2-Listes

- Définition
- Liste simplement chainée
- Liste doublement chainée
- Implémentation par un tableau du type listeSC
- Implémentation par allocation dynamique du type listeSC

2-Listes

Définition 2.1

- 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**.

2-Listes

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 2.2 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);
/* positionne la clé sur le premier objet de la liste */

fonction suivant(ref L:liste d'objet);
/* 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 */
```

2- Liste simplement chainée

Modification

```
fonction creerListe(ref L:liste d'objet):vide;

fonction insererApres(ref L:liste d'objet, val x:objet;):vide;
/* insère un objet après  la clé, la clé ne change pas */

fonction insererEnTete(ref L:liste d'objet, val x:objet):vide;
/* insère un objet en debut de liste, la clé est positionnée sur la tête de liste */

fonction supprimerApres(ref L:liste d'objet):vide;
/* supprime  l'objet après  la clé, la clé ne change pas */
```

2- Liste simplement chainée

Modification

```
fonction supprimerEnTete(ref L:liste d'objet):vide;
/* supprime un objet en debut 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:liste d'objet):vide;
```

2- Liste simplement chainée

Détection fin de liste

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

Chercher un élément dans une liste

```
fonction chercher(ref L:liste d'objet; ref x:objet): booleen;
  debut
    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)

2- Liste simplement chainée

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 */
  debut
    debutListe(L);
    tmp=NIL;
       /*on cherche l'objet dans la liste */
    tant que !estFinListe(L) et contenu(getCléListe(L))!=x faire
      tmp= getCléListe(L);
      suivant(L);
    fintantque
       .
       .
       .
```

2- Liste simplement chainée

Supprimer un élément dans la liste s'il existe

```
fonction supprimer(ref L:liste d'objet; ref x:objet): vide;
       .
       .
       .
   /* 2 cas en sortie de la boucle tantque */
   si tmp==NIL alors /*la clé est sur la tête de liste */
      supprimerEnTete(L)
   sinon
       setCléListe(L,tmp);
       /*la clé est sur l'objet précédent l'objet à supprimer*/
      supprimerAprès(L);
    finsi
fin
finfonction
```

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

2- Liste simplement chainée

```
fonction supprimer(ref L:liste d'objet; ref x:objet): vide;
  var tmp:curseur;
  debut
    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)
    sinon
      setCléListe(L,tmp);
      supprimerAprès(L)
   finsi
  fin
finfonction
```

2- Liste simplement chainée

Exercice:
Réfléchir aux problèmes que soulèvent l'introduction de `getCléListe` et surtout `setCléListe`?

Que faut-il en déduire?
Doit-on vraiment les garder?

2- Liste doublement chainée

Définition 2.2: 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 precedent(ref L::liste d'objet): vide;
/* recule la clé d'une position dans la liste */
```

2- Liste doublement chainée

Supprimer un élément

```
fonction supprimer(ref L:liste d'objet; ref x:objet): booleen;
  debut
    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)

2- Implémentation du type liste par un tableau

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).
Pour une liste de caractère la zone de stockage peut donc être décrite par :

```
stockListe = tableau[1..tailleStock] d'elementListe ;
elementListe=structure
          valeur : car;
          indexSuivant : entier ;
finstructure;
```

2- Implémentation du type liste par un tableau

Dans ce contexte, le type curseur est un entier compris entre `1` et `tailleStock`.

Il faut coder la valeur `NIL` : on peut par exemple choisir `0`
La valeur du champ `indexSuivant` est donc un entier compris entre `0` et `tailleStock`.

Le **premier élément** doit être accessible en `O(1)`, il faut donc **conserver son index**.

2- Implémentation du type liste par un tableau

On peut donc représenter une liste par la structure suivante :

```
listeSC_Car=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.

2- Implémentation du type liste par un tableau

Primitives d'accès

Ces fonctions sont immédiates.

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

fonction suivant(ref L:listeSC_Car):vide;
  debut
    L.cle=L.vListe[L.cle].indexSuivant;
  fin
finfonction
```

2- Implémentation du type liste par un tableau

Primitives d'accès

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

2- Implémentation du type liste par un tableau

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_Car :

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

2- Implémentation du type liste par un tableau

Gestion de l'espace de stockage

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_Car):booléen;
  debut
    retourner(L.premierLibre==0);
  fin
finfonction
```

2- Implémentation du type liste par un tableau

Gestion de l'espace de stockage

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

1. **mettreCellule** : insère une cellule libre en tête de la liste des cellules libres

L'opération correspondante est de type **insererEnTete**

```
fonction mettreCellule (ref L:listeSC_Car,val
                                P:curseur):vide;
  debut
    L.vListe[P].indexSuivant=L.premierLibre;
    L.premierLibre=P;
  fin
finfonction
```

2- Implémentation du type liste par un tableau

Gestion de l'espace de stockage

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

2. **prendreCellule** : prend la cellule libre en tête de la liste des cellules libres.

L'opération correspondante est de type `supprimerEnTete`.

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

2- Implémentation du type liste par un tableau

Deux primitives de modifications

```
fonction creer_liste(ref L:listeSC_Car):vide;
  var i:curseur;
  debut
    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
```

2- Implémentation du type liste par un tableau

Deux primitives de modifications

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

2- Implémentation du type liste par un tableau

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

Dans ce choix d'implémentation 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 lors de 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.

2- Implémentation par allocation dynamique

Pointeur : Définition et syntaxe

Définition :

Un **pointeur** est une variable qui contient une adresse mémoire.

Pour **déclarer** un pointeur on écrit :

```
nom_pointeur=^type_predefini;
```

Par convention un pointeur qui ne donne accès à aucune adresse contient la valeur NIL.

Pour **accéde**r à l'emplacement mémoire désigné par le pointeur on écrit :

```
nom_pointeur^
```

2- Implémentation par allocation dynamique

Pointeur : Définition et syntaxe

La primitive **new** permet d'allouer dynamiquement de la mémoire au cours d'une exécution. On écrira:

```
new(nom_pointeur);
```

Lorsque la mémoire n'est plus utilisée par le pointeur il faut impérativement la libérer. La primitive **`delete`** permet de libérer la mémoire allouée par l'intermédiaire d'un pointeur, on écrira :

```
delete(nom_pointeur);
```

2- Implémentation par allocation dynamique

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:car;
      pointeurSuivant:^cellule;
finstructure;

curseur=^cellule;
```

2- Implémentation par allocation dynamique

La zone de stockage peut donc être décrite par :

```
cellule=structure
    valeurElement:car;
    pointeurSuivant:^cellule;
finstructure;

curseur=^cellule;

listeSC_car=structure
     premier:curseur;
     cle:curseur;
finstructure
```

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

2- Implémentation par allocation dynamique

- **Primitives d'accès**

```
fonction listeVide(val L:listeSC_car):booléen;
  debut
      retourner L.premier==NIL;
  fin
finfonction
```

listeVide est utilisée si nécessaire avant les autres.

```
fonction valeur(val L:listeSC_car):car;
 debut
      retourner L.cle^.valeurElement;
 fin
finfonction
```

2- Implémentation par allocation dynamique

- **Primitives d'accès**

```
fonction premier(val L:listeSC_car):vide;
  debut
    L.cle=L.premier;
  fin;
finfonction

fonction suivant(val L:listeSC_car):vide;
  debut
      L.cle=L.cle^.pointeurSuivant;
  fin
finfonction
```

2- Implémentation par allocation dynamique

- **Trois primitives de modifications**

```
fonction creer_liste(ref L:listeSC_Car):vide;
  debut
    L.premier=NIL;
    L.cle =NIL;
  fin
finfonction
```

2- Implémentation par allocation dynamique

- **Trois primitives de modifications**

On suppose que la clé est au debut de la liste

```
fonction supprimerEnTete(ref L:listeSC_Car):vide;
  var P:curseur;
  debut
    P=L.premier;
    suivant(L);
    L.premier=L.cle;
    delete(P);
  fin
finfonction
```

2- Implémentation par allocation dynamique

- **Trois primitives de modifications**

```
fonction insererApres(val x:car; ref :listeSC_Car):vide;
var nouv:curseur;
  debut
      new(nouv);
      nouv^.valeurElement=x;
      nouv^.pointeurSuivant=L.cle^.pointeurSuivant;
      L.cle^.pointeurSuivant=nouv;
 fin
finfonction
```

2- 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