INF2010 – ASD

Hashage et
tables de dispersement

Plan

- Fonctions de dispersement
- Résolution des collisions
  - Chaînage
  - Sondage
  - Dispersement double
- Rehash
- Tables de dispersement

Plan

- Fonctions de dispersement
- Résolution des collisions
  - Chaînage
  - Sondage
  - Dispersement double
- Rehash
- Tables de dispersement

Fonctions de dispersement

Problématique :

- La recherche dans une base de données est une tâche importante qui doit être effectuée efficacement et rapidement.
- Si le temps exigé pour effectuer la recherche était illimité, alors une simple recherche séquentielle serait adéquate; l'utilisation de la mémoire serait alors optimale.
- Si la mémoire disponible était infinie, un simple adressage direct à l'aide de la clé donnerait instantanément la position de l'élément recherché.
- La meilleure solution se trouve entre ces deux solutions extrêmes.

Caractéristiques

- Complexité
  - Recherche avec succès
  - Recherche sans succès
  - Insertions
  - Élimination

Fonctions de dispersement

Définitions :

- **Clé**: Variable choisie pour la recherche.
- **Fonction de dispersement**: Fonction qui est associée à une clé.
- **H(clé)**: Donne un emplacement d'espace mémoire.
- **Collision**: Lorsque deux clés différentes correspondent au même espace mémoire (tel qu'obtenu par une fonction de dispersement), il y a collision.
- **Enregistrement synonyme**: Se dit de deux éléments dont les clés sont en collision.
- **Facteur de compression**:
  fc =nombre d'éléments/espace mémoire disponible

Fonctions de dispersement

Qualités d'une fonction de dispersement :

Une bonne fonction de dispersement doit:

- Distribuer le plus uniformément possible les clés sur l'espace mémoire disponible afin de réduire le nombre de collisions;
- Se calculer facilement et rapidement.

Exemples de fonctions de dispersement

1° H(clé) = clé mod N, où N = taille de la table

Cette fonction est simple mais elle est très utilisée;
Le résultat est assurément dans l'intervalle [0, N-1];
Choix intéressant: poser N = P, le plus petit nombre premier tel que P $\geq$ N.

2° Troncature: H(clé) = troncature de la clé

Exemple:
Soit un fichier contenant 1000 positions réservées, la clé utilisée est le numéro d'assurance sociale. Par troncation, la clé 252 313 345 devient:
H(252 313 345) = 345

Exemples de fonctions de dispersement

3° Le recoupement

Encore une fois, avec un numéro d'assurance sociale et 1000 positions réservées, la fonction de dispersement pourrait être:

H(clé) = (somme par groupe de trois) mod 1000

Par exemple:

252 313 450 -> 252 + 313 + 450 = 1005

et 1005 mod 1000 = 5

4° La transformation de base

La clé est utilisée comme si elle était représentée dans une autre base que la base 10, puis reconvertie en base 10.

Par exemple, 1234 en base 13 donnerait 2578 en base 10.

Enfin, le modulo 1000 est appliqué pour obtenir 578

Exemples de fonctions de dispersement

5° Les clés alphabétiques

Cette situation se présente lorsque la clé est une chaîne de caractères alphanumériques.

Par exemple: F37 BA2

Les lettres peuvent être traitées selon leur code ASCII ou leur rang alphabétique.

Exemples de fonctions de dispersement

5° Les clés alphabétiques (suite)

```
public static int hash(String key, int tableSize)
{
   int hashVal = 0;

   for(int i = 0; i < key.length( ); i++ )
      hashVal = 37 * hashVal + key.charAt( i );

   hashVal %= tableSize;

   if( hashVal < 0 )
      hashVal += tableSize;

   return hashVal;
}
```

Plan

- Fonctions de dispersement
- Résolution des collisions
  - Chaînage
  - Sondage
  - Dispersement double
- Rehash
- Tables de dispersement

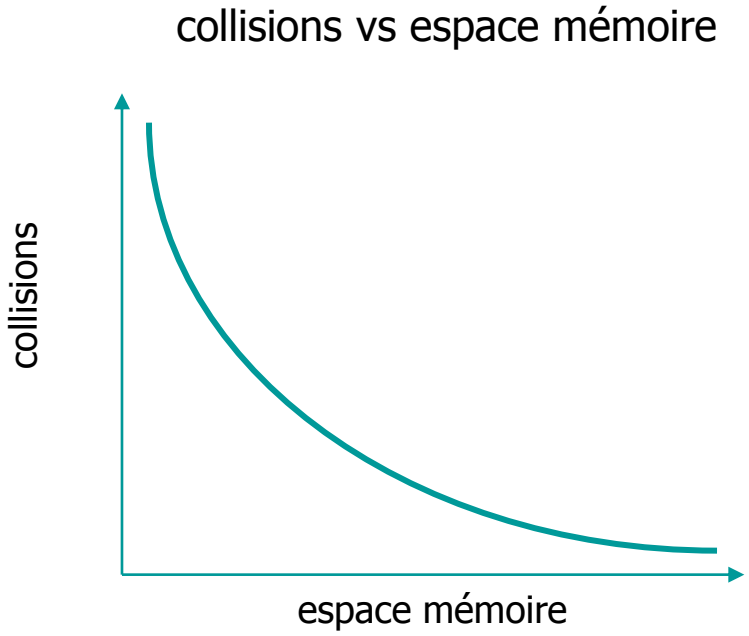
Résolution de collisions

Le nombre de collisions augmente généralement très vite lorsque fc (le facteur de compression) devient supérieur à 90%.

Il est impensable d'éliminer toutes les collisions. De toute façon, ceci se ferait au détriment de l'espace mémoire requis.

Que faire alors? Il faut relocaliser les enregistrements synonymes.

Résolution de collisions



Plan

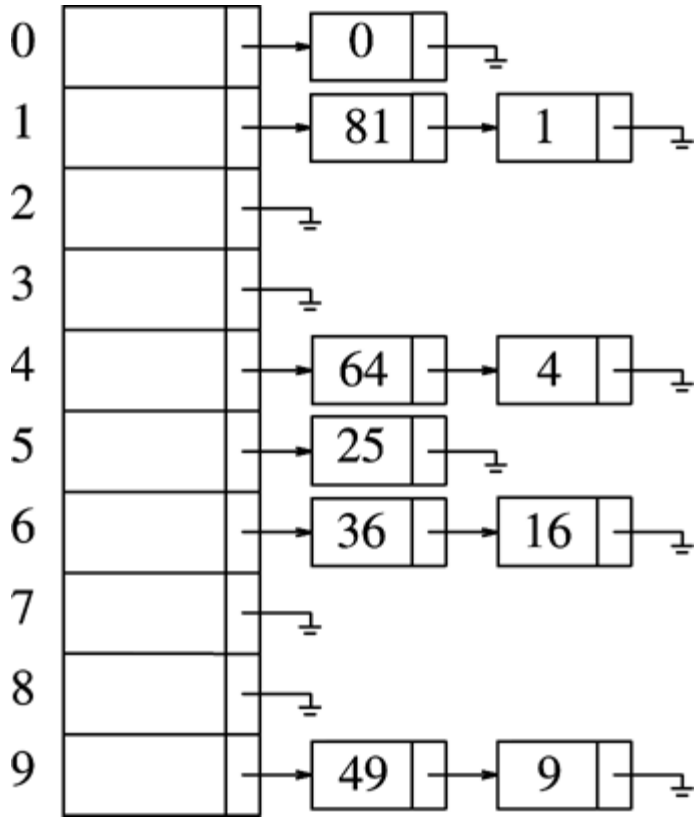
- Fonctions de dispersement
- Résolution des collisions
  - Chaînage
  - Traitement linéaire des collisions
  - Traitement quadratique des collisions
  - Dispersement double
- Rehash
- Tables de dispersement

Résolution des collisions par chaînage

Idée générale :

- Lorsqu'il y a collision, l'enregistrement synonyme est lié à l'emplacement mémoire donné par la fonction de dispersement grâce à une liste chaînée débutant à cet endroit.

Schema d'implantation

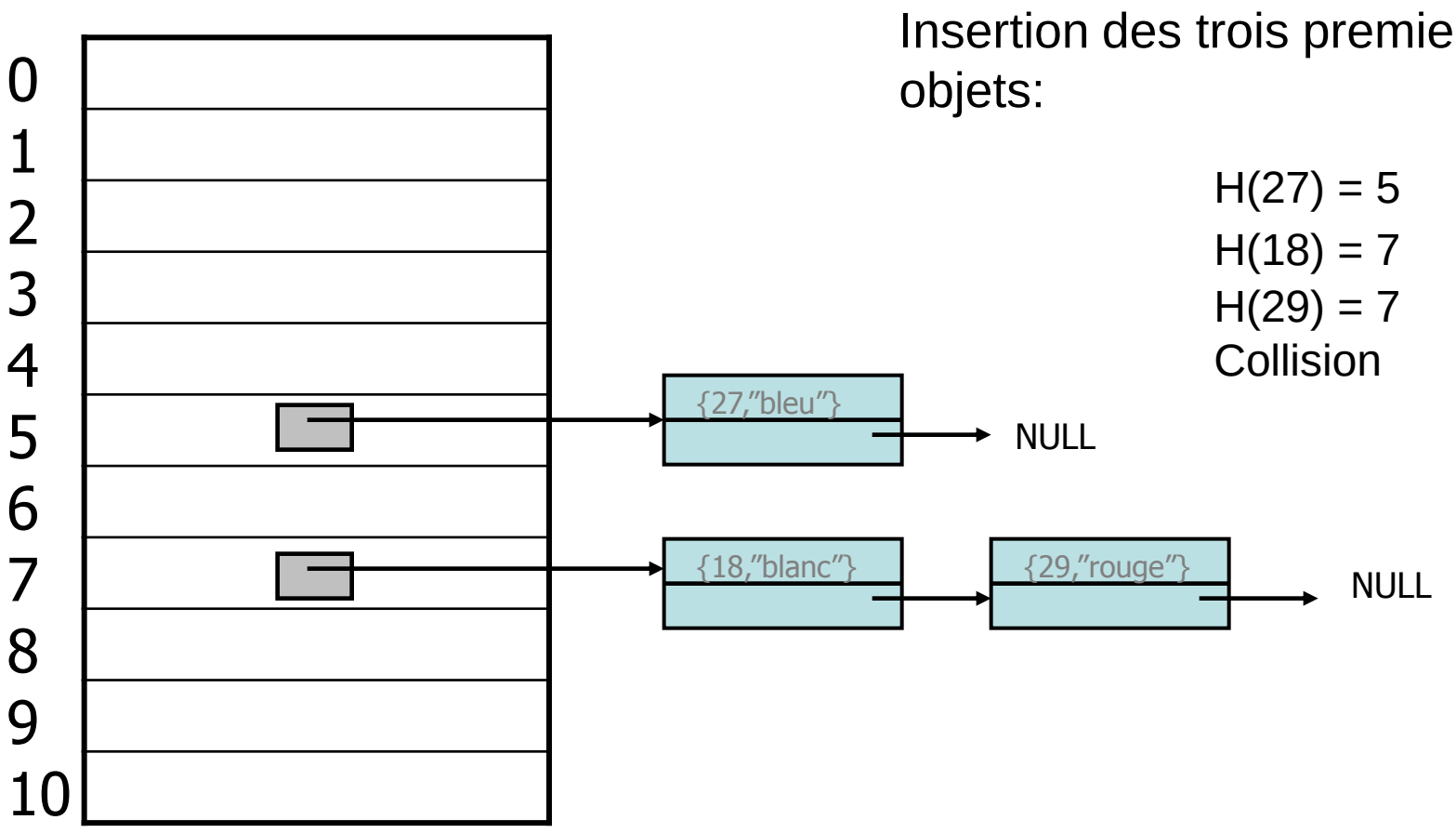


Résolution des collisions par liste chaînée

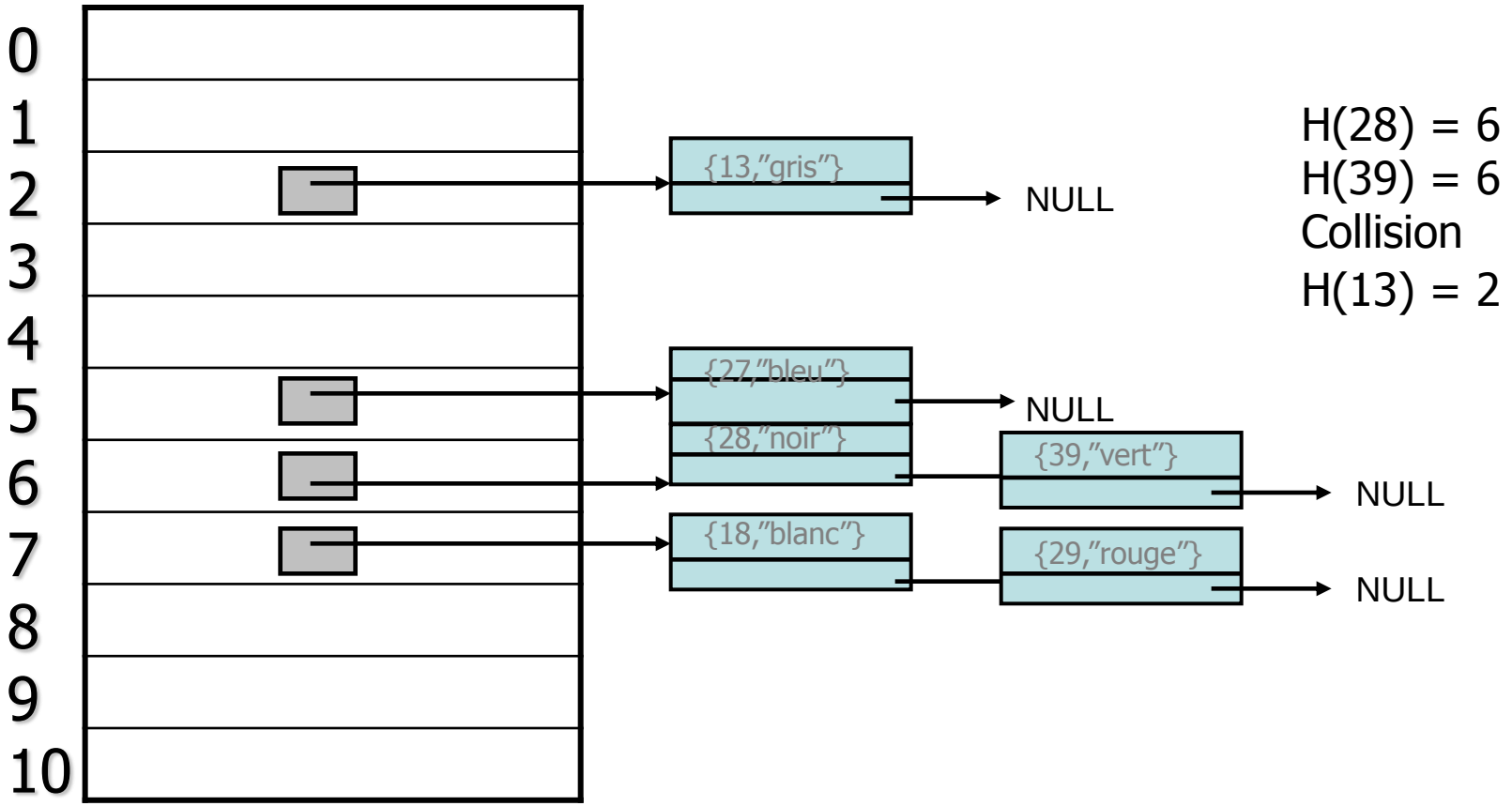
- Soit un tableau de N=11 éléments, la clé est un attribut de la classe
- H(clé) = clé % N
- Les objets à insérer sont les suivants:

  {27, "bleu"}, {18, "blanc"}, {29, "rouge"}, {28, "noir"},{39, "vert"}, {13, "gris"},{16, "mauve"}, {42, "cyan"}, {17, "rose"}.

Résolution des collisions par liste chaînée (suite)



Résolution des collisions par liste chaînée



Résolution des collisions par liste chaînée

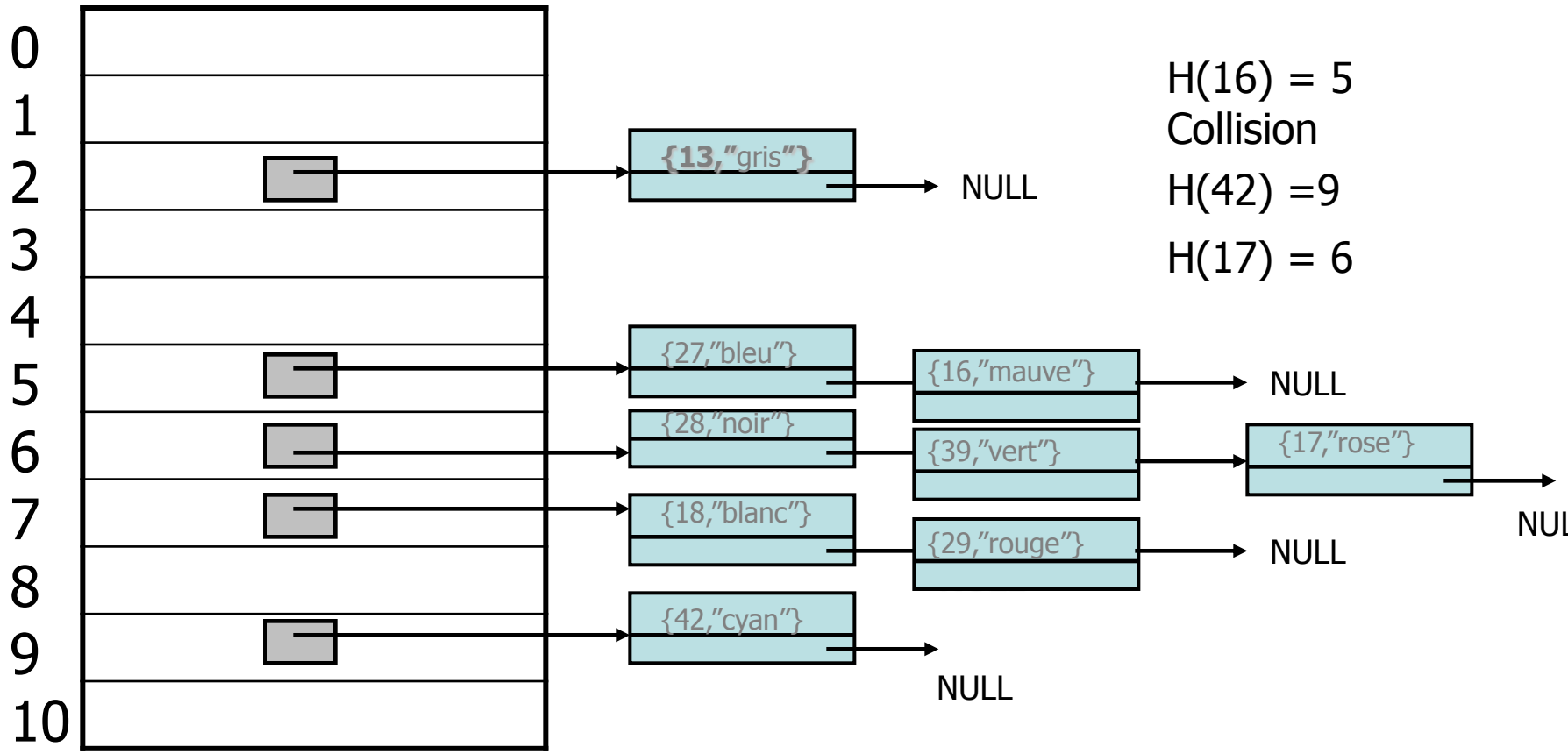


Fig 5.6

```
public class SeparateChainingHashTable<AnyType>
{
   public SeparateChainingHashTable( )
   { /* Figure 5.9 */ }
   public SeparateChainingHashTable( int size )
   { /* Figure 5.9 */ }
   public void insert( AnyType x )
   { /* Figure 5.10 */ }
   public void remove( AnyType x )
   { /* Figure 5.10 */ }
   public boolean contains( AnyType x )
   { /* Figure 5.10 */ }
   public void makeEmpty( )
   { /* Figure 5.9 */ }
   private static final int DEFAULT_TABLE_SIZE = 101;
   private List<AnyType> [ ] theLists;
   private int currentSize;
   private void rehash( )
   { /* Figure 5.22 */ }
   private int myhash( AnyType x )
   { /* Figure 5.7 */ }
   private static int nextPrime( int n )
   { /* See online code */ }
   private static boolean isPrime( int n )
   { /* See online code */ }
}
```

Fig 5.7

```
private int myhash( AnyType x )
{
    int hashVal = x.hashCode( );

    hashVal %= theLists.length;

    if( hashVal < 0 )
        hashVal += theLists.length;

    return hashVal;
}
```

Fig 5.8

```
public class Employee
{
   public boolean equals(Object rhs)
   {
      return rhs instanceof Employee &&
         name.equals( ((Employee)rhs).name );
   }

   public int hashCode( )
   {
      return name.hashCode( );
   }

   private String name;
   private double salary;
   private int seniority;
   // Additional fields and methods
}
```

Fig 5.9

```
public SeparateChainingHashTable()
{
   this( DEFAULT_TABLE_SIZE );
}

public SeparateChainingHashTable(int size)
{
   theLists = new LinkedList[ nextPrime( size ) ];

   for(int i = 0; i < theLists.length; i++ )
      theLists[ i ] = new LinkedList<AnyType>();
}
```

Fig 5.9 (suite)

```
public void makeEmpty( )
{
    for(int i = 0; i < theLists.length; i++ )
        theLists[ i ].clear( );

    theSize = 0;
}
```

Fig 5.10

```
public boolean contains( AnyType x )
{
   List<AnyType> whichList = theLists[ myhash( x ) ];
   return whichList.contains( x );
}
```

Fig 5.10 (suite)

```
public void insert( AnyType x )
{
   List<AnyType> whichList = theLists[ myhash( x ) ];

   if( !whichList.contains( x ) )
   {
      whichList.add( x );

      // Rehash; see Section 5.5
      if( ++currentSize > theLists.length )
         rehash( );
   }
}
```

Fig 5.10 (suite)

```
public void remove( AnyType x )
{
   List<AnyType> whichList = theLists[ myhash( x ) ];

   if( whichList.contains( x ) )
   {
      whichList.remove( x );
      currentSize--;
   }
}
```

Plan

- Fonctions de dispersement
- Résolution des collisions
  - Chaînage
  - Sondage
  - Dispersement double
- Rehash
- Tables de dispersement

Dispersement par sondage

- $h_i(x) = (hash(x) + f(i)) \bmod m$
  - $f(i) = i$
    - Sondage linéaire
  - $f(i) = i^2$
    - Sondage quadratique
  - $f(i) = c_1 i + c_2 i^2$
    - Sondage quadratique

Dispersement par sondage

Idée générale:

- Lorsqu'il y a collision, l'enregistrement est relocalisé au prochain emplacement libre

- Algorithme :

```
Insertion

-Appliquer la fonction de dispersement
-Si la position est vide
 |  -Insérer le nouvel enregistrement
-Sinon
 |  - Rechercher la 1e position libre à partir de la  position calculée
 |  - Si la fin du tableau est atteinte
 |   |    -Poursuivre la recherche au début
```

Sondage linéaire (exemple 1)

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

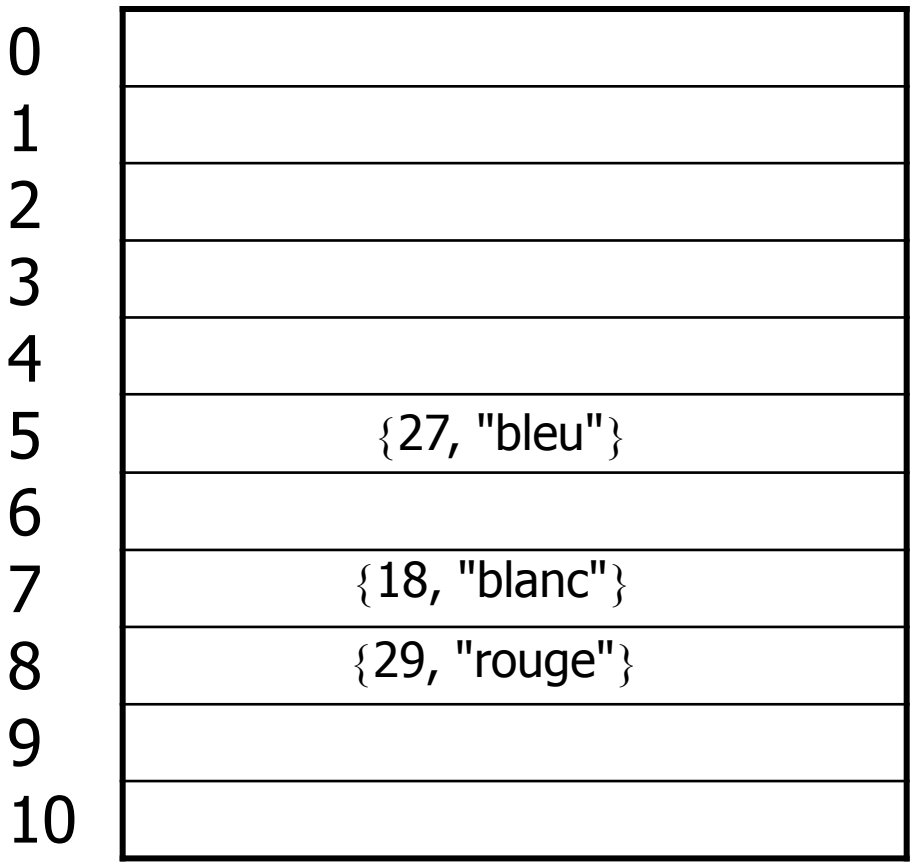
Sondage linéaire (exemple 2)

- Soit un tableau de N=11 éléments (représentant une base de données pouvant contenir 11 enregistrements), avec comme fonction de dispersement:
  - H(clé) = clé % N où la clé est un attribut de la classe

- Les objets à insérer sont les suivants:

  {27, "bleu"}, {18, "blanc"}, {29, "rouge"}, {28, "noir"},{39,"vert"} , {13, "gris"},{16, "mauve"}, {42, "cyan"}, {17, "rose"}.

Sondage linéaire (exemple 2)



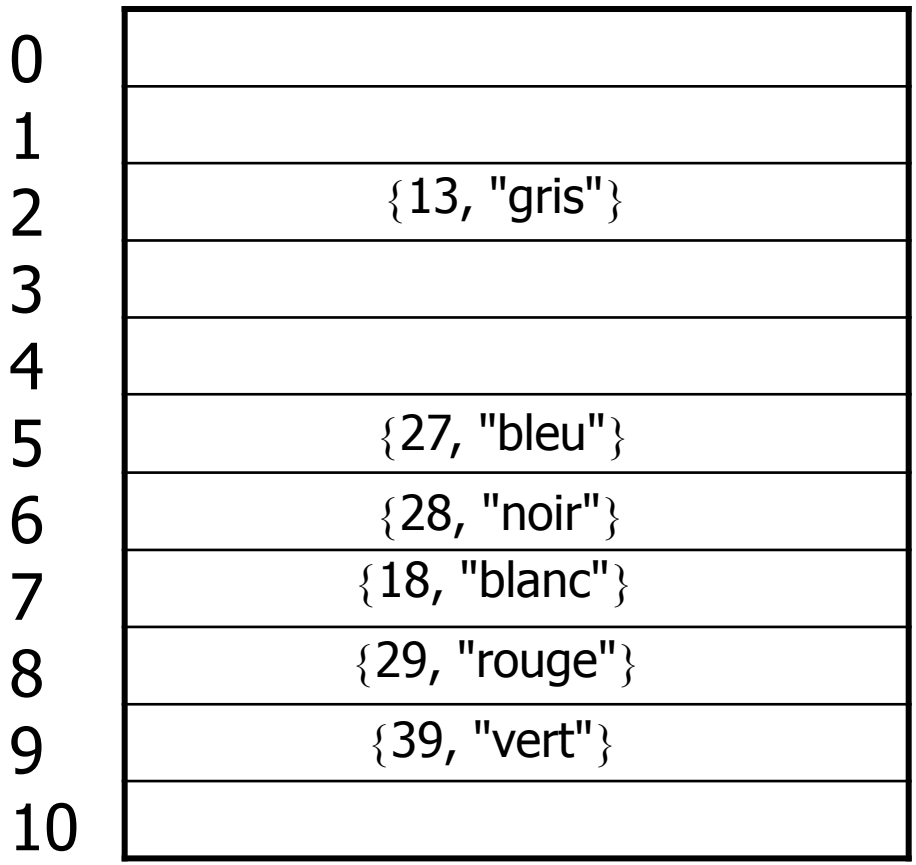
Insertion des trois premiers objets:
{27, "bleu"},
{18, "blanc"},
{29, "rouge"}

H(27) = 5
Insertion à l'index 5

H(18) = 7
Insertion à l'index 7

H(29) = 7
On insère 29 à la
Première position libre

Sondage linéaire (exemple 2)



Insertion des trois premiers objets:

{28, "noir"},

{39, "vert"},

{13, "gris"},

H(28) = 6
Insertion à l'index 6

H(39) = 6
On insère 39 à la Première position libre

H(13) = 2
Insertion à l'index 2

Sondage linéaire (exemple 2)

0	{42, "cyan"}
1	{17, "rose"}
2	{13, "gris"}
3	
4	
5	{27, "bleu"}
6	{28, "noir"}
7	{18, "blanc"}
8	{29, "rouge"}
9	{39, "vert"}
10	{16, "mauve"}

Insertion des trois premiers objets:
{16, "mauve"},
{42, "cyan"},
{17, "rose"}

H(16) = 5
Collision

H(42) = 9
Collision

H(17) = 6
Collision

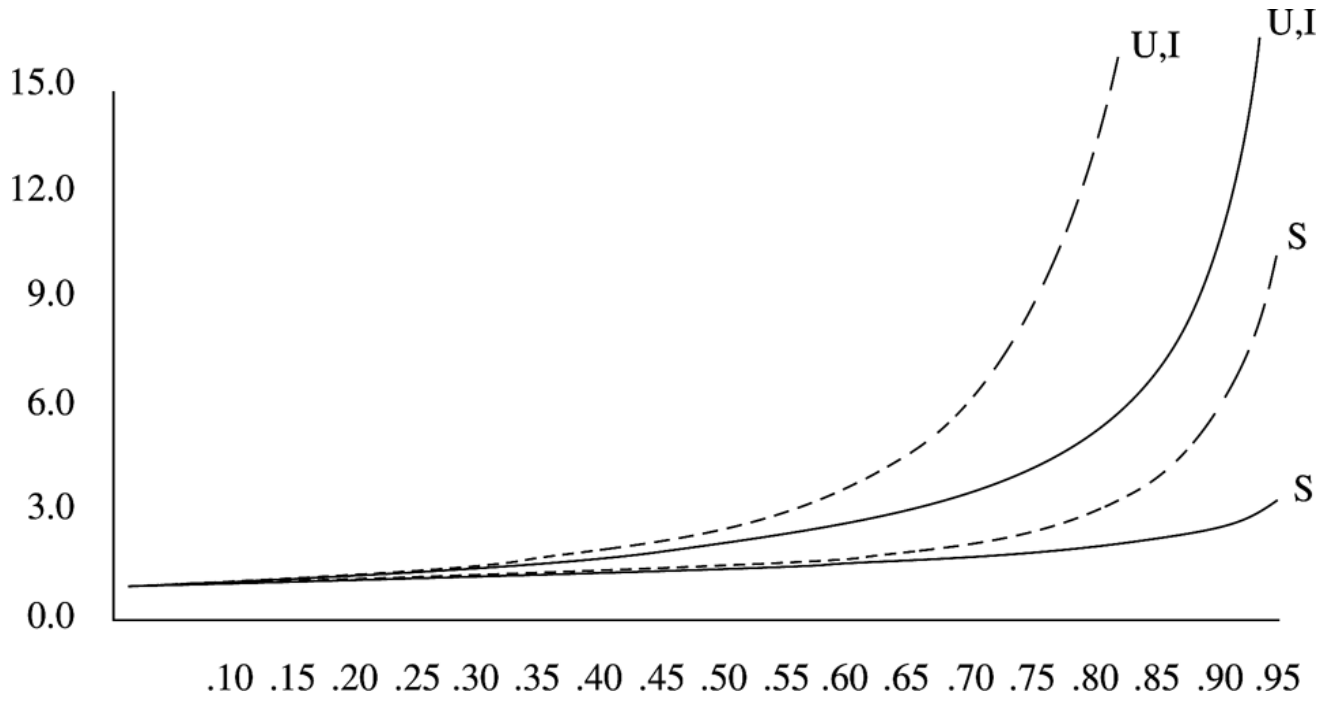
Sondage quadratique (exemple)

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Propriété

- Il est toujours possible d'insérer un élément dans une table de dispersement à débordement progressif par dispersement **quadratique**, si
  - la table a une taille *p* (*p* est premier) et
  - la table est au moins à moitié vide

Performance de sondage



- - - linéaire
----- quadratique

Implantation (fig 5.14)

```
public class QuadraticProbingHashTable<AnyType>
{
   public QuadraticProbingHashTable( )
   { /* Figure 5.15 */ }
   public QuadraticProbingHashTable( int size )
   { /* Figure 5.15 */ }
   public void makeEmpty( )
   { /* Figure 5.15 */ }
   public boolean contains( AnyType x )
   { /* Figure 5.16 */ }
   public void insert( AnyType x )
   { /* Figure 5.17 */ }
   public void remove( AnyType x )
   { /* Figure 5.17 */ }
```

Implantation (fig 5.14 - suite)

```
private static class HashEntry<AnyType>
{
   public AnyType  element;  // the element
   public boolean isActive;  // false if marked deleted
   public HashEntry( AnyType e ) { this( e, true ); }
   public HashEntry( AnyType e, boolean i )
   { element  = e; isActive = i; }
}

private static final int DEFAULT_TABLE_SIZE = 11;
private HashEntry<AnyType> [] array;  // The array of elements
private int currentSize;              // The number of occupied cells
```

Implantation (fig 5.14 - suite)

```
    private void allocateArray( int arraySize )
    { /* Figure 5.15 */ }
    private boolean isActive( int currentPos )
    { /* Figure 5.16 */ }
    private int findPos( AnyType x )
    { /* Figure 5.16 */ }
    private void rehash( )
    { /* Figure 5.22 */ }
    private int myhash( AnyType x )
    { /* See online code */ }
    private static int nextPrime( int n )
    { /* See online code */ }
    private static boolean isPrime( int n )
    { /* See online code */ }

}// Fin de la classe QuadraticProbingHashTable
```

Implantation (fig 5.15)

```
public QuadraticProbingHashTable( )
{
    this( DEFAULT_TABLE_SIZE );
}

public QuadraticProbingHashTable( int size )
{
    allocateArray( size );
    makeEmpty( );
}
```

Implantation (fig 5.15 - suite)

```
public void makeEmpty( )
{
   currentSize = 0;
   for(int i = 0; i < array.length; i++ )
      array[ i ] = null;
}

private void allocateArray( int arraySize )
{
   array = new HashEntry[ arraySize ];
}
```

Implantation (fig 5.16)

```
public boolean contains( AnyType x )
{
    int currentPos = findPos( x );
    return isActive( currentPos );
}
```

Implantation (fig 5.16 - suite)

```
private int findPos( AnyType x )
{
    int offset = 1;
    int currentPos = myhash( x );

    while( array[ currentPos ] != null &&
        !array[ currentPos ].element.equals( x ) )
    {
        currentPos += offset;  // Compute ith probe
        offset += 2;

        if( currentPos >= array.length )
            currentPos -= array.length;
    }

    return currentPos;

}
```

Implantation (fig 5.16 - suite)

```
private boolean isActive( int currentPos )
{
   return array[ currentPos ] != null && array[ currentPos ].isActive;
}
```

Implantation (fig 5.17)

```
public void insert( AnyType x )
{
    // Insert x as active
    int currentPos = findPos( x );

    if( isActive( currentPos ) )
        return;
    array[ currentPos ] = new HashEntry<AnyType>( x, true );

    // Rehash; see Section 5.5
    if( ++currentSize > array.length / 2 )
        rehash( );
}
```

Implantation (fig 5.17 - suite)

```
public void remove( AnyType x )
{
   int currentPos = findPos( x );

   if( isActive( currentPos ) )
      array[ currentPos ].isActive = false;
}
```

Plan

- Fonctions de dispersement
- Résolution des collisions
  - Chaînage
  - Sondage
  - Dispersement double
- Rehash
- Tables de dispersement

Double fonction de dispersement

Idée générale :

- Pour éviter la profusion d'amas dispersés, une variante consiste à faire varier l'incrément à chaque passage entre les enregistrements. Par exemple, l'incrément peut être une fonction de la clé à insérer.

Les tables de dispersement

Algorithme :

- H1(clé) nous donne une position dans la table;
- À cette position dans la table, on relève la position correspondante dans le fichier de données;
- Lecture de l'enregistrement dans le fichier de données;
- **Si** la clé dans le fichier diffère de la clé recherchée:
  |- Calculer H2(clé) pour obtenir l'incrément dans la table.

Double fonction de dispersement

- Une seconde fonction de hachage H2(x) est utilisée pour calculer le décalage en cas de collision
- On essait les positions H2(x)%N, 2*H2(x)%N, 3*H2(x)%N, et ainsi de suite jusqu'à la première position libre trouvée
- Exemples:
  - H2(X) = R – (X % R), où R < N et est un nombre premier

Exemple (fig 5.18)

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

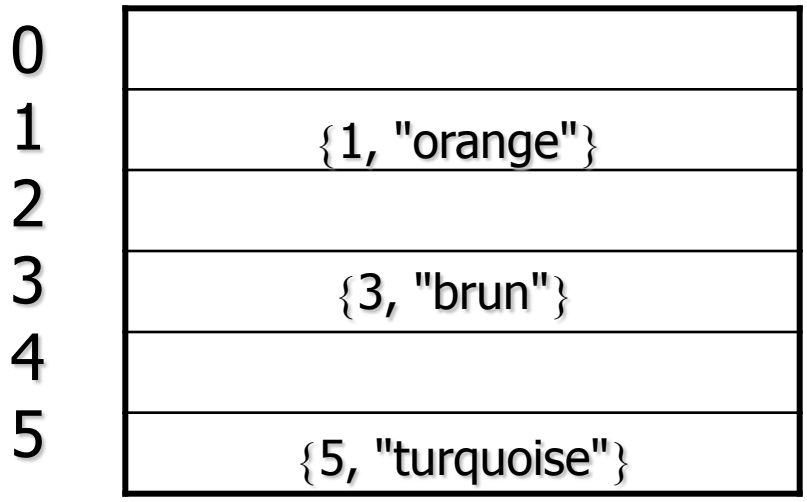
Double fonction de dispersement

- Un incrément variable permet de briser les amas;
- L'économie en espace mémoire pour les liens diminue la performance par rapport à l'utilisation du chaînage.

Double fonction de dispersement

Exemple de cycle ; Soit P=N=6 (cas où P n'est pas premier)

Les éléments à insérer sont: {1, "orange"}, {3, "brun"}, {5, "turquoise"} et {13, "violet"}



H1(1) = 1%6 = 1

H1(3) = 3%6 = 3

H1(5) = 5%6 = 5

H1(13) = 13%6 = 1

Collision

H2(13) = (13/6)%6 = 2

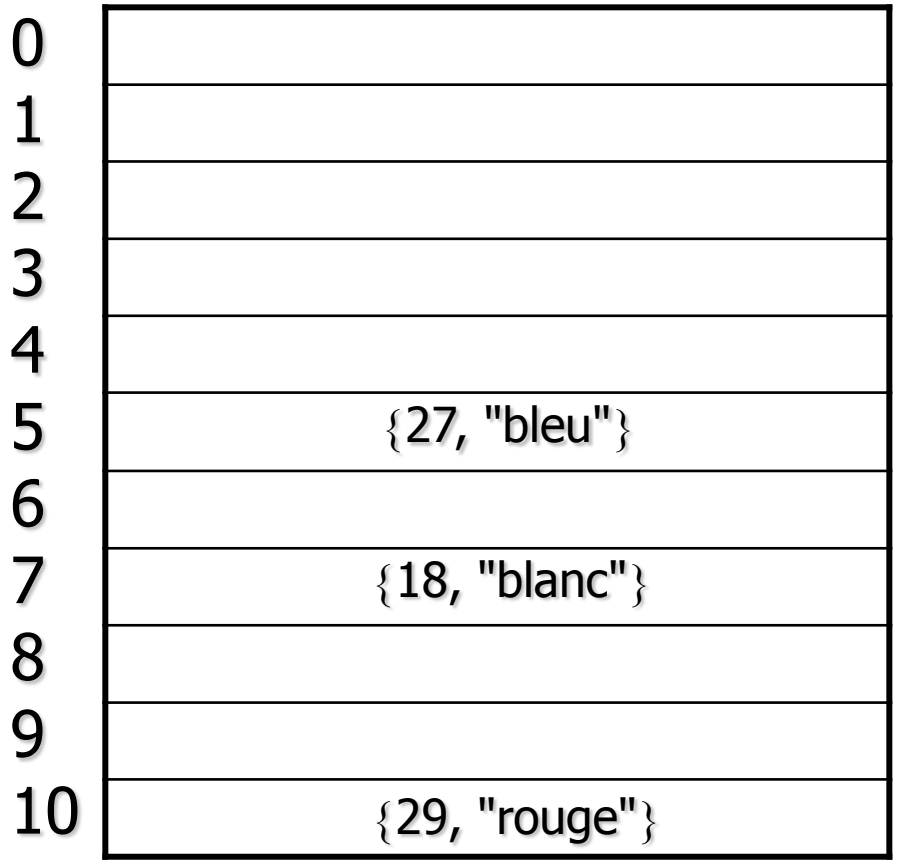
Nouvel index est = 2+1 = 3%6=3

Collision. . .

Dans ce cas, il y a seulement 3 enregistrements occupés et aucun espace libre ne peut être localisé.

Double fonction de dispersement

Soit P=N=11 (cas où P est un nb premier)



Les trois premiers éléments à insérer sont:
{27,"bleu"},
{18,"blanc"},
{29,"rouge"},

H1(27) = 27%11 = 5

H1(18) = 7

H1(29) = 7
Collision

H2(29) = (29/11)+1 = 3

Nouvelle position
7+3 = 10%11 = 10

Double fonction de dispersement

Soit P=N=11 (cas où P est un nb premier)

0	
1	
2	{13, "gris"}
3	{39, "vert"}
4	
5	{27, "bleu"}
6	{28, "noir"}
7	{18, "blanc"}
8	
9	
10	{29, "rouge"}

Les trois éléments suivants à insérer sont:
{28,"noir"},
{39,"vert"},
{13,"gris"},

H1(28) = 28%11 = 6

H1(39) = 6
Collision

H2(39) = (39/11)+1 = 4
Nouvelle position
(6+4)%11 = 10
Collision

Nouvelle position
(10+4)%11 = 3

H1(13) = 2

Double fonction de dispersement

Soit P=N=11 (cas où P est un nb premier)

0	{42, "cyan"}
1	
2	{13, "gris"}
3	{39, "vert"}
4	
5	{27, "bleu"}
6	{28, "gris"}
7	{18, "blanc"}
8	{17, "rose"}
9	{16, "mauve"}
10	{29, "rouge"}

Les trois derniers éléments à insérer sont:
{16,"mauve"},
{42,"cyan"},
{17,"rose"},

Double fonction de dispersement

Objet	Nombre d'cellules visitées pour localiser l'enregistrement	
{27, "bleu"}	1	
{18, "blanc"}	1	
{29, "rouge"}	2	
{28, "noir"}	1	
{39, "vert"}	3	
{13, "gris"}	1	
{16, "mauve"}	3	
{42, "cyan"}	7	
{17, "rose"}	2	
Nombre moyen:	**21/9 = 2.3**	

Plan

- Fonctions de dispersement
- Résolution des collisions
  - Chaînage
  - Sondage
  - Dispersement double
- Rehash
- Tables de dispersement

Rehash

- Nous avons déjà vu que lorsque le taux d'occupation de la table atteint 50%, le nombre de collisions augmente sensiblement
- Solution: augmenter la table aussitôt que l'on atteint ce taux d'occupation
- Il faut maintenir une taille de table qui soit un nombre premier.

Implantation (rehash)

```
private void rehash( )
{
   HashEntry<AnyType> [] oldArray = array;

   // Doubler (au moins) l'espace mémoire
   allocateArray( nextPrime( 2 * oldArray.length ) );
   currentSize = 0;

   // Recopier
   for(int i = 0; i < oldArray.length; i++ )
      if( oldArray[ i ] != null && oldArray[ i ].isActive )
         insert( oldArray[ i ].element );
}
```

Implantation (nextPrime)

```
private static int nextPrime(int n )
{
    if( n % 2 == 0 )
        n++;

    for( ; !isPrime( n ); n += 2 )
        ;

    return n;
}
```

Implantation (isPrime)

```
private static boolean isPrime(int n )
{
   if( n == 2 || n == 3 )
      return true;

   if( n == 1 || n % 2 == 0 )
      return false;

   for(int i = 3; i * i <= n; i += 2 )
      if( n % i == 0 )
         return false;

   return true;
}
```

Plan

- Fonctions de dispersement
- Résolution des collisions
  - Chaînage
  - Sondage
  - Dispersement double
- Rehash
- Tables de dispersement

Les tables de dispersement

Problématique :

- Il n'est pas toujours pratique, à moins d'avoir un support externe dédié, de réserver l'espace pour N éléments en laissant des enregistrements vides pour appliquer une fonction de dispersement. En pratique, tous les enregistrements d'un fichier sur support externe devraient être occupés.

Les tables de dispersement

Idée générale :

- Deux fichiers sont utilisés.
  - Le premier contient la table de dispersion ("hashing table") qui donne la relation entre le résultat de la fonction de dispersement et la position des enregistrements dans le fichier. Ce fichier est appelé un index.
  - Le deuxième renferme la base de données, ordonnée de façon logique.

Les tables de dispersement

- La recherche s'effectue donc dans la table. L'accès aux enregistrements du fichier de données ne se fait que pour accéder à l'information et vérifier les collisions.

Remarque :

- L'usage de fichiers d'index ne se limite pas à l'adressage indirect. Il est possible d'effectuer une recherche séquentielle ou binaire dans un fichier index ordonné selon le champ clé. Dans certaines applications, ces fichiers peuvent être hiérarchiques pour en faciliter le chargement en mémoire principale.

Tables de dispersement

Fichier séquentiel
{27, "bleu"}
{18, "blanc"}
{29, "rouge"}
{28, "noir"}
{39, "vert"}
{13, "gris"}
{16, "mauve"}
{42, "cyan"}
{17, "rose"}

fichier d'index Double fonction de dispersement	
7	0
	1
5	2
4	3
	4
0	5
3	6
1	7
8	8
6	9
2	10