IPO – TP 3 : structures de données

http://www.lri.fr/~blsk/IPO/

Dans ce TP, on réalise en java quelques structures de données que vous avez étudiées l'année dernière en cours d'ASD. Tout au long de ce TP, on vous encourage à avoir à portée de main du papier et un crayon, pour visualiser des schémas de ces structures et des opérations à réaliser. Note : chacune de ces structures existe par ailleurs dans la bibliothèque standard, et à partir de la semaine prochaine vous pourrez en utiliser les versions « officielles » dès que vous en aurez besoin.

Listes chaînées. Une *liste chaînée* est constituée de *cellules*, contenant chacune deux choses :

— un élément de la liste,

— un pointeur vers une autre cellule (la « prochaine »), ou le pointeur nul (s'il n'y a pas de prochaine cellule).

La *tête* d'une liste est sa première cellule.

Définir une classe `Cell` représentant une cellule. Elle doit contenir :

1. un attribut `elt` de type `Object`,
2. un attribut `nextCell` de type `Cell`,
3. un constructeur
4. des méthodes `getElt`, `getNextCell` et `setNextCell` conventionnelles.

Le code client donné dans le fichier `TestCell.java` doit afficher, dans l'ordre, les mots `the`, `world`, `is` et `square`.

Piles. La *pile* est une structure de données dans laquelle on peut ajouter ou retirer des éléments, avec la discipline suivante : l'élément retiré est systématiquement l'élément le plus récent de la structure (*last in, first out*). On peut réaliser une telle structure à l'aide d'une liste chaînée de la manière suivante :

— la structure contient un pointeur vers la première cellule d'une liste chaînée,

— pour ajouter un élément au sommet de la pile, on ajoute une cellule en tête de la liste (et on met à jour le pointeur vers la première cellule),

— pour retirer l'élément au sommet de la pile, on retire la cellule de tête (en mettant à jour le pointeur de tête).

Définir une classe `Stack` réalisant une structure de pile. Elle doit contenir :

1. un attribut `top` de type `Cell` pointant vers la première cellule de la liste chaînée sous-jacente (ou valant **`null`** si la liste est vide), et un attribut `size` enregistrant le nombre d'éléments dans la pile,
2. un constructeur,
3. une méthode **`boolean`** `isEmpty()` renvoyant **`true`** si la pile est vide, et une méthode **`int`** `getSize()`,
4. une méthode **`void`** `push(Object)` ajoutant un nouvel élément au sommet de la pile,
5. une méthode `Object pop()` retirant et renvoyant l'élément présent au sommet de la pile (en supposant la pile non vide).

Le code client donné dans le fichier `Compute.java` implémente une calculatrice en notation polonaise inverse. La ligne de commande

```
java Compute 1 2 3 x + 4 +
```

doit calculer $1 + 2 \times 3 + 4 = 11$ et afficher le résultat. Consigne supplémentaire : ajoutez également du code pour tester chaque méthode indépendamment des autres.

Files. La *file* est une structure de données dans laquelle on peut ajouter ou retirer des éléments, avec la discipline suivante : l'élément retiré est systématiquement l'élément le plus vieux de la structure (*first in, first out*). On peut réaliser une telle structure à l'aide d'une liste chaînée de la manière suivante :

— la structure contient un pointeur vers la première cellule et un pointeur vers la dernière cellule d'une liste chaînée,

— pour ajouter un élément à la fin de la file, on ajoute une nouvelle cellule après la dernière (et on met à jour le pointeur arrière),

— pour retirer l'élément du début de la file, on retire la cellule de tête (et on met à jour le pointeur avant).

Définir une classe `Queue` réalisant une structure de file. Elle doit contenir :

1. deux attributs `first` et `last` de type `Cell` pointant respectivement vers la première et la dernière cellule de la liste chaînée sous-jacente (ou valant **`null`** si la liste est vide), et un attribut `size` enregistrant le nombre d'éléments,
2. un constructeur,
3. une méthode **`boolean`** `isEmpty()` renvoyant **`true`** si la pile est vide, et une méthode **`int`** `getSize()`,
4. une méthode **`void`** `enqueue(Object)` ajoutant un nouvel élément à la fin de la file,
5. une méthode `Object dequeue()` retirant et renvoyant l'élément présent à l'avant de la file (en supposant la file non vide).

Le code client donné dans le fichier `Count.java` attend un argument k sur la ligne de commande et affiche tous les nombres binaires à k chiffres. Ajoutez également du code pour tester chaque méthode.

Tables de hachage. Une *table de hachage* est une structure de données dans laquelle on peut enregistrer des paires (k, v) associant une *clé* k et une *valeur* v, et dans laquelle on peut effectuer des requêtes basées sur les clés. On peut réaliser une telle structure de la manière suivante :

— on prend comme support un tableau `t` d'une taille N arbitraire, dont chaque case contiendra une listes de paires (k, v),

— chaque clé est associée à un *code* entier, que l'on peut obtenir en java avec la méthode `hashCode()`,

— la case d'indice i du tableau `t` contient toutes les paires (k, v) telles que k`.hashCode()` est égal à i modulo N.

— pour ajouter une nouvelle paire (k, v), il suffit d'ajouter une cellule en tête de la liste dans la case adaptée,

— pour tester la présence d'une clé k, il faut déterminer la case adaptée, puis parcourir la liste.

Créer une nouvelle classe `Cell`, inspirée de la précédente, pour représenter une cellule d'une liste chaînée contenant une clé et une valeur. La classe doit contenir :

1. des attributs `key` et `value` de type `Object`, et un attribut `nextCell`,
2. un constructeur,
3. des méthodes `getKey`, `getValue`, `setValue`, `getNextCell` et `setNextCell`,
4. une méthode **`boolean`** `hasKey(Object k)` qui renvoie **`true`** si la clé contenue par la cellule est égale à la clé `k` prise en argument.

Créer ensuite une classe `Hashtable` pour représenter une table de hachage. La classe doit contenir :

1. un attribut `t` de type `Cell[]` pour le tableau de listes de paires, et un attribut `size` pour le nombre de paires (k, v) présentes dans la structure,
2. un constructeur, qui initialise le tableau avec une taille de départ arbitraire (par exemple : 8),
3. une méthode **`void`** `put(Object k, Object v)`, qui place la nouvelle paire `(k, v)` donnée en argument en tête de la bonne liste (dont le numéro est déterminé par le code de la clé `k`),
4. une méthode **`boolean`** `contains(Object k)` qui renvoie **`true`** si et seulement si la table contient une paire (k, v) avec une clé k égale à la clé `k` donnée en argument (indication : cette méthode doit commencer par déterminer dans quelle liste la clé est susceptible de se trouver),
5. une méthode `Object get(Object k)` qui cherche la valeur associée à la clé `k`, en supposant que la table contienne bien une paire (k, v) avec la bonne clé,

La classe telle que définie jusque là suffit à faire fonctionner le code client donné dans le fichier `MostFrequent.java`, qui détermine le mot le plus fréquent parmi tous les mots donnés en argument sur la ligne de commande.

Pour aller plus loin, quelques améliorations de la classe `Hashtable` :

6. ajouter une méthode **`void`** `remove(Object k)` qui retire une paire (k, v) portant la clé `k` donnée en argument, en supposant qu'il en existe bien au moins une,
7. modifier la méthode `put` pour qu'elle retire une éventuelle association précédente avec la même clé avant de procéder à l'ajout de la nouvelle paire,
8. ajouter une méthode `extend` qui remplace le tableau `t` sous-jacent par un nouveau tableau de taille double, et y redistribue toutes les paires (k, v) qui étaient présentes (comme le modulo n'est plus le même, les listes pointées par chaque case ne seront plus les mêmes),
9. modifier la méthode `put` pour qu'elle fasse appel à `extend` dès que le nombre de paires dans la structure dépasse la taille N du tableau sous-jacent.