NFP136 – VARI 2

Algorithmique et Structures de Données (suite) :

Listes, Files et Piles

Les listes

Qu'est-ce qu'une liste ?

- On peut définir une liste comme une collection ordonnée de données de même type (une certaine alternative au tableau).
- En d'autres termes, une liste non vide est la concaténation (= mise bout à bout) d'un premier élément avec une autre liste.

Rappel : une implémentation des listes est donnée par la classe ArrayList (package *util* de l'API JAVA standard)

Le type abstrait Liste

- **Liste** à laquelle on peut appliquer les 4 opérations suivantes :
  - Tester si elle est vide,
  - La construire par l'ajout d'un élément à une liste existante (éventuellement vide),
  - Accéder à la valeur du premier élément (celui en tête de liste),
  - Accéder à la queue de la liste (c.-à-d. la liste privée de l'élément en tête).

Le type abstrait Liste

- On peut dès lors écrire des algorithmes basés sur ces 4 opérations, comme par exemple le calcul de la taille d'une liste L.

```
Algo : taille
Entrée : une liste L
Sortie : un entier n égal au nombre d'éléments de L
entier n=0;
liste p=L;
Tant que p n'est pas vide faire
  n=n+1;
  p=queue(p); /* queue(p) renvoie la queue de p */
Fait
```

Le type abstrait Liste

- Ou encore tester si un élément x appartient à une liste L

```
Algo : appartient
Entrée : une liste L et une donnée x
Sortie : un booléen b égal à vrai ssi x appartient à L
booléen b=faux;
liste p=L;
Tant que p n'est pas vide ET b est faux faire
  si (valeurTete(p)=x) alors b=vrai; finsi
  p=queue(p);
Fait
```

- Etc…

Implémentations d'une Liste

Plusieurs façons de faire :

- **Listes chaînées** : les différents éléments de la liste sont créés au fur et à mesure des besoins (implémentation dynamique).

- **Listes contigües :** la liste est représentée par un tableau. Ce n'est pas toujours une bonne solution en pratique, mais nous allons le faire ici, dans l'objectif d'illustrer la dissociation entre un type abstrait et son implémentation.

Listes chaînées en JAVA

- Une liste chaînée est une suite finie de cellules, chacune formée :
  - D'une valeur (par exemple un `int` pour une liste d'entiers),
  - De la référence (c'est-à-dire de l'adresse) vers la cellule suivante.

Les valeurs sont donc contenues dans des cellules, ce qui peut parfois être source de confusion !

Classe Liste en JAVA

```
public class Liste {
   private int valeur;
   private Liste suivant;

     public Liste(int premier, Liste reste)
     {valeur = premier; suivant = reste;}

     public int valeurTete() {return(valeur);}

     public Liste queue() {return(suivant);}
} //fin class
```

Une implémentation JAVA d'une liste contiguë

Liste contiguë : principes

- Une liste contiguë est en fait un tableau.
- Dans cette implémentation, la liste est représentée à l'aide de 2 informations :
  - Un tableau dont chaque case contient un élément de la liste,
  - Un indice (entier) qui indique l'emplacement de la tête de la liste (les éléments étant stockés *de droite à gauche*).

Listes contiguës en JAVA

```
class TabListe{
   private int[] tab;
   private int indiceTete;

   //le constructeur public TabListe() {} n'existe pas
   public TabListe(int premier, TabListe reste) {
      if(reste==null) {
         tab=new int[100];
         tab[0]=premier;
         indiceTete=0;
        }
        else {
           tab=reste.tab;
           indiceTete=reste.indiceTete+1;
           tab[indiceTete]=premier;
        }
     }
```

Listes contiguës en JAVA (suite)

```
   public int valeurTete() {
      return (tab[indiceTete]);
   }

   public TabListe queue() {
      if(indiceTete==0) //un seul élément
         return (null);
      TabListe L=new TabListe();
      L.tab=tab;
      L.indiceTete=indiceTete-1;
      return(L);
   }
}//fin class
```

Conclusion sur nos deux implémentations de Liste (1/2)

- Clairement transparent au programmeur comme à l'utilisateur du programme.

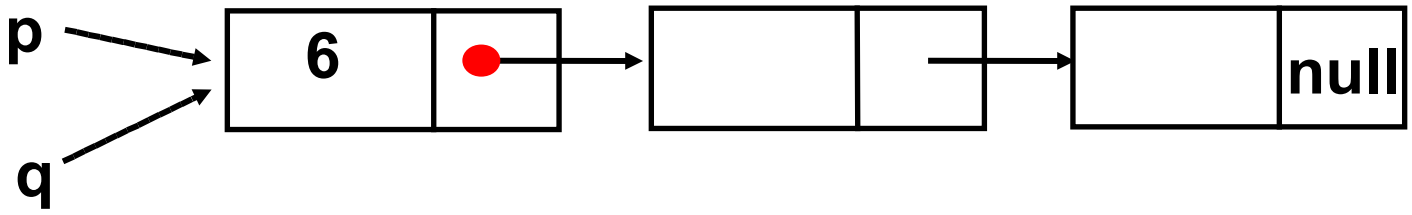
- Mais, dans l'implémentation par un tableau, nous avons « ignoré » des problèmes importants :
  - Initialisation à 100 de la taille du tableau ==> pourquoi pas une autre taille ?
  - Ce choix de taille maximale peut être sous-dimensionné (liste pleine trop vite) ou sur-dimensionné suivant l'usage.
  - Il est bien sûr possible de tenir compte de ces problèmes dans une implémentation plus réaliste et plus fine.

Conclusion sur nos deux implémentations de Liste (2/2)

- L'implémentation de listes par la classe Liste (c'est-à-dire via des **listes chaînées**) est :
  - La plus dynamique,
  - La plus compacte.

C'est celle qu'on privilégie en général !

Quelques mots sur les références et méthodes supplémentaires pour les listes chaînées

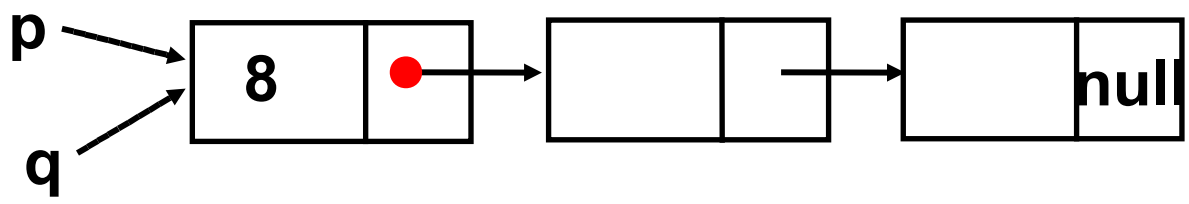


Liste p, q; */* références sur une « cellule» (objet) */*

(…)

p.valeur = 6;

q = p; */*q et p référencent le même objet, et on a donc notamment p.valeur=q.valeur et p.suivant=q.suivant*/*



q.valeur = 8; */* alors p.valeur=8 aussi */*

q = new Liste(p.valeur, p.suivant);

/ alors, q.valeur=p.valeur et q.suivant=p.suivant,*
*mais pas pour les mêmes raisons qu'avant ! */*

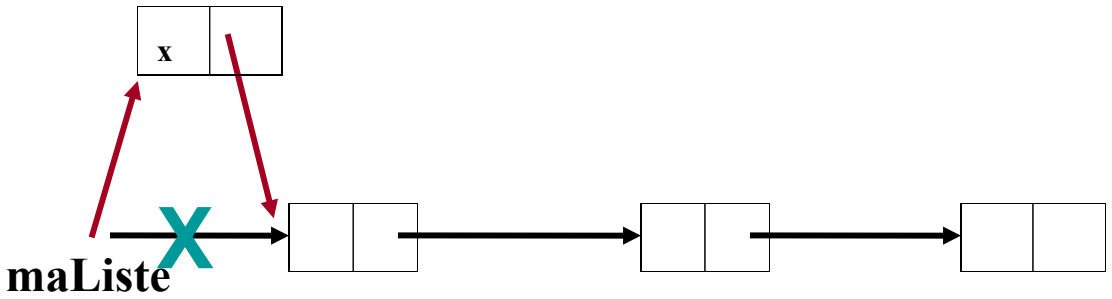
p = null; */* « vide » la liste p */*

LES METHODES

```
public static Liste
  insererEnTete(int x, Liste L){
  Liste NL = new Liste(x, L);
  return NL;
}
```

Utilisation dans un programme :

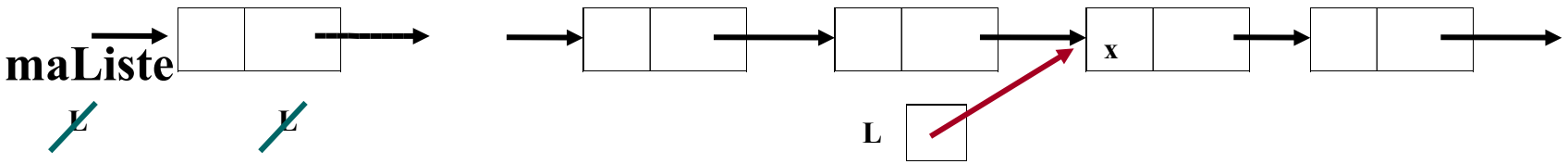
```
maListe = insererEnTete(x, maListe)
```



```
public static boolean appartient(int x, Liste L) {
  while(L != null){
      if(L.valeur == x) return true;
      L=L.suivant;
  }
  return false; /* l'élément n'a pas été trouvé */
}
```

Utilisation dans un programme :

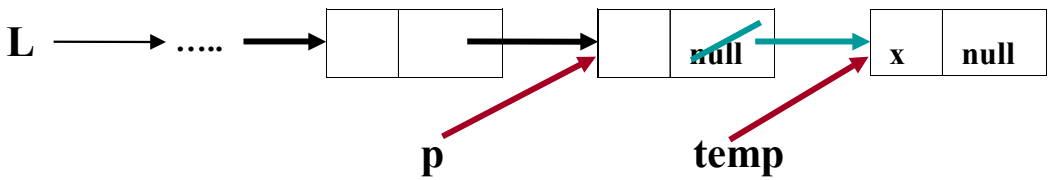
```
boolean present = appartient(x, maListe)
```



LES METHODES

```
public static Liste insererEnQueue(int x, Liste L) {
 /* insère x en queue de la liste L */
 Liste temp = new Liste(x, null);
 if(L == null)
   return temp;
 //si L n'est pas nulle, on va à la fin de la liste
 Liste p = L;
 while(p.suivant != null)
   p = p.suivant;
 p.suivant = temp;
 return L;
}
```

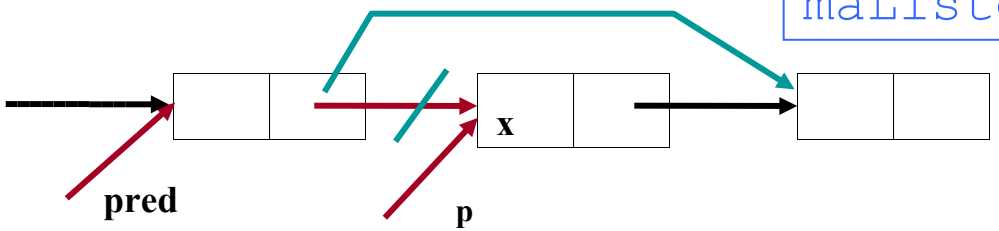
```
maListe = insererEnQueue(x, maListe)
```



LES METHODES

```
public static Liste supprimer(int x, Liste L) {
    Liste p = L;
    Liste pred = null; /* pred = prédécesseur de p */
    while(p != null && p.valeur != x) {
        pred = p;
        p = p.suivant;
    }
    if(p != null) { /* x a été trouvé */
        if (pred == null) /* x = premier élément de la liste */
            return (L.suivant);
        else {
            pred.suivant = p.suivant;
            return L;
        }
    }
    return L;
}
```

```
maListe = supprimer(x, maListe)
```



Conclusion sur les listes chaînées

- C’est une structure de données très souple car on peut ajouter ou supprimer à n’importe quel endroit de la liste (et en temps constant, une fois le bon emplacement trouvé).

- C'est une alternative à la structure de tableau, MAIS sans accès direct au ième élément.

- Voyons à présent d'autres structures simples, mais plus restrictives : les piles et les files.

Les Piles

Pile - Définition

- Une pile est une structure de données pouvant contenir un ensemble d'éléments de même type, et où les insertions et suppressions d'éléments se font toujours du même côté (*en haut*) :
  - Insérer un élément=*empiler*
  - Supprimer un élément=*dépiler*
  - Structure LIFO (Last-in, First-out)

- Autres opérations de base : on doit pouvoir tester si la pile est vide, et également accéder à la valeur de l'élément au sommet de la pile.

Une implémentation des piles en JAVA

- Dans l'implémentation proposée ici, on va s’appuyer sur la classe JAVA Liste.

- Autre(s) implémentation(s) possible(s) ? Par exemple, à l'aide de tableaux (cf TP).

Classe Pile en java

```
class Pile {
    private Liste L = null;

   public boolean estVide(){
          return (L == null);
   }
   public void empiler(int x) {
          L = new Liste(x, L);
   }
   public int depiler() { //suppose que L != null
          int t = L.valeurTete();
          L = L.queue();
          return t;
   }
   public int sommet(){
          return (L.valeurTete());
   }
}
```

Pile – utilisations en informatique

- 3 applications (parmi d'autres...) :
  - Appels de fonctions dans un programme.
  - Liste de modifications d'un fichier dans un éditeur de texte (pour annulation, etc.).
  - Vérification du bon parenthésage d'une chaîne de caractères (cf TP) :
    - Empiler 0 si « (»
    - Dépiler si «) »
    - Si la pile est vide à la fin : ok !

Utilisation pour la vérification des parenthèses (méthode testPile)

Exemples d'utilisation :

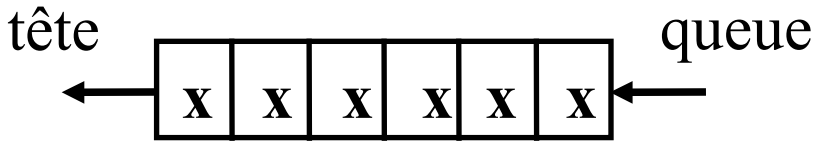
```
java testPile a ( b ) ( ( c ) )
Expression bien parenthésée

java testPile a b cc ( ( (
Expression mal parenthésée (pile non vide)

java testPile a ( ) f )
Expression mal parenthésée (exception)
```

Les Files

File - Définition

- Une file est une structure de données pouvant contenir un ensemble d'éléments de même type, et où les insertions se font d'un côté (queue) et les suppressions se font de l'autre côté (tête) :
  - Insérer un élément=*enfiler*
  - Supprimer un élément=*défiler*
  - Structure FIFO (First-in, First-out)

File - Illustration



- Par exemple :
  - File d'attente à un guichet,
  - *Buffer* (tampon) de messages,
  - etc.

méthodes	conditions
enfiler(entier x)	file non pleine insertion en queue
defiler( )	file non vide retrait en tête
estVide( )	retourne booléen
estPleine( )	retourne booléen

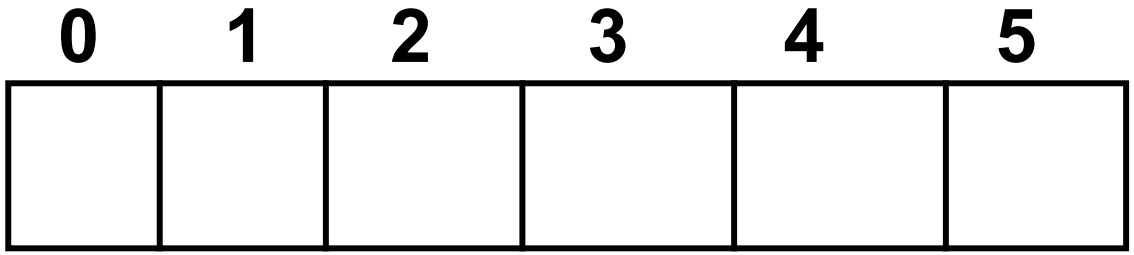
Implémentation des files en JAVA

- A l'aide de listes chaînées (classe Liste).

- A l'aide d'un tableau (cf TP) :
  - Un indice (int) pour garder trace de la queue.
  - Nombre limité d'éléments dans la file !
  - Ne pas oublier le décalage de tous les éléments du tableau lors du défilage (suppression d'un élément), pour que le premier élément soit toujours stocké à l'indice 0 !

EXEMPLE

File file = new File(6)



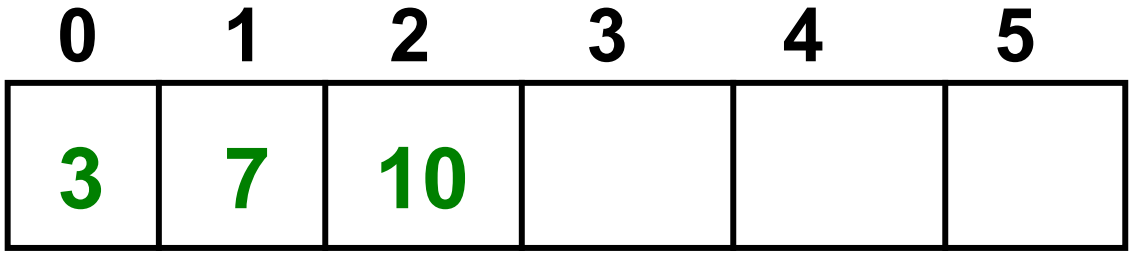
file.queue vaut -1

file.estVide() est *vrai*

file.enfiler(3)

file.enfiler(7)

file.enfiler(10)



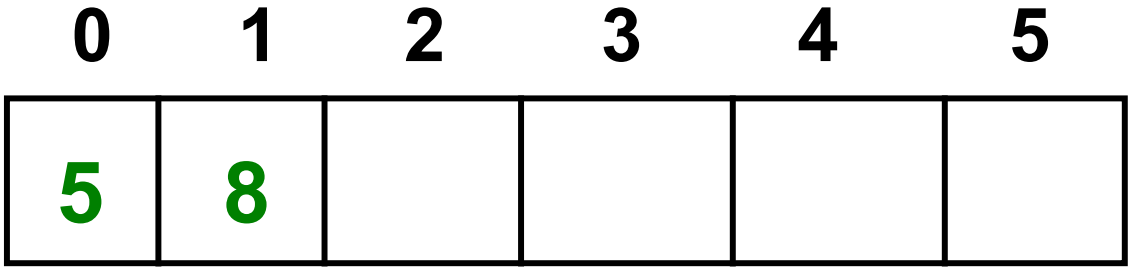
file.queue = ~~0~~

~~1~~

2

file.estVide() est *faux*

EXEMPLE



file.queue = ~~2~~ ~~1~~ ~~0~~ ~~1~~ ~~0~~ ~~1~~ ~~0~~ 1

x=file.defiler()
//décalage
x=file.defiler()

file.enfiler(2)

x=file.defiler()

file.enfiler(5)

x=file.defiler()

file.enfiler(8)

Conclusion : Listes, Files et Piles

- Après les tableaux, nous avons vu une structure de données « linéaire » : les listes
- Deux autres structures linéaires (notions de successeur/prédécesseur) : piles et files
- Nous avons illustré la notion de type abstrait :
  - A travers l'implémentation contiguë (via des tableaux) ou chaînée d'une liste,
  - En « cachant » comment est vraiment représentée notre structure de données (*encapsulation*).

Tables de hachage

Tables de hachage - aperçu

- Accès aux éléments via des identifiants (clés).
- Structure illustrant l'utilisation conjointe de tableaux et de listes chaînées (entre autres).
- Utiles pour le stockage de (très) grandes quantités d'informations :
  - en garantissant un accès « rapide » aux données,
  - en limitant l'espace mémoire occupé.

Tables de hachage - principe

- On dispose d'un tableau de taille m fixée.
- Nombre de cases potentiellement insuffisant pour stocker toutes les données.
- Idée simple : définir une fonction qui associe à chaque donnée un identifiant, et si besoin stocker plusieurs données dans la même case, indicée par cet identifiant.
- On appelle *fonction de hachage* cette fonction.

Exemple de fonction de hachage

Dans un annuaire téléphonique, on veut pouvoir retrouver l'information **nom, prénom** à partir de la clé **numéro de téléphone**.

Fonction de hachage (***on ne dit pas ici comment on la calcule***) :

h(0381111144)=0 ; h(0381333333)=2 ;
h(0381222222)=3 ; h(0381123456)=6 ;

Avantage : accès direct dès lors qu'on a calculé h(c) (= valeur de hachage de c), où c est la *clé*.

indice	information
0	Pierre Durand 0381111144
1	
2	Paul Dupont 0381333333
3	Yvette Bon 0381222222
4	
5	
6	Gilles Dupont 038123456

Table de hachage

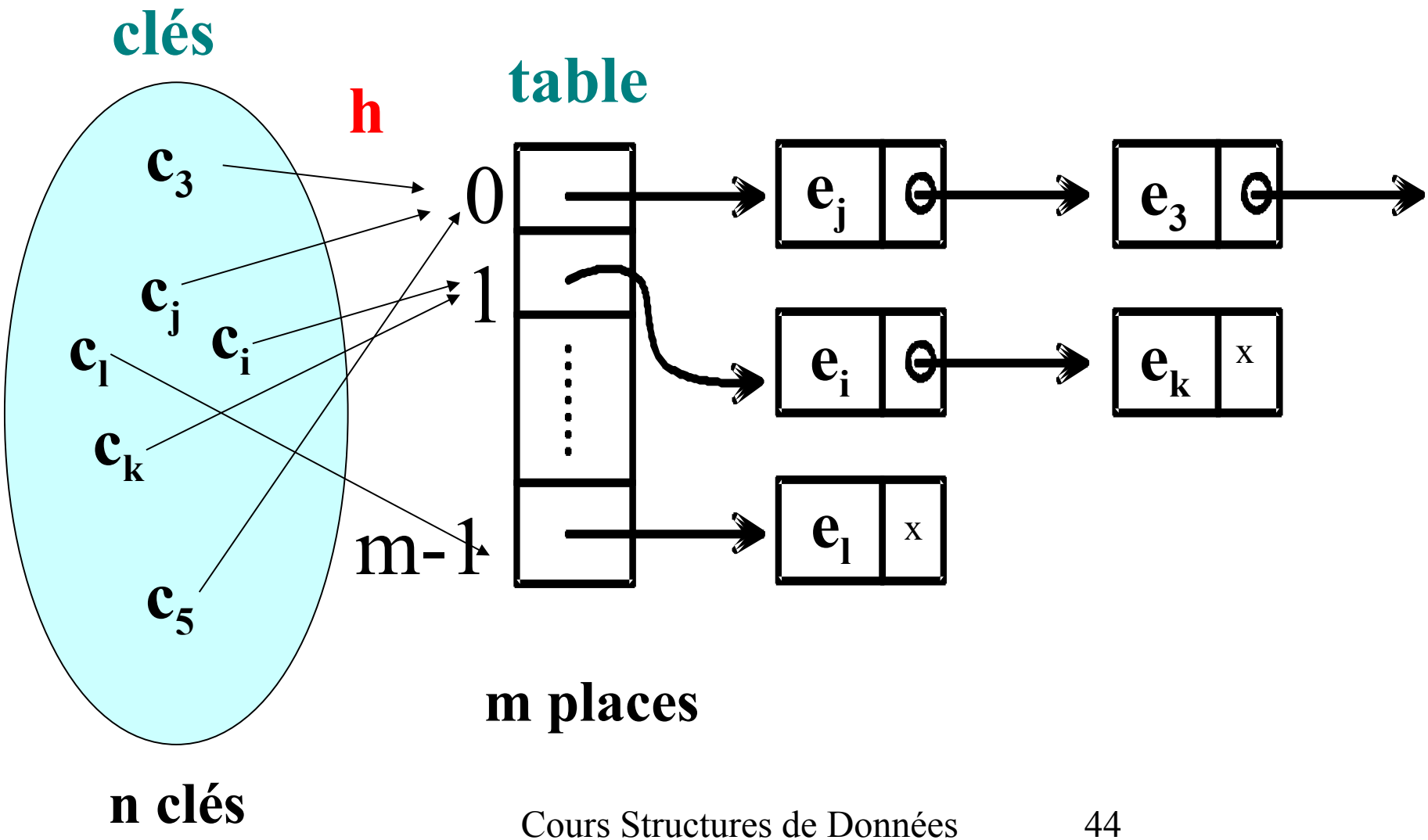
Collisions

- Dans le cas le plus simple, au plus m données à stocker ==> une clé = une case (h *injective*).
- En général, on a plus que m données à stocker
- Dans ce cas, on peut avoir deux clés c_1 et c_2 avec $h(c_1)=h(c_2)$ (c_1 et c_2 ont des *valeurs de hachage* identiques).

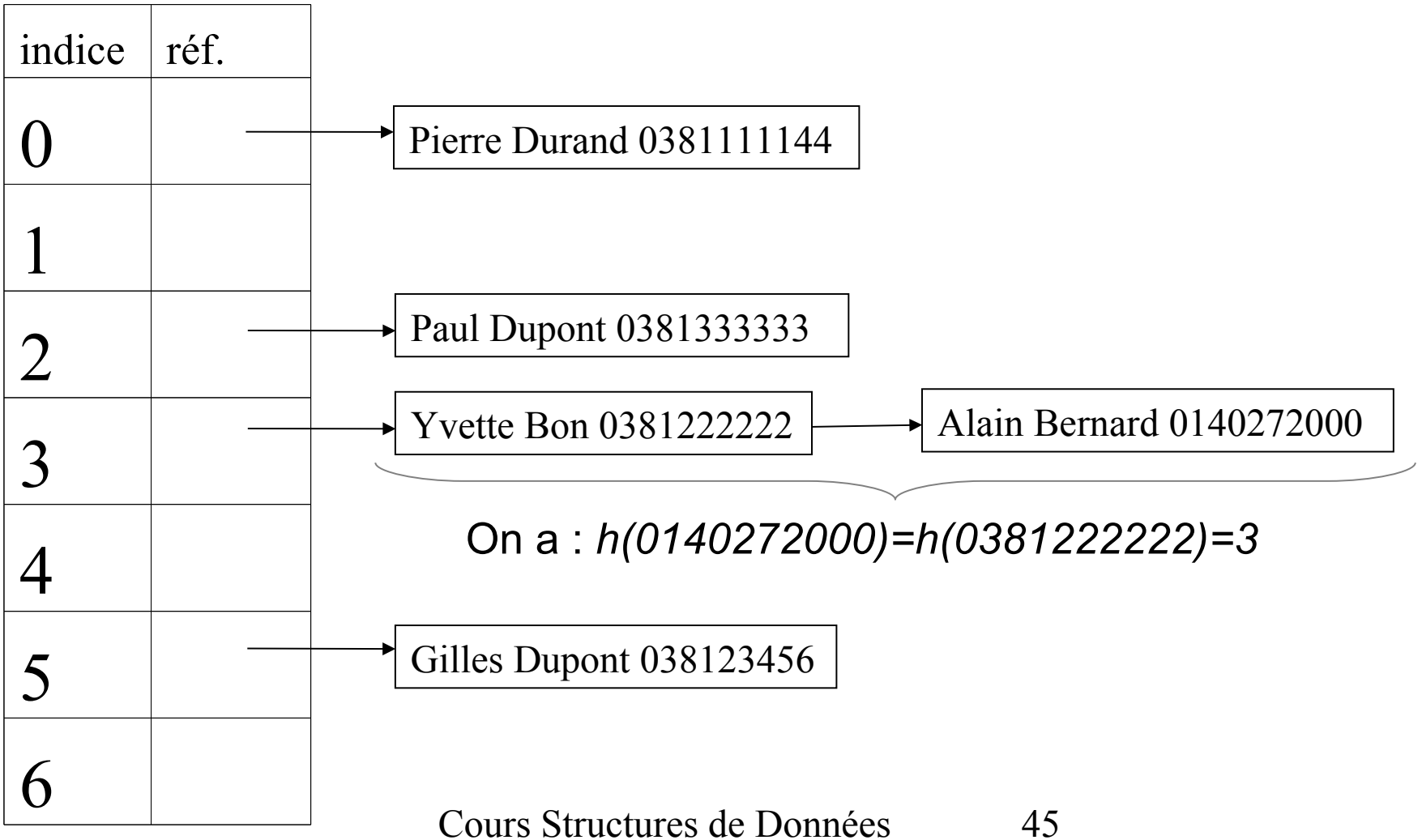
Gestion des collisions

- Principe (méthode de chaînage) :
  - Chaque case du tableau référence en fait sa propre liste chaînée,
  - Les éléments dont les clefs ont la même valeur de hachage sont mis dans la même liste !

Principe du chaînage



Exemple


indice
réf.
0
1
2
3
4
5
6
Pierre Durand 0381111144
Paul Dupont 0381333333
Yvette Bon 0381222222
Alain Bernard 0140272000
On a : h(0140272000)=h(0381222222)=3
Gilles Dupont 038123456

==> Rechercher l'élément de clé c

=

Calculer l'indice h(c) dans T (en O(1))

+ Parcourir la liste chaînée (en $O(L_{h(c)})$)

Idéal : éviter les collisions

Réaliste : limiter les collisions

==> Choisir une « bonne » fonction h !

==> Une fonction de hachage associe un entier à un objet (une chaîne, un entier, ou un objet quelconque) : quelles propriétés doit-elle posséder ?

4 règles d'or (peu réalistes à garantir simultanément) :

1. La valeur de hachage est calculée à partir (d'une partie) de l'objet d'entrée (et uniquement de lui).
2. La fonction de hachage distribue uniformément les données (ou autant que possible).
3. La fonction de hachage génère des valeurs (très) différentes pour des objets quasi identiques.
4. La fonction de hachage doit être rapide à calculer.

Fonctions de hachage pour les entiers

- h(c) le reste de la division de c par m : h(c) = c mod m

(Note : une bonne valeur pour m en pratique est un nombre premier, pas trop proche d'une puissance de 2 ; peut se comporter mal quand même.)

- Remarques :
  - Si les entiers en entrée sont distribués assez uniformément, la Règle 2 est à peu près respectée,
  - Règle 3 respectée « à l'économie ».

- Variante : h(c) = c(c + 3) mod m (cf D. Knuth).

Deux dernières remarques sur l'implémentation des tables de hachage

- Il existe d'autres mécanismes plus élaborés, non détaillés ici, pour gérer les collisions (par exemple : adressage ouvert et sondage).
- Il existe une implémentation des tables de hachage dans l'API JAVA standard : il s'agit de la classe Hashtable (dans le package *util*).

Tables de hachage : bilan

- Avantage : identifiants associés aux données
- Parfaite illustration de la complémentarité entre tableaux et listes chaînées via l'implémentation
  - Tableaux=adressage direct (accès rapide).
  - Listes chaînées=espace mémoire optimisé.
- Une « bonne » fonction de hachage est parfois difficile à définir, et peut dépendre fortement de la nature des données à stocker.