Initiation au C
cours n°10

Antoine Miné

École normale supérieure

10 mai 2007

Plan du cours

- pointeurs de fonctions,
- structures de données dynamiques :
  - listes simplement chaînées,
  - listes doublement chaînées.

Pointeurs de fonctions

Rappels sur les prototypes de fonctions

Prototype = déclaration précisant :

- le nom de la fonction,
- son type de retour, ou `void`,
- le nombre et le type de ses arguments.

Exemples

```
int additionne(int x, int y) ;
void affiche(char*) ;
```

Différences avec une définition de fonction :

- le corps est omis (remplacé par ;),
- le nom des arguments est facultatif,
- une fonction peut être déclarée plusieurs fois.

Rappels sur les prototypes de fonctions

Utilité du prototype :
information nécessaire pour compiler un appel de fonction.

Si un prototype est donné, la fonction peut être définie

- après l'appel, ou
- dans une autre unité de compilation.

Exemple :

a.c

```
int fun1(int x);

int fun2(void)
{ return fun1(1); }

int fun1(int x)
{ return x+1; }
```

b.c

```
int fun2(void);

int main()
{ return fun2(); }
```

Pointeurs de fonctions

Chaque fonction a une adresse en mémoire.

Pointeur de fonction = variable

- contenant l'adresse d'une fonction,
- dont le type indique le prototype de la fonction.

Déclaration d'un pointeur de fonction `ptr`

```
type-ret (*ptr)(type1,...,typeN) ;
```

`ptr` peut contenir l'adresse de toute fonction

- prenant `N` arguments, de types `type1` à `typeN`, et
- retournant une valeur de type `type-ret`.

Affectation de pointeur de fonction

Affectation : `ptr = f ;` où

- `f` est le nom d'une fonction de prototype compatible avec `ptr`,
- `f` est un pointeur de fonction de même type que `ptr`.

Exemples d'affectation

```
int truc(int x)
{ ... }

int bidule(int y);

void main()
{
  int (*ptr)(int);
  ptr = truc;
  ptr = bidule;
}
```

Appel de fonction par pointeur

Appel par pointeur : identique à un appel de fonction classique

- `ptr(arg1,...,argN);` ou
- `x = ptr(arg1,...,argN);`

Exemple d'appel

```
int bidule(int y);

int main()
{
  int (*ptr)(int);
  ptr = bidule;
  return ptr(2);
}
```

Application : fonction d'ordre supérieur

Fonction paramétrée par une fonction.

$\Longrightarrow$ on passe un pointeur de fonction en argument.

Exemple

```
void affiche_tableau(char* tab[], void (*f)(char*)) {
  int i;
  for (i=0; tab[i]; i++)  f(tab[i]);
}

void affiche(char* s)
{ printf("%s\n", s); }

void main() {
  char* t[] = { "toto", "titi", NULL };
  affiche_tableau(t, affiche);
}
```

Listes simplement chaînées

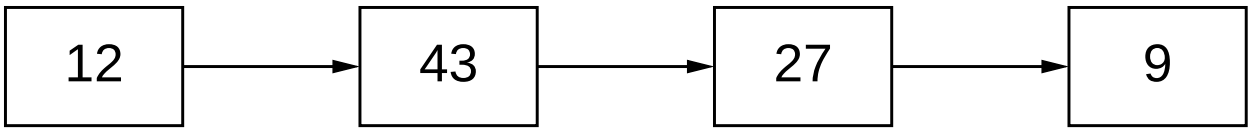
Principe

Liste = séquence ordonnée d'éléments de même type.

Exemple : liste d'entiers $(12, 43, 27, 9)$.

- l'ordre des éléments compte : $(12, 43, 27, 9) \neq (12, 27, 43, 9)$
- la multiplicité des éléments compte $(12, 43, 27, 9) \neq (12, 43, 27, 9, 9)$

Liste simplement chaînée =
représentation où chaque élément pointe sur le suivant.



Représentation des listes

Implantation en C =

structure de cellule pour représenter un élément.

Type cellule

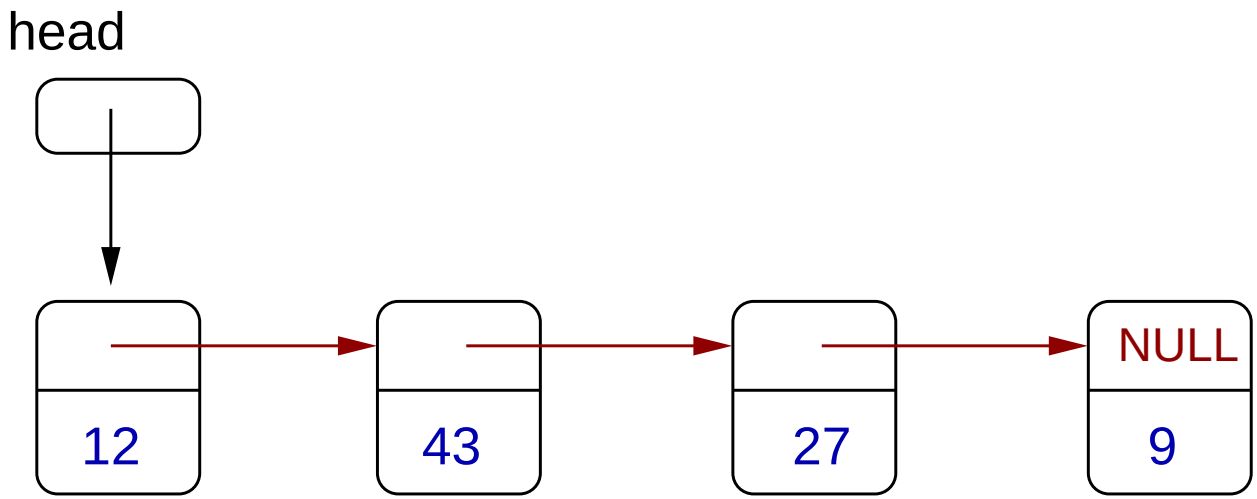
```
struct cell {
    struct cell* next;
    int          data;
};
```

- `data` est le contenu de la cellule, (ici, un entier)
- `next` pointe vers la cellule suivante,
  ou vaut `NULL` (fin de liste).

Le type de `cell` est récursif (autoréférentiel).

Représentation des listes

Une liste est représentée par un **pointeur de tête** `struct cell*` = pointeur sur la première cellule.



Tous les éléments sont accessibles depuis la tête de liste.

Par convention, `head` vaut `NULL` si la liste est vide.

Opérations sur les listes

Structure de données = type + algorithmes de manipulation.

On va développer des fonctions pour les opérations suivantes :

- calcul de la longueur d'une liste,
- recherche d'un élément,
- insertion d'un élément,
- suppression d'un élément,
- concaténation de deux listes,
- destruction d'une liste.

Toutes nos fonctions prennent une tête de liste en argument.

Calcul de la longueur d'une liste

Longueur d'une liste

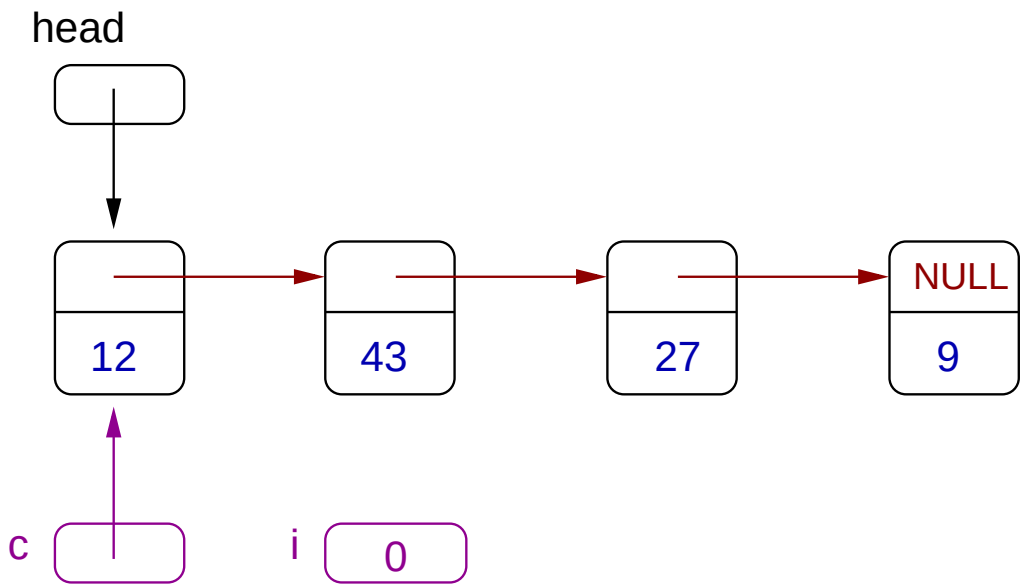
```
int longueur(struct cell* head) {
  int i; struct cell* c;
  for ( i=0, c=head; c; i++, c = c->next );
  return i;
}
```

Principe : on suit les pointeurs `next` jusqu'à rencontrer `NULL` et on compte le nombre de cellules rencontrées.

Calcul de la longueur d'une liste

Longueur d'une liste

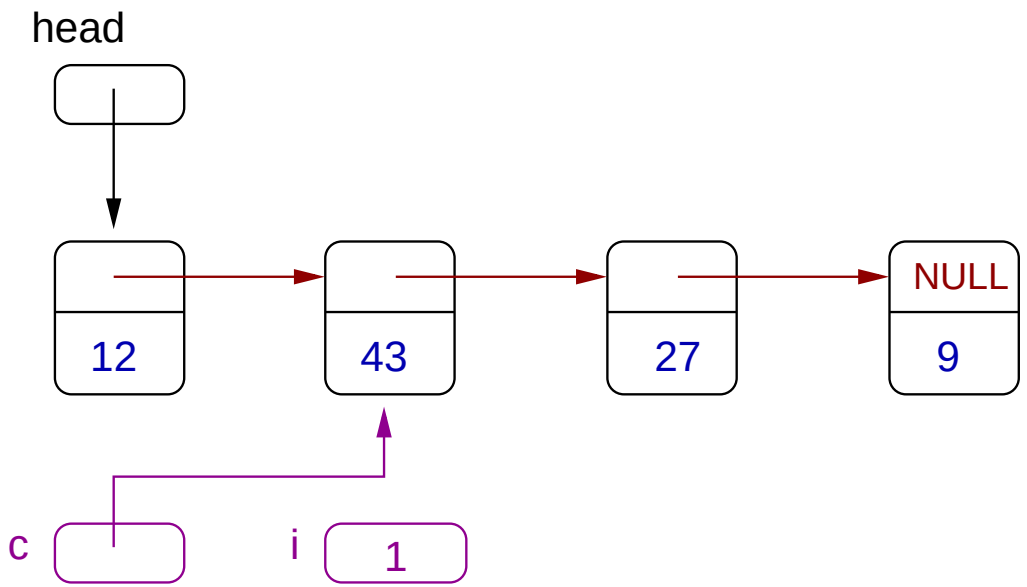
```
int longueur(struct cell* head) {
  int i; struct cell* c;
  for ( i=0, c=head; c; i++, c = c->next );
  return i;
}
```



Calcul de la longueur d'une liste

Longueur d'une liste

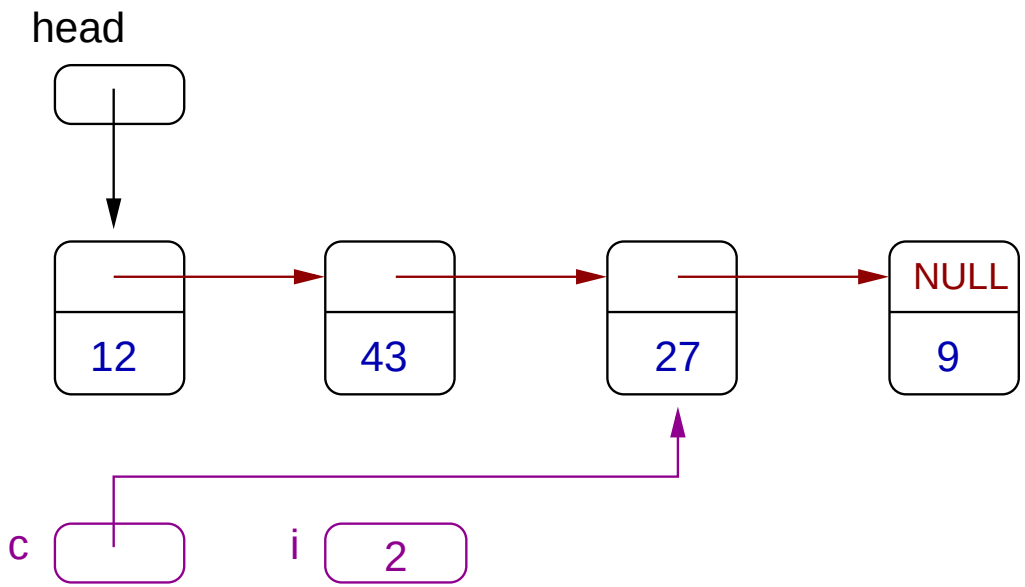
```
int longueur(struct cell* head) {
  int i; struct cell* c;
  for ( i=0, c=head; c; i++, c = c->next );
  return i;
}
```



Calcul de la longueur d'une liste

Longueur d'une liste

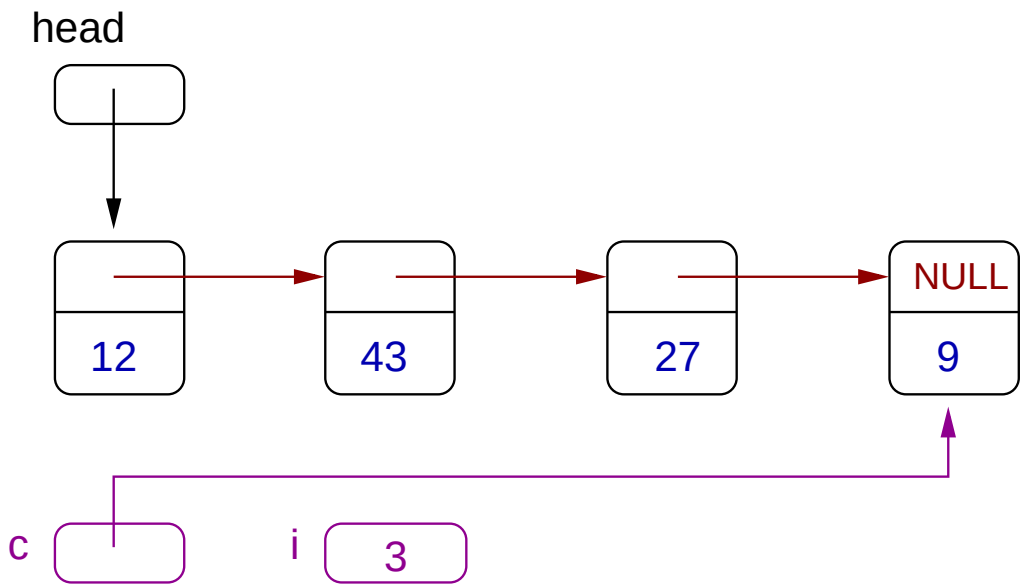
```
int longueur(struct cell* head) {
  int i; struct cell* c;
  for ( i=0, c=head; c; i++, c = c->next );
  return i;
}
```



Calcul de la longueur d'une liste

Longueur d'une liste

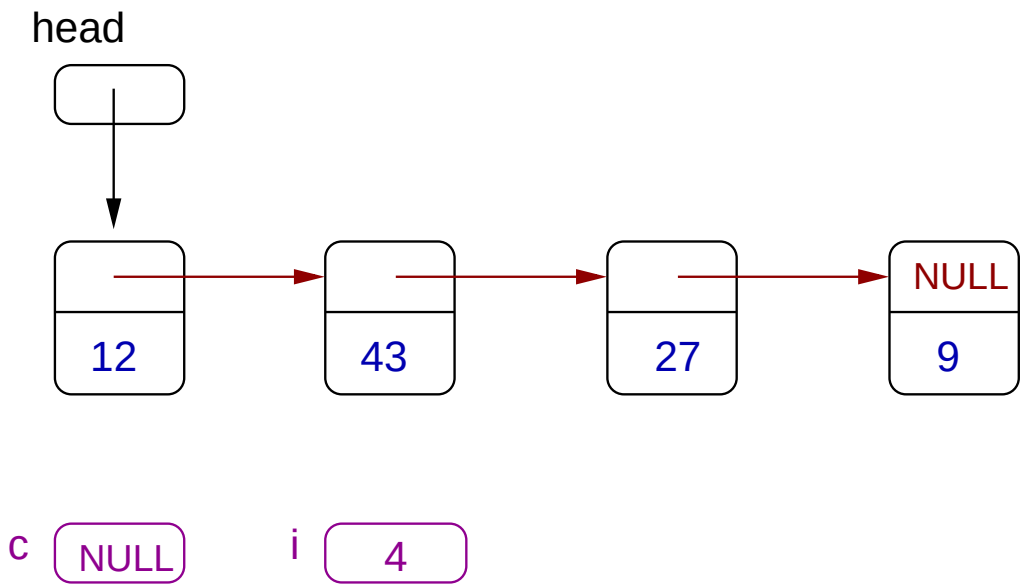
```
int longueur(struct cell* head) {
  int i; struct cell* c;
  for ( i=0, c=head; c; i++, c = c->next );
  return i;
}
```



Calcul de la longueur d'une liste

Longueur d'une liste

```
int longueur(struct cell* head) {
  int i; struct cell* c;
  for ( i=0, c=head; c; i++, c = c->next );
  return i;
}
```



Recherche d'un élément

Recherche de `elem`

```
int recherche(struct cell* head, int elem)
{
  while (head) {
    if (head->data == elem) return 1;
    head = head->next;
  }
  return 0;
}
```

Effet :
renvoie 1 si la liste contient un élément égal à `elem`, 0 sinon.

Rappels sur la mémoire dynamique

Pour plus de flexibilité, les cellules sont allouées dynamiquement.

Gestion de la mémoire dynamique

```
#include <stdlib.h>
void* malloc (size_t size);
void  free   (void*  ptr);
```

Effet :

- `malloc` alloue un bloc de `size` octets,
- `free` libère le bloc,
- le bloc est accessible uniquement par pointeur.

Création d'une liste (exemple à ne pas suivre)

Allocation de $(12, 43, 27, 9)$

```
struct cell* head;
head = malloc(sizeof(struct cell));
head->data = 12;
head->next = malloc(sizeof(struct cell));
head->next->data = 43;
head->next->next = malloc(sizeof(struct cell));
head->next->next->data = 27;
head->next->next->next = malloc(sizeof(struct cell));
head->next->next->next->data = 9;
head->next->next->next->next = NULL;
```

Peu pratique.

On préfère construire une liste par insertions successives.

Insertion en tête de liste

Insertion de e en tête de h

```
struct cell* insere(struct cell* h, int e) {
  struct cell* c = malloc(sizeof(struct cell));
  c->data = e;
  c->next = h;
  return c;
}
```

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Illustration de l'insertion en tête de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Insertion

```
struct cell* insere
 (struct cell* h, int e) {
  struct cell* c = malloc(...);
  c->data = e;
  c->next = h;
  return c;
}
```

Illustration de l'insertion en tête de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Insertion

```
struct cell* insere
 (struct cell* h, int e) {
  struct cell* c = malloc(...);
  c->data = e;
  c->next = h;
  return c;
}
```

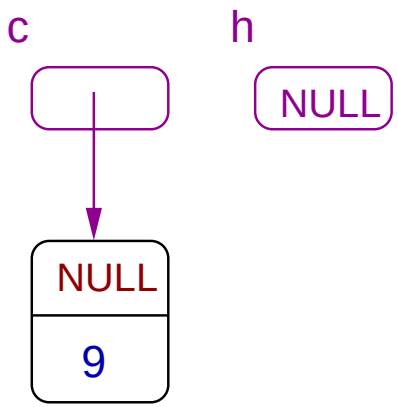


Illustration de l'insertion en tête de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Insertion

```
struct cell* insere
 (struct cell* h, int e) {
  struct cell* c = malloc(...);
  c->data = e;
  c->next = h;
  return c;
}
```

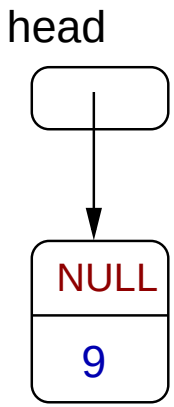


Illustration de l'insertion en tête de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Insertion

```
struct cell* insere
 (struct cell* h, int e) {
  struct cell* c = malloc(...);
  c->data = e;
  c->next = h;
  return c;
}
```

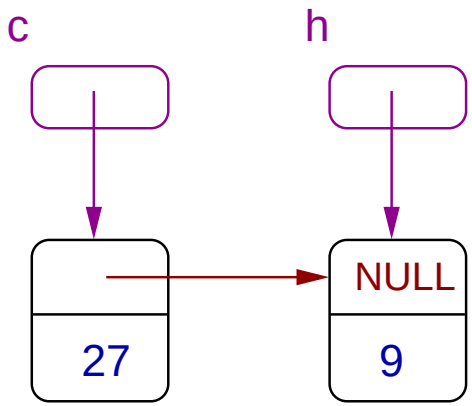


Illustration de l'insertion en tête de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Insertion

```
struct cell* insere
 (struct cell* h, int e) {
  struct cell* c = malloc(...);
  c->data = e;
  c->next = h;
  return c;
}
```

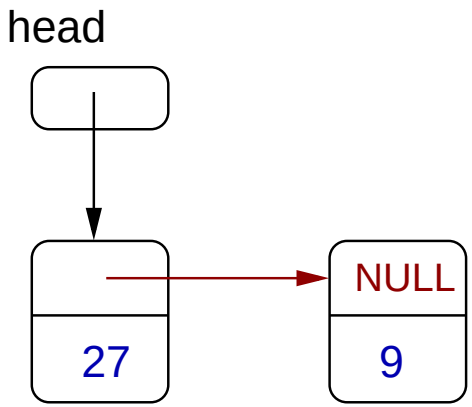


Illustration de l'insertion en tête de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Insertion

```
struct cell* insere
 (struct cell* h, int e) {
  struct cell* c = malloc(...);
  c->data = e;
  c->next = h;
  return c;
}
```

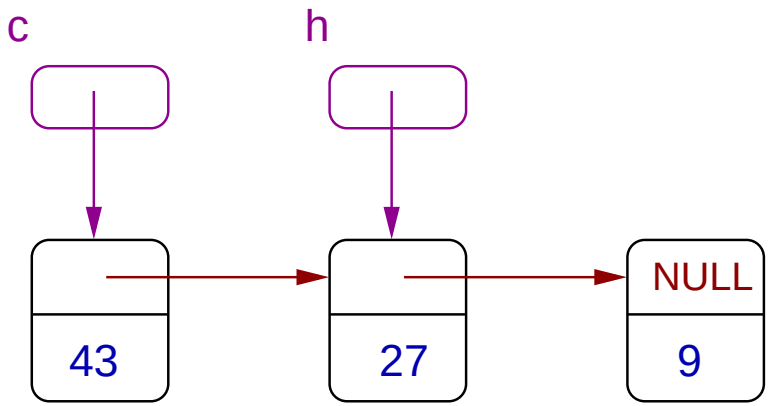


Illustration de l'insertion en tête de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Insertion

```
struct cell* insere
 (struct cell* h, int e) {
  struct cell* c = malloc(...);
  c->data = e;
  c->next = h;
  return c;
}
```

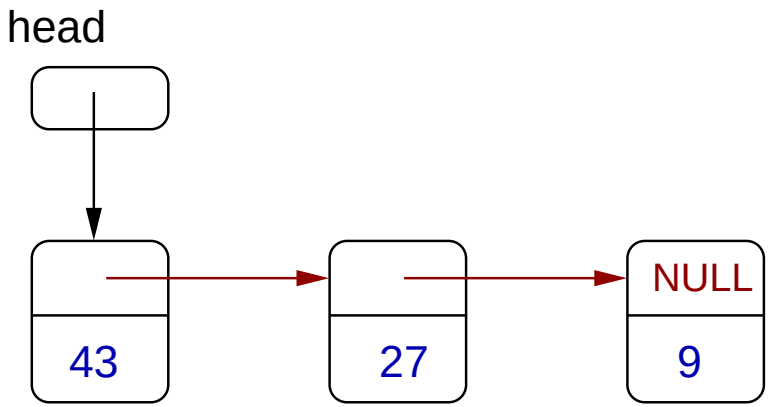


Illustration de l'insertion en tête de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Insertion

```
struct cell* insere
 (struct cell* h, int e) {
  struct cell* c = malloc(...);
  c->data = e;
  c->next = h;
  return c;
}
```

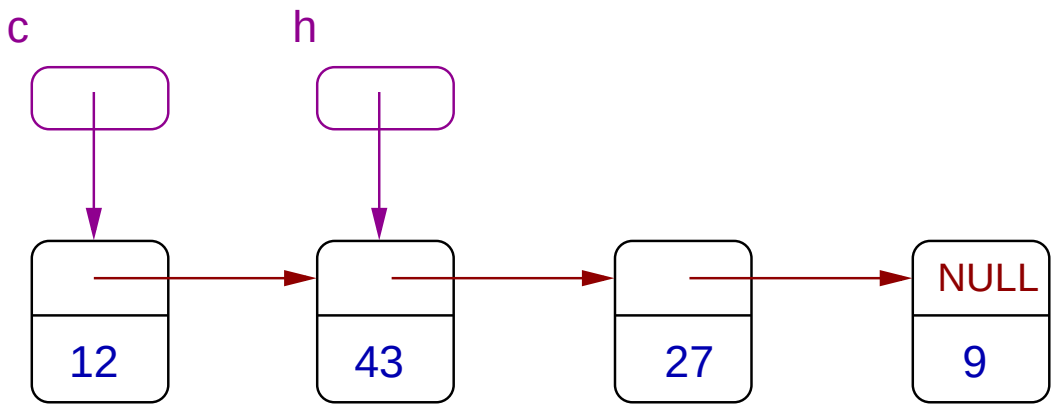


Illustration de l'insertion en tête de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Insertion

```
struct cell* insere
 (struct cell* h, int e) {
  struct cell* c = malloc(...);
  c->data = e;
  c->next = h;
  return c;
}
```

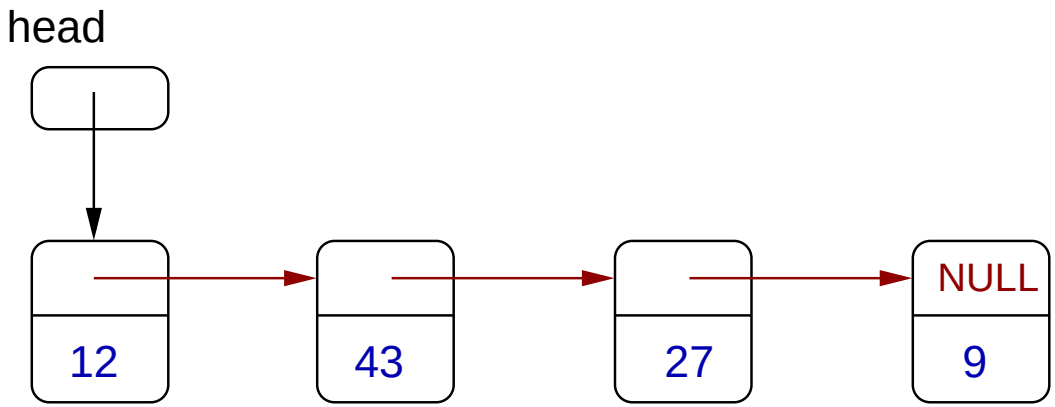


Illustration de l'insertion en tête de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head,  9);
head = insere(head, 27);
head = insere(head, 43);
head = insere(head, 12);
```

Insertion

```
struct cell* insere
 (struct cell* h, int e) {
  struct cell* c = malloc(...);
  c->data = e;
  c->next = h;
  return c;
}
```

Remarques :

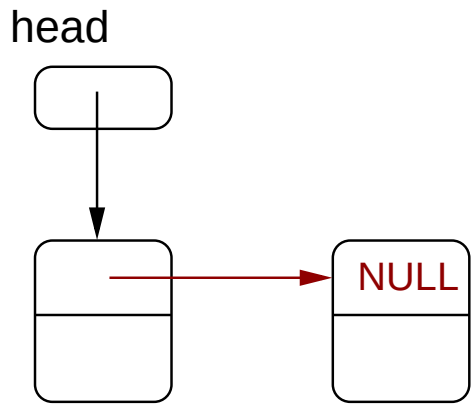
- la liste est dans l'ordre inverse de celui des insertions,
- `insere` fonctionne sur une liste vide ou non-vide,
- la tête de liste est modifiée à chaque insertion,
- le coût d'une insertion est constant.

Insertion en queue de liste

Deux cas à considérer :

- liste vide : on fait pointer la tête de liste sur la nouvelle cellule,
- liste non vide : on fait pointer le champ `next` de la dernière cellule sur la nouvelle cellule.

liste vide



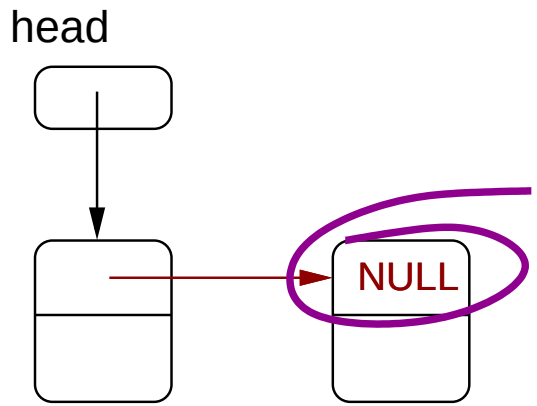
liste non vide

Insertion en queue de liste

Deux cas à considérer :

- liste vide : on fait pointer la tête de liste sur la nouvelle cellule,
- liste non vide : on fait pointer le champ `next` de la dernière cellule sur la nouvelle cellule.

liste vide



liste non vide

Insertion en queue de liste

Insertion en queue

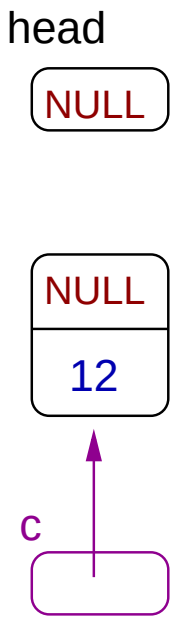
```
struct cell* insere(struct cell* head, int elem)
{
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = NULL;

  if (head) {
    struct cell* l = head;
    while (l->next) l = l->next;
    l->next = c;
    return head;
  }
  else return c;
}
```

Exemple d'insertion en queue de liste

Utilisation

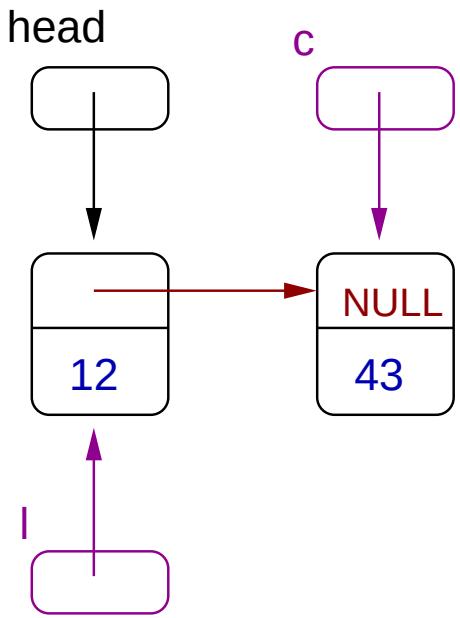
```
struct cell* head = NULL;
head = insere(head, 12);
head = insere(head, 43);
head = insere(head, 27);
head = insere(head,  9);
```



Exemple d'insertion en queue de liste

Utilisation

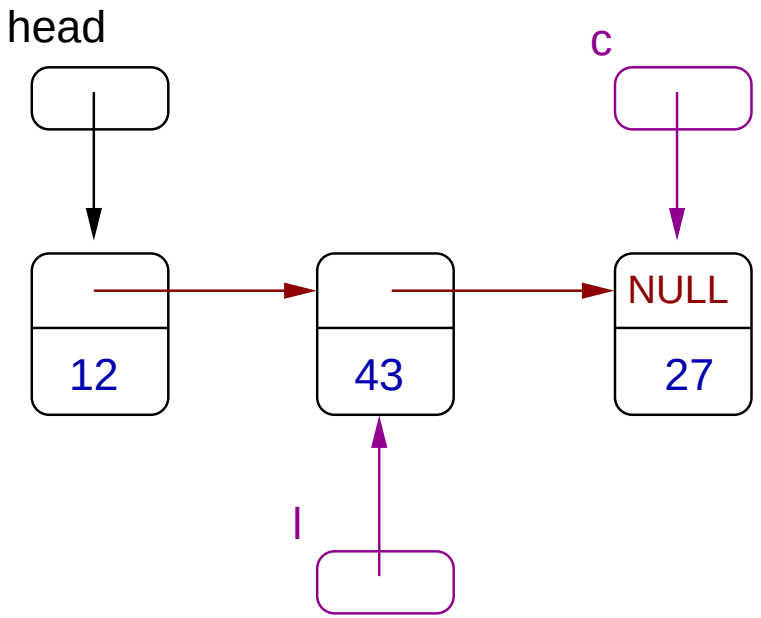
```
struct cell* head = NULL;
head = insere(head, 12);
head = insere(head, 43);
head = insere(head, 27);
head = insere(head,  9);
```



Exemple d'insertion en queue de liste

Utilisation

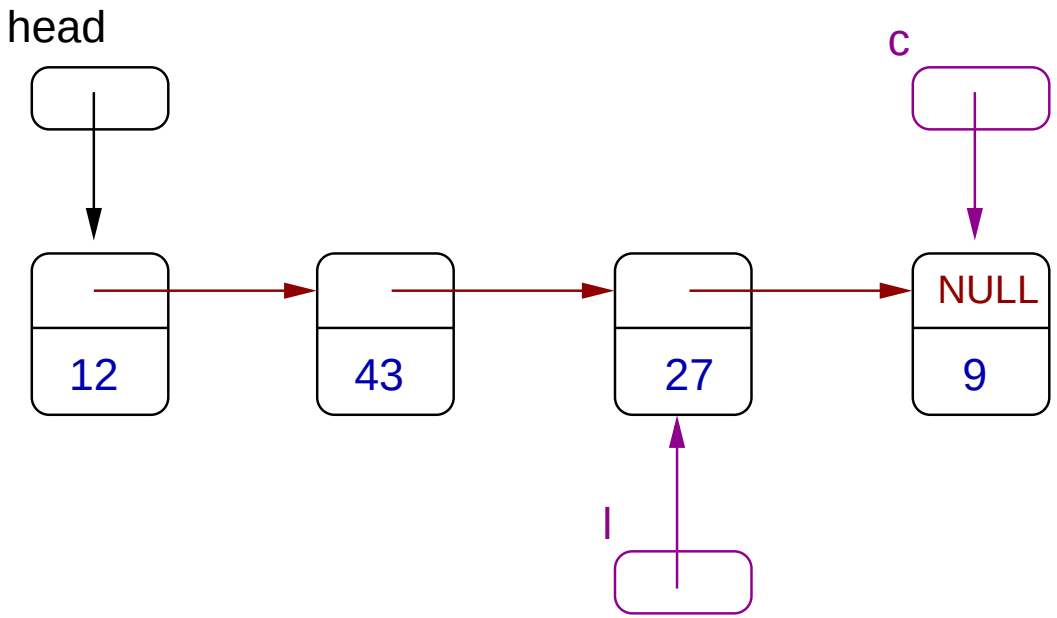
```
struct cell* head = NULL;
head = insere(head, 12);
head = insere(head, 43);
head = insere(head, 27);
head = insere(head,  9);
```



Exemple d'insertion en queue de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head, 12);
head = insere(head, 43);
head = insere(head, 27);
head = insere(head,  9);
```



Exemple d'insertion en queue de liste

Utilisation

```
struct cell* head = NULL;
head = insere(head, 12);
head = insere(head, 43);
head = insere(head, 27);
head = insere(head,  9);
```

Remarques :

- la liste est dans le même ordre que celui des insertions,
- la tête de liste n'est modifiée que lors de la première insertion,
- le coût d'une seule insertion est linéaire, $\mathcal{O}(n)$
  le coût de la construction est quadratique. $\mathcal{O}(n^2)$

Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

Insertion après `pred`

```
void insere_apres(struct cell* pred, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = pred->next;
  pred->next = c;
}
```

Effet : insère `elem` après la cellule pointée par `pred`.

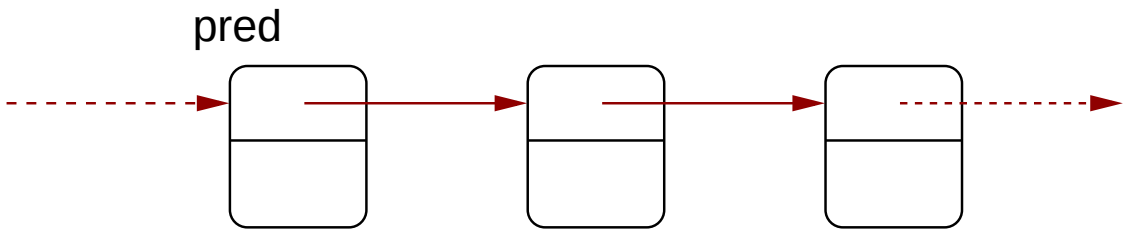
L'insertion *avant* une cellule donnée est plus complexe...

Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

Insertion après pred

```
void insere_apres(struct cell* pred, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = pred->next;
  pred->next = c;
}
```

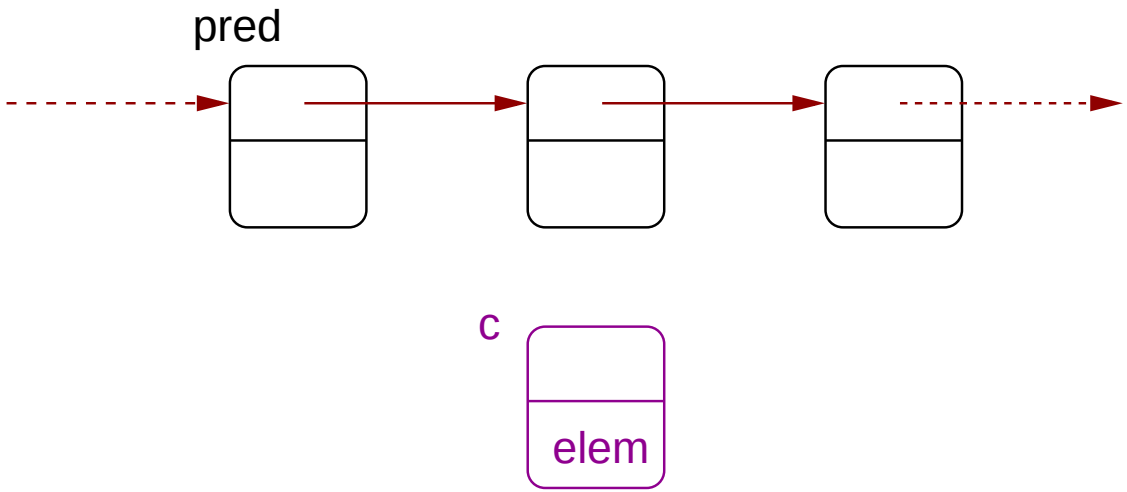


Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

Insertion après pred

```
void insere_apres(struct cell* pred, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = pred->next;
  pred->next = c;
}
```

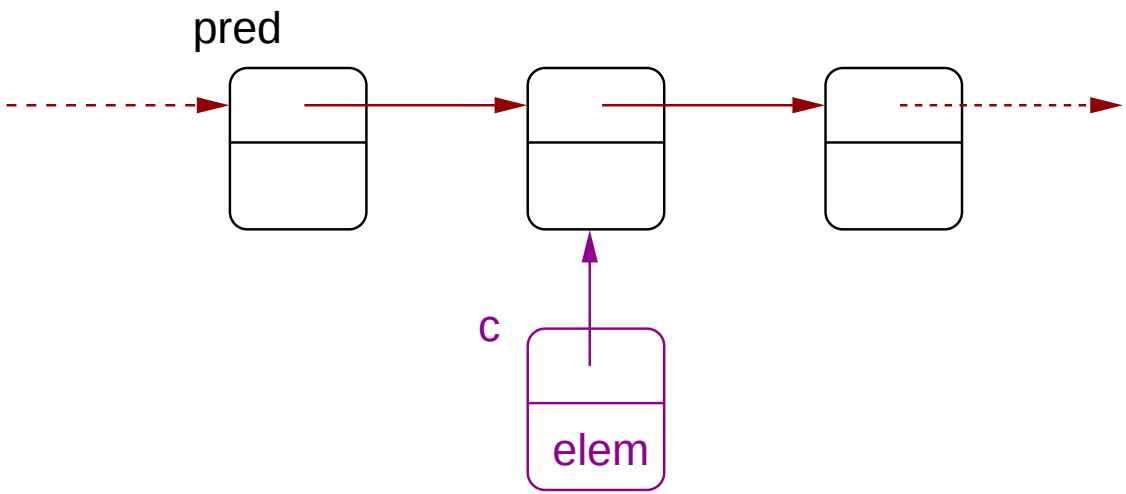


Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

Insertion après pred

```
void insere_apres(struct cell* pred, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = pred->next;
  pred->next = c;
}
```

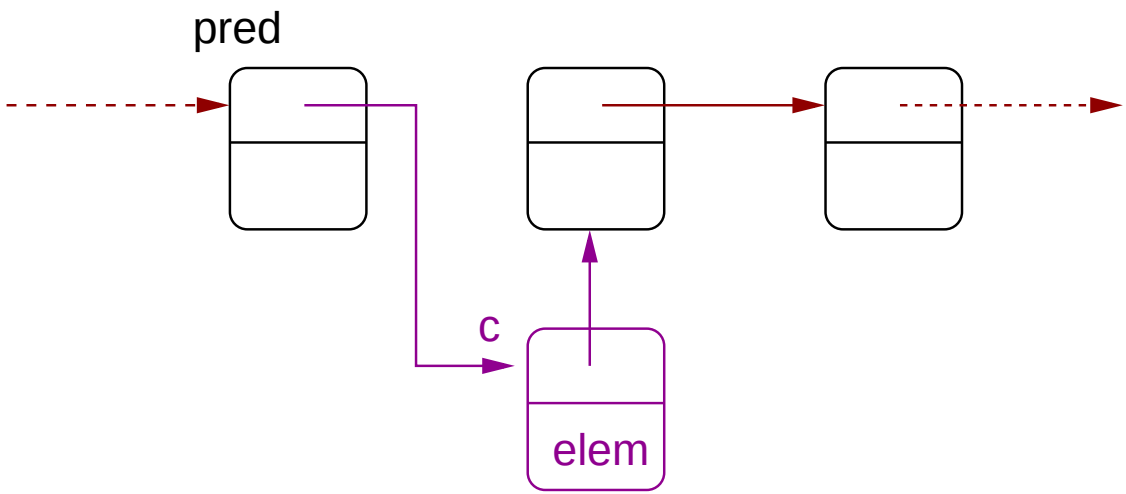


Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

Insertion après pred

```
void insere_apres(struct cell* pred, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = pred->next;
  pred->next = c;
}
```



Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

Insertion après `pred`

```
void insere_apres(struct cell* pred, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = pred->next;
  pred->next = c;
}
```

Notes :

- coût constant, hors calcul de `pred`,
- `pred` peut être obtenu par une variante de `recherche`,
- on suppose qu'on n'insère pas en première position.
  ($\Rightarrow$ on suppose que la liste n'est pas vide)

Suppression d'un élément

Suppression facile si on a un pointeur vers la cellule précédente.

Suppression après `pred`

```
void supprime_apres(struct cell* pred) {
  struct cell *c = pred->next;
  pred->next = c->next;
  free(c);
}
```

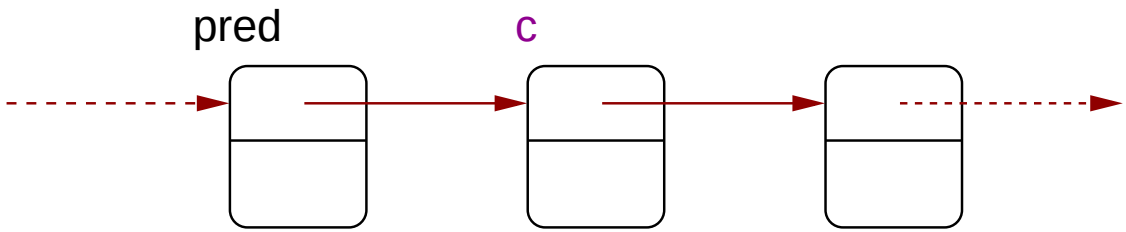
Effet : supprime la cellule suivant `pred`.

Suppression d'un élément

Suppression facile si on a un pointeur vers la cellule précédente.

Suppression après pred

```
void supprime_apres(struct cell* pred) {
  struct cell *c = pred->next;
  pred->next = c->next;
  free(c);
}
```

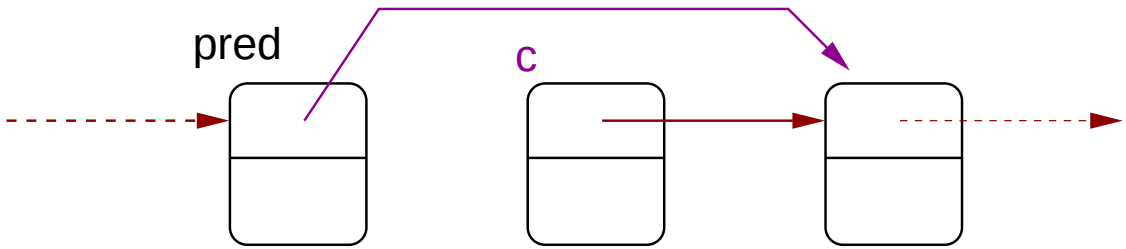


Suppression d'un élément

Suppression facile si on a un pointeur vers la cellule précédente.

Suppression après pred

```
void supprime_apres(struct cell* pred) {
  struct cell *c = pred->next;
  pred->next = c->next;
  free(c);
}
```

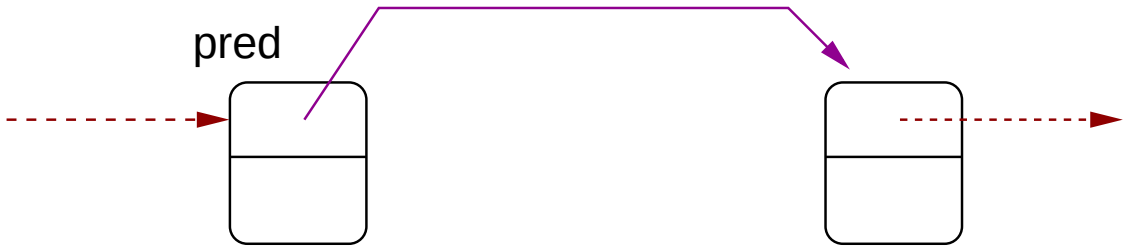


Suppression d'un élément

Suppression facile si on a un pointeur vers la cellule précédente.

Suppression après pred

```
void supprime_apres(struct cell* pred) {
  struct cell *c = pred->next;
  pred->next = c->next;
  free(c);
}
```



Suppression d'un élément

Suppression facile si on a un pointeur vers la cellule précédente.

Suppression après `pred`

```
void supprime_apres(struct cell* pred) {
  struct cell *c = pred->next;
  pred->next = c->next;
  free(c);
}
```

Notes :

- coût constant, hors calcul de `pred`,
- on suppose qu'on ne détruit pas en première position,
  ($\Rightarrow$ on suppose que la liste n'est pas vide)
- on suppose que `pred` a un suivant, (`pred->next`$\neq$`NULL`)
  (on peut par contre avoir `c->next=NULL`).

Suppression d'un élément (version complète)

On propose maintenant une version plus complète de `supprime` de prototype :

Prototype de la suppression

```
struct cell* supprime(struct cell* head, int elem);
```

Effet : supprime le premier élément égal à `elem`,

- calcule automatiquement `pred`,
- gère les cas limites :
  - liste vide, liste à un seul élément,
  - `elem` en tête ou en fin de liste,
  - `elem` non présent dans la liste,
- la tête de liste peut changer,
- coût linéaire au pire, à cause de la recherche de `pred`.

Suppression d'un élément (version complète)

Suppression

```
struct cell* supprime(struct cell* head, int elem)
{
  struct cell* pred = head, *c;

  if (!head) return head;
  if (head->data==elem)
    { c = head->next; free(head); return c; }

  while (pred->next && pred->next->data!=elem)
    pred = pred->next;

  if (pred->next)
    { c = pred->next; pred->next = c->next; free(c); }

  return head;
}
```

Concaténation de listes

Concaténation

```
struct cell* concat(struct cell* head1,
                    struct cell* head2) {
  struct cell*c;
  if (!head1) return head2;
  for (c=head1; c->next; c=c->next);
  c->next = head2;
  return head1;
}
```

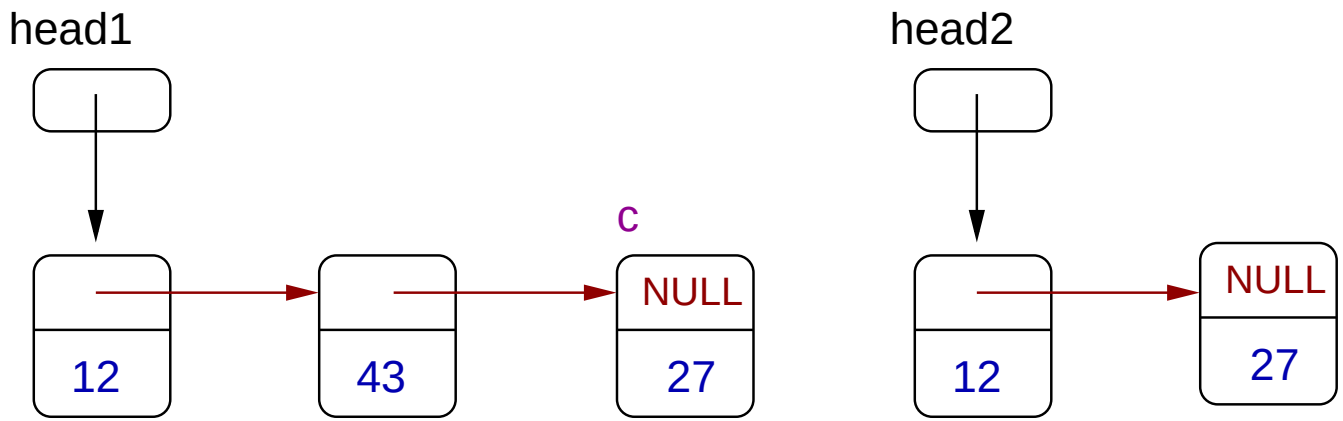
Effet : concatène la 2ème liste au bout de la 1ère.

Utilisation : `head1 = concat(head1, head2);`

Concaténation de listes

Concaténation

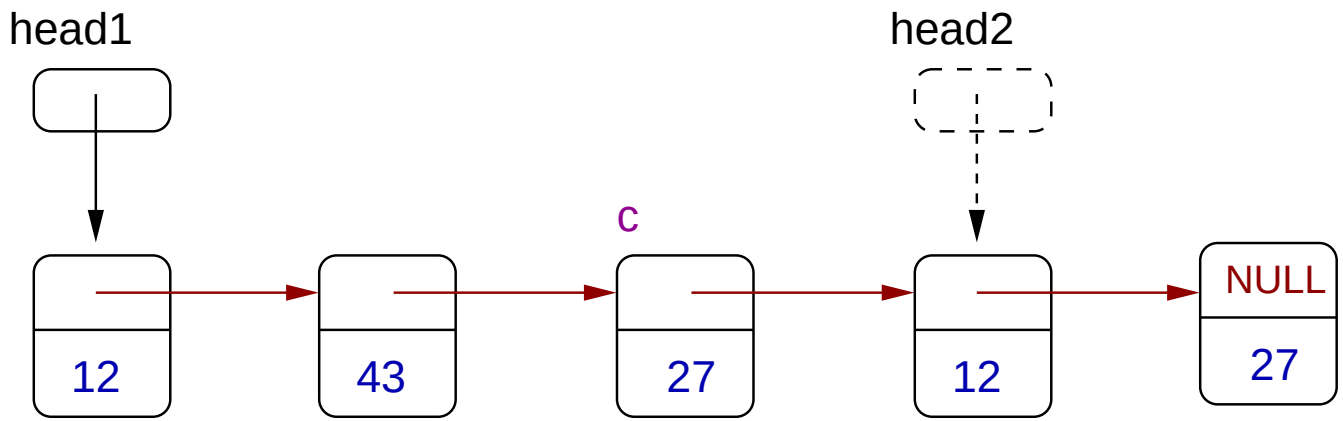
```
struct cell* concat(struct cell* head1,
                    struct cell* head2) {
  struct cell*c;
  if (!head1) return head2;
  for (c=head1; c->next; c=c->next);
  c->next = head2;
  return head1;
}
```



Concaténation de listes

Concaténation

```
struct cell* concat(struct cell* head1,
                    struct cell* head2) {
  struct cell*c;
  if (!head1) return head2;
  for (c=head1; c->next; c=c->next);
  c->next = head2;
  return head1;
}
```



Concaténation de listes

Concaténation

```
struct cell* concat(struct cell* head1,
                    struct cell* head2) {
  struct cell*c;
  if (!head1) return head2;
  for (c=head1; c->next; c=c->next);
  c->next = head2;
  return head1;
}
```

Remarques :

- coût linéaire (parcours complet de la 1ère liste),
- la tête de la première liste peut changer,
- il vaut mieux ne plus accéder à la liste par `head2`...

Destruction totale d'une liste

Destruction

```
void detruit(struct cell* head)
{
  struct cell *c;
  while (head) {
    c = head->next;
    free(head);
    head = c;
  }
}
```

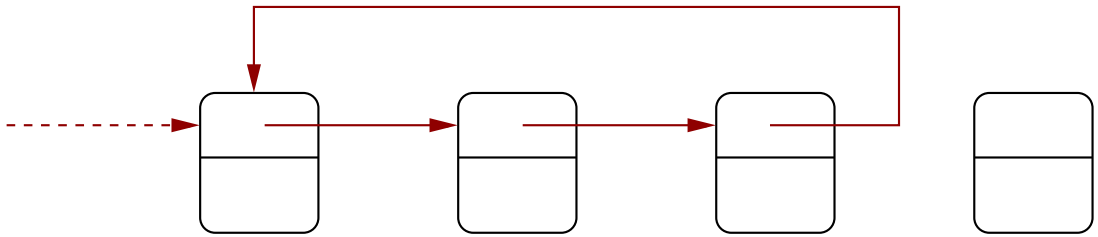
Effet : détruit totalement la liste.

Note : on a besoin d'une variable temporaire c.

Erreurs courantes sur les listes

- Erreurs courantes sur les pointeurs et la mémoire dynamique :
  - déréférencer un pointeur `NULL`,
  - utiliser un bloc après l'avoir libéré,
  - libérer deux fois le même bloc,
  - oublier de libérer un bloc (fuites de mémoire).

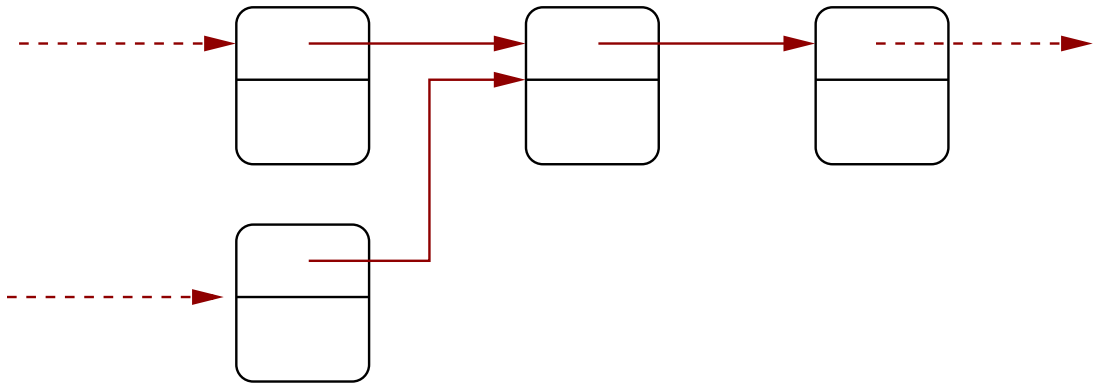
- Introduction de cycles :



(génère des boucles infinies lors des parcours,
cause des fuites de mémoire, . . .)

Erreurs courantes sur les listes

- Partage de cellules entre plusieurs listes :



(effets de bord lors de la modification d'une liste,
cause des libérations multiples de blocs, . . .)

- Oubli des cas limites :
  - liste vide, listes à un élément,
  - insertion/suppression en première/dernière position,
  - etc.

Comparaison entres listes et tableaux

Coût comparé des listes simplement chaînées et des tableaux.

	listes	tableaux
recherche d'une valeur	$\mathcal{O}(n)$	$\mathcal{O}(n)$
accès par indice	$\mathcal{O}(n)$	$\mathcal{O}(1)$
insertion en tête	$\mathcal{O}(1)$	$\mathcal{O}(n)$
insertion en queue	$\mathcal{O}(n)$	$\mathcal{O}(n)$
insertion au milieu	$\mathcal{O}(1)$ / $\mathcal{O}(n)$	$\mathcal{O}(n)$

Les listes sont particulièrement efficaces pour :

- l'insertion et la suppression en tête de liste,
- l'insertion et la suppression en milieu de liste, si la cellule précédente est connue.

Les listes d'association

Liste d'association = associe une valeur à chaque clé.

Cellule

```
struct cell {
    struct cell* next;
    int          key;
    float        val;
};
```

Recherche

```
float recherche(struct cell* head, int key) {
  for (; head; head = head->next)
    if (head->key == key) return head->val;
  return 0.;
}
```

Effet : renvoie la valeur associée à une clé, ou 0.

Listes avec sentinelle

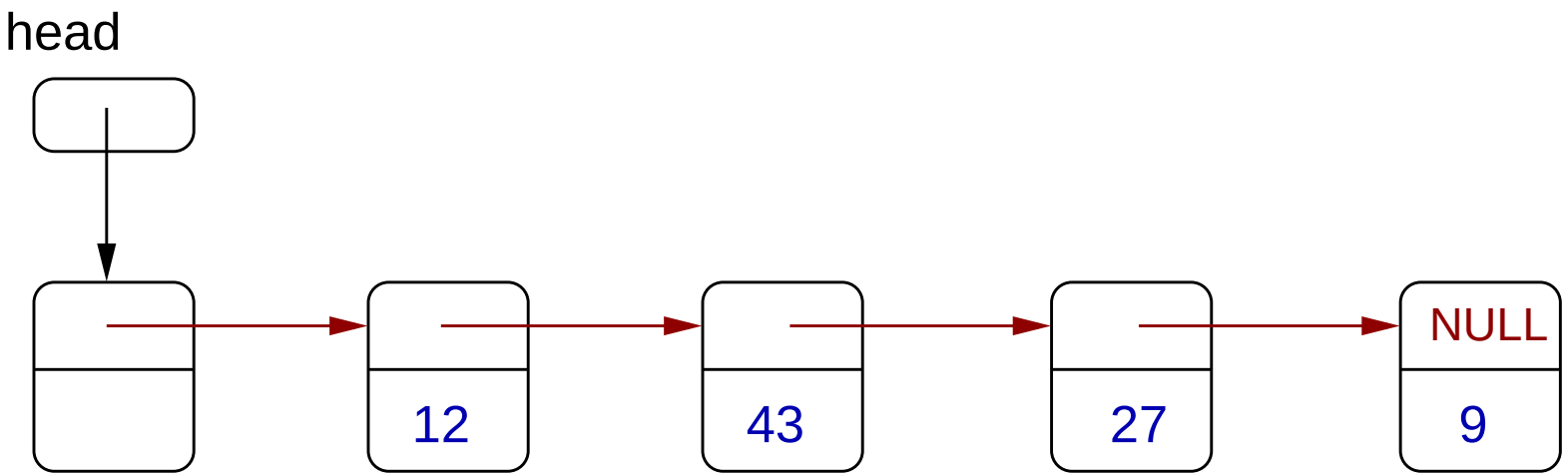
Idée : ajout d'une cellule sentinelle en tête de liste :

- `head->next` pointe sur la première cellule de la liste,
- `head->data` n'est pas utilisé,
- toutes les fonctions prennent en argument un pointeur vers la sentinelle.

Avantage : simplifie la gestion des cas limites :

- cas où la liste est vide,
  (la liste contient toujours au moins une cellule)
- cas où le premier élément de la liste est modifié.
  (`head->next` est modifié, pas de tête de liste à retourner)

Illustration d'une liste avec sentinelle



Opérations sur les listes avec sentinelle

Exemples d'opérateurs

```
struct cell* create()
{
  struct cell* c = malloc(sizeof(struct cell));
  c->next = NULL;
  return c;
}

void insere_tete(struct cell* head, int data)
{
  struct cell* c = malloc(sizeof(struct cell));
  c->data = data;
  c->next = head->next;
  head->next = c;
}
```

Opérations sur les listes avec sentinelle

Exemples d'opérateurs

```
void insere_queue(struct cell* head, int data)
{
  struct cell* c = malloc(sizeof(struct cell));
  c->data = data;
  c->next = NULL;
  while (head->next) head = head->next;
  head->next = c;
}

void concat(struct cell* head1, struct cell* head2)
{
  while (head1->next) head1 = head1->next;
  head1->next = head2->next;
  free(head2);
}
```

Listes doublement chaînées

Listes doublement chaînées

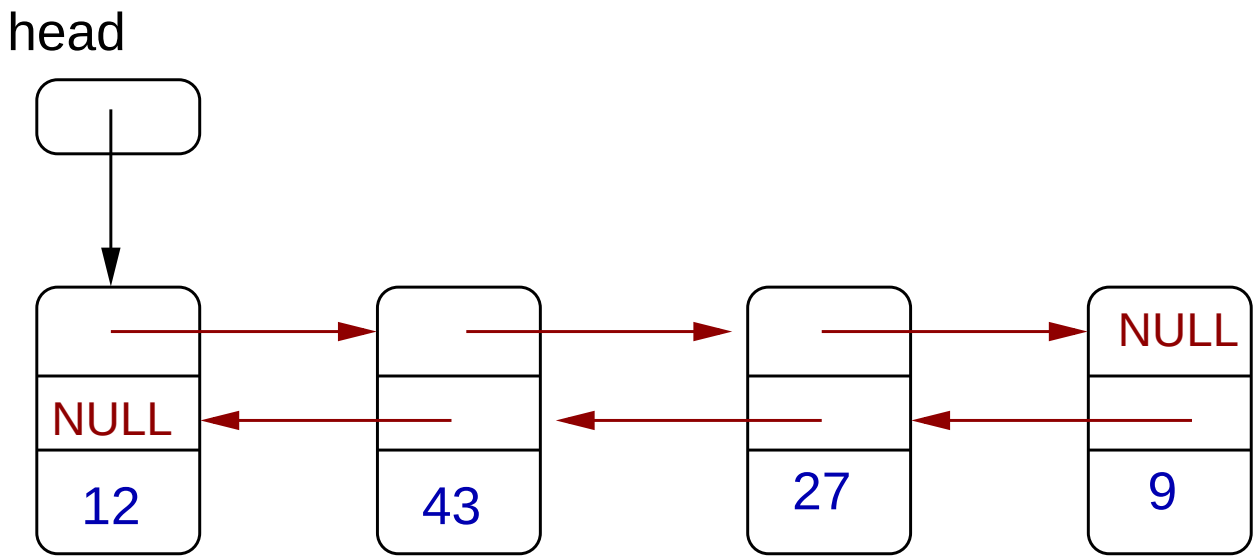
Liste doublement chaînée = on maintient :

- un pointeur vers la cellule suivante,
- un pointeur vers la cellule précédente.

Type cellule

```
struct cell {
    struct cell* next;
    struct cell* prev;
    int          data;
};
```

Illustration d'une liste doublement chaînée



- `head` pointe vers la première cellule,
- `c->next=NULL` pour la dernière cellule, sinon `c->next->prev=c`,
- `c->prev=NULL` pour la première cellule, sinon `c->prev->next=c`.

Insertion dans une liste doublement chaînée

Insertion après `prev`

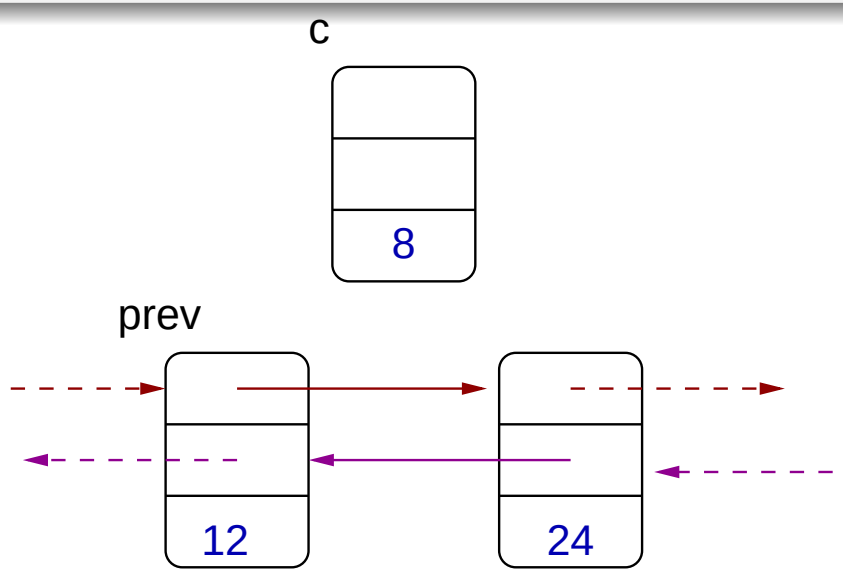
```
void insere_apres(struct cell* prev, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = prev->next; prev->next = c;
  c->next->prev = c;    c->prev = prev;
}
```

Effet : insère une cellule après `prev`.

Insertion dans une liste doublement chaînée

Insertion après `prev`

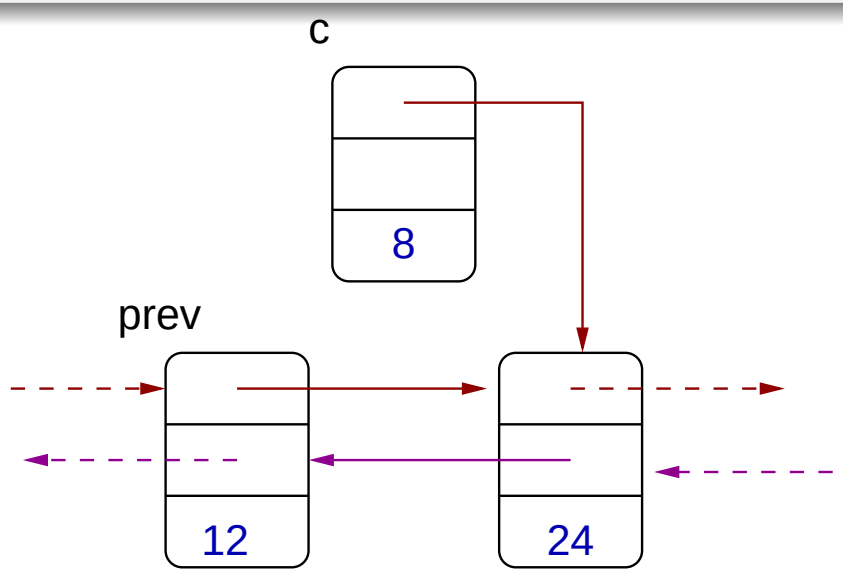
```
void insere_apres(struct cell* prev, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = prev->next; prev->next = c;
  c->next->prev = c;    c->prev = prev;
}
```



Insertion dans une liste doublement chaînée

Insertion après `prev`

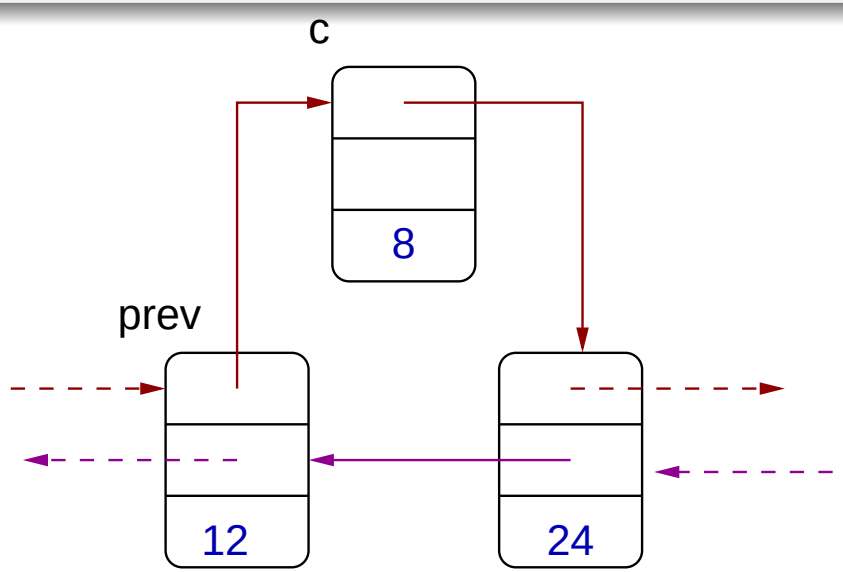
```
void insere_apres(struct cell* prev, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = prev->next; prev->next = c;
  c->next->prev = c;    c->prev = prev;
}
```



Insertion dans une liste doublement chaînée

Insertion après `prev`

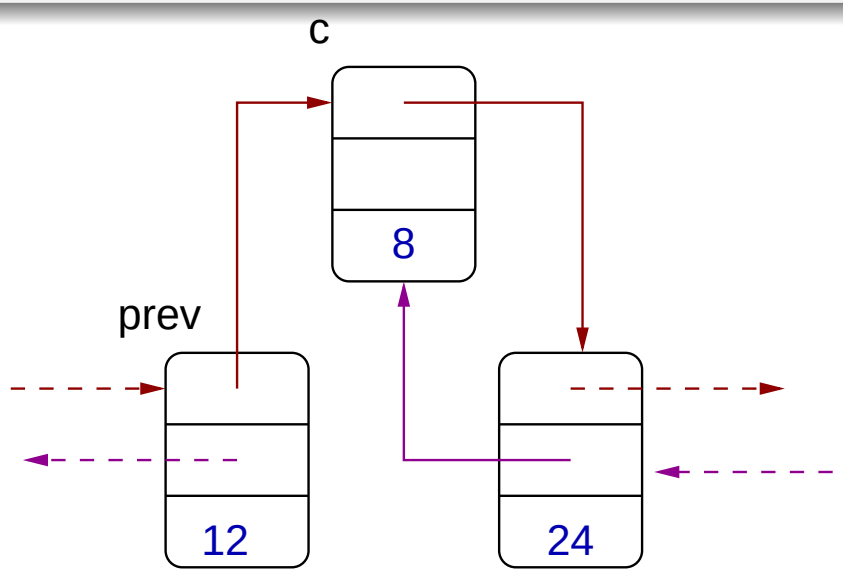
```
void insere_apres(struct cell* prev, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = prev->next; prev->next = c;
  c->next->prev = c;    c->prev = prev;
}
```



Insertion dans une liste doublement chaînée

Insertion après `prev`

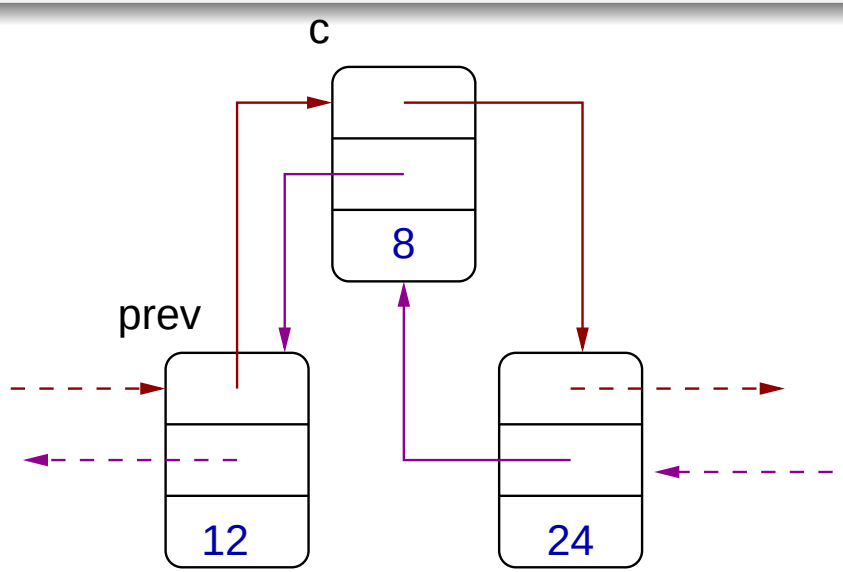
```
void insere_apres(struct cell* prev, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = prev->next; prev->next = c;
  c->next->prev = c;    c->prev = prev;
}
```



Insertion dans une liste doublement chaînée

Insertion après `prev`

```
void insere_apres(struct cell* prev, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = prev->next; prev->next = c;
  c->next->prev = c;    c->prev = prev;
}
```



Insertion dans une liste doublement chaînée

Insertion après `prev`

```
void insere_apres(struct cell* prev, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->next = prev->next; prev->next = c;
  c->next->prev = c;    c->prev = prev;
}
```

Remarques :

- coût constant, sans compter le calcul de `prev`,
- ne marche pas pour insérer en première position,
- pour permettre l'insertion en dernière position :
  changer `c->next->prev=c ;`
  en `if (c->next) c->next->prev = c ;`

Insertion dans une liste doublement chaînée

Insertion avant `next`

```
void insere_avant(struct cell* next, int elem) {
  struct cell *c = malloc(...);
  c->data = elem;
  c->prev = next->prev; next->prev = c;
  c->prev->next = c;    c->next = next;
}
```

Effet : insère une cellule avant `pred`.

Remarques :

- on a simplement inversé les mots `prev` et `next`,
- ne marche pas pour insérer en dernière position,
- facilement modifiable pour permettre l'insertion en première position.

Comparaison des types de listes

Listes doublement chaînées *vs.* listes simplement chaînées :

- on doit maintenir deux pointeurs par cellule au lieu d'un :
  - chaque opération est légèrement plus coûteuse,
  - plus complexe, risques de bug accrus,
- certaines opérations sont plus faciles.
  (*e.g.*, parcours en arrière, insertion avant une cellule)

Schéma classique en informatique : l'ajout de redondance permet des calculs plus rapides, au prix d'une maintenance plus complexe.

Variantes : listes avec sentinelles, listes circulaires, etc.