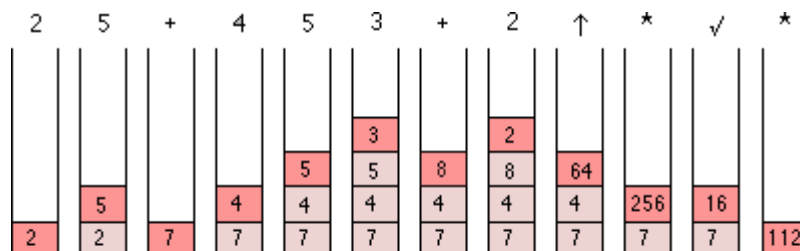
Exercice

Avec l'expression ci-dessous donnez les modifications de la pile en respectant les règles (en supposant la pile vide au départ)

$$2 \ 5 \ + \ 4 \ 5 \ 3 \ + \ 2 \ ^ \ * \ \sqrt{\quad} \ *$$

Quelle est l'écriture courant (infixée) de cette expression?

Correction



Il s'agit de $(2 + 5) \times \sqrt{4 \times (5 + 3)^2}$.

Traducteur

L'inconvénient le plus important du calcul en postfixé consiste en la gymnastique intellectuelle de traduction d'infixé en postfixé qui croît en complexité avec la taille de l'expression à traduire.

Une question naturelle devient donc : y-a-t-il un algorithme pour passer d'une infixé à une postfixé ?

La réponse est oui : la méthode consiste à empiler les opérateurs. On lit l'expression infixé caractère par caractère. Si c'est un opérande, on l'écrit. Sinon (c'est un opérateur), si la pile est vide ou bien si l'opérateur est (strictement) plus prioritaire que l'opérateur en sommet de pile, alors on l'empile. Sinon, on dépile jusqu'à pouvoir l'empiler. Lorsqu'il n'y a plus d'opérandes, on dépile.

Remarque

x^y est un opérateur binaire. Il s'agit de $x \ ^ \ y$.

Exemples

$2 + 3 * 4 - 5 + 6$ devient $2 \ 3 \ 4 \ * \ + \ 5 \ - \ 6 \ +$
 $3 + 2 - 5 * 6 / 2 + 3$ devient $3 \ 2 \ + \ 5 \ 6 \ * \ 2 \ / \ - \ 3 \ +$

Pour les parenthèses : on doit empiler les ouvrantes. Quant aux fermantes, elles imposent de tout dépiler jusqu'à l'ouvrante associée.

Exemple

$2 * 3 + 5 * (1 + 6 * 4) + 7$ devient $2 \ 3 \ * \ 5 \ 1 \ 6 \ 4 \ * \ + \ * \ + \ 7 \ +$
 $2 * (5 + 4 * (3 + 2) ^ 2)$ devient $2 \ 5 \ 4 \ 3 \ 2 \ + \ 2 \ ^ \ * \ + \ *$

Remarque

Ne pas oublier qu'il y a parfois des parenthèses sous-entendues.

Exemple

$\sqrt{3+5 \times 4} + (1+2)^2 = \sqrt{(3+5 \times 4)} + ((1+2)^2)$ et devient donc $3 \ 5 \ 4 \ * \ + \ \sqrt{\ } \ 1 \ 2 \ + \ 2 \ ^ \ +$

Exercice

Traduire en postfixé l'expression $3 * (4 + \sqrt{5^2 + 2}) + 1/6 * 7$.

Correction

$3 \ 4 \ 5 \ 2 \ ^ \ 2 \ + \ \sqrt{\ } \ + \ * \ 1 \ 6 \ / \ 7 \ * \ +$