TP n° 3: Héritage et polymorphisme

1 Implémentation des Piles et des files d'attente

Une **file** d'attente est une **liste** de données qui suit le principe de *Premier Arrivé Premier Servi : First In First Out (FIFO)*. Ce principe consiste à ajouter un élément, seulement, en fin de liste, et à supprimer un élément de la tête de liste. D'un autre coté, une **pile** de données est une **liste** qui suit le principe de *Dernier Arrivé Premier Servi : Last In First Out (LIFO)*, où l'ajout de nouveaux éléments se fait en fin de liste, et la suppression d'éléments se fait également à partir de la fin de liste.
Nous voulons utiliser la classe `Liste`, définie dans la **première partie** du TP n° 1, pour implémenter les files et piles de données, en utilisant la notion de l'héritage vue en cours :

1. Définir une classe `Pile`, sous-classe de la classe `Liste`, et qui contient un entier représentant le *sommet* de la pile (c'est à dire le dernier élément ajouté).
2. Définir des constructeurs qui permettent de créer un pile :
   (a) vide d'une taille égale à 0, ou
   (b) vide mais d'une taille donnée, ou
   (c) à partir d'un tableau d'entier, ou
   (d) à partir d'un tableau de chaîne de caractères.
3. Dans la classe `Pile` définir les méthodes `public void inserer(int a)` et `public int supprimer()` qui permettent d'ajouter l'élément `a` (empiler) et de supprimer un élément (dépiler en retournant l'élément supprimé), selon le principe *LIFO*.
4. Semblablement, définir la classe `File` qui hérite de `Liste` et qui contient deux attributs représentant la tête et la fin de la file (c'est à dire le premier élément ajouté et le dernier élément ajouté, respectivement).
5. Une file peut également être créée en 4 manières :
   (a) vide d'une taille égale à 0, ou
   (b) vide mais d'une taille donnée, ou
   (c) à partir d'un tableau d'entier, ou
   (d) à partir d'un tableau de chaîne de caractères.
6. Définir également les méthodes `public void inserer(int a)` et `public int supprimer()` permettant, respectivement, d'ajouter l'élément `a` (enfiler) et de supprimer un élément (défiler en retournant l'élément supprimé), selon le principe *FIFO*.

Redéfinir, dans les classes `File` et `Pile`, les méthodes `public void inserer(int i, int a)` et `public void supprimer(int i)` qui permettent respectivement d'ajouter l'élément `a` à la position `i`, et de supprimer l'élément de la position `i` de la structure de données, en respectant les conditions suivantes :

1. Dans une pile, ajouter un élément à la position `i` revient à dépiler tous les éléments à partir du sommet jusqu'à arriver à la position voulue, d'empiler le nouvel élément, puis empiler les autres éléments dans le même ordre.
2. De la même manière, supprimer un élément de la position i, dans une pile, revient à dépiler tous les éléments jusqu'à arriver à la position `i`, dépiler l'élément voulu, puis d'empiler les éléments dépilés à la pile, en gardant le même ordre.
3. Dans une file d'attente, pour ajouter ou supprimer un élément d'une position `i`, il faut défiler tous les éléments de la file, puis de les enfiler en ajoutant ou en supprimant l'élément voulu, en suivant le même ordre.

2 Exemple d'application

1. Modifier les classes `Liste`, `Pile` et `File` pour qu'elles permettent de manipuler **n'importe quel objet Java**.
2. Définir la classe `Tache` qui contient un message de type chaîne de caractère, et un entier représentant la priorité d'une tâche. Les attributs doivent être définis *privés*
3. Définir le constructeur de la classe `Tache` qui permet d'initialiser ses attributs par des valeurs passées en paramètre.
4. Définir la classe `Processeur` qui contient trois files d'attentes de `Tache` : une file pour les tâches à priorité basse (entre 0 et 3), une file pour les tâches à priorité moyenne (entre 4 et 6) et une autre pour les tâches à priorité élevée (entre 7 et 10).
5. Dans la classe `Processeur`, définir une méthode `public void produire(Tache tache)` qui permet d'enfiler un tâche dans une des files d'attente internes, en fonction de sa priorité.
6. Définir également une méthode `public Tache consommer()` qui retourne la tâche la plus prioritaire. Si toutes les files d'attente sont vide, la méthode doit retourner une tâche qui contient le message "*vide*" avec une priorité égale à -1.
7. Définir une classe qui contient la méthode principale `main` dans laquelle on fait le jeu de test suivant :
   - Dans un processeur produire les tâches suivantes dans l'ordre :
     - tâche1 (message : "a one-way", priorité =1)
     - tâche2 (message : "a good", priorité =9)
     - tâche3 (message : "always looks", priorité =6)
     - tâche4 (message : "street", priorité =3)
     - tâche5 (message : "both ways", priorité =5)
     - tâche6 (message : "programmer is", priorité =8)
     - tâche7 (message : "before crossing", priorité =4)
     - tâche8 (message : "someone who", priorité =10)
   - Consommer toutes les tâches à partir des files d'attente et afficher le message de chaque tâche. Quel est le résultat affiché ?