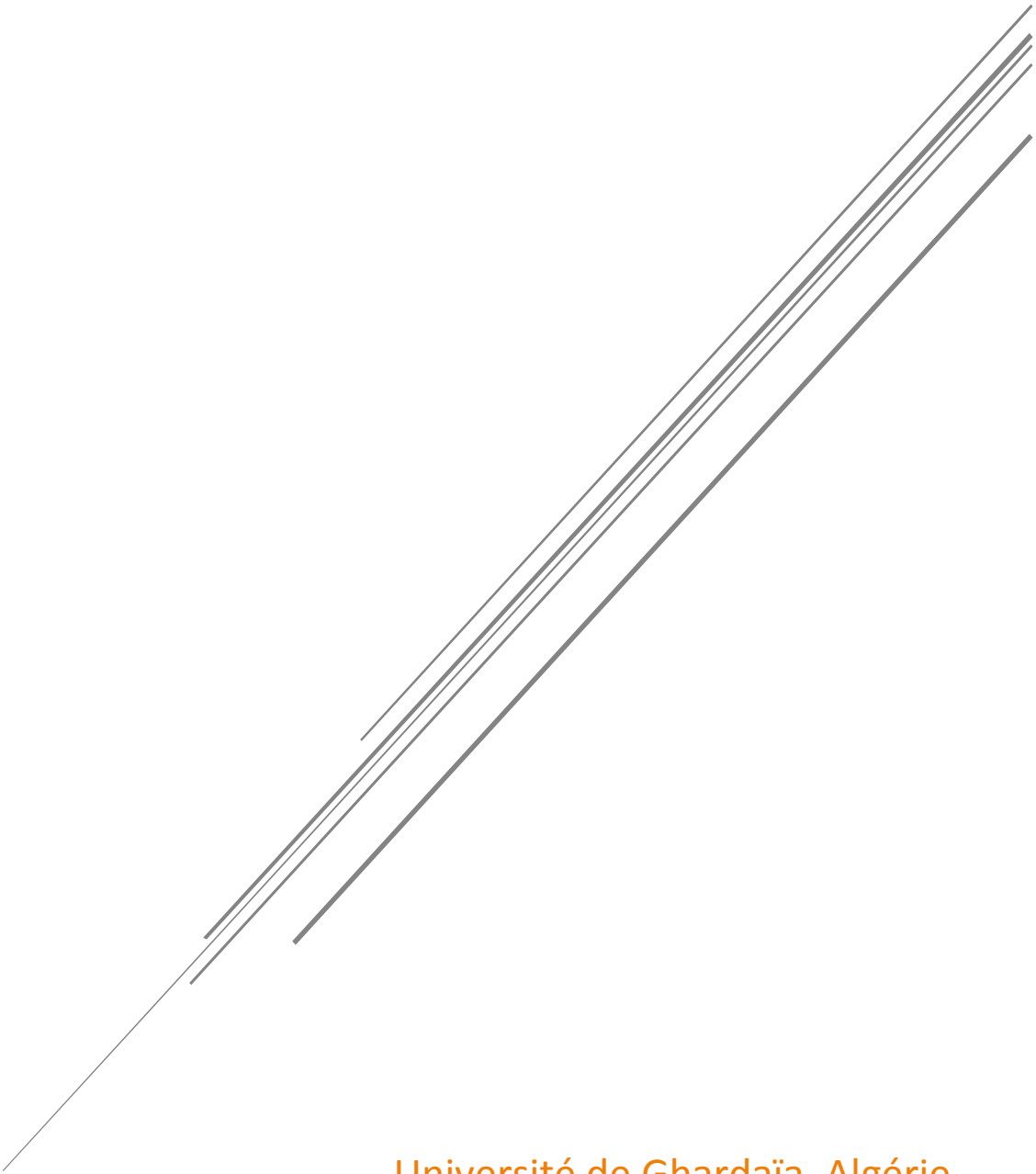
RAPPORT DE TRAVAIL

Algorithmique et Structure de Données 1



Université de Ghardaïa, Algérie --

Faculté des Sciences et de Technologie | Département Informatique et Mathématique

Ayman Nedjmeddine•B - Merzak•B

INTRO:

L'idée de ce mini-projet est principalement d'introduire la structure de données Tables de Hachage, et d'illustrer son fonctionnement à travers son implémentation en langage de programmation Java.

QUEL EST LE JOB?

Il s'agit de faire l'implémentation d'une Table De Hachage utilisant le principe de Chaînage Séparé pour la gestion des collisions.

Suivant, est le plan détaillé du rapport:

PLAN DU RAPPORT:

CONTENTS

RESSOURSES FOURNIES:

On dispose d'une archive dont le contenu est 3 class java fournies (Objet.java ; Objet3.java ; Test.java). L'archive est disponible sur le groupe Yahoo MI.

VOUS TROUVEREZ CI-JOINT A CE RAPPORT :

Un Disque Compact dont le contenu :

```
assets/ASD1 - Mini-Projet 2013-14.pdf
assets/Rapport.pdf
assets/asd1_mini_projet_2013_2014
src/Test.java
src/ExtraClass.java
src/CORE/Objet.java
src/CORE/Objet1.java
src/CORE/Objet2.java
src/CORE/Objet3.java
src/CORE/Liste.java
src/CORE/TableDeHachage.java
out/production/ASD1 Mini Project - finalVersion/Test.class
out/production/ASD1 Mini Project - finalVersion/ExtraClass.class
out/production/ASD1 Mini Project - finalVersion/Prng.class
out/production/ASD1 Mini Project - finalVersion/Liste.class
out/production/ASD1 Mini Project - finalVersion/Liste$Node.class
out/production/ASD1 Mini Project - finalVersion/TableDeHachage.class
out/production/ASD1 Mini Project - finalVersion/Objet.class
out/production/ASD1 Mini Project - finalVersion/Objet1.class
out/production/ASD1 Mini Project - finalVersion/Objet2.class
out/production/ASD1 Mini Project - finalVersion/Objet3.class
```

Le code source sera disponible aussi sur github.com/IamAyman

PROCEDURE DE TRAVAIL ET IMPLEMENTATION ATTENDUE:

On est demandé de coder, en Java, une TAD *(type abstrait de données)* Table de Hachage, suivant une structure unifiée en procédant par l'implémentation de `public abstract class Objet` ***(fournie avec le package).***

Objet Classes:

Et puisqu'elle est abstraite, on va en hériter trois fois:

`public class Objet1...` //Dont est l'implémentation de la fonction de hachage standard de Java

`public class Objet2...` //Dont est l'implémentation de la fonction alternative proposée

`public class Objet3...` //Dont est une fonction déjà implémentée *(utilisée pour le test)*

Liste Class:

Notre Table de Hachage utilisera, pour gérer ses collisions, le chaînage séparé. On est donc aussi demandé de faire notre propre implémentation des Listes Linéaires simplement Chaînées Impératives.

Il est attendu de faire une implémentation comme tel:

`public Liste ()` // Constructeur par défaut

`public Liste ajouteTete (Objet objet)` // Qui ajoutera `objet` en tête de la `Liste`, et retournera la `Liste` elle-même.

`public Liste supprimeTete () …`

/ Supprime l'élément en tête de Liste, (mais peut générer une erreur de non-existence d'élément dans le cas où la Liste est vide) et renvoie la Liste elle-même. */*

`public boolean contient (Objet objet)` // Si ou pas la `Liste` contient un `objet` de même valeur `nom`

Table de Hachage:

Comme combinaison du précédent, notre Table de Hachage sera une table de `Liste`s.

Il est attendu une telle API:

```
public TableDeHachage (int taille)  //Constructeur avec taille initiale de la table

public void ajoute (Objet objet)          // Ajouter un objet dans la table

public boolean contient(Objet objet)      //Si ou pas la table contient un objet donné

public int[] remplissageMax()
```

***/**Renvoie un tableau dont la 1re case est l'index de la Liste élément la plus longue dans la Table de Hachage, et la seconde case est sa taille.*

*Dans le cas de plusieurs Listes éléments de même taille on aura l'index et la taille de celle du plus petit hash-après-modulo (la 1re) ***/***

ALLONS PLUS EN DETAIL! =)

Commençons par la

public class Objet1 extends Objet {

Cette class contient:

`Constructeur` qui initialise l'attribut nom de la class à la valeur en paramètre

Et pour le récupérer (l'attribut) on a la méthode `public String nom()`

public int hash() {

Ce n'est qu'une seule boucle for implémentant la fonction de hachage standard de Java fournie par l'énoncé.

$$\sum_{i=0}^{n-1} s[i] * 31^{n-1-i}$$

Cette méthode fait appel à une autre méthode pour faire le calcul de puissance.

En prenant `N = nom.length()` , et considérons la multiplication pour opération significative, on aura ce ***calcul rapide de la complexité asymptotique :***

$$\sum_{i=1}^{n} 1 * coutDePuissance = \sum_{i=1}^{n} O(n) = O(n^2)$$

}

private int power (int base, int puissance){

Une méthode pour calculer a^n

Considérant la multiplication comme opération significative, on verra que cette méthode est d'ordre $O(n)$.

$$\sum_{i=1}^{n} 1 = n$$

}

}

public class Objet2 extends Objet {

Cette class contient:

`Constructeur` qui initialise l'attribut nom de la class à la valeur en paramètre

Et pour le récupérer (l'attribut) on a la méthode `public String nom()`

public int hash() {

On est supposé de faire, ici, l'implémentation de la fonction de hachage alternative proposée par l'énoncé :

$$hash'(i) = \begin{cases} 5381\,, & i = -1 \\ hash'(i-1) * 31 \oplus s[i], & i > -1 \end{cases}$$

Mais pour des raisons simplificatrices et pour une meilleure structuration du travail nous avons vu d'en faire une propre méthode et d'y faire appel dans celle-ci avec l'argument adéquat.

Un calcul rapide de sa complexité asymptotique, considérant le coût de la multiplication et de l'opération $\oplus$ est constant == 1, on déduira comme suit :

$$\begin{cases} nbrOp\ (-1) = 0 \\ nbrOp\ (i) = nbrOp\ (i-1) + 1 \end{cases}$$

$$\begin{aligned} nbrOp\ (i) &= nbrOp\ (i-1) + 1 \\ &= nbrOp\ (i-2) + 2 \\ &= nbrOp\ (i-3) + 3 \\ &= nbrOp\ (i-4) + 4 \\ &\quad \dots \\ &= nbrOop\ (i-i) + i \\ &= nbrOp\ (-1) + i + 1 \\ &= 5381 + i + 1 = i + 5382 = O(i) \end{aligned}$$

}

private int hash(int len) {

Voici donc la méthode dont on a mentionné précédemment :

```
return (i==-1)? 5381 : hash(i-1)*33^this.nom.charAt(i);
```

Récursion simple, Claire et net.

}

}

public class Objet3 extends Objet {

Celle-là a aussi un `Constructeur` pas si différent ainsi que la méthode `public String nom()` permettant de récupérer son attribut de nom.

Elle aussi sa propre fonction de hachage *(déjà implémentée)*.

Pour ce qui est de ***complexité asymptotique***, nous avons déduit qu'elle est linéaire, considérant comme constante toutes les opérations de comparaisons internes, principalement, et celle du shifting aussi :

$$\sum_{i=1}^{n} 1 = n = O(n)$$

}

public class Liste {

Nous avons maintenant besoin de listes pour stocker les éléments qui possèdent le même hash. Des Listes Linéaires simplement Chaînées suffisent. L'objet de cette partie est de faire une implémentation des Listes Impératives;

[Implémentation attendue : Liste Class:] *{Cliquez dessus}*

Comme n'importe quelle LLC, notre `Liste` Impérative à une tête ainsi qu'un indice de contrôle de taille (longueur) de la liste

```
private Node head;
private int size;
```

private class Node {

C'est bien la class définissant les cellules de la Liste.

Son constructeur a pour arguments un Objet à stocker ainsi qu'un pointeur vers le suivant.

Nous avons voulu ajouter deux autres petites méthodes dans le but de la lisibilité et compréhension de notre code plus tard, et aussi pour ne pas nous confondre avec les comparaisons des pointeurs contre `null` !

```
private boolean hasNext()
```

/* Peut être compris de deux façons:
Y a-t-il un élément suivant? Ou bien : NON (Est-ce la fin de la Liste?) */

```
private String getNodeValue()      // Quelle est la valeur de la cellule courante?
```

}

Le constructeur de la Liste `public Liste()` va initialiser la `Liste` par la création de la tête, ce n'est qu'un maillon dont le la valeur est `null` et il pointe, pour l'instant, vers le `null` aussi. Et bien sûr initialiser l'indice de taille (`size`) à 0.

La méthode `public int longueur()`, de cout constant, renvoie la valeur de l'indice de taille de `Liste`.

public Liste ajouteTete(Objet itemToAdd){

L'opération d'ajout doit être faite sur place ;

Donc, la ligne :

```
this.head.next = new Node(itemToAdd, this.head.next);
```

fera le chaînage de la tête de Liste avec le nouveau maillon, juste créé, il deviendra ainsi le nouveau premier élément de la `Liste`. Celui-là va alors nous indiquer l'ancien premier élément via son champ de pointeur suivant.

On incrémentera l'indice de taille de l'objet `Liste` et, comme demandé, renvoyer l'objet `Liste` tout entier via la ligne : `return this ;`

L'opération se termine en un temps constant.

}

public Liste supprimeTete() throws NoSuchElementException{

L'opération de suppression doit être faite sur place ;

Il est clair qu'on ne veut surement pas plonger dans les dégâts en demandant de supprimer un élément d'une `Liste` vide ! On commencera par traiter le cas critique :

```
if ( !head.hasNext() ) throw new NoSuchElementException();
```

Dans l'autre cas :

On va sauvegarder le 1re nœud *(on en aura besoin)* :

```
Node oldFirstNode = this.head.next;
```

Et on va chaîner la tête avec le 2e nœud, ce dernier deviendra ainsi le 1re nœud :

```
this.head.next = oldFirstNode.next;
```

En théorie, l'ancien 1er nœud est déjà supprimé et on ne peut lui accéder. Mais à vrai dire, on n'a cassé que le lien entre lui et la tête, mais le nœud lui-même pointe toujours vers quelque chose ! Le `Runtime.gc()` *(Garbage Collector) ne le libérera pas au prochain tour, la raison pourquoi on l'a forcé à* `null`. *{reste encore à vérifier}*

```
oldFirstNode = null;
```

Enfin, ne pas oublier de décrémenter l'indice de taille de `Liste` et renvoyer l'objet.

L'opération se termine en un temps constant.

}

public boolean contient(Objet item){

Pour vérifier si ou pas un Objet donné est un élément de la `Liste`.

Une simple boucle for dont l'indice est un nœud :

```
for (Node step=this.head.next; step!=null; step=step.next)

/* pour chaque nœud (step) de la Liste (commençant par celui en tête) et tant que le nœud existe
( !=null) On compare les noms */

    if (step.getNodeValue().equals(item.nom()))

        return true;    // Sortir avec affirmation dans ce cas

return false; //Atteindre cette instruction veut surement dire qu'il n'y existe pas de tel élément
dans la Liste
```

Cette méthode fera au maximum `size` ***comparaisons.***

}

Voilà donc finie l'implémentation de la Liste Linéaire Impérative simplement Chaînée.

}

public class TableDeHashage {

Il s'agit maintenant de combiner le travail effectué jusqu'ici pour l'implémentation de notre Table de Hachage.

[Implémentation attendue : Table de Hachage:] *{Cliquez dessus}*

`private Liste[] table ;` // Sera notre table statique de `Listes`

Le constructeur de notre table prendra la taille initiale de la table en argument pour la prendre en considération à l'initialisation.

private int getTheRightPosition(Objet item) {

Cette méthode fait en sorte de nous renvoyer la bonne position *(index)* d'un `Objet` donné, en argument, dans la table :

```
int hash = item.hash();   // Calculer le hash de l'Objet
return (hash<0)? ((hash%tableSize)+tableSize) : (hash % tableSize);
```

Renvoyer la bonne position *(hash après-modulo)* directement si le `hash` est positif, sinon additionner la longueur de la table avant le renvoi.

Et pour vous répondre explicitement en ce qui concerne votre question à propos du comportement de l'opérateur modulo avec les nombres négatifs en Java ; ça renvoie l'inverse du bon résultat (ça renvoie le modulo en négatif).

L'exécution de cette méthode ne prendra pas plus de temps que ce qu'il faut pour calculer le hash de l'`Objet` ; Toute autre opération est constante.

Autrement dit ; la ***complexité asymptotique temporelle*** de cette méthode dépend directement de `l'Objet` passé en argument (`Objet1` ou `2 ou3`), donc du temps d'exécution de sa fonction de hachage.

}

public void ajoute(Objet objetToAdd) {

Pour ajouter un `Objet` fourni en argument dans la `table`, on va d'abord lui trouver la `position` adéquate :

```
int position = getTheRightPosition(objetToAdd);
```

La table n'est pas forcement remplie ! Donc si la `Liste` à la `position` (`table[position]`) n'est pas encore initialisée on le fera d'abord :

```
table[position] = new Liste();

table[position].ajouteTete(objetToAdd);
```

Sinon, si l'`Objet` n'est pas déjà ajouté précédemment il le sera après cette ligne :

```
table[position].ajouteTete(objetToAdd);
```

En ce qui est de ***complexité asymptotique temporelle*** de cette méthode-là, elle ne prendra surement pas plus de temps que ce qu'il faudrait pour obtenir la bonne position *[voir la méthode private int getTheRightPosition(Objet item) {}] {Cliquez dessus}*. Toute autre opération est constante ainsi que la méthode `Liste.ajouteTete()` [1]

}

public boolean contient(Objet wantedObjet) {

C'est bien ce qui est si spécial à propos des Table de Hachage, vérifier (chercher) rapidement si ou pas un élément donné est dans la table.

On va calculer la position où il devrait être :

```
int position = getTheRightPosition(wantedObjet);
```

Renvoyer l'affirmation si la `Liste` à la `position` y est, et elle contient l'`Objet` recherché :

```
return table[position] != null && table[position].contient(wantedObjet);
```

Sinon, ça serai une réponse négative.

Parlons un peu ***complexité asymptotique temporelle ;*** Ce qu'on peut en dire c'est qu'elle est exactement= $O(getTheRightPosition) + O(contient)$ [2] [3]

Allons encore plus loin, puisque c'est la fonction principale !

$Liste.contient() = O(N)$ Dans le pire des cas, tel que N est la taille de la `Liste`.

On a aussi de ce qui a précédé :

$$getTheRightPosition() = \begin{cases} = O(N^2), & item.getClass() == Objet1.class \\ = O(N), & item.getClass() == Objet2.class \\ = O(N), & item.getClass() == Objet3.class \end{cases}$$

On peut donc déduire que dans le pire des cas, $getTheRightPosition() = O(N^2)$

Finalement, on peut conclure que dans le pire des cas *(une table dont les `Liste`s contiennent des Objet1)*

$$TableDeHachage.contient() = O(N) + O\left(N^2\right) = O\left(N^2\right)$$

}

public int[] remplissageMax() {

Pour trouver la `Liste` la plus longue et renvoyer un tableau dont les éléments sont, successivement, son index dans la Table de Hachage et sa longueur, on va commencer le parcours de la table cherchant le 1er index dont la Liste associée existe déjà, puisque la table n'est pas forcement pleine.

---

[1] *Voir public Liste ajouteTete(Objet itemToAdd){ } {Cliquez dessus}*

[2] *Voir méthode private int getTheRightPosition(Objet item) {} {Cliquez dessus}*

[3] *Voir Liste.public boolean contient(Objet item){} {Cliquez dessus}*

Il peut y avoir le cas où on finira le parcours mais … tout est `null` ! C'est surement le cas d'une table vide !

```
if ( table[positionMax] == null ) return new int[] {0, 0};
```

L'indice restera surement `positionMax==0`. On retournera alors un tableau dont les deux cases == 0 ;

Sinon, *(il existe au moins une* `Liste` *non-vide)* on est donc sorti de la boucle avec l'index de la 1re `Liste` non `null` qu'on considère la plus longue, à moins qu'on trouvera une autre.

On continuera ensuite le parcours de la table vérifiant toutes les longueurs des autres `Liste`s non `null` restantes dans la table. On sauvegardera à la fin l'index de celle la plus remplie.

L'analyse asymptotique temporelle de cette méthode diffère en :

Meilleur cas : Qui se réalisera au cas où la table est vide ;

Considérant la comparaison interne à la 1re boucle for, on déduira rapidement :

$$remplissageMax(\,) = \sum_{i=1}^{n} 1 = n = O(n)\,, \qquad telque\ n = tableSize$$

Pire des cas : Qui se réalisera si toutes les cases de la table existent, la 1re boucle for aura alors à faire qu'une seule itération… On procèdera donc à la 2e. Celle-ci a, par contre à la précédente, 2 comparaisons dans le ventre :

$$remplissageMax(\,) = \sum_{i=1}^{n} 2 = 2n \sim 2n = O(n)\,, \qquad telque\ n = tableSize$$

}

Voilà donc fini l'implémentation de la Table de Hachage.

}

NOUS REVOILA ENCORE!

Réponses à vos questions

LAQUELLE, PARMI LES TROIS FONCTIONS DE HACHAGE, EST LA MEILLEURE ?

D'abord, pour rappeler les principes de jugement : *« La fonction de hachage ultime n'existe pas, mais une bonne fonction est celle qui permet une bonne répartition (dite uniforme) des données dans la table, ce qui a pour effet un minimum nombre de collisions ».*

Après de nombreux tests et du debugging, on a constaté que `Objet3.hash()` semble être la meilleure parmi les trois ;

Dans la plupart des cas, la `Liste` la plus longue ne dépasse pas une taille de 5 éléments. Ce qui est encore plus clair via les résultats de la méthode `TableDeHachage.remplissageMax()`. Hors que les autres fonctions n'ont pas un si stable comportement dans la plupart des cas *(leurs valeurs diffèrent plus souvent).*

Pour les autres questions posées, on en a déjà répondu précédemment dans ce rapport, sinon sur le Code Source Java délivré.

Il y a des tests, des réponses, des questions et des questions réponses.

FINAL THOUGHTS ☺

En attendant votre réponse à nos petites questions, la version générique est en route inSha'Allah.

الأعمال العظيمة ما هي إلا أعمال صغيرة كتب لها الاستمرار

Can I Do Better?