CH2: Liste Dynamique Simplement Chainée

Université Sétif 1

Faculté des sciences

Département d'informatique

Algorithmique et Structures de Données

2017-2018 / Dr. L.Douidi

Liste chainée

- Une liste chaînée est une suite d'objet de même type accessible un à un du premier au dernier élément.
- La liste chaînée est une structure de données dynamiques, c'est-à-dire qu'elle permet de faire des allocations de mémoire selon la demande. La taille des données ou de la liste n'est pas fixée à l'avance, c'est le cas avec les tableaux (qui sont les structures de données statiques).

Liste chainée & Tableaux

On utilise une liste chaînée plutôt qu'un tableau lorsque l'on doit traiter des objets représentés par des suites sur lesquelles on doit effectuer de nombreuses suppressions et de nombreux ajouts. Les manipulations sont alors plus rapides qu'avec des tableaux.

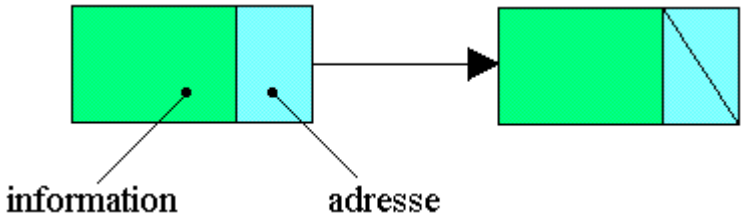
Résumé

Structure	Dimension	Position d'une information	Accès à une information
Tableau	Fixe (statique)	Par son indice	Directement par l'indice
Liste chainée	Evolue selon les actions (Dynamique)	Par son adresse	Séquentiellement par le pointeur de chaque élément

Les listes chaînées simples

C'est une structure de données telle que chaque élément forme un couple constitué:

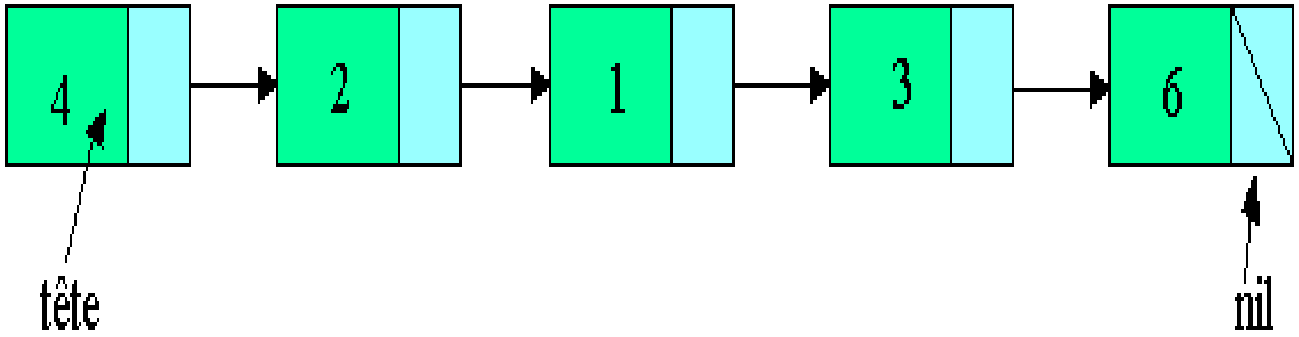
- Des informations caractéristiques de l'application, exemple: les caractères d'une personne.
- Une adresse ou pointeur vers un autre élément (le suivant) ou une marque de fin s'il y'a pas d'élément successeur.



Remarque 1:

- L'information peut être simple ou structurée (composée).
- Le couple information/adresse est appelé cellule.

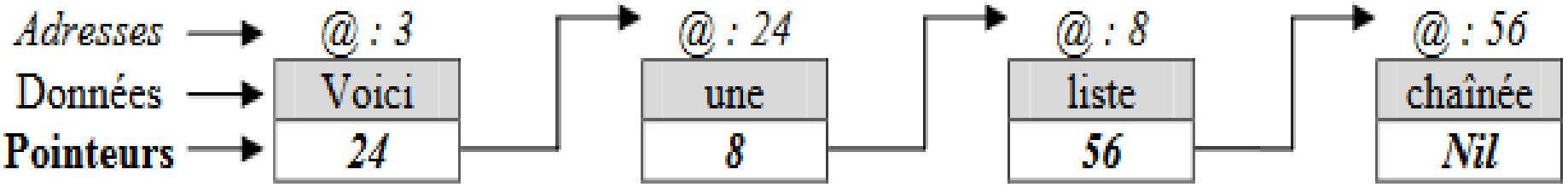
Exemple



Remarque 2:

- C'est l'adresse de la première cellule qui détermine la liste.
- Si le premier élément de la liste (tête) égale à nil (tête=nil), la liste est vide.
- Il y'a autant de cellule que d'élément.
- Le champ suivant d'une cellule contient l'adresse de la cellule suivante.
- Le champ suivant de la dernière cellule contient nil.

Soit la liste chaînée suivante (@ indique que le nombre qui le suit représente une adresse) :



Déclaration d'une liste chaînée

Type pointeur = liste chaînée : pointeur

listechainee = Structure
    val: type
    suivant: pointeur
fin structure
var P: pointeur

Exemple:

Déclaration des patients chez un médecin
    type Ptr = liste patient : pointeur

liste patient = Structure
    nom: chaîne
    numéro: entier
    suivant: Ptr
fin Structure
var P: Ptr

Exemple

- Voici la déclaration d'une liste simplement chaînée d'entiers en C:

Code : C

```
#include <stdlib.h>  // inclure stdlib.h afin de pouvoir utiliser la macro NULL
typedef struct element element;
struct element {
        int val;
        struct element *nxt;};
typedef element* llist;
```

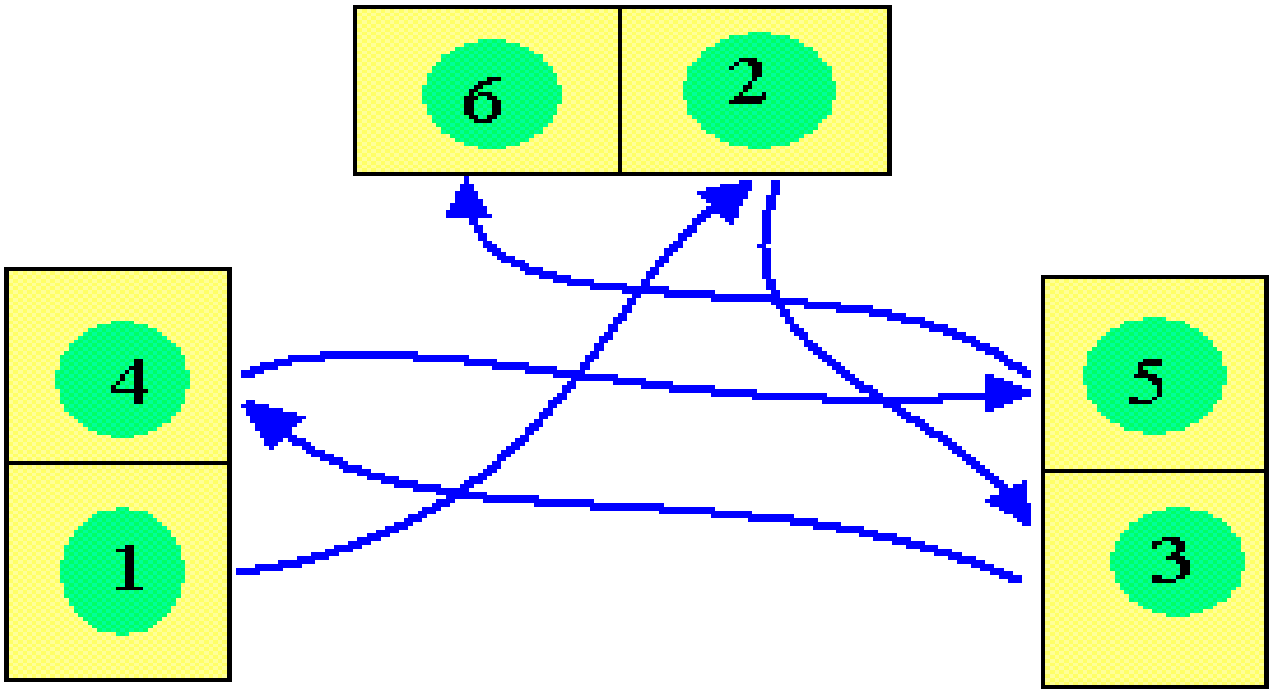
On crée le type element qui est une structure contenant un entier (val) et un pointeur sur élément (nxt), qui contiendra l'adresse de l'élément suivant.
Ensuite, il nous faut créer le type llist (pour *linked list* = liste chaînée) qui est en fait un pointeur sur le type element.
Lorsque nous allons déclarer la liste chaînée, nous devrons déclarer un pointeur sur element, l'initialiser à NULL, pour pouvoir ensuite allouer le premier élément.
nous avons juste crée le type llist afin de simplifier la déclaration.

comment déclarer une liste chaînée (vide pour l'instant)

```
#include <stdlib.h>
typedef  struct element  element;
struct element
{
   int val;
   struct element *nxt;
};
typedef  element* llist;

int main(int argc, char **argv)
{
   /* Déclarons de 3 listes chaînées de façons différentes mais équivalentes */
   llist  ma_liste1  =  NULL;        // toujours initialiser la liste chaînée à NULL
   element  *ma_liste2  =  NULL;
   struct element  *ma_liste3  =  NULL;

   return 0;
}
```

Liste dans la mémoire centrale

Présentation dans la mémoire centrale



Affectation

P ← tete

P1 ← P

Val(p) ← 5

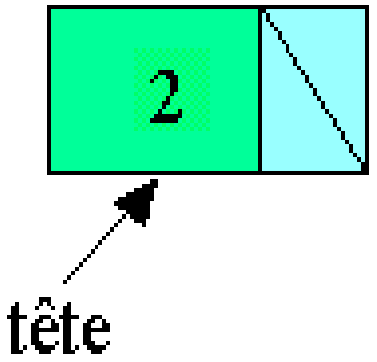
Lire(val(p))

Suivant(p) ← p1

P1 ← suivant(P)

Création d'une liste chaînée

Nous utilisons la fonction **allouer** pour l'allocation d'une nouvelle cellule.



procédure creer_liste(tete: Ptr)
var v: entier
début
Allouer(tete)
lire(v)
val(tete)← v
suivant(tete) ← nil
fin

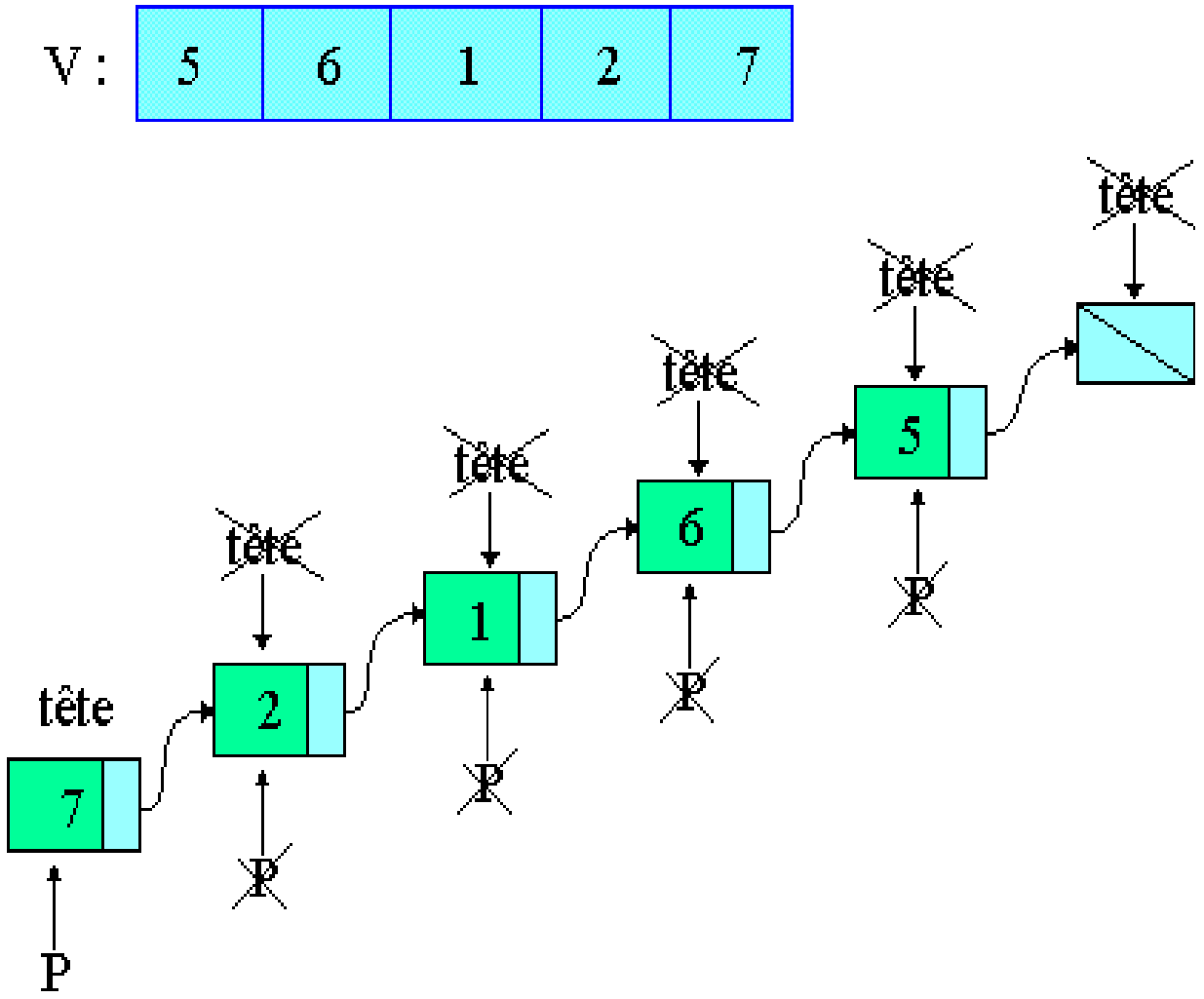
Activité:

Créer une liste chaînée à partir des éléments d'un vecteur.

procédure transfert_T_L(P: Ptr, V: vecteur, n: entier)

```
var i: entier
Pt: Ptr
début
P ← nil
pour i=1 à n faire
allouer(Pt)
val(Pt)← V[i]
suivant(Pt) ← P
P ← Pt
fin pour
fin
```

Créer une liste chaînée à partir des éléments d'un vecteur.



Créer une liste chaînée à partir des éléments d'un vecteur.

```
procédure creation(V: vecteur, tete: Ptr, N: entier)
    va i: entier
    P: Ptr
    début
    tete ← nil
    pour i=1 à N faire
        allouer(P)
        val(P) ← V[i]
        suivant(P) ← tete
        tete ← P
    fin pour
    fin
```

Création dans l'ordre normal

```
procédure creation_dans_l'ordre_normal(V: vecteur, tete: Ptr,
    n:entier)
    var i: entier
    P1, P: Ptr
    début
    allouer(tete)
    val(tete) ← V[1]  // si n>1
    P ← tete
    pour i=2 à n faire
          allouer(P1)
          val(P1)←V[i]
          suivant(P) ← P1
          P ← P1
    fin pour
    suivant(P1) ← nil
    fin
```

Activité

- Ecrire une procédure qui affiche les éléments d'une liste chaînée.
- Ecrire une fonction qui prend en paramètre une liste chaînée et renvoie sa taille

procédure qui affiche les éléments d'une liste chaînée

```
procédure affiche(tete: Ptr)
    var P1: Ptr
    début
    P1 ← tete
    tant que P1 ≠ nil faire
            écrire(val(P1))
            P1 ← suivant(P1)
    fin tant que
    fin

    procédure affiche(P: Ptr)
    var Pt: Ptr
    début
    Pt ← P
    tant que Pt ≠ nil faire
            écrire(val(Pt))
            Pt ← suivant(Pt)
    fin tant que
    fin
```

fonction qui prend en paramètre une liste chaînée et renvoie sa taille

```
fonction taille(L: Ptr):entier
    var S: entier
    P: Ptr
    début
    P ← L
    S ← 0
    tant que P ≠ nil faire
            S ← S+1
            P ← suivant(P)
    fin tant que
    taille ← S
    fin
```

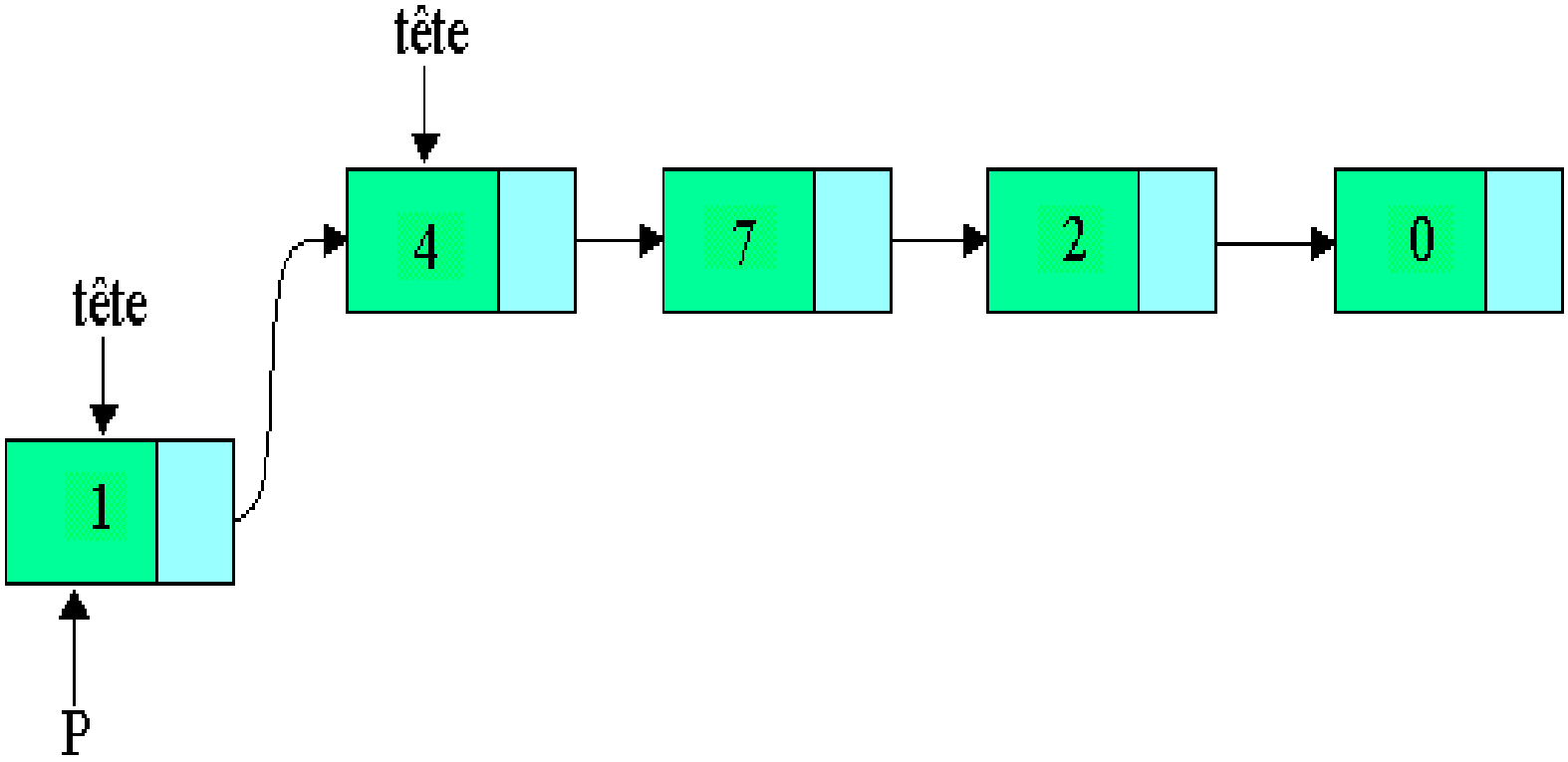
Opérations sur les Listes Simplement chainées

- Ajout d'un élément dans une liste chaînée
  - **Ajout en tête**

Pour ajouter un élément au début de la liste on a les étapes suivantes:

- Créer une nouvelle cellule.
- Remplir la nouvelle valeur dans la partie information.
- Effectuer le chaînage.

Ajout en tête de liste



Ajout en tête de liste

Algorithme 1 :

```
procédure inserer_au_debut( tete: Ptr)
    var P: Ptr;
    v: entier   // peut être éliminée
début
    allouer(P)

    lire(val(P))  // ou lire(v) ; val(P) ← v
    suivant(P) ← Nil

    suivant(P) ← tete
    tete ← P
fin
```

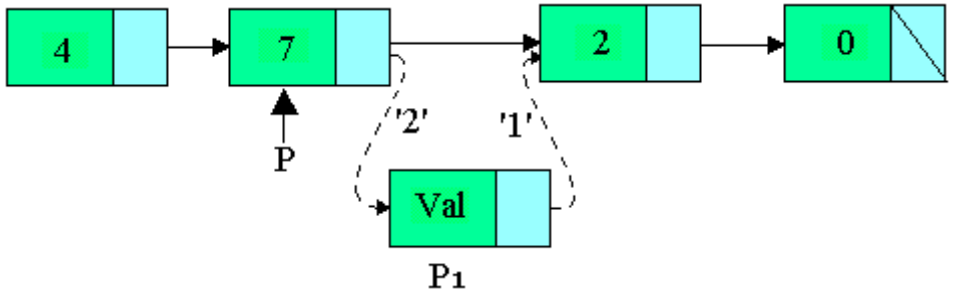
Ajout en tête de liste

Algorithme 2 :

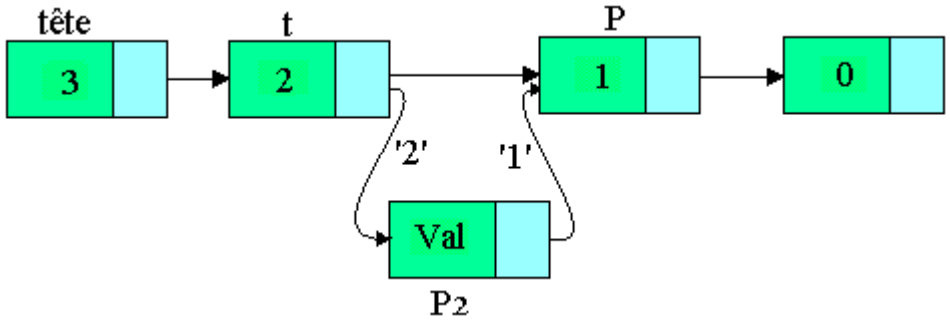
```
procédure inserer_au_debut(Pt: Ptr, v: entier)
var P: Ptr
début
    allouer(P)
    val(P) ← v   // v : valeur lue au prgme appelant
    suivant(P) ← Pt

    Pt ← P
fin
```

Activité

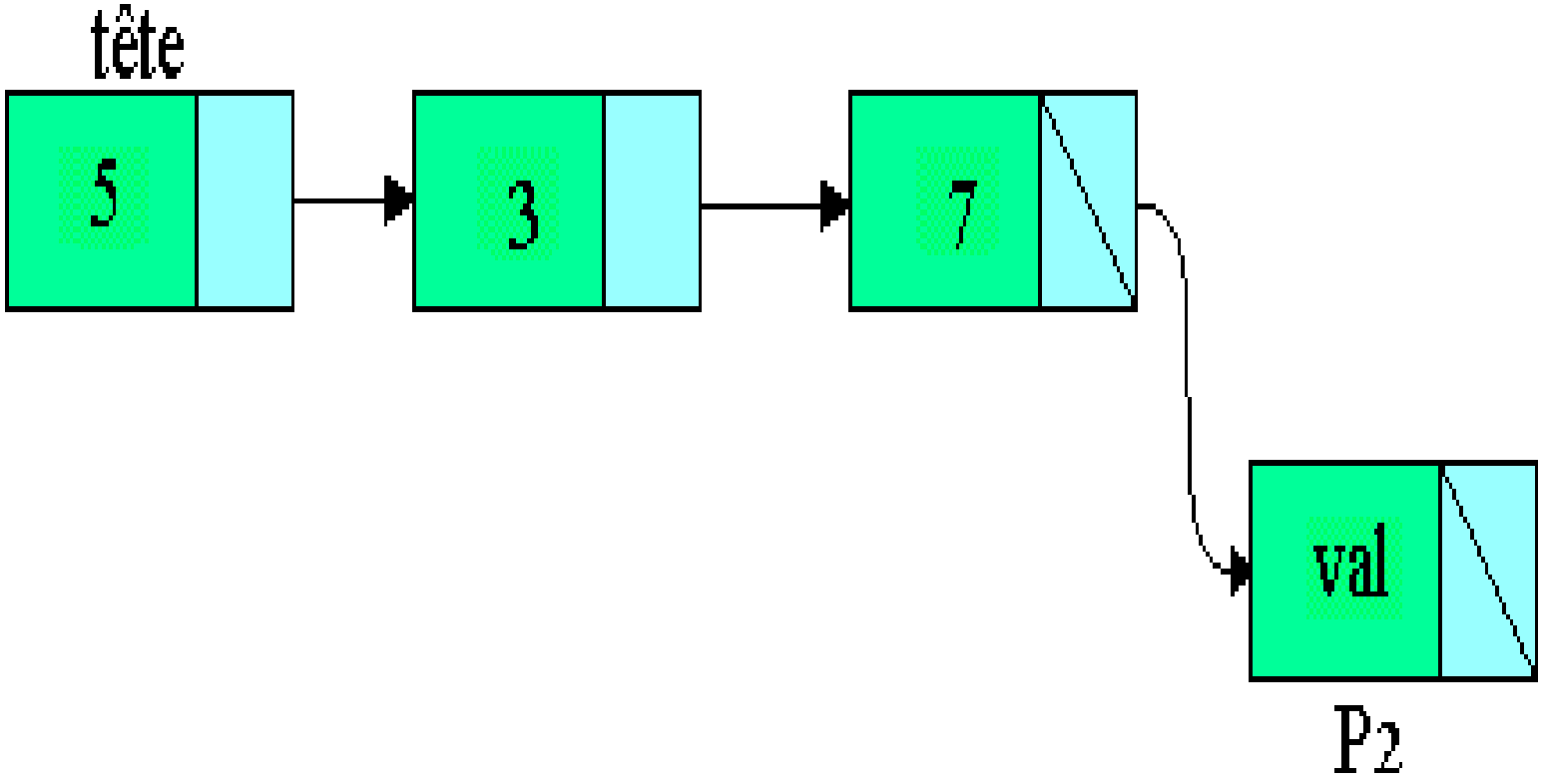
- Ajouter un nouvel élément **<u>après</u>** une cellule pontée par P



- Ajouter un nouvel élément **<u>avant</u>** une cellule pontée par P



Ajout en fin de liste



Ajout en fin de liste

procédure ajout_fin(tete: Ptr)
var P_1, P_2: Ptr;
v: type_v
début
P_1 ← tete
tant que suivant (P_1) ≠ nil faire
    P_1←suivant(P_1)
fin tant que

lire(v)
allouer(P_2)
val(P_2) ← v
suivant(P_2) ← nil
Suivant(P_1) ← P_2
fin

Activité

- Insérer une cellule **après** une cellule de **valeur V**
- Insérer une cellule **avant** une cellule de **valeur V**
- Compter le nombre de cellules

Compter le nombre d'occurrence d'une cellules

- *fonction compte(tête: Ptr, v: entier): entier*
- *var P: Ptr*
- *i: entier*
- *début*
- *P ← tete*
- *t ← 0*
- *tant que P ≠ nil faire*
- *si val(P)=v alors*
- *t ← t+1*
- *fin si*
- *P ← suivant(P)*
- *fin tant*
- *compte ← t*
- *fin*

Suppression d'un élément dans une liste chaînée

Utilisons la fonction libérer pour **libérer** un emplacement occupé en mémoire.

Remarque:

- La fonction "libérer" ne libère pas un emplacement nil et il libère une seule cellule à la fois (pas la liste toute entière).

Suppression en tête de liste

- Il s'agit de retirer l'élément en tête de la liste si la liste est vide, on ne retire aucun élément et on renvoi un message correspondant

```
procédure supp_tete(tete: Ptr)
var P: Ptr
P ← tete
si tete=nil alors
    écrire('pas de suppression')
sinon
    tete ← suivant(tete)
    liberer(P)
fin si
fin
```

Activité

- Ecrire une fonction/procédure qui permet la suppression d'un élément P de la liste.
- Ecrire une fonction/procédure qui permet la concaténation de deux listes L1 et L2.
- Ecrire une fonction qui renvoie le nombre d’éléments d’une liste chaînée ayant une valeur donnée (champ valeur/Info).

Suppression en fin de liste

- Pour supprimer un élément en fin de liste, il faut maintenir un pointeur à l'avant dernière cellule, ensuite libérer la dernière cellule et enfin mettre le suivant de l'avant dernière cellule à nil. La procédure correspondante est la suivante.

Procédure Supp_fin(tete: Ptr)

var P, P_1: Ptr

début
Si tete ≠ Nil alors
P_1 ← tete
P ← suivant(P_1)
tant que suivant(P) ≠ nil faire
P_1 ← suvant(P_1)
P ← suivant(P)
fin tant que
suivant(P1) ← nil
liberer(P)
fin

Autres activitées

Ecrire les fonctions qui permettent:

Fonction de comptage d'occurrences dans une liste chaînée

Fonction de vérification d'une liste chaînée triée

Procédure d'insertion à une position donnée

Procédure de suppression d'un élément d'une liste chaînée à une position donnée

....

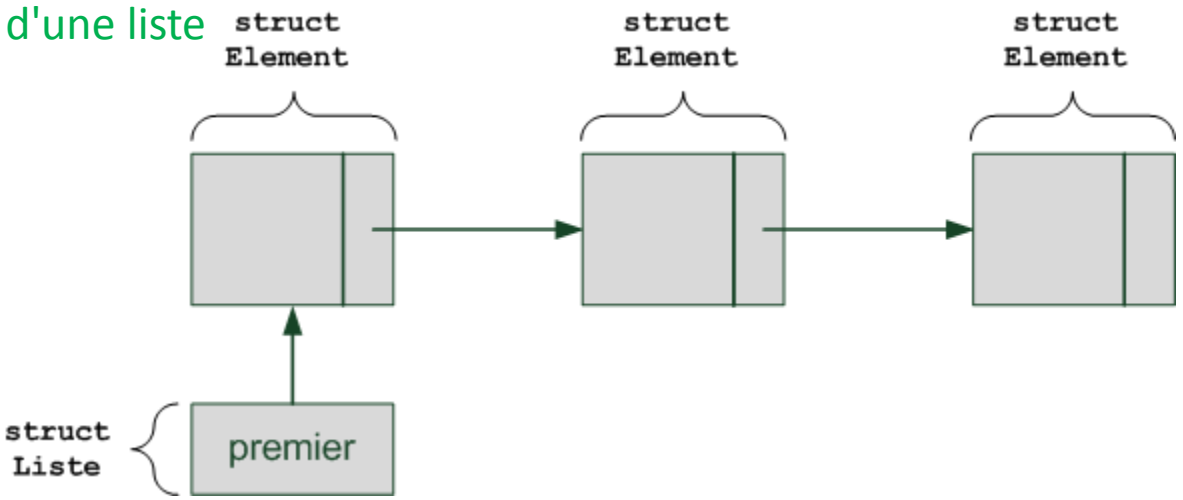
Listes en C & C++

```
//Déclaration de la structure:
typedef struct slist
{
    int valeur;
    struct slist *suiv;
} slist ;
// le point d'entrée sera un pointeur sur l'élément
   de début de liste impérativement initialisé à
   NULL :
slist *Mysl = NULL;
```

Liste chainées en c/c++

//Nous avons créé ici un élément d'une liste chaînée, correspondant à la fig.

```
typedef struct Element Element;
struct Element
{
int valeur;
Element *suivant;
};
```



//Cette structure Liste contient un pointeur vers le premier élément de la liste. En effet, il faut conserver l'adresse du premier élément pour savoir où commence la liste. Si on connaît le premier élément, on peut retrouver tous les autres en « sautant » d'élément en élément à l'aide des pointeurs suivant

```
typedef struct Liste Liste;
struct Liste
{
Element *premier;
};
```

Liste chainées en c/c++ suite

- Une donnée, ici un valeur de type int: on pourrait remplacer cela par n'importe quelle autre donnée (un double, un tableau…). Cela correspond à ce que vous voulez stocker, c'est à vous de l'adapter en fonction des besoins de votre programme.
- Si on veut travailler de manière générique, l'idéal est de faire un pointeur sur void :

  void*. Cela permet de faire pointer vers n'importe quel type de données.

Exemple corrigé

- Voici la déclaration d'une liste simplement chaînée d'entiers en C:

Code : C

```
#include <stdlib.h>  // inclure stdlib.h afin de pouvoir utiliser la macro NULL
typedef struct element element;
struct element {
        int val;
        struct element *nxt;};
typedef element* llist;
```

On crée le type element qui est une structure contenant un entier (val) et un pointeur sur élément (nxt), qui contiendra l'adresse de l'élément suivant.
Ensuite, il nous faut créer le type llist (pour *linked list* = liste chaînée) qui est en fait un pointeur sur le type element.
Lorsque nous allons déclarer la liste chaînée, nous devrons déclarer un pointeur sur element, l'initialiser à NULL, pour pouvoir ensuite allouer le premier élément.
nous avons juste crée le type llist afin de simplifier la déclaration.

comment déclarer une liste chaînée (vide pour l'instant)

```
#include <stdlib.h>
typedef  struct element  element;
struct element
{
   int val;
   struct element *nxt;
};
typedef  element* llist;

int main(int argc, char **argv)
{
   /* Déclarons de 3 listes chaînées de façons différentes mais équivalentes */
   llist  ma_liste1  =  NULL;        // toujours initialiser la liste chaînée à NULL
   element  *ma_liste2  =  NULL;
   struct element  *ma_liste3  =  NULL;

   return 0;
}
```

Ajouter en tête

```
llist ajouterEnTete(llist liste, int valeur)
{
   /* On crée un nouvel élément */
   element* nouvelElement = malloc(sizeof(element));

   /* On assigne la valeur au nouvel élément */
   nouvelElement->val = valeur;

   /* On assigne l'adresse de l'élément suivant au nouvel élément */
   nouvelElement->nxt = liste;

   /* On retourne la nouvelle liste, i.e. le pointeur sur le premier
    élément*/
   return nouvelElement;
}
```

Ajouter en fin de liste

```
llist ajouterEnFin(llist liste, int valeur)
{
   /* On crée un nouvel élément */
      element* nouvelElement = malloc(sizeof(element));
   /* On assigne la valeur au nouvel élément */
      nouvelElement->val = valeur;
   /* On ajoute en fin, donc aucun élément ne va suivre */
      nouvelElement->nxt = NULL;
   if(liste == NULL)
   {        /* Si la liste est vide il suffit de renvoyer l'élément créé */
      return nouvelElement;
   }
   else   {       /* Sinon, on parcourt la liste à l'aide d'un pointeur temporaire et on indique que le dernier
                  élément de la liste est relié au nouvel élément */
                  element* temp=liste;
                  while(temp->nxt != NULL)
                  {
                              temp = temp->nxt;
                  }
                  temp->nxt = nouvelElement;
                  return liste;
          }
}
```

Exercice

En utilisant les trois fonctions que nous avons vues jusqu'à présent :

- ajouterEnTete
- ajouterEnFin
- afficherListe

Vous devez écrire le main permettant de remplir et afficher la liste chaînée ci-dessous.

NB:Vous ne devrez utiliser qu'une seule boucle for.

10 9 8 7 6 5 4 3 2 1 1 2 3 4 5 6 7 8 9 10

Correction exercice

```
int main(int argc, char **argv)
{
   llist ma_liste = NULL;
   int i;

   for(i=1;i<=10;i++)
   {
      ma_liste = ajouterEnTete(ma_liste, i);
      ma_liste = ajouterEnFin(ma_liste, i);
   }
   afficherListe(ma_liste);

   return 0;
}
```

Supprimer un élément en tête

```
llist supprimerElementEnTete(llist liste)
{
   if(liste != NULL)
   {
      /* Si la liste est non vide, on se prépare à renvoyer l'adresse de
      l'élément en 2ème position */
      element* aRenvoyer = liste->nxt;
      /* On libère le premier élément */
      free(liste);
      /* On retourne le nouveau début de la liste */
      return aRenvoyer;
   }
   else
   {
      return NULL;
   }
}
```

Supprimer un élément en fin de liste

```
llist supprimerElementEnFin(llist liste)
{       /* Si la liste est vide, on retourne NULL */
        if(liste == NULL)
                return NULL;
        /* Si la liste contient un seul élément */
        if(liste->nxt == NULL)
        {
                /* On le libère et on retourne NULL (la liste est maintenant vide) */
                free(liste);
                return NULL;
        }
   /* Si la liste contient au moins deux éléments */
   element* tmp = liste;
   element* ptmp = liste;
   /* Tant qu'on n'est pas au dernier élément */
   while(tmp->nxt != NULL)
   {    /* ptmp stock l'adresse de tmp */
      ptmp = tmp;
      /* On déplace tmp (mais ptmp garde l'ancienne valeur de tmp */
      tmp = tmp->nxt;
   }
   /* A la sortie de la boucle, tmp pointe sur le dernier élément, et ptmp sur l'avant-dernier. On indique que l'avant-
      dernier devient la fin de la liste et on supprime le dernier élément */
   ptmp->nxt = NULL;
   free(tmp);
   return liste;
}
```

Supprimer d'une liste chaînée un élément portant une valeur donnée

Il faut:

- traiter à part la suppression du premier élément car il faut modifier le pointeur de tête,

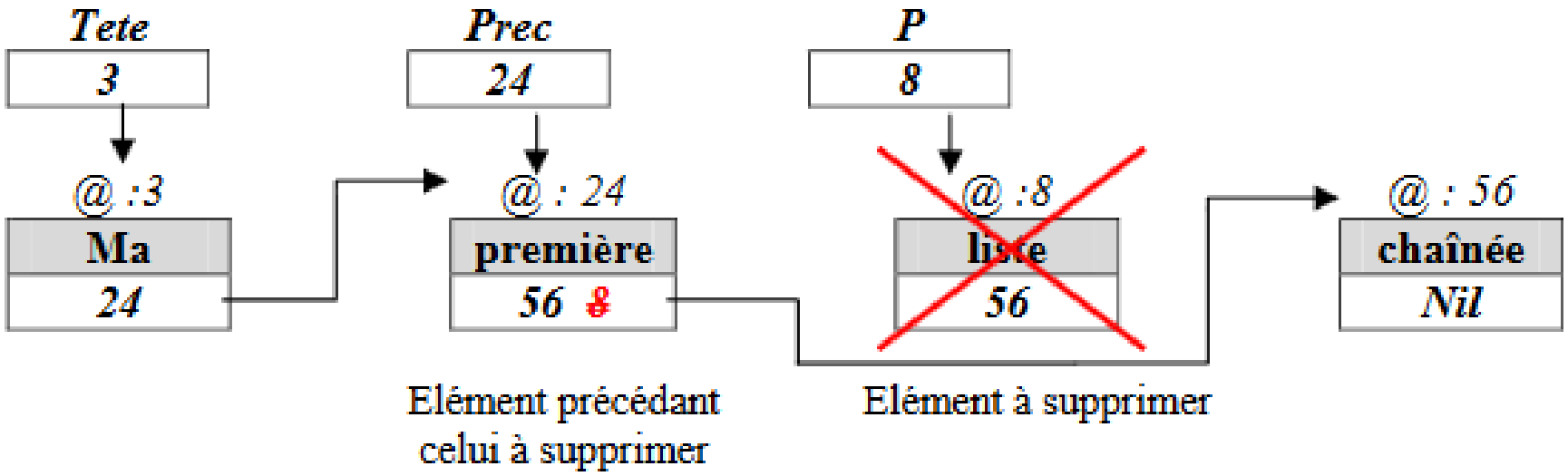
- trouver l'adresse P de l'élément à supprimer,

- sauvegarder l'adresse Prec de l'élément précédant l'élément pointé par P pour connaître l'adresse de l'élément précédant l'élément à supprimer, puis faire pointer l'élément précédent sur l'élément suivant l'élément à supprimer,

- Libérer l'espace mémoire occupé par l'élément supprimé.

Supprimer d'une liste chaînée un élément portant une valeur donnée

L'exemple considère que l'on souhaite supprimer l'élément contenant la valeur "liste" de la liste ci-dessus.



```
Procedure SupprimerElement (Tete : Ptr, V : variant)   /* Supprime l'élément dont la valeur est passée en paramètre */
Variables locales
P : Ptr   /* pointeur sur l'élément à supprimer */
Prec : Ptr   /* pointeur sur l'élément précédant l'élément à supprimer */
Trouvé : Booleen    /* indique si l'élément à supprimer a été trouvé */
DEBUT
SI Tete <> Nil ALORS   /* la liste n'est pas vide on peut donc y chercher une valeur à supprimer */
      SI Val(Tete) = V ALORS /* l'élément à supprimer est le premier */
                  P ← Tete
                  Tete  ←  Suivant (Tete)
                  Liberer(P)
      SINON    /* l'élément à supprimer n'est pas le premier*/
                  Trouve ←  Faux
                  Prec ← Tete     /* pointeur précédent */
                  P  ← Suivant(Tete)    /* pointeur courant */
                  TANTQUE P <> Nil ET Non Trouve FAIRE
                              SI Val(P) = V ALORS    /* L'élément recherché est l'élément courant */
                                          Trouve ← Vrai
                              SINON    /* L'élément courant n'est pas l'élément cherché */
                                          Prec ← P    /* on garde la position du précédent */
                                          P ← Suivant(P)     /* on passe à l'élément suivant dans la liste */
                              FINSI
                  FIN TANT QUE
                  SI Trouve ALORS
                              Suivant(Prec) ← Suivant(P)     /* on "saute" l'élément à supprimer "/
                              Liberer(P)
                  SINON
                              Ecrire ("La valeur ", V, " n'est pas dans la liste")
                  FINSI
      FINSI
SINON
      Ecrire("La liste est vide")
FINSI
FIN
```