10 - Implantation de listes en mode chaîné

© J. Morinet Lambert, M. Cadot -UHP

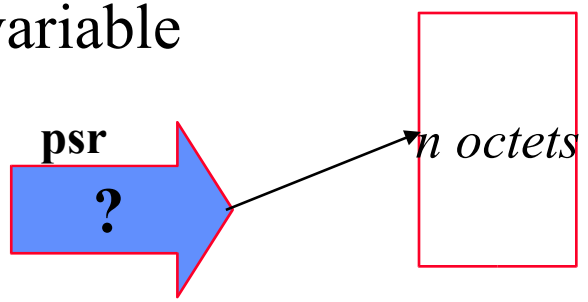
L. Pierron

Mars 2001- Modifié le 17/11/04

Rappels C : pointeurs et variable dynamique

1) Déclaration d'un pointeur sur le type de variable

```
float  *psr;
```



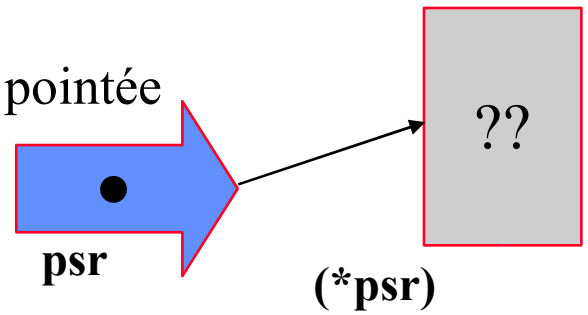
2) Création de la variable pointée

```
psr= (*float) malloc(sizeof (float));
```

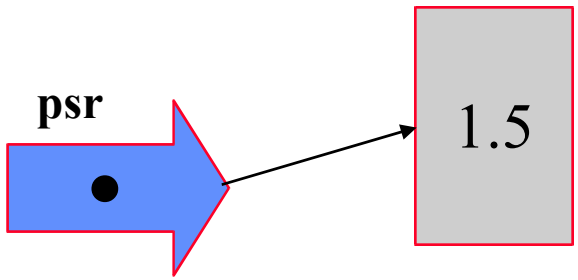
le pointeur a pour valeur l'adresse de la variable pointée

3) Valorisation de la variable pointée

```
*psr = 1.5;
```

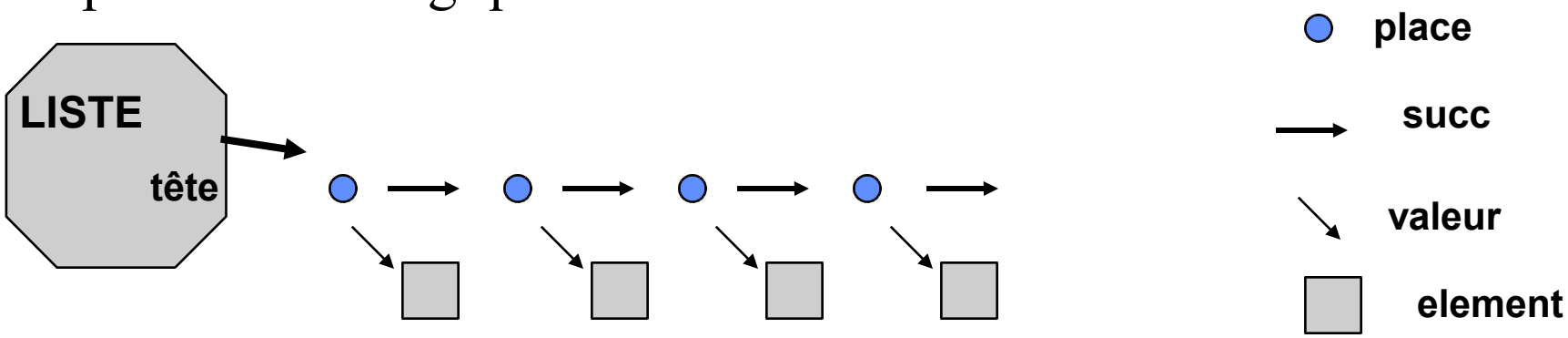


Valeur particulière de pointeur : NULL



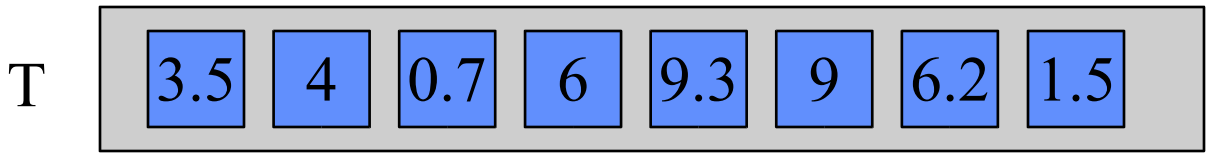
Notion de liste

- Définition itérative
  - ensemble fini et ordonné de places (ordre = rang) = collection d'objets
  - une application assigne à chaque place, un élément
  - il existe une fonction qui permet de passer d'une place à la suivante : succ
  - à partir de la première place (tête), il est possible d'accéder aux suivantes en appliquant de manière répétée la fonction succ
- Représentation logique



Liste contigüe

- Utilise un tableau déclaré **float T [max];**
- accès
  - la place correspond au rang : i, index varie de 0 à max-1
  - successeur : c'est la fonction qui à i associe i+1
- chaque élément
  - possède une valeur T[i]
  - la capacité de passer au RANG suivant [i+1]
  - association d'informations de types différents
    - valeur de l'élément (ici de type réel)
    - rang de type entier (correspondant à indice +1)
    - place (de type adresse) : &T[i] (dans la mémoire)



Représentation de liste par variable dynamique

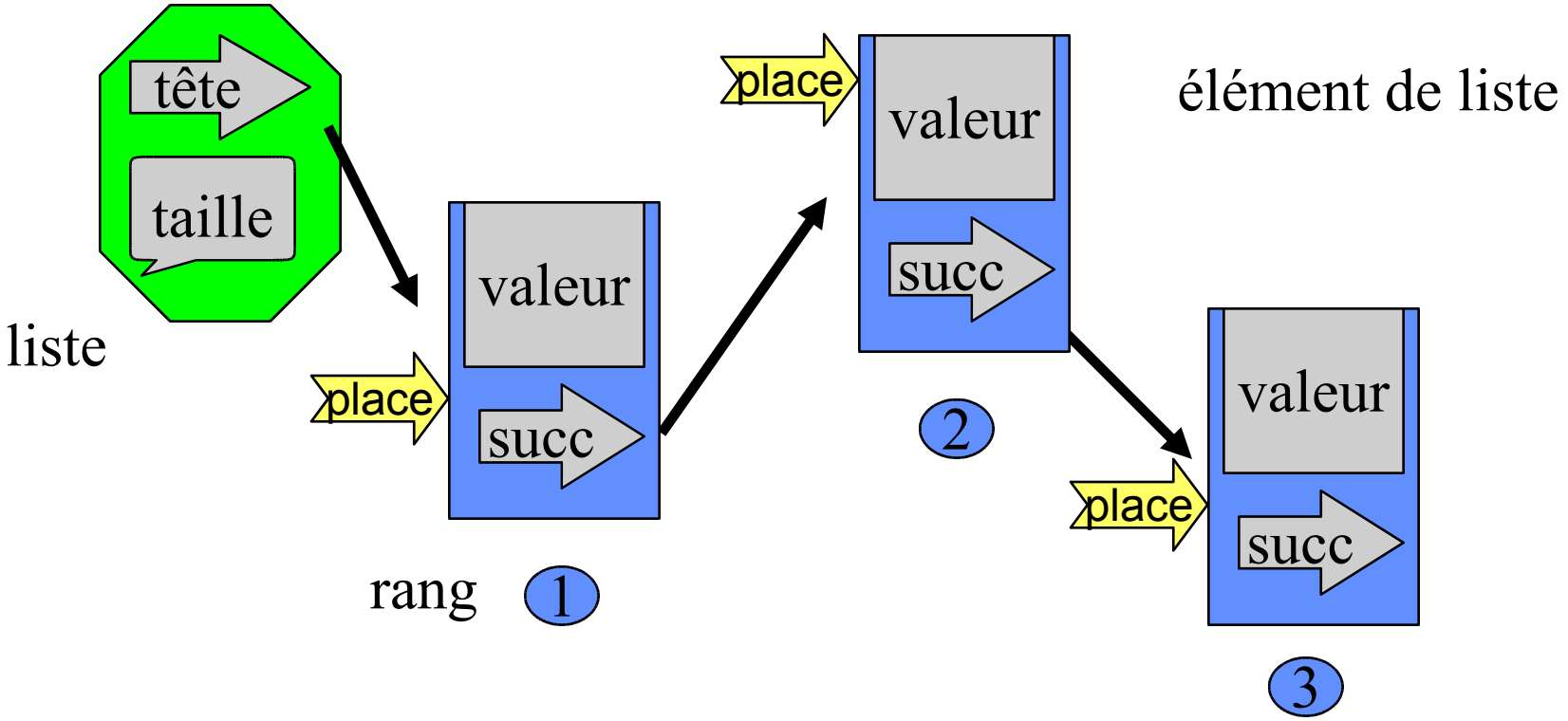
- ce sont les éléments de la liste qui sont implantés par des variables dynamiques qui sont créées lors de l'exécution du programme selon les besoins

- Chaque élément comporte
  - une valeur
  - et la capacité d'accéder à l'élément suivant

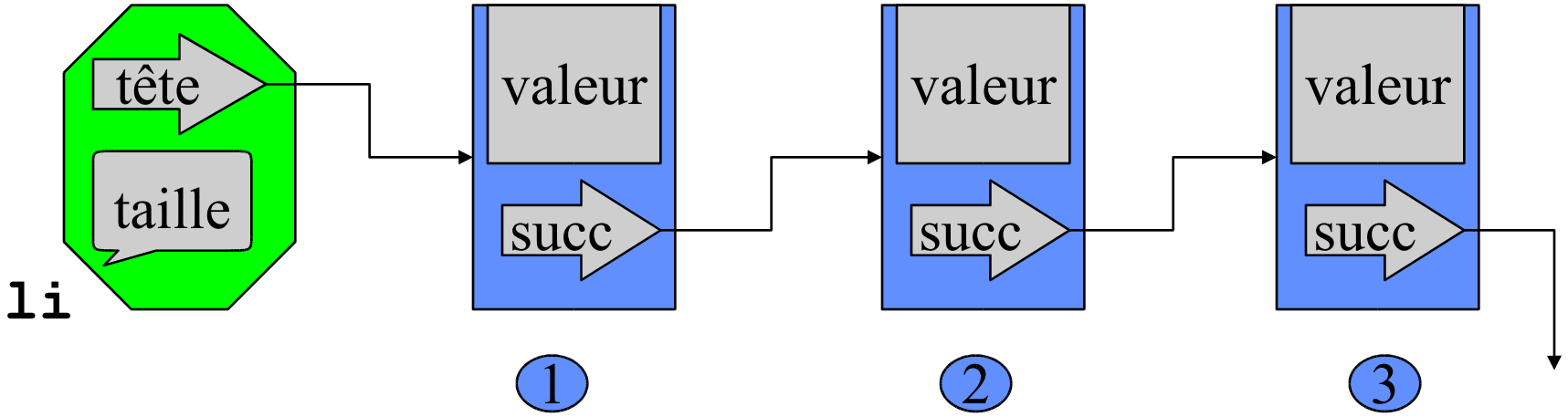
- une variable représente la liste : elle est déclarée et comporte des informations dont l'accès à l'élément de tête
- Cette variable n'est **pas** la liste

Implantation par pointeurs et variables dynamiques

- **`liste`** : permet l'accès à l'élément de tête de la liste (le premier)
- accès à tout autre élément de la liste (créé dynamiquement) par son adresse (**`succ`**)



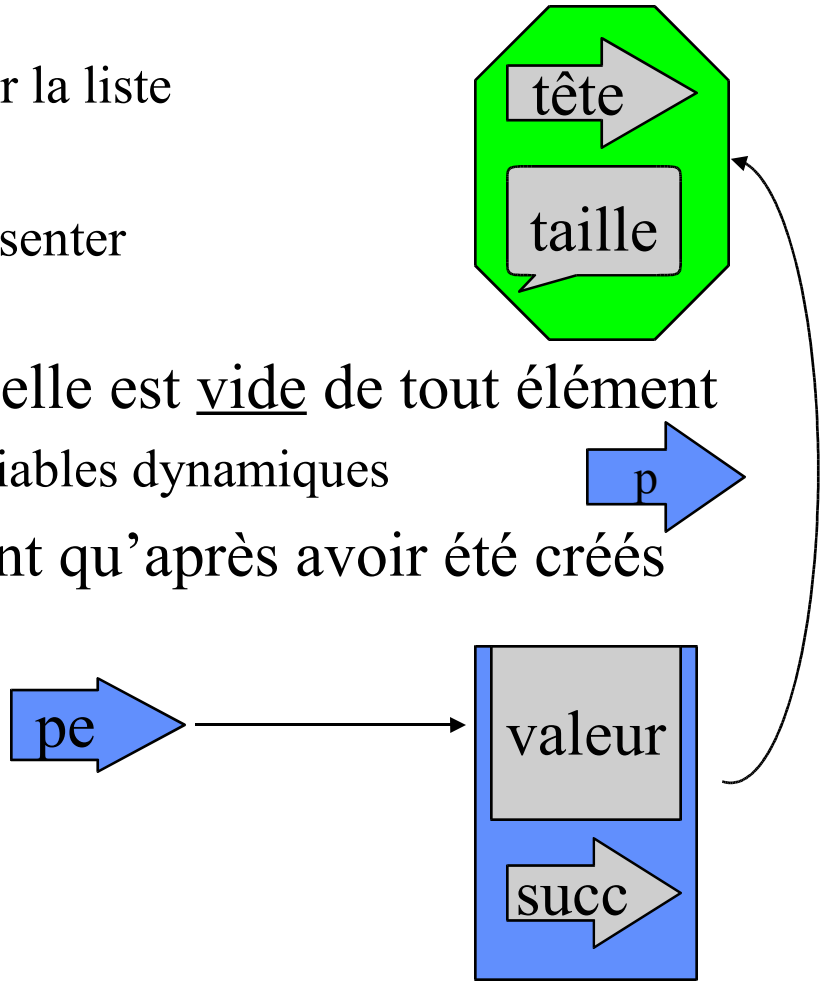
Exercice : comment accéder à...



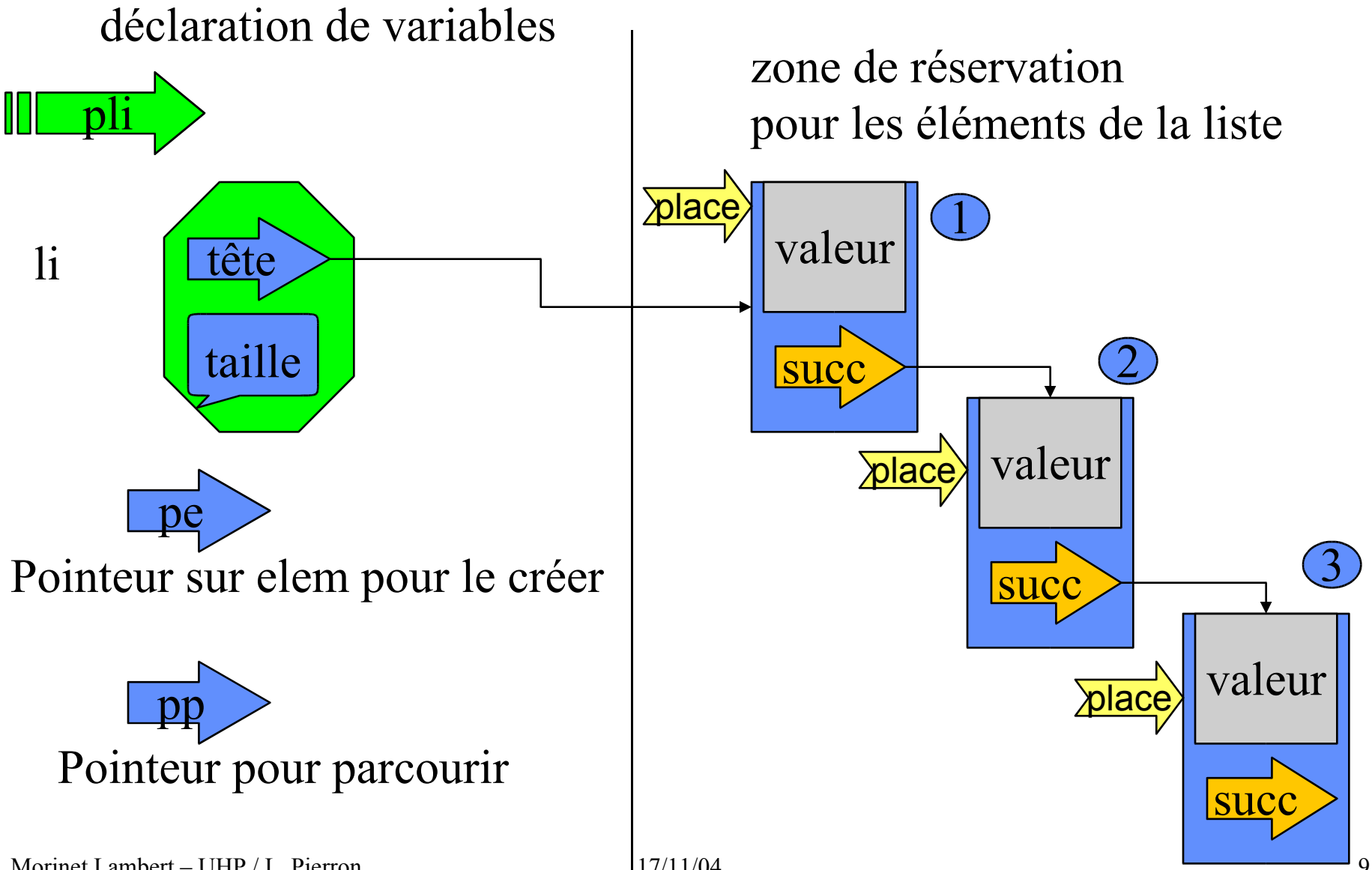
- Adresse du premier élément de la liste
- Valeur du premier élément de la liste
- Adresse du deuxième élément de la liste
- Valeur du deuxième élément de la liste
- Adresse du troisième élément de la liste
- Valeur du troisième élément de la liste
- Adresse du quatrième élément de la liste

Création de liste

- Il faut faire la différence entre
  - la variable qui permet de représenter la liste
  - et
  - les variables qui permettent de représenter les éléments de la liste
- La liste existe dès sa déclaration : elle est vide de tout élément
  - il faut un pointeur pour créer les variables dynamiques
- Les éléments de la liste n'existeront qu'après avoir été créés
  - par une instruction malloc
  - puis
  - insérés dans la liste : chaînage



Représentation en mémoire (1)



Déclarations

- Catégories de variables
  - liste : association de l'accès au premier élément et de la taille
  - élément : association de la valeur et du successeur
  - successeur : adresse qui permet d'accéder à l'élément suivant
- Déclarations C ***

```
typedef struct s_elem {float val;
                       struct s_elem *succ ; } elem;
typedef struct {elem *tete; int taille;} list;
```

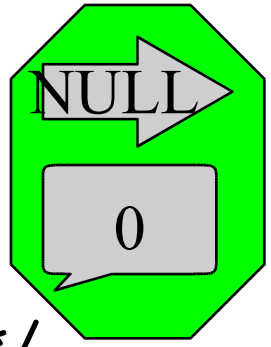
- Remarques sur la syntaxe C:
  - Nécessite un tag *s_elem* pour pouvoir définir avant de l'utiliser dans la structure,ensuite on utilise uniquement **elem**
  - importance de l'ordre de la déclaration : **elem** d'abord
  - `elem *pe; list li; /*déclarations de variables*/`

Initialisations de liste

- Initialisation de variable dès la déclaration

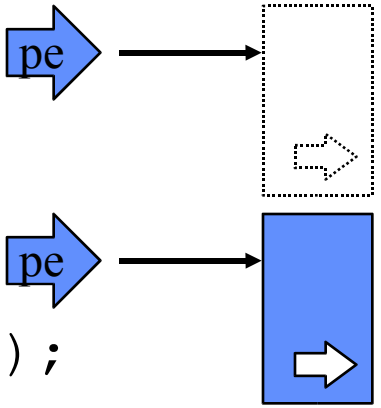
```
list li={ NULL, 0};
```

- La liste est créée (déclarée), il faut l'initialiser




```
void initliste(list *pli){
 pli->tete = NULL; pli->taille = 0;}
...
int main(void ){  /*programme principal */
  list li; /* déclaration la zone réservée contient
              n'importe quoi */
  initliste(&li); /*on range les bonnes valeurs
                    dans la zone réservée */
  /* à faire obligatoirement avant utilisation */
}
```

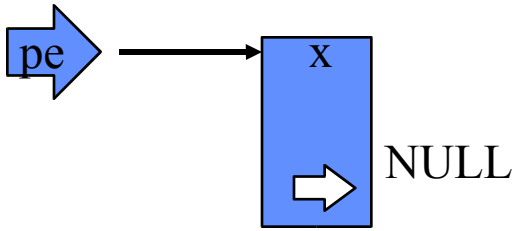
Insertion d'un nouvel élément



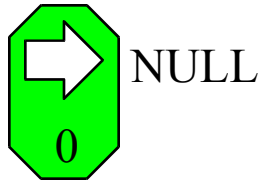
- Inclure `<stdlib.h>` pour utiliser `malloc` :

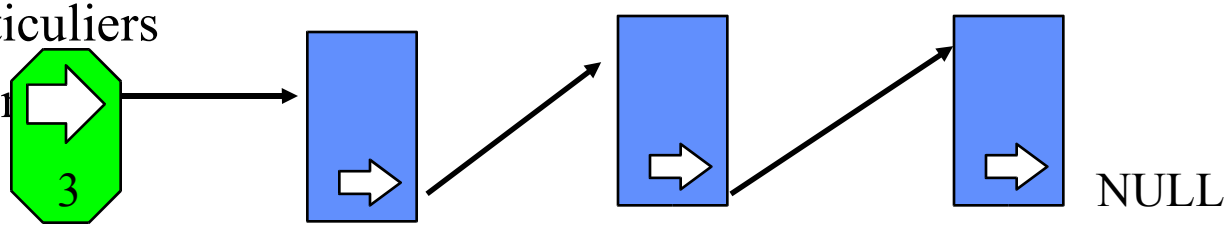
```
#include <stdlib.h>
```

- Il faut le créer : `pe = malloc(sizeof *pe);`
- Le valuer
  - les deux champs :
    - valeur : **`pe->val =`**...
    - succ : **`pe->succ =`**…



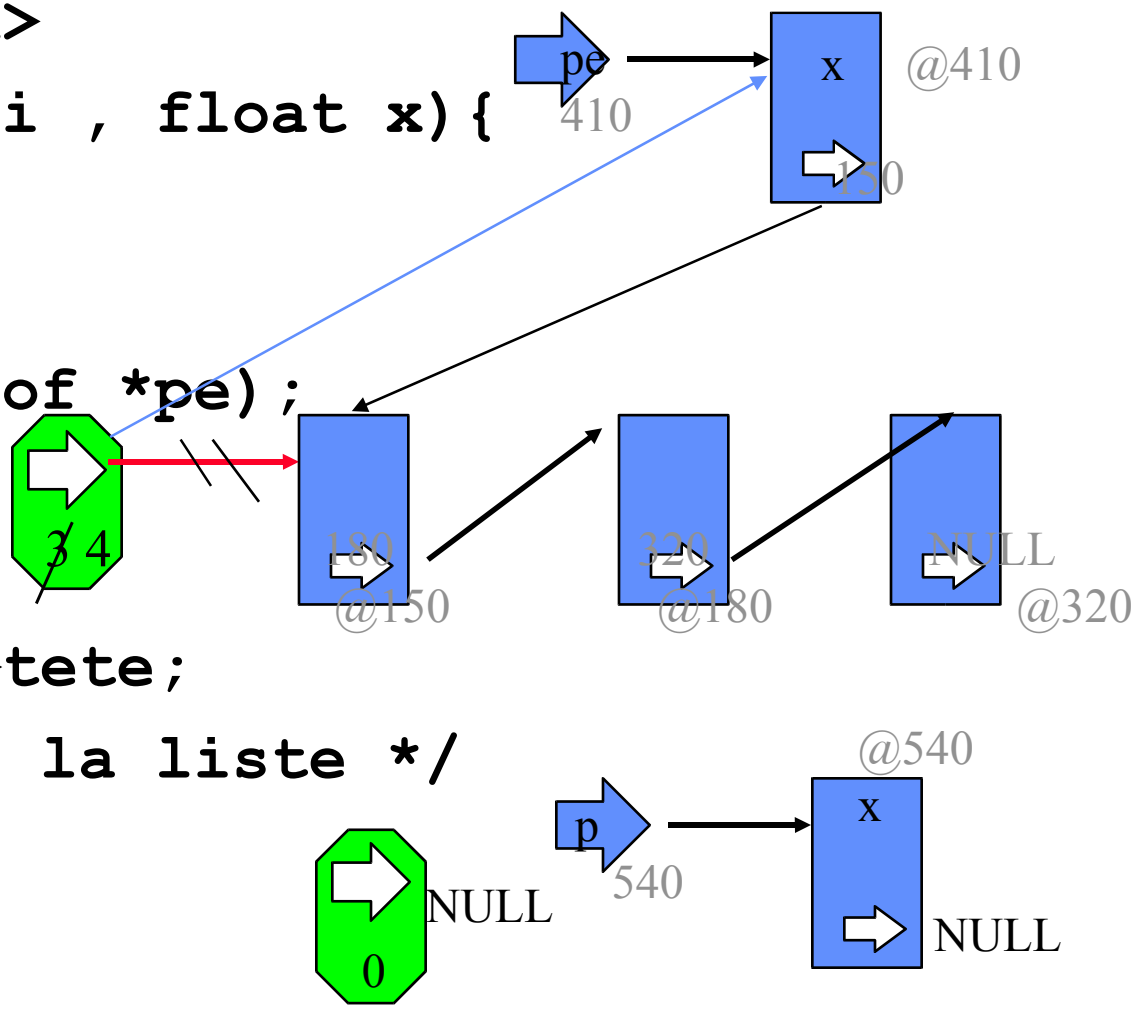
- L'insérer dans la liste
  - selon le rang qu'il doit occuper
    - premier, dernier, ailleurs
  - selon les cas particuliers
    - liste vide ou





Adjonction en tête dans une liste de réels

```
#include <stdlib.h>
void adjt(list *pli , float x){
   elem *pe ;
   /*le créer */
   pe = malloc(sizeof *pe);
   /*le valuer*/
   pe->val = x;
   pe->succ = pli->tete;
   /*l'insérer dans la liste */
   pli->tete = pe;
   pli->taille ++;
} //  que se passe-t-il si la liste était vide ?
```



Parcours de liste

- 1 Utilisation d'un index -> d'un pointeur de parcours

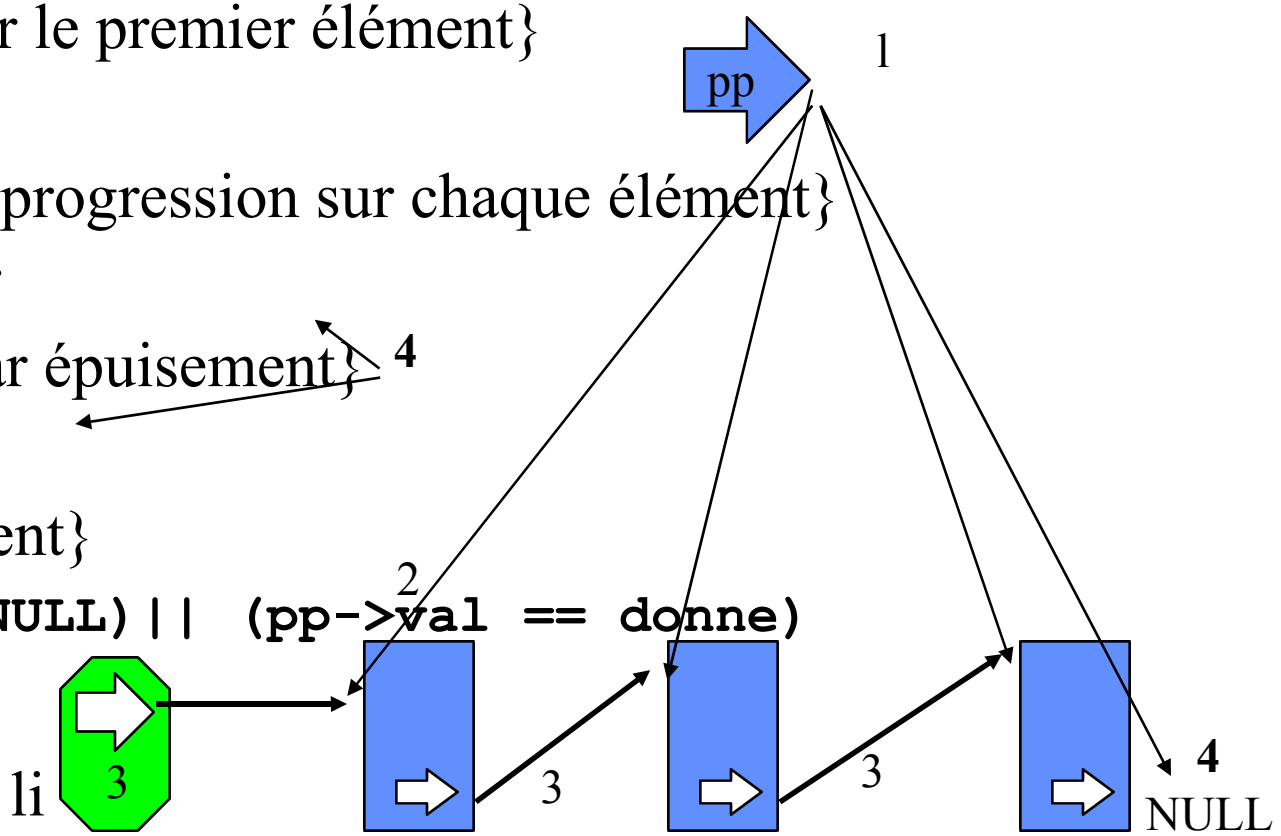
  `struct elem *pp ;` {**p**ointeur pour **p**arcours}
- 2 {initialisation sur le premier élément}

  `pp= li.tete;`
- 3 {traitement puis progression sur chaque élément}

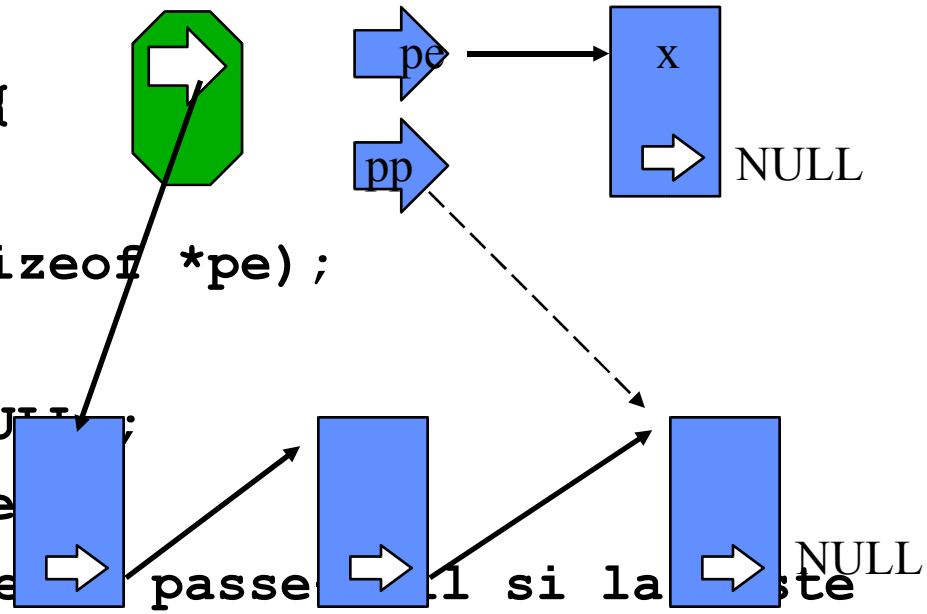
  `pp= pp->succ ;`
- {condition arrêt par épuisement}

  `(pp==NULL)`
- {arrêt sur un élément}

  `(pp->succ == NULL)|| (pp->val == donne)`



Adjonction en queue

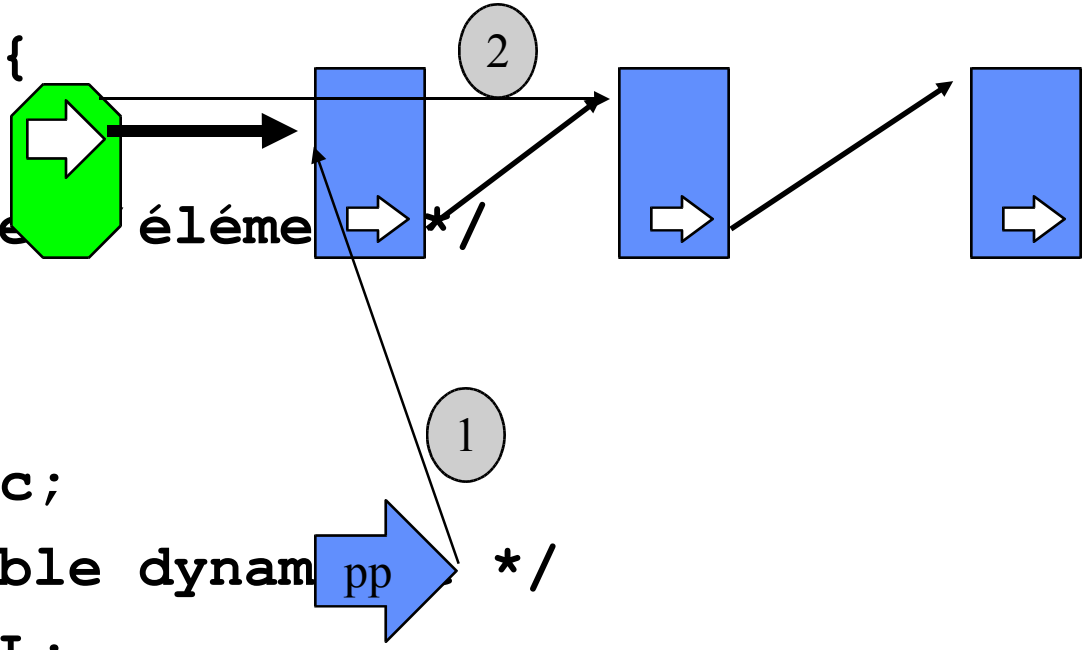



```
void adjq (list *l,float x){
  elem *pe, *pp;
  /*le créer */ pe=malloc(sizeof *pe);
  /* le valuer */
  pe->val = x; pe->succ = NULL;
  /* l'insérer dans la liste
  /*rechercher la queue, que passe-l si la ste
    était vide ?*/
  if (l->tete == NULL) l->tete = pe;
  else {
     pp=l->tete;
     while (pp->succ <>NULL) pp = pp->succ;
     pp->succ = pe ; } /* chaîner dernier element */
  l->taille ++;}
```

Effacement , suppression d'un élément

- Comment identifier l'élément : sa valeur, son rang, sa place ?
- Effacer le premier d'une liste non vide

```
void efft(list *pli){
  elem *pp;
/*rechercher place de l'éléme */
  pp = pli->tete;
/*modifier liens */
  pli->tete= pp->succ;
/*supprimer la variable dynam */
  free(pp); pp = NULL;
/*cohérence de la liste */
  pli->taille -- ;}
```



attention à l'ordre !!!!

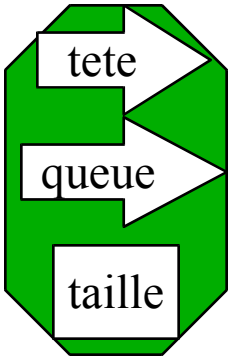
Gestion de listes

- Création de procédures
  - adjonction
  - suppression
  - taille (si elle n'est pas gérée directement)
  - parcours
    - recherche séquentielle
    - traitement
- Placer dans des bibliothèques
  - Pour gérer la représentation contiguë
  - Pour gérer la représentation chaînée
- Choisir la bibliothèque selon les caractéristiques de la liste
  - nombre d'éléments très variables
  - majorant inconnu etc.
  - C'est la généricité sur les types abstraits

Autres représentations en mode chaîné

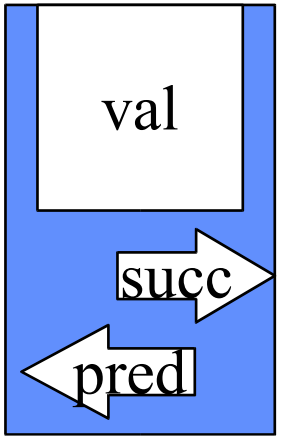
- Pour faciliter la gestion du dernier élément

```
typedef struct {float val;
                elem *succ;} elem ;
typedef struct {elem *tete ;
                elem *queue;
                int taille;} list;
```



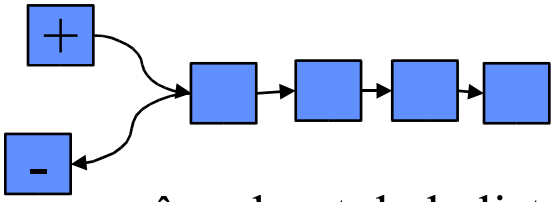
- Pour faciliter les parcours dans les deux sens

```
typedef struct {float val ;
                elem *succ;
                elem *pred ;} elem;
typedef struct {elem *tete ;
                elem *queue;
                int taille;} list;
```

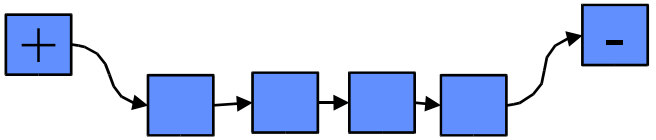


Listes particulières : Pile, File...

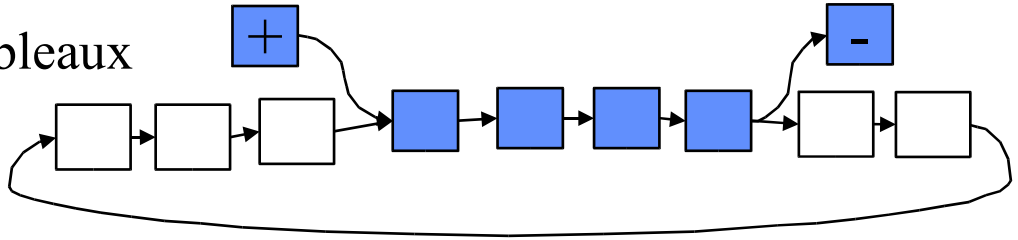
- Listes particulières
- pile
  - adjonction et suppression au même bout de la liste (début ou fin)
  - adjonction en tête, suppression en tête
  - dernier arrivé premier servi : Last In First Out (LIFO)



- file
  - adjonction et suppression à chaque bout
  - adjonction en tête (ou queue), suppression en queue (respectiv. tête)
  - premier arrivé, premier servi : First In First Out (FIFO)



- Listes circulaires
  - Implantée dans des tableaux
  - index varie modulo n



Exemple d'utilisation de double chaînage

- La représentation de matrices creuses
  - Eviter de représenter de nombreux 0 (90% d'éléments vides)
- Taille de la zone de donnée insuffisante pour déclarer plusieurs matrices
- Taille initiale
  - matrice n sur m
  - n*m*t(aille élém)
- Taille transformation

  t -> t+@(adresse)
- Têtes : (n+m) * @
- Total éléments + têtes

  0,1*n*m*(t+@) + (n+m)*@