
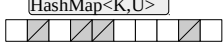


JAVA DONNEE


- 1.1 **HashSet<T>**
- 
- ensemble (non ordonné)
 - pas de doublons
 - fonction : discriminer un ensemble d'objets

add(x)	Insérer x	0(1)
contains(x)	X est-il présent	0(1)
remove(x)	Supprimer x	0(1)


- 1.2 **HashMap<K,U>**
- 
- table d'association (clé, valeur)
 - pas de doublons sur les clés
 - fonction : retrouver rapidement la valeur associée à une clé

put(k,v)	Insérer k,v	0(1)
get(k)	Récupère v associé à k	0(1)
remove(k)	Supprimer k,v	0(1)

- get(k) renvoie null si la clé est absente

- 2 Tableaux
- 2.1 **T[]**
- 
- Tableau de taille fixe
 - on peut laisser des trous
 - peut servir de HasMap avec des clés entières et consécutives
 - Types primitifs autorisés

New T[size]	Créer un tableau	0(n)
New T[] {a,b,c}	Créer et initialiser un tab	0(n)
T[i]	Accès en lecture/ Écriture à la case i	0(1)
Arrays.toString(t)	Conversion en chaîne	0(n)

- 2.2 **ArrayList<T>**
- 
- tableau dynamique (de taille variable)
 - les éléments sont ajoutés à la fin
 - suppression déconseillée

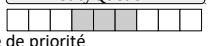
New ArrayList<T> (capacity)		0(1)
add(x)	Ajouter x à la fin	0(1)
get(i)	Accès en lecture à la case n°i	0(1)
toString()	Conversion en chaîne	0(n)

- 3 Queues
- 3.1 **ArrayDeque<> comme file**
- 
- extraction dans l'ordre d'insertion
 - FIFO : First In First Out

add(x)	Insérer x en queue	0(1)
remove()	Supprimer l'élément	0(1)
contains(x)	X est-il présent	0(n)
remove(x)	Sup un exemp de x	0(n)

- 3 Queues
- 3.1 **ArrayDeque<> comme pile**
- 
- extraction dans l'ordre inverse de l'insertion
 - LIFO : Last In First Out

push(x)	Insérer x en tête	0(1)
pop()	Supprimer l'élément ne tête	0(1)
contains(x)	X est-il présent	0(n)
remove(x)	Sup un exemp de x	0(n)

- 3.3 **PriorityQueue<T>**
- 
- file de priorité
 - pour extraire les éléments dans un ordre déterminé par une fonction de comparaison
 - l'ordre d'itération est indéterminé !

add(x)	Insérer x	0(log n)
element(x)	Récupérer l'élément min	0(n)
contains(x)	X est-il présent	0(1)
remove(x)	Sup un exemp de x	0(log n)

4 Itération sur une collection

HashSet, ArrayDeque, tableau :

```
for (Object o : collection) {
    // traitement
}
```

HashMap :

```
for (Map.Entry<K,V> e : map.entrySet()) {
    // traitement
}
```

PriorityQueue :

```
while (!pq.isEmpty()) {
    // traitement
    pq.remove();
}
```

- ↳ **Panne (alimentation électrique)**
 - en plein milieu d'une opération (renommage, copie...)
 - risque de laisser le système de fichiers dans un état incohérent
- ↳ **Perte de données stockées (usure...)**

- Transaction**
- But : rendre atomique et durable un ensemble d'opération
 - Tout ou rien : Validation (commit) ou annulations (Roll back)

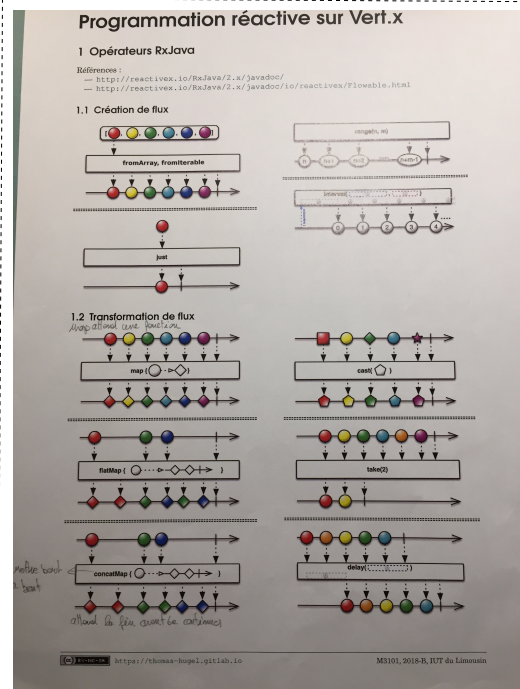
- Algorithme de journalisation**
- Ecrire sur un journal (log) l'ensemble de l'opération à regrouper
 - Sauvegarder le journal
 - **Ecrire <<Validation>> sur le journal**
 - Effectuer les opérations écrites
 - Effacer du journal tout ce qui a été effectué et validé

- Détection d'erreurs : sommes de contrôle**
- ↳ Fonction de hachage cryptographique :
 - produit peu de collisions
 - est difficile à inverser (garantit l'authenticité d'un fichier échangé sur le réseau)
 - La famille de fonctions SHA (Secure Hash Algorithm)

- Cadet correcteur d'erreur**
- On ajoute des bits de telle sorte qu'on puisse détecter et corriger une petite erreur
 - => Algèbre linéaire

- Utilisation de disques en série**
- RAID : Redundant Array of inexpensive Disks**
- Que se passe-t-il si un seul disque a un problème ?
 - détection + correction possible
 - Pourquoi répartir les blocs de contrôle sur tous les disques ?
 - Pour équilibrer la charge (blocs les plus sollicités)

- Comment les gens Web stock les données**
- Protection matérielle de chaque disque : somme de contrôle + codes correcteurs
 - Disques en série (RAID) avec plusieurs redondances
 - Vérifications de sommes de contrôle au niveau du système de fichiers :
 - chaque bloc de données peut stocker avec lui sa propre somme de contrôle + son numéro d'inoeud.
 - répartition dans des centres informatiques géographiquement distants.



- Les fichiers**
- Cahier des charges d'un système de fichiers
- Fiabilité
 - grand capacité (par fichier + total)
 - Rapidité
 - chemin d'accès nommés par utilisateur
 - Permissions

- Volume**
- Entité physique où est déployé le système de fichiers
 - Exemples
 - disque dur
 - clé USB
 - chaque secteur d'une partition d'un disque
 - Pour lire une clé il faut la monter et la démonter à chaque utilisation, c'est à dire crée un lien symbolique

- Disque Magnétique**
- Le temps de rotation est de 5 ms + 1 à 20 ms pour trouver la bonne piste.

- Efficacité**
- Temps d'accès en mémoire est de
 - 100 ms, i.e 100 000 fois moins que 10 ms

- Conséquences pour les systèmes**
- mutualiser les lecteurs / écritures (tampons)
 - maximiser la localité spatiale d'un même fichier
 - * voir des fichiers d'un même répertoire.

- Politique d'ordonnement**
- L'ordre du disque pour aller chercher les données demandées est en :
- FIFO
 - SPTF (shortest Positionnong Time First), i.e au plus près avec le risque d'affamer certaines requêtes. **Ascenseur** : du centre vers le bord et inverse

- Disque électronique**
- SSD (Solid State Drive)
- Ensemble de transistors (comme la mémoire vive)
 - Pour écrire sur un bloc, il faut d'abord l'effacer
 - au lieu d'effacer un bloc, on écrit ailleurs
 - Lecture / écriture environ 0,1 ms
 - 100 fois plus rapide que dans un disque magnétique
 - 1000 fois moins rapide que dans la mémoire
 - Usure/corruption des données
 - Une cellule peut être effacée entre 10³

- Et 10⁶ fois l'endurance d'un disque de 1 To et 1 Po.
- Code correcteur d'erreurs et espace de réserve

- Disque Magnétique VS Disque électronique**
- Plus Rapide, Moins gourmand en énergie électrique, moins d'espace total transit limité (usure) et plus cher.

- Inoeud**
- Métadonnées du fichier stockés dans un inoeud.
- Chaque inoeud a un numéro l'identifiant
 - Les inoeuds sont stockés dans la table des inoeuds
 - L'inoeud contient des pointeurs vers les blocs de données

- Le chemin d'une inoeuds se trouve dans les repertoire**
- ls -l

- Liens Physique et symboliques**
- Lien physiques** : association (nom de fichier, numéro d'inoeud)
 - Lien symboliques** : association (nom de fichier, Chemin)

- Organisation des données**
- Ntfs, ext 4, HFS+

- La table de fichiers principale (MFT : Master File Table)
- Contient inoeuds :

- Un fichier minuscule est Stocké directement dans l'inoeuds
- pour un fichier plus gros, seuls les pointeurs vers les segments (base + borne) sont stockés
- Si le nombre de segments est trop grand, plusieurs inoeuds sont chaînés

- Allocation d'un segment**
- Politiques First fit et Best fit**
- First fit** : on cherche le premier espace suffisamment grand
 - Best fit** : on cherche l'espace disponible le plus petit qui convienne.

Exemple : création d'un répertoire

- Mkdir IUT/M3101**
- Le système d'exploitation
- analyse la ligne de commande entrée et la découpe en (commande + arguments)
 - Cherche la commande dans le PATH
 - Vérifier que l'utilisateur a le droit d'exécuter cette commande
 - Découpe le chemin selon la barre oblique
 - cherche dans le répertoire courant le numéro d'inoeud à IUT
- Vérifie que :**
- IUT est bien un répertoire ;
 - l'utilisateur a le droit d'accéder à IUT et le droit d'y écrire
 - vérifier que M3101 n'existe pas déjà dans IUT
 - allouer un numéro d'inoeud pour M3101 parmi les numéros disponibles
 - Alloue un bloc de contenu pour M3101 sur le disque

Exemple : création d'un répertoire

- Mkdir IUT/M3101**
- ↳ initialise l'inoeud de M3101 avec :
- le propriétaire et le groupe propriétaire ;
 - la date ;
 - les permissions (fournies par le masque°)
 - le type de fichier
 - le compteur liens à 2
 - un pointeur vers le bloc de contenu
- ↳ initialise le contenu de M3101 avec :
- un lien physique vers lui-même
 - un autre lien vers son parent (IUT)
 - écrit dans IUT un nouveau lien (M3101, numéro d'inoeuds))

FIFO

```
public class FifoCache<K, V> extends Cache<K,V> {
    private ArrayDeque<K> cles;
    private Hashap<K,V> valeurs;
    public FifoCache(int bufferSize) {
        super(bufferSize);
        this.cles = new ArrayDeque<K>();
        this.valeurs = new HashMap<K,V>();
    }

    @Override
    protected int size() {
        return cles.size();
    }

    @Override
    protected V getValue(K key) {
        return valeurs.get(key);
    }

    @Override
    protected void removeEntry() {
        K key = cles.remove();
        valeurs.remove(key);
    }

    @Override
    protected void insertEntry(K key, V value) {
        cles.add(key);
        valeurs.put(key, value);
    }

    @Override
    protected void reorder(K key) {
        //On ne reorder jamais en FIFO
    }

    @Override
    protected K[] toArray() {
        return (K[]) cles.toArray();
    }
}
```

MRU

```
public class MruCache<K,V> extends Cache<K,V> {
    private ArrayDeque<K> cles;
    private HashMap<K,V> valeurs;

    public MruCache(int bufferSize) {
        super(bufferSize);
        this.cles = new ArrayDeque<K>();
        this.valeurs = new HashMap<K,V>();
    }

    @Override
    protected int size() {
        return cles.size();
    }

    @Override
    protected V getValue(K key) {
        return valeurs.get(key);
    }

    @Override
    protected void removeEntry() {
        K key = cles.pop();
        valeurs.remove(key);
    }

    @Override
    protected void insertEntry(K key, V value) {
        cles.push(key);
        valeurs.put(key, value);
    }

    @Override
    protected void reorder(K key) {
        if (cles.contains(key)) {
            cles.remove(key);
            cles.push(key);
        }
    }

    @Override
    protected K[] toArray() {
        return (K[]) cles.toArray();
    }
}
```

Pošte
Système considéré, une file avec son guichet assimilé à une file M/M/1
L = ?
Mu = ?
Tattente = 6min, donc le système est stable
D'où $L=0,75Mu = L=3/4Mu$
 $Treponse = 1/Mu - L = 1/Mu - 3/4Mu = 1/1/4Mu = 4/Mu$
Tservice = 1/Mu
 $Treponse = Tservice + Tattente = 4/Mu = 1/Mu + 6 = 4/Mu - 1/Mu = 6 = 3/Mu = 6 = Mu = 1/2client/minute$
Nouvelle config
Mu' = Mu
 $L' = 5/43/4Mu = 15/16Mu$ système stable $L' > Mu'$
T'Attente = $Treponse + Tservice = 32 - 2 = 30 minutes$
File M/M/1
Système considéré bijouterie
L=temps arrivée moyen, L=6 clients/heure
Mu = taux service moyen , Mu=10 clients/heure
On suppose que l'on est dans une file M/M/1
Mu > L donc système stable
Temps reponse = $1/10 - 6 = 1/4$ Tservice = 1/Muy= $1/10 = 6$ Duree reponse moyenne = 15minutes
Tattente = Treponse-Tservice = 15-6 = 9minutes

Système considéré boucherie
L = 40 client/heure Mu = 30 client/heure
On suppose que c'est une file M/M/1, Mu < L donc système instable, le temps d'attente est infini
Tservice = 1/Mu = 2minutes
Structure de données
Dictionnaires :
Hashset<T> -> add(x), contains(x), remove(x)->O(1)
HashMap<K,V>->put(k,v),get(k),remove(k)->O(1)
Tableaux
T[]->T[size],t(i)->O(n)
ArrayList<T>->add,get,toString -> O(1),O(n)
Queues :
ArrayDeque<T> -> add,remove,contains,remove ->O(1),O(n) ->file
ArrayDeque<T> -> push,pop,contains, remove ->O(1),O(n) ->pile
PriorityQueue<T> -> add,contains,element,remove ->O(log n),O(n),O(1),O(logn)

TP/TD

```
public static double[] sort(double[] array) {

    PriorityQueue<Double> priorityQueue = new PriorityQueue<Double>();
    int tailleTableau = array.length;
    double retour[] = new double[tailleTableau]
    for(int i = 0 ; i < tailleTableau ; i++) {
        double temp = array[i];
        priorityQueue.add(temp);
    }
    for(int i = 0 ; i < tailleTableau ; i++) {
        double temp = (double) priorityQueue.toArray()[i];
        retour[i] = temp;
    }
    return retour;
}

public static ArrayList<Double> sortArrayList(ArrayList<Double> array) {

    PriorityQueue<Double> priorityQueue = new PriorityQueue<Double>();
    int tailleArray = array.size();
    ArrayList<Double> retour = new ArrayList<Double>();
    for(int i=0; i < tailleArray ; i++) {
        double temp = array.get(i);
        priorityQueue.add(temp);
    }
    for(int i=0;i<tailleArray;i++) {
        double temp = priorityQueue.remove();
        retour.add(i , temp);
    }
    return retour;
}

public static void afficherTableau(double[] array) {
    int tailleTableau = array.length;

    for(int i = 0; i < tailleTableau ; i++) {
        System.out.println(array[i]);
    }
}
```

LRU

```
public class LruCache<K,V> extends Cache<K,V> {
    private ArrayDeque<K> cles;
    private HashMap<K,V> valeurs;

    public LruCache(int bufferSize) {
        super(bufferSize);
        this.cles = new ArrayDeque<K>();
        this.valeurs = new HashMap<K,V>();
    }

    @Override
    protected int size() {
        return cles.size();
    }

    @Override
    protected V getValue(K key) {
        return valeurs.get(key);
    }

    @Override
    protected void removeEntry() {
        K key = cles.remove();
        valeurs.remove(key);
    }

    @Override
    protected void insertEntry(K key, V value) {
        cles.add(key);
        valeurs.put(key, value);
    }

    @Override
    protected void reorder(K key) {
        if (cles.contains(key)) {
            cles.remove(key);
            cles.add(key);
        }
    }

    @Override
    protected K[] toArray() {
        return (K[]) cles.toArray();
    }
}
```

Loi de LITTLE :
Soit le système considéré, l'ensemble des messages reçus
N : taux moyen de messages dans le système
L : taux arrivée
T : temps dans le système
Donc $N=150$, $L=50/j$, $T = ?$
Système stable car N est fini, on à $N=LT$ donc $T= N/L = 150/50 = 3$ jours

Système considéré boulangerie
N : nombre moyen de client dans le système
L : taux d'arrivée
T : temps d'attente dans le système
Donc, $N=10$, $L = ?$, $T=3$
Système stable car N est fini, on a $N=LT$ donc $L=N/T= 10/3 = l=3.33$
Il y a 3.33 personnes/minutes, Pour 3 heures, 600clients entre 7 et 10h donc CA = 3000€

CACHE

```
public class StructuresTest {

    public static void main(String Args[]) {
        demoQueue();
        demoStack();
        DemoPriorityQueue();
    }

    public static void demoQueue() {
        System.out.println("File");
        ArrayDeque<Integer> file = new ArrayDeque<Integer>();
        file.add(3);
        file.add(1);
        file.add(2);
        int taille = file.size();
        for(int i=0; i < taille ; i++) {
            System.out.println(file.remove());
        }

        public static void demoStack() {
            System.out.println("Pile");
            ArrayDeque<Integer> pile = new ArrayDeque<Integer>();
            pile.add(3);
            pile.add(1);
            pile.add(2);
            int taille = pile.size();
            for(int i=0; i < taille ; i++) {
                System.out.println(pile.remove());
            }

            public static void demoPriorityQueue() {
                System.out.println("File de priorité");
                PriorityQueue<Integer> filePrio = new PriorityQueue<Integer>();

                filePrio.add(3);
                filePrio.add(1);
                filePrio.add(2);
                int taille = filePrio.size();
                for(int i=0; i < taille ; i++) {
                    System.out.println(filePrio.remove());
                }
            }
        }
    }
```

```
Public abstract class Cache <K,V>{
    private int bufferSize;
    private boolean succes;
    private int nombreSucces;
    private int nombreEchec;

    public Cache(int bufferSize) {
        this.bufferSize = bufferSize;
    }

    public V get(K key) {
        if(this.getValue(key)==null) {
            succes = false;
            nombreEchec++;
            return null;
        }else {
            succes = true;
            nombreSucces++;
            return this.getValue(key);
        }
    }

    public void put (K key, V value) {
        if(this.bufferSize < this.size()) {
            this.insertEntry(key, value);
        }else {
            this.removeEntry();
            this.insertEntry(key, value);
        }

        @Override
        public String toString() {
            String retour = "Voici le tableau de l'ordre des clés du cache :" + this.toArray().toString()
            + "\n";
            String succes = "Succes du get " + this.succes + "\n";
            String nombre = "Nombre succes : " + this.nombreSucces + " Nombre echec : " +
            this.nombreEchec + "\n";
            return retour + succes + nombre;
        }

        public void print() {
            this.toString();
        }

        protected abstract int size();
        protected abstract V getValue(K key);
        protected abstract void removeEntry();
        protected abstract void insertEntry(K key, V value);
        protected abstract void reorder(K key);
        protected abstract K[] toArray();
    }
```

Loi de LITTLE :
Soit le système considéré, l'ensemble des messages reçus
N : taux moyen de messages dans le système
L : taux arrivée
T : temps dans le système
Donc $N=150$, $L=50/j$, $T = ?$
Système stable car N est fini, on à $N=LT$ donc $T= N/L = 150/50 = 3$ jours

Système considéré boulangerie
N : nombre moyen de client dans le système
L : taux d'arrivée
T : temps d'attente dans le système
Donc, $N=10$, $L = ?$, $T=3$
Système stable car N est fini, on a $N=LT$ donc $L=N/T= 10/3 = l=3.33$
Il y a 3.33 personnes/minutes, Pour 3 heures, 600clients entre 7 et 10h donc CA = 3000€