TP 3 : gestion de tableaux redimensionnables

L'objectif de ce TP est d'effectuer l'implémentation de tableaux de taille variable, autrement dit, de tableaux dont le nombre d'éléments peut varier au cours du temps. Nous allons nous intéresser à trois implémentations différentes correspondant à plusieurs stratégies de gestion de tableaux de taille variable.

1 Spécification

```
// spécification du TA Tableau
// T est le type des éléments du tableau

public interface Tableau<T> {

  /** Déterminer la taille du tableau
   *  @return nombre d'éléments présents dans le tableau
   */
  public int size();

  /** Déterminer si le tableau est vide
   *  @return vrai si le tableau est vide
   */
  public boolean empty();

  /** Déterminer si le tableau est plein
   *  @return vrai s'il n'est plus possible d'ajouter d'élément dans le tableau
   */
  public boolean full();

  /** Renvoyer l'élément d'indice i
   *  @param i : indice de l'élément à consulter
   *  @pre 0 <= i < this.size()
   *  @return valeur de l'élément d'indice i
   */
  public T get(int i);

  /** Modifier l'élément d'indice i
   *  @param i : indice de l'élément à modifier
   *  @pre 0 <= i < this.size()
   *  @param v : nouvelle valeur de l'élément d'indice i
   */
  public void set(int i, T v);

  /** Ajouter un élément en fin de tableau
   *  @param x : élément à ajouter en fin de tableau
   *  @pre : ! this.full()
   */
  public void push_back(T x);

  /** Supprimer le dernier élément du tableau
   *  @pre : ! this.empty()
   */
  public void pop_back();
}
```

Dans nos implémentations de tableaux de taille variable nous distinguerons :

- la *capacité* du tableau : nombre de cases réservées pour gérer ce tableau ;
- la *taille* du tableau : le nombre de cases utilisées au sein du tableau ; ces cases utilisées seront obligatoirement les premières du tableau.

La taille (T) et la capacité (C) d'un tableau devront donc toujours vérifier les inégalités suivantes :
$0 < C \, et \, 0 \leq T \leq C$.

Remarques :

- on dira qu'un tableau est *plein* s'il n'est pas possible de lui ajouter de nouvel élément : toutes les cases sont occupées et la capacité est fixe ;
- l'ajout d'un nouvel élément dans un tableau ne peut se faire qu'en fin de tableau, avec la méthode *push_ back()* et uniquement si le tableau n'est pas plein ;

— les méthodes *get()* et *set()* ne peuvent s'appliquer que pour les éléments présents, donc pour les éléments d'indice i tel que $0 \leq i < size()$;
— la méthode *pop_ back()* sert à supprimer le dernier élément d'un tableau non vide.

2 Implémentation par tableau de capacité fixe : Block<T>

La classe Block<T> représentera un tableau de *capacité fixe* mais de *taille variable*. Cette classe devra disposer d'un ***constructeur prenant en paramètre la capacité du tableau*** (la taille initiale du tableau sera égale à 0).

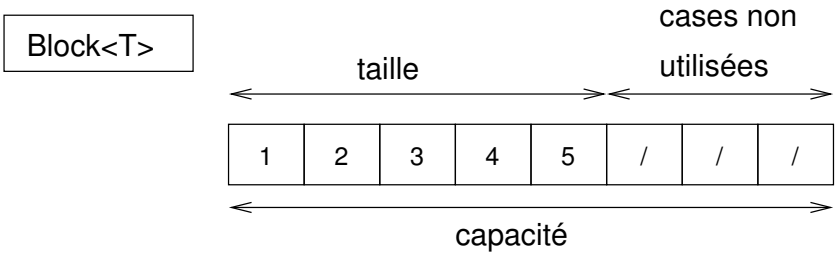

FIGURE 1 – Exemple d'une instance de Block<T> de taille 5 et de capacité 8.

1. Réalisez l'implémentation de la classe Block<T> dans le paquetage tableau. Pour le stockage des éléments, vous utiliserez la classe Array<T> qui vous est fournie dans la bibliothèque types.
2. **Testez votre implémentation** avec le programme de test unitaire fourni ; il suffira de mettre en commentaire les parties qui correspondent aux fonctionnalités non programmées.

3 Programme client à réaliser

Dans ce programme, vous allez calculer l'ensemble des entiers premiers compris dans l'intervalle [2 ; N], N étant fourni par l'utilisateur en suivant la méthode présentée ci-dessous. Ce programme client sera placé dans le paquetage main.

3.1 Calcul de nombres premiers

1. Écrivez une fonction booléenne *estPremier* qui détermine si un entier $n \geq 2$ est premier. Cette fonction prendra en paramètre l'entier n à tester et **un tableau contenant tous les entiers premiers compris dans l'intervalle** [2 ; n-1]. Elle renverra vrai si l'entier testé est premier, faux dans le cas contraire.
2. Écrivez une fonction *calculerNombresPremiers* qui calculera l'ensemble des entiers premiers dans l'intervalle [2 ; N] *à l'aide de la fonction estPremier*.
   Cette fonction prendra en paramètre l'entier N ainsi qu'un tableau, initialement vide, qui servira à stocker l'ensemble des entiers premiers trouvés dans cet intervalle.
   Dans la mesure où le tableau pourrait ne pas avoir une capacité suffisante pour stocker tous les entiers premiers, la fonction renverra le dernier entier testé si le tableau est plein avant la fin du calcul ou l'entier N dans le cas contraire.
3. Écrivez un programme principal qui crée un tableau de type Block d'une capacité de 100 éléments. Demandez à l'utilisateur d'entrer un entier N et calculez les entiers premiers dans l'intervalle [2 ; N]. Le programme affichera (à l'aide d'une fonction) les entiers premiers trouvés ainsi que le dernier entier testé. Testez votre programme.

3.2 Fonctionnalités complémentaires

1. Écrivez une fonction *remplirHasard* qui prend en paramètre un nombre entier *nb*, crée un tableau de type Block de capacité *nb* et le remplit avec des entiers tirés au hasard dans l'intervalle [0 ; nb[(voir la classe Random dans l'API java). La fonction renverra le tableau ainsi créé et initialisé.
   Complétez le programme principal précédent pour créer puis afficher un tableau de nombres entiers aléatoires dans l'intervalle [0 ; dernier[où *dernier* est le nombre renvoyé par la fonction *calculerNombresPremiers*.
2. Écrivez une fonction *eliminerPresents* qui prend en paramètre deux tableaux *t1* et *t2* de nombres entiers ; *t1* est quelconque et *t2* est trié en ordre croissant. Cette fonction éliminera du tableau *t1* tous les éléments présents dans le tableau *t2* ;
   — vous réfléchirez à une solution qui n'effectue aucun décalage des éléments de t1...
   — vous réfléchirez aussi à un découpage fonctionnel pertinent ainsi qu'à un algorithme performant...
   — la fonction devra renvoyer le nombre d'éléments éliminés.
   Complétez le programme principal pour éliminer tous les entiers premiers du tableau de nombres aléatoires ; le programme devra ensuite afficher le nombre d'entiers premiers éliminés puis le tableau de nombres aléatoires.

4 Tableau redimensionnable par doublement de la capacité : Tableau2x<T>

La classe Tableau2x<T> représentera un tableau de taille et de capacité variables. La gestion est la suivante : lorsque toutes les cases d'un Tableau2x sont occupées, le tableau n'est pas considéré comme plein mais l'ajout d'un nouvel élément est possible : il faut allouer un nouveau tableau d'une capacité double de celle du tableau actuel. Les éléments du tableau actuel seront recopiés dans le nouveau tableau et le nouvel élément pourra ainsi être ajouté (voir figure 2).

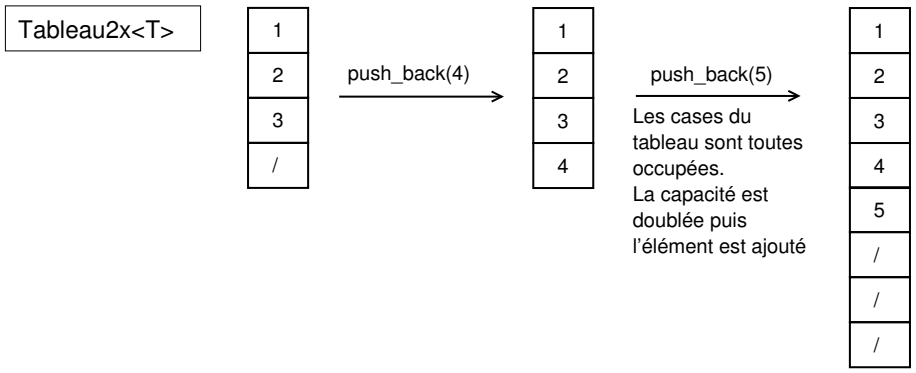

FIGURE 2 – Exemple d'ajout de valeurs dans un Tableau2x avec doublement de la capacité

La classe Tableau2x<T> possédera un ***constructeur prenant en paramètre la capacité initiale du tableau***.

Question : Pour le stockage des données dans un Tableau2x<T> vous avez maintenant le choix entre utiliser la classe Array<T> ou la classe Block<T> programmée à la question précédente. Laquelle de ces deux classes vous semble la plus adaptée ? Pourquoi ?

1. Programmez l'implémentation de la classe Tableau2x<T>.
2. Testez votre implémentation avec le programme de test unitaire donné.
3. Modifiez le programme client de calcul des nombres premiers afin d'utiliser votre implémentation de Tableau2x<T>.

5 Tableau redimensionnable de blocs : TableauBlock<T>

La classe TableauBlock<T> représentera elle aussi un tableau de taille et de capacité variables. Le principe de la représentation interne est le suivant : les éléments du TableauBlock<T> seront stockés dans des blocs de données de capacité fixée. Ces blocs seront eux-mêmes référencés dans un tableau de taille et de capacité variables (voir figure 3). Lors de l'ajout d'un élément, plusieurs cas pourront se produire :

— le bloc contenant les derniers éléments du TableauBlock<T> n'est pas plein : dans ce cas, l'élément est ajouté dans ce bloc ;

— le bloc contenant les derniers éléments du TableauBlock<T> est plein : dans ce cas, il faut créer un nouveau bloc, ajouter ce bloc dans le tableau de blocs et ajouter l'élément dans le bloc nouvellement créé.

La figure 3 présente les différents cas et la modification de la structure de donnée associée.

La classe TableauBlock<T> possèdera deux constructeurs :

— un constructeur prenant en paramètre la capacité initiale du tableau et la capacité des blocs utilisés pour le stockage des données.

— un constructeur prenant en paramètre la capacité initiale du tableau. Ce constructeur fixera alors une capacité de bloc par défaut de 128 éléments.

Question : Quel sera le type du tableau de blocs utilisé comme représentation interne de la classe TableauBlock<T> ?

Question : Connaissant la capacité des blocs et l'indice d'un élément à récupérer dans le TableauBlock, quelles sont les formules permettant de calculer l'indice du bloc contenant cet élément dans le tableau de blocs et l'indice de l'élément dans ce bloc ?

1. Implémentez la classe TableauBlock<T>.
2. Testez votre implémentation avec le programme de test unitaire donné.
3. Modifiez le programme client de calcul des nombres premiers afin d'utiliser votre implémentation de TableauBlock<T>.

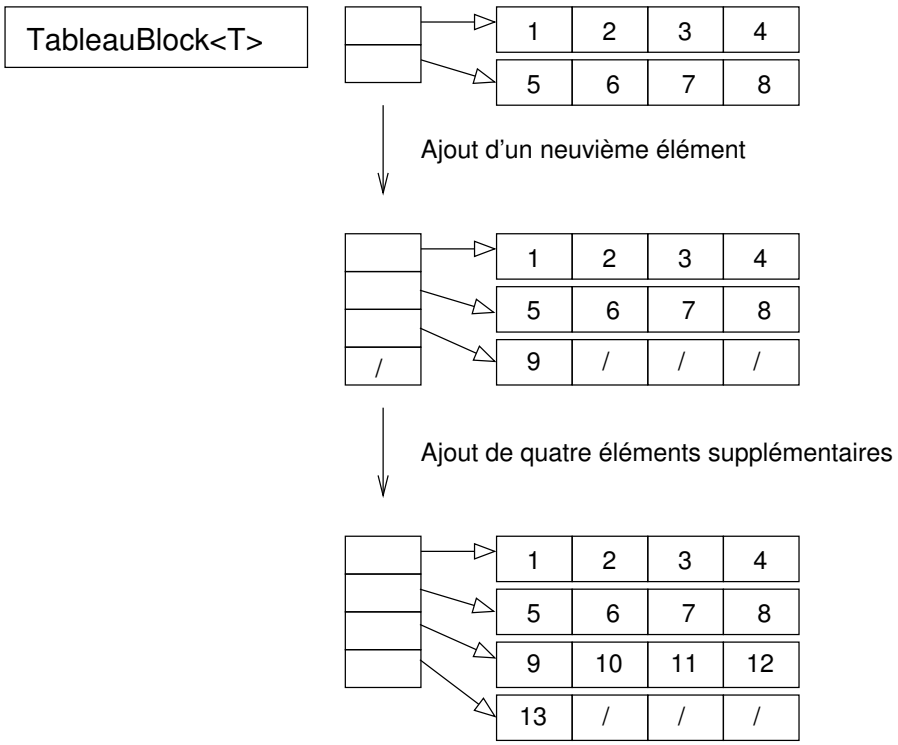

FIGURE 3 – Exemple d'ajouts d'élément dans un TableauBlock<T>