

TC Informatique

PC N° 5
7 Décembre 2000

François Sillion

Recherche et hachage **Listes, files et piles...**

<http://w3.edu.polytechnique.fr/informatique>

Recherche en table

- Représentation d'un ensemble d'objets
- Opération de base : recherche
- Autres opérations : insertion, suppression, énumération, fusion...
- Applications : dictionnaire, index, bases de données...

- Recherche séquentielle

- Tableau

```
for ( int i=0; i < a.length ; ++i )  
    if ( a[i] == val ) return true;
```

- Liste

```
while ( (t != null) && (t.val != val) )  
    t = t.suiv;
```

```
return ( t != null );
```

- Complexité linéaire

Recherche dichotomique

- Travail dans un tableau, supposé trié

```
public static boolean cherche
    ( int a[], int val, int g, int d ){
    if ( g < d-1 ) {
        int m = (g+d)/2;
        if ( val < a[m] ){
            return cherche(a, val, g, m);
        } else {
            return cherche(a, val, m, d);
        }
    } else {
        return ( a[g] == val );
    }
}
```

Appel de `cherche(a, val, 0, a.length)`

Recherche dichotomique par interpolation

- Meilleur choix du point de subdivision

```
public static boolean cherche
    ( int a[], int val, int g, int d ){
    if ( g < d-1 ) {
        //int m = (g+d)/2;
        int m = g + (d-g)*(val-a[g])/(a[d]-a[g]);
        if ( val < a[m] ){
            return cherche(a, val, g, m);
        } else {
            return cherche(a, val, m, d);
        }
    } else {
        return ( a[g] == val );
    }
}
```

- Complexité $\log \log n + 1$ (clés aléatoires)

Principe du hachage

- Recherche d'éléments quelconques (chaines de caractères...)
- Cas où on ne peut pas indexer dans une table directement par la clé de recherche
 - Grand nombre de valeurs possibles
 - Clé complexe
- On suppose qu'on dispose d'une fonction h qui transforme les objets recherchés en un (petit) nombre entier.
- On range l'élément c en position $h(c)$
- Compromis entre temps de calcul et espace
 - Il faut une table assez grande (plus grande que le nombre d'éléments à ranger)
 - La recherche est meilleure que linéaire

Recherche avec hachage

- Implicitement on suppose h injective

```
public static boolean cherche
    ( Chose a[], Chose truc ){
    int key = h(truc);
    if ( a[key] == null ) {
        return false;
    } else if ( comparer( a[key], truc ) ) {
        return true;
    } else
        ...
}
```

Hachage avec tableau fini

- Tableau de taille M (M>N!)
- Collisions : plusieurs clés « tombent » au même endroit
- Première solution : adressage ouvert
 - En cas de collision, on fait une recherche linéaire à partir de l'entrée donnée par la fonction de hachage :

```
public static boolean cherche
    ( Chose a[], Chose truc ){
    int key = h(truc) % a.length;
    while ( a[key] != null ) {
        if ( comparer( a[key], truc ) ) {
            return true;
        } else {
            key = (key+1) % a.length;
        }
    }
    return false;
}
```

Adressage ouvert

- Insertion
- Suppression ?
- Complexité : si $\alpha = N/M$, nombre d'opérations en moyenne
 - Recherche avec succès, $\frac{1}{2}(1 + \frac{1}{1-\alpha})$
 - Recherche avec échec $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$
- Pour $\alpha = 2/3$, cela donne 2 et 5
- Pour $\alpha = 90\%$, 5 et 50

Hachage multiple

- Pour éviter les regroupements, on remplace

```
key = (key+1) % a.length;
```

par

```
key = (key+ u(truc) ) % a.length;  
// deuxième fonction de hachage
```

- Il faut que u et M soient premiers entre eux. Si M est premier, $u < M$ suffit.
- Si h et u sont « indépendantes » le nombre moyen d'opérations est

– Recherche avec succès, $\frac{-\log(1-\alpha)}{\alpha}$

– Recherche avec échec $\frac{1}{1-\alpha}$

- Pour $\alpha = 80\%$, 2 et 5
- Pour $\alpha = 99\%$, 5 et 100

Hachage avec listes

- Chaque case du tableau contient une liste d'objets
- Si la répartition est bonne, chaque liste a en moyenne N/M éléments

```
static void ajouter  
    ( ListeChose[] a, Chose truc ){  
    int key = h(truc) % a.length;  
    a[key] = new ListeChose( truc, a[key] );  
}
```

```
static void cherche  
    ( ListeChose[] a, Chose truc ){  
    int key = h(truc) % a.length;  
    return cherche( truc, a[key] );  
    // surcharge : fonction de même nom  
    // travaillant sur une liste...  
}
```

Fonctions de hachage

- Répartition « uniforme » dans le tableau

```
public static int h ( String s ) {  
    int result = 0;  
    for ( int i=0; i < s.length ; ++i ) {  
        result = 256*result+s.charAt(i);  
    }  
    return result;  
}
```

- Choix de 256 ou 128 comme "base", pour permettre des multiplications rapides
- Résultat modulo M, typiquement premier
- Fonction Java `x.hashCode()` des `java.lang.object`

File

- Utilisation d'une liste gardée

```
class Liste {  
    int val;  
    Liste suiv;  
    Liste( int v, Liste s ) {... }  
}  
class File {  
    Liste premier; // cellule de garde en tête  
    Liste dernier;  
    File( Liste p, Liste d ) {...}  
}  
static File CreeFile() {  
    Liste l = new Liste( 0, null );  
    return new File( l, l );  
}  
static void Ajoute( int val, File f ) {  
    f.dernier.suiv = new Liste( val, null );  
    f.dernier = f.dernier.suiv;  
}  
static int Retire(File f ) { // file non vide!  
    Liste l = f.premier.suiv;  
    f.premier.suiv = l.suiv;  
    return l.val;  
}  
static boolean FileVide( File f ) {  
    return f.premier == f.dernier;  
}
```

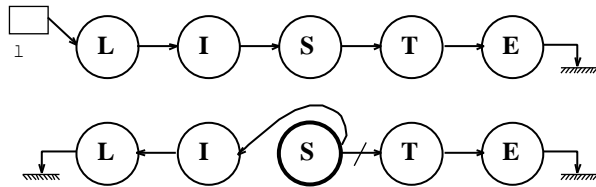
Exercices

- Réaliser une pile dans un tableau
- Recopie d'une liste dans une autre (réursive et non-réursive)
- Concaténation de deux listes
 - Avec modification de la première (nConc)
 - Sans modification (append)
- Fusion de deux listes triées
- Renversement d'une liste (réursive ou non-réursive)

Concaténation de listes

```
static Liste append( Liste a, Liste b ) {
    if ( a == null )
        return b;
    else
        return new Liste(a.val,append(a.suiv,b));
}
static Liste nConc( Liste a, Liste b ) {
    if ( a == null )
        return b;
    else {
        Liste c = a;
        while ( c.suivant != null )
            c = c.suivant;
        c.suivant = b;
        return a;
    }
}
}
Ou récursivement :
static Liste nConc( Liste a, Liste b ) {
    if ( a == null )
        return b;
    else {
        a.suivant = nConc( a.suivant, b );
        return a;
    }
}
}
```

Renversement de listes



```
static Liste nReverse( Liste a ) {  
    Liste b = null;  
    while ( a != null ) {  
        Liste c = a.suivant;  
        a.suivant = b;  
        b = a;  
        a = c;  
    }  
    return b;  
}
```

Version récursive (quadratique !):

```
static Liste reverse( Liste a ) {  
    if ( a == null ) return a;  
    else  
        return append(  
            reverse( a.suivant ),  
            new Liste( a.val, null) );  
}
```