Les listes

Nour-Eddine Oussous, Éric Wegrzynowski

Licence ST-A, USTL - API2

26 octobre 2009

Objectif :

- Trouver une représentation de la structure linéaire de liste
- à l'aide de listes chaînées par des pointeurs
- et implémenter les opérations primitives à l'aide de cette représentation
- de telle sorte que toute autre opération sur les listes puisse être implémentée à l'aide de ces opérations primitives sans manipulation explicite des pointeurs.

Structure de liste

Notion de liste

Définition (1)

liste = suite finie, éventuellement vide, d'éléments

- exemple : (3, 1, 4, 1, 5, 9, 2) est une liste d'entiers de longueur 7
- une liste non vide possède
  1. un élément de tête : 3
  2. et un reste qui est une liste : (1, 4, 1, 5, 9, 2)
- () est la liste vide, de longueur 0
- (2) est une liste de longueur 1 dont le reste est ()

Structure récursive des listes

Toute liste non vide peut être vue comme un couple $< \text{tete}; \text{reste} >$.
Par exemple :

$$(3,1,4,1,5,9,2) \quad = \quad < \underbrace{3}_{\text{tete}} ; \underbrace{(1,4,1,5,9,2)}_{\text{reste}} >$$

et en poursuivant sur les différents restes :

$$\begin{aligned} &= \quad < 3; < 1; (4,1,5,9,2) >> \\ &\vdots \\ &= \quad < 3; < 1; < 4; < 1; < 5; < 9; < 2; () >>>>>>> \end{aligned}$$

Définition récursive

Définition (2)

Une liste est donc

1. soit la liste vide $()$;
2. soit un couple $<$ tete; liste $>$.

En notant $Liste(E)$ l'ensemble des listes dont les éléments sont pris dans un ensemble E, on a

$$Liste(E) = \{()\} \cup \left(E \times Liste(E)\right)$$

Ajout d'un élément en tête de liste

Spécification

$$\begin{array}{rccc} \texttt{ajouteEnTete} : & E \times Liste(E) & \longrightarrow & Liste(E) \\ & e, l & \longmapsto & < e; l > \end{array}$$

Exemple

$$\texttt{ajouteEnTete}(3, (1,4,1,5,9,2)) = (3,1,4,1,5,9,2)$$

C'est une opération de construction de liste.

Accès à la tête d'une liste

Spécification

$$\begin{array}{rccc} \texttt{tete} : & Liste(E) & \longrightarrow & E \\ & < e; l > & \longmapsto & e \end{array}$$

CU : La liste passée en paramètre ne doit pas être vide.

Exemple

$$\texttt{tete}((3,1,4,1,5,9,2)) = 3$$

C'est un sélecteur.

Accès au reste d'une liste

Spécification

$$\begin{array}{rccc} \texttt{reste} : & \mathit{Liste}(E) & \longrightarrow & \mathit{Liste}(E) \\ & < e; l > & \longmapsto & l \end{array}$$

CU : La liste passée en paramètre ne doit pas être vide.

Exemple

$$\texttt{reste}((3,1,4,1,5,9,2)) = (1,4,1,5,9,2)$$

C'est un sélecteur.

Test de vacuité d'une liste

Spécification

$$\begin{array}{rccc} \texttt{estListeVide} : & \mathit{Liste}(E) & \longrightarrow & \mathit{Booleen} \\ & l & \longmapsto & \begin{cases} \text{Vrai} & \text{si } l\text{est vide} \\ \text{Faux} & \text{sinon} \end{cases} \end{array}$$

Exemples

$$\begin{aligned} \texttt{estListeVide}(()) &= \text{Vrai} \\ \texttt{estListeVide}((3,1,4,1,5,9,2)) &= \text{Faux} \end{aligned}$$

C'est un prédicat.

Changer la tête d'une liste

Spécification

$$\begin{array}{rccc} \texttt{modifierTete} : & \mathit{Liste}(E), E & \longrightarrow & \mathit{Liste}(E) \\ & < e'; l >, e & \longmapsto & < e; l > \end{array}$$

CU : La liste passée en paramètre ne doit pas être vide.

Exemple

$$\texttt{modifierTete}((5,1,4,1,5,9,2),3) = (3,1,4,1,5,9,2)$$

C'est une opération de modification de liste.

Changer le reste d'une liste

Spécification

$$\begin{array}{rccc} \texttt{modifierReste} : & \mathit{Liste}(E), \mathit{Liste}(E) & \longrightarrow & \mathit{Liste}(E) \\ & < e; l' >, l & \longmapsto & < e; l > \end{array}$$

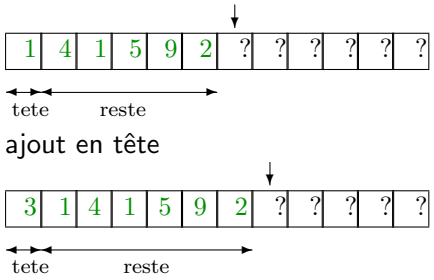
CU : La première liste passée en paramètre ne doit pas être vide.

Exemple

$$\texttt{modifierReste}((3,2),(1,4,1,5,9,2)) = (3,1,4,1,5,9,2)$$

C'est une opération de modification de liste.

Représentation contigüe par tableaux



ajout en tête

Inconvénients

- réservation mémoire qui peut être inutilisée
- ou insuffisante
- les opérations primitives sont coûteuses.

Objectifs

- occuper juste ce qu'il faut de mémoire
- avoir des opérations primitives aussi peu coûteuses que possible

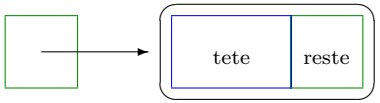
Moyen

- Utiliser l'allocation dynamique de mémoire
- Une liste est un pointeur
  - vers rien pour la liste vide (NIL)
  - vers un couple $< \text{tete}; \text{reste} >$ que l'on nommera cellule

- La liste vide est un pointeur vers rien

$L =$ 

- Une liste non vide est un pointeur vers une cellule
  $< \text{tete}; \text{reste} >$



Exemple de représentation chaînée

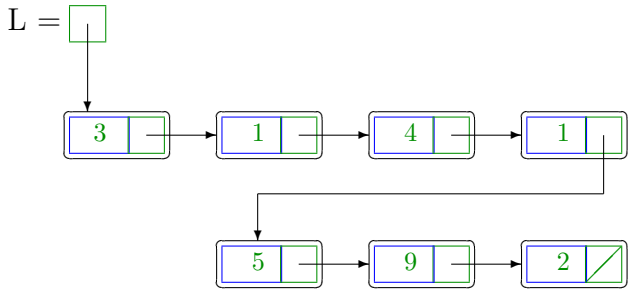


Fig.: Représentation chaînée de la liste L = (3, 1, 4, 1, 5, 9, 2)

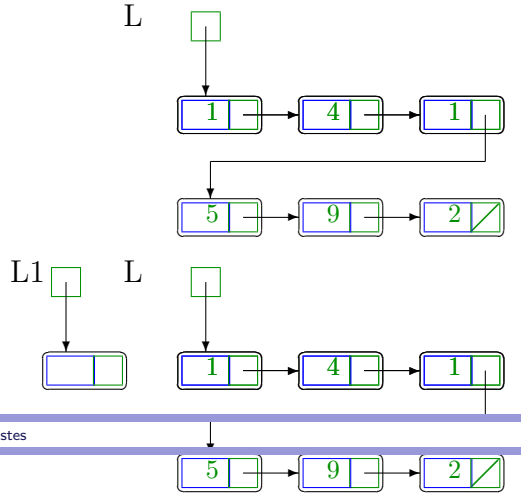
Déclaration en Pascal

```
LISTE    = ^CELLULE;
CELLULE = record
     info     : ELEMENT;
     suivant  : LISTE;
end {CELLULE};
const
  LISTEVIDE : LISTE = NIL;
```

Le type `ELEMENT` doit être déclaré par ailleurs.

Ajout en tête : algorithme

Ajout de 3 en tête de la liste L=(1,4,1,5,9,2).




```
new(L1)
L1^.info := 3
L1^.suivant := L
ajouteEnTete :=L1
```
Terminé !

La fonction ajouteEnTete

```
// ajouteEnTete(e,l) = <e;l>
function ajouteEnTete(e : ELEMENT;
                      l : LISTE) : LISTE;
var
   l1 : LISTE;
begin
   new(l1);
   l1^.info := e;
   l1^.suivant := l;
   ajouteEnTete := l1;
end {ajouteEnTete};
```


Coût de l'ajout en tête

- indépendant de la longueur de la liste
- indépendant de la valeur des éléments de la liste
- indépendant de la valeur de l'élément ajouté

Conclusion

coût constant : $\Theta(1)$

La fonction tete

```
//tete(l)=le premier element de la liste l
//CU : l non vide
function tete(l :  LISTE) : ELEMENT;
begin
   tete := l^.info;
end {tete};
```


Coût de l'accès à la tête

- indépendant de la longueur de la liste
- indépendant de la valeur des éléments de la liste

Conclusion

coût constant : $\Theta(1)$

La fonction `reste`

```
// reste(l) = la liste qui suit le premier
//            element de l
// CU : l non vide
function reste(l : LISTE) : LISTE;
begin
   reste := l^.suivant;
end {reste};
```


Coût de l'accès au reste

- indépendant de la longueur de la liste
- indépendant de la valeur des éléments de la liste

Conclusion

coût constant : $\Theta(1)$

Le prédicat `estListeVide`

```
// estListeVide(l) ⇔ l est vide
function estListeVide(l: LISTE):BOOLEAN;
begin
   estListeVide := l=NIL;
end {estListeVide};
```


Coût du test de vacuité

- indépendant de la longueur de la liste
- indépendant de la valeur des éléments de la liste

Conclusion

coût constant : $\Theta(1)$

La procédure `modifierTete`

```
// modifierTete(l,e) modifie la valeur de
// l'element en tete de l
// CU : l non vide
procedure modifierTete(const l : LISTE;
                       const e : ELEMENT);
begin
   l^.info := e ;
end {modifierTete};
```

Remarque : le paramètre *l* est constant car c'est un pointeur.

Coût de la modification de la tête

- indépendant de la longueur de la liste
- indépendant de la valeur des éléments de la liste
- indépendant de la nouvelle valeur de la tête

Conclusion

coût constant : $\Theta(1)$

La procédure `modifierReste`

```
// modifierReste(l,ll) modifie la valeur
// du reste de l en lui attribuant ll
// CU : l non vide
procedure modifierReste(const l : LISTE;
                        const ll : LISTE);
begin
   l^.suivant := ll;
end {modifierReste};
```

Remarque : le paramètre *l* est constant car c'est un pointeur.

Coût de la modification du reste

- indépendant de la longueur de la liste
- indépendant de la valeur des éléments de la liste à modifier
- indépendant de la valeur des éléments du nouveau reste.

Conclusion

coût constant : $\Theta(1)$

Utilisation des primitives

```
{ L une liste qque, e est un element qque}
L1 := ajouteEnTete(e, L) ;
{ L1 = <e; L> }
```

```
{ L = <t; R> une liste non vide}
e := tete(L) ;
{ e = t et L est inchangee }
```

```
{ L = <t; R> une liste non vide}
L1 := reste(L) ;
{ L1 = R et L est inchangee }
```


Utilisation des primitives

```
{ L = <t; R> une liste non vide,
             e est un element qque}
modifierTete(L, e) ;
{ L = <e; R> et L est inchangee }
```

```
{ L = <t; R> une liste non vide,
             S est une liste qque}
modifierReste(L, S) ;
{ L = <t; S> et L est inchangee }
```