TP/TD 3-4

1 Introduction

Le but de ce TP est de créer des structures de données représentant un dictionnaire. Un dictionnaire stockera simplement des couples (mots, définitions), (aussi appelés (clé, valeur)), un mot et une définition étant tous les deux représentés par des *string*. Les seules fonctions accessibles depuis l'extérieur pour un dictionnaire sont les suivantes :

- *put(key, value)*, qui insère le mot *key* et sa définition *value* dans le dictionnaire
- *get(key)*, qui retourne la définition du mot *key*. On redéfinira également l'opérateur [ ] pour cela, dans le but d'écrire des choses du type *dico["mot"]* pour retrouver une définition.
- *size()* qui retourne le nombre d'entrées du dictionnaire.

On veut pouvoir utiliser deux types de dictionnaires différents : un type *OrderedDictionnary* et un autre *FastDictionnary*. Dans un soucis de factorisation de code, tous les deux hériteront de la classe abstraite *AbstractDictionnary*. Ces dictionnaires fonctionneront à l'aide d'un tableau, cependant, la manière de gérer ce tableau sera différente selon le type de dictionnaire. On veut les spécificités suivantes :

- dans un *OrderedDictionnary*, les entrées sont stockées séquentiellement, et on doit parcourir tout le tableau pour retrouver une entrée. On optimisera cependant l'espace mémoire de façon à avoir des tableaux les plus pleins possibles.
- dans un *FastDictionnary*, on pourra se permettre d'utiliser plus de place que nécessaire, mais on optimisera le temps pour retrouver une entrée. On utilisera pour cela le principe d'une table de hachage.

Pour plus de lisibilité, on renommera le type *string* de cette façon:

typedef string Key
typedef string Value

2 La classe *AbstractDictionnary*

Cette classe possède les attributs *protected* suivants :

- *Key* arrayKey* : un tableau de *Key* qui stockera les clés (mots) de chaque entrée
- *Value* arrayValue* : un tableau de *Value* qui stockera les valeurs (définintions) de chaque entrée. Le mot qui se situe à l'indice i du tableau arrayKey correspond à la définition qui se situe à l'indice i du tableau arrayValue.
- *arrayLength* : on est obligé de stocker la taille de nos tableaux séparément en C++

Elle possède un constructeur *AbstractDictionnary(int length)* qui initialisera les tableaux, et un destructeur *~ AbstractDictionnary()*, qui ne fera rien.
Elle possède les méthodes publiques suivantes :

- *Value get(Key key)* qui retourne la définition associée au mot *key*, ou une chaine vide sinon.
- *void put(Key key, Value value)* qui insère la nouvelle entrée si la clé n'existe pas, ou écrase l'ancienne définition sinon.
- *Value operator[](Key key)* qui surcharge l'opérateur [ ]. L'opérateur agit comme la méthode *get*.

Afin de factoriser le plus de code possible, on a également les méthodes *abstraites* et *protected* suivantes :

- *int indexOf(Key key)* qui retourne l'index du tableau où se trouve la clé *key*
- *int newIndexOf(Key key)* qui retourne l'index du tableau où insérer un nouvel élément.
- *int size()* qui retourne le nombre d'entrée du dictionnaire.

Exercice 1
Ecrire le constructeur *AbstractDictionnary* et les méthodes *get, put* et l'opérateur [ ].

3 La classe *OrderedDictionnary*

Une instance de *OrderedDictionnary* sera initialisée à partir d'une capacité initiale *length*. On peut insérer les entrées les unes à la suites des autres dans le tableau tant qu'on ne dépasse pas la capacité. Le cas échéant, on crée deux nouveaux tableaux (pour les clés et pour les entrées) qui comportent une case de plus que l'ancien, on recopie ces derniers, et on ajoute la nouvelle entrée. On délocalisera cette manipulation dans la méthode privée *void grow()*. On aura également besoin d'un attribut privé *int numElements* qui stocke le nombre d'entrées du dictionnaire.

Exercice 2

Ecrire les méthodes suivantes :

- *int newIndexOf(Key key)*
- *int indexOf(Key key)*
- *void grow()*

4 La classe *FastDictionnary*

Une instance de *FastDictionnary* aura ses tableaux initialisés à une capacité *FastDictionnary::INIT_LENGTH* (attribut statique, exemple : 4). On insère ensuite des éléments tant que le ratio entre le nombre d'entrées et la capacité du tableau ne dépasse pas *FastDictionnary::MAX_RATIO* (exemple : 3/4). Le cas échéant, on augmente la taille des tableaux de telle sorte à avoir un ratio de *FastDictionnary::WANTED_RATIO* (exemple : 1/4), et on recopie les anciennes données. Le test pour savoir s'il est nécessaire d'augmenter la taille des tableaux lors d'un ajout sera délocalisé dans la méthode privée *bool mustGrow()*, et l'augmentation des tableaux ainsi que la recopie seront délocalisées dans la méthode privée *void grow()*. Pour insérer un nouvel élément, on utilisera une fonction de hachage (donnée, à placer en méthode statique de *FastDictionnary*) qui, étant donné la clé (le mot à insérer), retournera un entier qui sera l'index de ce nouvel élément (modulo la taille du tableau). Si deux clés ont le même hashcode, on prend la première case vide qui suit.

Exercice 3

Ecrire les méthodes suivantes :

- *int newIndexOf(Key key)*
- *int indexOf(Key key)*
- *int size()*
- *bool mustGrow()*
- *void grow()*