Université Paris Diderot **Programmation Orientée Objet**
L2 **Année 2021-2022**

TD - Séance n°10

Contrôle Continu

Exercice 1 Nous allons implémenter une version simplifiée de table de hachage. Il s'agit d'une structure de données qui permet d'organiser et de chercher efficacement dans une collection d'*entrées*. Chaque entrée de la table de hachage comporte deux parties : une partie *clef*, représentant un identifiant (c-a-d -il n'y a pas deux entrées dans une même table de hachage avec clefs égales), et une partie *valeur* arbitraire. Clef et valeur sont deux objets (class `Object`). Sur une clef, la méthode `equals` sera utilisée pour tester l'égalité avec une autre clef. Une table de hachage permet de chercher une entrée, donnée sa clef, insérer une nouvelle entrée, ainsi que supprimer une entrée existante. Nous allons aussi implementer une méthode de tri des entrées par clef.

Ecrire une interface `TableHash` avec les méthodes suivantes :

- `size` qui retourne la taille (nombre d'entrées) de la table de hachage.
- `searchVal` qui reçoit une clef en paramètre et est censée retourner la valeur de l'(unique) entrée de la table de hachage possédant cette clef ; cette méthode soulève une exception `ClefAbsenteException` si aucune entrée avec cette clef n'est présente dans la table.
- `insert` qui reçoit en paramètre deux objets, clef et valeur, et est censée insérer dans la table de hachage une entrée formée par cette clef et cette valeur ; cette méthode lève une exception `ClefPresenteException` si une entrée avec cette clef est déjà présente dans la table.
- `remove` qui reçoit en paramètre une clef et un `Predicate<Object> testVal`. Cette méthode est censée supprimer de la table de hachage l'(unique) entrée possédant cette clef, à condition que la valeur de cette entrée passe le test `testVal`.
- `remove` qui reçoit en paramètre une clef et supprime de la table de hachage l'(unique) entrée possédant cette clef ; cette méthode doit avoir une définition par défaut : elle invoque la méthode abstraite `remove` du point précèdent en lui passant le bon `Predicate<Object>` (en tant qu'expression lambda).
- `sort` qui reçoit en entrée un `Comparator<Object> clefComp` permettant de comparer deux clefs, et renvoie la liste de toutes les entrées de la table, triées par clef selon le critère fourni par `clefComp`.

Pour representer les entrées de la table, définir la classe `Entrée` comme classe interne de l'interface `TableHash`. Ne pas oublier les méthodes d'accès `getClef()` et `getVal()`. Définir les classes exceptions également comme classes internes de l'interface. Justifier les modificateurs choisis pour toutes ces classes internes (`static` ou pas, `public` ou `private`). Pour les classes exceptions justifier le choix d'exception *checked* ou *unchecked*.

Rappels utiles :

- `Predicate<Object>` est une interface fonctionnelle possédant l'unique méthode `boolean test(Object o)`.
- `Comparator<Object>` est une interface fonctionnelle possédant l'unique méthode `int compare(Object o1, Object o2)`.

Exercice 2 Définir une classe `TableHashChainage` qui implémente l'interface `TableHash`

de l'exercice précédent. Cette classe comporte un tableau `T` où les entrées sont rangées de la façon suivante.

Chaque élément `T[i]` du tableau est une liste d'entrées (objets de la classe `Entree`) appelée *liste d'overflow*. Une entrée de la table de hachage ayant clef `c` doit toujours se trouver dans la liste d'overflow `T[h(c)]`, où `h(c)` est définie comme la valeur absolue de `c.hashCode()`% `T.length`.

Pour représenter une liste d'overflow (c'est-à-dire un element du tableau `T`) on utilisera une classe `ListeOverflow` interne à la classe `TableHashChainage`.

La classe `ListeOverflow` doit étendre `LinkedList<Entree>` et fournir une méthode d'utilité `int searchClef(Object c)`. Cette méthode cherche dans la liste l'entrée de clef `c` et en retourne l'indice (position dans la liste). On pourra effectuer cette recherche en se servant de la méthode `int indexOf (Entree e)` de la classe `LinkedList<Entree>`. Cette méthode retourne la premiere position de la liste contenant un entrée `f` telle que `e.equals(f)` est vrai (renvoie -1 si l'entrée n'est pas trouvée).

(**Remarque**. Pour se servir utilement de la méthode `indexOf` il pourrait être nécessaire de modifier la classe `Entree`.)

Justifier les modificateurs choisis pour la classe interne `ListeOverflow` (`static` ou pas, `public` ou `private`).

La classe `TableHashChainage` a un constructeur qui crée une table de hachage vide de taille maximale `tailleMax` (donnant la taille du tableau `T`). Ensuite elle implémente toutes le méthodes abstraites de l'interface `TableHash`. En particulier :

- `insert(Object clef, Object val)` insère l'entrée `(c, v)` dans une position quelconque de la liste d'overflow `T[h(c)]`, par exemple en tête (méthode `addFirst` de `LinkedList`).
  La recherche et la suppression d'une entree de clef `c` nécessitent en revanche le parcours de la liste d'overflow `T[h(c)]`, ce qui peut être effectué par la méthode d'utilité `searchClef` de la classe `ListeOverflow`.
- `searchVal(Object clef)` peut utiliser la méthode `get(int index)` de `LinkedList` qui retourne l' `Entrée` en position `index`.
- `remove (Object clef, Predicate<Object> testVal)` peut se servir de la méthode `remove(int index)` de la classe `LinkedList` qui supprime l'entrée de la liste en position `index`.
- `sort (Comparator<Object> clefComp)`. insère toutes les entrées de la table de hachage dans une nouvelle liste (par la méthode `addAll (LinkedList<Entree>)` de la classe `LinkedList<Entree>`), sur laquelle on invoque ensuite la méthode `void sort(Comparator<Entree> comp)`. Pour ce faire il faut passer à cette méthode une expression lambda qui compare deux paramètres de classe `Entree` en comparant leur deux clefs (avec `clefComp`).
- `size()` retourne la somme des tailles de toutes les listes d'overflow.