SDA : TP8

DUT/INFO/M1103
version 2016-2017 (PPN 2013)

Table des matières

1. Le type File
2. Une file simple mais très gourmande en mémoire
3. Une meilleure file
4. De l'utilité d'une file
5. Aller plus loin
5.1. Autres implémentations
5.2. Une file qui s'agrandit

1. Le type File

La file dite FIFO (First In First Out) est une structure de données qui fournit deux fonctionnalités principales :

- enfiler un élément, ce qui permet de stocker cet élément
- défiler, ce qui permet de retirer un élément de la file.

La file assure que les éléments seront défilés dans l'ordre dans lequel ils ont été enfilés : si un élément A a été enfilé avant un élément B, cet élément A sera défilé avant l'élément B. Cela la différencie de la pile, que l'on a vue lors des précédents TPs, et qui, elle, assure que si un élément A est empilé avant un élément B, cet élément A sera dépilé **après** l'élément B.

L'image suivante décrit un cas d'utilisation d'une file.

La file est dite vide lorsqu'elle ne contient aucun élément, et est dite pleine lorsqu'on ne peut pas enfiler un nouvel élément.

Une file propose donc les opérations dont on donne les renseignements principaux dans ce qui suit :

Signatures

```
creerFile : Entier -> File
enfiler : File * Element -> File
defiler : File -> File
prochainElement : File -> Element
estVide : File -> Booleen
estPleine : File -> Booleen
```

Préconditions

```
creerFile() : -
enfiler(f,e) : f n'est pas pleine
defiler(f) : f n'est pas vide
prochainElement(f) : f n'est pas vide
estVide : -
estPleine : -
```

Une file peut être implémentée de différentes manières, et nous allons voir ici quelques unes de ses implémentations possibles. Ces implémentations comportent certains points communs : les éléments qui sont ajoutés à la file doivent être stockés dans un tableau (ou éventuellement d'autres structures simples pour stocker un ensemble d'éléments), et les cases des éléments sont gérés par un ou plusieurs indices. Par ailleurs, nous considérons pour commencer que des files dont les éléments sont des chaînes de caractères.

2. Une file simple mais très gourmande en mémoire

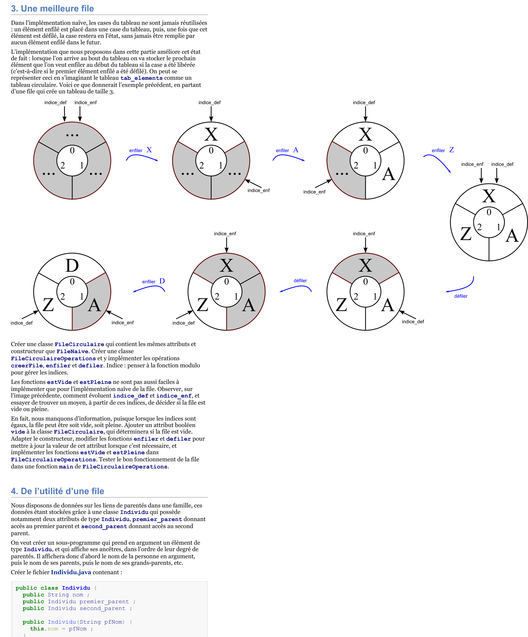
Nous nous placons d'abord dans le cas où nous disposons de beaucoup de mémoire, ce qui nous autorise à gérer le stockage des éléments de manière naïve mais simple.

Créer une classe **`FileNaive`**, contenant trois attributs :

- **`tab_elements`** : un tableau de chaines de caractères servant à stocker les éléments qui sont enfilés
- **`indice_enf`** : un entier représentant l'indice du tableau où le prochain élément enfilé sera stocké
- **`indice_def`** : un entier représentant l'indice du tableau du prochain élément défilé.

Créer également un constructeur prenant un seul argument, de type entier, cet entier décrivant le nombre total d'éléments qui pourront être enfilés dans cette file.

L'image suivante décrit comment ces attributs évoluent sur l'exemple précédent.



Créer une classe **`FileNaiveOperations`**, et y inclure les opérations de la file. Ajouter une fonction **`main`** pour vérifier le bon fonctionnement de la file, éventuellement sur l'exemple précédent.

L'utilisation de cette file reste limitée, puisque, quel que soit le nombre d'éléments que l'on défile, on ne pourra jamais enfiler plus d'éléments que le nombre utilisé dans le constructeur. On voudrait obtenir une implémentation plus efficace, de sorte, notamment, la file ne devienne pas pleine si on enfile, défile, enfile, défile, etc. L'implémentation de la partie suivante essaie de répondre à ces attentes.

3. Une meilleure file

Dans l'implémentation naïve, les cases du tableau ne sont jamais réutilisées : un élément enfilé est placé dans une case du tableau, puis, une fois que cet élément est défilé, la case restera en l'état, sans jamais être remplie par aucun élément enfilé dans le futur.

L'implémentation que nous proposons dans cette partie améliore cet état de fait : lorsque l'on arrive au bout du tableau on va stocker le prochain élément que l'on veut enfiler au début du tableau si la case a été libérée (c'est-à-dire si le premier élément enfilé a été défilé). On peut se représenter ceci en s'imaginant le tableau **`tab_elements`** comme un tableau circulaire. Voici ce que donnerait l'exemple précédent, en partant d'une file qui crée un tableau de taille 3.

Créer une classe **`FileCirculaire`** qui contient les mêmes attributs et constructeur que **`FileNaive`**. Créer une classe **`FileCirculaireOperations`** et y implémenter les opérations **`creerFile`**, **`enfiler`** et **`defiler`**. Indice : penser à la fonction modulo pour gérer les indices.

Les fonctions **`estVide`** et **`estPleine`** ne sont pas aussi faciles à implémenter que pour l'implémentation naïve de la file. Observer, sur l'image précédente, comment évoluent **`indice_def`** et **`indice_enf`**, et essayer de trouver un moyen, à partir de ces indices, de décider si la file est vide ou pleine.

En fait, nous manquons d'information, puisque lorsque les indices sont égaux, la file peut être soit vide, soit pleine. Ajouter un attribut booléen **`vide`** à la classe **`FileCirculaire`**, qui déterminera si la file est vide. Adapter le constructeur, modifier les fonctions **`enfiler`** et **`defiler`** pour mettre à jour la valeur de cet attribut lorsque c'est nécessaire, et implémenter les fonctions **`estVide`** et **`estPleine`** dans **`FileCirculaireOperations`**. Tester le bon fonctionnement de la file dans une fonction **`main`** de **`FileCirculaireOperations`**.

4. De l'utilité d'une file

Nous disposons de données sur les liens de parentés dans une famille, ces données étant stockées grâce à une classe **`Individu`** qui possède notamment deux attributs de type **`Individu`**, **`premier_parent`** donnant accès au premier parent et **`second_parent`** donnant accès au second parent.

On veut créer un sous-programme qui prend en argument un élément de type **`Individu`**, et qui affiche ses ancêtres, dans l'ordre de leur degré de parentés. Il affichera donc d'abord le nom de la personne en argument, puis le nom de ses parents, puis le nom de ses grands-parents, etc.

Créer le fichier **`Individu.java`** contenant :

```
public class Individu {
   public String nom;
   public Individu premier_parent ;
   public Individu second_parent ;

   public Individu(String pfNom) {
      this.nom = pfNom ;
   }

   public Individu(String pfNom, Individu pfp1,
Individu pfp2) {
      this.nom = pfNom ;
      this.premier_parent = pfp1 ;
      this.premier_parent = pfp2 ;
   }
}
```

Créer le fichier **`IndividuOperations.java`** contenant :

```
public class IndividuOperations {

   public static void afficherAncetres(Individu
pfIndividu) {

   }

   public static void main(String[] args) {
      Individu gmp = new Individu("Grand-mere
paternelle");
      Individu gmm = new Individu("Grand-mere
maternelle");
      Individu ggm = new Individu("Grand-pere
maternel");
      Individu gpp = new Individu("Grand-pere
paternel");
      Individu p = new Individu("Pere", gmp, ggp);
      Individu m = new Individu("Mere", gmm, ggm);
      Individu e = new Individu("Dernier", m, p) ;

      afficherAncetres(e);
   }
}
```

et compléter la fonction **`afficherAncetres`** pour afficher les ancêtres comme demandé.

Indice : créer une classe qui implémente une file dont les éléments sont des **`Individu`** (prendre n'importe quelle implémentation précédente, et remplacer les **`String`** par des **`Individu`**) et l'utiliser à bon escient.

5. Aller plus loin

5.1. Autres implémentations

Il n'est pas nécessaire de garder l'attribut booléen dans l'implémentation de la file.

5.1.1. En gardant le nombre d'éléments

Une première solution consiste à créer une classe dont les attributs sont :

- **`tab_elements`** : un tableau de chaines de caractères servant à stocker les éléments qui sont enfilés
- **`indice_def`** : un entier représentant l'indice du tableau du prochain élément défilé
- **`nb_elements`** : un entier représentant le nombre d'éléments dans la file
- **`max_nb_elements`** : un entier représentant le nombre d'éléments maximum dans la file (cet attribut n'est pas nécessaire si on s'autorise à utiliser **`tab_elements.length`**).

Implémenter cette file, avec toutes ses opérations.

5.1.2. En ne remplissant pas le tableau

Un seconde solution consiste à déclarer la file pleine avant que le tableau qui stocke les éléments ne soit plein (par exemple, lorsqu'une seule case n'est pas utilisée). La file aura donc pour attributs :

- **`tab_elements`** : un tableau de chaines de caractères servant à stocker les éléments qui sont enfilés
- **`indice_def`** : un entier représentant l'indice du tableau du prochain élément défilé
- **`indice_enf`** : un entier représentant l'indice du tableau où le prochain élément enfilé sera stocké
- **`nb_nb_elements`** : un entier représentant le nombre d'éléments maximum dans le tableau (cet attribut n'est pas nécessaire si on s'autorise à utiliser **`tab_elements.length`** et si on décrète que la pile est pleine lorsqu'il ne reste qu'une case inutilisée).

5.2. Une file qui s'agrandit

Malgré une meilleure gestion de la mémoire que la toute première implémentation de la file, nous nous exposons toujours au risque que la file devienne pleine si le nombre d'éléments toujours dans la file approche la taille réservée pour le tableau initial. Choisir une implémentation, ajouter l'opération **`agrandirFile`** qui :

- crée un nouveau tableau dont la taille est le double de celle de **`tab_elements`**
- recopie les éléments de **`tab_elements`** dans le nouveau tableau
- met à jour tous les attributs de la file.

Changer les opérations concernées pour agrandir le tableau lorsque la file est pleine.