Devoir 1 : Les tarbres

On se propose d'implémenter une structure de dictionnaire, sous la forme d'un arbre binaire de recherche que nous allons équilibré grâce à un algorithme randomisé (*i.e.* qui fait appel au hasard). La structure concrète porte le nom de *tarbre*, contraction de *tas* et *arbre*.

Nous utiliserons ensuite cette structure pour compter le nombre de mots d'un fichier texte et trouver le mot de longueur au moins 6 apparaissant le plus souvent.

1 Rendu

Le projet se déroule sur les trois premières séances de TP en monôme ou en binôme. À leur issue et à la date prescrite (selon votre campus), vous devrez rendre sur Ametice, sous forme d'un unique fichier archive d'extension `gz` ou `tar.gz` **exclusivement**, contenant l'ensemble de vos sources et un rapport décrivant comment utiliser votre projet, vos résultats, et le cas échéant une description des fonctionnalités non-implémentées. Si votre projet contient du code source que vous n'avez pas écrit vous-même, vous indiquerez précisément sa nature et sa provenance. Tout projet contenant du code non-correctement attribué à ses auteurs s'expose à recevoir une note nulle.

La date de rendu dépend de votre campus, demandez à votre chargé de TP.

2 Définition des tarbres

2.1 Introduction

Un *tarbre* est un arbre binaire de recherche tel que chaque nœud possède trois informations :

- une clé,
- une valeur associée à cette clé,
- une *priorité* de type entier associée à cette clé,

et qui vérifie les invariants suivants :

- les clés vérifient l'invariant des arbres binaires de recherche : pour tout nœud n, la clé associée à n est supérieure à toute clé dans le sous-arbre gauche de n et inférieure à toute clé dans le sous-arbre droit de n,
- les priorités vérifient l'invariant des tas minimum : la priorité de tout nœud est inférieure ou égale à la priorité de ses fils.

Le nom de cette structure provient donc du fait qu'elle vérifie ces deux invariants simultanément. Il s'agit néanmoins bien de l'implémentation d'un dictionnaire dans notre cas (la même idée pourrait permettre d'écrire une structure concrète de tas, en tirant

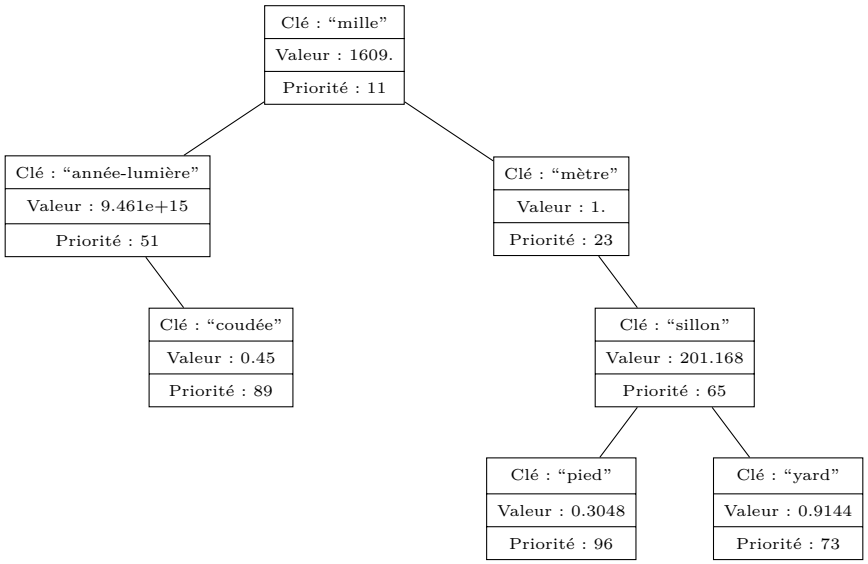

FIGURE 1 – Exemple de tarbre, avec pour clés des chaînes de caractères et pour valeur des flottants.

aléatoirement les clés, mais ce n'est pas le sujet de ce devoir). La Figure 1 montre un exemple de tarbre.

Il existe plusieurs manières d'implémenter les tarbres, une possibilité est d'utiliser des rotations pour préserver les invariants. Nous proposons d'utiliser une méthode différente que nous décrivons.

Les deux principales fonctions à programmer sont :

— l'opération split qui coupe un tarbre en deux selon une clé k. Ceci retourne deux tarbres, l'un contenant toutes les clés strictement plus petites que k, l'autre toutes les clés strictement plus grandes. Si k était présente dans le tarbre, elle ne le sera dans aucun des tarbres retournés. Un exemple est donné en appliquant split sur le tarbre de la Figure 1 avec la clé “pouce”, donnant les tarbres de la Figure 2.

— L'autre opération merge est son inverse, étant donné deux tarbres, toutes les clés du premier étant inférieures aux clés du deuxième, construire le tarbre union des deux. Un merge des tarbres de la Figure 2 donnerait donc la Figure 1.

Une fois ces deux fonctions programmées, les autres opérations sont élémentaires.

Puisque nous utilisons une structure d'arbre, donc inductive, les algorithmes que nous écrirons seront naturellement récursifs. Nous éviterons donc d'utiliser des boucles **for** ou **while**.

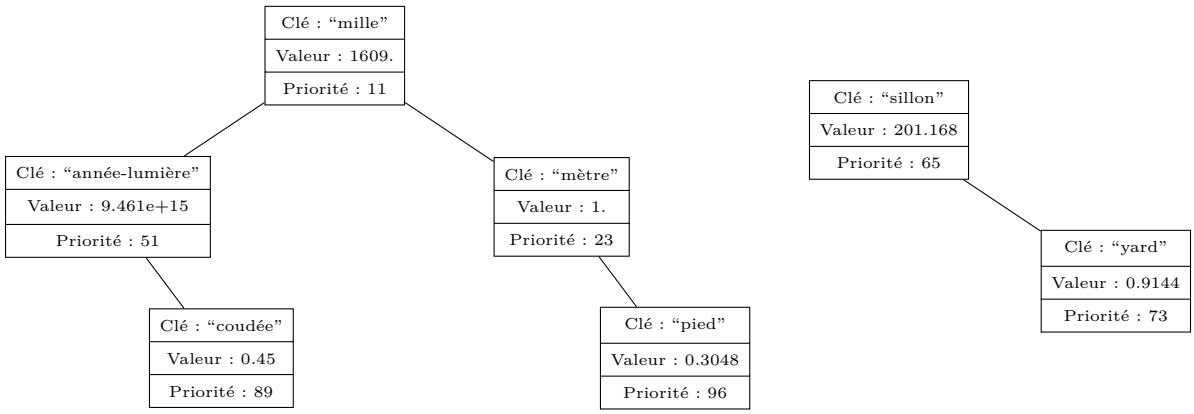

FIGURE 2 – Après l'opération split pour la chaîne “pouce”.

2.2 L'opération split

Pour couper un tarbre (fils_gauche, r, fils_droit) en deux selon une clé k, on compare la clé de la racine avec k. Si k est plus petite, on coupe récursivement le fils gauche en deux tarbres t_1, t_2, et on retourne t_1 et (t_2, r, fils_droit).

Remarquons alors que l'invariant de tas et l'invariant d'arbre binaire de recherche sont bien préservés par cette opération.

Si k est plus grande que la clé en racine, c'est fils_droit qui est coupé en deux récursivement en t_1, t_2, et le résultat est alors (fils_gauche, r, t_1) et t_2.

Enfin si les clés sont égales, on retourne simplement fils_gauche et fils_droit

2.3 L'opération merge

Cette opération doit fusionner deux tarbres t_1 et t_2 (toutes les clés de t_1 étant inférieures aux clés de t_2), il faut donc une racine, qui ne peut que être le nœud de plus petite priorité, par l'invariant des tas. On compare donc les priorités des racines t_1 et t_2.

Les deux cas sont symétriques, supposons que la priorité de la racine de t_1 est la plus petite des deux. Dans ce cas, le fils gauche de t_1 devient le fils gauche du tarbre résultat, et le fils droit de t_1 est récursivement fusionné avec t_2 pour former le fils droit du résultat.

2.4 L'opération insert

Une fois split codé, il est possible de réaliser l'insertion. Pour insérer une nouvelle clé k, on commence par générer une priorité p en choisissant un entier aléatoire dans un intervalle le plus grand possible. Ensuite, on effectue une descente dans l'arbre de recherche comme si on cherchait la clé k. La descente termine :

— soit parce qu'on atteint un nœud de priorité plus grande que p. C'est ici que k va être inséré. Pour cela, on utilise split pour couper le sous-tarbre courant par rapport à k, et on remplace ce sous-tarbre par le tarbre formé de k et ayant pour fils les tarbres retournés par split. On aura ainsi dans le fils gauche les clés plus petite que k, et dans le fils droit les clés plus grandes.
— soit parce qu'on atteint une feuille auquel cas on y insère k.

2.5 L'opération remove

Pour supprimer une clé, on opère une descente dans le tarbre comme pour une recherche de la clé, et une fois celle-ci trouvée, on remplace le sous-tarbre correspondant par la fusion de ses deux fils.

2.6 Les fonctions de recherche

Il faut aussi implémenter les fonctions de test de la présence d'une clé, et d'obtention de la valeur associée à une clé. Pour ces deux fonctions, on ignore les priorités et on utilise l'algorithme classique pour les arbres binaires de recherche.

3 Application

Afin de valider votre structure, nous vous demandons de réaliser une petite application, consistant à lire un texte sur l'entrée standard, pour ensuite écrire sur la sortie standard :

— le nombre de mots *distincts* lus (il faudra penser à les mettre tous en minuscule),
— le mot d'au moins 6 lettres le plus utilisé dans le texte.

Vous lirez donc le texte mot à mot en ajoutant ces mots dans un tarbre, avec comme valeur associée le nombre d'occurrences déjà repertoriées. Attention, ce texte en Français, comporte des caractères accentués.

Le texte utilisé est le premier tome du *Comte de Monte Cristo*, à récupérer sur le site du projet Gutenberg (lien sur Ametice).

S'il vous reste du temps, vous pouvez comparer l'efficacité de votre structure avec celles proposées par Java (parmi celles implémentant l'interface Map) :

— ConcurrentSkipListMap (basé sur les skip lists),
— TreeMap (basé sur les arbres rouge-noirs),
— HashMap (basé sur les tables de dispersion).

4 Implémentation

Voici quelques indications pour l'implémentation en Java.

— Utilisez la généricité pour coder les tarbres, c'est un cadre idéal pour cela. Les tarbres sont génériques par rapport à leurs clés de classe `Key` et par rapport aux valeurs associées aux clés, de classe `Assoc`.

```
public class Tarbre<Key extends Comparable<Key>,Assoc> { ... }
```

On vous donne le code en Figure 3, à titre d'exemple de classe générique. Vous pouvez commencer sans généricité, en supposant que les clés sont des `String` par exemple, puis rendre les classes génériques plus tard.

— La classe Tarbre doit utiliser une classe Node contenant 5 champs (clé, priorité, valeur associée, et les deux fils). Chaque instance contient un champ root contenant un objet Node. Par commodité, on peut regrouper les trois champs clé, priorité et valeur associée à l'aide d'une autre classe.

— split doit retourner deux tarbres. Il vous faudra donc coder une classe paire.

— Utilisez des constructeurs bien conçus pour ne pas rendre votre code trop verbeux. Pensez à factoriser votre code dès que possible, comme d'habitude.

— Pour l'application, vous aurez besoin d'un algorithme qui parcourt le tarbre. Si vous disposez de la version 8 de Java, c'est le moment d'apprendre à utiliser les fonctionnelles. Récupérez la fonction forEach définie en Figure 4. Consultez la documentation d'Oracle sur les lambda-expressions pour pouvoir ensuite vous en servir.

— Pour lire le fichier, vous supposerez qu'il est donné en entrée standard, et utiliserez la classe Scanner. Vous pouvez ensuite itérer sur ce scanner, ce qui permet d'obtenir les mots un par un. Consultez la documentation de cette classe sur le site d'Oracle pour savoir comment faire.

```
Pattern delim =
    Pattern.compile("[\\W_]",Pattern.UNICODE_CHARACTER_CLASS);
Scanner sc = new Scanner(System.in).useDelimiter(delim);
```

```java
public class Pair<F,S> {
    private final F fst;
    private final S snd;

    public Pair(F fst, S snd) {
        this.fst = fst;
        this.snd = snd;
    }
    public F fst() { return fst; }
    public S snd() { return snd; }
}
```

FIGURE 3 – La classe générique Pair

```java
    import java.util.function.*;

    // adaptez selon vos classes
    public void forEach(BiConsumer<Key,Assoc> consumer) {
        if (root == null) return;
        root.leftChild.forEach(consumer);
        consumer.accept(root.key,root.assoc);
        root.rightChild.forEach(consumer);
    }


    // le code pour ensuite compter le nombre de clefs dans le tarbre
    final Counter count = new Counter(); // vous savez faire un compteur
    dictionary.forEach( (key,assoc) -> count.increment() );
    return count.getValue();
```

FIGURE 4 – Utilisation des lambda-expressions en Java.