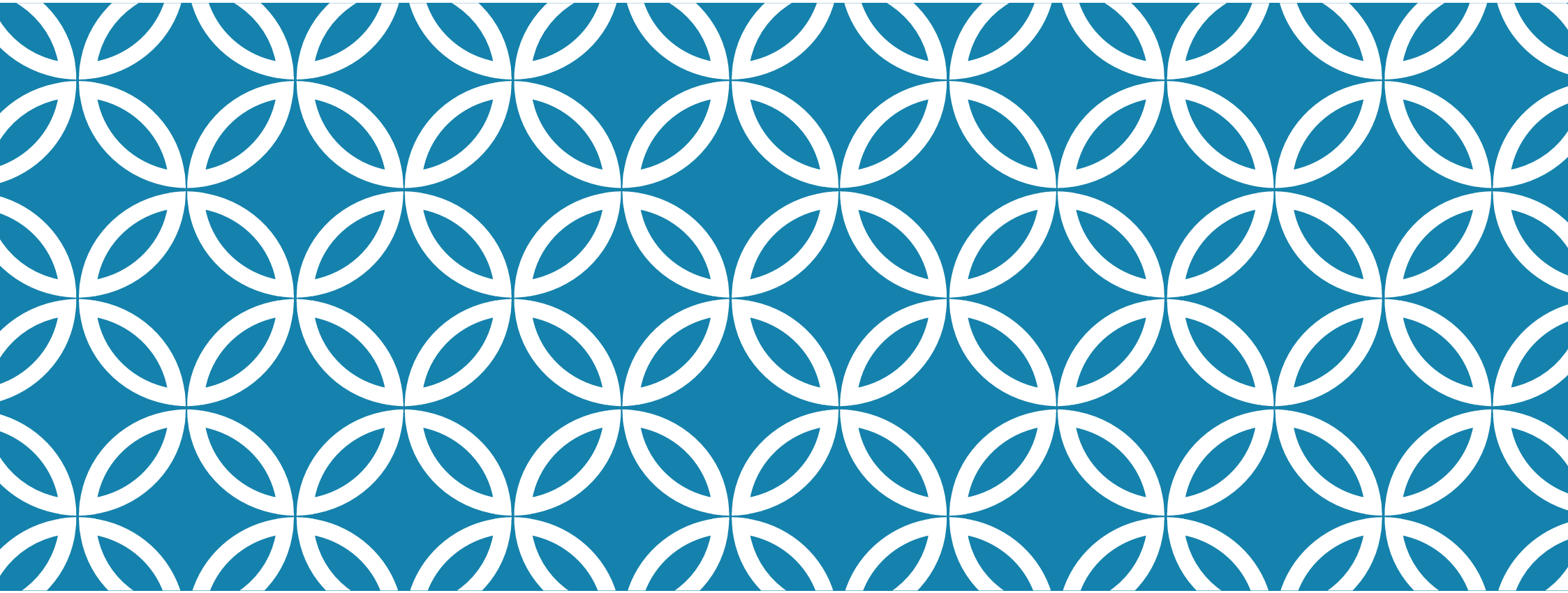


CHAPITRE 4 : LES LISTES CHAÎNÉES

Par Aïcha El Golli
aicha.elgolli@essai.ucar.tn

INTRODUCTION

✓Possible d'allouer de la mémoire à l'exécution (normalement la mémoire est allouée statiquement à la compilation).

✓Intérêts :

– Gestion Optimale de la mémoire

• Réservation d'une taille précise de mémoire quand on en a besoin

• Libération de la mémoire quand on ne s'en sert plus

– Possibilité de définir des listes d'éléments de taille variable (≠ des tableaux qui sont de taille fixe)

✓L'espace mémoire occupé par une structure de données dynamique est variable. C'est intéressant pour représenter des ensembles à tailles variables. On peut donc agrandir ou rétrécir la taille de l'ensemble durant l'exécution du programme. Certains problèmes nécessitent la gestion d'un ensemble dynamique.

Ex: trouver tous les nombres premiers <= n (avec n un paramètre donné) et les stocker en mémoire.

✓Le problème ici est la taille de la structure utilisée pour sauvegarder les nombres premiers. Un tableau ne conviendrait pas car on n'a aucune idée sur la taille à réserver au départ.

VARIABLES DYNAMIQUES ET POINTEURS

✓**Rappel :** une variable usuelle est caractérisée par quatre propriétés : (*nom*, *adresse*, *type*, *valeur*)
Retenir : *Une* ***variable dynamique*** **(VD)** ***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*

✓**Pointeurs nuls :**Un pointeur p peut valoir NULL : il ne repère aucune VD. NULL est une valeur commune à tous les pointeurs.

✓ ***Remarques :***

** Les constantes pointeurs NIL (en Pascal) ou bien NULL ou 0 (en C) indiquent l'absence d'adresse. Donc, par exemple en C, l'affectation p = NULL, veut dire que p ne pointe aucune variable.*

** Il ne faut jamais utiliser l'indirection (^ en pascal ou * en C) avec un pointeur ne contenant pas l'adresse d'une variable, il y aura alors une erreur de segmentation. La taille d'une VD de type* T *est celle de toute variable de type* T

ALLOCATION DE VARIABLES

Allocation de variables veut dire **création de variables**➔ réservation d'espace mémoire en associant à chaque variable l'adresse d'une zone vide en mémoire.

Il existe 2 types d'allocation : **statique** (gérée automatiquement par le système) et **dynamique** (gérée manuellement par le programmeur).

Toutes les variables déclarées représentent des variables allouées statiquement :

- Les variables globales d'un programme C sont automatiquement créées au début de l'exécution et détruites à la fin de l'exécution.
- Les variables d'une procédure ou fonction (ainsi que les paramètres d'appels) sont automatiquement créées au début de chaque appel et détruites à chaque retour de procédure ou fonction.

ALLOCATION DYNAMIQUE

On alloue de la mémoire quand on en a besoin et on peut la libérer quand on n'a plus besoin. L'espace mémoire alloué est dans une zone dynamique (on peut atteindre 1 meg dépendant du modèle de mémoire).

Cette zone est plus grande que la zone statique.

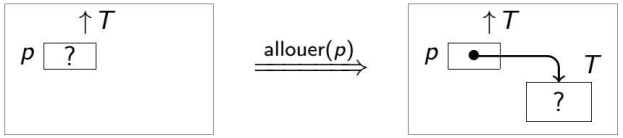
Retenir

*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.*

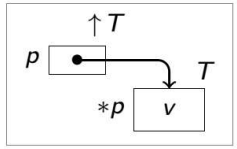


Soit p un pointeur du type $\uparrow T$.

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 $*p$.

ALLOCATION DYNAMIQUE:

langage C

- Il existe des fonctions prédéfinies en C pour **créer** de nouvelles **variables durant l'exécution** d'un programme et pour les **détruire** aussi, c'est l'allocation dynamique :
- En C, l'allocation de la mémoire se fait grâce la **fonction malloc()** de la bibliothèque **<stdlib.h>** :
- la fonction **malloc(nb_octets) alloue** une zone mémoire de taille nb_octets et **retourne son adresse** comme résultat.
- Synthaxe void * malloc(size_t size); Fonction retournant un pointeur sur un espace mémoire réservé à un élément de taille size ou bien NULL si l'allocation est impossible. La mémoire allouée **n'est pas initialisée.**
- Réservation de Mémoire pour un Elève :

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

- La **fonction free(p) détruit** la variable pointée par p.

Algorithmique

- Si p : pointeur sur T;
- allouer(p): **alloue** une zone mémoire (une variable dynamique) et affecte à p **son adresse** ou bien NULL si allocation impossible
- et **libérer(p)** ou **désallouer (p) détruit** la variable pointée par p

ALLOCATION DYNAMIQUE

```
exemple en C :
int main()
{
char *p;                    /* allocation statique d'une var ( p ) de type pointeur */
p = malloc( sizeof(char) );       /* allocation dynamique d'une var de même taille
                                  qu'un caractère : sizeof( type ) retourne le
                                nb d'octets nécessaire pour représenter une var de ce type */
*p = 'A';                                  /* utilisation indirecte de la var dynamique */
free(p);                                    /* destruction de la var dynamique */
return 0;
}
```

ALLOCATION DYNAMIQUE

```
void main()
{
int *p;       /* allocation statique d'une var ( p ) de type pointeur */
              /* le contenue de p est pour le moment indéterminé :
                 une adresse quelconque de la mémoire*/
*p = 10; /* erreur à l'exécution ; affectation d'un entier (10) dans une zone indéterminée, p ne pointe aucune variable de type entier */
return 0;
}
```

DURÉE DE VIE D'UNE VD

- Une VD existe de l'instant où elle a été créée et doit être 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.*

- En C, la libération de Mémoire se fait par la fonction free()

Syntaxe : void free(void * p); Libère l'espace mémoire pointé par p; elle ne fait rien si p vaut NULL. p doit être un pointeur sur un espace mémoire alloué par malloc,

DESTRUCTION D'UNE VD

➢ **ATTENTION:** Après l'action de désallocation, la valeur de *p* est indéterminée, il est interdit de référencer une variable dynamique après l'avoir détruite :

```
int main()
{
int *p;          /* allocation statique d'une var ( p ) de type pointeur */
p = (int *)malloc( sizeof(int) );
/* Le contenu de p est l'adresse d'une variable dynamique nouvellement créée */
*p = 10;
/*l'affectation est correcte, la zone concernée contient une variable de type entier */
free(p);   /* destruction de la var dynamique maintenant p pointe vers une zone
indéterminée */
*p=20; /*erreur à l'exécution : affectation d'un entier (20) dans une zone
indéterminée*/
}
```

À RETENIR…

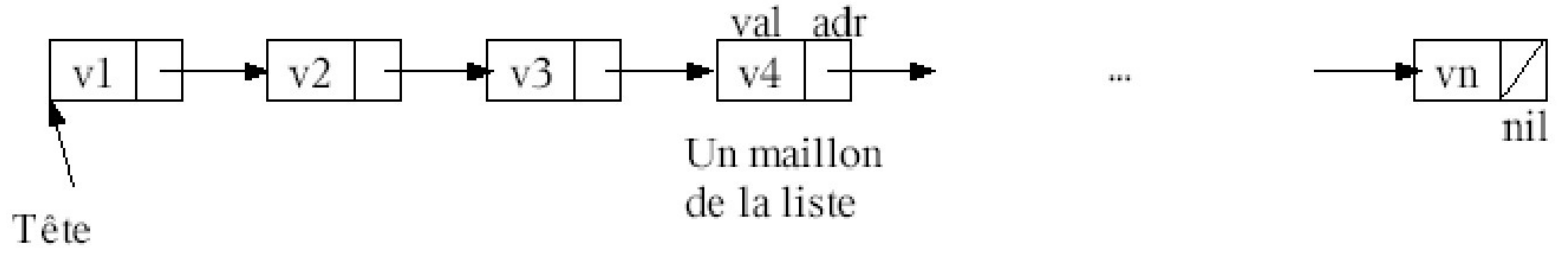
- ✓Les variables dynamiques peuvent être de n'importe quel type, simple ou complexe
- ✓Une **variable dynamique** n'a **pas de nom**, on ne peut la **manipuler qu'à travers un pointeur qui contiendrait son adresse**

LES LISTES CHAÎNÉES

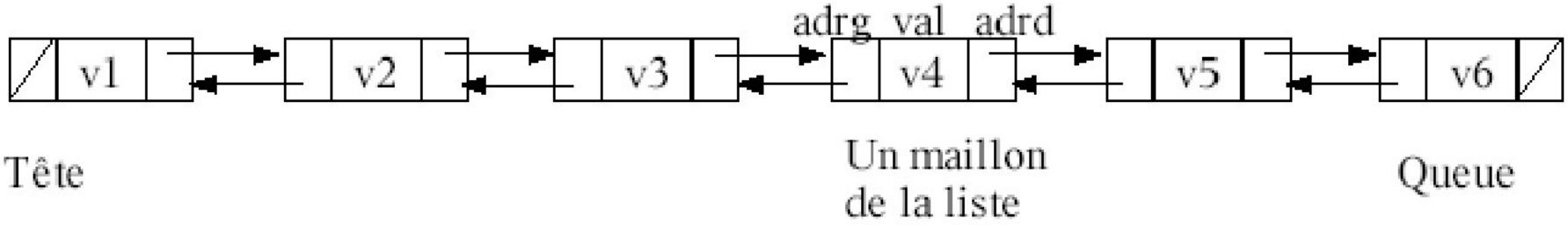
- Les listes chaînées sont des structures de données à accès indirect.
- Elles permettent de construire une suite d'éléments de même type ou plus rarement de types différents.
- L'avantage des listes est le fait qu'elles sont dynamiques d'une part. En effet, on réserve la mémoire pour chaque maillon de la chaîne, donc si la taille de la liste varie beaucoup on prend moins de ressource inutilement.
- Le deuxième avantage est le fait que l'on peut facilement ajouter des éléments au milieu de la liste. Dans un tableau, il faut d'abord déplacer les éléments du tableau pour faire un « trou ». De même, on peut facilement enlever des éléments dans une liste.
- On utilise le même principe que les listes pour plusieurs structures de données complexes.

LES LISTES CHAÎNÉES

Listes Dynamiques simplement chaînées (LDSC)



Listes dynamiques doublement chaînées (LDDC)



REPRÉSENTATION D'UNE CELLULE OU NŒUD D'UNE LISTE

On définie un type cellule/nœud d'un liste:

Type Structure cellule
donnee: <type de données>; {type simple ou structure}
suivant : pointeur vers cellule;
fin

```
typedef struct Cel{
<type> donnee  ;
struct Cel* suivant  ;
}cellule ;
```

LES LISTES SIMPLEMENT CHAINÉES

ALGO

```
Type Structure cellule
donnee: entier ;
suivant : pointeur vers cellule;
 fin

Type Liste =pointeur vers cellule;
```

C

```
typedef struct Cel{
int donnee  ;
struct Cel* suivant  ;
}cellule ;

typedef  cellule* Liste ;
```

Algo	C

Algorithme LC
L :Liste ;
rep,i, pos : entier
Début
Rep<-1 ;
creerListe(L) ;
ajoutDeb(L, 23) ;
ajoutFin(L, 6) ;
tantque(rep=1)**Faire**
afficher(entrer un entier) ;
entrer(i) ;
afficher("1- en debut, 2-en fin ?") ;
entrer(pos) ;
si(pos=2) **alors**
ajoutFin(L,i) ;
sinon
ajoutDeb(L,i) ;
FinSi ;
afficher("voulez vous continuer? tapez 1 pour continuer et 0 pour arreter ");
entrer(rep) ;
FinTQ
	`void main()`
	`{`
	`  Liste L ;`
	`  int rep=1, i, pos;`
	`  creerListe(&L);`
	`  ajoutDeb(&L,23);`
	`  ajoutFin(&L,6);`
	`  while(rep){`
	`  printf("entrer un entier ");`
	`  scanf("%d",&i);`
	`  printf("\n 1-en debut, 2-en fin ? ");`
	`  scanf("%d", &pos);`
	`  if(pos==2)`
	`  ajoutFin(&L,i);`
	`  else`
	`  ajoutDeb(&L,i);`
	`  printf("\n voulez vous continuer? tapez 1 pour continuer et 0 pour arreter ");`
	`  scanf("%d", &rep);`
	`  }`

Algo	C

afficheList(L);
si(rechercher(L,6)=1) alors
afficher("l'elem 6 existe") ;
sinon
afficher("l'elem 6 n'existe pas") ;
FinSi ;
suppDeb(L);
afficheList(L);
suppFin(L);
afficheList(L);
suppList(L);
afficheList(L);
Fin
	afficheList(L);
	rechercher(L,6)? printf("\nl'elem 6
	existe\n"):printf("l'elem 6 n'existe pas\n") ;
	suppDeb(&L);
	afficheList(L);
	suppFin(&L);
	afficheList(L);
	suppList(&L);
	afficheList(L);
	}

1) La procédure creeListe initialise une liste :

Remarque : Lorsque le pointeur L est NULL, cela signifie que la liste est vide.

2) ajoute un élément (de type entier) en début de chaîne

une procédure qui ajoute une valeur 'v' dans une liste L

3) ajoute un élément en fin de chaîne

1) La procédure creeListe initialise une liste :

```
Procédure creerListe( r L : pointeur vers cellule)
Début
L←NULL ;
Fin ;
```

```
void creerListe(cellule** L){
        *L=NULL;
    }
```

Remarque : Lorsque le pointeur L est NULL, cela signifie que la liste est vide.