

Algorithmique Structures de données

Florent Hivert

Mél : `Florent.Hivert@lri.fr`

Page personnelle : `http://www.lri.fr/~hivert`

Algorithmes et structures de données

La plupart des bons algorithmes fonctionnent grâce à une méthode astucieuse pour organiser les données. Nous allons étudier quatre grandes classes de structures de données :

- Les structures de données séquentielles (tableaux) ;
- Les structures de données linéaires (liste chaînées) ;
- Les arbres ;
- Les graphes.

Structures séquentielles : les tableaux

Structure de donnée séquentielle (tableau)

En anglais : array, vector.

Définition

Un **tableau** est une structure de donnée T qui permet de stocker un certain nombre d'éléments $T[i]$ repérés par un index i . Les tableaux vérifient généralement les propriétés suivantes :

- tous les éléments ont le même type de base ;
- le nombre d'éléments stockés est fixé ;
- l'accès et la modification de l'élément numéro i est en temps constant $\Theta(1)$, indépendant de i et du nombre d'éléments dans le tableau.

Tableau en C

- On suppose déclaré un type `elem` pour les éléments.
- Espace mémoire nécessaire au stockage d'un élément exprimé en mots mémoire (octets en général) : `sizeof(elem)`.

- définition *statique* : `elem t[taille];`

- définition *dynamique* en deux temps (déclaration, allocation) :

```
#include <stdlib.h>
```

```
elem *t;
```

```
...
```

```
t = (elem*) malloc(taille*sizeof(elem));
```

- l'adresse de `t[i]` est noté `t + i`. Calculée par

$$\text{Addr}(t[i]) = \text{Addr}(t[0]) + \text{sizeof}(elem) * i$$

Tableau en C

- On suppose déclaré un type `elem` pour les éléments.
- Espace mémoire nécessaire au stockage d'un élément exprimé en mots mémoire (octets en général) : `sizeof(elem)`.
- définition *statique* : `elem t[taille];`
- définition *dynamique* en deux temps (déclaration, allocation) :

```
#include <stdlib.h>
elem *t;
...
t = (elem*) malloc(taille*sizeof(elem));
```
- l'adresse de `t[i]` est noté `t + i`. Calculée par

$$\text{Addr}(t[i]) = \text{Addr}(t[0]) + \text{sizeof}(elem) * i$$

Tableau en C

- On suppose déclaré un type `elem` pour les éléments.
- Espace mémoire nécessaire au stockage d'un élément exprimé en mots mémoire (octets en général) : `sizeof(elem)`.
- définition *statique* : `elem t[taille];`
- définition *dynamique* en deux temps (déclaration, allocation) :

```
#include <stdlib.h>
elem *t;
...
t = (elem*) malloc(taille*sizeof(elem));
```
- l'adresse de `t[i]` est noté `t + i`. Calculée par

$$\text{Addr}(t[i]) = \text{Addr}(t[0]) + \text{sizeof}(elem) * i$$

Tableau en C

- On suppose déclaré un type `elem` pour les éléments.
- Espace mémoire nécessaire au stockage d'un élément exprimé en mots mémoire (octets en général) : `sizeof(elem)`.
- définition *statique* : `elem t[taille];`
- définition *dynamique* en deux temps (déclaration, allocation) :

```
#include <stdlib.h>
elem *t;
...
t = (elem*) malloc(taille*sizeof(elem));
```
- l'adresse de `t[i]` est noté `t + i`. Calculée par

$$\text{Addr}(t[i]) = \text{Addr}(t[0]) + \text{sizeof}(elem) * i$$

Tableau en Java

- On suppose déclaré un type `elem` pour les éléments.
- définition *dynamique* en deux temps (déclaration, allocation) :

```
elem[] t;
```

```
...
```

```
t = new elem[taille];
```

Opérations de base

Hypothèses :

- tableau de taille max_taille alloué
- éléments $0 \leq i < \text{taille} \leq \text{max_taille}$ initialisés

Retenir (Opérations de base)

- *accès au premier élément* : $\Theta(1)$
- *accès à l'élément numéro i* : $\Theta(1)$
- *accès au dernier élément* : $\Theta(1)$
- *insertion d'un élément au début* : $\Theta(\text{taille})$
- *insertion d'un élément en position i* : $\Theta(\text{taille} - i) \subset O(\text{taille})$
- *insertion d'un élément à la fin* : $\Theta(1)$

Problème de la taille maximum

On essaye d'insérer un élément dans un tableau où

$$\text{taille} = \text{max_taille}$$

Il n'y a plus de place disponible.

Comportements possibles :

- Erreur (arrêt du programme, exception)
- Ré-allocation du tableau avec recopie, coût : $\Theta(\text{taille})$

Problème de la taille maximum

On essaye d'insérer un élément dans un tableau où

$$\text{taille} = \text{max_taille}$$

Il n'y a plus de place disponible.

Comportements possibles :

- Erreur (arrêt du programme, exception)
- Ré-allocation du tableau avec recopie, coût : $\Theta(\text{taille})$

Ré-allocation (2)

Problème : On ajoute 1 par 1 n éléments.

On suppose que l'on réalloue **une case supplémentaire** à chaque débordement. Coût (nombre de copies d'éléments) :

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

Note : si on alloue des blocs de taille b , en notant $k = \lceil \frac{n}{b} \rceil$:

$$\sum_{i=1}^k bi = b \frac{k(k+1)}{2} \in \Theta(n^2)$$

La vitesse est divisée par b mais la **complexité reste la même**.

Ré-allocation (2)

Problème : On ajoute 1 par 1 n éléments.

On suppose que l'on réalloue **une case supplémentaire** à chaque débordement. Coût (nombre de copies d'éléments) :

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

Note : si on alloue des blocs de taille b , en notant $k = \lceil \frac{n}{b} \rceil$:

$$\sum_{i=1}^k bi = b \frac{k(k+1)}{2} \in \Theta(n^2)$$

La vitesse est divisée par b mais la **complexité reste la même**.

Ré-allocation par doublement de taille

Retenir (Solution au problème de la ré-allocation)

À chaque débordement, on ré-alloue $\lceil K \cdot \text{max_taille} \rceil$ où $K > 1$ est une constante fixée (par exemple $K = 2$).

- Début : tableau à 1 élément
- À la m -ième ré-allocation, la taille du tableaux : K^m
- Pour un tableau à n éléments : m est le plus petit entier tel que $K^m \geq n$, soit $m = \lceil \log_K(n) \rceil$
- Nombre de recopies d'éléments :

$$C = n + \sum_{i=1}^{m-1} K^i = n + \frac{K^m - 1}{K - 1} \in \Theta(n + K^m)$$

Finalement le coût est $\Theta(n)$.

Ré-allocation par doublement de taille

Retenir (Solution au problème de la ré-allocation)

À chaque débordement, on ré-alloue $\lceil K \cdot \text{max_taille} \rceil$ où $K > 1$ est une constante fixée (par exemple $K = 2$).

- Début : tableau à 1 élément
- À la m -ième ré-allocation, la taille du tableaux : K^m
- Pour un tableau à n éléments : m est le plus petit entier tel que $K^m \geq n$, soit $m = \lceil \log_K(n) \rceil$
- Nombre de recopies d'éléments :

$$C = n + \sum_{i=1}^{m-1} K^i = n + \frac{K^m - 1}{K - 1} \in \Theta(n + K^m)$$

Enfinement le coût est $\Theta(n)$.

Ré-allocation par doublement de taille

Retenir (Solution au problème de la ré-allocation)

À chaque débordement, on ré-alloue $\lceil K \cdot \text{max_taille} \rceil$ où $K > 1$ est une constante fixée (par exemple $K = 2$).

- Début : tableau à 1 élément
- À la m -ième ré-allocation, la taille du tableaux : K^m
- Pour un tableau à n éléments : m est le plus petit entier tel que $K^m \geq n$, soit $m = \lceil \log_K(n) \rceil$
- Nombre de recopies d'éléments :

$$C = n + \sum_{i=1}^{m-1} K^i = n + \frac{K^m - 1}{K - 1} \in \Theta(n + K^m)$$

Enfinement le coût est $\Theta(n)$.

Nombre moyen de copies

Selon la valeur de K , la constante de complexité varie de manière importante. Nombre de recopies d'éléments :

$$C = n + \sum_{i=1}^{m-1} K^i = n + \frac{K^m - 1}{K - 1} \approx n + \frac{n}{K - 1} = n \frac{K}{K - 1}$$

Quelques valeurs :

K	1.01	1.1	1.2	1.5	2	3	4	5	10
$\frac{K}{K-1}$	101	11	6	3	2	1.5	1.33	1.25	1.11

Interprétation :

- Si l'on augmente la taille de 10% à chaque étape, chaque nombre sera recopié en moyenne 11 fois.
- Si l'on double la taille à chaque étape, chaque nombre sera en moyenne recopié deux fois.

Nombre moyen de copies

Selon la valeur de K , la constante de complexité varie de manière importante. Nombre de recopies d'éléments :

$$C = n + \sum_{i=1}^{m-1} K^i = n + \frac{K^m - 1}{K - 1} \approx n + \frac{n}{K - 1} = n \frac{K}{K - 1}$$

Quelques valeurs :

K	1.01	1.1	1.2	1.5	2	3	4	5	10
$\frac{K}{K-1}$	101	11	6	3	2	1.5	1.33	1.25	1.11

Interprétation :

- Si l'on augmente la taille de 10% à chaque étape, chaque nombre sera recopié en moyenne 11 fois.
- Si l'on double la taille à chaque étape, chaque nombre sera en moyenne recopié deux fois.

Nombre moyen de copies

Selon la valeur de K , la constante de complexité varie de manière importante. Nombre de recopies d'éléments :

$$C = n + \sum_{i=1}^{m-1} K^i = n + \frac{K^m - 1}{K - 1} \approx n + \frac{n}{K - 1} = n \frac{K}{K - 1}$$

Quelques valeurs :

K	1.01	1.1	1.2	1.5	2	3	4	5	10
$\frac{K}{K-1}$	101	11	6	3	2	1.5	1.33	1.25	1.11

Interprétation :

- Si l'on augmente la taille de 10% à chaque étape, chaque nombre sera recopié en moyenne 11 fois.
- Si l'on double la taille à chaque étape, chaque nombre sera en moyenne recopié deux fois.

Bilan

Retenir (Nombre de copies)

Dans un tableau de taille n , coût de l'ajout d'un élément dans le pire des cas :

$$\text{Coût en temps} \approx n, \quad \text{Coût en espace} \approx (K - 1)n.$$

En, moyenne sur un grand nombre d'éléments ajoutés :

$$\text{Coût en temps} \approx \frac{K}{K - 1}, \quad \text{Coût en espace} \approx K.$$

*On dit que l'algorithme travaille en **temps constant amortis** (Constant Amortized Time (CAT) en anglais).*

Compromis Espace/Temps

Quand K augmente, la vitesse augmente mais la place mémoire gaspillée ($\approx (K - 1)n$) augmente aussi. Le choix de la valeur de K dépend donc du besoin de vitesse par rapport au coût de la mémoire.

Retenir

C'est une situation très classique : dans de nombreux problèmes, il est possible d'aller plus vite en utilisant plus de mémoire.

Exemple : on évite de faire plusieurs fois les mêmes calculs en stockant le résultat.

Compromis Espace/Temps

Quand K augmente, la vitesse augmente mais la place mémoire gaspillée ($\approx (K - 1)n$) augmente aussi. Le choix de la valeur de K dépend donc du besoin de vitesse par rapport au coût de la mémoire.

Retenir

C'est une situation très classique : dans de nombreux problèmes, il est possible d'aller plus vite en utilisant plus de mémoire.

Exemple : on évite de faire plusieurs fois les mêmes calculs en stockant le résultat.

Compromis Espace/Temps

Quand K augmente, la vitesse augmente mais la place mémoire gaspillée ($\approx (K - 1)n$) augmente aussi. Le choix de la valeur de K dépend donc du besoin de vitesse par rapport au coût de la mémoire.

Retenir

C'est une situation très classique : dans de nombreux problèmes, il est possible d'aller plus vite en utilisant plus de mémoire.

Exemple : on évite de faire plusieurs fois les mêmes calculs en stockant le résultat.

En pratique ...Python

On utilise `void *realloc(void *ptr, size_t size);`

Extrait des sources du langage Python

Fichier `listobject.c`, ligne 41-91 :

```
/* This over-allocates proportional to the list size, making room
 * for additional growth. The over-allocation is mild, but is
 * enough to give linear-time amortized behavior over a long
 * sequence of appends() in the presence of a poorly-performing
 * system realloc().
 * The growth pattern is: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
 */
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
```

En pratique ...Java

- Resize impossible sur un tableau : Copie.
- Mieux : utiliser `ArrayList<Elem>` :

Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

En pratique ... Oracle Java 8

Extrait des sources de Oracle java 8

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

Bilan $\text{newCapacity} = \text{oldCapacity} * 1.5$.

En pratique ...Java OpenJDK

Extrait des sources de OpenJDK 6-b27

```
public void [More ...] ensureCapacity(int minCapacity) {
    modCount++;
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        int newCapacity = (oldCapacity * 3)/2 + 1;
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}
```

Bilan $\text{newCapacity} = \text{oldCapacity} * 1.5 + 1$.

Variables Dynamiques et pointeurs

Variables dynamiques et pointeurs

Rappel : une variable usuelle est caractérisée par quatre propriétés :
(*nom, adresse, type, valeur*)

Retenir

Une **variable dynamique** est anonyme :
(*adresse, type, valeur*).

On y accède grâce à un **pointeur**.

Un **pointeur** p qui **repère** (ou **pointe vers**) une VD d'adresse a et de type T

- est de type $\uparrow T$ (lire « flèche T »);
- a pour valeur l'adresse a de la VD.

Variables dynamiques et pointeurs

les quatre propriétés d'une variable usuelle

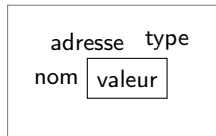
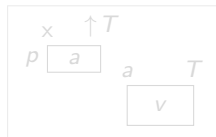
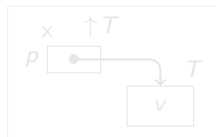


schéma pour le pointeur p qui repère la VD d'adresse a , de type T et de valeur v



le lien dynamique entre le pointeur p et la VD qu'il repère est illustré par une flèche



Variables dynamiques et pointeurs

les quatre propriétés d'une variable usuelle

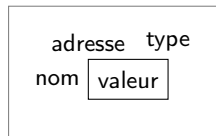
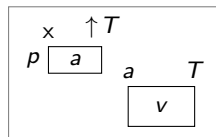
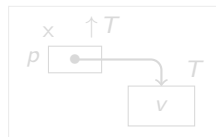


schéma pour le pointeur p qui repère la VD
d'adresse a , de type T et de valeur v



le lien dynamique entre le pointeur p et la VD
qu'il repère est illustré par une flèche



Variables dynamiques et pointeurs

les quatre propriétés d'une variable usuelle

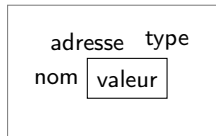
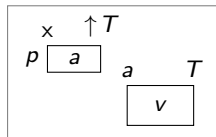
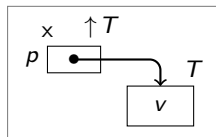


schéma pour le pointeur p qui repère la VD d'adresse a , de type T et de valeur v



le lien dynamique entre le pointeur p et la VD qu'il repère est illustré par une flèche

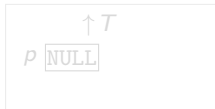


Les pointeurs nuls

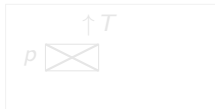
Retenir

Un pointeur p peut valoir NULL : il ne repère aucune VD.

schéma pour le pointeur p de type $\uparrow T$ qui ne repère aucune VD



l'accès à aucune VD pour le pointeur p de type $\uparrow T$ est illustré par une croix



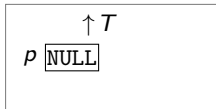
NULL est une valeur commune à tous les pointeurs, à ceux du type $\uparrow T$ comme à ceux des autres types.

Les pointeurs nuls

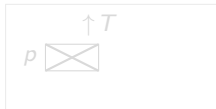
Retenir

Un pointeur p peut valoir NULL : il ne repère aucune VD.

schéma pour le pointeur p de type $\uparrow T$ qui ne repère aucune VD



l'accès à aucune VD pour le pointeur p de type $\uparrow T$ est illustré par une croix



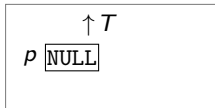
NULL est une valeur commune à tous les pointeurs, à ceux du type $\uparrow T$ comme à ceux des autres types.

Les pointeurs nuls

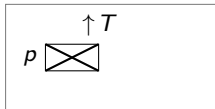
Retenir

Un pointeur p peut valoir NULL : il ne repère aucune VD.

schéma pour le pointeur p de type $\uparrow T$ qui ne repère aucune VD



l'accès à aucune VD pour le pointeur p de type $\uparrow T$ est illustré par une croix



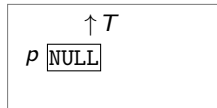
NULL est une valeur commune à tous les pointeurs, à ceux du type $\uparrow T$ comme à ceux des autres types.

Les pointeurs nuls

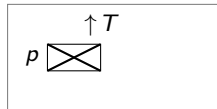
Retenir

Un pointeur p peut valoir NULL : il ne repère aucune VD.

schéma pour le pointeur p de type $\uparrow T$ qui ne repère aucune VD



l'accès à aucune VD pour le pointeur p de type $\uparrow T$ est illustré par une croix



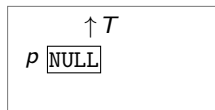
NULL est une valeur commune à tous les pointeurs, à ceux du type $\uparrow T$ comme à ceux des autres types.

Les pointeurs nuls

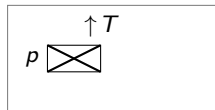
Retenir

Un pointeur p peut valoir NULL : il ne repère aucune VD.

schéma pour le pointeur p de type $\uparrow T$ qui ne repère aucune VD



l'accès à aucune VD pour le pointeur p de type $\uparrow T$ est illustré par une croix



NULL est une valeur commune à tous les pointeurs, à ceux du type $\uparrow T$ comme à ceux des autres types.

Taille des variables

La taille d'une VD de type T est celle de toute variable de type T .

La taille d'un pointeur est fixée lors de la compilation, elle ne dépend pas du type T . C'est la principale limitation quand à la mémoire disponible dans une machine.

- 32 bits, espace adressable ≤ 4 Gio (gibi-octets) $\approx 10^9$.
- 64 bits, espace adressable ≤ 16 Eio (exbi-octets) $\approx 10^{18}$.

Début de vie d'une VD

Soit p une variable du type $\uparrow T$.

Retenir

*Durant l'exécution du programme, l'action d'allocation
 $allouer(p)$*

provoque :

- 1** *la création d'une nouvelle VD de type T et de valeur indéterminée, par réservation d'une nouvelle zone de mémoire ;*
- 2** *l'affectation à p de l'adresse de cette VD.*



Début de vie d'une VD

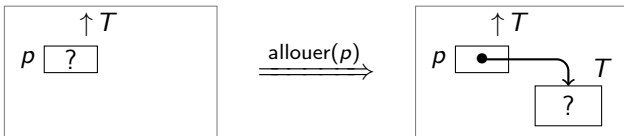
Soit p une variable du type $\uparrow T$.

Retenir

*Durant l'exécution du programme, l'action d'allocation
 $allouer(p)$*

provoque :

- 1** *la création d'une nouvelle VD de type T et de valeur indéterminée, par réservation d'une nouvelle zone de mémoire ;*
- 2** *l'affectation à p de l'adresse de cette VD.*



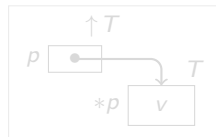
Utilisations

Soit p un pointeur du type $\uparrow T$.

Retenir

*Si p repère une VD, cette variable est notée $*p$.*

la VD de type T et de valeur v repérée par p
est notée $*p$



Toutes les opérations licites sur les variables et les valeurs de type T sont licites sur la variable $*p$ et la valeur $*p$.

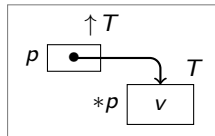
Utilisations

Soit p un pointeur du type $\uparrow T$.

Retenir

*Si p repère une VD, cette variable est notée $*p$.*

la VD de type T et de valeur v repérée par p
est notée $*p$



Toutes les opérations licites sur les variables et les valeurs de type T sont licites sur la variable $*p$ et la valeur $*p$.

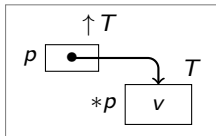
Utilisations

Soit p un pointeur du type $\uparrow T$.

Retenir

*Si p repère une VD, cette variable est notée $*p$.*

la VD de type T et de valeur v repérée par p
est notée $*p$



Toutes les opérations licites sur les variables et les valeurs de type T sont licites sur la variable $*p$ et la valeur $*p$.

Durée de vie d'une VD

Une VD existe de l'instant où elle a été créée à la fin de l'exécution du programme, sauf si elle est explicitement désallouée.

Retenir

Si le pointeur p repère une VD, l'action de désallocation explicite

désallouer(p)

*met fin à l'existence de la VD $*p$ et rend disponible l'espace mémoire qu'elle occupait.*

Après l'action de désallocation, la valeur de p est indéterminée.



Durée de vie d'une VD

Une VD existe de l'instant où elle a été créée à la fin de l'exécution du programme, sauf si elle est explicitement désallouée.

Retenir

Si le pointeur p repère une VD, l'action de désallocation explicite

désallouer(p)

*met fin à l'existence de la VD $*p$ et rend disponible l'espace mémoire qu'elle occupait.*

Après l'action de désallocation, la valeur de p est indéterminée.



Durée de vie d'une VD

Une VD existe de l'instant où elle a été créée à la fin de l'exécution du programme, sauf si elle est explicitement désallouée.

Retenir

Si le pointeur p repère une VD, l'action de désallocation explicite

désallouer(p)

*met fin à l'existence de la VD $*p$ et rend disponible l'espace mémoire qu'elle occupait.*

Après l'action de désallocation, la valeur de p est indéterminée.



Durée de vie d'une VD

Une VD existe de l'instant où elle a été créée à la fin de l'exécution du programme, sauf si elle est explicitement désallouée.

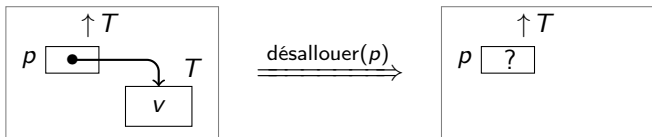
Retenir

Si le pointeur p repère une VD, l'action de désallocation explicite

désallouer(p)

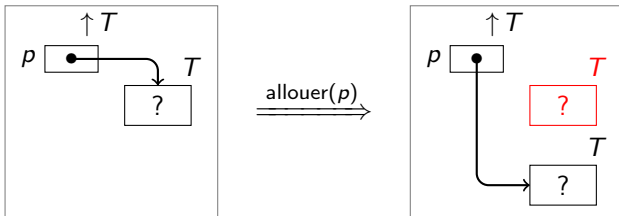
*met fin à l'existence de la VD $*p$ et rend disponible l'espace mémoire qu'elle occupait.*

Après l'action de désallocation, la valeur de p est **indéterminée**.



Double allocation, fuite de mémoire

Attention à ne pas perdre l'accès aux VD : par exemple, si on exécute deux `allouer(p)` à la suite, on ne peut plus accéder à la première VD allouée :



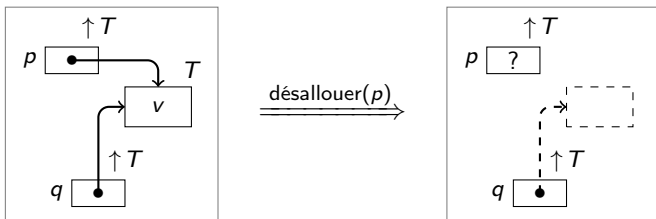
S'il n'existe pas d'autre pointeur dessus, la mémoire de la VD en rouge est perdue !

Note : certains langages (Java, Python, ...) utilisent un composant particulier appelé **ramasse miettes** (Garbage Collector en anglais) pour récupérer la mémoire ainsi perdue.

Double pointeur et désallocation

Attention aux **références fantômes** !

À la suite de l'exécution de l'action `désallouer(p)`, la valeur des éventuels autres pointeurs qui repéraient $*p$ est indéfinie.

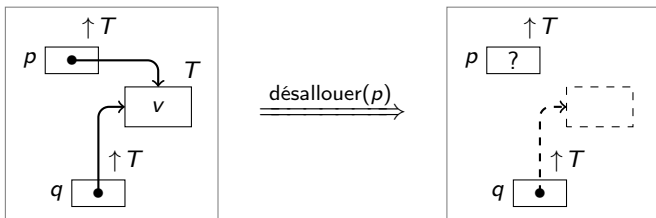


Chacune de ces valeurs constitue une référence au « fantôme » d'une VD.

Double pointeur et désallocation

Attention aux **références fantômes** !

À la suite de l'exécution de l'action désallouer(p), la valeur des éventuels autres pointeurs qui repéraient $*p$ est indéfinie.



Chacune de ces valeurs constitue une référence au « fantôme » d'une VD.

Variable dynamique en C

Retenir

- *Déclaration d'un pointeur p de type $\uparrow T$: `T *p;`*
- *Allocation d'une VD pointée par p :*
`p = (T *) malloc(sizeof(T));`
note : en cas d'échec de l'allocation malloc retourne NULL.
- *Accès à la VD et à sa valeur :*
`*p = ...; ... = (*p) + 1;`
- *Désallocation de la VD pointée par p :*
`free(p);`

Les pointeurs en C

Retenir

Tout type de pointeur

- *supporte l'opération d'affectation =*
- *peux être utilisé comme type de retour d'une fonction*
- *Note : ajout d'un pointeur à un entier (en cas de variable dynamique multiple ; tableau)*

Pour deux pointeurs de même type :

- *test d'égalité ==, de différence !=*
- *Note : comparaison <, <=, >, >= (en cas de variable dynamique multiple ; tableau)*

Le programme :

```
char *p, *q
```

```
p = (char *) malloc(sizeof(char));  
*p = 'A';  
q = (char *) malloc(sizeof(char));  
*q = 'B';  
*p = *q;  
printf("%i, %i\n", p == q, *p == *q);
```

affiche :

```
0, 1
```

Le programme :

```
char *p, *q
```

```
p = (char *) malloc(sizeof(char));
```

```
*p = 'A';
```

```
q = (char *) malloc(sizeof(char));
```

```
*q = 'B';
```

```
*p = *q;
```

```
printf("%i, %i\n", p == q, *p == *q);
```

affiche :

```
0, 1
```

Le programme :

```
char *p, *q
```

```
p = (char *) malloc(sizeof(char));  
*p = 'A';  
q = (char *) malloc(sizeof(char));  
*q = 'B';  
*p = *q;  
printf("%i, %i\n", p == q, *p == *q);
```

affiche :

```
0, 1
```


Cas des langages objets avec désallocation automatique

Exemple : Java, Python, ...

Retenir

- *variables = référence = pointeurs cachés sur var. dyn.*
- *allocation avec la commande new*
- *désallocation automatique si plus de références*
- *pas accès à l'adresse, **déréférencement automatique***
- *objet `Null` ou `None` = pointeur invalide*
- *pas d'appel de méthode sur `Null`*

Variables structurées, enregistrements

type structuré

Il est souvent pratique de regrouper logiquement plusieurs variables en une seule variable composée. On parle alors de structure ou d'enregistrement.

Retenir

*Un **type enregistrement** ou **type structuré** est un type T de variable v obtenu en juxtaposant plusieurs variables v_1, v_2, \dots ayant chacune un type T_1, T_2, \dots*

- *les différentes variables v_i sont appelées **champs** de v*
- *elles sont repérées par un **identificateur de champ***
- *si v est une variable de type structuré T possédant le champ ch , alors la variable $v.ch$ est une variable comme les autres :
(type, adresse, valeur)*

Les types structurés en algorithmique

Syntaxe

Déclaration de type structuré :

```
nom_de_type = structure:  
    nom_du_champ_1 : type_du_champ_1;  
    nom_du_champ_2 : type_du_champ_2;  
    ...
```

Déclaration de variable structurée :

```
v: nom_de_type;
```

Les types structurés en C

Syntaxe

Déclaration de type structuré :

```
struct nom_de_struct {  
    nom_du_champ_1 : type_du_champ_1;  
    nom_du_champ_2 : type_du_champ_2;  
    ...  
};
```

Déclaration de variable structurée :

```
struct nom_de_struct v;
```

Ou avec une définition de type :

```
typedef struct nom_de_struct nom_type;  
nom_type v;
```

Exemple de déclaration de type structuré

```
struct s_date {  
    char nom_jour[9]; // lundi, mardi, ..., dimanche  
    int num_jour;     // 1, 2, ..., 31  
    int mois;        // 1, 2, ..., 12  
    int annee;  
}
```

```
struct s_date date = {"vendredi", 21, 10, 2011};
```

Utilisation des types structurés

On suppose déclarées des variables v, w d'un type structuré.

On dispose donc de variables

`v.nom_du_champ_i` de type `type_du_champ_i`

Retenir

Toute opération valide sur une variable de type `type_du_champ_i` est valide sur `v.nom_du_champ_i`.

De plus, l'affectation $v = w$ est valide. Elle est équivalente aux affectations

`v.nom_du_champ_1 = w.nom_du_champ_1`

`v.nom_du_champ_2 = w.nom_du_champ_2`

...

Coût des accès :

Les champs ont toujours la même taille et sont toujours dans le même ordre.

On calcule donc l'emplacement d'un champ d'une variable en rajoutant une constante (le *Déplacement*) à l'emplacement de la variable.

Retenir

Accès à un champ en temps constant : $O(1)$.

Retenir

Du point de vue de la complexité, on peut voir les objets comme des structures avec une syntaxe particulière pour l'appel de fonction et des mécanismes de contrôle de type et de réutilisation de code amélioré (Héritage)

Coût des accès :

Les champs ont toujours la même taille et sont toujours dans le même ordre.

On calcule donc l'emplacement d'un champ d'une variable en rajoutant une constante (le *Déplacement*) à l'emplacement de la variable.

Retenir

Accès à un champ en temps constant : $O(1)$.

Retenir

Du point de vue de la complexité, on peut voir les objets comme des structures avec une syntaxe particulière pour l'appel de fonction et des mécanismes de contrôle de type et de réutilisation de code amélioré (Héritage)

Structures linéaires : les listes chaînées

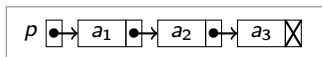
Idée

**Une variable dynamique peut elle même être
ou contenir un pointeur !**

Chaînages dynamiques

Listes Dynamiques simplement chaînées (LDSC)

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



Dans le schéma, trois types sont à distinguer :

■ le type des éléments de la suite :



■ le type des pointeurs :



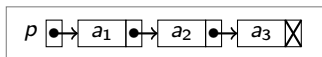
■ le type des éléments de la liste dynamique,
(composition des deux types précédents) :



Chaînages dynamiques

Listes Dynamiques simplement chaînées (LDSC)

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



Dans le schéma, trois types sont à distinguer :

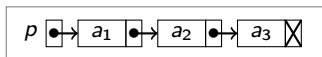
- le type des éléments de la suite :
- le type des pointeurs :
- le type des éléments de la liste dynamique, composition des deux types précédents :



Chaînages dynamiques

Listes Dynamiques simplement chaînées (LDSC)

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



Dans le schéma, trois types sont à distinguer :

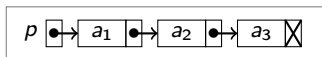
- le type des éléments de la suite :
- le type des pointeurs :
- le type des éléments de la liste dynamique, composition des deux types précédents :



Chaînages dynamiques

Listes Dynamiques simplement chaînées (LDSC)

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



Dans le schéma, trois types sont à distinguer :

- le type des éléments de la suite :
- le type des pointeurs :
- le type des éléments de la liste dynamique, composition des deux types précédents :



Déclaration des types cellule et liste

Soit `element` le type des éléments de la suite.

Retenir

Une liste chaînée est obtenue à partir du type cellule définie par

```
cellule = structure:  
    val: element  
    next : ^cellule  
liste = ^cellule
```

En C :

```
struct s_cell  
{  
    element val;  
    struct s_cell * next;  
};  
typedef struct s_cell cell; // Cellule  
typedef struct s_cell *list; // Liste Chaînée
```


Déclaration des types cellule et liste

Soit `element` le type des éléments de la suite.

Retenir

Une liste chaînée est obtenue à partir du type cellule définie par

```
cellule = structure:  
    val: element  
    next : ^cellule  
liste = ^cellule
```

En C :

```
struct s_cell  
{  
    element val;  
    struct s_cell * next;  
};  
typedef struct s_cell cell; // Cellule  
typedef struct s_cell *list; // Liste Chaînée
```

Pointeur et structure

Syntaxe

Accès aux données d'une liste chaînée :

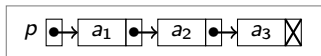
- *VD pointée par lst : *lst*
- *champ val de cell : cell.val*
- *champ val de la VD pointée par lst :*

*(*lst).val ou le raccourci lst->val*

invalide si lst vaut NULL

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

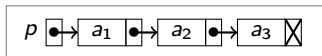
```
        p ≠ NULL ;  
        p → val = a1 ;  
        p → next ≠ NULL ;  
        p → next → val = a2 ;  
        p → next → next ≠ NULL ;  
        p → next → next → val = a3 ;  
        p → next → next → next = NULL.
```

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

```
        p(→ next)k-1 ≠ NULL,    pour 1 ≤ k ≤ 3 ;  
        p(→ next)k-1 → val = ak,    pour 1 ≤ k ≤ 3 ;  
        p(→ next)3 = NULL.
```

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

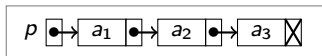
```
p ≠ NULL ;  
p → val = a1 ;  
p → next ≠ NULL ;  
p → next → val = a2 ;  
p → next → next ≠ NULL ;  
p → next → next → val = a3 ;  
p → next → next → next = NULL.
```

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

```
 $p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3 ;$   
 $p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3 ;$   
 $p(\rightarrow \text{next})^3 = \text{NULL}.$ 
```

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

$p \neq \text{NULL};$

$p \rightarrow \text{val} = a_1;$

$p \rightarrow \text{next} \neq \text{NULL};$

$p \rightarrow \text{next} \rightarrow \text{val} = a_2;$

$p \rightarrow \text{next} \rightarrow \text{next} \neq \text{NULL};$

$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} = a_3;$

$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} = \text{NULL}.$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

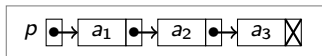
$p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3;$

$p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3;$

$p(\rightarrow \text{next})^3 = \text{NULL}.$

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

$p \neq \text{NULL};$

$p \rightarrow \text{val} = a_1;$

$p \rightarrow \text{next} \neq \text{NULL};$

$p \rightarrow \text{next} \rightarrow \text{val} = a_2;$

$p \rightarrow \text{next} \rightarrow \text{next} \neq \text{NULL};$

$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} = a_3;$

$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} = \text{NULL}.$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

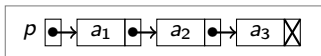
$p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3;$

$p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3;$

$p(\rightarrow \text{next})^3 = \text{NULL}.$

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

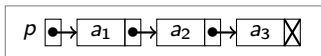
$$p \neq \text{NULL};$$
$$p \rightarrow \text{val} = a_1;$$
$$p \rightarrow \text{next} \neq \text{NULL};$$
$$p \rightarrow \text{next} \rightarrow \text{val} = a_2;$$
$$p \rightarrow \text{next} \rightarrow \text{next} \neq \text{NULL};$$
$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} = a_3;$$
$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} = \text{NULL}.$$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

$$p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3;$$
$$p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3;$$
$$p(\rightarrow \text{next})^3 = \text{NULL}.$$

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

$$p \neq \text{NULL};$$

$$p \rightarrow \text{val} = a_1;$$

$$p \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{val} = a_2;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} = a_3;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} = \text{NULL}.$$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

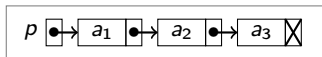
$$p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^3 = \text{NULL}.$$

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

$$p \neq \text{NULL};$$

$$p \rightarrow \text{val} = a_1;$$

$$p \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{val} = a_2;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} = a_3;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} = \text{NULL}.$$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

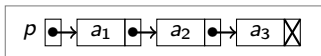
$$p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^3 = \text{NULL}.$$

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

$$p \neq \text{NULL};$$

$$p \rightarrow \text{val} = a_1;$$

$$p \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{val} = a_2;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} = a_3;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} = \text{NULL}.$$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

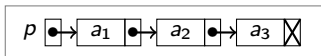
$$p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^3 = \text{NULL}.$$

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

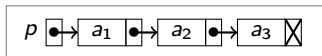
$$\begin{aligned} p &\neq \text{NULL}; \\ p \rightarrow \text{val} &= a_1; \\ p \rightarrow \text{next} &\neq \text{NULL}; \\ p \rightarrow \text{next} \rightarrow \text{val} &= a_2; \\ p \rightarrow \text{next} \rightarrow \text{next} &\neq \text{NULL}; \\ p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} &= a_3; \\ p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} &= \text{NULL}. \end{aligned}$$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

$$\begin{aligned} p(\rightarrow \text{next})^{k-1} &\neq \text{NULL}, && \text{pour } 1 \leq k \leq 3; \\ p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} &= a_k, && \text{pour } 1 \leq k \leq 3; \\ p(\rightarrow \text{next})^3 &= \text{NULL}. \end{aligned}$$

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

$$p \neq \text{NULL};$$

$$p \rightarrow \text{val} = a_1;$$

$$p \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{val} = a_2;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} = a_3;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} = \text{NULL}.$$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

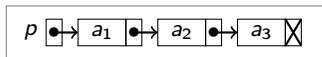
$$p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^3 = \text{NULL}.$$

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

$$p \neq \text{NULL};$$

$$p \rightarrow \text{val} = a_1;$$

$$p \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{val} = a_2;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} = a_3;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} = \text{NULL}.$$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

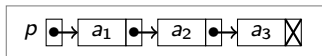
$$p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^3 = \text{NULL}.$$

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

$$p \neq \text{NULL};$$

$$p \rightarrow \text{val} = a_1;$$

$$p \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{val} = a_2;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} = a_3;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} = \text{NULL}.$$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

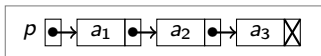
$$p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^3 = \text{NULL}.$$

Revenons à l'exemple schématisé plus haut :

le pointeur p repère la LDSC qui implante la suite $\langle a_1, a_2, a_3 \rangle$



On a :

$$p \neq \text{NULL};$$

$$p \rightarrow \text{val} = a_1;$$

$$p \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{val} = a_2;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \neq \text{NULL};$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{val} = a_3;$$

$$p \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{next} = \text{NULL}.$$

Soit, en convenant de noter « f^p » p répétitions de l'opérateur f :

$$p(\rightarrow \text{next})^{k-1} \neq \text{NULL}, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k, \quad \text{pour } 1 \leq k \leq 3;$$

$$p(\rightarrow \text{next})^3 = \text{NULL}.$$

Deux définitions équivalentes des LDSC

Définition (LDSC définie itérativement)

La suite $\langle a_1, a_2, \dots, a_n \rangle$ est implantée par la LDSC repérée par le pointeur p lorsque :

- $p(\rightarrow \text{next})^{k-1} \neq \text{NULL}$, pour $1 \leq k \leq n$
- $p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k$, pour $1 \leq k \leq n$
- $p(\rightarrow \text{next})^n = \text{NULL}$.

Définition (LDSC définie récursivement)

La suite U est implantée par la LDSC repérée par le p lorsque :

- soit $U = \langle \rangle$ et $p = \text{NULL}$;
- soit U est de la forme $U = \langle a \rangle \cdot V$ avec :
 - $p \neq \text{NULL}$
 - $p \rightarrow \text{val} = a$;
 - la suite V est implantée par la LDSC pointée par $p \rightarrow \text{next}$.

Deux définitions équivalentes des LDSC

Définition (LDSC définie itérativement)

La suite $\langle a_1, a_2, \dots, a_n \rangle$ est implantée par la **LDSC** repérée par le pointeur p lorsque :

- $p(\rightarrow \text{next})^{k-1} \neq \text{NULL}$, pour $1 \leq k \leq n$
- $p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k$, pour $1 \leq k \leq n$
- $p(\rightarrow \text{next})^n = \text{NULL}$.

Définition (LDSC définie récursivement)

La suite U est implantée par la **LDSC** repérée par le p lorsque :

- soit $U = \langle \rangle$ et $p = \text{NULL}$;
- soit U est de la forme $U = \langle a \rangle \cdot V$ avec :
 - $p \neq \text{NULL}$
 - $p \rightarrow \text{val} = a$;
 - la suite V est implantée par la **LDSC** pointée par $p \rightarrow \text{next}$.

Deux définitions équivalentes des LDSC

Définition (LDSC définie itérativement)

La suite $\langle a_1, a_2, \dots, a_n \rangle$ est implantée par la **LDSC** repérée par le pointeur p lorsque :

- $p(\rightarrow \text{next})^{k-1} \neq \text{NULL}$, pour $1 \leq k \leq n$
- $p(\rightarrow \text{next})^{k-1} \rightarrow \text{val} = a_k$, pour $1 \leq k \leq n$
- $p(\rightarrow \text{next})^n = \text{NULL}$.

Définition (LDSC définie récursivement)

La suite U est implantée par la **LDSC** repérée par le p lorsque :

- soit $U = \langle \rangle$ et $p = \text{NULL}$;
- soit U est de la forme $U = \langle a \rangle \cdot V$ avec :
 - $p \neq \text{NULL}$
 - $p \rightarrow \text{val} = a$;
 - la suite V est implantée par la LDSC pointée par $p \rightarrow \text{next}$.

Les notions sur les suites finies passent aux listes dynamiques qu'elles implantent : longueur, concaténation, position...

Calcul itératif :

```
int longueur(list lst) {
    int k = 0;
    list q = lst;
    while (q != NULL) {
        q = q->next; k++;
    }
    return k;
}
```

Calcul récursif :

```
int longueur(list lst) {
    if (lst == NULL) return 0;
    else return 1 + longueur(lst->next);
}
```

Les notions sur les suites finies passent aux listes dynamiques qu'elles implantent : longueur, concaténation, position...

Calcul itératif :

```
int longueur(list lst) {
    int k = 0;
    list q = lst;
    while (q != NULL) {
        q = q->next; k++;
    }
    return k;
}
```

Calcul récursif :

```
int longueur(list lst) {
    if (lst == NULL) return 0;
    else return 1 + longueur(lst->next);
}
```

Les notions sur les suites finies passent aux listes dynamiques qu'elles implantent : longueur, concaténation, position...

Calcul itératif :

```
int longueur(list lst) {
    int k = 0;
    list q = lst;
    while (q != NULL) {
        q = q->next; k++;
    }
    return k;
}
```

Calcul récursif :

```
int longueur(list lst) {
    if (lst == NULL) return 0;
    else return 1 + longueur(lst->next);
}
```

Définition (mode de programmation constructif)

*Dans le mode de programmation **constructif** (ou **pure**), les opérations laissent intacts leurs arguments.*

insertion en tête d'une LDSC, en mode constructif :

au
début



à la fin



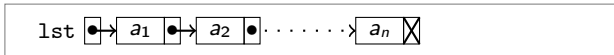
```
list insertion_en_tete(element x, list lst) {  
    list res = alloue_cellule();  
    res->val = x;  
    res->next = lst;  
    return res;  
}
```

Définition (mode de programmation constructif)

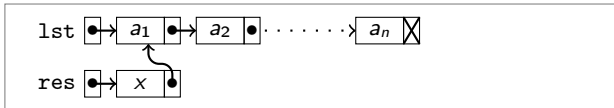
*Dans le mode de programmation **constructif** (ou **pure**), les opérations laissent intacts leurs arguments.*

insertion en tête d'une LDSC, en mode constructif :

au
début



à la fin



```
list insertion_en_tete(element x, list lst) {  
    list res = alloue_cellule();  
    res->val = x;  
    res->next = lst;  
    return res;  
}
```

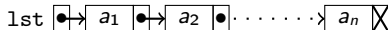
Définition (mode de programmation constructif)

*Dans le mode de programmation **constructif** (ou **pure**), les opérations laissent intacts leurs arguments.*

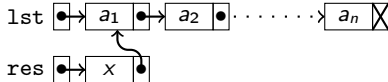
insertion en tête d'une LDSC, en mode constructif :

au

début



à la fin



```
list insertion_en_tete(element x, list lst) {  
    list res = alloue_cellule();  
    res->val = x;  
    res->next = lst;  
    return res;  
}
```


Définition (mode de programmation mutatif)

Dans le mode de programmation **mutatif** (ou **avec mutation**), ou **avec modification**), les opérations **modifient** leurs arguments.

Il faut donc passer un **pointeur ou une référence** vers l'argument à modifier.

insertion en tête d'une LDSC, en mode mutatif

au
début



à la fin



```
void insertion_en_tete(element x, list *plst) {  
    list tmp = alloue_cellule();  
    tmp->val = x;  
    tmp->next = *plst;  
    *plst = tmp;  
}
```

Définition (mode de programmation mutatif)

Dans le mode de programmation **mutatif** (ou avec **mutation**), ou avec **modification**), les opérations **modifient** leurs arguments. Il faut donc passer un **pointeur ou une référence** vers l'argument à modifier.

insertion en tête d'une LDSC, en mode mutatif

au
début



à la fin



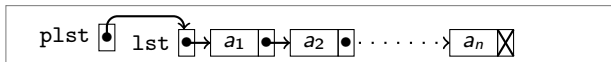
```
void insertion_en_tete(element x, list *plst) {  
    list tmp = alloue_cellule();  
    tmp->val = x;  
    tmp->next = *plst;  
    *plst = tmp;  
}
```

Définition (mode de programmation mutatif)

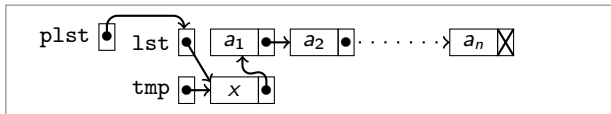
Dans le mode de programmation *mutatif* (ou avec *mutation*), ou avec *modification*), les opérations **modifient** leurs arguments. Il faut donc passer un **pointeur ou une référence** vers l'argument à modifier.

insertion en tête d'une LDSC, en mode mutatif

au
début



à la fin



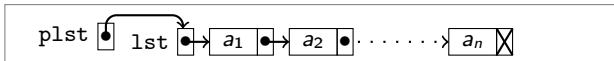
```
void insertion_en_tete(element x, list *plst) {  
    list tmp = alloue_cellule();  
    tmp->val = x;  
    tmp->next = *plst;  
    *plst = tmp;  
}
```

Définition (mode de programmation mutatif)

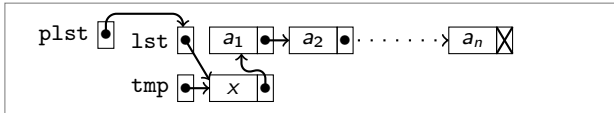
Dans le mode de programmation *mutatif* (ou avec *mutation*), ou avec *modification*), les opérations **modifient** leurs arguments. Il faut donc passer un **pointeur ou une référence** vers l'argument à modifier.

insertion en tête d'une LDSC, en mode mutatif

au
début



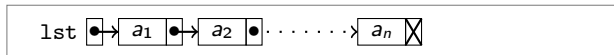
à la fin



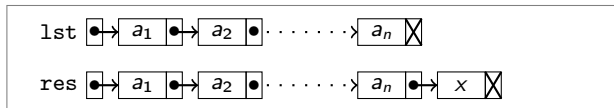
```
void insertion_en_tete(element x, list *plst) {  
    list tmp = alloue_cellule();  
    tmp->val = x;  
    tmp->next = *plst;  
    *plst = tmp;  
}
```

insertion en queue d'une LDSC, constructive, récursive

au
début

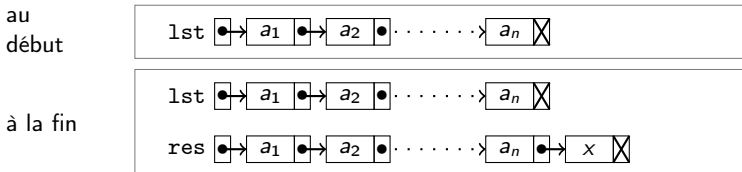


à la fin



```
list insertion_en_queue(element x, list lst) {  
    list res = alloue_cellule();  
    if (lst == NULL) {  
        res->val = x; res->next = NULL;  
        return res;  
    } else {  
        res->val = lst->val;  
        res->next = insertion_en_queue(x, lst->next);  
        return res  
    }  
}
```

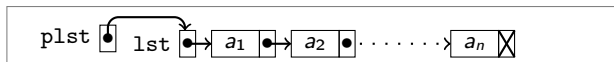
insertion en queue d'une LDSC, constructive, récursive



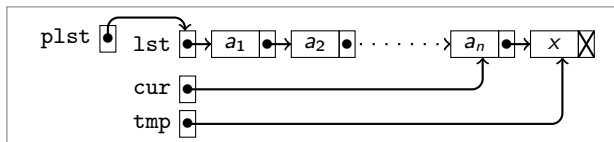
```
list insertion_en_queue(element x, list lst) {
    list res = alloue_cellule();
    if (lst == NULL) {
        res->val = x; res->next = NULL;
        return res;
    } else {
        res->val = lst->val;
        res->next = insertion_en_queue(x, lst->next);
        return res
    }
}
```

insertion en queue d'une LDSC, mutative, itérative

au
début

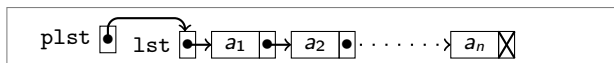


à la fin

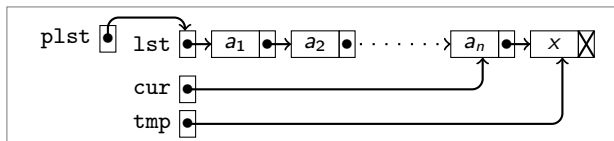


```
void insertion_en_queue(element x, list *plst) {  
    list tmp = alloue_cellule();  
    tmp->val = x; tmp->next = NULL;  
    if (*plst == NULL) *plst = tmp;  
    else {  
        list cur = *plst;  
        while (cur->next != null) cur = cur->next;  
        cur->next = tmp;  
    }  
}
```

insertion en queue d'une LDSC, mutative, itérative

au
début

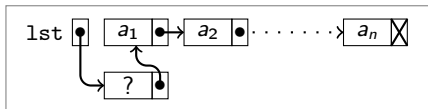
à la fin



```
void insertion_en_queue(element x, list *plst) {
    list tmp = alloue_cellule();
    tmp->val = x; tmp->next = NULL;
    if (*plst == NULL) *plst = tmp;
    else {
        list cur = *plst;
        while (cur->next != null) cur = cur->next;
        cur->next = tmp;
    }
}
```


Technique : LDSC avec une fausse tête

le pointeur p repère la LDSC avec une fausse tête qui implante la suite $\langle a_1, a_2, \dots, a_n \rangle$

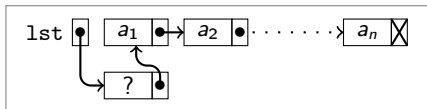


Intérêts :

- en mode avec mutation ;
- évite d'avoir à distinguer les cas de l'élément de tête ou de la liste vide dans les opérations d'insertion et de suppression.

Technique : LDSC avec une fausse tête

le pointeur p repère la LDSC avec une fausse tête qui implante la suite $\langle a_1, a_2, \dots, a_n \rangle$

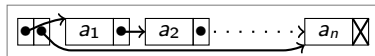


Intérêts :

- en mode avec mutation ;
- évite d'avoir à distinguer les cas de l'élément de tête ou de la liste vide dans les opérations d'insertion et de suppression.

Technique : LDSC avec pointeurs de tête et de queue

pointeurs repérant
respectivement la tête et la
queue de la LDSC qui implante
la suite $\langle a_1, a_2, \dots, a_n \rangle$



Intérêts :

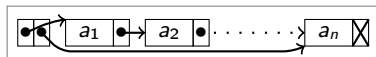
- en mode avec mutation ;
- ajout en queue sans parcours de la LDSC ;
- concaténation sans parcours des LDSC.

À définir, après la déclaration du type Liste, par :

```
struct s_list_tq {  
    struct s_cell *first, *last;  
}
```

Technique : LDSC avec pointeurs de tête et de queue

pointeurs repérant
respectivement la tête et la
queue de la LDSC qui implante
la suite $\langle a_1, a_2, \dots, a_n \rangle$



Intérêts :

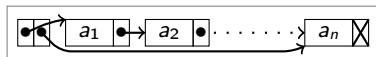
- en mode avec mutation ;
- ajout en queue sans parcours de la LDSC ;
- concaténation sans parcours des LDSC.

À définir, après la déclaration du type Liste, par :

```
struct s_list_tq {  
    struct s_cell *first, *last;  
}
```

Technique : LDSC avec pointeurs de tête et de queue

pointeurs repérant
respectivement la tête et la
queue de la LDSC qui implante
la suite $\langle a_1, a_2, \dots, a_n \rangle$



Intérêts :

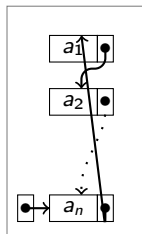
- en mode avec mutation ;
- ajout en queue sans parcours de la LDSC ;
- concaténation sans parcours des LDSC.

À définir, après la déclaration du type Liste, par :

```
struct s_list_tq {  
    struct s_cell *first, *last;  
}
```

Technique : LDSC circulaire

pointeur repérant la queue de la LDSC circulaire qui implante la suite $\langle a_1, a_2, \dots, a_n \rangle$

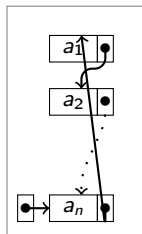


Intérêts :

- en mode avec mutation ;
- ajout en queue et suppression en tête sans parcours de la LDSC ;
- concaténation sans parcours des LDSC.

Technique : LDSC circulaire

pointeur repérant la queue de la LDSC circulaire qui implante la suite $\langle a_1, a_2, \dots, a_n \rangle$

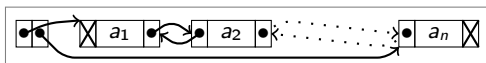


Intérêts :

- en mode avec mutation ;
- ajout en queue et suppression en tête sans parcours de la LDSC ;
- concaténation sans parcours des LDSC.

Listes dynamiques doublement chaînées (LDDC)

pointeurs repérant
respectivement la
tête et la queue de
la LDDC qui
implante la suite
 $\langle a_1, a_2, \dots, a_n \rangle$

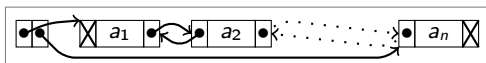


Intérêts :

- en mode avec mutation ;
- marches avant et arrière ;
- ajout en queue et suppression en tête sans parcours de la LDDC ;
- concaténation sans parcours des LDDC.

Listes dynamiques doublement chaînées (LDDC)

pointeurs repérant
respectivement la
tête et la queue de
la LDDC qui
implante la suite
 $\langle a_1, a_2, \dots, a_n \rangle$



Intérêts :

- en mode avec mutation ;
- marches avant et arrière ;
- ajout en queue et suppression en tête sans parcours de la LDDC ;
- concaténation sans parcours des LDDC.

Listes dynamiques doublement chaînées (LDDC) (2)

La structure peut être déclarée par :

```
struct s_cell
{
    element val;
    struct s_cell * next;
    struct s_cell * prev;
};

struct s_lddc
{
    struct s_cell * first;
    struct s_cell * last;
};
```

Applications

Implantation des types de données abstraits :

- piles (dernier entré - premier sorti)
- files d'attente (premier entré - premier sorti)
- double-queues
- ensembles, tables d'associations

Algorithmes de retour sur trace, essais/erreurs (backtracking)

Applications

Implantation des types de données abstraits :

- piles (dernier entré - premier sorti)
- files d'attente (premier entré - premier sorti)
- double-queues
- ensembles, tables d'associations

Algorithmes de retour sur trace, essais/erreurs (backtracking)

Application : exemple d'algorithme essais/erreurs

Problème (liste de candidats)

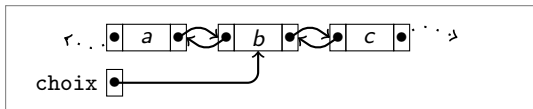
*On veut maintenir une **liste ordonnée de candidats***

qui supporte les opérations suivantes :

- 1** *choix d'un candidat avec **suppression** dans la liste (chaque candidat ne peut être choisi qu'une seule fois)*
- 2** *retour en arrière par **re-insertion du candidat** à sa place pour choisir un autre candidat*

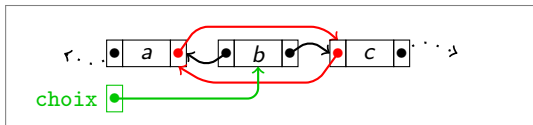
Retour en arrière dans une LDDC

Liste des
candidats ;
sélection de *b*



```
choix->next->prev = choix->prev;  
choix->prev->next = choix->next;
```

Suppression de
b de la liste

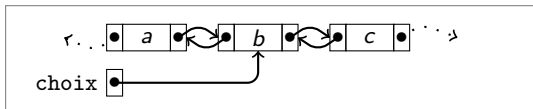


Le pointeur *choix* contient toute l'information nécessaire pour revenir à la position de départ ; il suffit de faire :

```
choix->next->prev = choix;  
choix->prev->next = choix;
```

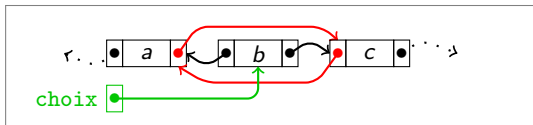
Retour en arrière dans une LDDC

Liste des
candidats ;
sélection de *b*



```
choix->next->prev = choix->prev;  
choix->prev->next = choix->next;
```

Suppression de
b de la liste



Le pointeur *choix* contient toute l'information nécessaire pour revenir à la position de départ ; il suffit de faire :

```
choix->next->prev = choix;  
choix->prev->next = choix;
```

Tables d'associations

Tables d'associations

Définition (table)

Une **table** est une fonction d'un ensemble d'entrées dans un ensemble de valeurs.

C'est un objet informatique : il s'agit de ranger une information (valeur) associée à chaque index (entrée), de retrouver l'information à partir de l'index, de la modifier ou de supprimer couple index-information.

Exemple : la table des numéros de département

La table NumDep : Chaîne \rightarrow Chaîne fait correspondre à tout département français son numéro :

- $\text{Dom}(\text{NumDep}) = \{ \text{"Ain"}, \text{"Aisne"}, \dots, \text{"Yvelines"} \}$;
- $\text{Im}(\text{NumDep}) = \{ \text{"01"}, \dots, \text{"2A"}, \text{"2B"}, \dots, \text{"974"} \}$;
- $\text{NumDep}(\text{"Martinique"}) = \text{"972"}$;
- $\text{"Informatique"} \notin \text{Dom}(\text{NumDep})$. Autrement dit $\text{NumDep}(\text{"Informatique"})$ n'est pas défini.