

# Université Bordeaux 1

## Licence Semestre 3 - Algorithmes et structures de données 1

Dernière mise à jour effectuée le 1 Septembre 2013

---

### Listes

---

- [Définition](#)
  - [Liste simplement chaînée](#)
  - [Liste doublement chaînée](#)
  - [Implémentation par un tableau du type listeSC](#)
  - [Implémentation par allocation dynamique du type listeSC](#)
- 

## 1. Définition

- Définition 6.1.** Une **liste** est un conteneur tel que
- le nombre d'objets (**dimension** ou **taille**) est variable,
  - l'accès aux objets se fait indirectement par le contenu d'une clé qui le localise de type **curseur**.

Un curseur est un type abstrait dont l'ensemble des valeurs sont des positions permettant de localiser un objet dans le conteneur. Dans le cas où il n'y a pas de position, la valeur par défaut est NIL. Si  $c$  est un curseur, les primitives considérées dans ce cours sont les suivantes :

- accès à l'élément désigné par le curseur,  $\text{contenu}(c) : \text{curseur} \rightarrow \text{objet}$
- accès à la valeur du curseur,  $\text{getCurseur}(c) : \text{curseur} \rightarrow \text{valeur\_de\_curseur}$
- positionnement d'un curseur,  $\text{setCurseur}(c, \text{valeur}) : \text{curseur} \times \text{valeur\_de\_curseur} \rightarrow \text{vide}$
- existence d'un élément désigné par le curseur,  $\text{estVide}(c) : \text{curseur} \rightarrow \{\text{vrai}, \text{faux}\}$

La manipulation des éléments de la liste dépend des primitives définies comme s'exécutant en temps  $O(1)$ .

---

## 2. Liste simplement chaînée

- Définition 6.2.** Une liste est dite **simplement chaînée** si les opérations suivantes s'effectuent en  $O(1)$ .
- accès

```
fonction valeur(val L:liste d'objet):objet;
/* si la clé==NIL alors le résultat est NULL */
fonction debutListe(val L:liste d'objet):vide;
/* positionne la clé sur le premier objet de la liste */
fonction suivant(val L:liste d'objet):vide;
/* avance la clé d'une position dans la liste */
fonction listeVide(val L:liste d'objet): booleen;
/* est vrai si la liste ne contient pas d'élément */
fonction getCléListe(val L: liste d'objet):curseur;
/* permet de récupérer la clé de la liste */
```

- o modification

```
fonction créerListe(ref L:liste d'objet):vide;
fonction insérerAprès(ref L:liste d'objet;
                    val x:objet):vide;
/* insère un objet après la clé, la clé ne change pas */
fonction insérerEnTete(ref L:liste d'objet
                    val x:objet):vide;
/* insère un objet en début de liste,
   la clé est positionnée sur la tête de liste */
fonction supprimerAprès(ref L:liste d'objet):vide;
/* supprime l'objet après la clé, la clé ne change pas */
fonction supprimerEnTete(ref L:liste d'objet):vide;
/* supprime un objet en début de liste,
   la clé est positionnée sur la tête de liste */
fonction setCléListe(ref L: liste d'objet;val c: curseur):vide;
/* permet de positionner la clé de la liste)
fonction detruireListe(ref L:lisetd'objet):vide;
```

## Détection fin de liste

```
fonction estFinListe(val L:liste d'objet):booléen;
début
    retourner(valeur(L)==NULL)
fin
```

## Chercher un élément dans une liste

```
fonction chercher(ref L:liste d'objet; ref x:objet): boolean;
début
    debutListe(L);
    tant que !estFinListe(L) et valeur(L)!=x faire
        suivant(L);
    fintantque
    retourner (!estFinListe(L))
/* la clé vaut NIL ou est positionné sur l'objet */
fin
finfonction
```

### Complexité:

- o minimum :  $O(1)$
- o maximum :  $O(n)$

## Supprimer un élément dans la liste s'il existe

```
fonction supprimer(ref L:liste d'objet; val x:objet): vide;
var tmp:curseur;
/* on suppose que l'objet se trouve dans la liste */
début
    debutListe(L);
    tmp=NIL;
    tant que !estFinListe(L) et contenu(getCléListe(L))!=x faire
        tmp= getCléListe(L);
        suivant(L);
    fintantque
    si tmp==NIL alors
        supprimerEnTete(L)
        /* la clé est sur la tête de liste */
    sinon
        setCléListe(L,tmp);
        supprimerAprès(L)
        /* la clé est sur l'objet précédent l'objet supprimé */
    finsi
fin
```

```
finfonction
```

#### Complexité:

- minimum :  $O(1)$
- maximum :  $O(n)$

### Exercice

Réfléchir aux problèmes que soulèvent l'introduction de `setCléListe` et surtout `getCléListe`? Que déduire? Faut-il vraiment les garder?

---

## 3. Liste doublement chaînée

**Définition 6.3.** Une liste **doublement chaînée** est une liste pour laquelle les opérations en temps  $O(1)$  sont celles des listes simplement chaînées auxquelles on ajoute les fonctions d'accès

```
fonction finListe(val L:liste d'objet):vide;
/* positionne la clé sur le dernier objet de la liste */
fonction précédent(val L:liste d'objet): vide;
/* recule la clé d'une position dans la liste */
```

### Supprimer un élément

```
fonction supprimer(ref L:liste d'objet; val x:objet): booleen;
début
  si chercher(L,x) alors
    précédent(L);
    supprimerAprès(L);
    retourner(vrai)
  sinon
    retourner(faux)
  finsi
fin
finfonction
```

#### Complexité

- minimum :  $O(1)$
  - maximum :  $O(n)$
- 

## 4. Implémentation par un tableau du type liste

Chaque élément du tableau est une structure (objet,indexSuivant). Le champ `indexSuivant` désigne une entrée du tableau. Ainsi l'accès au suivant est en complexité  $O(1)$ . La zone de stockage peut donc être décrite par :

```
curseur=entier;
elementListe=structure
  valeur:objet;
  indexSuivant:curseur;
finstructure;

stockListe=tableau[1..tailleStock] d'elementListe;
```

Dans ce contexte, le type `curseur` est un entier compris entre 1 et `tailleStock`. La valeur du champ `indexSuivant` est donc un entier compris entre 0 et `tailleStock`. Il faut coder la valeur NIL : on peut par exemple choisir la valeur 0. Le premier élément doit être accessible en  $O(1)$ , il faut donc conserver son index. On peut donc représenter une liste

par la structure suivante :

```
listeSC=structure
    tailleStock:entier;
    vListe:stockListe;
    premier:curseur;
    cle:curseur;
finstructure;
```

Le tableau de stockage étant grand mais pas illimité, il faudra prévoir que l'espace de stockage puisse être saturé.

### o Primitives d'accès

Ces fonctions sont immédiates.

```
fonction debutListe(val L:listeSC):vide;
    début
        L.cle=L.premier;
    fin;
finfonction

fonction suivant(val L:listeSC):vide;
    début
        L.cle=L.vListe[L.cle].indexSuivant;
    fin
finfonction

fonction listeVide(val L:listeSC): booléen;
    début
        retourner(L.premier==0);
    fin
finfonction
```

---

### o Gestion de l'espace de stockage

Pour ajouter un élément, il faut pouvoir trouver un élément "libre" dans le tableau. Une solution compatible avec la complexité des primitives consiste à gérer cet espace de stockage en constituant la liste des cellules libres ([voir un exemple](#)). On modifie donc en conséquence la description de listeSC :

```
listeSC=structure
    tailleStock:entier;
    vListe:stockListe;
    premier:curseur;
    premierLibre:curseur;
    cle:curseur;
finstructure;
```

Par convention, l'espace de stockage sera saturé lorsque l'index premierLibre vaut 0 (la liste des cellules libres est vide). On définit donc la fonction de test :

```
fonction listeLibreVide(val L:listeSC):booléen;
    début
        retourner (L.premierLibre==0);
    fin
finfonction
```

On définit deux primitives liées à la gestion de la liste des libres :

- mettreCellule : met une cellule libre en tete de la liste des cellules libres,
- prendreCellule : prend la cellule libre entete de la liste des cellules libres.

Les opérations sont respectivement de type insererEnTete et supprimerEnTete.

```

fonction prendreCellule(ref L:listeSC):curseur;
  var nouv:curseur;
  début
    nouv=L.premierLibre;
    L.premierLibre=L.vListe[nouv].indexSuivant;
    retourner nouv;
  fin
finfonction

fonction mettreCelluleEnTete(ref L:listeSC,val P:curseur):vide;
  début
    L.vListe[P].indexSuivant=L.premierLibre;
    L.premierLibre=P;
  fin
finfonction

```

---

## o Deux primitives de modification

```

fonction créer_liste(ref L:listeSC):vide;
  var i:curseur;
  début
    L.tailleStock=tailleMax;
    L.premier=0;
    L.premierLibre=1;
    pour i allant de 1 à L.tailleStock-1 faire
      L.vListe[i].indexSuivant=i+1;
    finpour
    L.vListe[tailleStock].indexSuivant=0;
    L.cle=0;
  fin
finfonction

fonction insérerAprès(ref L:listeSC;val x:objet):booléen;
var tmp,nouv:curseur;
  début
    si L.cle==0 ou L.premierLibre==0 alors
      retourner faux;
    sinon
      tmp=L.cle;
      nouv=prendreCellule(L);
      L.vListe[nouv].valeur=x;
      suivant(L);
      L.vListe[nouv].indexSuivant=L.cle;
      L.vListe[tmp].indexSuivant=nouv;
      L.cle=tmp;
      retourner vrai;
    finsi
  fin
finfonction

```

---

## 5. Implémentation par allocation dynamique du type listeSC

Chaque élément de la liste est une structure (valeurElement,pointeurSuivant). le champ pointeurSuivant est une adresse en mémoire, par suite, l'accès au suivant est en complexité  $O(1)$ . Dans ce contexte le type curseur est un pointeur vers un élément. La zone de stockage peut donc être décrite par :

```

cellule=structure
  valeurElement:objet;
  pointeurSuivant:^cellule;
finstructure;
curseur:^cellule;
listeSC=structure

```

```
    premier:curseur;  
    cle:curseur;  
  finstructure
```

La valeur NIL correspond donc à l'absence d'élément suivant.

## o Primitives d'accès

Ces fonctions sont immédiates.

```
fonction premier(val L:listeSC d'objet):vide;  
  début  
    L.cle=L.premier;  
  fin;  
finfonction  
  
fonction valeur(val L:listeSC):car;  
  début  
    retourner( L.cle^.valeurElement);  
  fin  
finfonction  
  
fonction listeVide(val L:listeSC):booléen;  
  début  
    retourner (L.premier==NIL);  
  fin  
finfonction  
  
fonction suivant(val L:listeSC):vide;  
  début  
    /* la liste vide n'est pas testée car la primitive existe */  
    L.cle=L.cle^.pointeurSuivant;  
  fin  
finfonction
```

## o Trois primitives de modifications

```
fonction créer_liste(ref L:listeSC):vide;  
  début  
    L.premier=NIL;  
    L.cle:=NIL;  
  fin  
finfonction  
  
fonction supprimerEnTete(ref L:listeSC):vide;  
  var P:curseur;  
  début  
    P=L.premier;  
    suivant(L);  
    L.premier=L.cle;  
    delete(P);  
  fin  
finfonction  
  
fonction insérerAprès(ref L:listeSC;val x:objet):vide;  
var nouv:curseur;  
  début  
    new(nouv);  
    nouv^.valeurElement=x;  
    nouv^.pointeurSuivant=L.cle^.pointeurSuivant;  
    L.cle^.pointeurSuivant=nouv;  
  fin  
finfonction
```

---

## 6. Avantages et inconvénients

L'implémentation dans un tableau permet d'avoir un bloc contigu de mémoire ce qui va minimiser les accès disques. Ceci n'est pas le cas pour l'implémentation par pointeurs. L'implémentation dans un tableau nécessite de fixer au préalable le nombre maximum de cellules qui va contraindre fortement les applications : la structure de donnée peut avoir beaucoup trop de cellules ou au contraire trop peu.

L'implémentation par pointeur va être très dépendante de l'implémentation des modules d'allocation dynamique du langage choisi