TP4 Java Avancé - Hacher menu

Max Ducoudré - INFO2

Exercice 2 - HashTableSet

1. Quels doivent être les champs de la classe Entry correspondant à une case d'une des listes chaînées utilisées par table de hachage Note : on va pas utiliser java.util.LinkedList, car on veut une liste simplement chaînée. Rappeler quelle est l'intérêt de déclarer Entry comme membre de la classe HashTableSet plutôt que comme une classe à coté dans le même package que HashTableSet ? Ne pourrait-on pas utiliser un record plutôt qu'une classe, ici ? Si oui, pourquoi ? Si non, pourquoi ? Écrire la classe HashTableSet dans le package fr.uge.set et ajouter Entry en tant que classe interne.

La classe `Entry` a deux champs : un next (de type `Entry`) et un data (d'un type `E` paramétré). Cette classe correspond à un maillon de liste chaînée.

L'intérêt de décalrer Entry comme membre de HashTable set est le suivant : La personne qui vas utiliser la classe `HashTableSet` n'a pas besoin de savoir comment elle est implémentée et donc n'a pas besoin de pouvoir voir la classe `Entry`

On souhaite ici utiliser un record pour `Entry` car une instance d'un maillon d'une chaîne ne vera pas ses champs être modifiés

Voici l'implémentation :

```
public final class HashTableSet<E> {
    static private final record Entry<E>(Entry<E> next, E data) {
        public Entry {
            Objects.requireNonNull(data);
        }
    }
}
```

2. On souhaite maintenant ajouter un constructeur sans paramètre, une méthode add qui permet d'ajouter un élément non null et une méthode size qui renvoie le nombre d'éléments insérés (avec une complexité en O(1)). Pour l'instant, on va dire que la taille du tableau est toujours 16, on fera en sorte que la table de hachage s'agrandisse toute seule plus tard. Dans la classe HashTableSet, implanter le constructeur et les méthodes add et size. Vérifier que les tests marqués "Q2" passent.

On ajoute un champ `entries` correspondant à un tableau d' `Entry`. Ce tableau nous permet de stocker toutes les listes chaînées de notre table de hashage.
On ajoute une méthode `add` qui, à l'index `element.hashCode()%SIZE`, ajoute l'élément au début de la liste chaînée s'il n'est pas déjà contenu dedans.

Enfin, la méthode `size` vas itérer toutes les listes chaînées du tableau pour compter le nombre d'éléments qui ont été ajoutés

Voici l'implémentation :

```
public final class HashTableSet<E> {

    private final static int SIZE = 16;

    @SuppressWarnings("unchecked")
    private final Entry<E>[] entries = new Entry[SIZE];

    static private final record Entry<E>(Entry<E> next, E data) {
        public Entry {
            Objects.requireNonNull(data);
        }
    }

    public HashTableSet() {
        for(int i = 0; i < capacity; i++) {
            entries[i] = null;
        }
    }


    public void add(E element) {
        Objects.requireNonNull(element);
        int index = Math.abs(element.hashCode() % SIZE);
        var entry = entries[index];
        while (entry != null) {
            if (entry.data.equals(element)) {
                return;
            }
            entry = entry.next();
        }
        entries[index] = new Entry<>(entries[index], element);

    }

    public int size() {
        int size = 0;
        for (Entry<E> entry : entries) {
            while (entry != null) {
                size++;
                entry = entry.next();
            }
        }
        return size;
    }
```

```
}
```

3. On cherche maintenant à implanter une méthode forEach qui prend en paramètre une fonction. La méthode forEach parcourt tous les éléments insérés et pour chaque élément, appelle la fonction prise en paramètre avec l'élément courant. Quelle doit être la signature de la functional interface prise en paramètre de la méthode forEach ? Quel est le nom de la classe du package java.util.function qui a une méthode ayant la même signature ? Écrire la méthode forEach.

La méthode `forEach` vas avoir bestoin d'appliquer une une interface fonctionelle `Consumer` qui prend un paramètre et ne renvois rien. Voici une implémentation de forEach :

```
    public void forEach(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        Arrays.stream(entries).forEach(entry -> {
            while (entry != null) {
                action.accept(entry.data);
                entry = entry.next();
            }
        });
    }
```

4. On souhaite maintenant ajouter une méthode contains qui renvoie si un objet pris en paramètre est un élément de l'ensemble ou pas, sous forme d'un booléen. Expliquer pourquoi nous n'allons pas utiliser forEach pour implanter contains (Il y a deux raisons, une algorithmique et une spécifique à Java). Écrire la méthode contains.

Voici une implémentation de contains :

```
    public boolean contains(E element) {
        int index = Math.abs(element.hashCode() % SIZE);
        var entry = entries[index];
        while (entry != null) {
            if (entry.data.equals(element)) {
                return true;
            }
            entry = entry.next();
        }
        return false;
    }

        public boolean contains(E element) {
        Objects.requireNonNull(element);
        int index = Math.abs(element.hashCode() % capacity);
        var entry = entries[index];
        while (entry != null) {
            if (entry.data.equals(element)) {
                return true;
```

```
            }
            entry = entry.next();
        }
        return false;
    }
```

Nous n'utilisons pas forEach pour implanter contains car il n'est pas possible de quitter la fonction contains depuis un `forEach`, ce qu'on souhaite faire quand on rencontre un élément égal à l'argument. De plus, forEach prend un consumer, et il n'est pas possible de renvoyer une valeur booléenne depuis un consumer.

5. On veut maintenant faire en sorte que la table de hachage se redimensionne toute seule. Pour cela, lors de l'ajout d'un élément, on peut avoir à agrandir la table pour garder comme invariant que la taille du tableau est au moins 2 fois plus grande que le nombre d'éléments. Pour agrandir la table, on va créer un nouveau tableau deux fois plus grand et recopier touts les éléments dans ce nouveau tableau à la bonne place. Ensuite, il suffit de remplacer l'ancien tableau par le nouveau. Expliquer pourquoi, en plus d'être plus lisible, en termes de performance, l'agrandissement doit se faire dans sa propre méthode. Modifier votre implantation pour que la table s'agrandisse dynamiquement.

Pour cela, on ajoute une condition au début de chaque add pour vérifier si le nombre d'élément dépasse la moitié de la capacité :

```
public void add(E element) {
    Objects.requireNonNull(element);
    if(size() >= capacity/2) {
        increaseCapacity();
    }
    /* [...] */
}
```

Pour améliorer la vitesse du programme, on vas stocker la taille du tableau dans un champs `size` qu'on incrémente à chaque plutôt que de le recalculer à chaque fois :

```
private int size = 0;
/* [...] */
public void add(E element) {
    Objects.requireNonNull(element);
    if(size >= capacity/2) {
        increaseCapacity();
    }
    /* [...] */
    size++;
    /* [...] */
}
```

La méthode `increaseCapacity` vas doubler la capacité du tableau en en créant un nouveau comme ceci :

```
    private void increaseCapacity() {
        Entry<E>[] newEntries = new Entry[capacity*2];

        forEach(element -> {
            int index = hash(element);
            newEntries[index] = entries[index];
        });

        capacity *= 2;
        entries = newEntries;
    }
```

6. L'implantation actuelle a un problème : même si on n'ajoute que des String lorsque l'on utilise forEach, l'utilisateur va probablement devoir faire des cast parce que les éléments envoyés par forEach sont typés Object.

Rappeler pourquoi en Java, il n'est pas possible de créer un tableau de type paramétré ? Quel est le work around ? Pourquoi celui-ci génère-t-il un warning ? Et dans quel cas et comment peut on supprimer ce warning ?

En java, il n'est pas possible de créer des tableaux de type paramétré car le type de l'objet passé en paramètre n'existe pas encore à l'exécution

7. En fait, la signature de la méthode forEach que vous avez écrite n'est pas la bonne. En effet, forEach appelle la lambda avec des éléments de type E, donc la fonction peut prendre en paramètre des valeurs qui sont des super-types de E.

On modifie la signature de la méthode `forEach` :

```
public void forEach(Consumer<? super E> action) {
    /* [Le code reste inchangé] */
}
```

8. On souhaite maintenant écrire une méthode addAll qui permet d'ajouter tous les éléments d'un HashTableSet dans le HashTableSet courant.

```
    public void addAll(HashTableSet<? extends E> hashTableSet) {
        hashTableSet.forEach(this::add);
    }
```