Listes

Plan

1. Définition
2. Implémentation par deux vecteurs
3. Implémentation par vecteur de couples
4. Implémentation par pointeurs

1. Définition

Une *liste* L d'éléments pris dans un ensemble E est une suite finie

$$L = (a_1, a_2, \ldots, a_n)$$

avec $a_1, \ldots, a_n$ dans E. Les *opérations* usuelles sur les listes comportent

- L'initialisation:
$$L := ()$$
- L'insertion en début de liste:
$$x,\ (a_1, \ldots, a_n) \mapsto (x, a_1, \ldots, a_n)$$
- La recherche d'un élément dans une liste:
$$x,\ (a_1, \ldots, a_n) \mapsto T, F$$
- La suppression d'un élément d'une liste:
$$(a_1, \ldots, a_i, x, a_{i+2}, \ldots, a_n) \mapsto (a_1, \ldots, a_i, a_{i+2}, \ldots, a_n)$$

Il y en a d'autres.

Deux implémentations sont envisageables.

L'implémentation *par tableau* fait coïncider la succession dans une liste (*succession logique* : élément suivant) avec la succession des places dans le tableau (*succession physique* : incrémentation de l'indice). Il en résulte

- une grande facilité d'accès : le i-ème élément est acessible en temps constant;
- une certaine rigidité dans la modification, car il faut déplacer des plages d'éléments pour rétablir la coïncidence de la succession.

Les implémentations en *listes chaînées* dissocient la succession logique et la succession physique: Une *liste chaînée* est une séquence d'objets dont chacun contient une indication explicite vers l'élément suivant. Une liste est faite de *cellules*. Chaque cellule contient un élément de la liste et l'*adresse* de l'élément suivant. Il en résulte

- une facilité certaine pour l'insertion en début de liste (temps constant);
- une certaine facilité pour la suppression d'un élément (temps constant après localisation);
- une lenteur pour la localisation du i-ème élément (parcours de la liste depuis le début).

On présente ici trois réalisations des listes chaînées:

- par deux vecteurs,
- par un vecteur de couples,
- par pointeurs.

2. Implémentation par deux vecteurs

Une liste chaînée est implémentée par deux vecteurs `cont` et `suiv`. Une cellules est formée du couple `cont[i]` , `suiv[i]`:

- `cont[i]` est le contenu de la cellule d'indice `i`;
- `suiv[i]` est l'indice de la cellule suivant la cellule d'indice `i`.

Une liste est l'*indice de sa première cellule*. Le nombre `0` sert à indiquer la fin de la liste.

Exemple:

```
Liste = 4
  X  B

        1  2  3  4  5  6  7  8  9
cont       B     X
suiv       0     2
```

Ici

- La liste `(X B)` est représentée par l'indice `4`.
- Son premier élément est `cont[4] = X`.
- L'indice de la cellule suivante est `2`.
- La cellule d'indice `2` est la dernière, car `suiv[2]` vaut `0`.

L'implémentation s'écrit:

```
#define taille 10
typedef char Element;  /* une liste d'elements    */
typedef int Indice;    /* une cellule a un indice */
typedef Indice Liste;  /* une liste est un indice */

Element cont[taille];  /* le vecteur des contenus */
Indice  suiv[taille];  /* le vecteur des suivants */
```

La *création* d'une liste vide se fait par

```
Liste FaireListe() {     /* retourne une liste vide */
  return 0;
}
```

L'*insertion* d'un élément en début de liste s'écrit:

```
Liste Lajouter(Element x, Liste a) {
  Indice  b;
  b = NouvelleCellule();
  cont[b] = x;
  suiv[b] = a;
  return b;
}
```

Ici, la fonction "magique" `NouvelleCellule()` fournit à la demande l'indice d'une nouvelle cellule libre. Ainsi, partant de

```
Liste = 4
  X  B

      1  2  3  4  5  6  7  8  9
cont     B     X
suiv     0     2
```

l'insertion de `Z` donne, si `NouvelleCellule()` retourne `1`:

```
Liste = 1
  Z  X  B

      1  2  3  4  5  6  7  8  9
cont  Z  B     X
suiv  4  0     2
```

La *suppression* d'un élément dans une liste s'écrit (par exemple):

```
Liste Lsupprimer(Element x, Liste a) {
  Indice b;
  if (a== 0) return a;
  if (cont[a] == x) {
    b = suiv[a];
    Liberer(a);
    return b;
  }
  else {
    suiv[a] = Lsupprimer (x, suiv[a]);
    return a;
  }
}
```

Dans cette fonction, la fonction "magique" `Liberer(a)` déclare que la cellule d'indice `a` est à nouveau utilisable.

Par exemple, en partant de:

```
Liste = 3
  W  Z  X  B

     1  2  3  4  5  6  7  8  9
cont Z  B  W  X
suiv 4  0  1  2
```

la suppression de `X` donne:

```
Liste = 3
  W  Z  B

     1  2  3  4  5  6  7  8  9
cont Z  B  W
suiv 2  0  1
```

Gestion des cellules libres

Encore faut-il gérer soi-même les cellules libres.

- Au début, toutes les cellules sont libres;
- une demande d'insertion requiert une cellule libre, à fournir par `NouvelleCellule()`;
- une suppression libère une cellule, à incorporer par `Liberer()`.

Les cellules libres ont elle-mêmes organisées en liste, l'indice de la première cellule libre est maintenu dans une variable globale `Libre`. Au début, cette liste contient toute la mémoire, et est représenté par l'indice `1`:

```
Libre = 1
     1  2  3  4  5  6  7  8  9
cont
suiv 2  3  4  5  6  7  8  9  0
```

A tout instant, la mémoire est partagée par deux listes:

- la liste des cellules occupées;
- la liste des cellules libres.

Par exemple:

```
Liste = 4
  X  B

Libre = 1
     1  2  3  4  5  6  7  8  9
cont    B     X
suiv 3  0  5  2  6  7  8  9  0
```

On fournit une cellule fraîche se fait par:

```
Indice NouvelleCellule()  {
  Indice t;
  t = Libre;
  Libre = suiv[Libre];
  return t;
}
```

On récupère une cellule libérée par:

```
void Liberer(Indice a) {
  cont[a] = ' '; /*facultatif*/
  suiv[a] = Libre;
  Libre = a;
}
```

3. Implémentation par vecteur de couples

Une liste chaînée est implémentée par un vecteur unique `M` de cellules. Chaque entrée `M[i]` est un *couple*.

- `M[i].cont` est le contenu de la cellule d'indice `i`;
- `M[i].suiv` est l'indice de la cellule suivant la cellule d'indice `i`.

Comme avant, une liste est *l'indice de sa première cellule*. Le nombre `0` sert à indiquer la fin de la liste.

L'implémentation s'écrit:

```
#define taille 10
typedef char Element;  /* une liste d'elements    */
typedef int Indice;    /* une cellule a un indice */
typedef Indice Liste;  /* une liste est un indice */
typedef struct {
  Element cont;
  Indice suiv;
} Cellule;

Cellule M[taille];  /* le vecteur des cellules */
```

Les fonctions sont très semblables aux précédentes.

La *création* d'une liste vide se fait par

```
Liste FaireListe() {    /* retourne une liste vide */
  return 0;
}
```

(Ecriture identique à la première version).

L'*insertion* d'un élément en début de liste s'écrit:

```
Liste Lajouter(Element x, Liste a) {
  Indice  b;
  b = NouvelleCellule();
  M[b].cont = x;
  M[b].suiv = a;
  return b;
}
```

Rappelons que dans la première version, c'était:

```
Liste Lajouter(Element x, Liste a) {
  Indice  b;
  b = NouvelleCellule();
  cont[b] = x;
  suiv[b] = a;
  return b;
}
```

La *suppression* d'un élément dans une liste s'écrit:

```
Liste Lsupprimer(Element x, Liste a) {
  Indice b;
  if (a== 0) return a;
  if (M[a].cont == x) {
    b = M[a].suiv;
    Liberer(a);
    return b;
  }
  else {
    M[a].suiv = Lsupprimer (x, M[a].suiv);
    return a;
  }
}
```

Dans la première version, c'était

```
Liste Lsupprimer(Element x, Liste a) {
  Indice b;
  if (a== 0) return a;
  if (cont[a] == x) {
    b = suiv[a];
    Liberer(a);
    return b;
  }
  else {
    suiv[a] = Lsupprimer (x, suiv[a]);
    return a;
  }
}
```

La gestion de la mémoire des cellules libres se fait de la même manière.

4. Implémentation par pointeurs

Un *pointeur* est une variable contenant l'*adresse* d'un objet en mémoire. La gestion de la mémoire est accessible au programmeur par des fonctions prédéfinies.

Une liste chaînée est implémentée par une suite de *cellules*. Chaque cellule `C` est un *couple*.

- `C.cont` est le contenu de la cellule `C`;
- `C.suiv` est l'adresse (l'indice) de la cellule suivant la cellule `C`.

Comme avant, une liste est *l'indice (adresse) de sa première cellule*. Le nombre `0` sert à indiquer la fin de la liste. Dans cette représentation, la gestion de la mémoire est prise en charge directement par le programme, et n'a pas à être implémentée.

```
#include <stdlib.h>
#define new(X) (X*) malloc(sizeof(X))

typedef char Element;           /* une liste d'elements    */
typedef struct cellule Cellule; /* une cellule est une structure */
typedef Cellule* Indice;        /* une cellule a un indice */
typedef Indice   Liste;         /* une liste est un indice */

typedef struct cellule {
  Element cont;
  Indice suiv;
};
```

Remarque: La gestion correcte de la mémoire demande l'inclusion du fichier d'en-tête `stdlib.h`. Traditionnellement, l'adresse nulle est notée `NULL` plutôt que `0`.

Les fonctions sont très semblables aux précédentes.

La *création* d'une liste vide se fait par

```
  Liste FaireListe() {   /* retourne une liste vide   */
    return NULL;         /* la tradition... NULL == 0 */
}
```

C'est la même écriture !

L'*insertion* d'un élément en début de liste s'écrit:

```
  Liste Lajouter(Element x, Liste a) {
    Indice  b;
    b = new(Cellule);
    b -> cont = x;
    b -> suiv = a;
    return b;
  }
```

au lieu de

```
  Liste Lajouter(Element x, Liste a) {
    Indice  b;
    b = NouvelleCellule();
    M[b].cont = x;
    M[b].suiv = a;
    return b;
  }
```

La macro `new()` joue le rôle tenu par la fonction `NouvelleCellule()`.

La *suppression* d'un élément dans une liste s'écrit:

```
Liste Lsupprimer(Element x, Liste a) {
  Indice b;
  if (a== 0) return a;
  if (a -> cont == x) {
    b = a -> suiv;
    free(a);
    return b;
  }
  else {
    a -> suiv = Lsupprimer (x, a -> suiv);
    return a;
  }
}
```

au lieu de

```
Liste Lsupprimer(Element x, Liste a) {
  Indice b;
  if (a== 0) return a;
  if (M[a].cont == x) {
    b = M[a].suiv;
    Liberer(a);
    return b;
  }
  else {
    M[a].suiv = Lsupprimer (x, M[a].suiv);
    return a;
  }
}
```

Ici, la fonction prédéfinie `free` réalise la fonction `Liberer()`.

Il n'est pas difficile de *voir* une liste.

```
  d  c  b  a

Liste = 24970544
adresse   cont   suiv
24970544   d   24970536
24970536   c   24970528
24970528   b   24970520
24970520   a   0
```

Après suppression de `b`:

```
  d  c  a

Liste = 24970544
adresse   cont   suiv
24970544   d   24970536
24970536   c   24970520
24970520   a   0
```

Ceci montre que le principe est exactement le même.