Cours Structures de Données Avancées

Option G.L.
Semestre 3

Chapitre 2 : Structures de données linéaires (suite)

2. Le type abstrait LISTE

Définition

Deux définitions peuvent être données pour une liste:

1. Définition itérative: une liste linéaire est une suite finie, éventuellement vide, d'éléments repérés selon leur rang dans la liste.
2. Définition récursive: une liste d'éléments est soit la liste vide, soit une paire (t, r) où t est le premier élément de la liste et r est la liste restante.

L'ordre des éléments dans une liste est fondamental. Ce n'est pas un ordre sur les éléments, mais un ordre sur les places des éléments. Les places sont totalement ordonnées, c.à.d. qu'il existe une fonction succ (successeur) pour les places. Chaque place a un contenu qui est un élément. Le nombre n d'éléments (donc de places) est appelé la longueur de liste. Si n = 0, la liste est vide. La fonction succ n'est pas définie pour la nième place (successeur de la dernière place n'est pas défini).

Il existe plusieurs façons de définir le type abstrait liste suivant l'utilisation de la notion de liste qu'on veut avoir. Nous donnons deux types abstraits LISTE selon ses définitions précédentes: LISTE ITÉRATIVE ET LISTE RÉCURSIVE.

TAD LISTE ITÉRATIVE
Sorte Liste
Utilise ENTIER, ELÉMENT
Opérations:
liste_vide: → liste
insérer: Liste, entier, élément → liste
supprimer: liste, entier → liste
ième: liste, entier → élément
longueur: liste → entier
accés: liste , entier → entier
Préconditions:
Pré(accés(l,i)) est définie ssi 1≤ i ≤ longueur(l)
Pré(ième(l,i)) est définie ssi 1≤ i ≤ longueur(l)
Pré(supprimer(l,i)) est définie ssi 1≤i≤ longueur(l)
Pré(insérer(l,i,e)) est définie ssi 1≤i≤ longueur(l)+1
Axiomes:
longueur(liste_vide) ≡ o
longueur(insérer(l,i,e)) ≡ longueur(l) + 1
i < j ⇒ accés(insérer(l,i,e),j) ≡ accés(l, j - 1)
i > j ⇒ accés(insérer(l,i,e),j) ≡ accés(l, j)
ième(insérer(l,i,e),i) ≡ e
i < j ⇒ ième(insérer(l,i,e),j) ≡ ième(l, j - 1)
i > j ⇒ ième(insérer(l,i,e),j) ≡ ième(l, j)
supprimer(insérer(l,i,e),i) ≡ l
i > j ⇒ supprimer(insérer(l,i,e),j) ≡ insérer(supprimer(l, j),i - 1, e)

i < j ⇒ supprimer(insérer(l,i,e),j) ≡ insérer(supprimer(l, j - 1),i , e)
Variables:
l: liste; i,j: entiers; e:élément

TAD LISTE RÉCURSIVE
Sorte Liste
Utilise ELÉMENTN ENTIER
Opérations:
liste_vide: → liste
cons: élément, liste → liste
reste: liste → liste
premier: liste → élément
longueur: liste→ entier
Préconditions:
Pré(reste(l)) est définie ssi l ≠ liste_vide
Pré(premier(l)) est définie ssi l ≠ liste_vide
Axiomes:
premier(cons(e,l)) ≡ e
reste(cons(e,l)) ≡ l
longueur(liste_vide) ≡ o
longueur(cons(e,l)) ≡ longueur(l) + 1
Variables:
l: liste; e:élément

Exemple: Construire la liste d'entier L=<6, 55, 444>.
1. liste récursive
   L=cons(6,cons(55,cons(444, liste_vide)))
2. liste iterative
   L=insérer(insérer(insérer(liste_vide ,3,444),2,55),1,6)

Les deux définitions itérative et récursive des listes sont équivalentes. C'est à dire qu'on peut définir les opérations de la liste itérative à l'aide de celles de la liste récursive et vice versa.

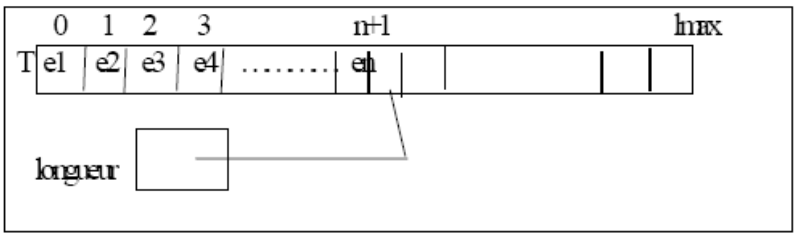
3. Implémentation des listes

Il peut exister plusieurs manières de représentation du TAD LISTE. Nous présentons dans ce cours deux implémentations possibles l'une à base des tableaux (contigue) pour le TAD LISTE ITERATIVE et l'autre à base des pointeurs (chaînée) pour le TAD LISTE RECURSIVE.

2.1 Représentation contiguë

La liste est représentée par un tableau dont la ième case est la ième place de la liste. Elle est donc désignée par le couple: (tableau, longueur).

Liste= <e1, e2, ..., en> est représentée par:



L'implémentation en JAVA de ce TAD LISTE ITERATIVE nécessite la déclarations de la classe suivante et l'ensemble de ses méthodes. On peut réaliser de façon analogue l'implémentation du TAD LISTE RECURSIVE.

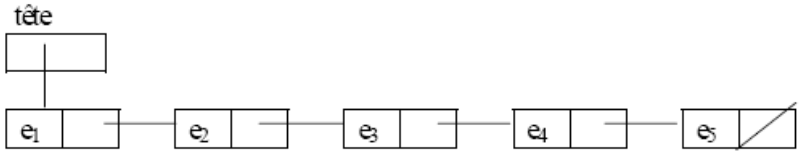
```
public class Liste-Iterative {
```

```
        element [ ]  T = new element [Lmax] ;
        public liste  Liste-vide ()
        {
            liste L= new element [0];
            return(L);
        }
        public element  Ieme (liste L, int I)
        {
          Return L.T[I-1];
        }
        public  liste Inserer (liste L, int I, element E)
        {
            int  J ;
            for (J = L.longueur ; J > I ; J - -)
            L.T[J] = L.T[J - 1];
            L.T[I - 1] = E;
            return(L);
        }
        public  liste Supprimer (liste L, int I)
        {
            Int J ;
            for (J = I - 1; J ≤ L.longueur ; J ++)
            L.T[J] = L.T[J-1];
            return(L);
        }
        public int   Longueur (liste L)
        {
          return L.Longueur  ;
         }
        public int   Est-vide (liste L)
        {
          return L == Liste-vide();
         }
    }
```

2.2 Représentation chaînée

Une liste peut être représentée par un ensemble de cellules où chaqu'une contient en plus de la valeur d'un élément, l'adresse de l'élément suivant. La liste est alors donnée par le pointeur du premier élément.

Par exemple la liste L = (e_1,e_2,e_3,e_4,e_5) peut être représentée de la façon suivante :



En langage JAVA , on peut donc déclarer ce type de liste et les opérations qui le gèrent par l'implantation du TAD LISTE RECURSIVE sous forme de classe JAVA (présentée ci dessous) et ses méthodes respectives. Le même principe est utilisé pour implémenter le TAD LISTE ITERATIVE en utilisant la représentation chaînée.

```
    public class Liste-récursive {
        public class cellule {
                element info ;
                cellule suiv ;
            } ;
        cellule  Tête = new cellule ;

        public  liste  Liste-vide ()
;         {
              Return Null;
          };

        public liste reste (liste l)
          {
              return(l.tête.suiv);
           };
```

```
public liste cons (liste l; élément e)
   {
     cellule  T = new cellule ;

     T.info = e ;
     T.suiv = l;
      l.tête = T
     return(T);
     }

public int longueur (liste l)
     {
       if (l==null)
          return(0);
       else
          return(1 + longueur(reste(l));
     }
```

4. Exemples de listes: Piles et Files

Deux cas particuliers des listes jouent un rôle important en informatique: les piles où les données sont ajoutées et supprimées en une même extremité (sommet); les files, où les données sont ajoutées à une extrémité de la liste (queue) et supprimées à l'autre extrémité (tête).

4.1 Les piles

Une pile est une collection d'objets qui obeit au protocole LIFO (Last In First Out). Toutes les insertions et les suppressions se font à une seule extrêmité appellée sommet de pile.
Les opérations sur les piles sont: créer une pile vide, empiler un nouvel objet au sommet de la pile, dépiler l'objet actuellement au sommet, accéder à l'élément en tête de pile, tester la vacuité d'une pile. Le TAD spécifiant les piles est donné par:

TAD Pile
Sorte pile
Utilise Booléen, Elément
Opérations:
pile_vide: → pile
empiler: pile, élément → pile
dépiler: pile → pile
sommet: pile → élément
est_vide: pile → booléen
Préconditions:
Pré(dépiler(p)) est définie ssi est_vide(p)=faux
Pré(sommet(p)) est définie ssi est_vide(p)=faux
Axiomes:
dépiler(empiler(p,e)) ≡ p
sommet(empiler(p,e))≡ e
est_vide(pile_vide) ≡ vrai
est_vide(empiler(p,e)) ≡ faux
Variables:
p:pile; e:élément

Les opérations internes sont: pile_vide, empiler et d´epiler. Les opérations observateurs qu'on peut donner ici sont: sommet et est_vide.
Nous pouvons enrichir le TAD pile par d'autres opérations analogues à celles des listes linéaires.

4.2 Les files

Par analogie avec les files d'attente, la liste linéaire dite file est une structure qui permet de stocker des objets dans un ordre donné et de les retirer dans le même ordre, c'est à dire selon le protocole FIFO ("First In First Out"). On ajoute toujours un élément en queue de liste et on retire celui qui est en tête de liste. On utilise la notion de file pour gérer l'allocation d'une ressource à plusieurs "clients". Dans le cas le plus simple on procède sur la base du "premier

arrivé premier servi". On peut donner aux éléments de la file des priorités diffrentes qui leur permettent de dépasser les éléments moins prioritaires. De tels problèmes apparaissent dans les systèmes d'exploitation d'ordinateurs, dans les réseaux de télécommunications, ou dans les programmes de gestion du courrier électronique. Les files sont utilisées également comme support du protocole "producteur/consommateur" entre deux processus asynchrones. Dans ce cas un processus P produit des données qui sont envoyées à un processus C pour les consommer. Il faut passer par l'intermédiaire d'une file pour gérer l'attente des données avant d'être consommer. Les files informatiques peuvent également servir pour simuler les files d'attentes réelles qui se cérent dans diverses situations (guichets, trafic automobile, etc.).
Les opérations internes sur les files sont: ajouter un élément à la file, retirer le premier élément de la file, et la création d'une file vide. Les opérations observateurs sont entre autre: tester si une file est vide et accéder au premier élément de la file. La spécification algébrique de cette structure est alors donnée par le TAD suivant:

TAD File
Sorte file
Utilise Booléen, Elément
Opérations:
file_vide: → file
ajouter: file, élément → file
supprimer: file → file
tête, queue: file → élément
est_vide: file → booléen
Préconditions:
Pré(supprimer(f)) est définie ssi est_vide(f) = faux
Pré(queue(f)) est définie ssi est_vide(f) = faux
Pré(tête(f)) est définie ssi est_vide(f) = faux
Axiomes:
supprimer(ajouter(f,e)) ≡ si est_vide(f) alors file_vide sinon ajouter(supprimer(f),e)
tête(ajouter(f,e))≡ si est_vide(f) alors e sinon tête(f)
queue(ajouter(f,e))≡ e
est_vide(file_vide) ≡ vrai
est_vide(ajouter(f,e)) ≡ faux
Variables:
f: file; e: élément