

INF421-B

Bases de la programmation et de l'algorithmique

(Bloc 3/ 9)

Philippe Baptiste

CNRS LIX, École Polytechnique

17 novembre 2010

Un exemple : Simuler une file d'attente

- ▶ Attente dans une file pour un service
- ▶ Phénomène stochastique :
 - ▶ Arrivée aléatoire des clients
 - ▶ Temps de service aléatoire
 - ▶ Patience limitée des clients (seuil de tolérance aléatoire)
 - ▶ ...

Les arrivées, les temps d'attente, etc., sont des variables aléatoires qui suivent des lois de probabilité données. On cherche des caractéristiques du systèmes (temps d'attente moyen, ratio de clients exaspérés, etc).

- ▶ Étude théorique des files d'attente (RO stochastique, résultats analytiques)
- ▶ Simulation informatique (seule méthode possible pour des situations complexes)

Aujourd'hui

Piles / Files (suite & fin)

Les génériques

La problématique de l'association

Les tables de hachage

Hacher en java

Un exemple : Simuler une file d'attente

- ▶ Un unique guichet ouvert 8h00 ($8 * 3600 = 28800$ secondes) consécutives.
- ▶ Les clients arrivent et font la queue. La probabilité d'arrivée d'un nouveau client sur un intervalle $[t, t + 1)$ est p (elle ne dépend pas de ce qui s'est passé avant l'instant t et la probabilité d'arrivée de plusieurs clients dans $[t, t + 1)$ est "négligeable" \rightarrow Poisson).
- ▶ Temps de service (loi uniforme sur $[30, 300)$).
- ▶ Chaque client est plus ou moins patient (le temps après lequel il part sans être servi \rightarrow loi uniforme sur $[120, 1800)$).

OBJECTIF : Calculer le ratio de clients qui partent sans être servis en fonction de p .

Simuler une file d'attente : Le client & la file

```
class Client {
    int arrivée;
    int seuil;
    Client(int arrivée, int seuil) {
        this.arrivée = arrivée;
        this.seuil = seuil;
    }
}
class File {
    static final int maxF = 1000;
    int début, fin;
    Client[] contenu;
    File() {
        début = 0;
        fin = 0;
        contenu = new Client[maxF];
    } ...
}
```

Simuler une file d'attente : La simulation

On dispose

- ▶ d'un générateur aléatoire
 - ▶ `RandGen.rnd()` renvoie un réel dans $[0, 1)$ tiré aléatoirement
 - ▶ `RandGen.rnd(a)` renvoie un entier dans $[0, a)$ tiré aléatoirement
 - ▶ `RandGen.rnd(a, b)` renvoie un entier dans $[a, b)$ tiré aléatoirement
- ▶ d'un module qui permet de lancer plusieurs simulations et de faire des moyennes
- ▶ de `gnuplot` qui permet d'afficher des tableaux de données

Simuler une file d'attente : La simulation

Un algorithme naïf : simulation discrète (seconde par seconde)

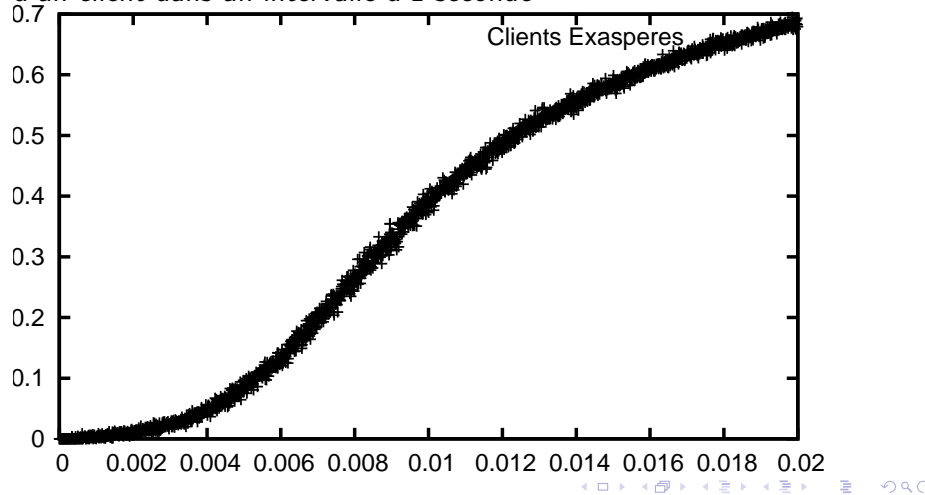
- ▶ À chaque seconde t , faire un tirage aléatoire pour simuler l'arrivée d'un client
 - ▶ Si un client arrive, "l'enfiler" et tirer aléatoirement son seuil d'attente
- ▶ Conserver dans une variable `libre` la prochaine date à laquelle le guichet se libère
- ▶ Quand un client est disponible et que le guichet est libre,
 - ▶ "défiler" le client
 - ▶ Vérifier que le client n'est pas parti
 - ▶ tirer aléatoirement un temps de traitement,
 - ▶ mettre à jour `libre`

Simuler une file d'attente : La simulation

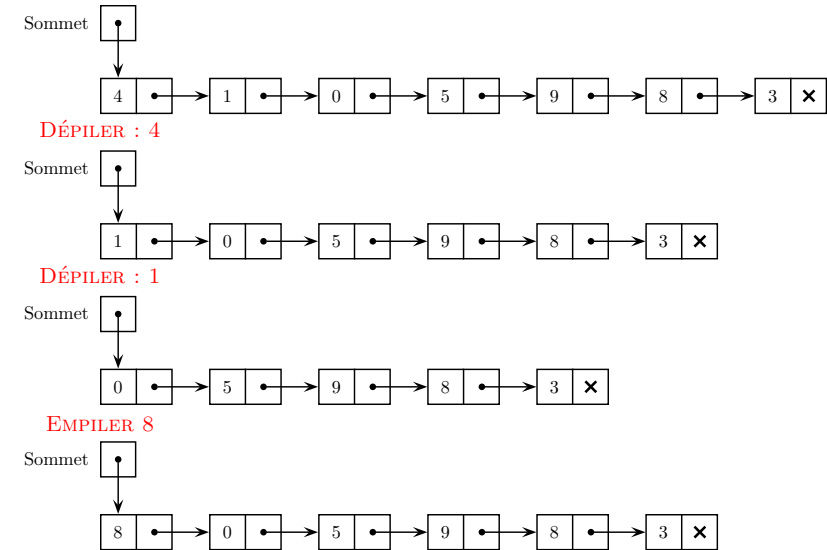
```
File f = new File (); int libre = 0;
int clientsArrivés = 0; int clientsExaspérés = 0;
for (int t = 0; t < tMax; t++) {
    if (RandGen.rnd() ≤ probArriveeT) {
        clientsArrivés ++;
        f.ajouter(new Client(t, RandGen.rnd(seuilMin, seuilMax)));
    }
    if (libre ≤ t) {
        Client c = null;
        while (! f.estVide()) { // exaspérés -> partis
            c = f.défiler();
            if (t - c.arrivée ≤ c.seuil)
                break;
            clientsExaspérés ++;
            c = null;
        } // maintenant le client à servir est c
        if (c ≠ null) libre = t + RandGen.rnd(serviceMin, serviceMax);
    }
}
```

Simuler une file d'attente

Le ratio de clients exaspérés en fonction de la probabilité d'arrivée d'un client dans un intervalle d'1 seconde



Comment coder une pile ? avec une liste



Comment coder une pile ?

Très simplement avec une liste (ou un tableau).

```

class Liste {
    int valeur;
    Liste suivant;
    Liste(int v, Liste s) {
        valeur = v;
        suivant = s;
    }
}

class Pile {
    Liste l;
    public Pile() {
        l = null;
    }
}

public static boolean estVide(Pile p) {
    return p.l == null;
}

void vider() {
    l = null;
}

static void ajouter(Pile p, int a) {
    p.l = new Liste(a, p.l);
}

static int retirer(Pile p) {
    int a = p.l.valeur;
    p.l = p.l.suivant;
    return a;
}
    
```

Un investissement : la HP35 ou la fin de la règle à calcul



- ▶ janvier 1972 : la HP35, 1ère calculatrice de poche scientifique pour \$ 395.
- ▶ Très bon investissement (\$ 515 sur ebay 33 ans plus tard)
- ▶ RPN (Reverse Polish Notation)
 - ▶ développée en 1920 par Jan Lukasiewicz (formules sans parenthèses ni de crochets)
 - ▶ Exemple : $(3+5) / (7+6) = ?$. "Appuyez sur 3 puis sur la touche ENTER. Appuyez sur 5 puis sur la touche +. Appuyez sur 7, puis sur la touche ENTER. Appuyez sur 6 puis sur la touche +. Appuyez sur la touche de division et la calculatrice vous donne le résultat : 0,62."

La notation Polonaise inverse (RPN)

- ▶ Placer des nombres dans une pile
- ▶ effectuer des opérations sur les 2 nombres situés au sommet de la pile

Évaluation de "4 12 23 7 + + 3 * 121 - * "

```
LIT : 4      Pile : 4
LIT : 12     Pile : 12 4
LIT : 23     Pile : 23 12 4
LIT : 7      Pile : 7 23 12 4
LIT : +      Pile : 30 12 4
LIT : +      Pile : 42 4
LIT : 3      Pile : 3 42 4
LIT : *      Pile : 126 4
LIT : 121    Pile : 121 126 4
LIT : -      Pile : 5 4
LIT : *      Pile : 20
```



RPN

Lire les caractères l'un après l'autre et appliquer les règles suivantes

- ▶ Si on rencontre un chiffre et que le caractère précédent en était déjà un, "poursuivre" la construction du nombre
- ▶ Si on rencontre un chiffre et que le caractère précédent n'était pas un, "commencer" la construction du nombre
- ▶ Si on rencontre un caractère qui n'est pas un chiffre alors que le caractère précédent en était un, "empiler" le nombre construit
- ▶ Si on rencontre un caractère qui est un opérateur, dépiler deux fois, calculer et empiler



RPN

Algorithme : Utiliser une pile pour enregistrer les nombres et traiter les opérateurs "on the fly".

Pour analyser la chaîne de caractères :

- ▶ La classe String permet les opérations usuelles sur les chaînes de caractères.
- ▶ Attention, les objets de la classe String sont **immuables** (variante destructrice des chaînes de caractères, la classe StringBuffer)
- ▶ Quelques méthodes utiles de la classe String : **int** length() **char** charAt(**int** index), **boolean** equals(String s)



RPN

```
Pile p = new Pile(); int nbEnCours = 0; boolean lectNbEnCours = false;
for (int i = 0; i < args[0].length(); i++) {
    char c = args[0].charAt(i);
    if ((c <= '9') && (c >= '0')) {
        lectNbEnCours = true; nbEnCours = 10 * nbEnCours + c - 48;
    }
    else if (lectNbEnCours) {
        Pile.ajouter(p, nbEnCours);
        lectNbEnCours = false; nbEnCours = 0;
    }
    if (c == '+')
        Pile.ajouter(p, Pile.retirer(p) + Pile.retirer(p));
    if (c == '-')
        Pile.ajouter(p, - Pile.retirer(p) + Pile.retirer(p));
    if (c == '*')
        Pile.ajouter(p, Pile.retirer(p) * Pile.retirer(p));
}
```



Aujourd'hui

Piles / Files (suite & fin)

Les génériques

La problématique de l'association

Les tables de hachage

Hacher en java

Classes génériques (paramétrées)

- ▶ Il existe déjà une classe des piles dans la bibliothèque, la classe `Stack` du package `java.util`
 - ▶ implémentée selon la technique des tableaux redimensionnés
- ▶ Mais une classe des piles de quoi ?
- ▶ C'est une classe `Stack<E>`, où `E` est n'importe quelle classe, et les objets de la classe `Stack<E>` sont des piles d'objets de la classe `E`

Classes génériques (paramétrées)

- ▶ Coder une pile d'entiers
- ▶ Coder une pile de réels
- ▶ Coder une pile de chaînes
- ▶ Coder une pile de piles
- ▶ Coder une pile de files
- ▶ Coder une pile de ...

C'est toujours la même chose et presque le même code

Classes génériques (paramétrées)

La documentation de java nous dit

- ▶ `Stack()` Creates an empty Stack. (le constructeur)
- ▶ `boolean empty()` Tests if this stack is empty.
- ▶ `E peek()` Looks at the object at the top of this stack without removing it from the stack.
- ▶ `E pop()` Removes the object at the top of this stack and returns that object as the value of this function.
- ▶ `E push(E item)` Pushes an item onto the top of this stack.

Ce n'est qu'un sous ensemble des méthodes applicables

Classes génériques (paramétrées)

Que fait le code ci-dessous ? (pièges)

```
import java.util.*;
class rpn {
    public static void main (String [] arg) {
        Stack<String> stack = new Stack<String> ();
        for (int k = 0; k < arg.length; k++)
            if (args[k].equals("+") ||
                args[k].equals("-") ||
                args[k].equals("*") ||
                args[k].equals("/")) {
                String i1 = stack.pop(), i2 = stack.pop();
                stack.push("(" + i2 + args[k] + i1 + ")");
            }
            else
                stack.push(args[k]);
        System.out.println(stack.pop());
    }
}
```

Classes génériques, le retour de la hp

```
import java.util.* ;
class Calc {
    public static void main (String [] arg) {
        Stack<Integer> stack = new Stack<Integer> ();
        ...
        int i1 = stack.pop().intValue(), i2 = stack.pop().intValue() ;
        stack.push(Integer.valueOf(i2+i1)) ;
        ...
    }
}
```

Classes génériques (paramétrées)

Comment coder une pile d'entiers ?

- ▶ Le paramètre E dans `Stack<E>` est une classe, → pas de piles de `int`
- ▶ La bibliothèque fournit une classe associée par type scalaire, par exemple `Integer` pour `int`
- ▶ Un objet `Integer` c'est un objet dont une variable d'instance (privée) contient un `int`
- ▶ Il existe deux méthodes pour convertir un scalaire `int` en un objet `Integer` et réciproquement, `valueOf` (logiquement statique) et `intValue` (logiquement dynamique).

Classes génériques, le retour de la hp

Mieux ! et magique ?

```
import java.util.* ;
class Calc {
    public static void main (String [] arg) {
        Stack<Integer> stack = new Stack<Integer> ();
        ...
        int i1 = stack.pop(), i2 = stack.pop() ;
        stack.push(i2+i1) ;
        ...
    }
}
```

Le compilateur Java sait insérer les appels aux méthodes de conversion automatiquement.

Aujourd'hui

Piles / Files (suite & fin)

Les génériques

La problématique de l'association

Les tables de hachage

Hacher en java



Recherche d'information dans un ensemble dynamique

- ▶ Associer des données (e.g., clients / commandes)
- ▶ Les retrouver
- ▶ Les manipuler (insertion, suppression, etc)

Exemple : Les pages blanches

- ▶ Associer un **nom** (i.e., une chaîne de caractères)
- ▶ à un **numéro de téléphone**

```
"Camille Abate" "04.91.63.50.30"
"Philippe Zito" "06.94.16.13.38"
"Laurie Zimmerman" "04.32.32.39.63"
"Ugo Zelaya" "04.70.61.50.56"
"Laure Zamor" "02.67.37.88.41"
"Oceane Archambault" "06.76.06.22.09"
...
```



Une idée simple : Les listes d'associations

- ▶ Créer une structure qui enregistre les références vers les objets à associer
 - ▶ ici deux String, le nom et le téléphone

```
class ListeAssoc {
    String nom;
    String tel;
    ListeAssoc suivant;
    ListeAssoc(String nom, String tel, ListeAssoc suivant) {
        this.nom = nom;
        this.tel = tel;
        this.suivant = suivant;
    }
}
```



Une idée simple : Les listes d'associations

```
String telDe(String nm) {
    if (nom.equals(nm))
        return tel;
    if (suivant == null)
        return null;
    return suivant.telDe(nm);
}
void changerTel(String nm, String nouvTel) {
    if (nom.equals(nm)) {
        tel = nouvTel;
        return;
    }
    if (suivant != null)
        suivant.changerTel(nm, nouvTel);
    throw new Error(nm + " pas enregistré");
}
```



Une idée simple : Les listes d'associations

```
ListeAssoc supprimer(String s) {
    if (nom.equals(s))
        return suivant;
    if (suivant ≠ null)
        suivant = suivant.supprimer(s);
    return this;
}
```



Une idée simple : Les listes d'associations

```
long tme = System.currentTimeMillis(); // pour mesurer le temps
ListeAssoc la = null;
for (;;) { // lecture d'un fichier (format "Alex Yu" "04.91.63.50.30")
    String nom = ST.ReadString(); // lire une chaîne
    if (nom.equals("ENDFILE")) break; // fin de la lecture du fichier
    la = new ListeAssoc(nom, ST.ReadString(), la);
    System.out.println("CHARGE EN " +
        (System.currentTimeMillis() - tme) + "ms");
    tme = System.currentTimeMillis();
    System.out.println("Tel de Laure Archambault : " +
        la.telDe("Laure Archambault") + " " +
        (System.currentTimeMillis() - tme) + "ms");
    tme = System.currentTimeMillis();
    System.out.println("Tel de Frederic Zito : " +
        la.telDe("Frederic Zito")+ " " +
        (System.currentTimeMillis() - tme) + "ms");
}
```



Les listes d'associations sur 248000 Enregistrements

```
java hash < PrenomNomTel.txt
CHARGÉ 1863ms
Tel de Laure Archambault : 02.32.61.45.56 40ms
Tel de Frederic Zito : 06.46.87.43.15 0ms
```

- ▶ Avantage de la simplicité
- ▶ Inconvénient majeur : Complexité algorithmique catastrophique
 - ▶ Parfois utile quand "l'univers" est petit
- ▶ Amélioration ? Trier les abonnés (tableau trié).
 - ▶ Une fausse bonne idée. Pourquoi ?



Tableau trié

- ▶ Insérer un nouvel élément : $O(n)$
 - ▶ Il faut déplacer "à droite" les $O(n)$ éléments déjà présents
- ▶ Enlever un élément prend : $O(n)$
 - ▶ Il faut déplacer "à gauche" les $O(n)$ éléments déjà présents
- ▶ Chercher un élément prend dans le pire des cas un temps $O(\log n)$, lorsqu'on utilise la **recherche binaire**



Recherche binaire dans un tableau trié

On cherche un élément de clef k :

- ▶ Comparer k avec la clef de l'élément en milieu de séquence
 - ▶ Si k est $<$ a cette clef, chercher l'élément de clef k dans la partie gauche
 - ▶ sinon, dans la partie droite
- ▶ Division par 2 à chaque itération $\rightarrow O(\log n)$

Rechercher la clef 9

- ▶ 1 4 5 6 9 12 13 15 19 21 22 24 29 30
- ▶ 1 4 5 6 9 12 13 15 19 21 22 24 29 30
- ▶ 1 4 5 6 9 12 13 15 19 21 22 24 29 30
- ▶ 1 4 5 6 9 12 13 15 19 21 22 24 29 30
- ▶ 1 4 5 6 9 12 13 15 19 21 22 24 29 30



Aujourd'hui

Piles / Files (suite & fin)

Les génériques

La problématique de l'association

Les tables de hachage

Hacher en java



Rechercher de l'information dans un ensemble dynamique

- ▶ Les éléments de l'ensemble sont constitués de plusieurs champs
 - ▶ L'un des champs, la clef \in ensemble totalement ordonné U , appelé l'univers des clefs
 - ▶ une clef = un entier ; l'univers = un intervalle d'entiers (muni de l'ordre naturel)
 - ▶ une clef = une chaîne ; l'univers = un ensemble de chaînes de longueur bornée (muni de l'ordre lexicographique)
- Pour avoir accès à toutes les informations, on recherche l'élément à l'aide de sa clef. Attention plusieurs éléments peuvent avoir la même clef.
- ▶ On veut aussi pouvoir insérer ou supprimer un élément.

Une telle structure = dictionnaire (nom générique ; plusieurs codages possibles, table de hachage, arbres binaires de recherche, "skiplists", etc.)



Un cas trivial : un petit univers

Si (1) l'univers est petit et (2) les clefs sont uniques (*i.e.*, des éléments distincts ont des clefs distinctes)

- ▶ Associer un entier à chaque clef (dans la suite on suppose que la clef = un entier)
- ▶ Construire un tableau $T[]$ tel que $T[c]$ est l'élément dont la clef est c



Hachage : la fonction de hachage

- ▶ Ranger dans un tableau T de taille m (\ll l'univers) toutes les clefs
- ▶ **Fonction de hachage** $h : U \rightarrow \{0, \dots, m - 1\}$
- ▶ Un élément de clef c est rangé dans $T[h(c)]$

Problèmes :

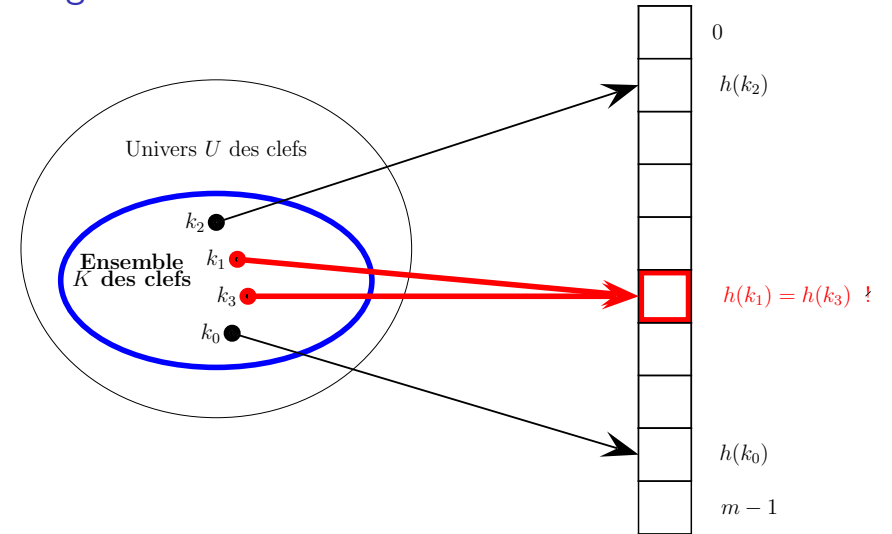
- ▶ Comment choisir m ? Que faire quand le tableau déborde?
- ▶ Comment construire h ?
- ▶ Plusieurs clefs sont hachées identiquement. Que faire?
 - ▶ T ne peut donc pas être un simple tableau

Hachage : les collisions

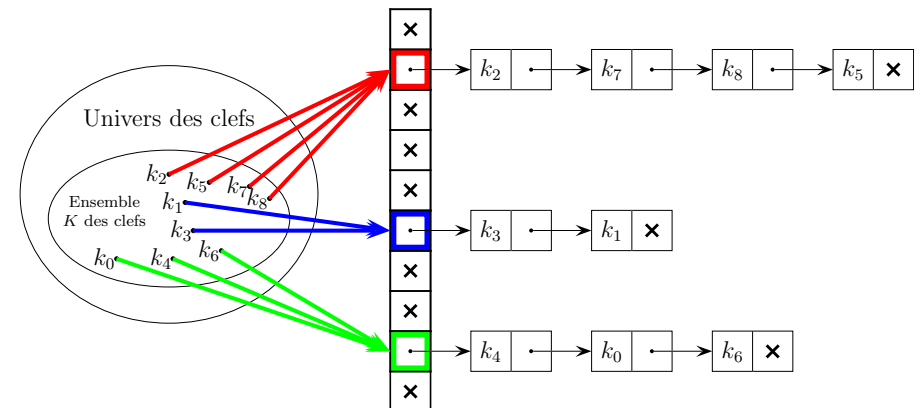
Deux solutions pour résoudre les collisions :

- ▶ **Le chaînage** : Les clefs qui ont la même valeur de hachage sont mises dans la même liste
- ▶ **L'adressage ouvert** : On cherche la première place libre dans le tableau après la valeur de hachage

Hachage : les collisions



Hachage : le chaînage



Hachage : le chaînage par l'exemple

- ▶ Une table de taille 12 et une (très mauvaise) fonction de hachage qui renvoie la somme des codes ASCII modulo 12.
- ▶ Insertion de Lea Abate 02.77.32.38.62, Sarah Zhao 05.46.69.77.66, Lea Eabat 05.77.89.98.00, Ugo McAdam 06.72.75.09.34, Quentin Dupaul 03.60.22.93.66, Bela Atea 05.77.39.38.21.
- ▶ Les codes :
 - ▶ Lea Abate → 3
 - ▶ Sarah Zhao → 5
 - ▶ Lea Eabat → 3
 - ▶ Ugo McAdam → 2
 - ▶ Quentin Dupaul → 11
 - ▶ Bela Atea → 3

Beaucoup de collisions (*i.e.*, de clefs rangées dans le même "bucket"). Pourquoi ?



Hachage : le chaînage

```
class TableHachageChaine {
    static final int m = 350000; // taille de la table
    ListeAssoc[] table;
    TableHachageChaine() {
        table = new ListeAssoc[m]; }
    static int codeHachage(String s) { // la mauvaise fonction de h
        int sum = 0;
        for (int i = 0; i < s.length(); i++) {
            sum += s.charAt(i); }
        return sum % m; }
    static void ajout(String nom, String tel, TableHachageChaine t) {
        int h = codeHachage(nom);
        t.table[h] = new ListeAssoc(nom, tel, t.table[h]); }
    static String telDe(String nom, TableHachageChaine t) {
        int h = codeHachage(nom);
        return ListeAssoc.telDe(nom, t.table[h]); }
}
```



Digression : ASCII

- ▶ American Standard Code for Information Interchange
- ▶ Les caractères sont codés sur 7 bits
 - ▶ 128 caractères possibles, de 0 à 127
- ▶ Les codes 0, 1, ... , 31 = caractères de contrôle (CR, BEL)
- ▶ Les codes 65 à 90 : LES MAJUSCULES
- ▶ Les codes 97 à 122 : les minuscules

$A \rightarrow 41, B \rightarrow 42, C \rightarrow 43, D \rightarrow 44, E \rightarrow 45, F \rightarrow 46, \dots, a \rightarrow 61,$
 $b \rightarrow 62, c \rightarrow 63, d \rightarrow 64, e \rightarrow 65, f \rightarrow 66, \dots,$



Hachage : le chaînage

- ▶ Insertion en $O(1)$
- ▶ Recherche d'un élément se fait
 - ▶ dans le pire des cas en $O(n)$
 - ▶ en moyenne en temps $O(1 + \alpha)$, où $\alpha = n/m$ (la charge) si les $h(k)$ sont uniformément distribués



Hachage : le chaînage

Que faire quand la table est pleine (ou presque) ?

- ▶ Doubler la capacité de la table
- ▶ Trouver une nouvelle fonction de hachage
- ▶ Recopier la vieille table dans la nouvelle
- ▶ Espérer que ca n'arrive pas trop souvent

Hachage : l'adressage ouvert

Une version simplifiée de ListeAssoc

```
class Couple {
    String nom;
    String tel;
    Couple(String nom, String tel) {
        this.nom = nom;
        this.tel = tel;
    }
}
```

Hachage : l'adressage ouvert

On cherche la première place libre dans le tableau après la valeur de hachage.

| | | | | | | | | | | | | |
|---------------------|---|---|---|---|---|---|---|---|---|---|----|----|
| Lea Abate (3) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Sarah Zhao (5) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Lea Eabat (3) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Ugo McAdam (2) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Quentin Dupaul (11) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Bela Atea (3) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

Hachage : l'adressage ouvert

```
class TableHachageAdrOuvert {
    static final int m = 550000; // taille de la table
    static long nbCollisions = 0;
    static long nbIn = 0;
    Couple[] table;
    TableHachageAdrOuvert() {
        table = new Couple[m];
    }
    static int codeHachage(String s) { // une mauvaise fonction de H
        int sum = 0;
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            sum += c;
        }
        return sum % m;
    }
}
```

Hachage : l'adressage ouvert

```

static void ajout(String nom, String tel, TableHachageAdrOuvert t) {
    int h = codeHachage(nom);
    for (int i = h; i != ((h-1) % m); i = (i+1) % m)
        if (t.table[i] == null) {
            t.table[i] = new Couple(nom, tel);
            return; }
    throw new Error("Table pleine");
}
static String telDe(String nom, TableHachageAdrOuvert t) {
    int h = codeHachage(nom);
    for (int i = h; i < ((h-1) % m); i = (i+1) % m) {
        if (t.table[i] == null)
            return null;
        if (t.table[i].nom.equals(nom))
            return t.table[i].tel; }
    return null;
}
}

```

Hachage : l'adressage ouvert

Attention aux phénomènes de regroupements.

- ▶ Même avec une distribution de $h(k)$ uniforme, la probabilité d'occupation d'une case n'est pas constante
 - ▶ Bien plus forte quand la case est immédiatement après un bloc de cases déjà utilisées
- ▶ Solution : **double hachage** .
 - ▶ Deux fonctions de hachage $h : U \rightarrow \{0, \dots, m-1\}$ et $h' : U \rightarrow \{0, \dots, r-1\}$.
 - ▶ Collision \rightarrow on choisit successivement les cases $h(k) + h'(k)$, $h(k) + 2h'(k)$, $h(k) + 3h'(k)$, etc.

Hachage : l'adressage ouvert

- ▶ Attention aux suppressions !
- ▶ Si on supprime un element, le test de recherche peut ne pas fonctionner !
- ▶ En cas de suppression, il faut tout re-hacher

Hachage : complexité de l'adressage ouvert

Quelle est la complexité en moyenne d'une recherche ?

- ▶ Piste : au moins 1 test et avec probabilité α ($\alpha = \frac{1}{m}$, la charge) 2 tests; avec probabilité α^2 tests, etc.
- ▶ Si la table est à moitié pleine \rightarrow 2 itérations (en moyenne)
- ▶ Si la table est à pleine à 90% \rightarrow 10 opérations (en moyenne)
- ▶ Qu'en est-il des autres opérations ?
- ▶ Règle de cuisine : table saturée si $\alpha > 40\%$

Les fonctions de hachage, les recettes

Une fonction de hachage associe un entier à un objet (une chaîne, un entier, ou un objet quelconque)

4 règles d'or (irréalisables)

1. La valeur de hachage ne dépend **que de l'objet d'entrée**
2. La fonction de hachage utilise **tous les champs** de l'objet d'entrée
3. La fonction de hachage distribue **uniformément** les données
4. La fonction de hachage génère des **valeurs très différentes** pour des objets **quasi identiques**

De plus, la fonction de hachage doit être rapide à calculer.

Les fonctions de hachage pour les entiers

- ▶ $h(k)$ le reste de la **division** de k par m : $h(k) = k \bmod m$
 - ▶ Attention si $m = 2^r$, $h(k)$ ne dépend que des r derniers bits de k .
 - ▶ Une bonne valeur pour m est un nombre premier (pas trop proche d'une puissance de 2). Peut se comporter très mal quand même.
- ▶ Variante de la **division** $h(k) = k(k + 3) \bmod m$ (Knuth)
- ▶ **Multiplication** . $m = 2^r$ et A un réel $h(k) = \lfloor m(Ak - \lfloor Ak \rfloor) \rfloor$.
 - ▶ Knuth propose le nombre d'or $A = \frac{1}{2}(\sqrt{5} - 1)$

Les fonctions de hachage, les recettes

Les 4 règles sont-elles vérifiées ?

```
static int codeHachage(String s) { // une mauvaise fonction de H
    int sum = 0;
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        sum += c;
    }
    return sum % m;
}
```

Mauvaise fonction de hachage : “Bela Atea” et “Lea Abate” donnent la même sum et donc ont le même H code.

Les fonctions de hachage pour les String

- ▶ Partir du code ASCII du caractère (entier sur 7 bits, compris entre 0 et 127)
- ▶ H code d'une chaîne de caractère = entier en base 128.
- ▶ Le H code de “java” est alors $106 \times 128^3 + 97 \times 128^2 + 118 \times 128 + 97 = 223902561$

Aujourd'hui

Piles / Files (suite & fin)

Les génériques

La problématique de l'association

Les tables de hachage

Hacher en java



En pratique, la `class` `Hashtable<K,V>`

- ▶ `<K,V>` est un couple d'objets (Key / Value)
- ▶ Exemple :

```
Hashtable <String, Integer> t = new Hashtable <String, Integer>
```
- ▶

```
Hashtable <MaListe, Integer> t = new Hashtable <MaListe, Integer>
```
- ▶ Attention (bis) : `Integer` et pas `Int`
- ▶ **public** `V put(K key, V value)` : ajoute un élément et sa clef à la table. retourne la valeur précédente associée à la clef dans la table (ou `null`)
- ▶ **public** `V get(K key)` retourne la valeur de la table dont la clef est `key`.
- ▶ **public** `remove(K key)` supprime l'élément de la table dont la clef est `key`.

Attention ! `put` et `get` ne sont pas `static`



En pratique, la classe `Hashtable`

- ▶ Dans le package "java.util"
- ▶ un package = ensemble de classes java
 - ▶ Utilisation d'une classe `Hashtable` d'un package `java.util` :
`java.util.Hashtable`
 - ▶ ou plus simplement `import java.util.*` au début du fichier puis `Hashtable`
- ▶ `Hashtable` implémente une structure associée à une donnée (un objet) associée une clef (un autre objet).
- ▶ "Comme" un tableau d'objets avec des indices non numériques.



En pratique, la `class` `Hashtable<K,V>`

```
class test {
    public static void main (String[] args) {
        Hashtable <String, Integer> numbers =
            new Hashtable<String, Integer> ();
        numbers.put("un", new Integer(1));
        numbers.put("deux", new Integer(2));
        numbers.put("quatre", new Integer(4));
        numbers.put("dix", new Integer(10));
        Integer n = (Integer)numbers.get(args[0]);
        System.out.println(args[0] + " = " + n);
    }
}
```



Object ?

- ▶ Tout ce qui n'est pas primitif est un Object
- ▶ A la création d'un objet (comme Hashtable ou Couple), on **hérite** des méthodes définies par la classe Object.
- ▶ Certaines de ces méthodes peuvent être redéfinies
 - ▶ Attention, suivre exactement la signature
- ▶ Un exemple déjà vu : toString

```
class Personne {  
    String nom; int age;  
    public String toString() {  
        return nom + " j'ai " + age + " ans";  
    }  
}
```

- ▶ D'autres méthodes utiles :
 - ▶ **public boolean** equals(Object o)
 - ▶ **public int** hashCode()