Dictionnaire

François Hallereau
Benjamin Sientzoff

25 avril 2014

Table des matières

Introduction

Ce projet consiste à implémenter un dictionnaire (structure de données non ordonnée avec unicité). Cette implémentation a été réalisé de deux manières différentes.

L'une utilise une table de hachage dans laquelle on stocke les mots. L'autre est un arbre où sont stocké un à un les caractères des mots à sauvegarder.

Le projet devra aussi contenir un code permettant de tester le bon fonctionnement des deux implémentations.

Pour compiler notre projet, la commande suivante est à utiliser :

```
$ g++ -std=c++0x application.cpp application
```

Puis pour l'exécution :

```
$ ./application <le_nom_du_fichier_a_lire>
```

1 Le *dictionhache*

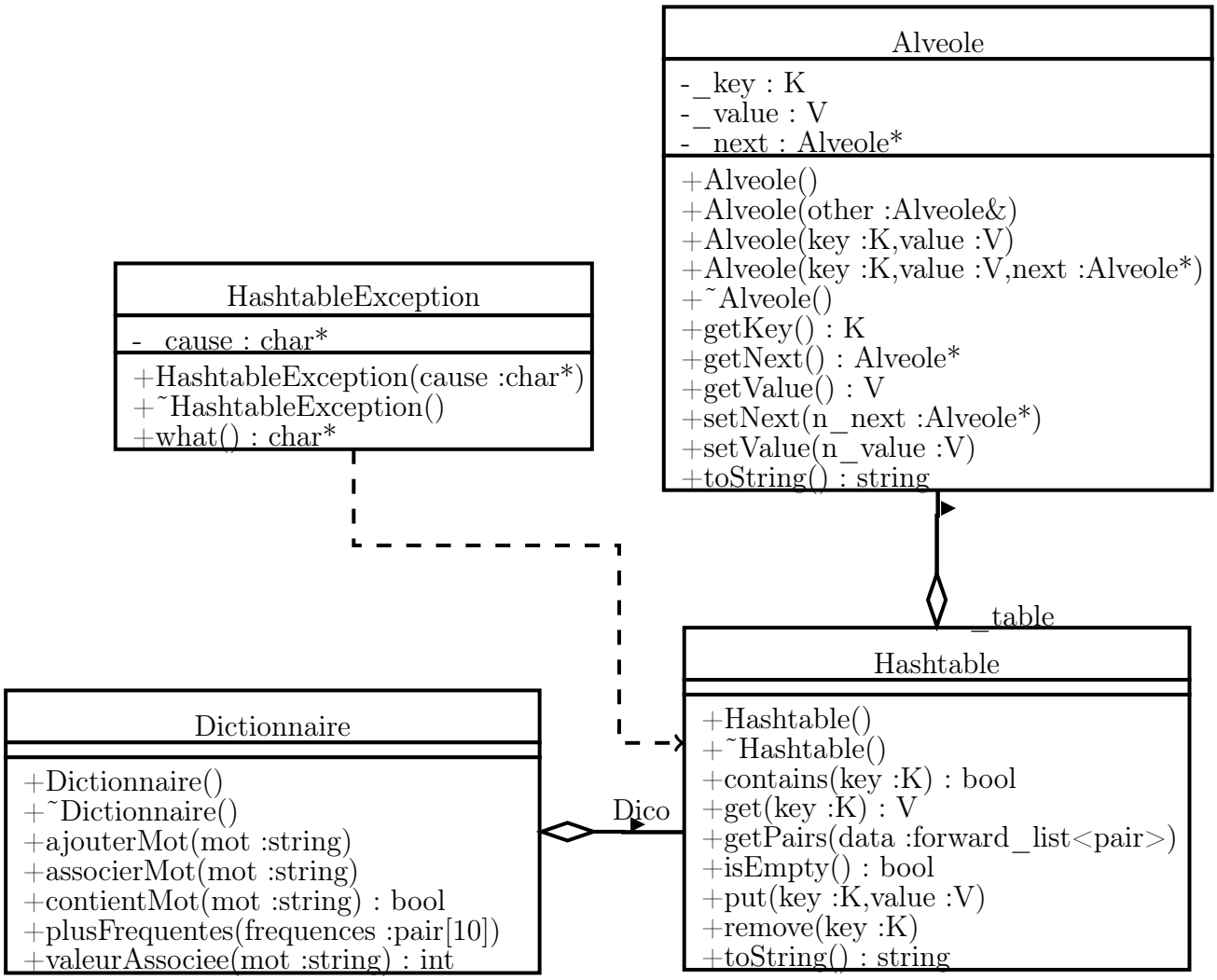

FIGURE 1 – Diagramme UML de classes pour le dictionnaire utilisant la *Hashtable*.

Cette version reprend la table de hachage du précédent projet tout en précisant les types de données qui sont ici : une chaîne de caractères pour la clé et un entier pour la valeur associée au mot. Ce qui permet de stocker la fréquence du mot en question.
Pour cette implémentation, les fonctions d'ajout et de modification reprennent

les fonctions déjà existantes de la table de hachage à la différence que lors de la modification, la valeur associée passé en paramètre est la valeur actuellement associé au mot que l'on incrémente. Une documentation est fournit au format PDF pour plus de précision.

La fonction d'ajout n'a aucune précondition cependant, si on ajoute un mot déjà présent alors l'occurrence du mot est incrémenté. La fonction de modification quant à elle a pour précondition que le mot passé en paramètre est déjà présent. Ces deux fonctions ont pour postcondition : le mot est présent dans le dictionnaire avec l'occurrence qui est à jour.

La fonction qui renvoie les dix mots les plus fréquents tri les mots de la table de hachage puis récupère les dix premiers éléments. La fonction de tri appliquée sur une liste à une complexité en $n \log(n)$. Cette fonction n'a pas de précondition et a pour postcondition : un tableau contenant au maximum dix mots correspondant au mots les plus fréquents dans le texte. Donc la fonction peut être utilisées même lorsque moins de dix mots sont stockés.

2 L'*Arbramots*

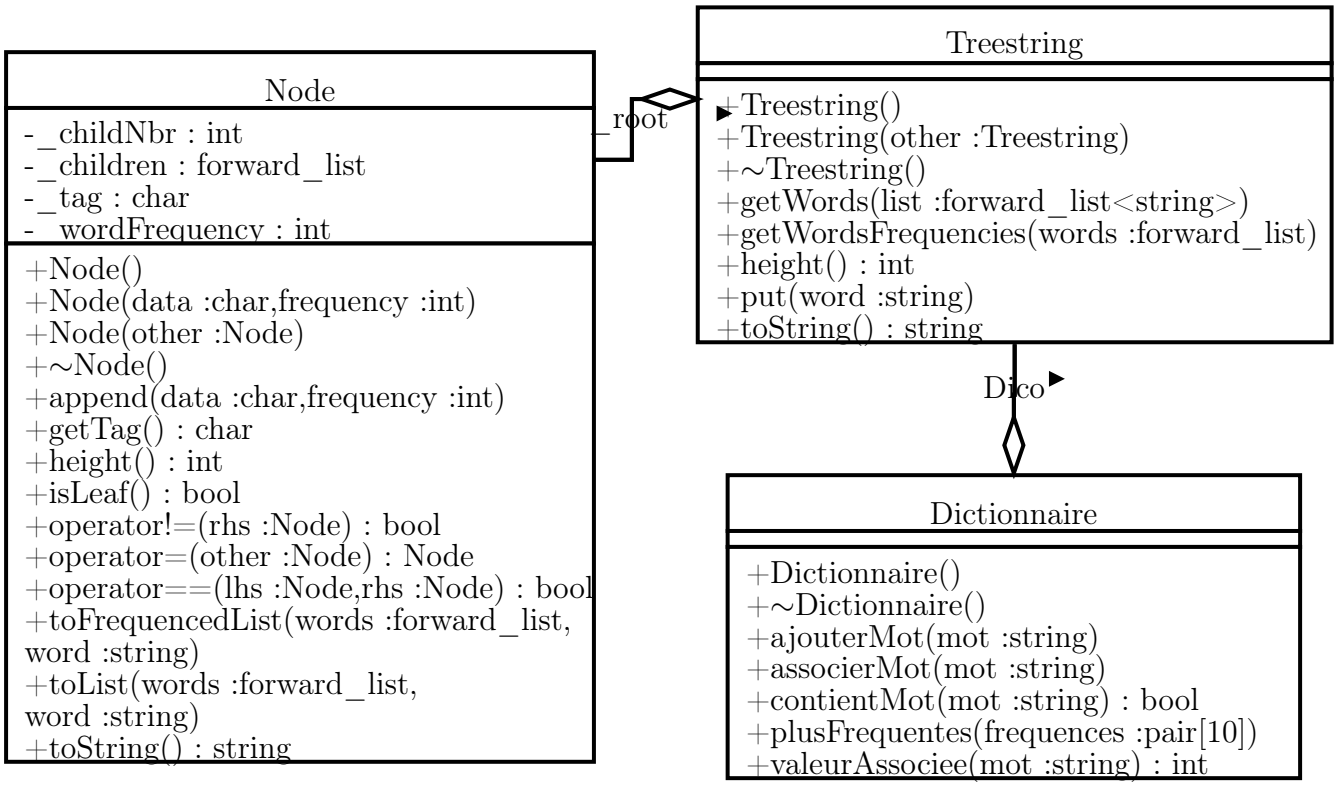

FIGURE 2 – Diagramme UML de classes pour le dictionnaire utilisant un arbre pour stocker les caractères des mots.

Le dictionnaire utilise la version de l'arbre implémenté dans la classe *TreeString*. Un diagramme de classe à la figure 2 présente comment est implémenté l'arbre.

Les fonctions d'ajout et de modification de mots dans l'arbre sont similaire. Elle repose toutes deux sur la fonction *put* de la classe *TreeString*. Lorsqu'elle est invoquée, elle met le mot passé en paramètre dans l'arbre avec une fréquence à 1. Si le mot est déjà présent, elle incrémente la fréquence associé au mot.

La fonction *valeurAssociee* fonctionne de la manière suivante : elle récupère

tous les couples $< mot, frequence >$ contenu dans l'arbre puis renvoie la valeur associée du mot passé en paramètre.

C'est deux fonctions ont les mêmes préconditions que celle de la table de hachage, bien qu'en pratique la fonction de modification fonctionne si le mot passé en paramètre n'existe pas. En effet, il sera alors ajouté. De même pour les postconditions qui sont les mêmes que pour la version avec la table de hachage.

Pour la fonction qui récupère les dix mots les plus fréquents, elle est semblable à celle de la table de hachage. La seule différence étant la fonction qui permet de récupérer tous les mots contenus dans la structure. La encore les pré et postconditions sont les mêmes que pour la table de hachage.

3 Tests et remarques

Lors des tests, nous avons put remarquer que le temps d'exécution était beaucoup plus important lorsqu'on utilise l'arbre comparé à la table de hachage.

Et pour cause, la fonction *contientMot* de l'arbre utilise plus de ressource que celle de la table de hachage. De plus, l'arbre a été conçu pour qu'il n'y est pas besoin d'utiliser cette fonction à chaque ajout.

On aurait donc très bien put n'utiliser que la fonction *ajouterMot* même dans le cas ou le mot est déjà présent. Mais nous avons décider de garder les fonctions *ajouterMot* et *associerMot* dans un soucis du respect du cahier des charges.

Conclusion

Dans ce dernier projet de l'année, nous avons put appliquer les dernières notions vu en cours telle que la structure de données sous forme d'arbre ainsi que la complexité et les encombrements mémoires.

Par ailleurs, on remarque que la table de hachage à un temps d'exécution inférieur à celui de l'arbre. En effet la table de hachage manipule directement des mots or l'arbre utilise des caractères qui nécessite d'être concaténé pour reformer les mots stockés. Il en est de même pour l'espace mémoire.

Pour que la structure d'arbre soit plus efficace que la table de hachage, il aurait fallut mettre les mots dans un arbre binaire équilibré, par exemple. L'un des meilleurs critères de tri utilisable est alors la fréquence du mot. Cependant, il faut alors garder l'arbre équilibré lors d'ajouts des mots.