Recherche d'un élément

- Recherche par parcours séquentiel : on parcourt le tableau séquentiellement jusqu'à ce qu'on trouve l'élément

- *Exemple : recherche séquentielle dans un tableau de chaines*

```
// cette fonction renvoie vrai si x est présente dans tab, faux sinon
fonction avec retour booléen rechercheElement1(chaine[] tab, chaine x)
  entier i;
début
  i <- 0;
  tantque (i < tab.longueur) faire
      si (tab[i] = x) alors
          retourne VRAI;
      sinon
          i <- i + 1;
      finsi
  fintantque
  retourne FAUX;
fin
```


Recherche dans un tableau trié

- Si le tableau est trié par ordre croissant, on peut s'arrêter dès que la valeur rencontrée est plus grande que la valeur cherchée (ou plus petite si le tableau est trié par ordre décroissant)

```
// cette fonction renvoie vrai si x est présente dans tab, faux sinon
// le tableau tab est supposé trié par ordre croissant
fonction avec retour booléen rechercheElement2(chaine[] tab, chaine x)
  entier i;
début
  i <- 0;
  tantque (i < tab.longueur) faire
      si (tab[i] = x) alors
          retourne VRAI;
      sinon
          si (tab[i] > x) alors
              retourne FAUX;
          sinon
              i <- i + 1;
          finsi
      finsi
  fintantque
  retourne FAUX;
fin
```

Recherche multidimensionnelle

- Dans un tableau multidimensionnel, une double boucle est nécessaire pour rechercher un élément

```
// cette fonction renvoie vrai si x est présente dans tab, faux sinon
fonction avec retour booléen rechercheElement3(chaine[][] tab, chaine x)
  entier i,j;
début
  i <- 0;
  tantque (i < tab.longueur) faire
      j <- 0;
      tantque (j < tab[i].longueur) faire
          si (tab[i][j] = x) alors
              retourne VRAI;
          sinon
              j <- j + 1;
          finsi
      fintantque
      i <- i + 1;
  fintantque
  retourne FAUX;
fin
```


Recherche des occurences

- Si on suppose que la valeur recherchée peut apparaître plusieurs fois dans le tableau, on peut vouloir retourner le nombre d'occurences de la valeur

```
// cette fonction renvoie le nombre de fois où x est présente dans tab
fonction avec retour entier rechercheElement4(chaine[] tab, chaine x)
  entier i, nbOc;
début
  i <- 0;
  nbOc <- 0;
  tantque (i < tab.longueur) faire
      si (tab[i] == x) alors
          nbOc <- nbOc + 1;
      finsi
      i <- i + 1;
  fintantque
  retourne nbOc;
fin
```

Tableau associatif (1/3)

- Dans un tableau classique indicé par des entiers, les indices n'ont aucun rapport avec les valeurs des cases, il ne font qu'indiquer une position

- Mais si l'indice permet d'identifier la valeur de la case, la recherche de la présence de la valeur est facilitée : il suffit de vérifier que l'indice existe

- *Exemple :*
  - *les numéros d'étudiant ne permettent pas de retrouver un étudiant dont on connait le nom*
  - *un tableau d'étudiants indicé par les noms des étudiants permettrait de retrouver directement un étudiant dont on connait le nom*

- Il faut bien choisir l'élément qui identifie les valeurs : la clé

Tableau associatif (2/3)

- Les tableaux associatifs sont des tableaux dont les indices sont pris dans n'importe quel sous ensemble fini de valeurs d'un type donné (composé ou non)
  - les indices sont appelés les clés et chaque case est un couple clé-valeur

- *Exemple : tableau de valeurs du type Personne indicé par des chaines de caractères (les noms des personnes)*

prénom : Robert nom : Toto age : 29 métier : informaticien	prénom : Marcel nom : Tutu age : 31 métier : boulanger	prénom : Julie nom : Titi age : 28 métier : professeur
case « Toto »	**case « Tutu »**	**case « Titi »**

Tableau associatif (3/3)

- Problèmes posés par les tableaux associatifs :
  - il vaut mieux que les clés soient toutes différentes pour éviter les ambiguités
  - travailler avec de « gros » indices n'est pas efficace par rapport aux indices entiers
- Comment choisir entre tableau à indice entier et tableau associatif?
  - travailler avec des indices entiers, qui occupent peu de place en mémoire, mais obligent à parcourir le tableau pour trouver un élément
    - solution très efficace quant à l'occupation mémoire, la pire quand au temps de recherche
  - travailler avec des indices complexes (clés) qui identifient les valeurs des cases et permettent de trouver un élément en un seul accès au tableau
    - solution optimum quant au temps de recherche, mais très inefficace quant à l'occupation mémoire
    - problème d'ambiguités potentielles à gérer
- Compromis entre les deux : le hachage

Table de hachage (1/2)

- Principe d'une table de hachage :
  - les éléments sont identifiés par leurs clés mais placés dans un tableau indicé par des entiers
  - pour savoir dans quelle case placer un élément, on utilise une fonction de hachage qui à toute clé associe un entier
  - la place occupée en mémoire est faible comparée à un tableau associatif, mais l'accès à une case demande un petit calcul (moins gourmand généralement que la comparaison de grosses clés)

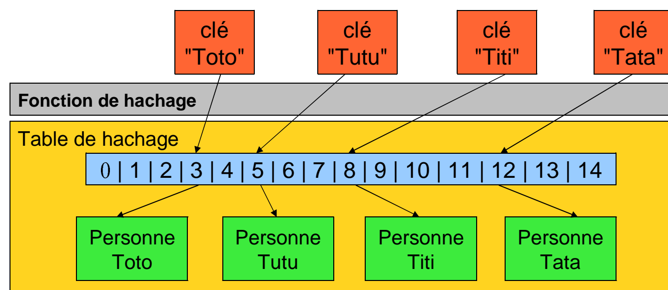


Table de hachage (2/2)

- *<u>Exemple</u> : on doit ranger dans un tableau N enregistrements de type Personne et on veut utiliser les noms des personnes comme clés (en supposant qu'il n'y a pas d'homonyme)*
  - *une manière de hacher les noms consiste à construire une représentation numérique des noms à l'aide des codes ASCII des caractères, puis à prendre pour indice ce nombre modulo N*
  - *pour chaque nom x, de longueur l, l'indice de l'enregistrement correspondant sera :*

$$h(x) = (\sum_{i=1..l} x[i]\, B^{l-i}) \bmod N$$

où B est un nombre choisi plus grand que N et premier généralement

  - *pour un tableau de 5 cases (N=5), on aura pour hachage des chaines "Toto" "Titi" et "Tutu" les valeurs 2, 0 et 4*

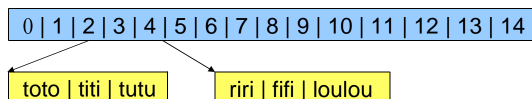
Fonction de hachage

- L'idéal est que la fonction de hachage soit injective :
  - deux clés différentes ont deux indices de hachage différents
  - il n'y a pas d'ambiguité lors d'un accès
  - en cryptographie, cette contrainte est forte (la fonction doit être “quasiment” injective)

- Si la fonction de hachage n'est pas injective, il peut y avoir deux clés différentes qui ont la même valeur de hachage
  - on parle de collision : l'accès à l'élément à partir de sa clé peut conduire à un autre élément
  - c'est le cas de beaucoup de fonctions de hachage car le nombre de clés possible est souvent plus grand que la quantité d'indices utilisés
  - le but étant d'accélérer l'accès aux éléments, plutôt que d'alourdir le calcul pour avoir une fonction injective, on complexifie la structure de données pour résoudre les collisions

Résolution des collisions

- Chainage externe (ou séparé ou ouvert) : chaque case du tableau contient non pas un élément mais la liste des éléments dont la clé a pour valeur de hachage l'indice de la case
  - problème : la recherche se fait sur deux niveaux et n'est plus immédiate



- Chainage fermé : on met les éléments ayant même valeur de hachage dans d'autres cases disponibles. Plusieurs stratégies :
  - sondage linéaire : on place les éléments dans des cases contigues, et pour les retrouver, on parcourt séquentiellement les cases contigues
  - hachage double : une deuxième fonction de hachage détermine la case où sera placer l'élément, et permet de le retrouver
  - sondage quadratique : on place les éléments dans des cases situées à une distance qui dépend quadratiquement de l'indice
  - ....

Table de hachage en Java

- EnJava, la classe **`Hashtable`** implémente une table de hachage.

```
// création d'une table de hachage avec clés de type chaine et valeurs
// de type Personne
Hashtable<String,Personne> ht = new Hashtable<String,Personne>();

// ajout d'une valeur dans la table : clé "Toto", valeur : une Personne
Personne toto = new Personne("Toto", "Robert", 24);
ht.put("Toto",toto);

// récupérer la valeur associée à la clé "Toto"
Personne p = ht.get("Toto");
```

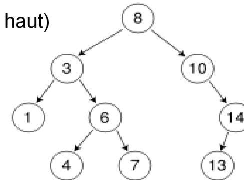
- Les clés utilisées doivent être des « objets » ayant un code de hachage. Ce code est retourné par la méthode **`hashCode()`**

- *<u>Exemple</u> : la classe* **`String`** *possède cette méthode*

```
String totoName = new String("Toto");
totoName.hashCode(); // retourne 2612822
int index = totoName.hashCode()%5; // index vaut 2
```

Arbres de recherche

- Les arbres de recherche (ou arbres binaires de recherche) sont des structures de données encore plus adaptées à la recherche
- Un arbre est constitué de noeuds reliés par des arcs, avec une structure arborescente :
  - tout noeud a un ensemble de fils sauf les noeuds feuilles
  - le noeud qui n'a pas de père est appelé racine
  - les arbres sont représentés à l'envers (racine en haut)



- Un arbre de recherche est tel que
  - tout noeud a deux fils, ou aucune si c'est une feuille
  - pour tout noeud non feuille, les noeuds de la sous-branche gauche sont tous plus petits que le noeud, ceux de la sous-branche droite sont tous plus grands

Recherche par dichotomie (1/3)

- La structure de données utilisée influe sur l'efficacité de la recherche d'un élément, mais la stratégie de recherche peut également jouer sur l'efficacité
- Principe de la recherche par dichotomie (du grec dikhotomia, division en deux parties égales) ou recherche binaire :
  - on sépare en deux parties égales les données d'entrée
  - on relance la recherche sur chacune des parties ou sur une seule
- Dans le cas général, si on doit relancer la recherche sur les deux parties, ça n'apporte rien en terme d'efficacité
- L'intérêt apparait si on peut ne traiter qu'une des sous parties
  - dans la recherche d'un élément dans un tableau, si le tableau est trié, on peut ne traiter qu'une des sous parties

Recherche par dichotomie (2/3)

- On travaille sur les cases d'indices compris entre i et j. Cet intervalle est divisé par 2 à chaque itération.

```
// cette fonction renvoie vrai si x est présente dans tab, faux sinon
// le tableau tab est supposé trié par ordre croissant
fonction avec retour booléen rechercheElement3(chaine[] tab, chaine x)
  entier i, j;
début
  i <- 0;
  j <- tab.longueur-1;
  tantque (i <= j) faire
      si (tab[(j+i)/2] = x) alors
          retourne VRAI;
      sinon
          si (tab[(j+i)/2] > x) alors
              j <- (j+i)/2 - 1;
          sinon
              i <- (j+i)/2 + 1;
          finsi
      finsi
  fintantque
  retourne FAUX;
fin
```


Recherche par dichotomie (3/3)

- *<u>Exemple</u> : on applique l'algorithme précédent sur un tableau trié d'entiers et on cherche si 490 est présent dans le tableau.*

4 | 17 | 25 | 26 | 45 | 45 | 87 | 102 | 234 | 237 | 490 | 1213 | 5681 | 5690 | 7012

234 | 237 | 490 | 1213 | 5681 | 5690 | 7012

234 | 237 | 490

490

Recherche par interpolation (1/2)

- *Remarque : si on cherche "zèbre" dans le dictionnaire, on ouvre le dictionnaire vers la fin, pas au milieu*

- Recherche par interpolation : on améliore la dichotomie en cherchant à couper le tableau non pas au milieu, mais à un endroit proche de la valeur cherchée
  - cela nécessite de pouvoir évaluer (par interpolation) la position probable de la valeur cherchée, et donc que la répartition des valeurs suive à peu près une fonction connue

- Si on suppose une répartition ordonnée et linéaire des valeurs dans le tableau, on peut interpoler linéairement la place de l'élément recherché
  - si tab est le tableau et e l'élément cherché, l'indice de e est interpolé par

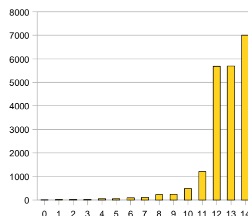
(e – tab[0]) * tab.longueur/(tab[tab.longueur – 1] – tab[0])

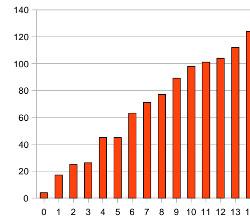
Recherche par interpolation (2/2)

- *Dans le tableau jaune, la formule d'interpolation linéaire donne 1 comme indice probable de 490 (l'indice réel est 11)*

- *Dans le tableau rouge, la formule d'interpolation linéaire donne 8 comme indice probable de 71 (l'indice réel est 8)*

4 | 17 | 25 | 26 | 45 | 45 | 87 | 102 | 234 | 237 | 490 | 1213 | 5681 | 5690 | 7012





4 | 17 | 25 | 26 | 45 | 45 | 63 | 71 | 77 | 89 | 98 | 101 | 104 | 112 | 124