Langage C

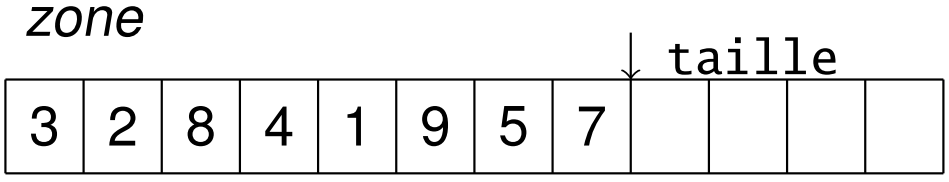
Les listes chaînées

L2 Mathématique et Informatique

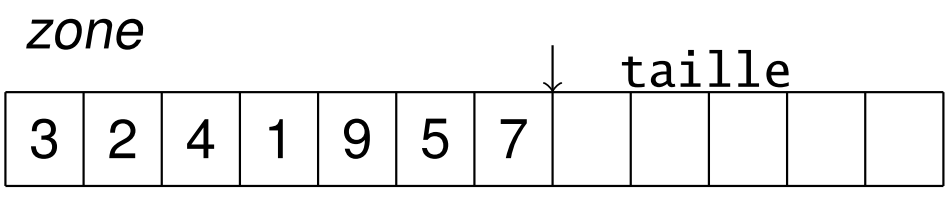
Université de Marne-la-Vallée

Tableaux dynamiques

L'utilisation de tableaux ne se justifie pas si de nombreux ajouts ou suppressions doivent avoir lieu sur des éléments quelconques du tableau. Ceux ci entrainent en effet de nombreux décalages pour créer ou supprimer une place libre, ce qui nuit à la rapidité.



La suppression de 8 entraine 5 décalages:

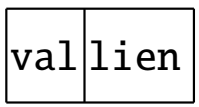


Liste chainée

Une solution consiste à utiliser des structures à deux champs : les cellules de la liste.
Chaque cellule contient un élément et un lien permettant l'accès à la cellule suivante.
On appellera liste le lien vers sa première cellule.
On a ainsi une structure de donnée qui calque la définition récursive de liste.
Une liste est

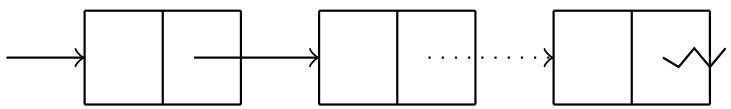
- soit vide,
- soit un élément suivi d'une liste.

cellule

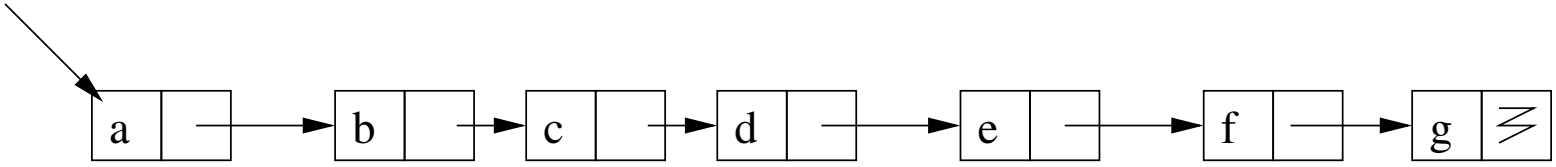


liste vide:lien sur rien ⏦

liste non vide - lien sur la première cellule

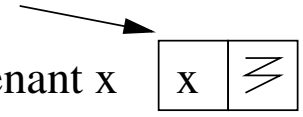


Utilisation de liste, insertion

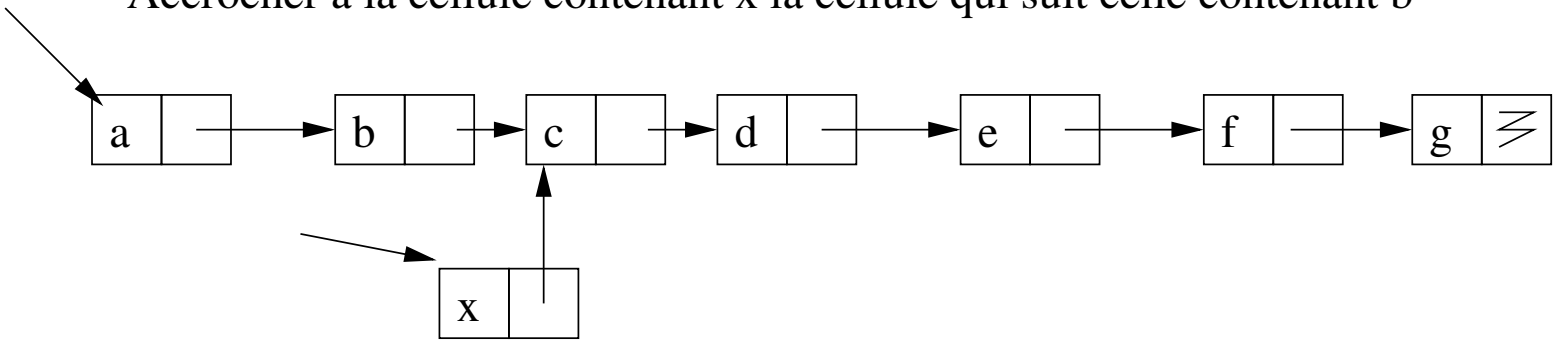


Insérer x après b:

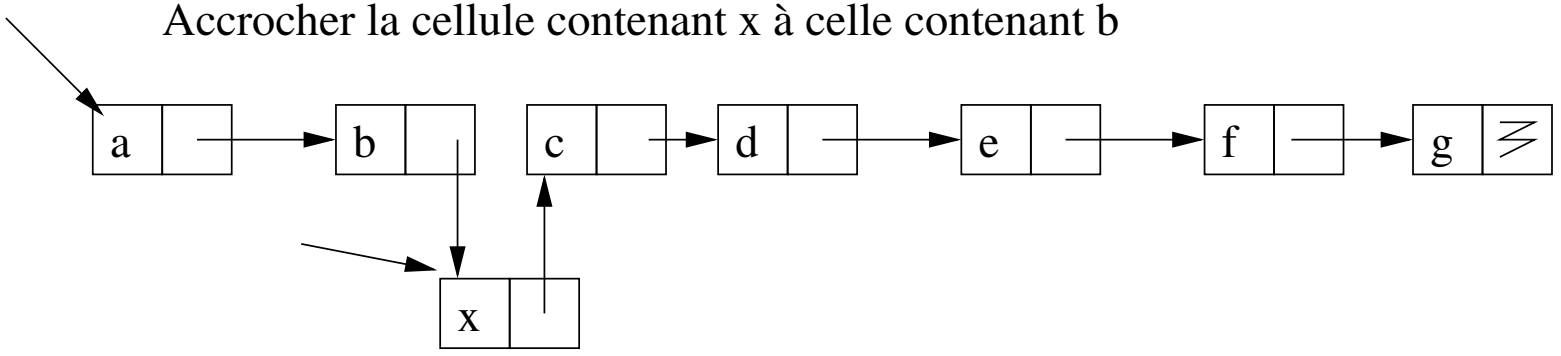
créer une cellule contenant x



Accrocher à la cellule contenant x la cellule qui suit celle contenant b



Accrocher la cellule contenant x à celle contenant b



Implantation par tableau, chaînage par indice

La mémoire utilisable est un tableau de cellules, le lien vers une cellule est son indice dans le tableau. Une liste est donc de type `int`

```
1  typedef struct{
2     TypeElement valeur;
3     int suivant;
4  } Cellule;
5  typedef int Liste;
```

On doit gérer la liste des cellules libres.

Liste l,libre;

l vaut 4

libre vaut 7

0		9
1	d	8
2	g	−1
3	c	1
4	a	6
5		−1
6	b	3
7		0
	e	10
9		5
10	f	2

Liste chaînée par pointeur

On travaille directement dans la mémoire, le lien vers la cellule suivante est son adresse. C'est cette implantation que nous développerons.
Définition du type
le lien est l'adresse d'une cellule,
le C autorise des définitions récursives pour les types récursifs:

```
1 typedef struct cel{
2   TypeElement valeur;
3   struct cel * suivant;
4 }Cellule;
```

Une liste est un lien sur une cellule:

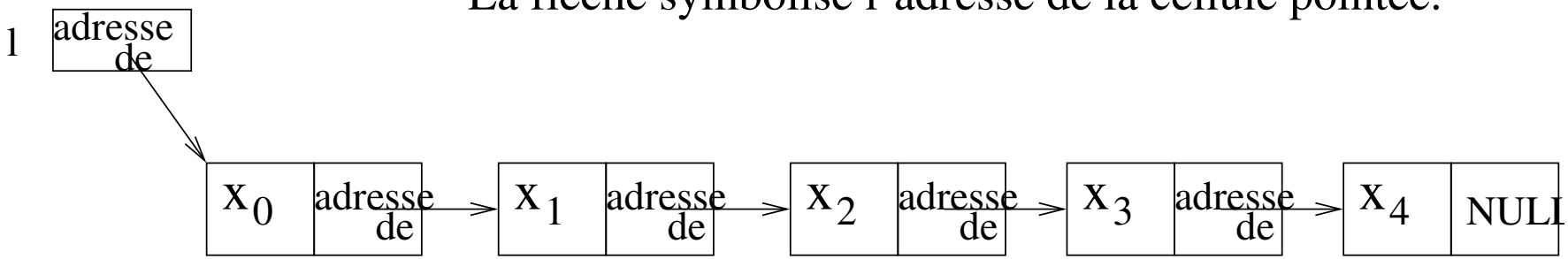
```
1 typedef Cellule *Liste;
```

une liste vide est un lien vers rien soit l'adresse `NULL`,
une liste non vide est un lien vers la première cellule.
Le champs `suivant` de la dernière cellule contient `NULL`.

Une liste vide

Une liste non vide

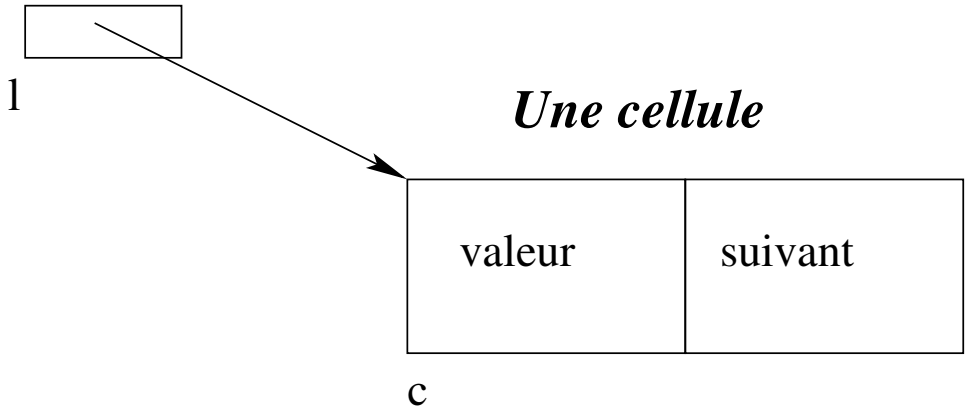
La flèche symbolise l'adresse de la cellule pointée.



Notation pour l'accès aux champs d'une cellule

```
1  Cellule c;Liste l;
2  L=&C;
```

Une liste



Les champs de `C` sont `C.valeur` et `C.suivant`,
`l` contient l'adresse de `c`, donc `*L` désigne c,
soit `(*L).valeur` et `(*L).suivant`. qui peuvent être écrit plus directement
`L->valeur` et `L->suivant`.

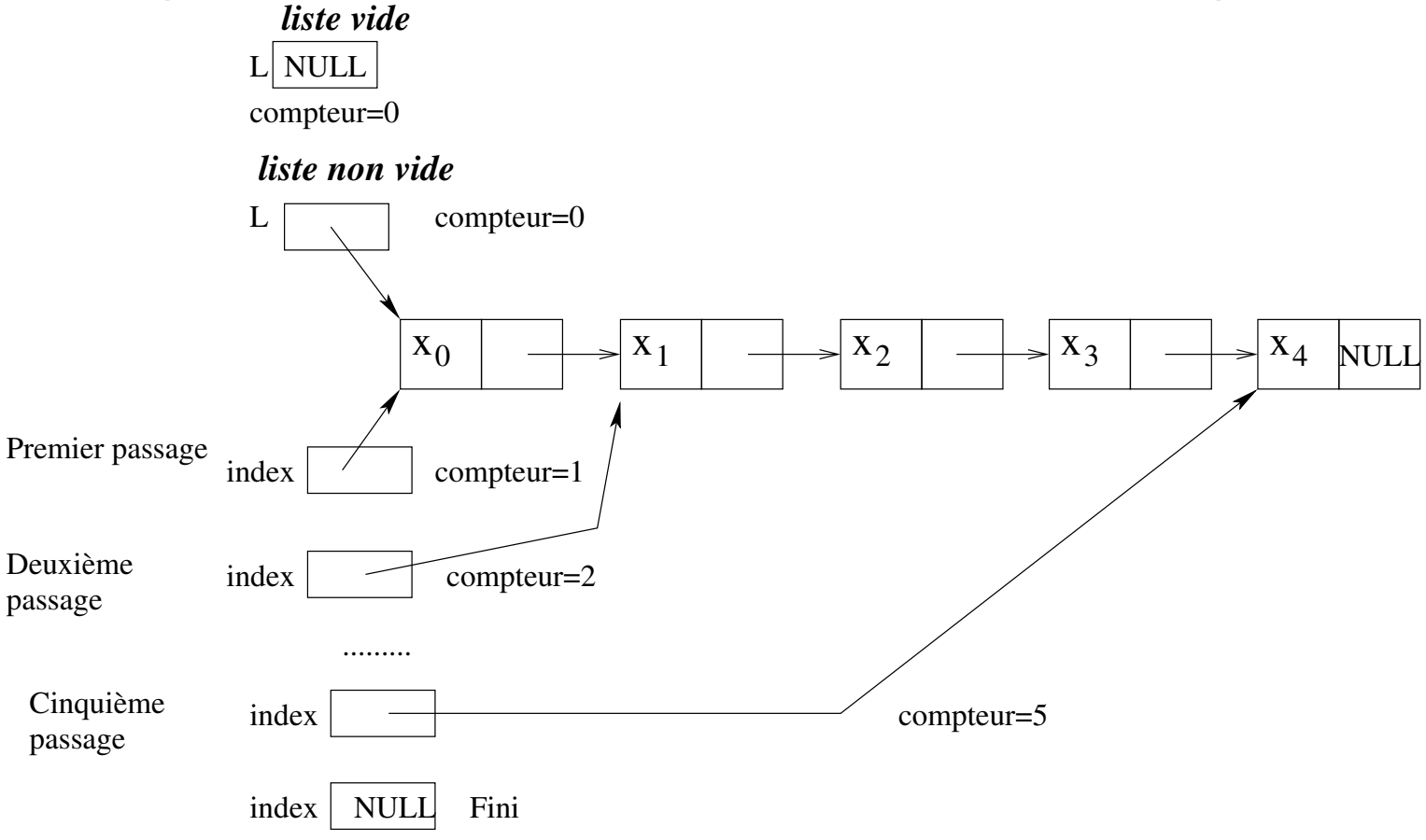
Manipulation de liste chainée par pointeur

Dans cette présentation, on choisit des listes d'entiers

```
1 typedef struct cel{
2   int valeur;
3   struct cel * suivant;
4 }Cellule,*Liste;
```

Compter les éléments de la liste

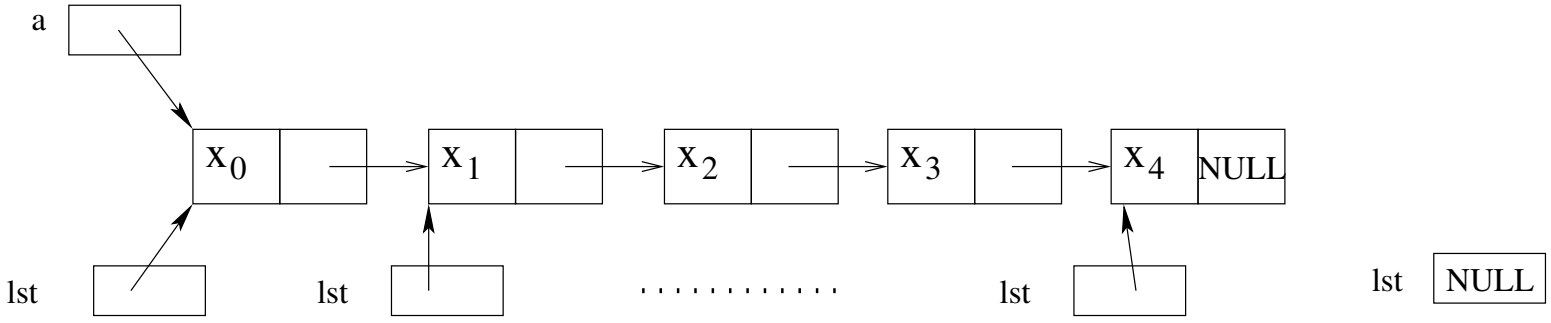
Un pointeur se déplace dans la liste d'une cellule à la cellule suivante, tant qu'il y a des cellules! .
tant que la fin de liste n'est pas atteinte
    on compte
    on passe à la cellule suivante en utilisant le champ `suivant`.



```
1  int compteListe(Liste lst){
2    Liste index=lst;
3    compteur=0;
4    while(index!= NULL){
5      compteur++;
6      index=index->suivant;
7    }
8    return compteur;
9  }
10
11 int main(void){
12   Liste a=NULL;
13   saisirListe(&a)
14   printf("nombre d'elements %d \n",compteListe(a));
15   return 0;
16 }
```

Paramètre

`lst` est une variable locale à la fonction `compteListe` .
La valeur du paramètre d'appel (`a`) ne peut pas être changée par la fonction. On peut utiliser directement `lst` pour se déplacer dans la liste:




```
1  int compteListe(Liste lst){
2     compteur=0;
3     while(lst!= NULL){
4        compteur++;
5        lst=lst->suivant;
6     }
7     return compteur;
8  }
```

mais les cellules sont envoyées par adresse.

Création d'une nouvelle cellule

Les cellules sont allouées dans le tas. Il faut réserver avec `malloc` la place nécessaire pour une cellule.
On écrit une fonction qui effectue cette demande de place. En cas d'allocation réussie elle remplit les champs, . La fonction renvoie l'adresse de la cellule crée ou `NULL`

```
1  Liste allouecellule(int val){
2    Liste tmp;
3  tmp=(Cellule *)malloc(sizeof(Cellule));
4  if(tmp!=NULL){
5     tmp->valeur=x;
6     tmp->suivant=NULL;
7     }
8  return tmp;
9  }
```

On pourrait affecter tester:

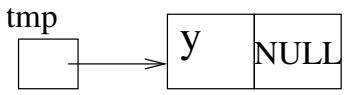
```
1  if((tmp=(Cellule *)malloc(sizeof(Cellule))) !=NULL)
```

Insertion d'un élément en tête de liste

Inserer y

demander la place pour une cellule

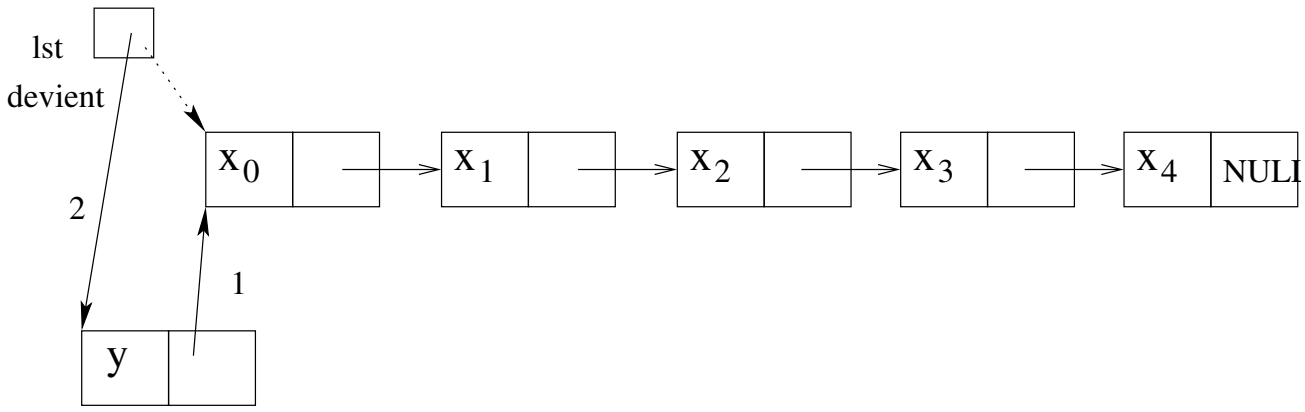
Allocation réussie



Ajout

liste vide

liste non vide



La cellule en tête de liste a changé.
L'adresse de la cellule qui est en tête de liste n'est donc plus la même.
La valeur de la liste, c'est à dire l'adresse de la première cellule, doit être changée par la fonction.
Il faut donc transmettre la liste par adresse.
Les actions à effectuer:

- créer une cellule contenant `y`;
- si cela a été possible, lui accrocher le début de liste en donnant au champs `suivant` la valeur de l'adresse de la première cellule de la liste;
- donner à la liste la valeur de l'adresse de la cellule contenant `y`

```
1  Liste insereEnTete(Liste *l,int y){
2  Liste tmp;
3  tmp=AlloueCellule(y);
4  if(tmp!=NULL){
5      tmp->suivant=*l;
6      *l=tmp;
7     }
8  return tmp;
9  }
```

Le fait de retourner `tmp` permet de savoir si l'insertion a été possible.

Extraction d'un élément de la liste

On choisit de récupérer la cellule pour avoir une fonction plus générale. C'est la fonction appellante qui décidera que faire de la cellule récupérée :

- l'intégrer à une autre liste
- la détruire
- ...

Extraire la cellule contenant y

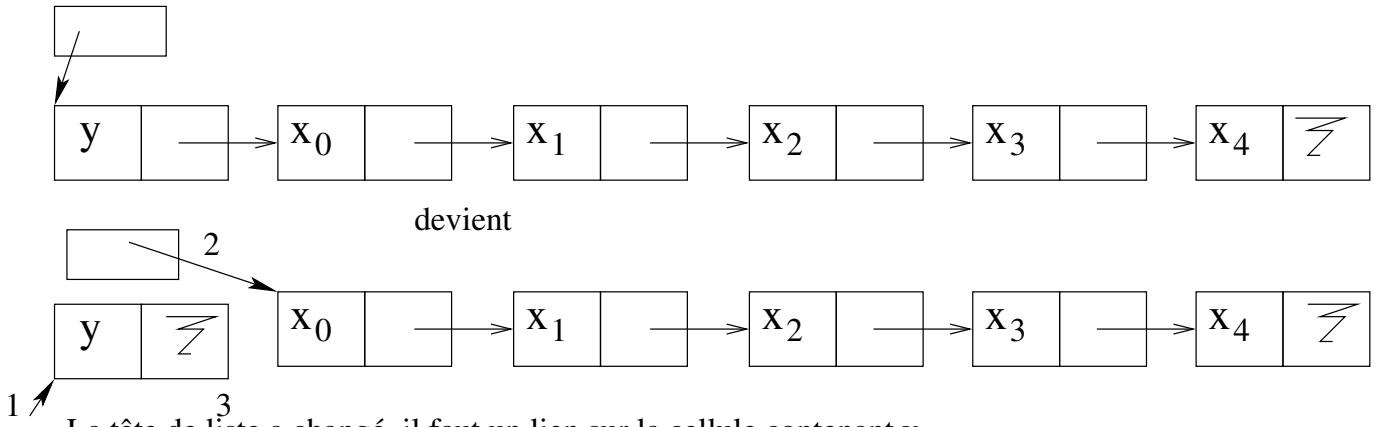
liste vide

NULL

Extraction impossible

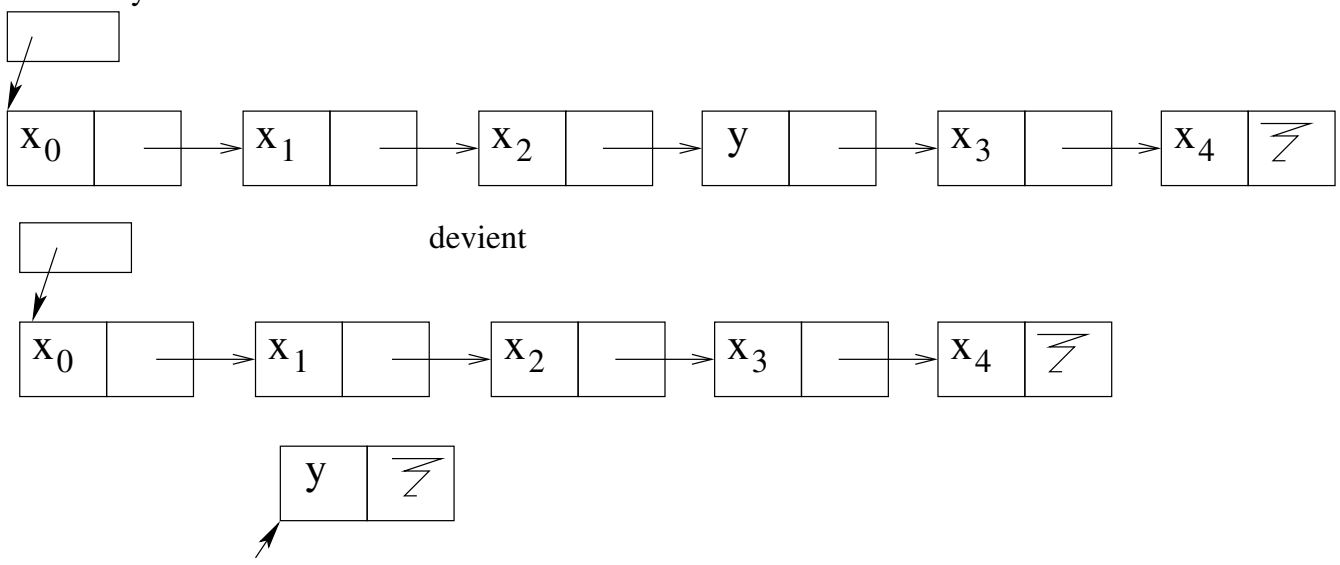
liste non vide

– cas 1: y est dans la cellule en début de liste

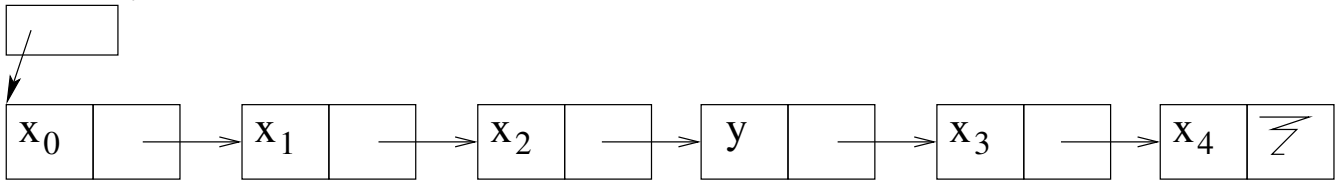


La tête de liste a changé, il faut un lien sur la cellule contenant y

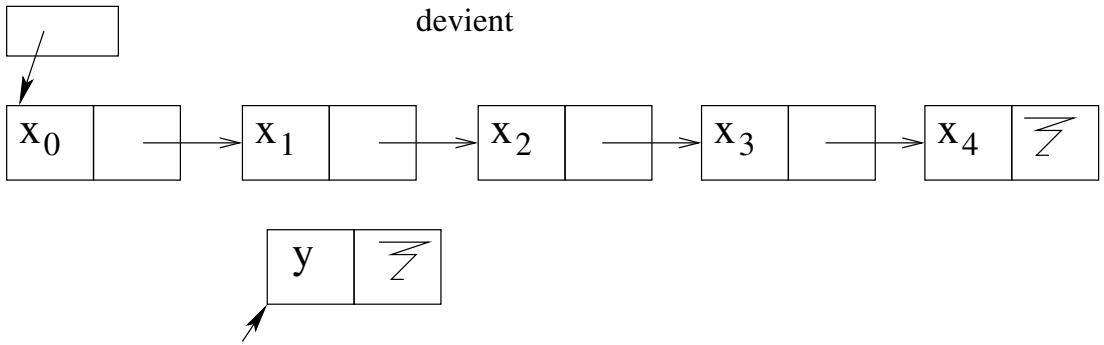
– cas 2: y est ailleurs dans la liste



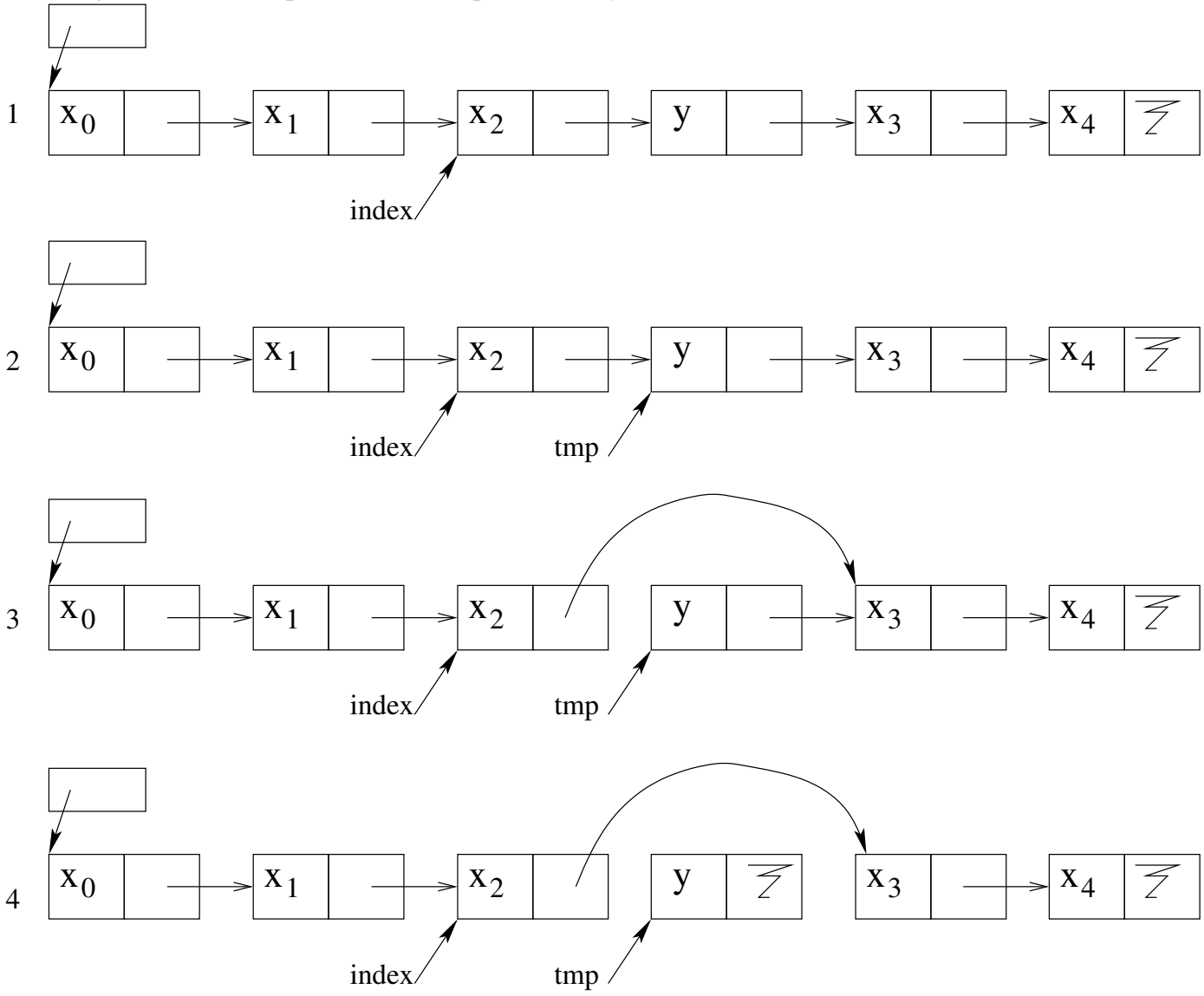
– cas 2: y est ailleurs dans la liste



devient



il faut agir sur la cellule précédant celle qui contient y



Extraire la cellule contenant y

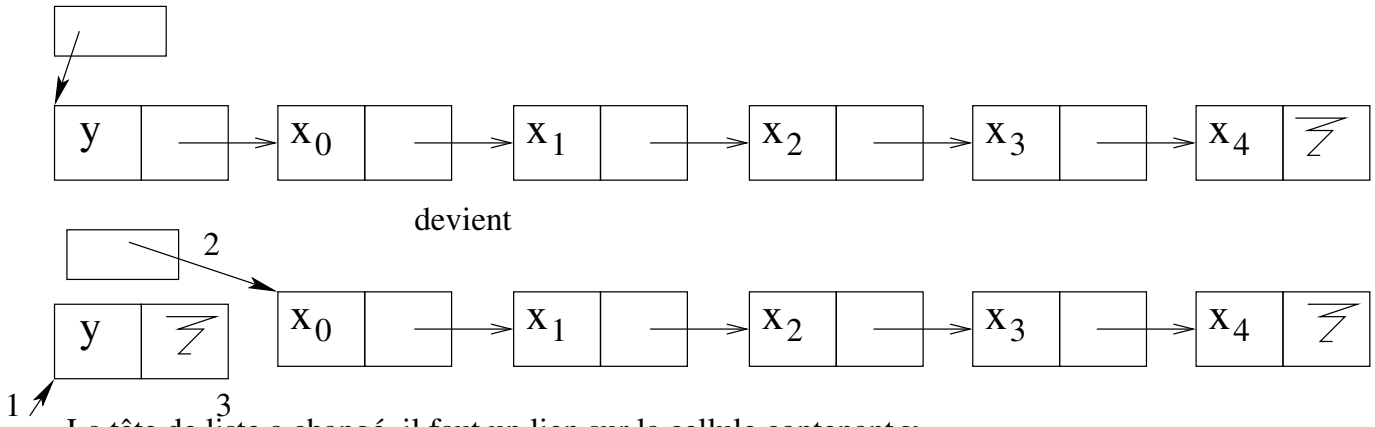
liste vide

NULL

Extraction impossible

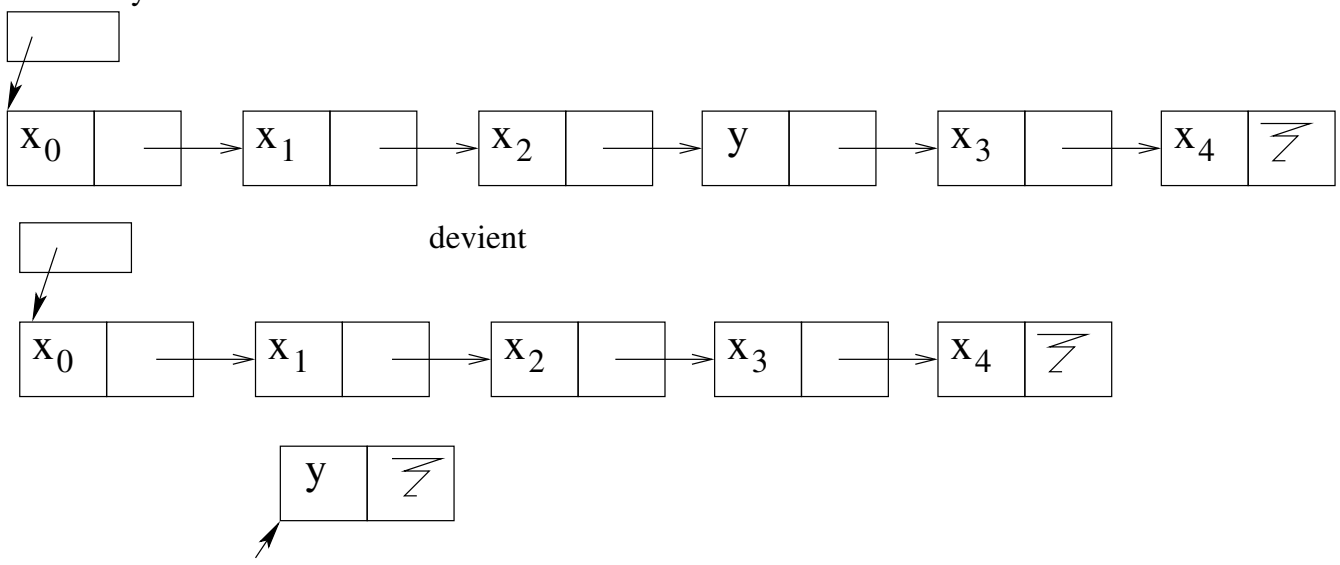
liste non vide

– cas 1: y est dans la cellule en début de liste

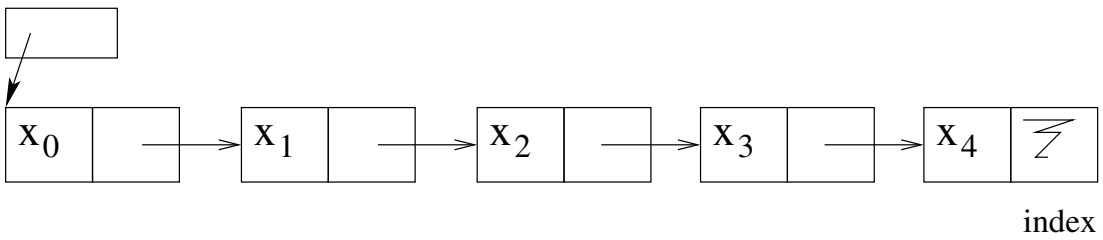


La tête de liste a changé, il faut un lien sur la cellule contenant y

– cas 2: y est ailleurs dans la liste



– Cas 3 : y n'est pas dans la liste



Il faut transmettre la liste par adresse (cas où l'élément est en tête de liste).
Les actions à effectuer:

- si la liste est vide echec;
- si la première cellule contient `y` on l'accroche à un lien `tmp`, la liste est l'adresse de la cellule suivante;
- parcourir la liste à l'aide d'un lien `index` commençant en début de liste tant que la cellule suivante existe et ne contient pas `y`;
- si on n'est pas en fin de liste, c'est qu'on a trouvé la cellule contenant `y`, on l'accroche à `tmp`, on accroche la précédente à la suivante ;
- si on a trouvé `y`, on met le champs suivant à `NULL`.

```
1  Liste Extraire(Liste *lst,int y){
2    Liste index=*lst,tmp=NULL;
3    if(*lst==NULL)
4      return NULL
5    if((*lst)->valeur)==y){
6      tmp=*lst;
7      *lst=(*lst)->suivant;
8      tmp->suivant=NULL;
9    }
10   else{
11     while(index->suivant!=NULL&&index->suivant->valeur!=
12         index=index->suivant ;
13     if(index->suivant!=NULL){
14       tmp=index->suivant ;
15       index->suivant=tmp->suivant;
16       tmp->suivant=NULL;
17     }
18   }
19   return tmp;
20 }
```

Liste et récursivité

Puisque les listes se définissent facilement récursivement, il est très facile d'écrire récursivement les opérations de manipulation de liste.

```
Si la liste est vide Fin
sinon
    agir sur la première cellule
    traiter par appel récursif sur la liste qui suit la
cellule
```

On a souvent une récursivité finale (ce qui revient à écrire un while avec un if et la pile. Dans certains cas cependant l'intérêt est important.

Nombre d'éléments

La liste vide a 0 éléments
sinon le nombre d'élément est 1 (première cellule) + le nombre d'éléments du reste de la liste.

```
1  void compteListe(Liste lst){
2     if(lst==NULL)
3        return 0;
4     return 1+compteListe(lst->suivant);
5     }
```

Recherche d'un élément

Si la liste est vide l'élément n'est pas présent.
S' il est dans la première cellule, on l'a trouvé;
Sinon le résultat est celui de la recherche dans le reste de la liste.

```
1  int Recherche(Liste lst,int x){
2     if(lst==NULL)
3        return 0;
4     if(lst->valeur ==x)
5        return 1;
6     return Recherche(lst->suivant, x);
7     }
```

On pourrait plutot retourner l'adresse de la cellule contenant `x`, ou `NULL`

Affichage des éléments

Si la liste n'est pas vide
afficher le contenu de la première cellule
afficher le reste de la liste

```
1  void afficheListe(Liste lst){
2     if(lst !=NULL){
3        printf("%d␣",lst->valeur);
4        afficheListe(lst->suivant);
5     }
6     else
7        printf("\n");
8  }
```

Affichage des éléments en ordre inverse

Dans tous les exemples précédents, l'utilisation de récursivité est inutile.
Résoudre celui-ci sans récursivité nécessite une pile.
Si la liste n'est vide
afficher le reste de la liste en ordre inverse afficher l'élément de la cellule

```
1  void afficheInverse(Liste lst){
2     if(lst !=NULL){
3          afficheInverse(lst->suivant)
4          printf("%d ",lst->valeur);
5     }
6  }
```

Où placer le passage à la ligne?

Le plus simple est d'effectuer le passage à la ligne avec une autre fonction

```
1  void afficheInverseAux(Liste lst){
2     if(lst !=NULL){
3        afficheInverseAux(lst->suivant);
4        printf("%d␣",lst->valeur);
5     }
6  }
7  void afficheInverse(Liste lst){
8
9        afficheInverseAux(lst)
10       printf("\n");
11    }
12 }
```