

INF601 : Algorithmes et Structure de données

Cours 2 : TDA Liste

B. Jacob

IC2/LIUM

15 février 2010

Plan

- 1 Définition du TDA Liste
- 2 Réalisation du TDA Liste
- 3 Type de stockage des éléments
- 4 Recherche d'un élément
 - Dans une liste non triée
 - Dans une liste triée
- 5 Fin

Plan

1 Définition du TDA Liste

Les Primitives

La définition du TDA Liste

- 1 définir un ensemble d'opérations de base : les primitives
- 2 définir les opérations : fonctions que l'on fait couramment sur les listes mais
 - qui ne sont pas élémentaires
 - qui peuvent être réalisées avec les primitives

Définition des primitives

On peut déterminer que pour "manipuler" une liste il faut :

- tester si elle est vide
- accéder aux éléments
- accéder à la fin de la liste
- accéder au début de la liste
- si on est sur une position, accéder à la position suivante
- insérer un élément à une position
- supprimer un élément à une position
- créer une liste vide
- détruire une liste

Traduction en C

Primitives

- dans `lstprim.h`
- utilise les "services" de ELEMENT donc

```
#include <eltprim.h>
```

```
bool ListeVide (LISTE);  
ELEMENT ListeAcceder (POSITION, LISTE);  
POSITION ListeSentinelle (LISTE);  
POSITION ListePremier (LISTE);  
POSITION ListeSuivant (POSITION, LISTE);  
bool ListeInsérer (ELEMENT, POSITION, LISTE);  
bool ListeSupprimer (POSITION, LISTE);  
LISTE ListeCréer (void);  
void ListeDétruire (LISTE);
```

Les Opérations

On peut déterminer qu'il y a aussi des opérations courantes telles que :

- afficher une liste
- accéder à l'élément précédent
- rechercher la position (localiser) un élément
- supprimer les éléments identiques (purger)
- effacer les éléments d'une liste (RAZ)

Traduction en C

Opérations

- dans `lstop.h`
- utilise les "services" des primitives donc

```
#include <lstoprim.h>
```

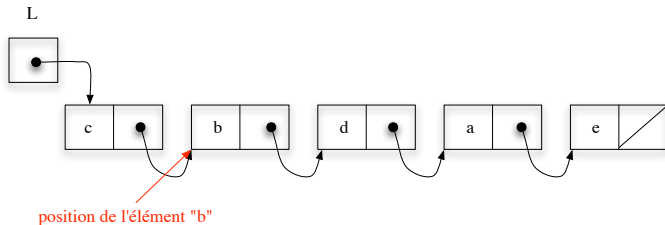
```
void ListeAfficher(LISTE L) ;  
POSITION ListePrecedent (POSITION p, LISTE L) ;  
POSITION ListeLocaliser (ELEMENT x, LISTE L);  
LISTE ListePurger (LISTE L, LISTE LL) ;  
void ListeRaz(LISTE L) ;
```


Plan

2 Réalisation du TDA Liste

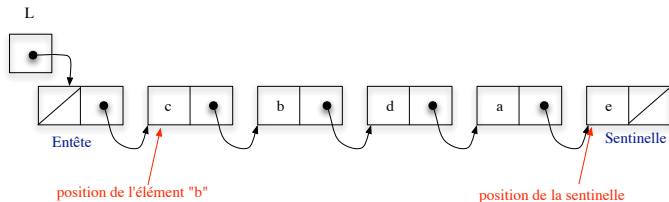
Naturelle

- 1 cellule / élément
- une liste = pointeur sur le 1^{er} élément
- fin = NULL



Avec entête/sentinelle

- 1^{ere} cellule vide (entête)
- 1 cellule / élément
- une liste = pointeur sur l'entête
- fin = position de la Sentinelle



Intérêt : insertion/suppression → gestion d'un seul pointeur

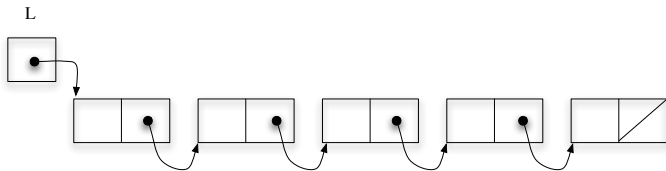
Les \neq types de chaînages

Il existe \neq types de réalisation avec chaînage :

- chaînage simple
- double chaînage
- chaînage circulaire...

Traduction en C dans [lstptr.h](#)

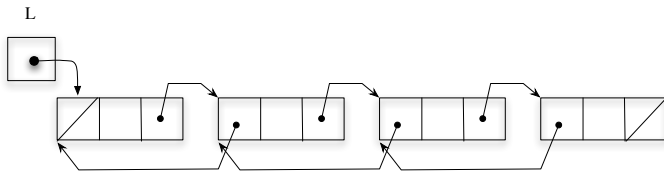
Chaînage simple



Accès :

depuis L	depuis un élément
Début : $O(1)$	Suivant : $O(1)$
Fin : $O(n)$	Précédent : $O(n)$
Milieu : $O(n)$	

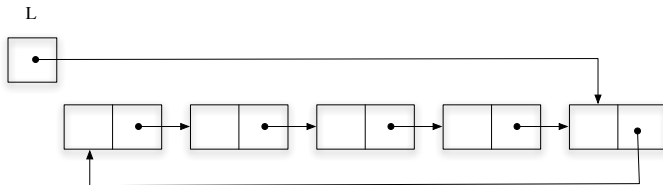
Double chaînage



Accès :

depuis L	depuis un élément
Début : $O(1)$	Suivant : $O(1)$
Fin : $O(n)$	Précédent : $O(1)$
Milieu : $O(n)$	

Chaînage circulaire



Accès :

depuis L	depuis un élément
Début : $O(1)$	Suivant : $O(1)$
Fin : $O(1)$	Précédent : $O(n)$
Milieu : $O(n)$	

Inconvénients des listes chaînées

- Si taille pointeur $>$ taille d'un élément
 - plus de place pour organisation que pour les données
 - efficacité ↘
- Si taille liste ↗
 - nombre d'allocations dynamiques ↗
 - Indirections vers des cellules dispersées ↗
 - rapidité d'accès ↘

Par cellule contiguës

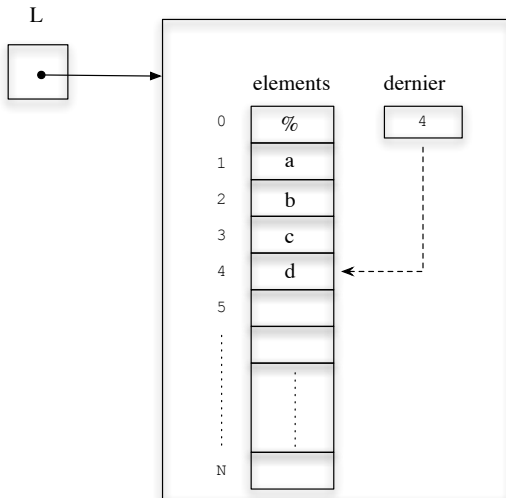
une liste = pointeur une structure qui contient

- un tableau alloué automatiquement (taille max réservée à la compilation)
- la position du dernier élément (taille réelle du tableau)

Si N listes alors N structures

Traduction en C dans [lsttab.h](#)

Par cellule contiguës



Par cellule contiguës

Intérêt :

Accès :

recherche d'un élément depuis L	depuis une position d'élément
Début : $O(1)$	Suivant : $O(1)$
Fin : $O(1)$	Précédent : $O(1)$
Milieu : $O(n)$	

Inconvénient :

- la taille de la liste n'est pas toujours adaptée à son utilisation
- si liste ↘ alors gaspillage mémoire ↗
- si liste ↗ alors risque de débordement mémoire ↘

Simulation des pointeurs en mémoire

Simulation de la mémoire dans un tableau

- **Tableau Mémoire** allouée automatiquement
- une liste
 - = @ du 1^{er} élément dans le tableau mémoire
 - = un indice
- N listes dans le même tableau Mémoire

Traduction en C dans [lstfxptr.h](#)

Par faux pointeurs

L1



L2

*Simulation de la mémoire*

	element	suivant
0	b	3
1	%	6
2		
3	c	4
4	d	-1
5	x	N
6	a	0
⋮	⋮	⋮
⋮	⋮	⋮
⋮	⋮	⋮
N	y	-1

Par faux pointeurs

Intérêt :

pallie les inconvénients des "vrais" pointeurs

Inconvénient :

si tailles des listes ↗ et/ou

si nombres des listes ↗

alors risque de débordement mémoire ↗

Traduction en C

Dans lstprim.h

```
#include <lststd.h>
```

lststd.h

- structure de donnée choisie pour réaliser les primitives du TDA liste
- Contenu :

```
/* si réalisation par cellules chainees */  
#include <lstptr.h>  
/* si réalisation par cellules contigues */  
#include <lsttab.h>  
/* si réalisation par faux pointeurs */  
#include <lstfxptr.h>
```

Plan

3 Type de stockage des éléments

Vocabulaire

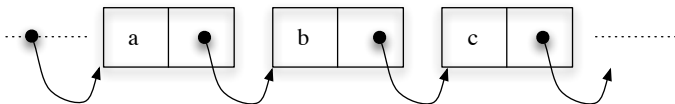
La liste (du TDA LISTE)
contient
des éléments (du TDA ELEMENT)

On dit alors que

- le TDA LISTE est le **Conteneur**
- TDA ELEMENT est le **Contenant**

Stockage direct

La valeur de l'élément est copiée dans une cellule de la liste
⇒ La copie de l'élément \in à la liste

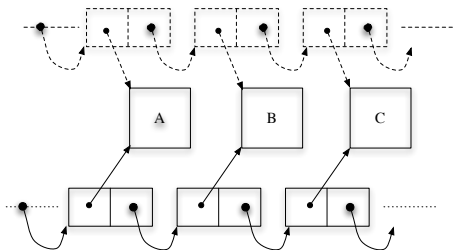


Cas d'utilisation :

- quand taille élément ↘
- quand il n'y a qu'une copie (élément \notin plusieurs listes)

Stockage indirect

La valeur de l'élément est pointée depuis une cellule de la liste
 \Rightarrow l'élément \notin à la liste



Cas d'utilisation :

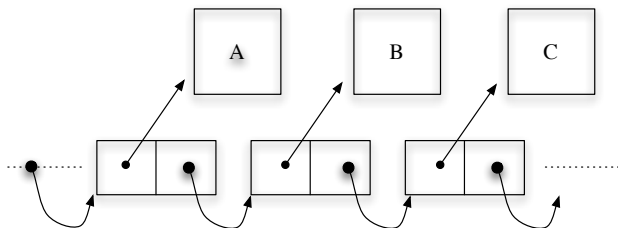
- quand taille élément \nearrow
- quand élément \in plusieurs listes

Précautions

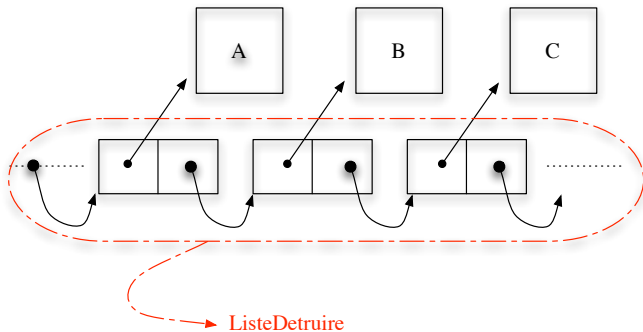
Attention à la suppression de la liste :

- Si `ListeDetruire` détruit liste + éléments \Rightarrow risque de pointeur fou
- si `ListeDetruire` détruit uniquement liste \Rightarrow risque de fuite de mémoire

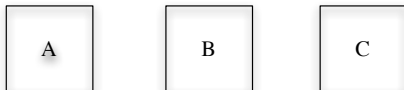
Stockage indirect/Fuite mémoire



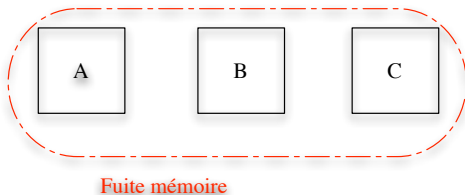
Stockage indirect/Fuite mémoire



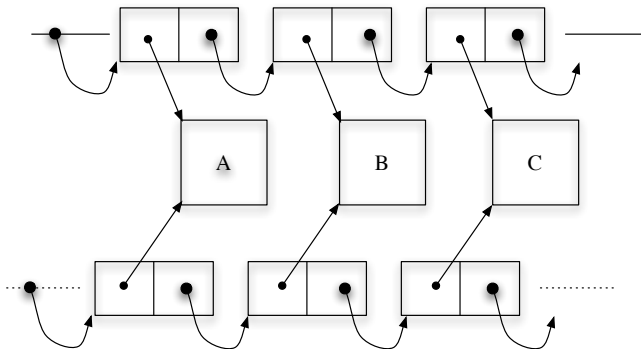
Stockage indirect/Fuite mémoire



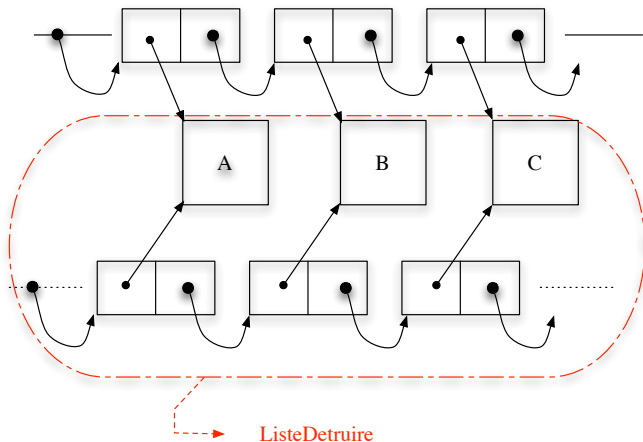
Stockage indirect/Fuite mémoire



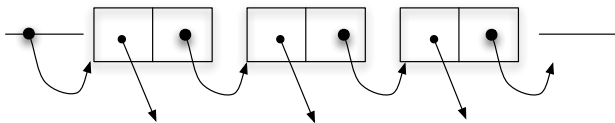
Stockage indirect/Pointeurs fous



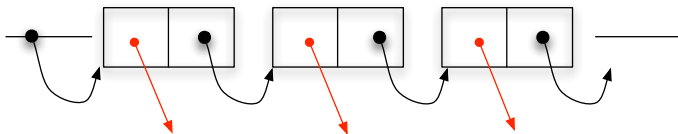
Stockage indirect/Pointeurs fous



Stockage indirect/Pointeurs fous



Stockage indirect/Pointeurs fous



Pointeurs fous

Règles

Règle : un conteneur ne supprime **QUE** ce qui lui appartient

- stockage **direct** : ListeDetruire supprime liste + éléments
- stockage **indirect** :
 - ListeDetruire supprime **uniquement** les cellules de la liste
 - si besoin, faire ListeNettoyer qui supprime liste + éléments

Plan

- 1
- 2
- 3
- 4 Recherche d'un élément**
 - Dans une liste non triée
 - Dans une liste triée

Principe

Cadre de l'étude :

- rechercher un élément dans une liste sachant sa valeur
- optimiser la fonction `ListeLocaliser`
- comment organiser les éléments de la liste ?

On peut prouver que :

- si nombre d'éléments \searrow
 \Rightarrow recherche séquentielle dans liste non triée suffit
- si nombre d'éléments \nearrow
 \Rightarrow envisager tri de la liste

Vocabulaire : si on recherche un élément X

- **recherche positive** : $X \in$ la liste
- **recherche négative** : $X \notin$ la liste

Plan

- 4 Recherche d'un élément
 - Dans une liste non triée
 - Dans une liste triée

Recherche séquentielle simple

(Avec réalisation en tableau)

```
ListeLocaliser( LISTE L , ELEMENT X)
{
    int i = 0 ;

    while( (i < L->dernier) &&
           (L->elements[i] != X) )
        i++;
    if( i == L->dernier )
        return(-1);
    return(i) ;
}
```

Recherche séquentielle avec sentinelle

On X ajoute à la position de la Sentinelle

```
ListeLocaliser( LISTE L , ELEMENT X)
{
    int i = 0 ;
    int fin = L->dernier+1 ;
    L->elements[fin] = X ;

    while( L->elements[i] != X )
        i++;
    if( i == fin )
        return(-1);
    return(i) ;
}
```

Recherche séquentielle

Rappel :

- n nombre d'éléments dans L
- accès à un élément dans une liste en $o(n)$ en moyenne

Analyse du nombre de comparaisons :

Recherche négative : $n + 1$ comparaisons

Recherche positive : si X est le k^{ieme} élément de L
alors il faut k comparaisons

- si k proche de 0, temps d'accès rapide
- si k proche de n , temps d'accès lent

Recherche séquentielle

Il faut donc que pour une majorité de recherches

- k soit proche de 0
- X soit au début de L
- mettre les éléments les plus recherchés en début de liste.

⇒ Listes autoadaptatives

Recherche auto-adaptative

Principe : réorganiser la liste des éléments au fur et à mesure des recherches

Méthodes : si X est le dernier élément recherché

- 1 Mettre X en tête de liste
(coût \searrow si réalisation par liste chaînée)
- 2 Faire avancer X d'une position
(\approx tri bulle, coût \searrow si réalisation par tableau)
- 3 Commencer la recherche à partir de la position de X
(utilisation de l'entête)

Nb comparaisons : dépend beaucoup de l'utilisation mais toujours en $O(n)$ en moyenne

Plan

- 4 Recherche d'un élément
 - Dans une liste non triée
 - Dans une liste triée

Liste triée

On suppose maintenant qu'il y a une relation d'ordre entre tous les éléments. Les éléments sont donc triés par

- ordre croissant
- ordre décroissant

Présentation de 3 méthodes de recherche :

- séquentielle
- dichotomique
- par interpolation

Recherche séquentielle : algorithme

≠ avec la liste non triée = arrêt lorsque élément de la liste $> X$

```
ListeLocaliser( LISTE L , ELEMENT X)
{
    int i = 0 ;

    while( (i < L->dernier) &&
           (L->elements[i] <= X) )
        i++;
    if(L->elements[i] == X )
        return(i);
    return(-1) ;
}
```


Recherche séquentielle : analyse

Performances :

- nombre de comparaisons légèrement meilleur que celui de la liste non triée
- tiens peu compte du fait que les éléments soient triés
- complexité de la recherche encore en $O(n)$

Meilleur *rendement* du tri :

- recherche par dichotomie
- recherche par extrapolation

Recherche par dichotomie : principe

Principe de la recherche d'un élément X dans une liste triée L :

- ① Si L est vide \rightarrow arrêt **échec**, l'élément $\notin L$
- ② $M \leftarrow L[\text{milieu}]$
- ③ Comparer X à M
 - si $X = M$ arrêt **succès** \rightarrow on a trouvé l'élément
 - si $X < M$ alors $X \notin$ moitié droite de L
 - \rightarrow on continue à chercher dans la moitié gauche de L
 - \rightarrow recommencer en 1
 - si $X > M$ alors $X \notin$ moitié gauche de L
 - \rightarrow on continue à chercher dans la moitié droite de L
 - \rightarrow recommencer en 1

\Rightarrow Se prête bien au traitement récursif

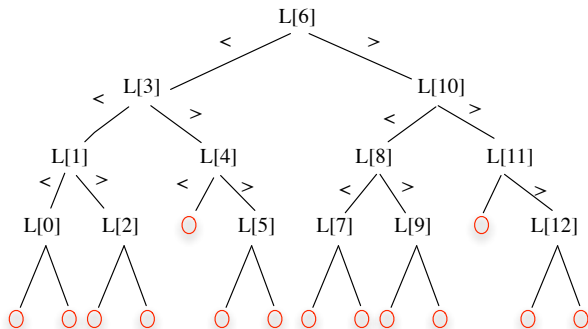
Recherche par dichotomie : algorithme

```
ListeLocaliser(LISTE L, ELEMENT X, int debut, int fin)
{
    int milieu;
    if( fin >= debut )
    {
        milieu = (debut+fin)/2 ;
        if(X == L->elements[milieu] )
            return(milieu);
        if(X < L->elements[milieu] )
            return(ListeLocaliser(L, X, debut, milieu+1));
        if(X > L->elements[milieu] )
            return(ListeLocaliser(L, X, milieu-1, fin));
    }
    else
        return(-1) ;
}
```

Recherche par dichotomie : analyse

Exemple d'une recherche sur une liste de 12 éléments

Arbre de décision :



Recherche positive : si $L[k] = X$ alors $Profondeur(L[k]) + 1$

Recherche négative : alors on arrive sur une feuille \bigcirc

$\rightarrow Profondeur(Arbre)$

Recherche par dichotomie : analyse

Complexité :

- H = Hauteur de l'arbre de décision
- nombre de comparaisons $\in [1..H]$
- Complexité = Calcul de la Hauteur = $O(\log_2 n) + 1$

Optimalité :

- beaucoup mieux que la recherche séquentielle
- Optimale pour recherche négative
- optimale pour recherche positive en moyenne

Recherche par interpolation : principe

En pratique, on tient compte de la valeur de l'élément pour déterminer la liste de recherche.

Exemple : pour rechercher le mot "Ananas" dans un dictionnaire, on ouvrira celui ci plutôt au début qu'au milieu

⇒ C'est le principe de base de la recherche par interpolation

La méthode est la même mais le calcul des limites de la liste est plus compliqué

Recherche par interpolation : algorithme

Soient :

- d = début de la liste
- f = fin de la liste
- s = séparation de la liste

Dans recherche dichotomique :

- séparation de la liste systématiquement au milieu $\forall X$
- $s \leftarrow (d + f)/2$

Dans la recherche par interpolation :

- séparation en fonction de la valeur de X
- $s \leftarrow d + ((f - d)(X - L[f] - L[d]))$ (par exemple)

Recherche par interpolation : analyse

On montre (difficilement) que la complexité = $O(\log_2(\log_2(n)))$

Exemple : pour une liste de $10^9 \dots 2^{30}$ éléments

- recherche dichotomique $O(\log_2(2^{19})) \approx 30$ comparaisons
- recherche par interpolation : ≈ 5 comparaisons à peine

Quand interpolation ?

- lorsque les éléments sont numériques (valeur)
- lorsqu'il en a un grand nombre
- lorsque les valeurs des éléments augmentent de façon à peu près constante

The End

That all folks...