table des matières

Université Bordeaux 1

Licence Semestre 3 - Algorithmes et structures de données 1

Dernière mise à jour effectuée le 30 Septembre 2013

Listes

1. Définition

Définition 6.1. Une **liste** est un containeur 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 containeur. 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, contenu(c): curseur → objet
- accès à la valeur du curseur, getCurseur(c) : curseur → valeur_de_curseur
- positionnement d'un curseur, setCurseur(c,valeur) : curseur X valeur_de_curseur → vide
- existence d'un élément désigné par le curseur, estVide(c) : curseur → {vrai,faux}

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

2. Liste simplement chainée

Définition 6.2. Une liste est dite **simplement chainée** 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(ref L:liste d'objet):vide;
/* positionne la clé sur le premier objet de la liste */
```

```
fonction suivant(ref 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 */
```

- 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): booleen;
  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:

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

Supprimer un élément dans la liste s'il existe

```
fonction supprimer(ref L:liste d'objet; ref 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 getCléListe et surtout setCléListe? Que déduire? Faut-il vraiment les garder?

> **Le fait d'avoir introduit ces primitives permet à l'utilisation du type abstrait de modifier la clé de type curseur et donc par la même de pouvoir rendre la structure de donnée incohérente en cas de mauvaise utilisation. Dans la suite on ne conservera pas ces primitives**

---

3. Liste doublement chainée

> **Définition 6.3.** Une liste **doublement chainée** est une liste pour laquelle les opérations en temps O(1) sont celles des listes simplement chainées auxquelles on ajoute les fonctions d'accès
>
> ```
> fonction finListe(ref L:liste d'objet):vide;
> /* positionne la clé sur le dernier objet de la liste */
> fonction précédent(ref L::liste d'objet): vide;
> /* recule la clé d'une position dans la liste */
> ```

Supprimer un élément

```
fonction supprimer(ref L:liste d'objet; ref x:objet): booleen;
  début
    si chercher(L,x) alors
      précédent(L);
      si valeur(L)!=NULL alors
        supprimerAprès(L);
      sinon
        supprimerEnTete(L)
      fin
      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.

Primitives d'accès

Ces fonctions sont immédiates.

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

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

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

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(ref 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
```

Deux primitives de modifications

```
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
```

Exercice (suite)

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

Ici, on a curseur=entier. Supposons tailleStock=1000. La séquence suivante mènera à une incohérence.

```
setCléListe(L,10000);
suivant(L);
```

Le fait d'avoir introduit ces primitives permet à l'utilisation du type abstrait de modifier la clé de type curseur et donc par la même de pouvoir rendre la structure de donnée incohérente en cas de mauvaise utilisation.

---

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.

Primitives d'accès

Ces fonctions sont immédiates.

```
fonction debutListe(ref L:listeSC d'objet):vide;
  début
    L.cle=L.premier;
  fin;
finfonction

fonction listeVide(val L:listeSC d'objet):booléen;
  début
    retourner (L.premier==NIL);
  fin
finfonction

fonction suivant(ref L:listeSC d'objet):vide;
  début
    /* la liste vide n'est pas testée car la primitive existe */
    L.cle=L.cle^.pointeurSuivant;
  fin
finfonction

fonction valeur(val L:listeSC d'objet):objet;
  début
     /* la liste vide n'est pas testée car la primitive existe */
     /* on peut donc supposer que l'utilisateur testera si la liste */
     /* est vide avant l'appel ou que la personne qui programme */
     /* s'assurera que valeur(NIL) vaut NULL                    */
      retourner cle^.valeurElement
    finsi
  fin
finfonction
```

Tris primitives de modifications

```
fonction créer_liste(ref L:listeSC d'objet):vide;
  début
    L.premier=NIL;
    L.cle=NIL;
  fin
finfonction

fonction insérerEnTete(ref L:listeSC d'objet;
                       val x:objet):vide;
  var p:curseur;
  début
    debutListe(L);
    new(p);
    p^.valeurElement=x;
    p^.pointeurSuivant-L.premier;
    L.premier=p;
    L.cle=p;
  fin
finfonction

fonction supprimerEnTete(ref L:listeSC d'objet):vide;
  var P:curseur;
  début
    debutListe(L);
    P=L.premier;
    suivant(L);
    L.premier=L.cle;
    delete(P);
  fin
finfonction

fonction insérerAprès(ref L:listeSC d'objet;val x:objet):vide;
var nouv:curseur;
  début
      new(nouv);
      nouv^.valeurElement=x;
      nouv^.pointeurSuivant=L.cle^.pointeurSuivant;
      L.cle^.pointeurSuivant=nouv;
    finsi
  fin
finfonction

fonction supprimerAprès(ref L:listeSC d'objet):vide;
  var P:curseur;
  début
    P=L.clé;
    suivant(L);
    P^.pointeurSuivant=L.cle^.pointeurSuivant;
    delete(L.cle);
    L.cle=P;
  fin
finfonction

fonction detruireListe(ref L:listeSC d'objet):vide;
  debut
    tantque !estListeVide(L) faire
      supprimerEnTete(L)
    fintantque
  fin
finfonction
```

---

6. Remarques

Avantages et inconvénients Gestion de l'espace de stockage

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

Faut-il introduire un test de validité sur chaque primitive?

La réponse est oui si on souhaite que le code soit sécurisé dès la définition des primitives. Cependant, dans ce cas, il faut prévenir le futur utilisateur de l'existence de ces tests. En effet, dans le cas contraire on pourrait avoir par exemple un test de liste vide par l'utilisateur suivi de ce même test au sein de la primitive. On aura donc une perte d'efficacité. Dans la suite de ce cours, nous n'effectuerons pas les tests au sein des primitives pour ne pas alourdir les algorithmes.