

# Algorithmique et Complexité

## 3. Structures de données avancées

### 3.1 Rappel : Structures de données élémentaires

Nicole Bidoit

Université Paris XI, Orsay

---

Année Universitaire 2008–2009

## Motivation

---

”Mankind’s progress is measured by the number of things we can do without thinking”

attention : pour de futurs experts, **comprendre** ce qu’on fait reste essentiel ... au progrès

### Structures de données élémentaires :

liste, file, pile, tas, ... sont parmi les plus communes

”disponibles clé en main”

### Pour quoi faire ?

représenter des ensembles **dynamiques** et se munir d’opérations de base (recherche, insertion, suppression, ...)

↔ opérations abstraites caractéristiques utilisation transparente

(type abstrait, signature)

↔ implémentation de ces opérations (programmation) optimisation

Exemple (retour arrière) : Algo Voisin proche

Si  $P$  contient  $n$  points et est représenté par un tableau d’entiers  $T[1..n]$  of integer, alors ...

### Structures contiguës versus Structures chaînées

structures contiguës : tableaux, matrices, tas, tables de hachage ...

structures chaînées : listes chaînées, arbres (chaînés), ...

## Structures contiguës : tableaux

---

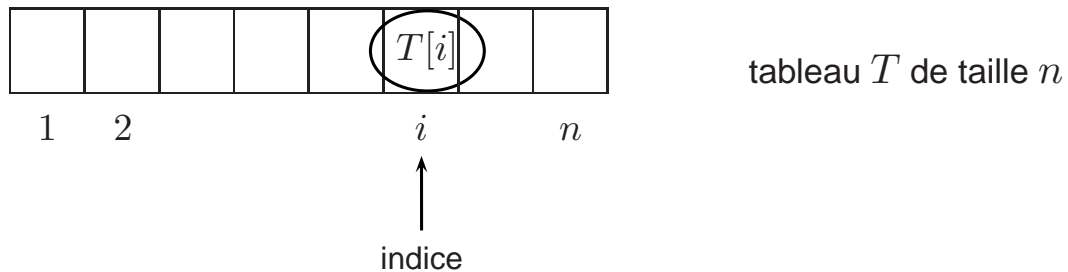
### Un tableau est une liste contiguë d'éléments

chaque élément est localisé "efficacement" par son indice ou adresse

temps d'accès constant à indice connu

aucun espace "perdu" pour stocker des liens ou autres ...

la localité (contiguïté physique) permet d'exploiter les mémoires caches rapides



la taille d'un tableau (simple) ne peut pas être modifiée au milieu d'un algorithme/programme

anticiper la croissance d'un tableau en prévoyant  $n$  (sa taille) très grande  $\rightarrow$  perte d'espace

## Structures contiguës : tableaux et tableaux dynamiques

Un **tableau dynamique** permet de doubler la taille de  $n$  à  $2n$  "à la demande".

En supposant que  $T$  soit de taille 1 au départ, combien de fois faut-il doubler la taille de  $T$  pour y insérer  $n$  éléments ?  $\lceil \log_2 n \rceil$

Coût de la procédure (structure) = recopie des éléments dans l'extension

la moitié des  $n$  éléments du tableau "final" a été copiée 1 fois

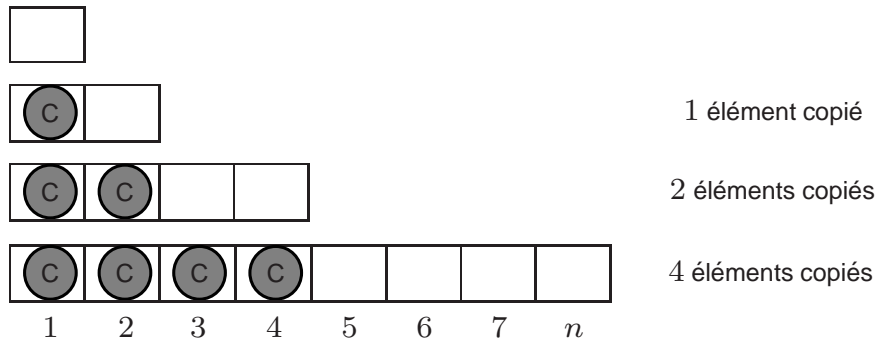
le quart des  $n$  éléments a été copiée 2 fois

le ...

$$\text{nb copies} = \sum_{i=1..lg n} i \cdot n / 2^i = n \sum_{i=1..lg n} i / 2^i \leq n \sum_{i=1..∞} i / 2^i = 2n$$

Réponse : chacun des  $n$  éléments a été copié deux fois "en moyenne";

la gestion du tableau dynamique est en  $O(n)$  : pareil que si on avait surdimensionné  $T$  à  $n$  au départ.



## Structures chaînées : Pointeurs et listes chaînées

### Un pointeur représente une adresse d'un élément

(le numéro de votre portable est un pointeur ... sur vous)

Notation :  $p$ : pointeur     $p \uparrow$  : accès élément d'adresse  $p$      $null$  : valeur particulière d'un pointeur

**Une liste chaînée** est une liste dont l'ordre des éléments est déterminé par le pointeur liant un élément à son successeur.

Déclaration - Notation : type listeE = ( el : elt, suiv : listeP)

type listeP = pointeur listeE

### Recherche d'un élément dans une liste chaînée :

Algo RechSeqCh

entrée :  $p$  : listeP

% (l'adresse d')une liste d'éléments

$e$  : elt

% élément recherché

sortie :  $In$  boolean

% vrai si  $e$  dans  $T$

---

**if** ( $p = null$ ) **then**  $In \leftarrow false$  **else**

**if** ( $e = p \uparrow . el$ ) **then**  $In \leftarrow true$  **else**  $In \leftarrow RechSeqCh(p \uparrow . suiv, e)$ ;

version récursive

---

**Exercice** : Complexité de Algo RechSeqCh ? Insertion dans, suppression de, longueur d'une liste chaînée.

## Structures chaînées : Pointeurs et listes chaînées

---

### Avantages des structures (listes) chaînées

pas de débordement ... à moins que la mémoire soit saturée

insertion et suppression sont simples ... plus simples que pour un tableau

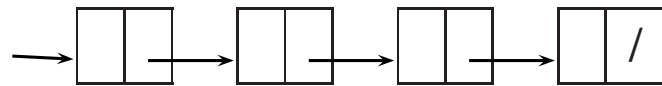
pour des éléments de grande taille, le déplacement d'éléments est plus facile et plus rapide (déplacement de pointeurs)

### Listes doublement chaînées

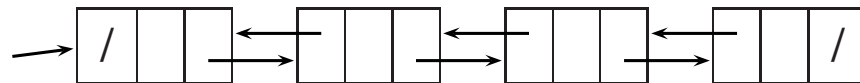
Rappel déclaration : type liste2E = ( el : elt, suiv : listeP, pred : listeP)

↪ flexibilité supplémentaire pour l'insertion, suppression

↪ coût en espace



Liste chaînée



Liste doublement chaînée

## Listes triées, Files et Piles

---

**Liste triée** : on dispose d'un ordre sur les éléments et la liste est organisée suivant cet ordre.

si la liste est représentée par un tableau alors

ordre des éléments  $\leftrightarrow$  ordre des indices :  $i \leq j \equiv T[i] \leq T[j]$ .

**File et Pile** sont des listes non triées (au sens ci-dessus) dont l'organisation respecte l'**ordre d'arrivée** des éléments.

$\leftrightarrow$  si la Xile est représentée par un tableau alors

$i \leq j \equiv \text{date-insertion}(T[i]) \leq \text{date-insertion}(T[j])$  ou inversement.

### Politiques d'insertion et suppression distinctes

	File	Pile
insertion	en queue	en tête
suppression	en tête	en tête

files d'attente au guichet de ..., de tâches, ...

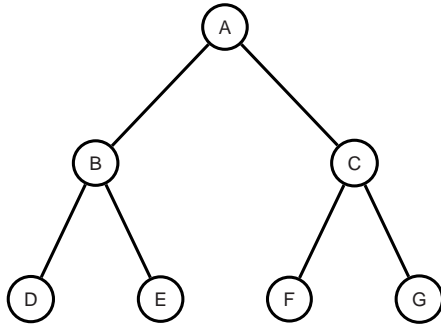
pile de livres, de disques, de tâches, ...

### Représentations contiguë et chaînée de files et de piles

classiques

## Impact du choix des structures de données

### File et Pile pour Parcours d'arbre



Arbre binaire

Parcours avec une file :

A B C D E F G

Parcours avec une pile :

A C G F B E D

### Brouillon pour l'algorithme de Parcours utilisant une Xile

initialisation

... on mettra la racine de l'arbre dans la Xile X

itération

... on itère tant que la Xile X n'est pas vide

$n \leftarrow$  supprimer(X);    print(...);

insérer(fils\_gauche(n),X);

insérer(fils\_droit(n),X);

### Parcours-File versus Parcours-Pile : Stratégies d'insertion et de suppression

**À retenir :**    Précisez bien les structures de données

↔ impact sur la **correction** de l'algorithme

↔ impact sur la **complexité** de l'algorithme

## Structures élémentaires et complexité (au pire)

	cas non trié			cas trié		
	tableau	liste chaînée	liste 2 x chaînée	tableau	liste chaînée	liste 2 x chaînée
Rechi( $E, k$ )	$\Theta(1)$			$\Theta(1)$		
Reche( $E, e$ )	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\lg n)^\dagger$	$\Theta(n)$	$\Theta(n)$
Inser( $E, e$ )	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Supp( $E, x$ )	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Succ( $E, x$ )	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Pred( $E, x$ )	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Min( $E$ )	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Max( $E$ )	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)^\ddagger$

**Terminologie :** clé de recherche  $\neq$  élément

hypothèse : clé de recherche = élément

Notation :  $k$  : indice tableau     $e$  : élément

$x$  : indice tableau / pointeur liste (en fonction du contexte)

**Ce tableau mérite quelques explications, schémas, commentaires ... (au tableau)**

$\dagger$  avec l'algorithme de recherche dichotomique

$\ddagger$  en faisant une liste 2 x chaînée et **circulaire!**

## Algorithmique et Complexité

### 3. Structures de données avancées

#### 3. 2 Tables de hachage

Nicole Bidoit

Université Paris XI, Orsay

---

Année Universitaire 2008–2009

## HACHAGE - Tables de Hachage

---

### Pour quel besoin ?

Recherche d'un élément (par sa clé) dans un grand ensemble dynamique

Insertion    Suppression

### Très nombreuses d'applications :

Structures de contrôle système : table des symboles, table des transactions, ...

Maintenir et accéder à un dictionnaire, une base de données, ...

Optimisation

### Principe de base :

Pour un tableau, le temps d'accès à un élément en fonction de son indice est **constant**

Rappel : complexité au pire de  $\text{Rechi}(T, k)$  est dans  $\Theta(1)$

**fonction de hachage :**     $h : \text{clé} \longrightarrow \text{indice}$

## HACHAGE - Tables de Hachage

---

### Cas simple : table à adressage direct

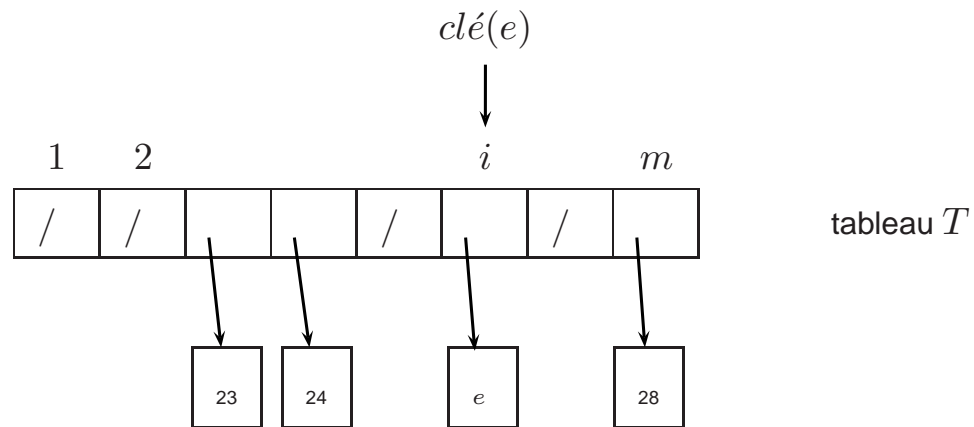
$E$  est un sous-ensemble de  $U$  contenant au plus  $m$  éléments connus;

$h$  est une fonction  $U \rightarrow [1..m]$  telle que  $e \neq e' \Rightarrow h(e) \neq h(e')$        $h(e)$  détermine  $e$

Rechercher un élément  $e$  :      accès au tableau  $T$  avec indice  $h(e)$

Insérer un élément  $e$  :      idem

Supprimer un élément  $e$  :      idem



## HACHAGE - Tables de Hachage

---

**Problème de l'adressage direct :**  $U$  grand

idée 1 : gérer un tableau  $T$  de grande taille

oui ... mais l'ensemble  $E$  peut être petit

**Exemple :**  $U$  ensemble de toutes les chaînes de caractères  
les mots du dictionnaire de langue française

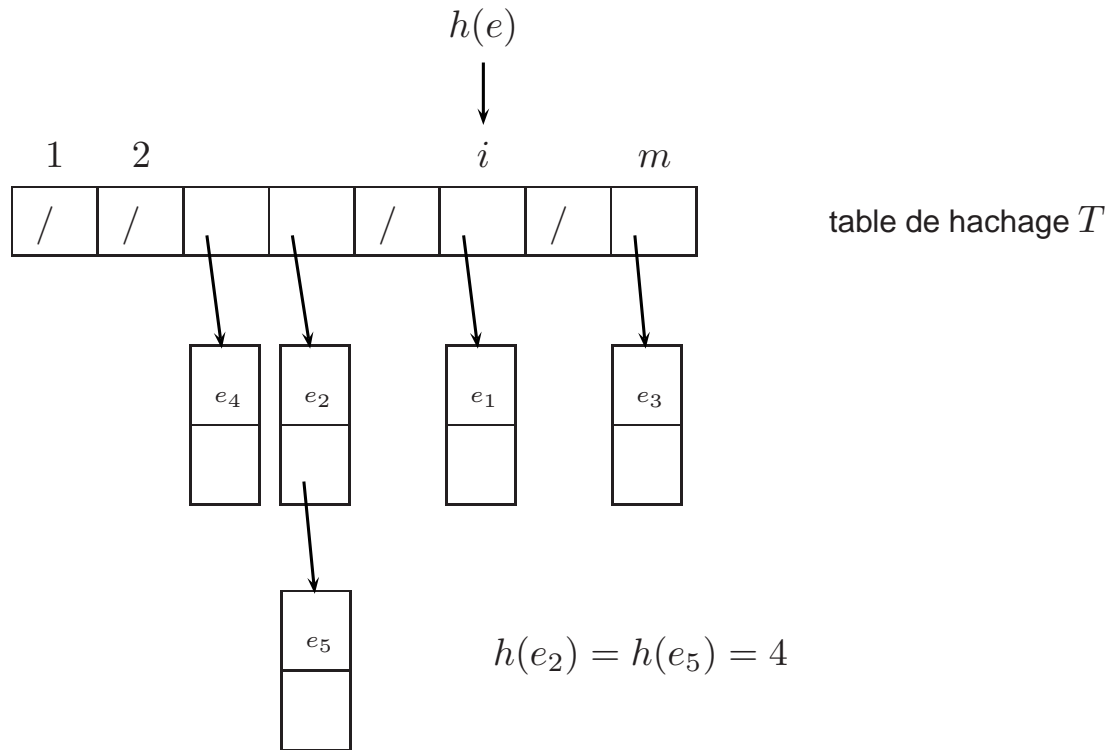
si  $|E|=n$  plus petit que  $|U|=m$  alors une **table de hachage** requiert moins d'espace que l'adressage direct.

**Objectif :** réduire l'intervalle des indices générés par  $h$

La fonction de hachage  $h$  ne vérifie plus

$$e \neq e' \Rightarrow h(e) \neq h(e') !$$

## HACHAGE - Tables de Hachage



**Problème des tables de hachage : Collision** ie  $e \neq e'$  et  $h(e) = h(e')$

**Solution simple : chaînage**

Chaque entrée  $T[i]$  contient un pointeur sur une liste d'éléments  $e$  tels que  $h(e) = i$ .

## Tables de hachage

---

**Recherche** d'un élément dans une table de hachage :

pire des cas<sup>†</sup> =  $O(n)$

RechTH( $T, e$ ) :

en moyenne = ??

rechercher  $e$  dans la liste  $T(h(e))$ ;

**Insertion** d'un élément dans une table de hachage :

pire des cas =  $O(1)$

InserTH( $T, e$ ) :

Insertion de  $e$  dans la liste  $T(h(e))$ ;

**Suppression** d'un élément dans une table de hachage :

pire des cas =  $O(1)$  si liste 2 x chaînée

SuppTH( $T, e$ ) :

Suppression de  $e$  de la liste  $T(h(e))$ ;

**La complexité en moyenne** de la recherche dans une table de hachage dépend de la "répartition" des éléments dans la table par la fonction de hachage.

<sup>†</sup> on suppose que le coût du calcul de  $h(e)$  est dans  $O(1)$

## Tables de hachage

---

### Qu'est-ce qu'une bonne fonction de hachage ?

1. facile (complexité) à calculer
2. utilisation des  $m$  entrées de la table avec une fréquence uniforme

Si  $P(e)$  est la probabilité que  $e$  soit "tiré", l'hypothèse de hachage uniforme correspond à :

$$\sum_{\{e|h(e)=j\}} P(e) = \frac{1}{m} \quad \text{pour } j = 1..m$$

3. le hachage des éléments similaires est discriminant :  
ces éléments "similaires" sont placés dans des entrées d'indices "éloignés".

### Concrètement ?

1.  $e \longrightarrow \text{entier}$  :  $\text{integer}(e) = \sum_{i=0..|e|} 128^i \cdot \text{char}(e[i])$

2. "ramener  $\text{integer}(e)$  à une valeur dans l'intervalle  $[1..m]$

méthode de la division :  $h(e) = (\text{integer}(e) \bmod m) + 1$

méthode de la multiplication :  $h(e) = \lfloor m(kA \bmod 1) \rfloor + 1$  avec  $0 < A < 1$

hachage universel : collection finie de fonctions de hachage

### Exemples :

hachage basé sur les 3 premiers chiffres du numéro de sécurité sociale

hachage basé sur les 3 derniers chiffres " "

## HACHAGE - Tables de hachage

---

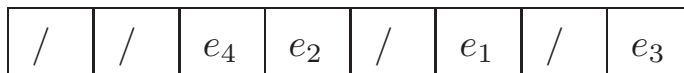
### Autre solution au problème de collision : Adressage ouvert

les éléments sont insérés dans la table (pas de liste d'éléments, pas de pointeurs)

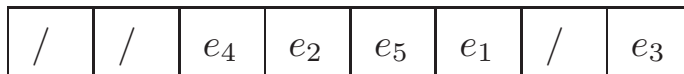
gain d'espace → table plus grande donc moins de collisions, ...

### Ça marche comment ?

1. pas d'indirection : la table contient les éléments
2. collision (lors de l'insertion) : recherche d'une place vide pour insérer l'élément



$T$  géré par adressage ouvert



insertion de  $e_5$  sachant que  $h(e_2) = h(e_5)$

$h(e_5)+1$

**Stratégie utilisée:** essayer de placer l'élément à coté (à droite)

... il y a d'autres stratégies

## HACHAGE - Tables de hachage

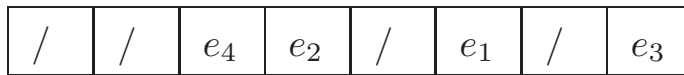
### Stratégies de placement pour la gestion des collisions :

Séquentielle :  $h(e'), h(e')+1, \dots, [h(e')+i \bmod m]+1, \dots, h(e')-1$

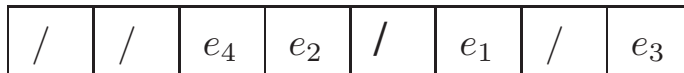
Linéaire :  $h(e'), h(e')+c_1, \dots, [h(e')+c_1.i \bmod m]+1, \dots,$

Quadratique :  $h(e'), h(e')+1^2, \dots, [h(e')+c_1.i+c_2.i^2 \bmod m]+1, \dots,$

Double hachage :  $h(e'), h(e')+h'(e'), \dots, [h(e')+i.h'(e') \bmod m]+1, \dots,$



$T$  géré par adressage ouvert



insertion de  $e_5$  sachant que  $h(e_2) = h(e_5)$

?? $e_5$ ??

**Discussion :** taux de remplissage versus stratégie, complexité versus stratégie, ...

## Fonction de hachage

---

**Problème :** recherche d'un mot  $M$  dans un texte  $T$

Exemple : Est-ce que recherche est dans mes transparents ?

**Méthode naïve :** à chaque position  $i$  du texte on essaie de vérifier la correspondance entre  $M$  et le texte à partir de  $i$ .

Complexité :  $O(tm)$  avec  $|T|=t$  et  $|M|=m$   
opération comptée= comparaison de 2 caractères

**Méthode hachage :** soit  $h$  une fonction de hachage; à chaque position  $i$  du texte on calcule  $h(T[i, \dots, i+m-1])$   
si  $T[i, \dots, i+m-1]=M$  alors  $h(T[i, \dots, i+m-1])=h(M)$   
si  $T[i, \dots, i+m-1] \neq M$  alors  $h(T[i, \dots, i+m-1])=h(M)$  peu probable (faux positif)

Complexité :  $O(tm)$  en supposant que le calcul de  $h$  est dans  $O(1)$

**Le truc** la fonction de hachage à position  $i$ :

$$h(T[i, \dots, i+m-1]) = \sum_{j=0..m-1} \alpha^{m-(j+1)} \cdot \text{char}(T[i+j])$$

donc à position  $i+1$ ,

$$h(T[i+1, \dots, i+m]) = (h(T[i, \dots, i+m-1]) - \alpha^{m-1} \cdot \text{char}(T[i])) \cdot \alpha + \text{char}(T[i+m])$$

Complexité : ??

## Autres applications du hachage :

---

### Accélérer la recherche ...

Bases de données

Recherche d'Information

Web

### Comparer des documents (ou des données massives)

Ex : est-ce qu'un nouveau document est vraiment nouveau relativement à un corpus de documents existants ?

Ex : Modification d'un fichier