PC 4
Listes chaînées par pointeurs

- Listes
- Pointeurs
- Listes chaînées
- Piles
- Files

Listes

But : représenter des suites $(a, b, c, \ldots)$ d'éléments d'un ensemble de façon à pouvoir facilement insérer un nouvel élément (ou en supprimer un).

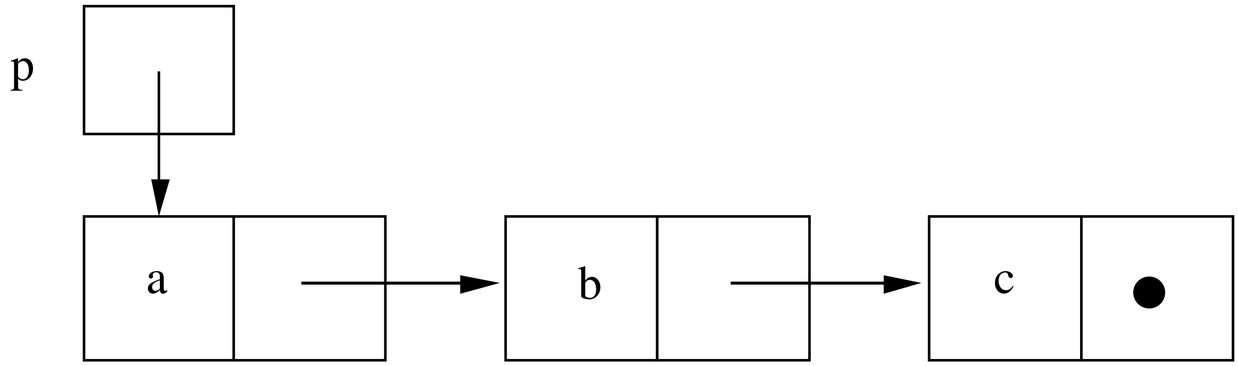

Figure 1:

Exemples d'application

1. Le *hachage ouvert* associe à chaque clé $h(x)$ une liste.

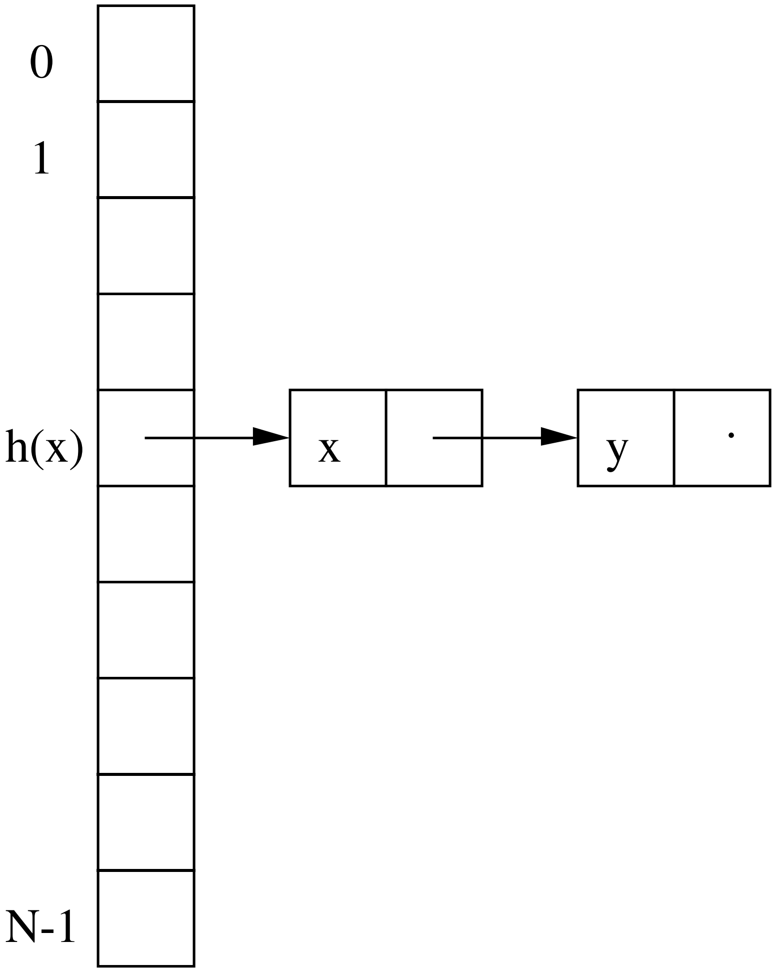

Figure 2: Hachage ouvert

2. L'espace mémoire sur disque est attribué par blocs de taille fixe et chaque fichier est représenté par une liste de blocs.

Pointeurs

Un *pointeur* a pour valeur une *adresse*. En C, un pointeur sur un objet de type T se déclare :

```
T *p;
```

Cet objet est nommé `*p`. Ainsi, le type de p est `T *` et `*p` est de type T.

Inversement, si x est une variable de type T, `&x` est un pointeur sur x.

L'arithmétique des pointeurs permet d'écrire `p+i` quand p est un pointeur et i un entier. La valeur est l'adresse de l'élément de type T situé i places plus loin (c'est en fait $p + i * sizeof(T)$)

Un pointeur qui ne pointe sur rien a pour valeur la constante `NULL`

Allocation mémoire

On crée un objet pointé par `p` par l'instruction

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

On le supprime par `free(p);`. Les informations concernant ces fonctions se trouvent dans `<stdlib.h>`.

Par exemple

```
int * p;
 p=(int *) malloc(sizeof(int));
 *p= 5;
end;
```

est équivalent à

```
int i=5;
```

La différence est que la place de `*p` a été créee de façon *dynamique*.

On peut utiliser

```
#define alloc(t) (t *) malloc(sizeof(t))
```

Pointeurs et tableaux

Si `a` est un tableau, la valeur de a est l'adresse de l'élément `a[0]`. Ainsi `*a` et `a[0]` ont la même valeur. Plus généralement, on a

```
a[i] = *(a+i)
```

De ce fait, on peut écrire une fonction

```
void Lire(int a[]);
```

ou

```
void Lire(int * a);
```

Les *chaînes de caractères* sont des tableaux comme

```
char  mot[10];
```

avec, par convention le caractère `'\0'` à la fin.

Structures

Une structure est un type qui groupe des données de types différents en un seul objet.

Par exemple, on peut grouper un tableau de coefficients et un entier pour faire un type représentant les polynômes.

```
typedef struct {
  float coef[100];
  int degre;
} Poly;
```

Pour initialiser une variable `u` de type `Poly` à la valeur $u = 1 + x^2$, on écrit :

```
u.degre= 2;
u.coef[0]= 1;
u.coef[1]= 0;
u.coef[2]= 1;
```

Si p est un pointeur sur une structure, on écrit `p->champ` comme abréviation de `(*p).champ`.

Listes chaînées par pointeurs

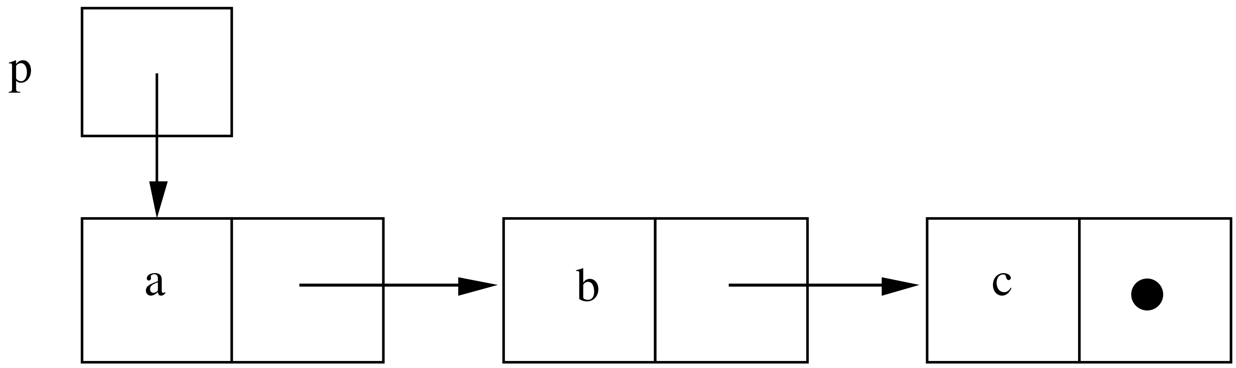

Figure 3:

Déclaration de type:

```
typedef int Element;
typedef struct cellule{
  Element contenu;
  struct cellule * suivant;
} Cellule;
typedef Cellule * Liste;
```

Ainsi une liste est représentée par un pointeur sur une cellule.

Insertion en tête de liste

```
Liste Inserer (Element x, Liste p) {
  Liste q;

  q= alloc(Cellule);
  q->contenu = x;
  q->suivant = p;
  return q;
}
```

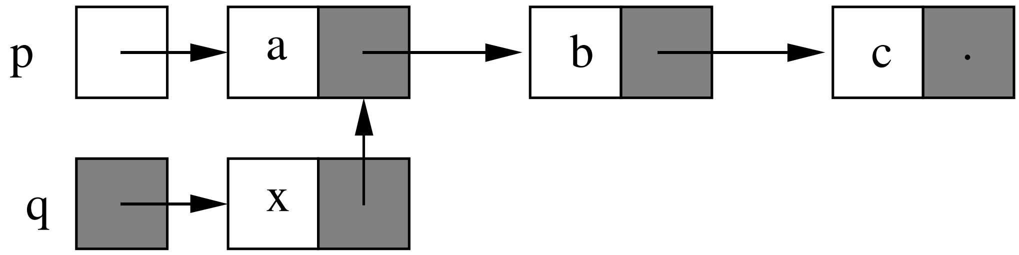

Figure 4:

Recherche d'un élément

```
Liste Recherche(Element x, Liste p) {
  return (p== NULL || p->contenu == x)?
       p : Recherche(x,p->suivant);
}
```

ou itérativement:.

```
Liste Recherche (Element x, Liste p) {
  Liste q;
  q = p;
  while (q != NULL && q->contenu != x)
    q = q->suivant;
  return q;
}
```

Suppression d'un élément

```
Liste Supprimer(Element x, Liste a)
{
if (a== NULL) return a;
if (a->contenu == x)
return a->suivant;
else {
 a->suivant= Supprimer (x, a->suivant);
 return a;
  }
}
```

Variantes

1. Liste avec cellule vide en tête: évite de traiter de façon spéciale l'insertion et la suppression en tête.

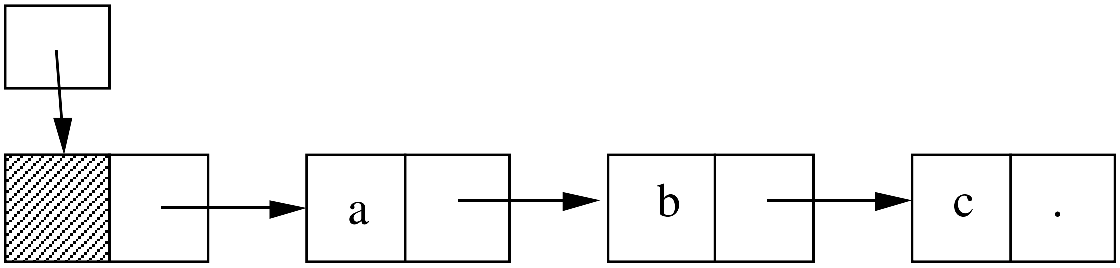

Figure 5:

Insertion en position p :

```
void Inserer(Element x, Liste p) {
  Liste q;
  q= alloc(Cellule);
  q->contenu= x;
  q->suivant= p->suivant;
  p->suivant= q;
}
```

2. Liste doublement chaînée.

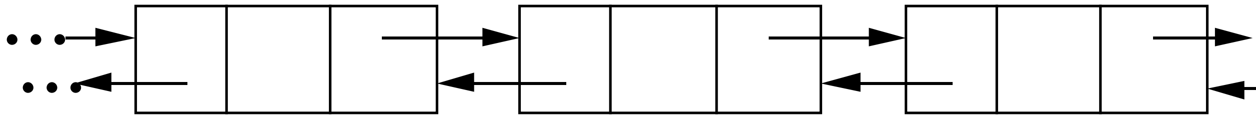

Figure 6:

Possibilité de chaînage circulaire.

Déclaration de type :

```
typedef struct cellule {
  Element contenu;
  struct cellule suivant;
  struct cellule precedent;
} Cellule;
```

Initialisation :

```
void FaireListe(p) {
  p=alloc(Cellule);
  p->suivant= p;
  p->precedent= p;
}
```

Insertion en position p :

```
void Inserer(Element x; Liste p) {
  Liste q= alloc(Cellule);
  q->contenu= x;
  q->suivant= p->suivant;
  q->precedent= p;
  p->suivant->precedent= q;
  p->suivant= q;
}
```

Piles

Liste avec accès restreint utilisée dans de très nombreux algorithmes. On insère et supprime toujours du même côté (lifo). La tête s'appelle traditionnellement le *sommet* de pile. Opérations de base:

- `Empiler(x,S)`: ajoute x en sommet de pile.
- `Depiler(S)`: supprime le sommet de pile.

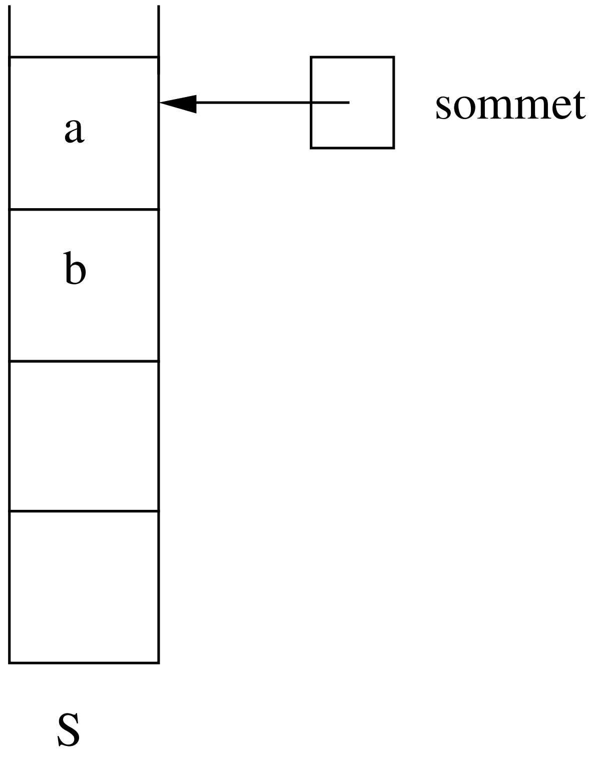

Figure 7:

Implémentation usuelle en tableau.

Files

Cette fois on insère d'un côté (la *queue*) et on supprime toujours de l'autre (la *tête*): stratégie *fifo*.

Implémentation possible: Dans un tableau avec gestion circulaire des indices.

Figure 8:

File vide:

`queue = tete mod N`

File pleine:

`tete = queue + 1 mod N`