Rapport de mini projet

Algorithmique Avancée

Réalisé par : Naim CHOULLIT

Sujet : ***représenter un dictionnaire de mots avec les Patricia-tries et les trie hybrides***

Summaire

Introduction

Le but de ce projet consiste à représenter un dictionnaire de mots. Dans cette optique, nous proposons l'implémentation de deux structures de tries concurrentes puis une étude expérimentale permettant de mettre en avant les avantages et inconvénients de chacun des modèles. En plus des implantations des structures de données et des primitives de base, nous envisageons une fonction avancée pour chacun des modèles.

Les deux modèles envisagés sont les Patricia-tries et les tries hybrides.

Ce projet sera implementé en java.

Structures

Les Patricia-tries

Parmi le code ASCII nous pouvons déterminer plusieurs marqeurs de fin de chain (ou de fin d'un mot). Par exemple, les caratères '*' ou '#' mais ici nous prendrons le caractère '$' .

Notre Patricia-trie est constitue de deux éléments basiques :

- un root qui est la Patricia elle-meme.
- le (s) fils de ce root qui est (sont) stocké (s) dans un treeMap.

Remarque :

Le conteneur *TreeMap* permet de stocker des couples (clé, valeur), dans une structure d'arbre binaire équilibré rouge-noir. Cette classe garantit que la collection *Map* sera triée selon un ordre croissant, conformément à l'ordre naturel des clés ou à l'aide d'un comparateur fournie au moment de la création de l'objet *TreeMap*

Pour nous aider dans la construction d'un arbre Patricia, nous allons utiliser des primitives de base :

- new PatriciaTrie () → initialise un Patricia-trie.
- getKey (PatriciaTrie trie) → retourne la clé de la Patricia-trie (trie).
- insert(PatriciaTrie trie, String word). → ajoute un mot à un arbre Patricia (trie).
- empty(PatriciaTrie trie)→ test si l'arbre est vide.

- fils(PatriciaTrie trie) → retourne le le fils de la Patricia-trie
- prefix (String word1, String word2) → retourne le prefixe des mots word1 et word2.

Nous allons construire un arbre Patricia à partir du texte exemple de base : «A quel genial professeur de dactylographie sommes nous redevables de la superbe phrase ci dessous, un modele du genre, que toute dactylo connaît par coeur puisque elle fait appel a chacune des touches du clavier de la machine a ecrire ?»

Tries Hybrides

Notre trie hybrid est constitué de de 5 élements basiques :

- un root qui l'arbre elle-meme.
- une clé de type String.

-un fils gauche de type TrieHybrid.

-un fils droite de type TrieHybrid.

-un fils centre de type TrieHybrid.

-un boolean qui indique la fin d'un mot

Comme pour l'arbre de Patricia, nous allons définir des primitives nous permettant de constuire notre trie hybride :

- TrieHybrid (String clé) → retourne une trie hybride (TrieHybrid).
- TrieVide(TrieHybrid trie) → test si la trie est vide.
- getKey(TrieHybrid trie) → retourne la clé de la trie.
- fils(TrieHybrid trie) → retourn le fils de la trie.
- insert(String word, TrieHybrid trie)→ ajoute le mot m aux tries hybride t.

Idem à la première partie (arbre de Patricia), nous allons construire une trie hybride à partir du texte exemple de base.

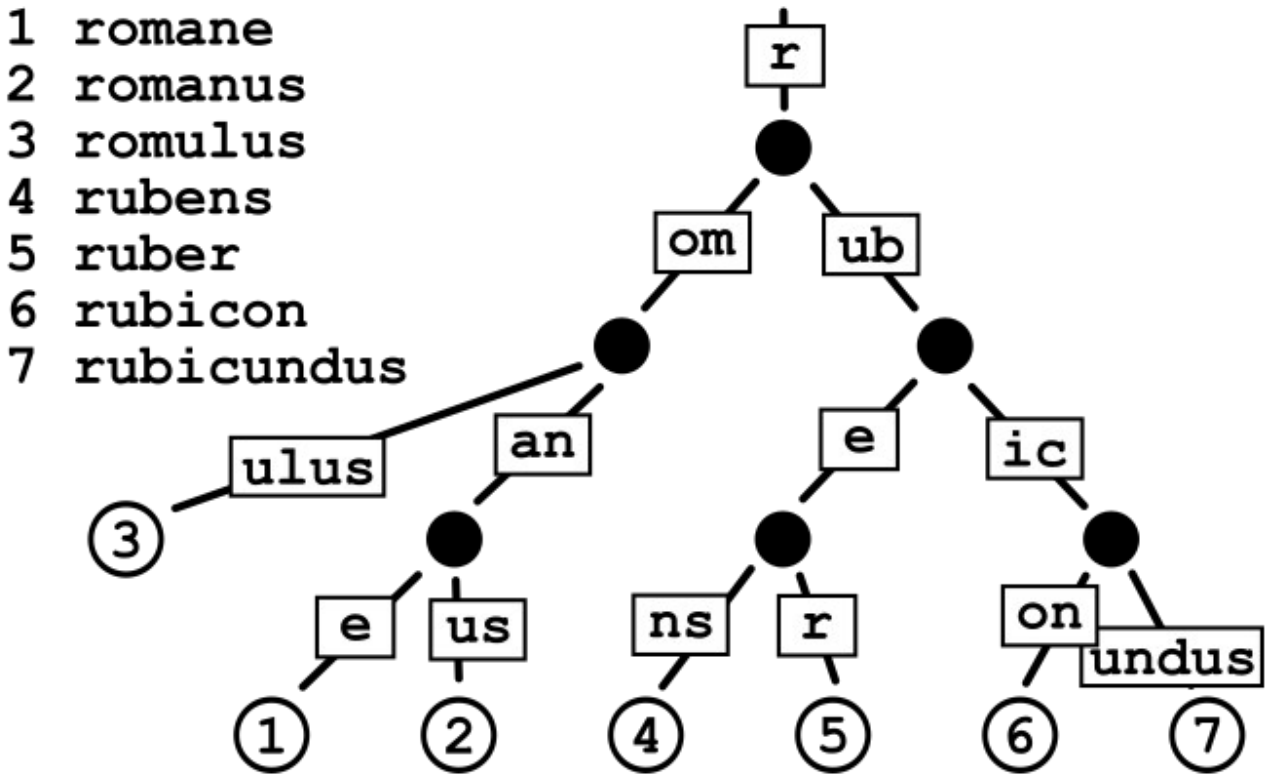

Figure 1 : Patricia-trie

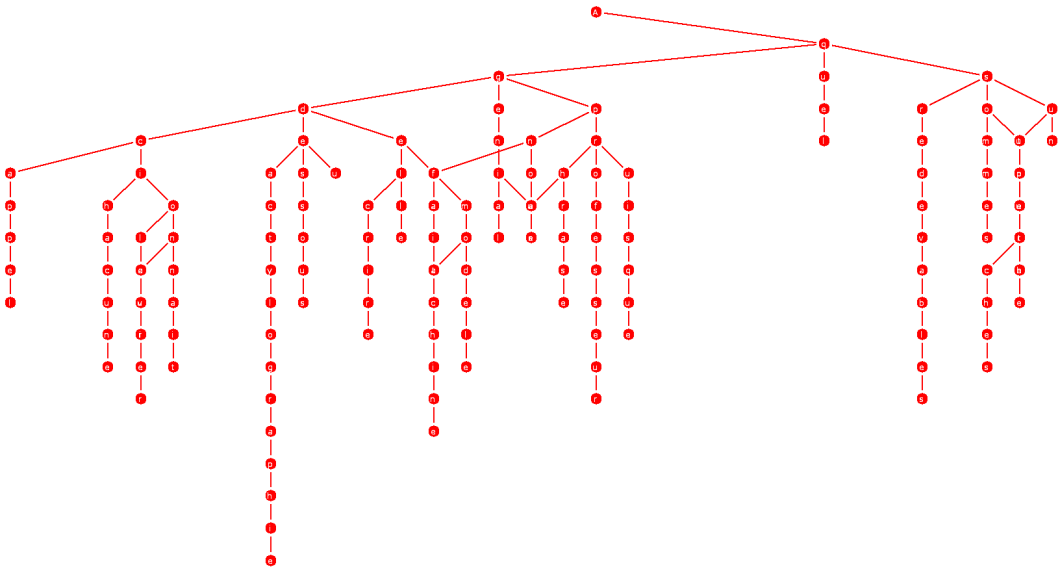

Figure 2 : Trie Hybride

Fonctions avancées pour chacune des structures

Fonctions pour les Patricia-tries:

fonction de recherche d'un mot dans un dictionnaire: Recherche(arbre, mot) → boolean.

Cette fonction est une fonction récurcive, elle fait une iteration sur la racine de mon arbre, jusqu'à elle trouve une clé qui contient un ou plusieures lettres de mot rechercher, dans ce cas elle continue la recherche dans le fils de mon arbre avec ce qui reste de mon mot.

Lorsqu'on on finit la recherche, si la clé de dernier nœud fini par le caratères '$' elle retourne true (le mot est trouvé), false sinon.

- Fonction qui compte les mots présents dans le dictionnaire : ComptageMots(arbre) → entier.

Cette fonction est fonction récurcive, elle cherche tous les mots dans mon Patricia-trie, elle cherche dans les nœuds de mon arbre, et à chaque fois qu'elle trouve un mot qui termine par le '$' elle incremente un compteur de mots.

- Fonction qui liste les mots du dictionnaire dans l'ordre alphabétique : ListeMots(arbre) → liste[mots].

Comme le comptage des mots, cette fonction a le meme principe, sauf que elle ajoute le mot dans une liste.

- Fonction qui compte les pointeurs vers NIL : ComptageNil(arbre) → entier .

  Cette fonction compte le nombre de nœud vide mon arbre.

- Fonction qui calcule la hauteur de l'arbre : Hauteur(arbre) → entier.

  Elle retourne le maximum des hauteurs des nœuds dans mon arbre.

- Fonction qui calcule la profondeur moyenne des feuilles de l'arbre ProfondeurMoyenne(arbre) → entier.

Elle calcule la profondeur moyenne des feuilles en se basant sur les niveaux, le nombre de nœud a chaque niveau, et le nombre total de nœuds.

Profondeur moyenne = (nombre de nœud a chaque niveau * niveau) / nombre total de nœuds.

- Fonction qui prend un mot A en argument et qui indique de combien de mots du dictionnaire le mot A est le préfixe : Prefixe(arbre,mot) → entier.

Elle cherche un prefixe dans les clés de mon arbre, lorsqu'elle le trouve, elle compte le nombre de mots dans le fils de ce nœud.

- Fonction qui prend un mot en argument et qui le supprime de l'arbre s'il y figure : Suppression(arbre,mot) → arbre.

  Elle cherche un prefixe dans les clés de mon arbre, si elle la trouve, elle continue sa

recherche recurcive dans le fils de nœud actuel jusqu'à elle tombe sur une clé qui fini par '$', dans ce cas, si ce nœud a des freres, on supprime ce nœud, sinon on remonte a sa racine et on la mis a jour en ajoutant le '$' dans sa clé, puis on supprime son fils.

Fonctions pour un trie hybride

- fonction de recherche d'un mot dans un dictionnaire : Recherche(arbre,mot) → booléen.

Elle compare la clé de la racine de l'arbre avec le mot recherché, si cette clé est inferieur (superieur) que le la premiere lettre de mot recherché, elle recherche dans son fils gauche (respectivement droite) sinon ,si cette clé est un prefixe de ce mot, elle fait une recherche recurcive sur son fils de milieu jusqu'à elle se trouve sur une feuille de cette arbre, si la valeur qui indique la fin d'un mot est verifier, cette fonction retiurne true.

- Fonction qui compte les mots présents dans le dictionnaire : ComptageMots(arbre) → entier.

Parcours tous les fils (droite, gauche, milieu) de la racine jusqu'à elle se trouve sur une feuille, dans ce cas elle incremente le nombre de mots.

- Fonction qui liste les mots du dictionnaire dans l'ordre alphabétique : ListeMots(arbre) → liste[mots].

  Meme principe que le comptage des mots, mes cette fois elle les sauvgarde dans une liste.

- Fonction qui compte les pointeurs vers NIL : ComptageNil(arbre) → entier.

Meme principe que pour le comptage des mots, mais dans notre cas elle compte le nombre feuilles vides (null).

- Fonction qui calcule la hauteur de l'arbre : Hauteur(arbre) → entier

en partant de la racine, à chaque fois que cette racine a un fils, la fonction incrémente la hauteur de celle-ci, au final elle prend le maximum de ces hauteurs.

- Fonction qui calcule la profondeur moyenne des feuilles de l'arbre : ProfondeurMoyenne(arbre) → entier.

Meme principe que pour le calcul de la profondeur moyenne de patricia-trie, mais cette fois, elle la calcule avec les fils (gauche, milieu, droite).

- Fonction qui prend un mot A en argument et qui indique de combien de mots du dictionnaire le mot A est le préfixe : Prefixe(arbre, mot) → entier

Meme principe que pour la recherche d'un mot, mais cette fois, dés qu'elle arrive vers la fin de ce prefixe, elle compte le nombre de mots qui se trouvent de les fils de cette racine.

- Fonction qui prend un mot en argument et qui le supprime de l'arbre s'il y figure : Suppression(arbre, mot) → arbre.

  Meme principe que pour la recherche d'un mot, vu qu'un mot se trouve dans le fils de milieu

de notre racine, il suffit juste de remplacer cette racine par son fils gauche, et inserer le droite de la racine dans ce dernier (fils gauche de la racine).

Fonctions complexes

- fonction qui prend deux arbres Patricia en argument et les fusionnent en un troisième : fusion(arbre1,arbre2) → PatriciaTrie.

Elle verifie les clés respectives de chaque racine d'un arbre, dans le cas ou elles sont différentes , il suffira alors d'inséré les deux arbres notre nouvelle arbre, dans le cas contraire, elle fait la meme etape sur chaque fils des deux arbres.

- Fonction permettant de passer d’un trie hybride à un arbre Patricia : ThybridToPatricia (trie) → BRDtree.

Recupere chaque mot dans notre arbre Hybrid et l'inserer dans la trie ;

- fonction permettant de passer d’un arbre Patricia à un trie hybride : PatriciaToTHybrid (arbre) → Thybrid : Meme pricipe que pour passer d’un trie hybride à un arbre Patricia.
- Fonction permettant de réquilibrer un trie hybride : rebalancing (trie) → Triehybrid.

Elle s'est basée sur les l'insertion avec rotation.

Complexités Arbres Patricia

Dans la comparaison suivante, on supposera que les mots sont de longueur L et que le nombre de mot est N.

- Les opérations de recherche, de suppression et d'insertion d'un mot ont des complexités en O(L). Par conséquent, la complexité temporelle de ces opérations ne dépend pas du nombre de données contenues par l'arbre Patricia.
- fonction de comptage de mots, on parcours tous l'arbre (dans le pire et le meilleur cas) soit L caractères et pour chaque L on parcours sa hauteur soit ln N donc notre complexité est en O(L * ln N).
- fonction liste de mots, on parcours simplement tous l'arbre donc comme pour le comptage de mots nous avons une complexité en O(L * ln N).
- fonction de comptage des NULL, idem que comptage de mots. fonction de calcul de la hauteur, on parcours tous l'arbre encore une fois donc notre complexité s'exprime en O(L * log N).
- fonction de calcul de la profondeur moyenne de l'arbre, on parcours tous l'arbre donc comme pour les fonctions précédentes on exprime notre complexité en O(L * ln N).
- fonction de comptage de prefix, cela est déterminée par la hauteur de l'arbre pour le prefix recherché donc nous avons une complexité en O(L + ln N).

Complexités Tries Hybrides

- fonction de recherche d'un mot dans un trie hybride, dans le pire cas on parcours L caractères (L correspond à la longueur du mot) plus sa haurteur donc notre complexité est en O(L + ln N).
- fonction de comptage de mots, on parcours tous le trie soit L caractères et pour chaque L on parcours sa hauteur soit ln N donc notre complexité est en O(L * ln N).
- fonction liste de mots, on parcours simplement le trie donc notre complexité est en O(L * ln N).
- fonction de comptage des NULL, idem que comptage de mots.
- fonction de calcul de la hauteur, on parcours tous le trie donc nous avons une complexité exprimée en O(L * log2 N).
- fonction de calcul de la profondeur moyenne du trie, on parcours tous le trie donc notre complexité est en O(L * ln N).
- fonction de comptage de prefix, cela est déterminée par la hauteur de l'arbre pour le prefix recherché donc notre complexité est en O(L + ln N).
- fonction de suppression d'un mot dans le trie, cette fonction dépend aussi de la hauteur du trie donc notre complexité est en O(log2 N).

La complexité en temps des opérations dans un arbre ternaire de recherche est similaire à celle d'un arbre binaire de recherche. C'est à dire les opérations d'insertion, de suppression et de recherche ont un temps proportionnelle à la hauteur de l'arbre ternaire de recherche. L'espace est proportionnelle à la longueur de la chaîne à mémoriser

Étude expérimentale

apres avoir effectué quelque tests sur mes arbres avec différentes entrées (exemple de base, l'un des fichier Shakespeare et tous les fichiers de Shakespeare) , on a eu ces résultats suivant :

Time for insert (ms)

	exemple de base	1henryiv	Shakespeare
Patricia	18	310	9418
Hybride	0,719	112	1623
Hybride Balanced	3	12694	456000

Figure 3 : temps d'insertion dans les arbres (en meliseconds)

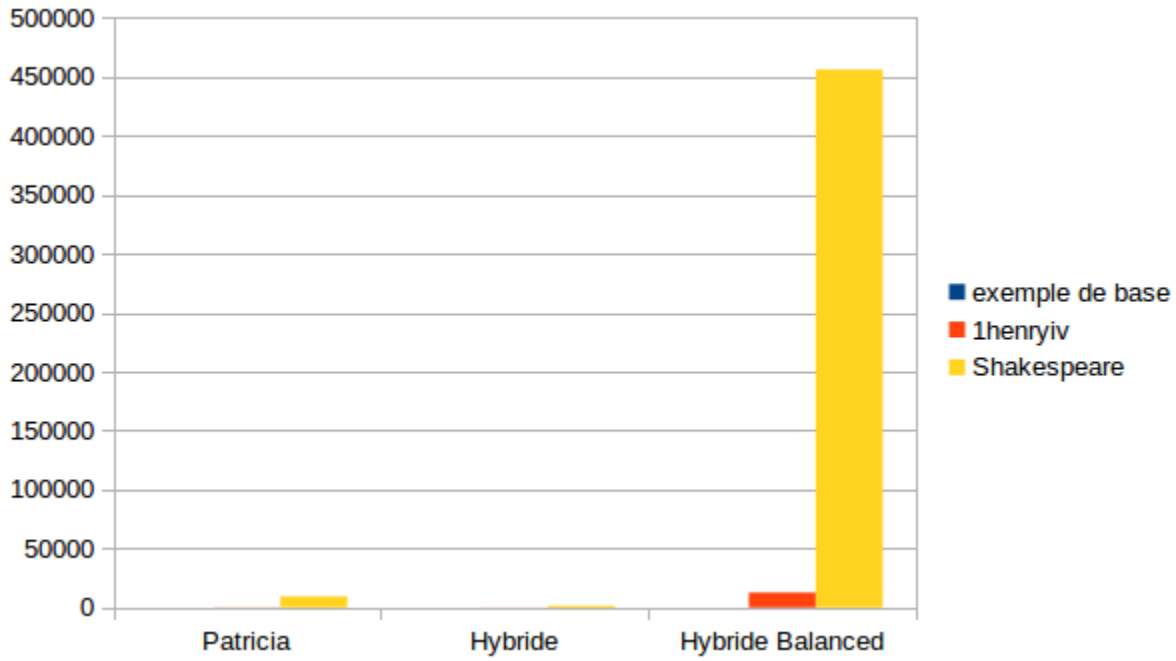

Figure 4 : diagramme pour le temps d'insertion dans les arbres (en meliseconds)

ce diagramme montre clairement que l'ajout des mots se fait plus rapidement dans les tries hibrydes, de ce fait le trie hibryde est mieulleur en terme rapidité.

Height in insert

	exemple de base	1henryiv	Shakespeare
Patricia	4	8	11
Hybride	18	29	37

Figure 5 : hauteur des arbres apres les insertion

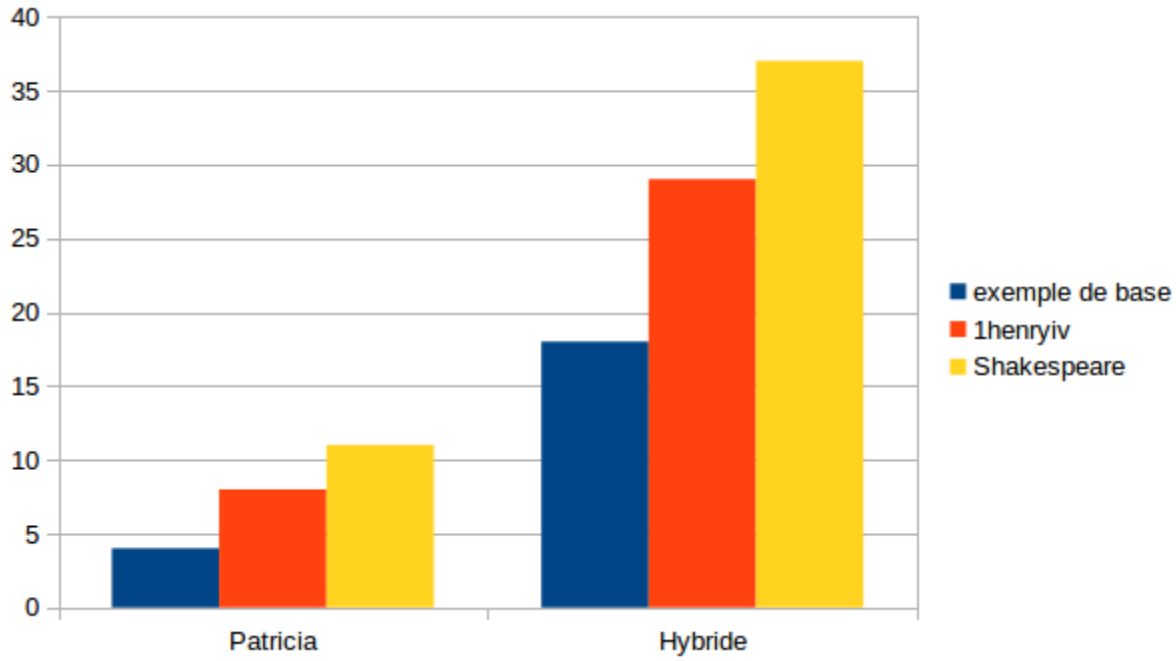

Figure 6 : diagramme de hauteur des arbres apres les insertion

ce diagramme montre clairement que l'occupation d'espace mémoire aprés l'ajout des mots est moins important dans les Patricia-tries, de ce fait les patricia-tries est mieulleur en terme d'occupation d'espace mémoire.

Time for delete (ms)

	exemple de base	1henryiv	Shakespeare
Patricia	19	359	1646
Hybride	20	173	340

Figure 7 : temps d'e suppression dans les arbres (en meliseconds)

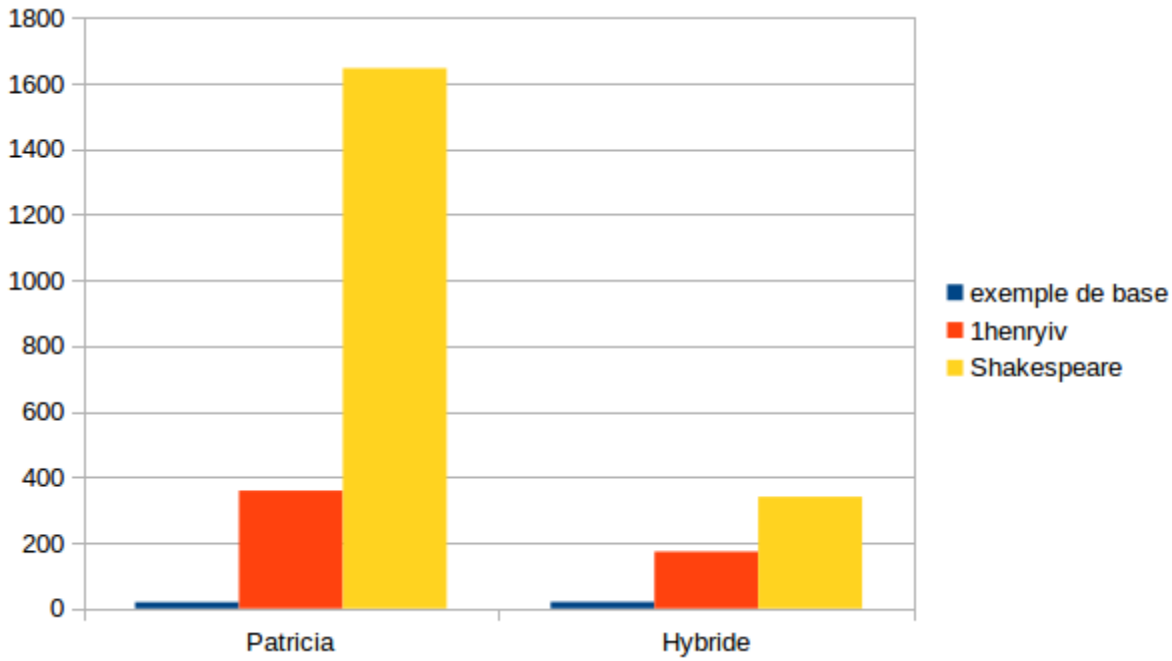

Figure 8 : diagramme de temps de suppression dans les arbres (en meliseconds)

ce diagramme montre clairement que la suppression des mots se fait plus rapidement dans les tries hibrydes, de ce fait le trie hibryde est mieulleur en terme rapidité.

Conclusion :

pour les deux type d'arbre qu'on a vu, chaqu'une d'elle a ses avantages et inconvénients, la Patricia-trie est mielleur en terme d'occupation d'espace mémoire, et moins bonne en terme de rapidité, pour la trie hibryde, elle comme avantage sa rapidité d'insertion en terme de temps, a comme inconvienient l'occupation de l'espace mémoire.

Table de figures