STRUCTURES DE DONNÉES

Chargée de cours: Lélia Blin
Transparents:http://www-npa.lip6.fr/~blin/Enseignements.html
Email: lelia.blin@lip6.fr

EQUATION DE BASE

Programme = Structures de données + algorithmes

Niklaus Wirth (concepteur du langage Pascal)

STRUCTURES DE DONNÉES

- But:
  - Stoker des données auxquelles le programme peut accéder
- Opérations élémentaires:
  - lecture
  - insertion
  - suppression

STRUCTURES DE DONNÉES

- Toutes les structures de données ne sont pas équivalentes:
  - Différences dans les opérations élémentaires.
  - Pas le même «coût».

STRUCTURES DE DONNÉES

- Structures élémentaires:
  - Tableaux.
  - Listes chaînées
- Autres structures :
  - Files.
  - Piles.
  - Tables de Hachages.
  - Arbres.

STRUCTURES DE DONNÉES

- Structures élémentaires:
  - Tableaux.
  - Listes chaînées
- Autres structures :
  - Files.
  - Piles.
  - Tables de Hachages.
  - Arbres.

STRUCTURES DE DONNÉES ÉLÉMENTAIRES

Lélia Blin

Université d'Evry

STRUCTURES LINÉAIRES

- Eléments d'un même type stockés dans :
  - Des tableaux
  - Des listes chaînées
- Deux cas possibles :
  - Éléments triés (ordre doit être maintenu)
  - L'ordre n'a aucune importance.

OPÉRATIONS SUR LES STRUCTURES LINÉAIRES

- Insérer un nouvel élément
- Supprimer un élément
- Rechercher un élément
- Affichage des éléments
- Concaténation de deux ensembles

TYPES DE STRUCTURES TABLEAU

```
Enregistrement Tab {
        T[NMAX] : entier;
        Fin : entier;
}
```

TYPES DE STRUCTURES
LISTE CHAINEE

```
Enregistrement Elément {
    Val : entier;
    Suivant : ↑Elément;
}
```

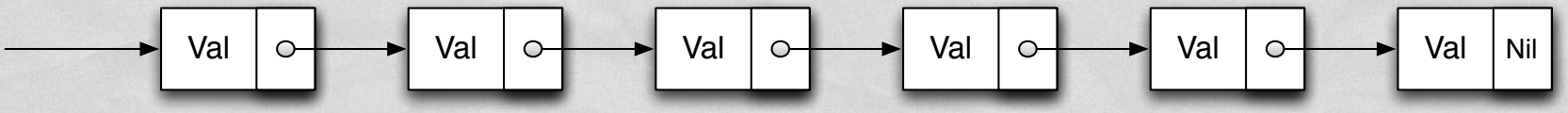


TABLEAU NON TRIÉ : RECHERCHE

```
Recherche(T: Tableau, elt : entier) : booléen
Début
    i : entier;
    pour i de 0 à fin faire
        si (T[i] == elt)
            retourner vrai;
        fin si
    fin pour
    retourner faux;
Fin
```

TABLEAU NON TRIÉ : RECHERCHE (COMPLEXITÉ)

- **Opération fondamentale :** comparaison
- A chaque itération :
  - 1 comparaison (si … fin si)
  - 1 comparaison (pour … fin pour)
- **Nombre d'itérations maximum** :
  - Nombre d'éléments du tableau.
- Complexité : si n est le nombre d'éléments du tableau, O(n).

TABLEAU NON TRIÉ : INSERTION

```
Insertion(T: Tableau, elt: entier)

Début

    fin ← fin + 1;

    T[fin] ← elt;

Fin
```

TABLEAU NON TRIÉ : INSERTION (COMPLEXITÉ)

- Opération fondamentale : affectation
- Nombre d'opérations fondamentales :
  - 2 affectations
- Complexité : O(1) (temps constant)

LISTE CHAÎNÉE NON TRIÉE : RECHERCHE

```
Recherche(L : ↑ Elément, elt : entier) : booléen
Début
  Si (L == NIL)
   retourner faux;
  sinon si (L.val == elt)
   retourner vrai;
  sinon
   retourner Recherche(L.suivant,elt);
  fin si
Fin
```

LISTE CHAÎNÉE NON TRIÉE : RECHERCHE

```
Recherche(L : ↑ Elément, elt : entier) : booléen
Entrées : elt (élément recherché), L (tête de liste)
Sortie : vrai si l'élément elt a été trouvé dans la liste L, faux sinon
Début
  p : ↑ Elément;
  p ← L;
  tant que (p <> NIL) faire
    si (p.val <> elt)
       p ← p.suivant;
    sinon
       retourner vrai;
      fin si
  fin tant que
  retourner faux;
Fin
```

LISTE CHAÎNÉE NON TRIÉE : RECHERCHE (COMPLEXITÉ)

- Opération fondamentale : comparaison
- A chaque itération :
  - Une comparaison (si … fin si)
  - Une comparaison (tant que … fin tant que)
- Nombre d'itérations :
  - Au pire : nombre d'éléments de la liste
- Complexité : si n est le nombre d'éléments de la liste : O(n).

LISTE CHAÎNÉE NON TRIÉE : INSERTION

- Insertion d'un nouvel élément :
  - En tête de liste
  - Au milieu de la liste
  - En fin de liste

LISTE CHAÎNÉE NON TRIÉE : INSERTION

```
Insertion (L : ↑Elément, elt : entier)
Entrée : une liste L et l'élément elt à insérer
Sortie : la liste L avec le nouvel élément.
Début
   p : ↑Elément;

   p.val ← elt;
   p.suivant ← L;
  L ← p;
Fin
```

LISTE CHAÎNÉE NON TRIÉE : INSERTION (COMPLEXITÉ)

- Opération fondamentale : affectation
- Nombre d'opérations fondamentales :
  - 3 affectations.
- Complexité : O(1)

TABLEAU TRIÉ : INSERTION

```
Insertion(S : Tab, elt : entier)

Entrée : le tableau S et un élément elt à
  insérer

Sortie : le tableau S dans lequel elt à été
  inséré

Pré-condition : le tableau S trié par ordre
  croissant.
```

TABLEAU TRIÉ : INSERTION

```
Insertion(S : Tab, x : entier)
Début
   i, k : entier;

   Si (S.fin = -1)
      S.fin ← 0;
      S.T[S.fin] ← x;
   sinon
      i ← 0;
      tant que (i<S.fin et S.T[i] < x)
          i ← i + 1;
      fin tant que
      si (i=S.fin et S.T[i]<x)
          k ← S.fin + 1;
      sinon
          k ← i;
      fin si
      pour i de S.fin + 1 à k+1 en décroissant faire
          S.T[i] ← S.T[i-1];
       fin pour
      S.T[k] ← x;
      S.fin ← S.fin + 1;
   fin si
Fin
```

TABLEAU TRIÉ : INSERTION (COMPLEXITÉ)

- Opération fondamentale : affectation
- Recherche de la bonne position : k comparaisons et affectations
- Décaler à droite : n-k affectations
- Insérer elt : 1 affectation
- Incrémentation de fin : 1 affectation
- Total : n+2 affectations
- Complexité : O(n) si n est le nombre d'éléments du tableau.

TABLEAU TRIÉ : RECHERCHE

- Première idée :
  - On compare l'élément recherché à tous les éléments du tableau comme on l'a fait pour un tableau non trié
  - Pb : on ne tient pas compte de l'ordre des éléments.
- Deuxième idée :
  - Recherche dichotomique
  - Utilisation du fait que les éléments sont triés

TABLEAU TRIÉ : RECHERCHE

- Soit M l'élément du milieu du tableau
  - Si elt = M on a trouvé
  - Si elt < M, elt est dans la première moitié du tableau.
  - Si elt > M, elt est dans la seconde moitié du tableau.
- Fonction Récursive.

TABLEAU TRIÉ : RECHERCHE

Recherche(elt : entier, S : Tab, g : entier, d :entier) : booléen

Entrées : **elt** *(élément recherché),* **S** *(espace de recherche),* **g** *(indice de gauche),* **d** *(indice de droite)*

Sortie : **vrai** *si l'élément est dans S,* **faux** *sinon*

Pré-conditions : **g** *et* **d** *sont des indices valides du tableau* **S et S est** *trié par ordre croissant*

TABLEAU TRIÉ : RECHERCHE

```
Recherche(elt : entier, S : Tab, g : entier, d :entier) : booléen
Début
   m : entier

   si (g<=d)
     m ← (g+d)/2
     si (elt = S.T[m])
          retourner vrai;
     sinon si (elt < S.T[m])
          retourner(Recherche(elt,S,g,m-1));
        sinon
          retourner(Recherche(elt,S,m+1,d));
     fin si
   sinon
     retourner faux;
   fin si
fin
```

TABLEAU TRIÉ : RECHERCHE (COMPLEXITÉ)

- Opération fondamentale : comparaison
- A chaque appel récursif, on diminue l'espace de recherche par 2.
  - Au pire on fera donc $O(\log_2 n)$ appels

LISTE CHAÎNÉE : RECHECHE

- Améliorations par rapport à une liste non triée :
  - Très peu …
  - On peut s'arrêter plus rapidement dans la recherche.

LISTE CHAÎNÉE TRIÉE : RECHERCHE

```
Recherche(L : ↑ Elément, elt : entier) : booléen
Entrées : elt (élément recherché), L (tête de liste)
Sortie : vrai si l'élément elt a été trouvé dans la liste L, faux sinon
Pré condition : la liste est triée par ordre croissant
Début
  p : ↑ Elément;

   p ← L;
  tant que (p <> NIL) faire
    si (p.val < elt)
       p ← p.suivant;
    sinon si (p.val = elt)
       retourner vrai;
    sinon
       retourner faux;
     fin si
  fin tant que
Fin
```

LISTE CHAÎNÉE TRIÉE : RECHERCHE (COMPLEXITÉ)

- Opération fondamentale : comparaison
- A chaque itération :
  - Une comparaison (si … fin si)
  - Une comparaison (tant que … fin tant que)
- Nombre d'itérations :
  - Au pire : nombre d'éléments de la liste
- Complexité : si n est le nbe d'éléments de la liste : O(n).

LISTE CHAÎNÉE : INSERTION

```
Inserer(L : ↑Elément, elt : entier)
Entrée : une liste L et l'élément elt à insérer
Sortie : la liste L avec le nouvel élément.
Début
  p : ↑Elément;

  si (L = NIL ou L.val >= elt)
      p.val ← elt;
      p.suivant ← L;
    l ← p;
  sinon
    L.suivant ← Inserer(L.suivant,elt);
  fin si
Fin
```

LISTE CHAÎNÉE : INSERTION (COMPLEXITÉ)

- Opération fondamentale : comparaison
- A chaque itération :
  - 2 comparaisons
- Nombre d'itérations :
  - Au pire il faut parcourir tous les éléments de la liste.
- Complexité : O(n) si n est le nombre d'éléments de la liste.

RÉSUMÉ : INSERTION (COMPLEXITÉ)

	Eléments triés	Eléments non triés
Tableaux	O(n)	O(1)
Liste Chaînée	O(n)	O(1)

RÉSUMÉ : RECHERCHE (COMPLEXITÉ)

	Eléments triés	Eléments non triés
Tableaux	O(log	O(n)
Liste Chaînée	O(n)	O(n)

PILES

Lélia Blin

Université d'Evry

UNE PILE

- **Analogie de la pile d'assiettes**
  - Le dernier arrivé est le premier sortie.
  - Last In First Out (LIFO)
- **Opérations principales:**
  - Insérer un élément dans une pile
  - Supprimer un élément dans une pile
    - le sommet de la pile (le plus récent)
  - Elément du sommet de la pile
  - Création du pile vide
  - Tester si une pile est vide

MISE EN OEUVRE D'UNE PILE

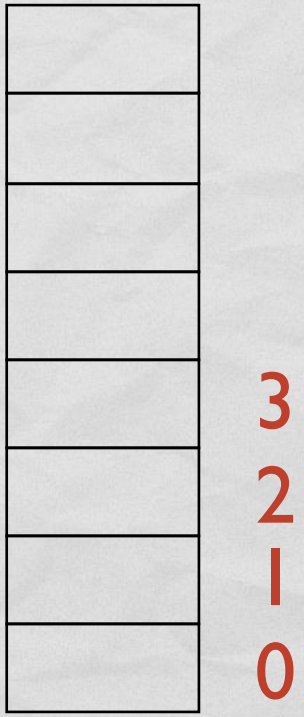
- Plusieurs façon de faire
  - A l'aide d'un tableau
    - Nombre max d'éléments dans la pile est fixée
  - A l'aide d'une liste chaînée
    - Autant d'élément que l'on veut
    - l'élément sur lequel on pointe
      - est l'élément sommet de la pile

MISE EN OEUVRE D'UNE PILE PAR UN TABLEAU

```
Enregistrement
Pile {

  T[NMAX]: entier;

  Sommet : entier;

  }
```

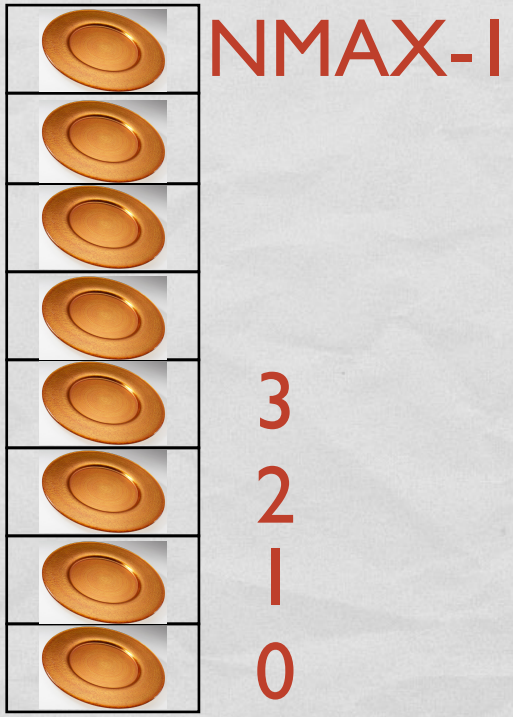
REPRÉSENTATION PAR TABLEAU: PILE VIDE

```
PileVide (p: Pile) : Booléen
Entrées: La pile P
Sortie: renvoie vrai si la pile est vide, sinon faux
Début
  si (p.Sommet = -1)
    retourner vrai;
  sinon
    retourner faux;
  fin si
Fin
```



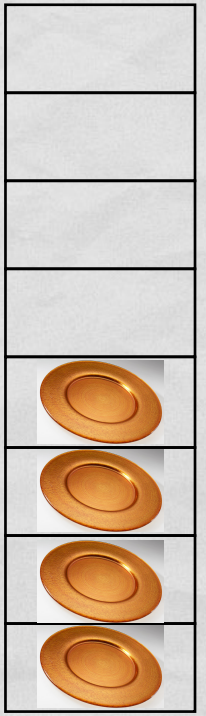
REPRÉSENTATION PAR TABLEAU: PILE PLEINE

```
PilePleine (p: Pile) : Booléen

Entrées: La pile P

Sortie: renvoie vrai si la pile est pleine, sinon
faux

Début
   si (p.Sommet = NMAX-1)
     retourner vrai;
   sinon
     retourner faux;
   fin si
Fin
```



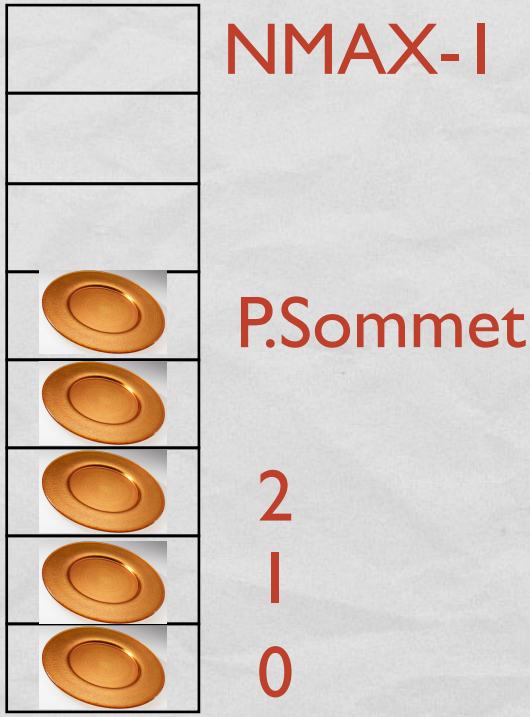
REPRÉSENTATION PAR TABLEAU: INSERTION DANS UNE PILE (EMPILER)

```
Empiler (p: Pile, elt : entier)
Entrées: La pile P et l'élément elt
Sortie: la pile P dans laquelle l'élément elt a été insérer
Début
   si (PilePleine(p)= faux)
     P.Sommet ← P.Sommet+1;
     P.T[P.Sommet] ← elt;
   sinon
     afficher un message d'erreur;
   fin si
Fin
```



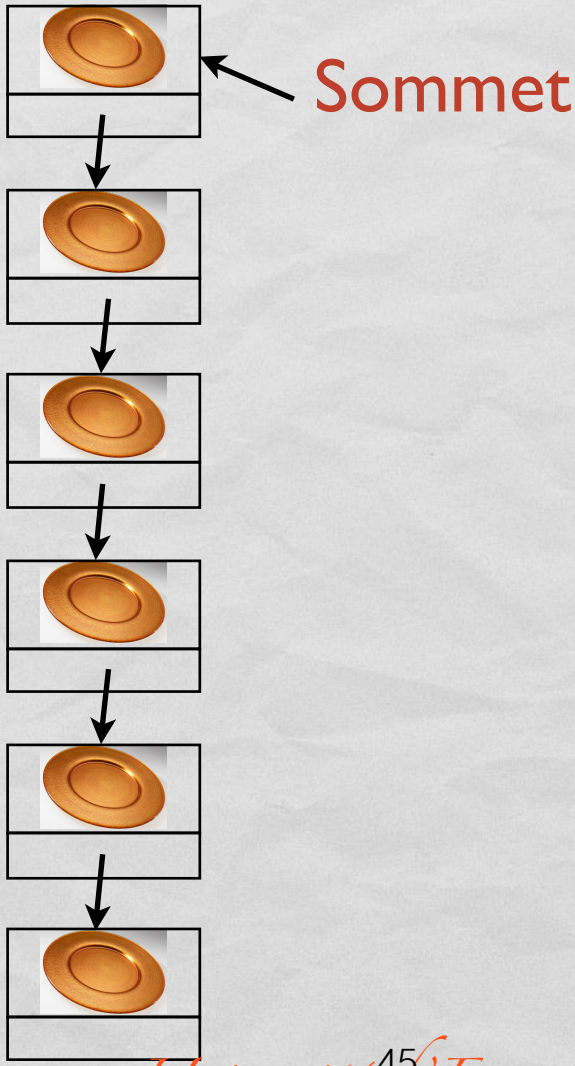
REPRÉSENTATION PAR TABLEAU: SUPPRESSION DANS UNE PILE (DÉPILER)

```
Dépiler (p: Pile, elt : entier)
Entrées: La pile P et l'élément elt
Sortie: la pile P dans laquelle l'élément elt a été supprimer
Début
   si (PileVide(p) = faux)
      elt ← P.T[P.Sommet];
      P.Sommet ← P.Sommet-1;
      retourner elt;
   sinon
      afficher un message d'erreur;
   fin si
Fin
```



MISE EN OEUVRE PAR UNE LISTE CHAÎNÉE

```
Enregistrement Elt_Pile {
  val : entier;
  Suivant : ↑Elt_Pile;
  }
Enregistrement Pile {
  Sommet : ↑Elt_Pile;
  }
```

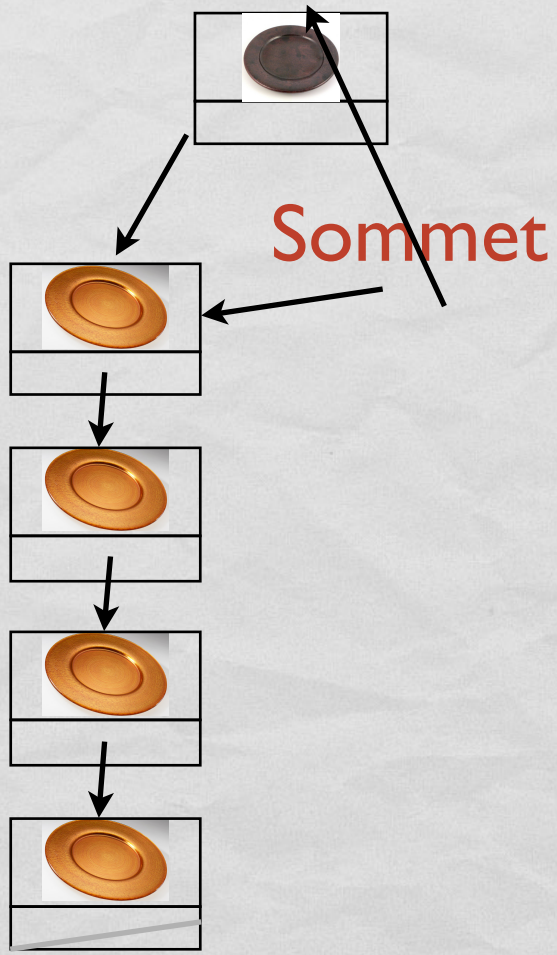


REPRÉSENTATION PAR LISTE CHAÎNÉES: PILE VIDE

```
PileVide (p: Pile) : Booléen
Entrées: La pile P
Sortie: renvoie vrai si la pile est vide, sinon faux
Début
  si (p.Sommet = Nil)
    retourner vrai;
  sinon
    retourner faux;
  fin si
Fin
```

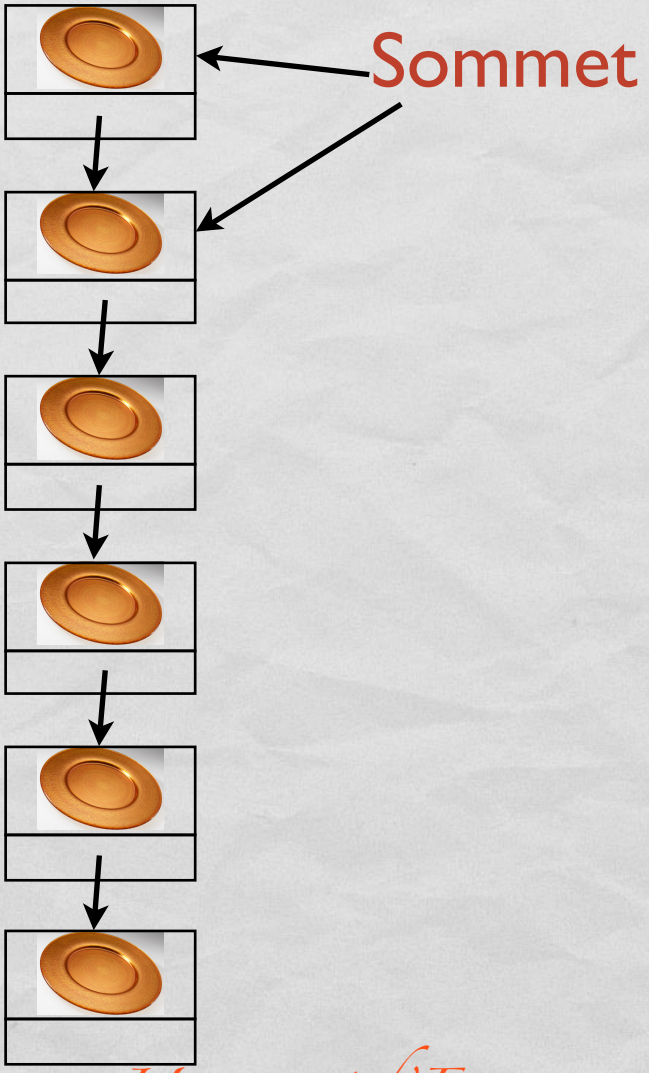
REPRÉSENTATION PAR LISTE CHAÎNÉES: INSERTION DANS UNE PILE (EMPILER)

```
Empiler (p: Pile, elt : entier)

Entrées: La pile P et l'élément elt

Sortie: la pile P dans laquelle l'élément elt a été inserer

Début

   e : Elt_Pile;

   e.Val ← elt;

   e.Suivant ← P.Sommet;

   P.Sommet ← e;

Fin
```



REPRÉSENTATION PAR LISTE CHAÎNÉES: SUPPRESSION DANS UNE PILE (DÉPILER)

```
Dépiler (p: Pile, elt : entier)
Entrées: La pile P et l'élément elt
Sortie: la pile P dans laquelle l'élément elt a été supprimer
Début
   si (PileVide(p) = faux)
      elt ← (P.Sommet).val;
      P.Sommet ← (P.Sommet).Suivant;
      retourner elt;
   sinon
      afficher un message d'erreur;
   fin si
Fin
```



EXEMPLE D'UTILISATION D'UNE PILE

- Je souhaite un programme
  - qui vérifie
    - si le nombre
    - et la place
  - de parenthèses «({[,]})»
  - sont corrects.
- Vérificateur syntaxique de parenthèse

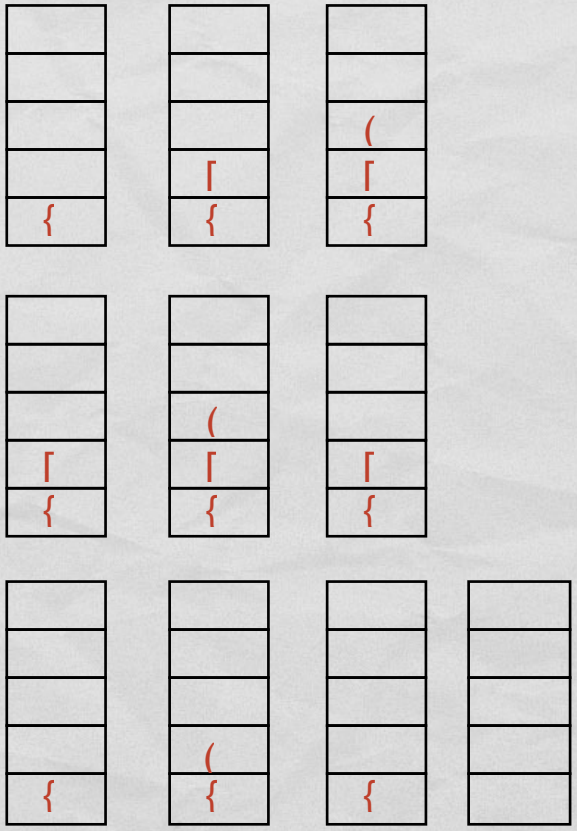
PROGRAMME VÉRIFICATION

```
VérifiePar (p: Pile, ch : chaîne de caractères) Début
  Tant que ch≠NIL faire
    c = premier caractère de Ch
    effacer le premier caractère de Ch
    si (c= «(» ou c=«{» ou c= «[») alors
      Empiler(P,c)
    sinon
      si (c= «)») alors
      e=Dépiler(P)
      si e≠ «(» alors retourner ERREUR
      si (c= «]») alors
      e=Dépiler(P)
      si e≠ «[» alors retourner ERREUR
      si (c= «}») alors
      e=Dépiler(P)
      si e≠ «{» alors retourner ERREUR
  Fin Tant que
  si (¬PileVide(P)) retourner ERREUR
```

PROGRAMME VÉRIFICATION: EXEMPLE

```
VérifiePar (p: Pile, ch : chaîne de caractères)
Début
Tant que ch≠NIL faire
    c = premier caractère de Ch
    effacer le premier caractère de Ch
    si (c= «(» ou c=«{» ou c= «[») alors
       Empiler(P,c)
    sinon
       si (c= «)») alors
          e=Dépiler(P)
          si e≠ «(» alors retourner ERREUR
       si (c= «]») alors
          e=Dépiler(P)
          si e≠ «[» alors retourner ERREUR
       si (c= «}») alors
          e=Dépiler(P)
          si e≠ «{» alors retourner ERREUR
Fin Tant que
si (¬PileVide(P)) retourner ERREUR
```

Ch={a[ORD(c)-ORD('a')],13..MAX(SET)}



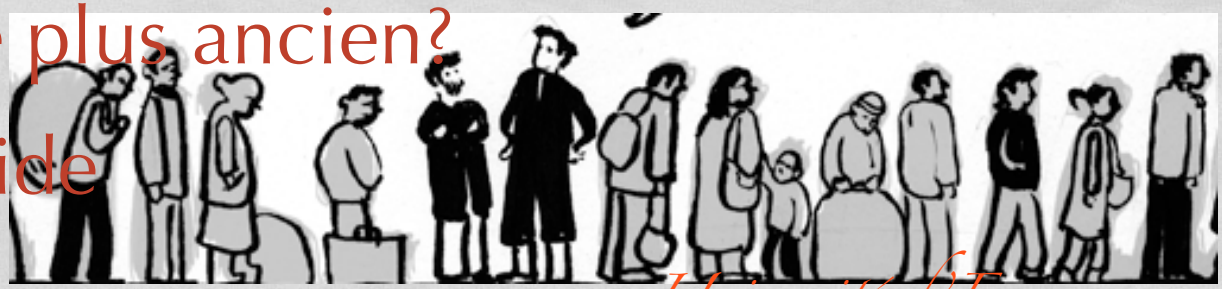
FILES

Lélia Blin

Université d'Evry

UNE FILE

- Analogie de la file d'attente
  - First In First Out (FIFO)
- Opérations principales:
  - Insérer un élément dans une file
  - Supprimer un élément
    - le plus ancien de la file
  - Quel est l'élément le plus ancien?
  - Création d'une file vide

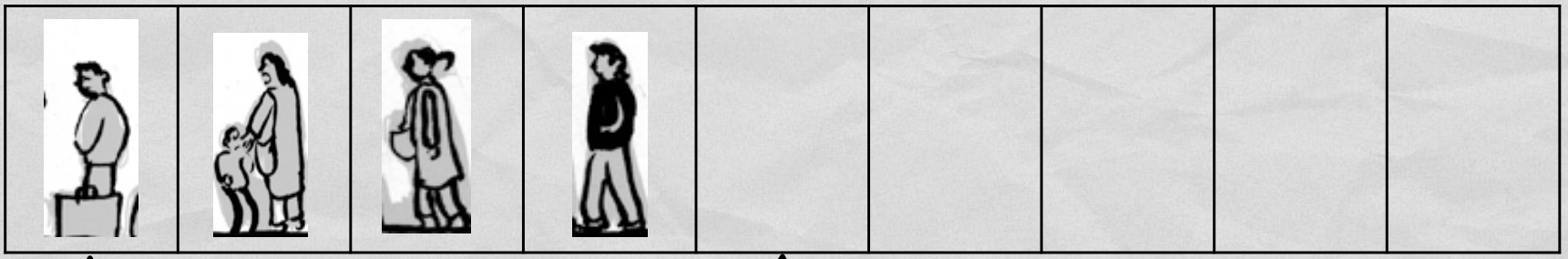


MISE EN OEUVRE D'UNE FILE

- Plusieurs façon de faire
  - A l'aide d'un tableau
    - Nombre max d'éléments dans la file est fixée
  - A l'aide d'une liste chaînée
    - Autant d'élément que l'on veut
    - Il faut mémorisé
      - l'élément le plus récent de la file
      - l'élément le plus ancien

MISE EN OEUVRE D'UNE FILE PAR UN TABLEAU

```
Enregistrement File {

   T[NMAX]: entier;

   Début : entier; Indice de l'élément le plus ancien de
   la file

   Fin: entier; Indice de l'élément le plus récent de la
   file

   }
```

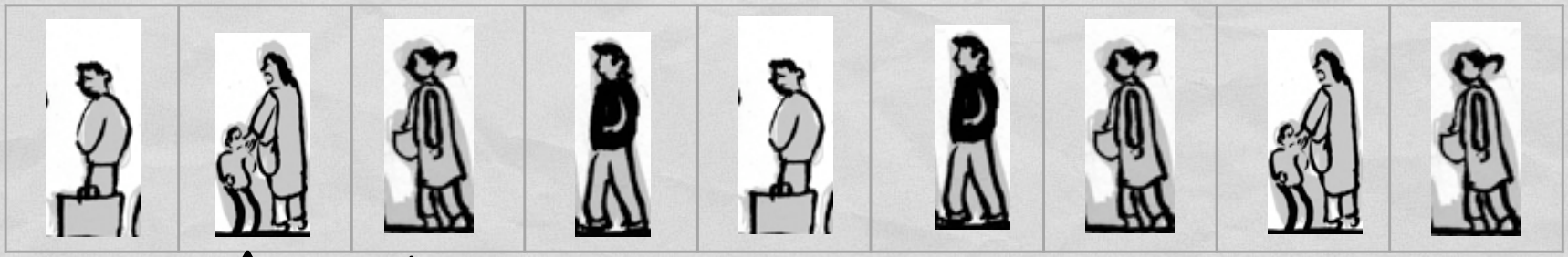


REPRÉSENTATION PAR TABLEAU: FILE VIDE

```
FileVide (f: File) : Booléen
Entrées: La file f
Sortie: renvoie vrai si la file est vide, sinon faux
Début
   si (f.Début = f.fin)
      retourner vrai;
   sinon
      retourner faux;
   fin si
Fin
```

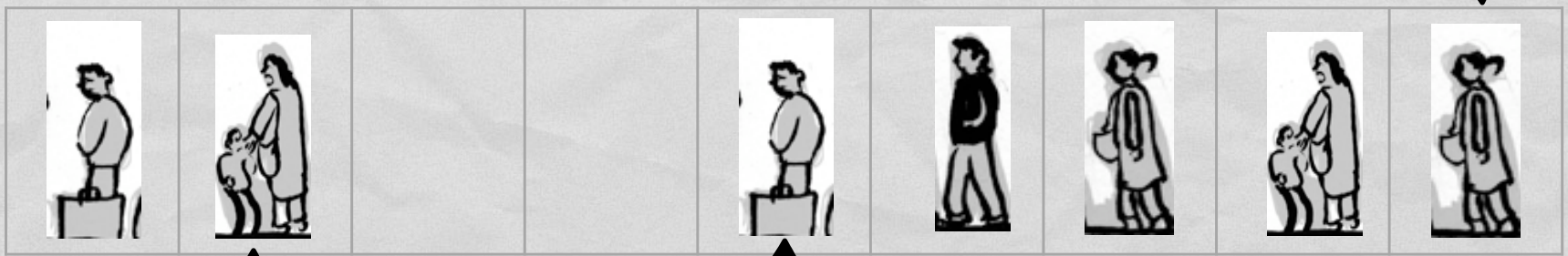
REPRÉSENTATION PAR TABLEAU: FILE PLEINE

```
FilePleine (f: File) : Booléen
Entrées: La file f
Sortie: renvoie vrai si la file est pleine, sinon faux
Début
   si (f.Début = (f.Fin+1) mod NMAX)
     retourner vrai;
   sinon
     retourner faux;
   fin si
Fin
```



REPRÉSENTATION PAR TABLEAU: INSERTION DANS UNE FILE

```
Insertion (f: file, elt : entier)
Entrées: La file f et l'élément elt
Sortie: la file f dans laquelle l'élément elt a été insérer
Début
    si (PilePleine(f) = faux)
        f.T[f.Fin] ← elt;
        f.Fin ← (f.Fin + 1) mod NMAX)
        retourner elt;
    sinon
        afficher un message d'erreur;
    fin si
Fin
```



TABLES DE HACHAGES

Lélia Blin

Université d'Evry

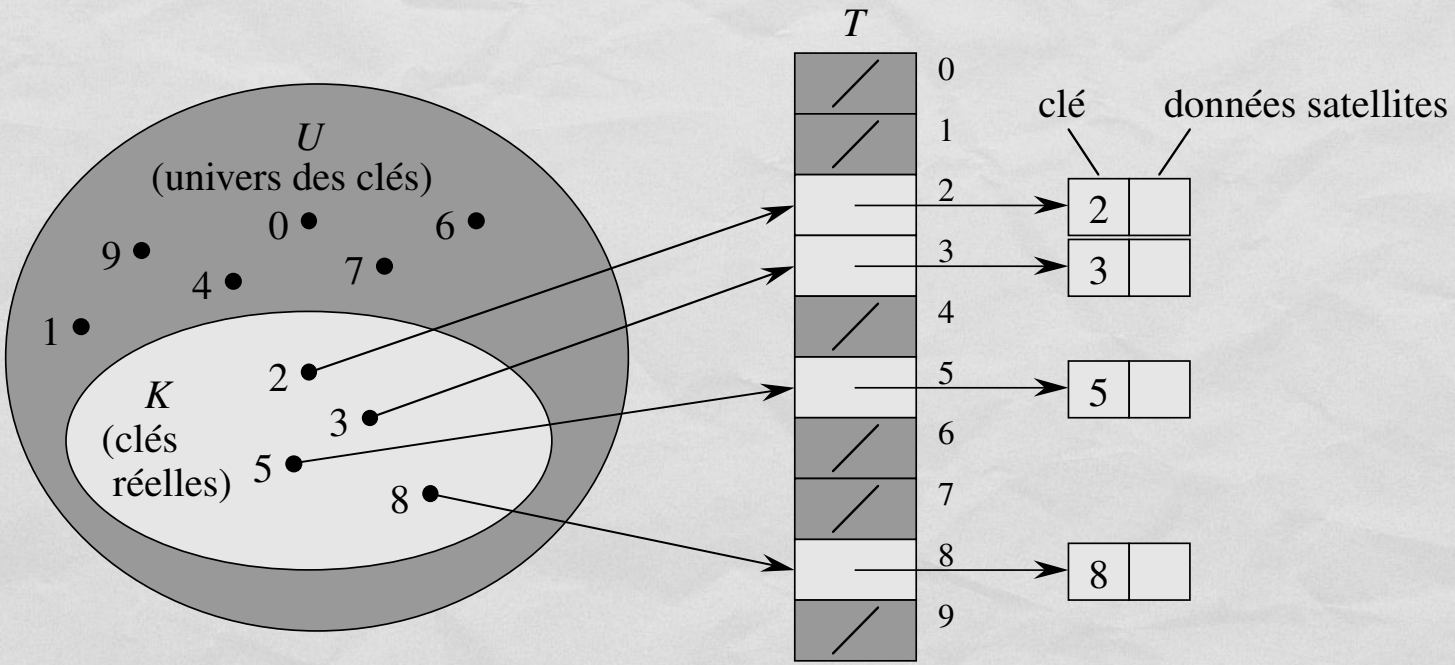
INTRODUCTION

- De nombreuses applications font appel aux ensembles dynamiques qui ne supportent que les opérations de dictionnaire
  - Inserer(x,T)
  - Supprimer(x,T)
  - Rechercher(x,T)
- Exemples :
  - Table des symboles tenue par un compilateur
  - Annuaire téléphonique qui lie les noms aux numéros de téléphone.
  - Table des numéros IP et des adresses Internet

TABLE DE HACHAGE

- Une table de hachage est:
  - une structure de données
  - permettant d'implémenter efficacement des dictionnaires
- **Rm**:
  - Bien que la recherche d'un élément dans une table de hachage
  - puisse être aussi longue que
  - la recherche dans une liste chaînée, en pratique cela reste très éfficace.
- Une table de hachage est la généralisation d'un tableau.

TABLES À ADRESSAGE DIRECT



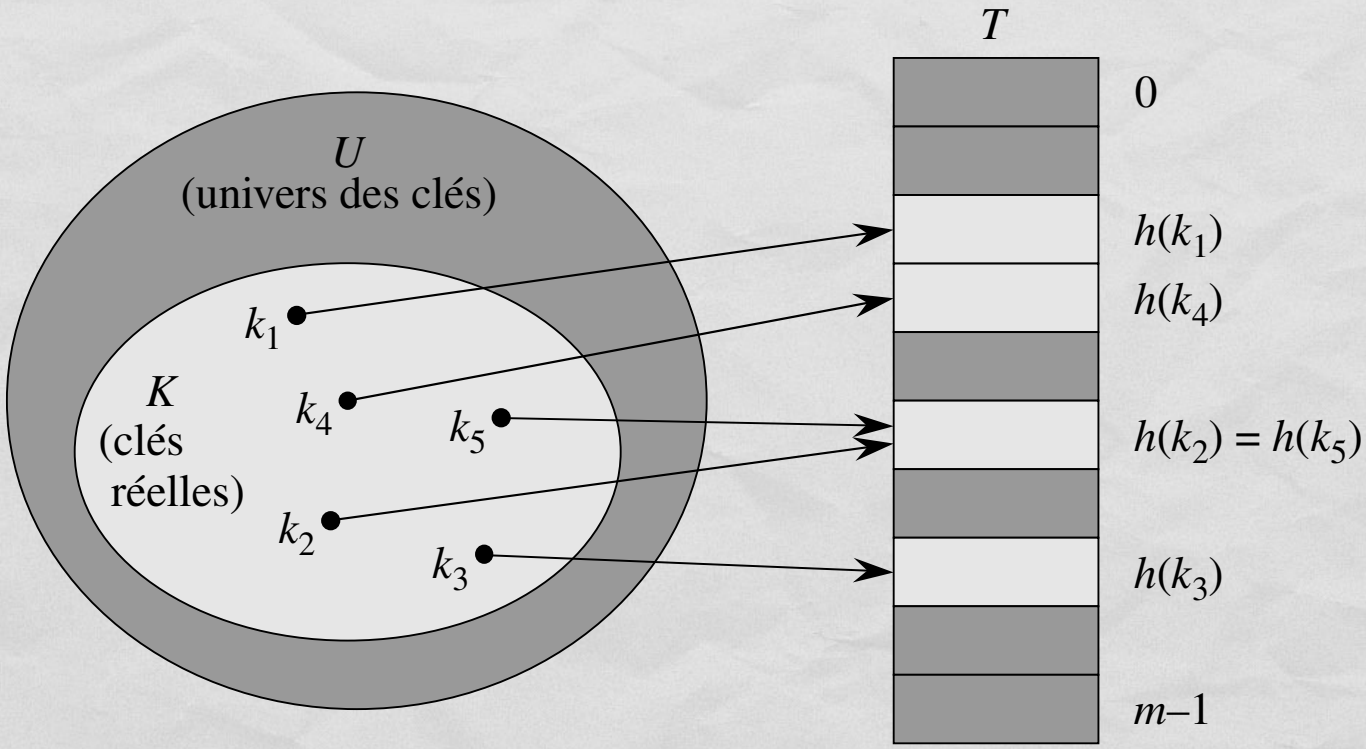
TABLES À ADRESSAGE DIRECT

- L'adressage direct est
  - une technique simple qui fonctionne bien
  - lorsque l'univers U des clés est raisonnablement petit.
- On suppose que deux éléments ne peuvent pas avoir la même clé

TABLES DE HACHAGE

- Inconvénient de l'adressage direct est évident:
  - si l'univers des cles U est grand
  - gérer une table de taille |U| des clés est compliqué.
- Par ailleurs:
  - L'ensemble K des clés réellement conservées
  - peut-être tellement petit comparé à U
  - que la majeure partie de l'espace alloué par T est gaspillé

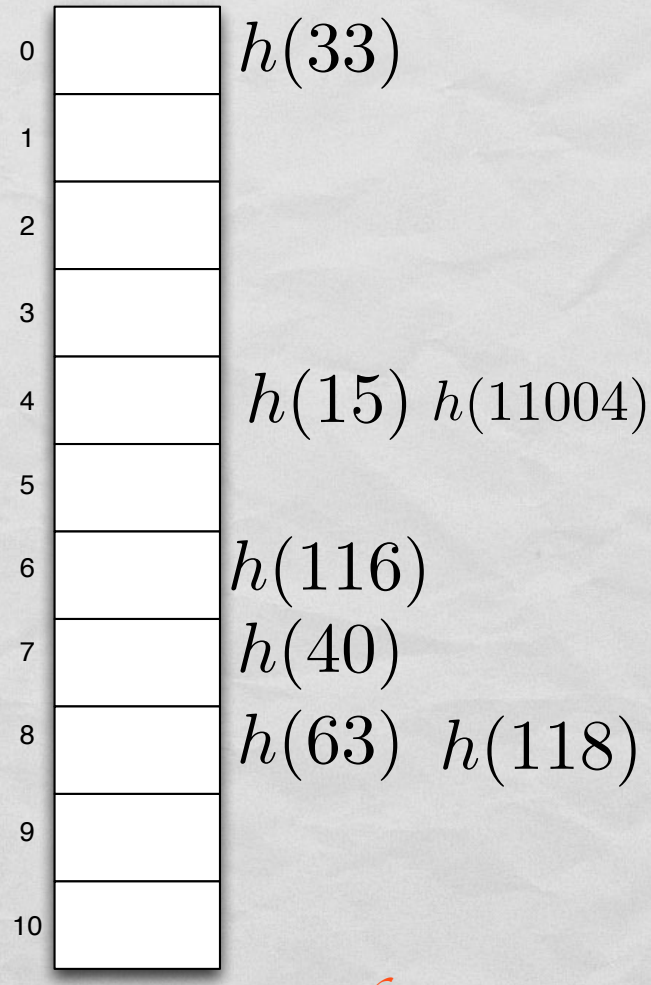
TABLES DE HACHAGE



EXEMPLE

- m=11 Clés={15,33,40,63,116,118,123,11004}
- Fonction de hachage h(k)=k mod m

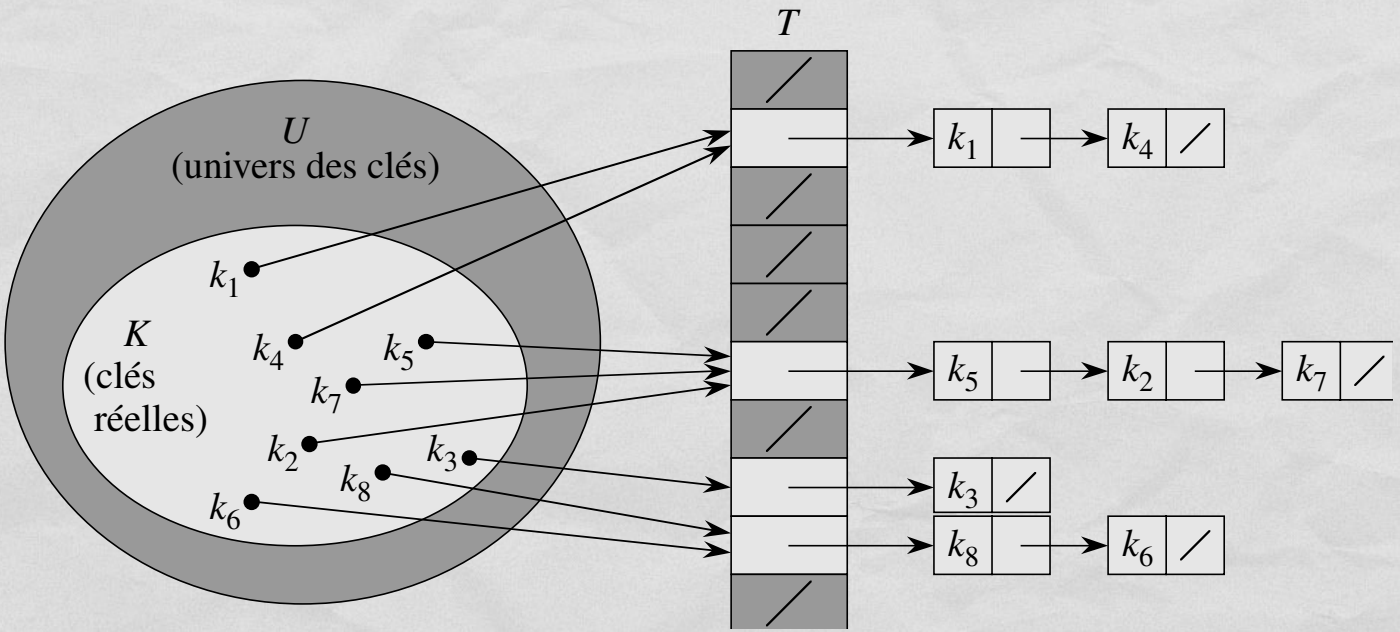
$$h(15) = 15 \pmod{11} = 4$$
$$h(33) = 33 \pmod{11} = 0$$
$$h(40) = 40 \pmod{11} = 7$$
$$h(63) = 63 \pmod{11} = 8$$
$$h(116) = 116 \pmod{11} = 6$$
$$h(118) = 118 \pmod{11} = 8$$
$$h(11004) = 11004 \pmod{11} = 4$$



RÉSOLUTION DES COLLISIONS PAR CHAÎNAGE

- Avec le chaînage:
  - on place dans une liste chaînée
  - tous les éléments hachés
  - vers la même alvéole.
- Chaque alvéole contient un pointeur vers la tête de la liste chaînée.
- Si il n'y a pas d'élément haché l'alvéole contient nil

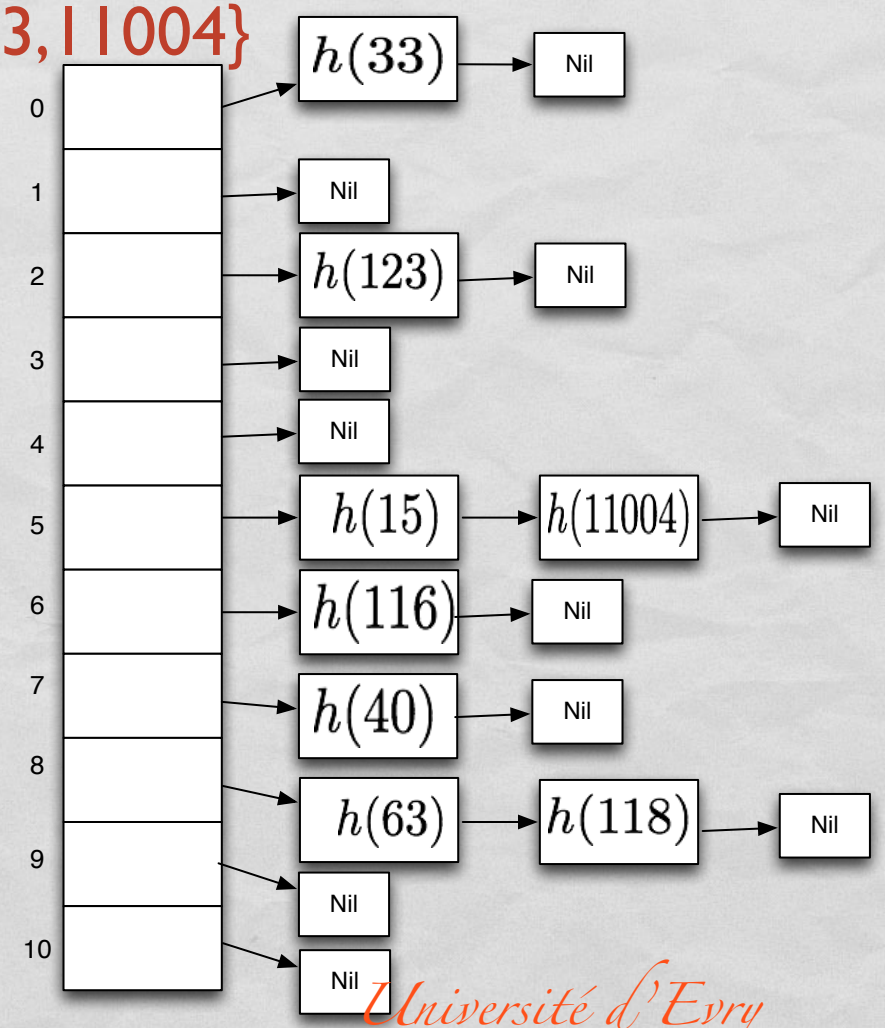
RÉSOLUTION DES COLLISIONS PAR CHAÎNAGE



EXEMPLE

- m=11 Clés={15,33,40,63,116,118,123,11004}
- Fonction de hachage h(k)=k mod m

$$h(15) = 15 \pmod{11} = 4$$
$$h(33) = 33 \pmod{11} = 3$$
$$h(40) = 40 \pmod{11} = 7$$
$$h(63) = 63 \pmod{11} = 8$$
$$h(116) = 116 \pmod{11} = 6$$
$$h(118) = 118 \pmod{11} = 8$$
$$h(11004) = 11004 \pmod{11} = 4$$



LES OPÉRATIONS DE DICTIONNAIRE

- Inserer(x,T):
  - inserer en tête de la liste chaînée: O(1)
- Recherche(k,T):
  - recherche dans la liste chaînée,
  - la complexité dépendent de la longueur de la liste chaîné,
    - si répartition des clés est uniforme: n/m
- Suppression: équivalent à la recherche.

LA FONCTION DE HACHAGE MODULO M

- On suppose que toutes les clés sont des entiers naturels
- A chaque clés on associe une case en prenant le reste de la division de k par m
  - h(k)=k mod m
- Cette méthode elle est simple, mais peut donner de mauvaises répartitions si m est mal choisi
  - Si m admet un petit diviseur d
  - Si $m=10^p$ est une puissance de 10
  - Si $m=2^p$ est une puissance de 2. Si p=6 k=1011000111011010 alors h(k)=011010

COMMENT BIEN CHOISIR M

- On choisit m assez éloigné des puissances de 2 et des puissances de 10
- Exemple: On veut une table de hachage pour conserver environs n=2000 chaînes de caractères.
  - L'examen de 3 éléments en moyenne en cas de recherche n'est pas une catastrophe
  - On alloue un table de m=701
    - 701 nombre premier
    - 701 est proche 2000/3
    - 701 n'est pas proche d'une puissance de 2 ($512=2^8<701<2^9=1024$)
    - Si l'on traite chaque clé comme un entier on h(k)=k mod 701

CHOIX DE LA FONCTION DE HACHAGE

- Pour des chaîne de caractères, on peut appliqué le même principe en prenant pour k le nombre formé en concaténant les caractères.
- Par exemple pour le mot c.h.a.i.n.e
- Dans la base 256
- $H(chaine)=(code(c)*256^5+ code(h)*256^4+ code(a)*256^3+ code(i)*256^2+ code(n)*256^1+ code(e)*256^0) \bmod m$

MÉTHODE DE MULTIPLICATION

- On suppose que toutes les clés sont des nombres entiers
- On suppose que $m=2^r$ et que l'ordinateur travaille avec des mots de w bits.
- On choisit un nombre $0<A<1$ et on défini la fonction de hachage de la façon suivante
  - $h(k)= \lfloor m*(k*A \bmod 1) \rfloor$
  - K*A mod 1= partie fractionnaire de k*A
  - Exemple 11.03015 mod1 =0.03015

MÉTHODE DE MULTIPLICATION

- Pour implémenter h(k)
  - On prend $m=2^r$. On suppose que la taille d'un mot machine est w et que k tient sur 1 mot machine.
  - On calcule $\lfloor (A.2^w) \rfloor$ ce sont les w premiers chiffres de la partie fractionnaire de A
  - On calcule $k^* \lfloor (A.2^w) \rfloor = r_1 2^w + r_0$
  - On prends les r bits les plus significatifs de r_0 et ce sera h(k)

MÉTHODE DE MULTIPLICATION

- Cette méthode fonctionne mieux avec certaines valeurs de A. Knuth a étudié et proposé
  - $A \approx \sqrt{5}-1/2=0.6180339887\ldots$
- Par exemple si k=123456 et m=10000 alors
  - $H(k)= \lfloor 10000.(123456.\ 0.6180339887\ldots.\mathrm{mod}\ 1) \rfloor$
  - $H(k)= \lfloor\ 41.151\ldots.\ \rfloor$