Rapport Algorithmique Avancée
Devoir de programmation : Tries

Dat NGUYEN
Tomohiro ISHIWATA

December 6, 2015

Table des matières

1 Introduction

Le but de ce projet consiste en la représentation d'un dictionnaire de mots. Dans cette optique nous avons dû implémenter deux structures de tries, les comparer entre elles par une étude expérimentale, et ainsi identifier les avantages et inconvénients de chacuns de ces modèles pour enfin conclure sur quelle structure de données est la mieux adaptée pour représenter notre problème.

2 Arbres de la Briandais

Definition Un arbre de la Briandais est un arbre binaire contenant sur chacuns de ses noeuds un caractère, un fils et un frère. Pour représenter la fin d'un mot nous utiliserons le caractère ϵ.

2.1 Structure de données

Nous avons représenter cette structure en Java par la classe BRDtree contenue dans le fichier src/BRDtree.java dans notre code source. Nous présentons ici uniquement les attributs de la classe.

```
1 public class BRDtree {
2     protected Character key;
3     protected BRDtree child;
4     protected BRDtree next;
5 }
```

key représente le caractère contenu dans le noeud , child représente la référence vers le fils et next représente la référence vers le frère. On encode le caractère ϵ avec le caractère '\0'.

2.2 Primitives de base

Nos primitives de base sont contenues dans le fichier src/BRDprimitives.java, voici un tableau récapitulatif de ces fonctions avec leurs complexités.
On note m le nombre de caractères contenu dans le mot word, N le nombre de lettres de l'alphabet de notre dictionnaire et M le nombre total de caractères contenu dans une liste de mots.

noms	rôles	complexités
emptyBRD()	construit un arbre vide	O(1)
isEmpty(T)	verifie si T est vide	O(1)
buildBRD(word)	construit un arbre à partir du mot word de taille m	O(m)
addBRD(T,word)	ajoute à l'arbre T le mot word	O(N*m)
addBRD(T,lw)	ajoute la liste de mots lw à l'arbre t	O(N*M)

Nous ne détaillerons ici que la fonction d'ajout, les autres fonctions étant assez simple à comprendre.
Voici le pseudo code de cette fonction:

Algorithm 1 Ajouter un mot

```
1: procedure ADDBRD(T,WORD)
2:     if T=Nil then
3:         if word=ϵ then return emptyBRD()
4:         elsereturn buildBRD(word)
5:     if word = ϵ then
6:         if T.key = ϵ then return T
7:         elsereturn new BRDtree(ϵ, Nil, T)
8:         if word.head < T.key then return new BRDtree(word.head, build-
   BRD(word.tail), T)
9:         else if word.head > T.key then return new BRDtree(T.key,
   T.child, addBRD(T.next,word))
10:        else return new BRDtree(T.key, addBRD(T.child,word.tail),
   T.next)
```

complexité On parcourt au pire tous les noeuds de l'arbre (on note N se nombre), puis pour chaque noeud on compare au pire le nombre de caractères du mot (on note m ce nombre), on a donc au final une complexité en O(N*m).

2.3 Fonctions avancées

Ces fonctions sont contenues dans le fichier src/BRDadvancedFunctions.java, voici un tableau récapitulatif de ces fonctions avec leurs complexités.
On note m le nombre de caractères contenu dans le mot word, N le nombre de lettres de l'alphabet de notre dictionnaire et M le nombre total de caractères contenu dans une liste de mots.

noms	rôles	complexités
search(T,w)	construit un arbre vide	O(1)
wordCount(T)	compte le nombre de mots dans T	O(N)
wordList(T)	renvoie la liste des mots contenus dans T	O(N)
nilCount(T)	renvoie le nombre de pointeur null dans T	O(N)
heigth(T)	renvoie la hauteur de T	O(N)
averageDepth(T)	renvoie la hauteur moyenne de T	O(N)
prefix(T,word)	renvoie le nombre de mots préfixe de word dans T	O(N*m)
delete(T,word)	renvoie T privé du mot word	O(N*m)

Nous détaillerons ici les algorithmes des fonctions de recherche et de supression d'un mot. Nous vous laissons vous référez au fichier src/BRDadvanced-Functions.java pour les autres fonctions.

Algorithm 2 Rechercher un mot

```
1: procedure SEARCH(T,WORD)
2:     if estVide(word) then return faux
3:     if T=Nil then return false
4:     if T.key = ϵ then
5:         if word.head = ϵ then return true
6:         elsereturn search(T.next,word)
7:     if word.head = T.key then return search(T.child, word.tail)
8:     if word.head < T.key then return search(T.next, word)
       return false
```

complexité Au pire on parcourt tous les noeuds de l'arbre et pour chaque noend on compare au pire le nombre de caractères du mot donc on a au final O(N*m).

Algorithm 3 supprimer un mot

```
1: procedure DELETE(T,WORD)
2:     if T=Nil then return Nil
3:     if T.key = word.head then
4:         child ← delete(T.child, word.tail)
5:         if child = Nil thenreturn t.next
6:         elsereturn new BRDtree(T.key, child, T.next)
7:     elsereturn new BRDtree(T.key,T.child, delete(T.next,word))
```

complexité Comme précédemment, au pire on parcourt tous les noeuds de l'arbre et pour chaque noend on compare au pire le nombre de caractères du mot donc on a au final O(N*m).

2.4 Fonctions complexes

Ces fonctions sont contenues dans le fichier src/BRDcomplexeFunctions.java, voici un tableau récapitulatif de ces fonctions avec leurs complexités.
On note m le nombre de caractères contenu dans le mot word et N le nombre de lettres de l'alphabet de notre dictionnaire.

noms	rôles	complexités
brdToHybrid(T)	convertit un BRDtree en HBDtree	O(N)
fusion(T1,T2)	renvoie un arbre résultant de la fusion de T1 et T2	O(N1+N2)

Pour la fusion on serait tenté de lister le mots des deux arbres puis d'ajouter successivement chaque mot de la liste dans un arbre, mais cette méthode n'est pas efficace en terme de complexité.
Nous proposons donc le pseudocode suivant pour la fusion.

Algorithm 4 fusionner deux arbres de la Briandais

1: **procedure** FUSION(T1,T2)
2: **if** T1=Nil **thenreturn** T2
3: **if** T2=Nil **thenreturn** T1
4: **if** $T1.key < T2.key$ **then return** new BRDtree(T1.key,T1.child,fusion(T1.next,T2))
5: **else if** $T1.key > T2.key$ **then return** new BRDtree(T2.key,T2.child,fusion(T1,T2.next))
6: **elsereturn** new BRDtree(T1.key,T2.fusion(T1.child,T1.child),fusion(T1.next,T2.next))

complexité La fusion se faisant en parallèle, au pire on fait le parcourt de tous les noeuds des deux arbres donc si on note N1 ce nombre pour l'arbre T1 et N2 pour celui de T2 on aurra du O(N1+N2).

La complexité de la fonction wordCount est en O(N) car dans tous les cas il faut parcourir tous les noeuds de l'arbre pour compter les mots, idem pour les fonctions heigth, averageDepth, nilCount, wordList.

3 Trie Hybride

Definition Un trie hybride est un arbre ternaire contenant deux caractères, un représentant le caractère contenu dans le noeud et un autre qui jouera le rôle de booléen pour signalé si le noeud représente la fin d'un mot ou non.
Chaque noeud a 3 pointeurs: un lien Inf (respectivement Eq , respectivement Sup) vers le sous arbre des clés dont le premier caractère est inférieur (respectivement égal, respectivement supérieur) à son propre caractère.

3.1 Structure de données

Nous avons représenter cette structure en Java par la classe HBDtree contenue dans le fichier src/HBDtree.java.

```
1 public class HBDtree {
2     protected Character car;
3     protected Character val;
4     protected HBDtree inf;
5     protected HBDtree eq;
6     protected HBDtree sup;
7 }
```

le caractère car représente la clé du noeud, val prend la valeur '0' si le noeud ne représente pas la fin d'un mot et '1' dans le cas contraire, puis les références inf, eq et sup représente respectivement les fils inférieur, égal et supérieur.

3.2 Primitives de base

Nos primitives sont contenues dans le fichier src/HBDprimitives.java, voici un tableau récapitulatif de ces fonctions avec leurs complexités.
On note m le nombre de caractères contenu dans le mot word, N correspond au

nombre de lettres de l'alphabet de notre dictionnaire et M est le nombre total de caractères contenu dans une liste de mots.

noms	rôles	complexités
buidHBD(word)	construit un arbre à partir du mot word de taille m	O(m)
addHBD(T,word)	ajoute à T le mot word	O(N*m)
addHBD(T,lw)	ajoute la liste de mots lw à l'arbre t	O(N*M)

Algorithm 5 Ajouter un mot

```
1:  procedure ADDHBD(T,WORD)
2:      if T=Nil then return buildHBD(word)
3:      if word.head < T.car then
4:          T.inf ← addHBD(T.inf, word)
5:      else if word.head > T.car then
6:          T.snf ← addHBD(T.snf, word)
7:      else
8:          if m.length = 1 then
9:              T.val ← NOTEMPTY
10:         else
11:             T.eq ← addHBD(T.eq, word.tail)
```

complexité Comme pour l'arbre de la Briandais on a du O(N*m)

3.3 Fonctions avancées

Ces fonctions sont contenues dans le fichier src/HBDadvancedFunctions.java, voici un tableau récapitulatif de ces fonctions avec leurs complexités.
On note m le nombre de caractères contenu dans le mot word, N correspond au nombre de lettres de l'alphabet de notre dictionnaire et M est le nombre total de caractères contenu dans une liste de mots.

noms	rôles	complexités
search(T,w)	construit un arbre vide	O(1)
wordCount(T)	compte le nombre de mots dans T	O(N)
wordList(T)	renvoie la liste des mots contenus dans T	O(N)
nilCount(T)	renvoie le nombre de pointeur null dans T	O(N)
heigth(T)	renvoie la hauteur de T	O(N)
averageDepth(T)	renvoie la hauteur moyenne de T	O(N)
prefix(T,word)	renvoie le nombre de mots préfixe de word dans T	O(N*m)
delete(T,word)	renvoie T privé du mot word	O(N*m)

3.4 Fonctions complexes

Ces fonctions sont contenues dans le fichier src/HBDcomplexeFunctions.java, voici un tableau récapitulatif de ces fonctions avec leurs complexités.
On note N le nombre de lettres de l'alphabet de notre dictionnaire.

noms	rôles	complexités
hybridToBRD(T)	convertit un HBDtree en BRDtree	O(N)
rotationDroite(T)	renvoie un arbre résultant de la rotation droite de T	O(1)
rotationGauche(T)	renvoie un arbre résultant de la rotation gauche de T	O(1)
equilibrage(T)	renvoie un arbre résultant de l'équilibrage de T si besoin	O(N)

équilibrage Pour équilibrer notre arbre nous avons au début pensé à lister les mots. A partir de cette liste, nous insérons le mot du milieu puis les mots supérieurs, puis inférieurs. Cette fonction marche mais a une complexité trop élevée. Nous nous sommes donc inspirés de l'équilibrage des arbres AVL, notre structure pouvant être considéré comme binaire car le fils du milieu n'intervient pas dans la vérification de l'équilibrage. Sur ce principe nous avons mis au point des fonctions de rotations, une vers la gauche (rotationGauche) et l'autre vers la droite (rotationDroite) qui nous permettent d'équilibrer notre arbre si besoin. Voici le pseudocode de la fonction d'équilibrage.

Algorithm 6 Equilibrage d'un trie hybride

```
1:  procedure EQUILIBRAGE(T)
2:      if T=Nil then return Nil
3:      d ← heigth(T.inf) − heigth(T.sup)
4:      if d ≥ 2 then
5:          dinf ← heigth(T.inf.inf) − heigth(T.inf.sup)
6:          if dinf ≤ −1 then
7:              T.inf ← rotationGauche(T.inf)
8:          else
9:              T ← rotationDroite(T)
10:     else if d ≤ −2 then
11:         dsup ← heigth(T.sup.inf) − heigth(T.sup.sup)
12:         if dsup ≥ 1 then
13:             T.sup ← rotationDroite(T.sup)
14:         else
15:             T ← rotationGauche(T)
        return T
```

complexité Etant donné qu'on calcul la hauteur à chaque appel récursif, notre fonction d'équilibrage a une complexité au pire de O(N*N).

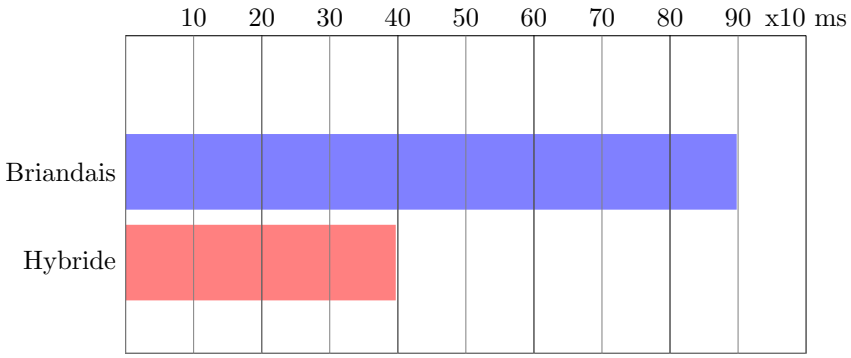
4 Etude expérimentale

Nous effectuons dans cette partie une suite de comparaisons entre les deux structures afin de déterminer celle la mieux adaptée pour modéliser notre problème. La construction de ces structures se fait à partir de l'ensemble des mots de l'oeuvre de Shakespeare qui nous a été fourni.

4.1 Compraisons des structures

4.1.1 Ajouts successifs

On considère ici que l'on dispose d'une liste de l'ensemble des mots de l'oeuvre, et on effectue des ajouts succéssifs (appel à la fonction addBRD) jusqu'à obtenir les deux arbres finaux.

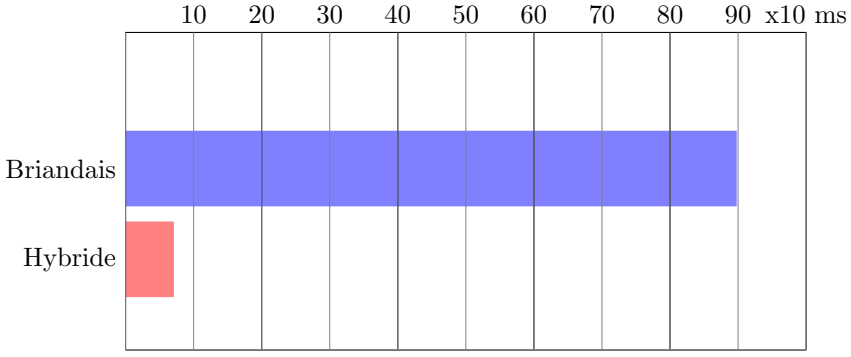


Temps d'ajout de tous les mots en ms x10

D'après le schéma précédent, le temps de construction par ajouts successifs de mots est beaucoup moins coûteux en temps pour l'arbre hybride.

4.1.2 fusions successives

On considère ici que l'on dispose d'une liste d'arbres déjà construits pour chacunes des oeuvres de Shakespeare, la comparaison se fait uniquement entre le temps de construction des arbres de la Briandais car nous n'avons pas implémenté la fonction de fusion de deux tries hybrides.

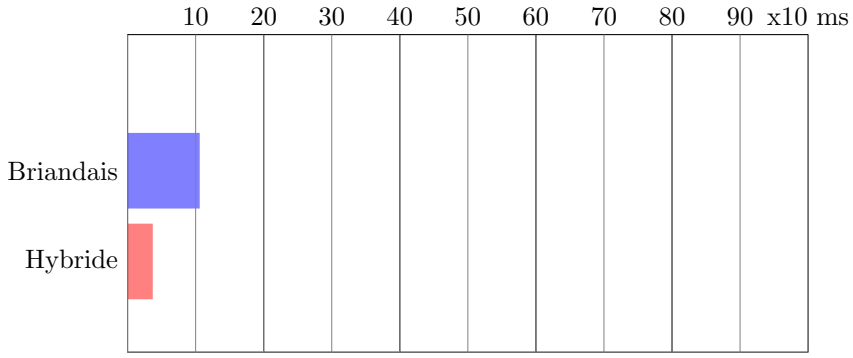


Temps de fusion de tous les arbres en ms x10

Si l'on considere qu'on a un ensemble d'arbre représentant chacun une des oeuvre, le temps de construction de l'arbre contenant tous les mots est fortement minimisé. Ce qui est tout à fait logique car l'algorithm de fusion change les pointeurs des fils et frères de l'arbre en parallèle plutôt que de parcourir tous les noeuds des deux arbres.

4.1.3 Supression d'une oeuvre

On part ici d'un arbre contenant tous les mots de l'oeuvre de Shakespeare sur lequel on fait la comparaisons de la supression d'une oeuvre.

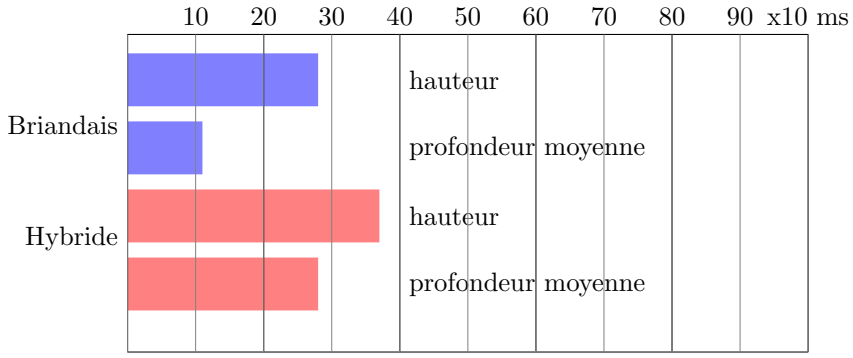


Temps de supression de l'oeuvre john en ms x10

Le temps de supression d'une oeuvre est minimisé pour le trie hybride.

4.1.4 Hauteur et profondeur moyenne

On part ici de deux arbres contenant tous les mots de l'oeuvre de Shakespeare sur lesquels on calcule la hauteur et la profondeur moyenne pour chacunes des structures.



Temps de supression de l'oeuvre john en ms x10

On compare ici les hauteurs et les profondeurs moyennes des deux structures. L'histogramme plus haut nous permet d'émettre une hypothèse quant à l'espace occupé par chacunes des deux structures.
Nous parlons d'hypothèse car, ayant codé en Java, nous n'avons pas d'outils précis nous indiquant l'espace occupé par chaques objets comme, par exemple, en C avec la fonction sizeof.
On pourrait donc penser que l'arbre de la Briandais occupe moins d'espace mémoire que le trie hybride car la profondeur de celui-ci étant plus grande.

5 Conclusion

Pour une même quantité de données à traiter, nous pencherons plus vers l'utilisation d'une structure de type trie hybride plutôt qu'un arbre de la Briandais car les performance en temps son biens meilleures.