Algorithmique Avancée
Devoir de programmation

Maxime Bittan - Redha Gouicem

2014

Chapitre 1

Introduction

La représentation d'un grand ensemble de données peut vite être très coûteuse en espace mémoire. Ce coût est d'autant plus important lorsque cet ensemble contient de nombreuses répétitions. Il devient alors primordial de trouver une structure de données permettant de diminuer un maximum cette perte d'espace inutile dûe au fait que les données sont représentées à plusieurs reprises en mémoire. De plus, il peut être intéressant, dans certains cas, que plusieurs éléments de cet ensembe de données partagent une partie de leurs données. De plus, il est important de conserver de bonnes performances en terme de temps d'exécution.

Dans ce devoir de programmation, nous allons étudier deux structures de données permettant de représenter un dictionnaire de mots (représentés sur 8 bits, en codage ASCII). Une des particularités de ce type de données tient au fait que de nombreux mots partagent des préfixes communs. Il parait alors intéressant d'essayer d'exploiter cette propriété pour réduire l'empreinte mémoire du dictionnaire et, par la même occasion, améliorer les performances en terme de temps. Les deux structures en question sont les arbres (ou tries) de la Briandais et les tries hybrides.

Définition 1 (Arbre de la Briandais)**.** *Un arbre (ou trie) de la Briandais est un arbre binaire. Chaque noeud contient un caractère, un frère et un fils. Le fils correspond au caractère suivant et le frère correspond à un caractère alternatif pour la même position dans le mot. De plus, on définit un caractère ϵ représentant une fin de mot.*

Définition 2 (Trie hybride)**.** *Un trie hybride est un arbre ternaire de recherche. Chaque noeud contient un caractère et trois fils qui sont eux mêmes des tries hybrides : un fils gauche contenant un caractère inférieur à celui du noeud, un fils droit contenant un caractère supérieur à celui du noeud, et un fils central contenant le caractère suivant du mot. Chaque noeud contient également un marqueur de fin de mot, si ce noeud correspond à la dernière lettre d'un mot.*

Chapitre 2

Arbres de la Briandais

2.1 Structure

Pour implanter cette structure de données en C, nous avons défini la structure suivante dans *include/briandais.h* :

```
typedef struct _briandais_ {
  char key;
  struct _briandais_ *son;
  struct _briandais_ *brother;
  int cpt;
} briandais_t;
```

Le compteur `cpt` représente le nombre de mots pour lesquels ce noeuds est utilisé, et sera utilisé lors de la suppression (voir 2.3.1).
Le caractère ϵ choisi est '\0' car il s'agit, dans de nombreux langagues, dont le C, du caractère de fin de chaîne, ce qui permet d'éviter l'ajout d'un caractère en fin de chaîne lors d'une insertion dans l'arbre. De plus, ce caractère n'est pas imprimable, et ne portera donc pas confusion lors de l'affichage des mots de l'arbre.

2.2 Primitives

Soient :

— $\mathcal{A}$ l'alphabet utilisé dans le dictionnaire,

— L la longueur de la clé,

— si T est un arbre, alors $|T|$ est le nombre de noeuds de T.

Fonction	Emplacement	Complexité
new_briandais(c, S, B)	src/briandais.c, $l5$	$\Theta(1)$
Renvoie un arbre de la Briandais contenant `c` *avec comme fils* `S` *et comme frère* `B`*.*		
new_empty_briandais()	src/briandais.c, $l16$	$\Theta(1)$
Renvoie un arbre de la Briandais contenant `'\0'` *avec comme fils* `nil` *et comme frère* `nil`*.*		
is_empty_briandais(T)	src/briandais.c, $l20$	$\Theta(1)$
Renvoie `vrai` *si* `T` *a* `'\0'` *comme caractère et comme fils et frère* `nil`*.*		
insert_briandais(T, mot)	src/briandais.c, $l24$	$O(L \times \lvert\mathcal{A}\rvert)$
Renvoie `T` *avec* `mot` *inséré.*		
destroy_briandais(T)	src/briandais.c, $l95$	$\Theta(\lvert T\rvert)$
Renvoie `T` *avec* `mot` *inséré.*		

Complexités Les complexités sont exprimées en nombre de comparaisons de caractères. Les fonctions `new_briandais`, `new_empty_briandais` et `is_empty_briandais` sont en temps constant puisque l'on ne parcourt qu'un seul noeud de l'arbre avec au plus une unique comparaison. Pour `insert_briandais`, on parcourt `L` noeuds vers le bas (vers les fils) et à chaque niveau, on parcourt au plus $|\mathcal{A}|$ noeuds, d'où cette complexité en $O(L \times |\mathcal{A}|)$. Quant à `destroy_briandais`, il s'agit d'un parcours complet de l'arbre, d'où cette complexité en $\Theta(|T|)$.

2.3 Fonctions avancées

Fonction	Emplacement	Complexité
delete_briandais(T, mot)	src/briandais.c, $l54$	$O(L \times \lvert\mathcal{A}\rvert)$
search_briandais(T, mot)	src/briandais.c, $l133$	$O(L \times \lvert\mathcal{A}\rvert)$
count_briandais(T)	src/briandais.c, $l150$	$\Theta(\lvert T\rvert)$
count_null_briandais(T)	src/briandais.c, $l158$	$\Theta(\lvert T\rvert)$
list_briandais(T)	src/briandais.c, $l180$	$\Theta(\lvert T\rvert)$
height_briandais(T)	src/briandais.c, $l190$	$\Theta(\lvert T\rvert)$
average_depth_briandais(T)	src/briandais.c, $l199$	$\Theta(\lvert T\rvert)$
prefix_briandais(T, mot)	src/briandais.c, $l215$	$O(L \times \lvert\mathcal{A}\rvert)$

2.3.1 Complexités

Suppression Le principe de l'algorithme récursif de suppression se déroule en deux étapes :

— On parcourt l'arbre comme pour une recherche : si le mot à supprimer n'est pas dans l'arbre, on s'arrête, sinon,

— à partir du noeud contenant ϵ représentant la fin du mot à supprimer, on remonte jusqu'à la racine en décrémentant le compteur de chaque noeud faisant parti du mot. Si le compteur vaut 0, on détruit le noeud en prenant soin de rattacher un éventuel frère au père ou au frère précédent.

Cet unique parcours explique cette complexité en $O(L \times |\mathcal{A}|)$, identique à celle de la recherche ou du préfixe.

Algorithm 1 Suppression Arbre de la Briandais

```
1: fonction SUPPRESSION(T, mot)
2:     ▷ On va utiliser la valeur de retour de la fonction pour dire à son père quoi faire
3:     si T = nil alors
4:         retourner -1                                   ▷ mot n'est pas dans l'arbre
5:     sinon si T.cle = '\0' et mot[0] = '\0' alors
6:         retourner 1                                    ▷ Détruis moi
7:     sinon si T.cle = mot[0] alors
8:         r := Suppression(T.fils, mot[1 :])
9:         si r = 1 alors
10:            T.cpt := T.cpt-1
11:            s := T.fils
12:            T.fils := s.frere
13:            Detruire(s)
14:            si T.cpt = 0 alors
15:                retourner 1                            ▷ Détruis moi
16:            fin si
17:        sinon si r = 0 alors
18:            T.cpt := T.cpt-1
19:        fin si
20:        retourner 0                                    ▷ Ne me détruis pas, mot est dans T
21:    sinon si T.cle < mot[0] alors
22:        r := Suppression(T.frere, mot)
23:        si r = 1 alors
24:            b := T.frere
25:            T.frere := b.frere
26:            Detruire(b)
27:        fin si
28:        retourner 0                                    ▷ Ne me détruis pas, mot est dans T
29:    fin si
30:    retourner -1                                       ▷ mot n'est pas dans l'arbre
31: fin fonction
```

2.4 Fonctions complexes

Fonction	**Emplacement**	**Complexité**
merge_briandais(T, U)	src/briandais.c, *l*275	$\Theta(\|T\| + \|U\|)$
convert_to_hybrid(T)	src/briandais.c, *l*286	$\Theta(\|T\|)$

Complexités La fonction de conversion correspond à un simple parcours de l'arbre, d'où cette complexité en $\Theta(|T|)$. La fusion, quant à elle, est fait en place. On parcourt donc les deux arbres "en même temps", en rattachant les parties non communes des deux arbres et en détruisant d'un des deux arbres les parties communes (pour éviter les fuites mémoire). Dans le pire des cas, les deux arbres n'ont rien en commun, on parcourt donc les deux arbres entièrement. De plus, on doit refaire un parcous de l'arbre fusionné pour mettre à jour les compteurs de chaque noeud. On obtient donc cette complexité en $\Theta(|T| + |U|)$. Ci-dessous, l'algorithme de fusion ne contient que la partie récursive, sans le parcours de mise à jour des compteurs effectué a posteriori.

Algorithm 2 Fusion Arbres de la Briandais

```
1:  fonction FUSION(T, U)
2:      si T = nil alors
3:          retourner U
4:      sinon si U = nil alors
5:          retourner T
6:      fin si
7:      si T.cle > U.cle alors
8:          U.frere := Fusion(T, U.frere)
9:          retourner U
10:     sinon si T.cle = U.cle alors
11:         T.frere := Fusion(T.frere, U.frere)
12:         T.fils := Fusion(T.fils, U.fils)
13:         Detruire(U)
14:     sinon
15:         T.frere := Fusion(T.frere, U)
16:     fin si
17:     retourner T
18: fin fonction
```

Chapitre 3

Tries hybrides

3.1 Structure

Pour implanter les tries hybrides en C, nous avons défini la structure suivante dans *include/structureTrieHybride.h* :

```
typedef struct _trie_hybride {
  char c;   //Caractere de la racine
  char fin; //Vaut 1 si le noeud represente la fin d'un mot
  struct _trie_hybride * inferieur;
  struct _trie_hybride * egal;
  struct _trie_hybride * superieur;
} TrieHybride;
```

3.2 Primitives

Soient :

— L la longueur de la clé,

— n est le nombre de mots insérés.

Fonctions	**Emplacement**	**Complexité**
trie_hybride	src/trieHybride_primitives.c, $l4-17$	$\Theta(1)$
Fonction permettant de créer un trie hybride.		
free_trie_hybride	src/trieHybride_primitives.c, $l20-27$	$\Theta(n)$
Libère la mémoire associée au trie hybride.		
racine	src/trieHybride_primitives.c, $l30-32$	$\Theta(1)$
Renvoie le caractère présent à la racine du trie hybride.		
inferieur	src/trieHybride_primitives.c, $l35-37$	$\Theta(1)$
Renvoie le sous-arbre qui représente tout les mots qui commencent par une lettre inférieure à celle présente à la racine.		
egal	src/trieHybride_primitives.c, $l39-41$	$\Theta(1)$
Renvoie le sous-arbre qui représente tout les mots qui commencent par la lettre qui se trouve à la racine.		
superieur	src/trieHybride_primitives.c, $l43-45$	$\Theta(1)$
Renvoie le sous-arbre qui représente tout les mots qui commencent par une lettre supérieure à celle présente à la racine.		
est_trie_vide	src/trieHybride_primitives.c, $l48-50$	$\Theta(1)$
Renvoie vrai si le trie est vide		
creer_mot	src/trieHybride_primitives.c, $l53-66$	$\Theta(L)$
Créé le trie hybride représentant le mot passé en paramètre.		
ajouter_trie_hybride	src/trieHybride_primitives.c, $l175-205$	$\Theta(log(n)+L)$

Complexités Insertion (idem pour recherche) : Si le trie est plus ou moins équilibré, on divise par trois l'espace dans lequel on doit chercher le mot à chaque appel récursif. On a donc l'équation de récurrence suivante :

$$T(n) = T\left(\frac{n}{3}\right) + \Theta(1)$$

D'après la deuxième règle du théorème maître ($a = 1, b = 3$), on a $T(n) = log_3(n)$. Sachant que l'on parcourt forcément toute la chaîne qui représente le mot, il faut prendre ceci en compte dans le calcul de la complexité. On a donc la complexité suivante : $log_3(n) + L$, avec L la longueur du mot que l'on cherche (ou insère).
Dans le cas où le trie est totalement déséquilibré, on a une complexité proche de celle d'un arbre de la Briandais. En effet, le trie hybride le plus déséquilibré est le trie qui n'a pas de fils gauche (ou droit). C'est le cas où tout les mots ont été insérés dans l'ordre alphabétique (ou l'inverse de l'ordre alphabétique). Chaque noeud a alors deux fils, le premier représentant les mots commençant par la lettre contenue dans le noeud, et le second représentant les mots commençant par une autre lettre.

3.3 Fonctions avancées

Fonctions	Emplacement	Complexité
recherche_trie_hybride	src/trieHybride_simple.c, $l7-36$	$\Theta(log(n)+L)$
comptage_mots	src/trieHybride_simple.c, $l39-52$	$\Theta(n)$
afficher_liste_mots	src/trieHybride_simple.c, $l58-76$	$\Theta(n)$
liste_mots	src/trieHybride_simple.c, $l81-101$	$\Theta(n)$
comptage_nil	src/trieHybride_simple.c, $l106-116$	$\Theta(n)$
hauteur	src/trieHybride_simple.c, $l130-140$	$\Theta(n)$
profondeur_moyenne	src/trieHybride_simple.c, $l145-154$	$\Theta(n)$
prefixe	src/trieHybride_simple.c, $l159-187$	$O(n)$
supprimer	src/trieHybride_simple.c, $l257-297$	$\Theta(log(n))$

3.3.1 Complexités

Comptage_mots Pour ces fonctions, dans tout les cas on parcourt l'ensemble de l'arbre. Si l'arbre est équilibré, on a l'équation de récurrence suivante :

$$T(n) = 3T\left(\frac{n}{3}\right) + \Theta(1)$$

D'après le cas 1 du théorème maître, le résultat de cette équation est :

$$T(n) = n^{log_3 3} = n \text{ avec } a = 3, b = 3, \epsilon = 1$$

Même principe pour afficher_liste_mots, liste_mots, hauteur, profondeur_moyenne et comptage_nil.

Suppression Principe : On parcourt d'abord le trie pour trouver le noeud qui correspond à la fin du mot que l'on veut supprimer. Si le mot n'est pas présent, le trie reste inchangé. Une fois que l'on a trouvé ce noeud, on indique que celui-ci ne représente plus la fin d'un mot. Si ce noeud ne possède pas de fils *egal*, cela veut dire qu'il n'y a aucun mot dans l'arbre prefixé par celui que l'on veut supprimer. On peut donc supprimer ce noeud. Trois cas peuvent alors se produire :

1. Le noeud n'a aucun fils. On peut alors le supprimer sans se soucier de ses fils.
2. Le noeud a un seul fils (soit le gauche, soit le droit). Dans ce cas, on peut remplacer le noeud à supprimer par son fils.
3. Le noeud a deux fils (gauche et droit donc). Il faut alors trouver un remplaçant pour le noeud que l'on va supprimer. Le bon candidat est le minimum (le fils le plus a gauche, qui n'a pas de fils gauche) du fils droit, comme on le ferait dans un ABR. En effet, dans ce cas, on a un noeud qui contient un caractère plus grand que tout les éléments du sous-arbre gauche, et plus petit que tout ceux du sous-arbre droit. Cela nous permet de garder les propriétés du trie hybride. De plus, comme le minimum n'a pas de fils gauche, on peut le remplacer par son fils droit sans problème.

Une fois le noeud supprimé, il est possible que l'on doive supprimer le père de celui-ci. On effectue la suppression si la branche du milieu du père est vide, et que le père ne représente pas la fin d'un mot.

Algorithm 3 Suppression Trie Hybride

```
1: fonction SUPPRESSION(T, mot)
2:     si T est vide alors
3:         retourner T
4:     fin si
5:     si lg(mot) = 1 et premier(mot) = racine(T) alors     ▷ Noeud de la fin du mot
6:         T.fin ← faux                                     ▷ Le noeud ne représente plus la fin du mot
7:         si egal(T) est vide alors
8:             retourner supprimer_racine(T)
9:         sinon
10:            retourner T
11:        fin si
12:    sinon si premier(mot) < racine(T) alors
13:        T.inferieur = suppression(T.inferieur, mot)
14:    sinon si premier(mot) > racine(T) alors
15:        T.superieur = suppression(T.superieur, mot)
16:    sinon
17:        T.egal = suppression(T.egal, reste(mot))
18:        si T.egal est vide et T.fin = faux alors
19:            retourner supprimer_racine(T)
20:        sinon
21:            retourner T
22:        fin si
23:    fin si
24:    retourner T
25: fin fonction
```

La fonction supprimer_racine est la fonction qui permet de gérer les trois cas énoncés précédemment.

La fonction extraire_min est une fonction qui permet de récupérer le minimum d'un trie. Le père du minimum reçoit alors le fils droit de celui-ci. Le minimum que l'on récupère avec cette fonction n'a donc ni fils gauche, ni fils droit.

3.4 Fonctions complexes

Fonctions	**Emplacement**	**Complexité**
conversion	src/trieHybride_complexe.c, $l4 - 49$	$\Theta(n)$
equilibrer	src/trieHybride_complexe.c, $l70 - 101$	$O(n^2)$

Algorithm 4 supprimer_racine

```
1: fonction SUPPRIMER_RACINE(T)
2:     si T.inferieur est vide et T.superieur est vide alors
3:         liberer T
4:         retourner Trie vide
5:     sinon si T.inferieur est vide alors
6:         T ← T.superieur                    ▷ On met l'arbre de droite a la place de T
7:     sinon si T.superieur est vide alors
8:         T ← T.inferieur                    ▷ On met l'arbre de gauche a la place de T
9:     sinon                                  ▷ Les sous-arbre droit et gauche ne sont pas vide
10:        (T.superieur, min) ← extraire_min(T.superieur)
11:        T ← min
12:    fin si
13:    retourner T
14: fin fonction
```

Principe : La fonction d'équilibrage fonctionne dans le même esprit que celle des AVL. Dans un premier temps on appelle récursivement la fonction sur les trois sous-arbres. Ensuite, on calcule la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit. Si cette différence est plus grande que deux, on effectue une rotation droite de l'arbre. De plus, si les deux fils du fils gauche ont une différence de hauteur plus grande (en valeur absolue) que 1, on effectue une rotation gauche sur ce fils. Si la différence de hauteur est plus petite que 2 (en valeur absolue), on effectue le symétrique des opérations précédentes (i.e. les rotations gauches deviennent des rotations droites et inversement).

Les fonctions de rotations sont quasiment identiques à celles présentées en cours. Leur complexité est donc $\Theta(1)$. La complexité de la fonction d'équilibrage est $O(n^2)$, car pour chaque noeud on calcule la hauteur de ses fils droit et gauche. On pourrait améliorer la complexité en ajoutant un champ *hauteur* dans la structure, tenu à jour à chaque insertion/suppression.

Algorithm 5 Equilibrage Trie Hybride

```
1: fonction EQUILIBRAGE(T)
2:     si T est vide alors
3:         retourner T
4:     fin si
5:     T.inferieur ← equilibrage(T.inferieur)
6:     T.egal ← equilibrage(T.egal)
7:     T.superieur ← equilibrage(T.superieur)
8:     diff ← hauteur(t.inferieur) − hauteur(t.superieur)
9:     si diff >= 2 alors
10:        si hauteur(t.inferieur.inferieur) − hauteur(t.inferieur.superieur) <= −1
   alors
11:            T.inferieur = rotG(T.inferieur)
12:        fin si
13:        T = rotD(T)
14:    sinon si diff <= −2 alors
15:        si hauteur(t.superieur.inferieur) − hauteur(t.superieur.superieur) >= 1
   alors
16:            T.superieur = rotD(T.superieur)
17:        fin si
18:        T = rotG(T)
19:    fin si
20:    retourner T
21: fin fonction
```

Chapitre 4

Comparaison

Pour comparer ces deux structures de données, nous prenons comme jeu de test 38 oeuvres de Shakespeare.

4.1 Temps d'exécution

Dans un premier temps, nous allons procéder à cinq tests de performances :

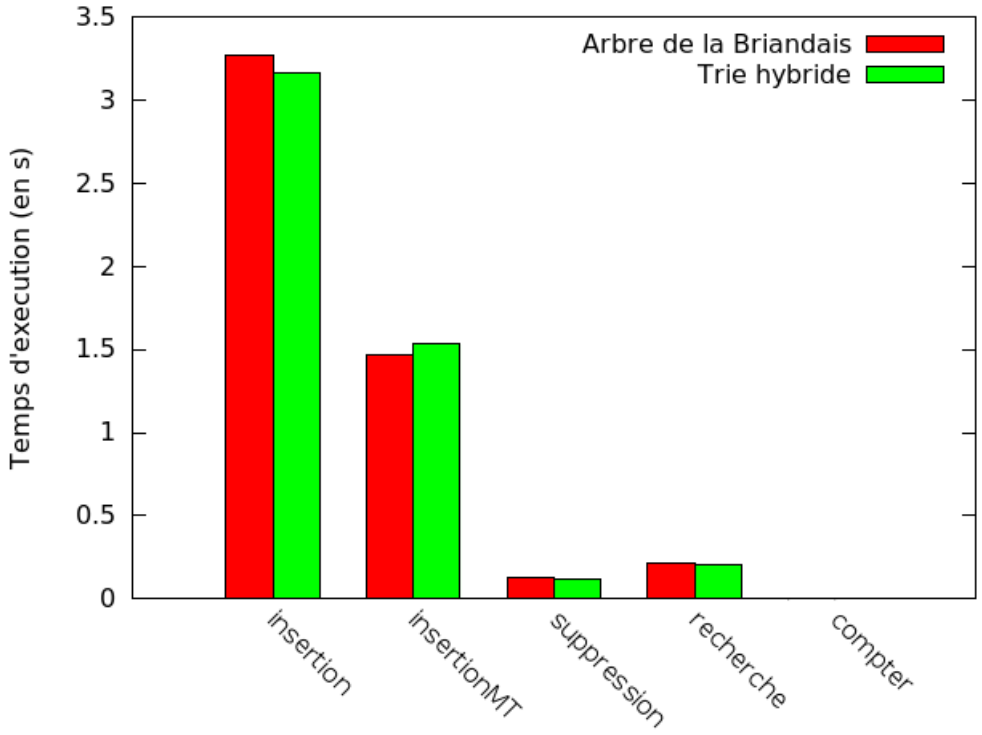
— insertion des mots de toutes ces oeuvres par ajouts successifs,

— insertion des mots de toutes ces oeuvres parallèlement puis fusion,

— suppression des mots de Hamlet de l'arbre comprenant toute les oeuvres,

— recherche des mots de All's Well et Hamlet dans l'arbre comprenant toute les oeuvres après suppression de Hamlet,

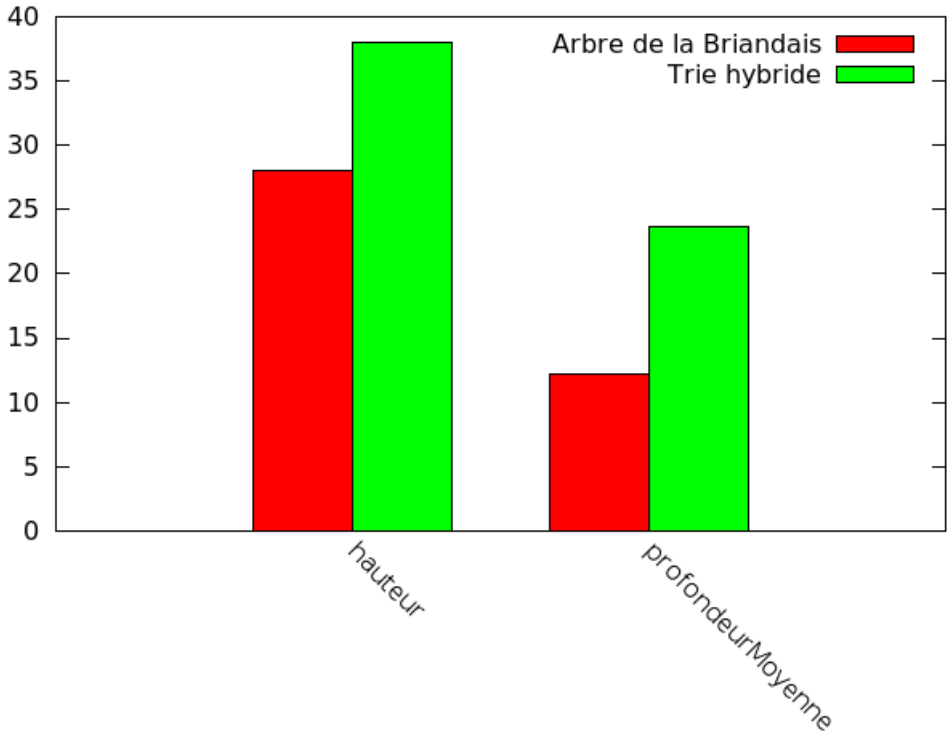
— comptage des mots de l'arbre l'arbre comprenant toute les oeuvres après suppression de Hamlet.



On remarque que les performances des deux structures sont très proches, voir identiques.

4.2 Caractéristiques de l'arbre

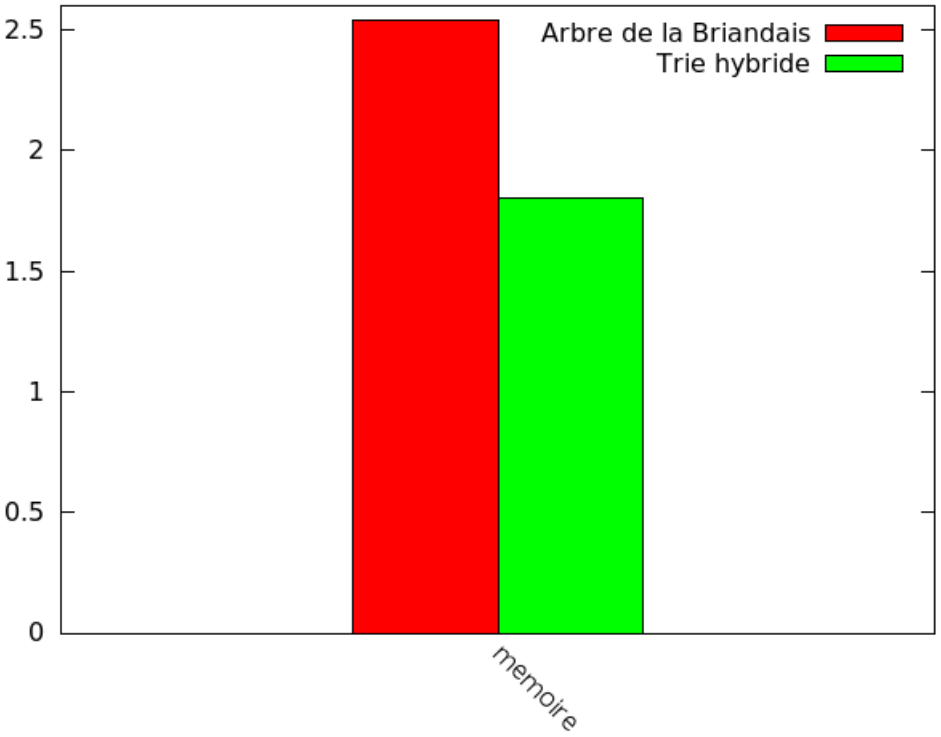
On va maintenant comparer les caractéristiques des arbres construits, à savoir la hauteur et la profondeur moyenne.



On remarque que l'arbre de la Briandais semble bien meilleur, mais les réultats sont en fait trompeurs car l'arbre de la Briandais s'étale "en largeur", la chaîne des frères pouvant être relativement longue, et la profondeur étant la longueur du plus long mot plus un. Quant au trie hybride, il s'étend plus en profondeur, mais ne perd pas en terme de performances grace à ses complexités logarithmiques.

4.3 Occupation mémoire

Un dernier critère est l'occupation en mémoire de ces structures. En effet, les deux structures étant similaires au niveau des performances, le choix se fera au niveau de l'occupation mémoire.



On remarque que le trie hybride est bien plus léger que l'arbre de la Briandais. Cela s'explique probablement par le fait que dans ce dernier, on a un noeud de plus par mot dans l'arbre pour stocker le caractère de fin de mot ϵ.

4.4 Conclusion

Pour conclure, on choisira probablement le trie hybride par rapport à l'arbre de la Briandais car, à performances similaires, le trie hybride limite l'occupation mémoire du dictionnaire.