POLYTECHNIQUE
MONTRÉAL

LOG2010
Structures de données et algorithmes

Laboratoire 2

Soumis par:
Marsolais, Edouard - 2154475
Mehdi EL Harami - 2113402

14 octobre 2022

Partie 2

a)

Le code suivant permet d'insérer les éléments d'un tableau selon la gestion des collisions voulue. On inscrit plutôt « QuadraticProbing() » ou « DoubleHashing() » et « Array2 » ou « Array3 » dans les autres cas.

```
HashTable table = new LinearProbing();
for(int i = 0; i < Array1.length; i++)
    table.insert(Array1[i]);
```

Dans la classe de base, cette méthode permet de mesurer la taille minimale, maximale et moyenne des amas :

```
public void getClusterInfo() {
    int min = array.length;
    int max = 0;

    int clusterSize = 0;
    float clusterSizeSum = 0;
    int nbCluster = 0;

    for(int i = 0; i < array.length; i++){

        if (isActive(i))
            ++clusterSize;

        else{
            if (min > clusterSize && clusterSize != 0)
                min = clusterSize;
            if (max < clusterSize)
                max = clusterSize;

            clusterSizeSum += clusterSize;

            if (clusterSize != 0){
                nbCluster++;
                clusterSize = 0;
            }
        }
    }

    System.out.println("Min: " + min);
    System.out.println("Max: " + max);
    System.out.println("Average: " + clusterSizeSum/nbCluster);
}
```

On obtient les résultats suivants :

```
Sondage linéaire - Array1
Min: 1
Max: 2
Average: 1.3636364
```

```
Sondage quadratique - Array1
Min: 1
Max: 2
Average: 1.3636364
```

```
Dispersement double - Array1
Min: 1
Max: 2
Average: 1.3
```

```
Sondage linéaire - Array2
Min: 1
Max: 7
Average: 1.6969697
```

```
Sondage quadratique - Array2
Min: 1
Max: 7
Average: 1.6470588
```

```
Dispersement double - Array2
Min: 1
Max: 5
Average: 1.5526316
```

```
Sondage linéaire - Array3
Min: 1
Max: 12
Average: 2.0833333
```

```
Sondage quadratique - Array3
Min: 1
Max: 7
Average: 1.9736842
```

```
Dispersement double - Array3
Min: 1
Max: 5
Average: 1.5625
```

Tailles des amas pour Array1

	LINÉAIRE	QUADRATIQUE	DISPERSEMENT DOUBLE
MINIMUM	1	1	1
MAXIMUM	2	2	2
MOYEN	1,36	1,36	1,30

Tailles des amas pour Array2

	LINÉAIRE	QUADRATIQUE	DISPERSEMENT DOUBLE
MINIMUM	1	1	1
MAXIMUM	7	7	5
MOYEN	1,70	1,65	1,55

Tailles des amas pour Array3

	LINÉAIRE	QUADRATIQUE	DISPERSEMENT DOUBLE
MINIMUM	1	1	1
MAXIMUM	12	7	5
MOYEN	2,08	1,97	1,56

On ajoute cet attribut et cette méthode dans la classe de base :

```
public int nbCollisions = 0;
public int getNbCollisions() {return nbCollisions;}
```

Pour chaque table de hachage, on peut compter le nombre de collisions de cette façon :

```
while( array[ currentPos ] != null &&
        !array[ currentPos ].element.equals( x ) )
{
    this.nbCollisions++;
    //...
```

On mesure aussi de temps de l'insertion des éléments dans chaque cas :

```
long temps1 = System.nanoTime();
for(int i = 0; i < Array1.length; i++)
    table.insert(Array1[i]);
System.out.println("Temps: " + (System.nanoTime()-temps1));
```

On obtient les résultats suivants :

```
Sondage linéaire - Array1
Nombre de collisions: 7
Temps (ns): 87900
Sondage linéaire - Array2
Nombre de collisions: 51
Temps (ns): 180500
Sondage linéaire - Array3
Nombre de collisions: 151
Temps (ns): 473500
```

```
Sondage quadratique - Array1
Nombre de collisions: 8
Temps (ns): 92600
Sondage quadratique - Array2
Nombre de collisions: 44
Temps (ns): 197400
Sondage quadratique - Array3
Nombre de collisions: 125
Temps (ns): 557400
```

```
Dispersement double - Array1
Nombre de collisions: 9
Temps (ns): 140400
Dispersement double - Array2
Nombre de collisions: 40
Temps (ns): 437300
Dispersement double - Array3
Nombre de collisions: 127
Temps (ns): 784200
```

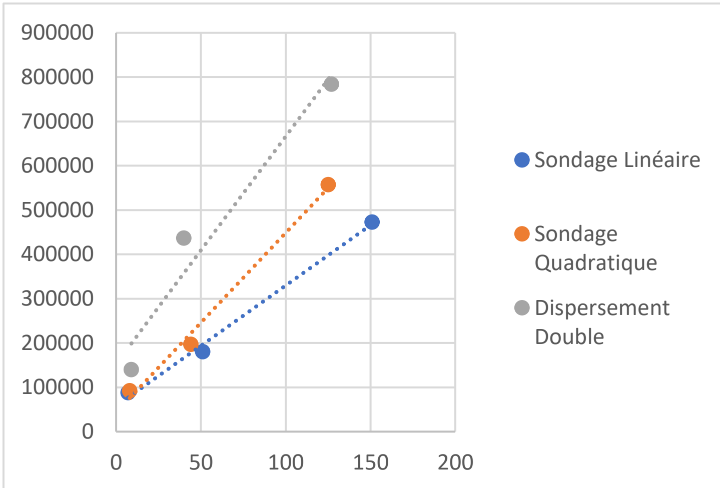
<u>Nombres de collisions et temps d'exécution</u>

	LINÉAIRE	**QUADRATIQUE**	**DISPERSEMENT DOUBLE**
ARRAY1	7 (87900 ns)	8 (92600 ns)	9 (140400 ns)
ARRAY2	51 (180500 ns)	44 (197400 ns)	40 (437300 ns)
ARRAY3	151 (473500 ns)	125 (557400 ns)	127 (784200 ns)

L'augmentation du temps d'exécution pour une même table de hachage est évidemment corrélée à l'augmentation du nombre de collisions, qui résulte d'un plus grand nombre d'éléments à insérer.

De plus, on remarque que le temps d'exécution augmente pour l'algorithme plus complexe qu'est le sondage quadratique, et encore pour le dispersement double. Ces dernières méthodes compromettent la performance pour briser les amas et économiser l'espace mémoire. Les échantillons plus élevés pour « Array2 » et « Array3 » semblent plus représentatifs de la capacité du sondage quadratique et du dispersement double à réduire le nombre de collisions par rapport au sondage linéaire.

D’après nos observations, la complexité temporelle de l’insertion d’un nombre N d’éléments pour chaque table est conforme à la théorie. Il est improbable que tous les éléments aient le même « hash », ce qui résulterait en une complexité d’une insertion de O(n). Ainsi, en considérant N insertions de O(1), la complexité semble se rapprocher de O(n).



b)

Nous modifions le « load factor » dans la méthode « Insert » de cette façon :

```
if(++currentSize > array.length * 0.25f)
    rehash();
```

« Load factor » de 0,25.

```
Sondage linéaire - Array1
Nombre de collisions: 3
```

```
Sondage quadratique - Array1
Nombre de collisions: 3
```

```
Dispersement double - Array1
Nombre de collisions: 3
```

```
Sondage linéaire - Array2
Nombre de collisions: 17
```

```
Sondage quadratique - Array2
Nombre de collisions: 17
```

```
Dispersement double - Array2
Nombre de collisions: 21
```

```
Sondage linéaire - Array3
Nombre de collisions: 50
```

```
Sondage quadratique - Array3
Nombre de collisions: 47
```

```
Dispersement double - Array3
Nombre de collisions: 53
```

<u>Nombres de collisions (« load factor » de 0,25)</u>

	LINÉAIRE	QUADRATIQUE	DISPERSEMENT DOUBLE
ARRAY1	3	3	3
ARRAY2	17	17	21
ARRAY3	50	47	53

« Load factor » de 0,75.

```
Sondage linéaire - Array1
Nombre de collisions: 24
```

```
Sondage quadratique - Array1
Nombre de collisions: 17
```

```
Dispersement double - Array1
Nombre de collisions: 13
```

```
Sondage linéaire - Array2
Nombre de collisions: 107
```

```
Sondage quadratique - Array2
Nombre de collisions: 89
```

```
Dispersement double - Array2
Nombre de collisions: 94
```

```
Sondage linéaire - Array3
Nombre de collisions: 402
```

```
Sondage quadratique - Array3
Nombre de collisions: 280
```

```
Dispersement double - Array3
Nombre de collisions: 290
```

<u>Nombres de collisions (« load factor » de 0,75)</u>

	LINÉAIRE	QUADRATIQUE	DISPERSEMENT DOUBLE
ARRAY1	24	17	13
ARRAY2	107	89	94
ARRAY3	402	280	290

Par rapport à notre tableau de la partie a), nous observons qu'en règle générale, diminuer le « load factor » réduit aussi le nombre de collisions et vice-versa. Le hachage d'un élément dépend de la taille de la table. Puisqu'un « load factor » de 0,25 signifie que la table s'agrandit fréquemment, il devient moins probable que deux éléments aient le même « hash » dans ce cas.

c)

Nous commentons les lignes de la partie b) pour suspendre le « Rehash ». La complexité asymptotique de cette opération était O(n).

```
Sondage linéaire - Array1
Nombre de collisions: 0
Temps (ns): 64900
```

```
Sondage quadratique - Array1
Nombre de collisions: 0
Temps (ns): 77400
```

```
Dispersement double - Array1
Nombre de collisions: 0
Temps (ns): 82400
```

```
Sondage linéaire - Array2
Nombre de collisions: 31
Temps (ns): 115800
```

```
Sondage quadratique - Array2
Nombre de collisions: 21
Temps (ns): 93100
```

```
Dispersement double - Array2
Nombre de collisions: 18
Temps (ns): 148400
```

```
Sondage linéaire - Array3
Nombre de collisions: 924
Temps (ns): 636000
```

```
Sondage quadratique - Array3
Nombre de collisions: 471
Temps (ns): 508900
```

```
Dispersement double - Array3
Nombre de collisions: 465
Temps (ns): 474800
```

Nombres de collisions et temps d'exécution

	LINÉAIRE	QUADRATIQUE	DISPERSEMENT DOUBLE
ARRAY1	0 (64900 ns)	0 (77400 ns)	0 (82400 ns)
ARRAY2	31 (115800 ns)	21 (93100 ns)	18 (148400 ns)
ARRAY3	924 (636000 ns)	471 (508900 ns)	465 (474800 ns)

D'après ces résultats, nous observons que lorsque la table est amplement grande pour contenir les éléments de « Array1 » ou « Array2 », le nombre de conflits et le temps d'exécution diminuent. Dans ces cas, on se rapproche de la situation idéale où une insertion est de complexité O(1), car il est moins probable que deux éléments partagent le même « hash ». On soustrait aussi les temps d'exécution associés au « Rehash ».

Lorsque la taille de la table peine à contenir les éléments, le nombre de collisions augmente drastiquement. Dans cette situation causée par le « Array3 », la meilleure gestion des collisions du sondage quadratique et du dispersement double est évidente. On en conclut que le sondage linéaire mène à de gros amas que doivent plus souvent parcourir les éléments à insérer. Toutefois, le temps d'exécution n'a pas drastiquement augmenté par rapport au tableau de la partie a). L'opération du « Rehash » a une forte influence sur ce temps d'exécution, à force de réinsérer N éléments dans une table plus grande à maintes reprises.