Projet ALGAV

Tries

19/12/2024

CROUZET Oriane 21414555
LOUIS Elise 2130854

Tables des matières

I. Introduction

Ce projet de programmation a pour but d'étudier deux structures de dictionnaires différentes : les Patricia Tries et les Tries hybrides. Ces structures sont particulièrement adaptées à la recherche et à l'insertion de nouveaux mots dans le dictionnaire, tout en assurant une organisation optimisée des clés.

L'objectif de ce devoir est d'implémenter les différentes structures ainsi que leurs primitives et fonctions avancées, utiles pour comparer l'efficacité des deux structures de données. Ensuite, une étude expérimentale permettra de comparer les performances respectives des deux structures, notamment en termes de temps de calcul, sur des ensembles de données variés, incluant des œuvres complètes de Shakespeare.

Ce rapport présente tout d'abord les implémentations (en Java) des Patricia Tries et des Tries hybrides, avant de détailler les algorithmes développés. Enfin, une analyse approfondie des résultats expérimentaux mettra en évidence les avantages et inconvénients de chaque modèle.

II. Structure des Patricia Tries

Un **Patricia trie** est une structure de données optimisée pour représenter efficacement un ensemble de mots dans un dictionnaire. Dans notre implémentation, deux classes permettent de les manipuler : PatriciaTrieNode et PatriciaTrie.

Principales caractéristiques :

1. **Structure compacte :**
    - Chaque noeud (PatriciaTrieNode) contient un ensemble trié de couple (clé, valeur) représentant les enfants. Ici l'ensemble sera constitué de maximum 127 enfants, car les mots constituant le dictionnaire sont codés sur les 127 premiers caractères du code ASCII.
        - **Clé :** le premier caractère de l'arête associée au noeud valeur
        - **Valeur :** pointeur vers le PatriciaTrieNode lié à la clé.
2. **Optimisation mémoire :**
    - Le trie Patricia réduit drastiquement l'utilisation de la mémoire en évitant de créer un nœud par caractères. Les nœuds contiennent majoritairement des préfixes communs à des sous ensemble de nœuds et/ou si aucun chemin divergent un mot peut être représenter par un seul nœud.
3. **Représentation des mots :**
    - Chaque PatriciaTrieNode possède une arête et un statut
    - L'arête indique la chaîne de caractères attribué au noeud
    - Le statut du noeud indique s'il s'agit d'une fin de mot ou non

Enfin, le PatriciaTrie est représenté par une racine (root) de type PatriciaTrieNode. Toutes les fonctions ont été implémentées dans la classe PatriciaTrie

III. Structure des Tries hybrides

Un **trie Hybride** est une structure de données optimisée pour représenter un ensemble de mots dans un dictionnaire. Dans notre implémentation, deux classes permettent de les manipuler : HybridTrieNode et HybridTrie.

Principales caractéristiques :

1. **Structure tertiaire :**
    - Chaque nœud (HybridTrieNode) peut avoir trois enfants :
        - **Gauche** (INF) : représente les caractères inférieurs au caractère du nœud.
        - **Milieu** (EQ) : correspond au même caractère que le nœud.
        - **Droite** (SUP) : représente les caractères supérieurs au caractère du nœud.
    - Ces pointeurs sont représentés dans le code par un attribut **tableau** (pointeurs) dans le HybridTrieNode : on peut accéder aux différentes branches avec les attributs HybridTrieNode.INF, HybridTrieNode.EQ et HybridTrieNode.SUP.

2. **Optimisation mémoire :**
    - Dans notre structure, les nœuds inutilisés (pointeurs null) ne sont pas explicitement créés, ce qui réduit la consommation de mémoire.

3. **Représentation des mots :**
    - Chaque HybridTrieNode possède un caractère et une valeur.
    - Le caractère (car) indique quel caractère a été attribué au nœud.
    - La valeur (val) du nœud indique s'il s'agit d'une fin de mot (-1) ou non (3).

NB : les valeurs ont été choisies arbitrairement, et sont manipulées dans le code à l'aide de constantes : HybrideTrieNode.NOT_END_WORD et HybrideTrieNode.END_WORD.

Enfin, le HybridTrie est représenté par une racine (root) de type HybridTrieNode. Toutes les fonctions ont été implémentées dans la classe HybridTrie.

IV. Fonction avancées

A. Patricia Tries

Plusieurs fonctions permettent de manipuler et analyser les Tries Patricias. Parmi elles, nous retrouvons les primitives de bases, qui concernent directement la structure du trie, mais aussi des fonctions plus avancées pour obtenir des informations sur l'arbre.

Insertion

Avant de procéder à l'insertion, elle effectue une vérification des caractères du mot pour s'assurer qu'ils appartiennent au code ASCII standard (0 à 127). En cas de caractère non valide, une exception est levée.

La fonction appelle la méthode récursive insertRec, qui ajoute le mot en naviguant dans l'arbre. Cette navigation respecte les principes d'un trie patricia :

- Si on arrive à la fin du mot à insérer, on change le statut du nœud courant pour marquer la fin du mot.
- Si il existe un nœud a un préfixe commun avec le mot à insérer, on continue le parcours à partir de ce nœud.
- Si il existe un noeud a un préfixe commun partiel avec le mot à insérer, le noeud est divisé en noeud :
    - Le préfixe commun partiel est conservé dans un nœud et l'autre partie du préfixe du nœud est ajoutée en tant qu'enfant de ce nœud, de même que l'autre partie du mot à ajouter.
- Si il n'existe pas de préfixe commun on ajoute un noeud au noeud courant ayant comme arête le mot à insérer

La méthode insert et insertRec elles-mêmes ne retournent rien. Elles modifient l'arbre en mettant à jour la racine après chaque insertion.

Recherche

La méthode search est une méthode qui appelle une méthode récursive interne searchRec pour effectuer la recherche d'un mot dans le trie.

La recherche suit les règles spécifiques à la structure du trie Patricia:

- Si on arrive à la fin du mot à rechercher, on renvoie le statut du nœud courant (si c'est une fin de mot ou non).
- Si il existe un nœud a un préfixe commun avec le mot à rechercher, on continue le parcours à partir de ce nœud.

- Si il n'existe pas de préfixe commun, on renvoie False car le mot n'a pas été trouvé dans le trie.

La méthode searchRec et search retourne un booléen :

- true si le mot est trouvé dans le trie et marqué comme une fin valide de mot.
- false sinon.

Suppression

La fonction deteleWord permet de retirer un mot présent dans un trie Patricia. La fonction publique deleteRec fonctionne de la manière suivante :

V. Si on arrive à la fin du mot à supprimer, on change son statut de fin de mot à False.
VI. Si il existe un nœud a un préfixe commun avec le mot à rechercher, on continue le parcours à partir de ce nœud.
VII. Si il n'existe pas de préfixe commun, on lève une exception car le mot n'est pas présent dans le trie.

À la fin du parcours, on vérifie si certains nœuds devenus inutiles doivent être supprimés, et, si nécessaire, on fusionne les nœuds quand c'est possible pour réduire le nombre total de nœuds dans le trie.

La méthode deleteRec retourne un booléen :

- true si on doit supprimer le nœud du trie.
- false sinon.

Comptage des mots

La méthode publique countWords sert de point d'entrée pour lancer le comptage. Elle appelle la méthode récursive interne countWordsRec, qui effectue le véritable calcul.

Le principe du comptage est le suivant :

1. Chaque nœud de l'arbre est examiné pour vérifier s'il marque la fin d'un mot dans le dictionnaire .
2. Si le nœud représente une fin de mot, un compteur est incrémenté.
3. La méthode explore récursivement les enfants du nœud courant pour cumuler les résultats.

La méthode s'arrête lorsqu'elle a fini son parcours préfixe.

Comptage des Nils

La méthode publique countNils appelle la fonction récursive countNilsRec à partir de la racine du trie. Elle fonctionne comme suit :

- On effectue un parcours préfixe à partir de la racine root et lorsqu'un nœud n'a plus d'enfant, cela veut dire qu'il est une racine du trie, on incrémente le compteur de Nil.

La méthode s'arrête lorsqu'elle a fini son parcours.

Liste des mots

La méthode listeMots sert de point d'entrée pour générer une liste des mots dans l'ordre alphabétique :

1. Elle utilise un StringBuilder pour accumuler les caractères des mots lors de l'exploration récursive de l'arbre.
2. La méthode listeMotsRec est appelée pour parcourir tous les nœuds du trie et construire la liste des mots.

La méthode récursive listeMotsRec fonctionne comme suit :

- Si le nœud courant est null, elle renvoie une liste vide (cas de base).
- Elle explore successivement les trois sous-arbres possibles :
    - **Sous-arbre gauche (INF)** : Pour les mots lexicographiquement inférieurs au caractère courant.
    - **Sous-arbre central (EQ)** : Pour les mots contenant le caractère courant.
    - **Sous-arbre droit (SUP)** : Pour les mots lexicographiquement supérieurs au caractère courant.
- Lorsqu'elle rencontre un nœud marquant la fin d'un mot (val != NOT_END_WORD), elle ajoute le mot complet dans la liste.

Les sous-arbres sont explorés récursivement, et un préfixe est maintenu pour construire les mots au fur et à mesure.

Hauteur

La méthode publique height appelle la fonction récursive heightRec à partir de la racine du trie. La méthode récursive heightRec fonctionne comme suit :

- La fonction appelle récursivement les sous-arbres du nœud courant pour calculer leur hauteur respective.
- La hauteur du nœud courant est déterminée comme 1 + max(hauteur_Children).

Profondeur moyenne des feuilles

La fonction meanHeight calcule la profondeur moyenne des feuilles d'un trie Patricia, c'est-à-dire la moyenne des distances entre la racine et chacune des feuilles. Elle repose sur une méthode récursive qui parcourt l'arbre, accumulant les profondeurs et comptant les feuilles rencontrées.

La méthode appelle la fonction récursive meanHeightRec, qui explore tout l'arbre, en passant la profondeur courante et qui renvoie un tableau d'entier tel que :

- **tableauInt[0] :** la somme des profondeurs.
- **tableauInt[1] :** la somme des nœuds du trie.

À la fin, la moyenne est calculée comme : **Profondeur moyenne** = $\frac{tableauInt[0]}{tableauInt[1]}$.

Préfixe

La fonction prefixe permet de calculer combien de mots dans le dictionnaire ont un mot donné A comme préfixe. La fonction combine deux opérations : une navigation pour localiser le nœud correspondant au préfixe et un comptage des mots à partir de ce nœud.

La fonction publique prefixe fait appel à la fonction prefixeRec. Cette fonction s'exécute en deux étapes :

VIII. **Localisation du préfixe** :
   A. On effectue les mêmes étapes de recherche que dans la fonction searchRec.

IX. **Comptage des mots** :
   A. Une fois le nœud correspondant localisé, la fonction utilise le même mécanisme que pour la fonction countWords, c'est-à-dire qu'elle incrémente le compteur dès que l'on croise une fois de mots.

B. Tries hybrides

Plusieurs fonctions permettent de manipuler et analyser les Tries hybrides. Parmi elles, nous retrouvons les primitives de bases, qui concernent directement la structure du trie, mais aussi des fonctions plus avancées pour obtenir des informations sur l'arbre.

Insertion

Avant de procéder à l'insertion, elle effectue une vérification des caractères du mot pour s'assurer qu'ils appartiennent au code ASCII standard (0 à 127). En cas de caractère non valide, une exception est levée.

Si le mot n'est pas déjà présent dans l'arbre (vérifié grâce à la fonction recherche), la fonction appelle la méthode récursive insertRec, qui ajoute le mot caractère par caractère en naviguant dans l'arbre. Cette navigation respecte les principes d'un trie hybride :

- Les sous-arbres gauche (INF) contiennent les caractères inférieurs au caractère du nœud courant.
- Le sous-arbre milieu (EQ) contient le prochain caractère du mot à insérer, si égal au caractère courant.
- Les sous-arbres droit (SUP) contiennent les caractères supérieurs.

Une fois un mot inséré, un rééquilibrage facultatif de l'arbre est effectué si le paramètre balance est activé.

La méthode insert elle-même ne retourne rien. Elle modifie l'arbre en mettant à jour la racine après chaque insertion. La méthode récursive insertRec retourne le nœud courant modifié. Ce retour est nécessaire pour mettre à jour correctement les pointeurs dans l'arbre.

Recherche

La méthode recherche est une méthode qui appelle une méthode récursive interne rechercheRec pour effectuer la recherche d'un mot dans le trie.

La recherche suit les règles spécifiques à la structure du trie hybride :

- Si le caractère actuel du mot est inférieur au caractère du nœud courant, la recherche se poursuit dans le sous-arbre gauche (INF).
- Si le caractère est supérieur, la recherche bascule vers le sous-arbre droit (SUP).
- Si le caractère est égal, la recherche progresse dans le sous-arbre central (EQ).

La recherche se termine lorsque :

1. La chaîne complète du mot a été parcourue, et le nœud final correspond à une fin valide de mot (val != NOT_END_WORD).
2. Un nœud null est rencontré, ce qui signifie que le mot n'existe pas dans l'arbre.

La méthode recherche retourne un booléen :

- true si le mot est trouvé dans le trie et marqué comme une fin valide de mot.
- false sinon.

Comptage des mots

La méthode publique comptageMots sert de point d'entrée pour lancer le comptage. Elle appelle la méthode récursive interne comptageMotsRec, qui effectue le véritable calcul.

Le principe du comptage est le suivant :

4. Chaque nœud de l'arbre est examiné pour vérifier s'il marque la fin d'un mot dans le dictionnaire (valeur de val égale à HybridTrieNode.END_WORD).
5. Si le nœud représente une fin de mot, un compteur est incrémenté.
6. La méthode explore récursivement les trois sous-arbres possibles (gauche, milieu et droit) pour cumuler les résultats.

La méthode s'arrête lorsqu'elle atteint un nœud vide (null), indiquant qu'il n'y a plus de mots à comptabiliser dans ce chemin.

Liste des mots

La méthode listeMots sert de point d'entrée pour générer une liste des mots dans l'ordre alphabétique :

3. Elle utilise un StringBuilder pour accumuler les caractères des mots lors de l'exploration récursive de l'arbre.
4. La méthode listeMotsRec est appelée pour parcourir tous les nœuds du trie et construire la liste des mots.

La méthode récursive listeMotsRec fonctionne comme suit :

- Si le nœud courant est null, elle renvoie une liste vide (cas de base).
- Elle explore successivement les trois sous-arbres possibles :
    - **Sous-arbre gauche (INF)** : Pour les mots lexicographiquement inférieurs au caractère courant.
    - **Sous-arbre central (EQ)** : Pour les mots contenant le caractère courant.

    - **Sous-arbre droit (SUP)** : Pour les mots lexicographiquement supérieurs au caractère courant.
- Lorsqu'elle rencontre un nœud marquant la fin d'un mot (val != NOT_END_WORD), elle ajoute le mot complet dans la liste.

Les sous-arbres sont explorés récursivement, et un préfixe est maintenu pour construire les mots au fur et à mesure.

Comptage des Nils

La méthode publique comptageNil appelle la fonction récursive comptageNilRec à partir de la racine du trie. La méthode récursive comptageNilRec fonctionne comme suit :

- **Cas de base** : Si le nœud courant est null, cela signifie qu'il s'agit d'un pointeur Nil, donc on retourne 1.
- **Cas récursif** :
    - On itère sur les trois sous-arbres possibles (gauche, égal et droit).
    - Pour chaque sous-arbre, on additionne le nombre de Nil trouvés récursivement.
    - Avant cela, on vérifie chaque pointeur du tableau pointeurs du nœud courant pour déterminer combien de pointeurs sont explicitement null.

Hauteur

La méthode publique hauteur appelle la fonction récursive hauteurRec à partir de la racine du trie. La méthode récursive hauteurRec fonctionne comme suit :

- **Cas de base** :
    - Si le nœud courant est null, cela signifie que nous avons atteint une feuille, donc la hauteur est définie comme 1 pour ce cas.
- **Cas récursif** :
    - La fonction appelle récursivement les sous-arbres INF, EQ, et SUP pour calculer leur hauteur respective.
    - La hauteur du nœud courant est déterminée comme 1 + max(hauteurInf, hauteurEq, hauteurSup).

Profondeur moyenne des feuilles

La fonction profondeurMoyenne calcule la profondeur moyenne des feuilles d'un trie hybride, c'est-à-dire la moyenne des distances entre la racine et chacune des feuilles. Elle

repose sur une méthode récursive qui parcourt l'arbre, accumulant les profondeurs et comptant les feuilles rencontrées.

La méthode publique profondeurMoyenne initialise un tableau result pour stocker deux valeurs :

- **result[0]** : La somme des profondeurs de toutes les feuilles.
- **result[1]** : Le nombre total de feuilles rencontrées.

La méthode appelle ensuite la fonction récursive profondeurMoyenneRec, qui explore tout l'arbre, en passant la profondeur courante et le tableau result pour mise à jour. À la fin, la moyenne est calculée comme : **Profondeur moyenne** = $\frac{result[0]}{result[1]}$. Si aucune feuille n'est rencontrée, la fonction retourne 0 pour éviter une division par zéro.

Préfixe

La fonction prefixe permet de calculer combien de mots dans le dictionnaire ont un mot donné A comme préfixe. La fonction combine deux opérations : une navigation pour localiser le nœud correspondant au préfixe et un comptage des mots à partir de ce nœud.

La fonction publique prefixe s'exécute en deux étapes :

1. **Localisation du préfixe** :
    - La fonction goToNodeFromWord parcourt le trie pour trouver le nœud correspondant au dernier caractère du préfixe donné. Si ce nœud n'existe pas, cela signifie qu'aucun mot avec ce préfixe n'est présent, et la fonction retourne 0.
2. **Comptage des mots** :
    - Une fois le nœud correspondant localisé, la fonction appelle comptageMotsRec pour compter tous les mots descendants de ce nœud.

Suppression

La fonction suppression permet de retirer un mot spécifique d'un trie hybride si ce mot y figure. La fonction publique suppression fonctionne en deux étapes principales :

1. **Vérification de l'existence du mot** :
    - Avant d'entamer la suppression, la fonction vérifie si le mot à supprimer existe dans le trie à l'aide de la fonction recherche. Si le mot n'existe pas, une notification est affichée et aucun traitement n'est effectué.
2. **Suppression récursive** :

- Si le mot existe, la fonction appelle suppressionRec, qui supprime le mot en modifiant la structure de l'arbre de manière récursive. La suppression implique :
  - Enlever la marque de fin de mot au niveau du nœud correspondant.
  - Supprimer les nœuds inutiles (ceux qui ne participent plus à un mot) pour optimiser la structure du trie.

V. Fonctions complexes

Rééquilibrage du Trie hybride

La fonction de rééquilibrage est particulièrement utile pour assurer une arborescence efficace, afin de garantir des temps de calculs corrects pour l'ajout, la recherche et la suppression d'un mot dans l'arbre. L'idée principale de ce rééquilibrage est d'effectuer des rotations si le seuil de déséquilibre dépasse 1. Cependant, il était nécessaire de faire attention à respecter l'ordre lexicographique imposé par la structure ternaire. Nous n'effectuons donc jamais de rotation sur les sous-arbres du milieu, afin de respecter cette contrainte.

La fonction commence par vérifier si le sous-arbre gauche ou droit d'un nœud est déséquilibré (**la différence de hauteur entre les deux dépasse 1**).

Si le sous-arbre gauche est trop profond :

- Une rotation gauche est effectuée sur le sous-arbre gauche si le sous-arbre gauche est lui-même déséquilibré à droite.
- Une rotation droite est ensuite effectuée.

Si le sous-arbre droit est trop profond :

- Une rotation droite est effectuée sur le sous-arbre droit si le sous-arbre droit est lui-même déséquilibré à gauche.
- Une rotation gauche est ensuite effectuée.

Un attribut boolean balance a été ajouté à la fonction insert. Si vous souhaitez réaliser les insertions des mots en rééquilibrant l'arbre à chaque fois, vous pouvez ajouter un 4ème argument dans votre script bash, comme suit :

Par défaut, l'insertion s'effectue sans rééquilibrage :
inserer x nom_fichier.txt

Insertion sans rééquilibrage (explicite) :
inserer x nom_fichier.txt false

Insertion avec rééquilibrage (explicite) :
inserer x nom_fichier.txt true

VI. Complexités

Dans cette section, vous trouverez l'analyse des complexités de chaque fonction. Les complexités attendues seront mises en comparaison avec les complexités réelles obtenues, afin de nous permettre de vérifier si l'on retrouve les complexités énoncées au départ.

A. Patricia Tries

Complexités théoriques

Insertion

Cas général (trie compressé) :

- L'insertion est proportionnelle au nombre de nœuds à parcourir pour insérer le mot. Dans le pire des cas, un nœud correspond à un caractère du mot à insérer, donc la complexité devient **O(k)**, où k est la longueur du mot.

Cas déséquilibré (rare) :

- Même si le trie est déséquilibré, le parcours dans le trie reste le même pour insérer le mot. Donc dans le pire des cas la complexité reste en **O(k)**, où k est la longueur du mot.

Recherche

Cas général (trie compressé) :

- La recherche est proportionnelle au nombre de nœuds à parcourir pour trouver le mot. Dans le pire des cas, un nœud correspond à un caractère du mot à rechercher, donc la complexité devient **O(k)**, où k est la longueur du mot.

Cas déséquilibré (rare) :

- Même si le trie est déséquilibré, le parcours dans le trie reste le même pour trouver le mot. Donc dans le pire des cas la complexité reste en **O(k)**, où k est la longueur du mot.

Suppression

Cas général (trie compressé) :

- La suppression est proportionnelle au nombre de nœuds à parcourir pour trouver le mot. Dans le pire des cas, un nœud correspond à un caractère du mot à rechercher, donc la complexité devient **O(k)**, où k est la longueur du mot.

- On effectue la méthode de compression sur chaque nœud qui à été impacté par la suppression d'un autre nœud. Cette méthode, dans le pire des cas, compresse tous les fils du nœud à compresser. Donc à pour complexité en **O(n')**, où n' est le nombre de sous-noeuds.
- On obtient donc une complexité en **O(k*n')**

Cas déséquilibré (rare) :

- Même si le trie est déséquilibré, le parcours dans le trie reste le même pour supprimer le mot. Donc dans le pire des cas la complexité reste en **O(k*n')**, où k est la longueur du mot et n' le nombre de sous-noeuds à compresser.

Comptage des mots

Cas général:

- On effectue un parcours préfixe, c'est-à-dire qu'on visite chaque nœud du trie une seule fois.
- Si le trie contient n nœuds, la complexité est **O(n)**.

Liste des mots

Cas général:

- On effectue un parcours préfixe, c'est-à-dire qu'on visite chaque nœud du trie une seule fois.
- Si le trie contient n nœuds, la complexité est **O(n)**.

Comptage des Nils

Cas général:

- On effectue un parcours préfixe, c'est-à-dire qu'on visite chaque nœud du trie une seule fois.
- Si le trie contient n nœuds, la complexité est **O(n)**.

Hauteur

Cas général:

- On effectue un parcours préfixe, c'est-à-dire qu'on visite chaque nœud du trie une seule fois.
- Si le trie contient n nœuds, la complexité est **O(n)**.

Profondeur moyenne des feuilles

Cas général:

- On effectue un parcours préfixe, c'est-à-dire qu'on visite chaque nœud du trie une seule fois.
- Si le trie contient n nœuds, la complexité est **O(n)**.

Préfixe

- La recherche est proportionnelle au nombre de nœuds à parcourir pour trouver le préfixe. Dans le pire des cas, un nœud correspond à un caractère du mot à rechercher, donc la complexité devient **O(k)**, où k est la longueur du mot.
- Ensuite, on effectue un parcours préfixe, c'est-à-dire qu'on visite chaque nœud du sous-trie une seule fois.
- Si le sous-trie contient n ' nœuds, la complexité est **O(n')**.

On obtient donc une complexité de l'ordre de **O(k+n')**.

NB : Le patricia trie étant une structure compacte, contient moins de nœuds qu'un trie classique, donc rend son parcours plus rapide.

B. Tries hybrides

Un compteur par complexité a été ajouté en attribut et initialisé dans la classe HybridTrie. Les compteurs sont remis à zéro à chaque appel de fonction, afin d'obtenir les complexités pour chaque appel de fonctions.

Complexités théoriques

Insertion

Insertion sans rééquilibrage (balance = false) :

- La fonction parcourt l'arbre en fonction du nombre de caractères du mot à insérer, soit une complexité en **O(m)**, où m est la longueur du mot.
- Si l'arbre est déséquilibré, dans le pire des cas (arbre dégénéré), la complexité atteint **O(m × h)**, où h est la hauteur de l'arbre.

Insertion avec rééquilibrage (balance = true) :

- Après chaque insertion, la fonction appelle reequilibrer. Le rééquilibrage, basé sur la hauteur des sous-arbres, a une complexité en O(log n) dans le cas moyen, où n est le nombre total de mots dans le trie.

- La complexité totale devient donc **O(m × log n)**.

Recherche

Cas équilibré (trie bien équilibré) :

- À chaque caractère du mot, un seul sous-arbre est exploré (INF, EQ ou SUP).
- La recherche peut être contrainte à descendre de manière linéaire dans un sous-arbre (gauche ou droit).
- La complexité devient **O(h)**, où h est la hauteur de l'arbre.

Cas déséquilibré (arbre dégénéré) :

- Si l'arbre est linéaire (par exemple, tous les mots sont dans une seule branche), la complexité reste **O(h)**, mais le parcours est inefficace.

Comptage des mots

Cas équilibré (trie bien équilibré) :

- Chaque nœud de l'arbre est visité exactement une fois.
- Si l'arbre contient n nœuds, la complexité est **O(n)**.

Cas déséquilibré (arbre dégénéré) :

- Si l'arbre est linéaire (par exemple, tous les mots sont dans une seule branche), la complexité reste **O(n)**, mais le parcours est inefficace.

Liste des mots

Cas équilibré (trie bien équilibré) :

- Chaque nœud est visité exactement une fois.
- La création de la liste des mots a une complexité de **O(n)**, où n est le nombre de nœuds dans le trie.
- L'utilisation de StringBuilder pour accumuler les préfixes optimise les opérations sur les chaînes, évitant des copies inutiles.

Cas déséquilibré (arbre dégénéré) :

- Dans le pire des cas, où l'arbre est linéaire, la complexité reste **O(n)**. Cependant, le parcours est inefficace.

Comptage des Nils

Cas équilibré (trie bien équilibré) :

- Chaque nœud du trie est visité exactement une fois, et pour chaque nœud, les trois pointeurs sont inspectés.
- La complexité est donc **O(n)**, où n est le nombre de nœuds du trie.

Cas déséquilibré (arbre dégénéré) :

- Même dans le pire des cas, où l'arbre est dégénéré (linéaire), chaque nœud est toujours visité une fois, et chaque tableau de pointeurs est inspecté.
- La complexité reste **O(n)**.

Hauteur

Cas équilibré (trie bien équilibré) :

- Chaque nœud du trie est visité exactement une fois.
- La complexité est **O(n)**, où n est le nombre total de nœuds dans le trie.

Cas déséquilibré (arbre dégénéré) :

- Même dans un cas où l'arbre est linéaire (chaque nœud ayant un seul sous-arbre non vide), chaque nœud est toujours visité une fois.
- La complexité reste **O(n)**.

Profondeur moyenne des feuilles

Cas équilibré (trie bien équilibré) :

- Chaque nœud du trie est visité exactement une fois.
- La complexité est **O(n)**, où n est le nombre total de nœuds dans le trie.

Cas déséquilibré (arbre dégénéré) :

- Même dans un cas où l'arbre est linéaire (chaque nœud ayant un seul sous-arbre non vide), chaque nœud est toujours visité une fois.
- La complexité reste **O(n)**.

Préfixe

- La complexité de goToNodeFromWord est proportionnelle à la longueur du préfixe A, soit O(|A|).
- La complexité de comptageMotsRec dépend du nombre de nœuds fils du nœud correspondant au préfixe, soit O(s), où s est le nombre total de nœuds fils visités.
- La complexité totale est donc **O(|A|+s)**.

Suppression

- La fonction parcourt les caractères du mot pour localiser le nœud à modifier : O(|A|), où |A| est la longueur du mot.
- Elle vérifie et nettoie éventuellement les sous-arbres inutiles après la suppression : O(h), où h est la hauteur du trie.
- Complexité totale : **O(|A|+h)**.

Rééquilibrage

- Le calcul des hauteurs des sous-arbres nécessite O(h) pour chaque appel à hauteurNoeud.
- Chaque rotation ne coûte que O(1) à réaliser.
- La complexité totale est donc **O(h).**

Complexités réelles

Afin de comparer les complexités théoriques et réelles, nous avons effectué des tests sur l'ensemble des œuvres de Shakespeare afin d'observer les données expérimentales obtenues lors de l'appel de nos différentes fonctions. Afin de comparer ces données, nous allons les opposer dans le tableau ci-dessous pour une meilleure visualisation. Les calculs ont été réalisés sur l'ensemble des œuvres, les résultats obtenus sont la moyenne des calculs sur chaque œuvre.

Trie non équilibré

Fonctions	Complexités théoriques	Nombres d'opérations en moyenne par appel de fonction
Insertion	**O(|A| × h) = 6 + 14 = 20**	14 = **O(|A| × h)**
Recherche	**O(h) = 14**	14 = **O(h)**
Comptage des mots	**O(n) = 9,712**	9,712 = **O(n)**

Liste des mots	**O(n) = 9,712**	9,712 = **O(n)**
Comptage des Nils	**O(n) = 9,712**	29,138 ≃ 3n = **O(n)**
Hauteur	**O(n) = 9,712**	9,712 = **O(n)**
Profondeur moyenne	**O(n) = 9,712**	9,712 = **O(n)**
Suppression	**O(|A| + h) = 6 + 14 = 20**	43 ≃ **O(|A| + h)**

Hauteur moyenne dans les tries : 14
Nombre de noeuds moyens dans les tries : 9 712
Longueur moyenne d'un mot dans les oeuvres de Shakespeare : 6
$log_2(9712) \simeq 13.25$

Les résultats obtenus par nos expériences montrent que nos complexités pour les tries non équilibrés sont en accord avec les résultats attendus décrits précédemment.

Trie équilibré

Fonctions	Complexités théoriques	Nombres d'opérations en moyenne par appel de fonction
Insertion	**O(|A| × log n) = 6 + 13.25 ≃ 19.25**	12 = **O(|A| × h)**
Recherche	**O(h) = 12**	12 = **O(h)**
Comptage des mots	**O(n) = 9,712**	9,712 = **O(n)**
Liste des mots	**O(n) = 9,712**	9,712 = **O(n)**
Comptage des Nils	**O(n) = 9,712**	29,138 ≃ 3n = **O(n)**
Hauteur	**O(n) = 9,712**	9,712 = **O(n)**
Profondeur moyenne	**O(n) = 9,712**	9,712 = **O(n)**
Suppression	**O(|A| + h) = 4 + 12 = 16**	38 ≃ **O(|A| + h)**
Rééquilibrage	**O(h) = 12**	24 = 2h = **O(h)**

Hauteur moyenne dans les tries : 12
Nombre de noeuds moyens dans les tries : 9 712
Longueur moyenne d'un mot dans les oeuvres de Shakespeare : 6
$log_2(9712) \simeq 13.25$

Les résultats obtenus par nos expériences montrent que nos complexités pour les tries équilibrés sont en accord avec les résultats attendus décrits précédemment.

Nous pouvons noter que la différence est mince entre nos résultats pour les tries équilibrés et ceux qui ne le sont pas. Deux hypothèses peuvent venir expliquer ce phénomène :

- soit la fonction de rééquilibrage n'est pas assez efficace, auquel cas il faudrait trouver de nouvelles pistes pour l'améliorer ;
- soit les tries obtenus sans cette transformation ne sont pas spécialement très déséquilibrés, ce qui pourrait parfaitement expliquer cette légère différence dans les calculs effectifs des complexités ;

VII. Etude expérimentale

Pour évaluer les performances réelles des structures de données de type hybride et Patricia Trie, une étude expérimentale a été menée avec l'œuvre de Shakespeare afin de comparer leur efficacité en termes de temps de recherche, d'insertion, et des autres fonctions avancées. Cette analyse permet de mieux comprendre les avantages et limites de chaque approche dans des applications pratiques.

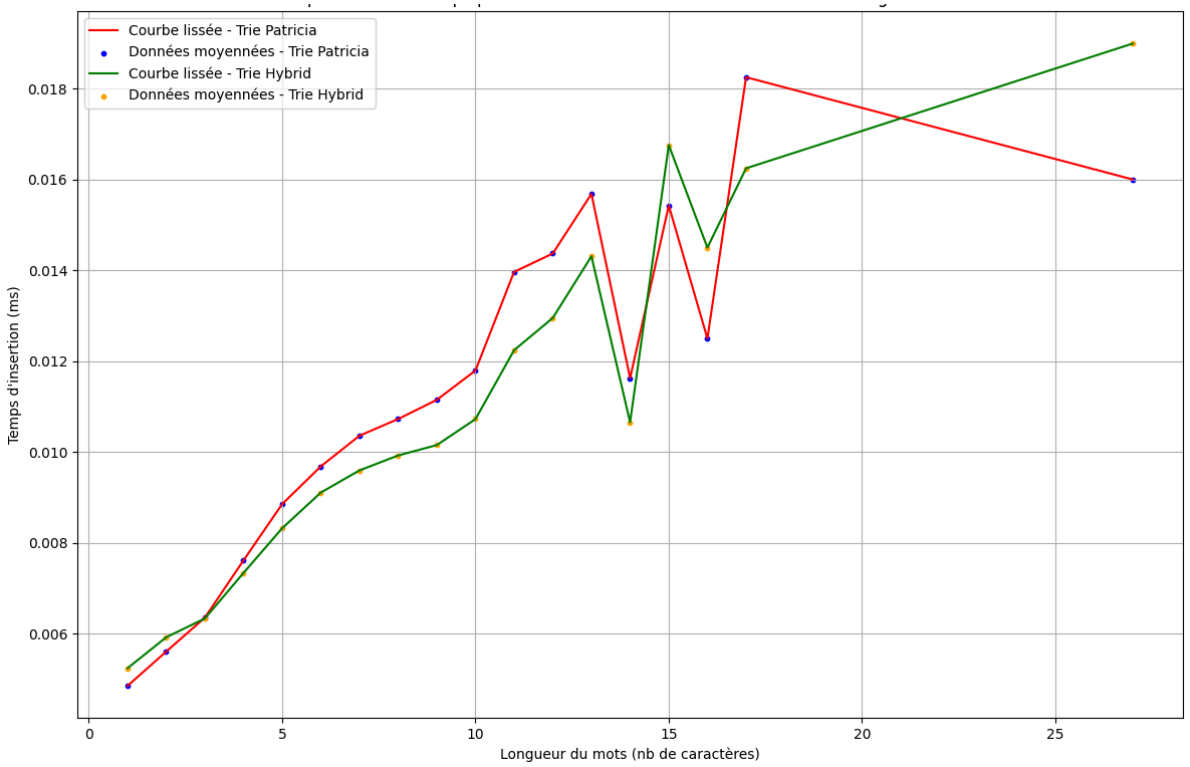

Figure 1 : Comparaison des temps pour la fonction d'insertion en fonction de la longueur du mot

On remarque que l'insertion des mots dans le trie Hybrid est légèrement plus efficace que celui du trie Patricia, mais cela est dû à une implémentation basée sur les TreeMap, pour organiser les fils des noeuds Patricia par ordre ASCII, ce qui fait légèrement perdre en performance celui-ci.

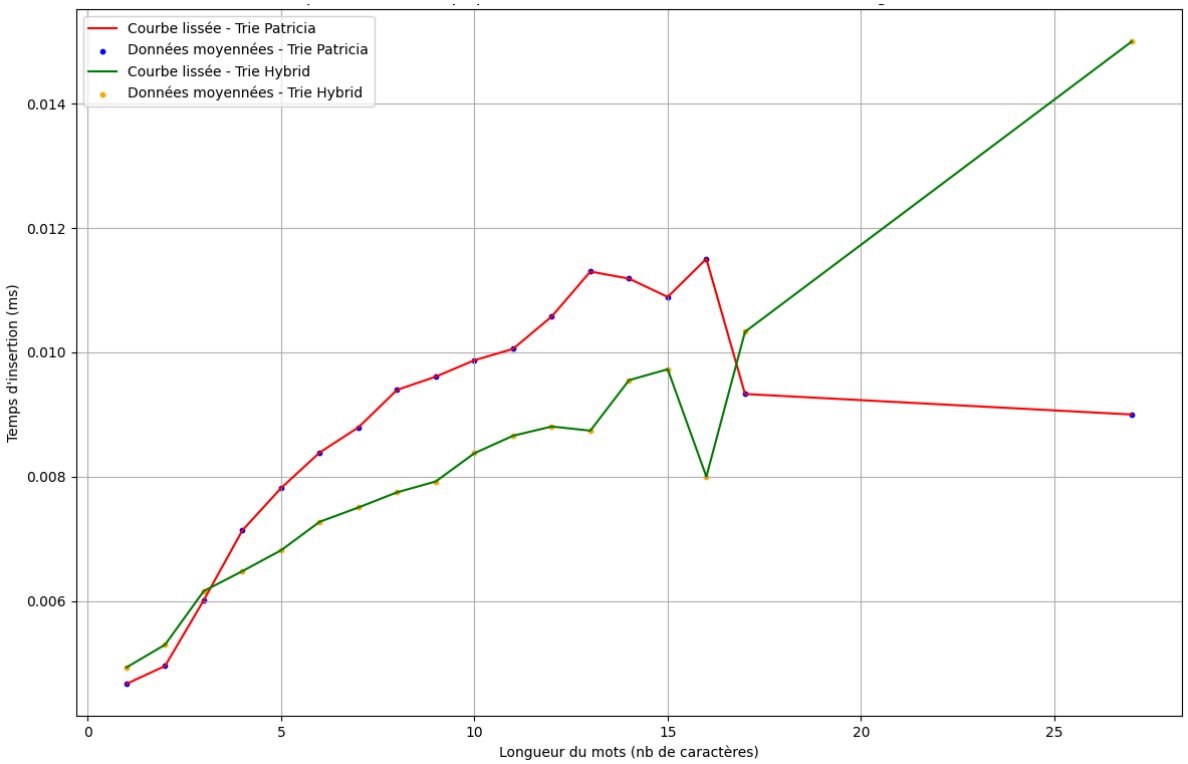

Figure 2 : Comparaison des temps pour la fonction de recherche en fonction de la longueur du mot

On remarque que l'insertion des mots dans le trie Hybrid est plus efficace que celle du trie Patricia. Excepté à partir d'un seuil ou les mots commencent à dépasser un certain nombre de caractères (ici 17). Néanmoins nous ne pensons pas que cela soit dû à l'implémentation du TreeMap.

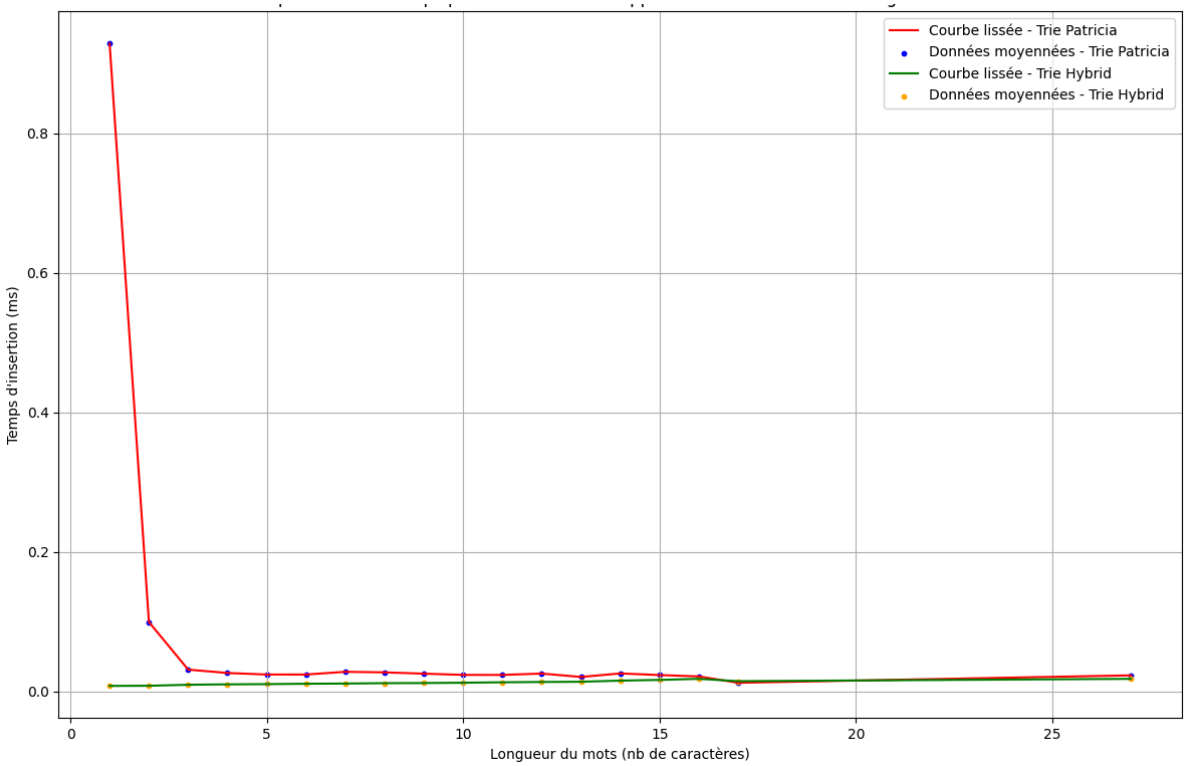

Figure 3 : Comparaison des temps pour la fonction de suppression en fonction de la longueur du mot

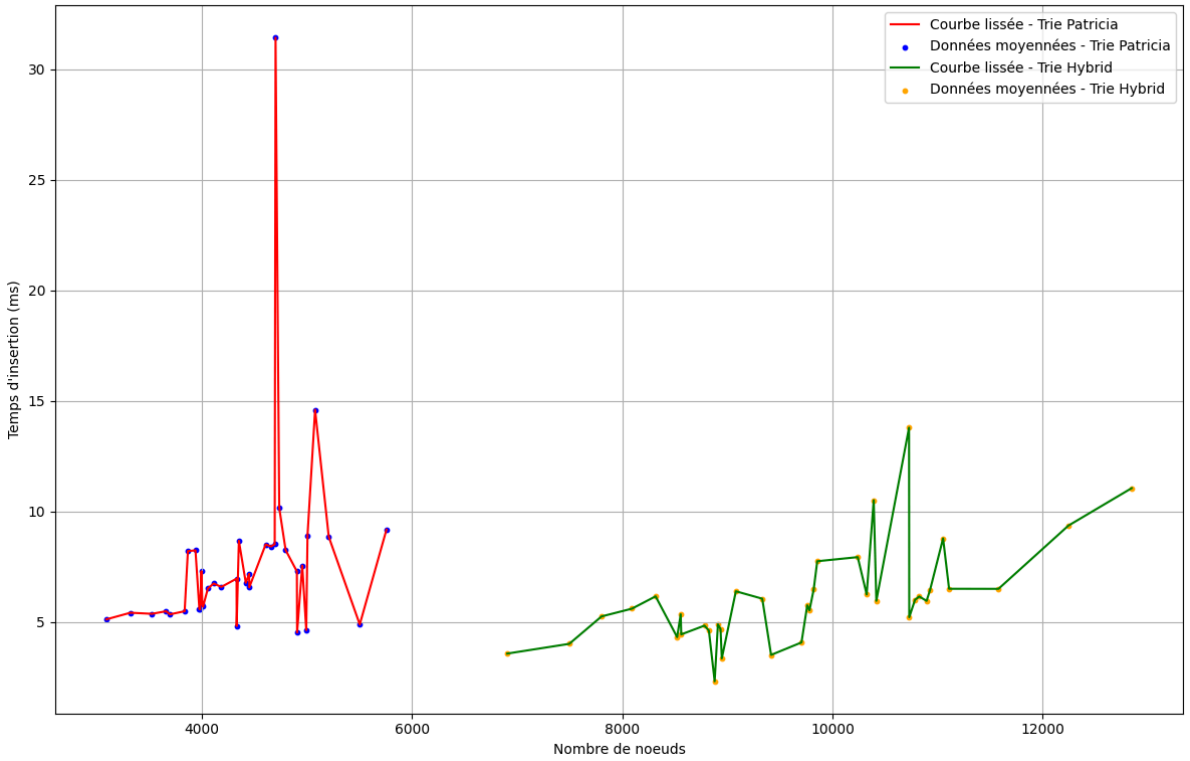

Figure 4 : Comparaison des temps pour la fonction countWords en fonction du nombre de noeuds dans le trie

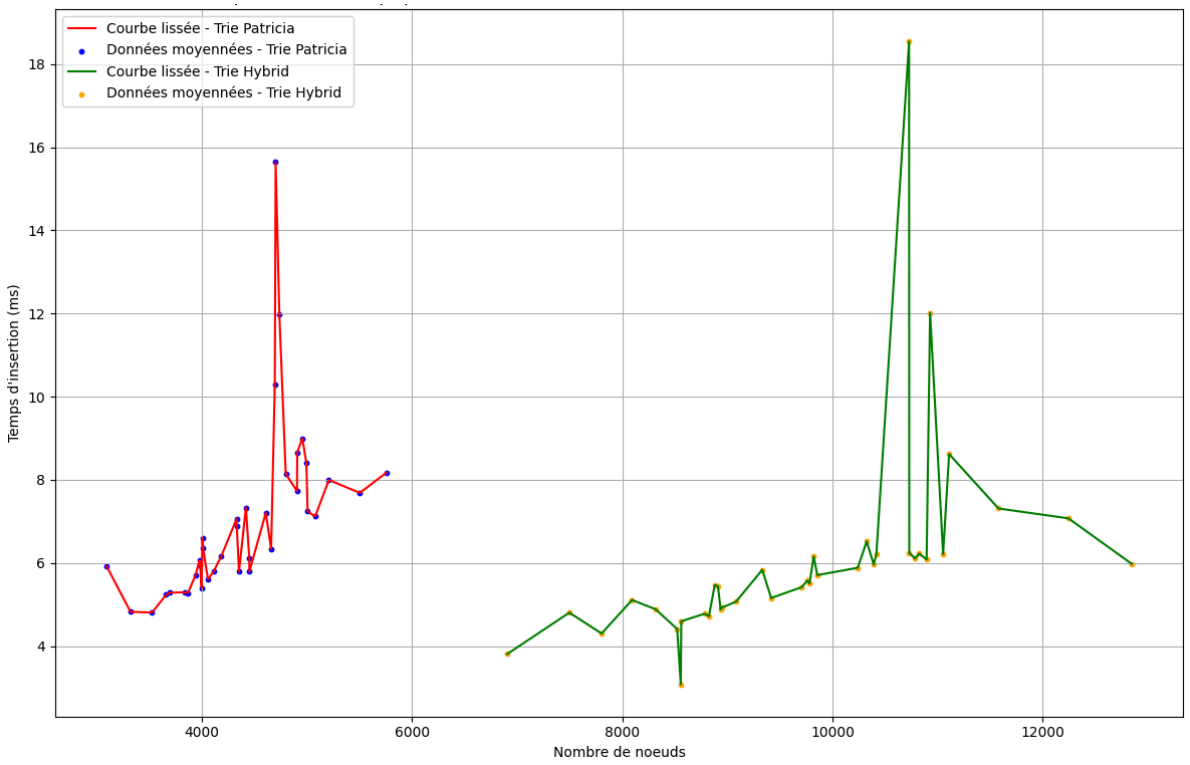

Figure 5 : Comparaison des temps pour la fonction countNils en fonction du nombre de noeuds dans le trie

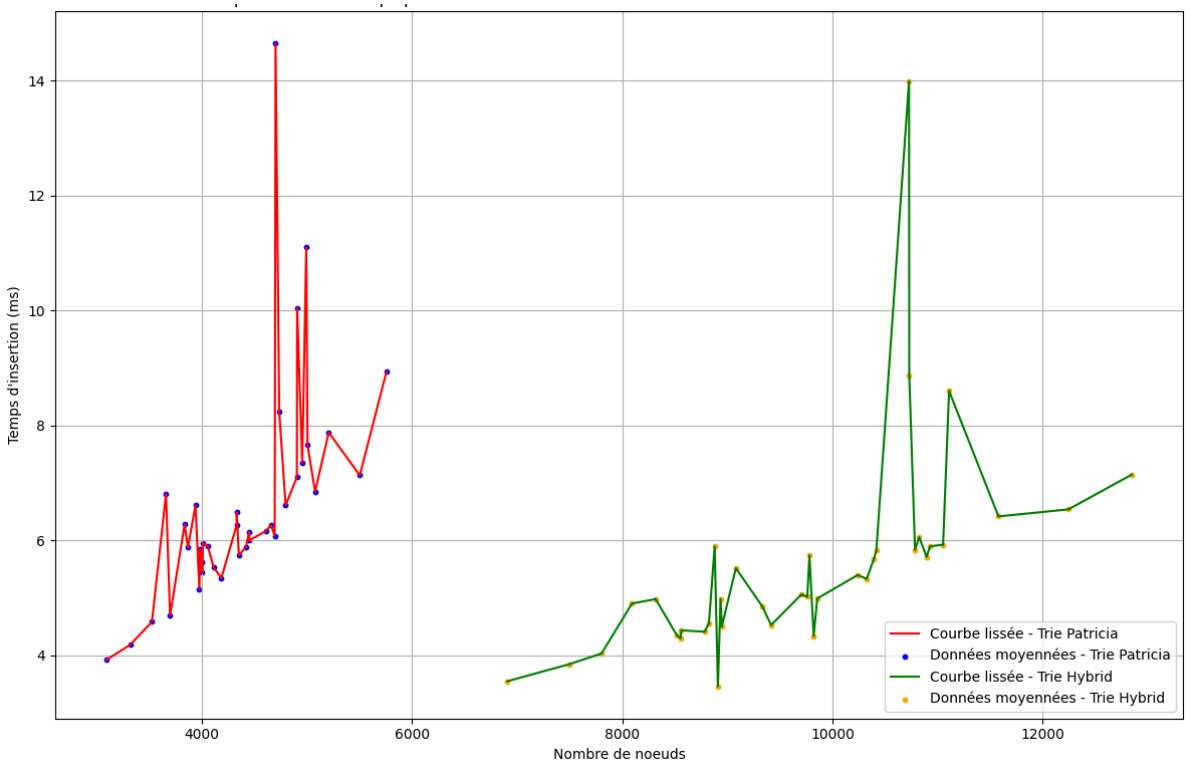

Figure 6 : Comparaison des temps pour la fonction height en fonction du nombre de noeuds dans le trie

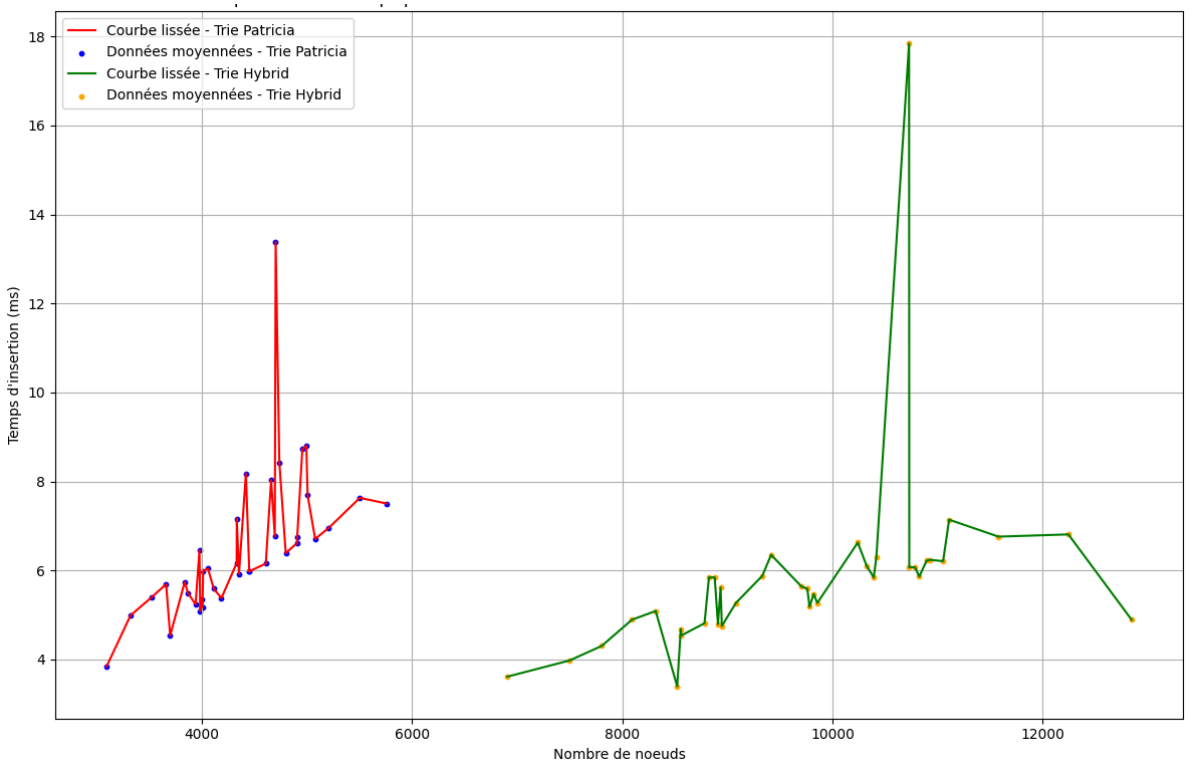

Figure 7 : Comparaison des temps pour la fonction meanHeight en fonction du nombre de noeuds dans le trie

On remarque que pour ces quatres fonctions, le trie Hybrid est plus efficace que le patricia trie malgré qu'il possède le double voir le triple de ses nœuds. Cela est toujours dû à l'implémentation avec le TreeMap et de ses utilisations. Malgré tout, ils obtiennent des performances similaires.

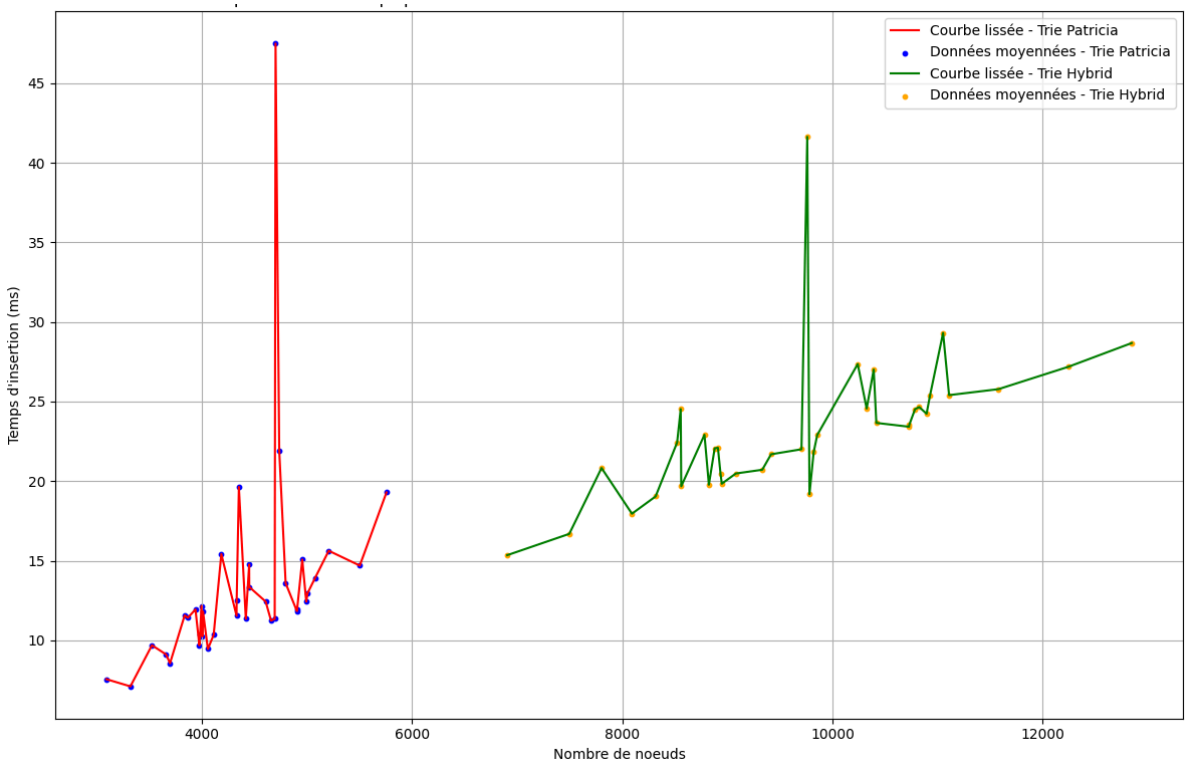

Figure 8 : Comparaison des temps pour la fonction listeMots en fonction du nombre de noeuds dans le trie

Ici, nous observons l'efficacité du trie Patricia, qui est quasiment deux fois plus rapide que le trie hybride, pour la moitié moins de nœuds. Ce qui revient à dire que son temps est proportionnel au nombre de nœuds dans le trie.

VIII. Conclusion

En conclusion, cette étude des structures de données de type trie, tant équilibré que non équilibré, montre une adéquation entre les complexités théoriques et les résultats expérimentaux obtenus à partir de l'analyse des œuvres de Shakespeare. Les essais ont permis de valider que, dans les deux cas, les complexités sont globalement cohérentes avec les prédictions théoriques, en particulier en ce qui concerne l'insertion, la recherche, et le comptage des mots, tant pour les tries équilibrés que non équilibrés.

Cependant, les résultats expérimentaux ont également révélé une légère différence de performance entre les deux types de tries. Les tries équilibrés, malgré leur coût supplémentaire lié au rééquilibrage, ne montrent qu'une amélioration marginale par rapport aux tries non équilibrés. Cela peut être dû à une faible déséquilibre dans l'arbre sans rééquilibrage ou à une inefficacité du mécanisme de rééquilibrage lui-même. En revanche, les performances en termes de suppression et de comptage des mots ou des nœuds montrent des résultats proches, ce qui met en évidence la robustesse des deux structures, mais aussi la faible différence de gain pratique.

L’étude des performances des différents types de tries, y compris le Patricia trie et le trie hybride, met en lumière les avantages et limites de chaque approche. Le trie hybride, utilisant une structure basée sur TreeMap, se révèle légèrement plus rapide pour certaines opérations comme l’insertion et la recherche, mais avec des résultats moins optimisés pour des fonctions comme la liste des mots. Le Patricia trie, bien qu'un peu moins performant sur certaines fonctions, se montre plus efficace pour d’autres, notamment en termes de gestion du nombre de nœuds et de la complexité relative à la taille de l’arbre.

Ainsi, bien que la théorie suggère des gains de performance avec des tries équilibrés, les tests expérimentaux révèlent que, dans de nombreux cas, les différences sont minimales, ce qui ouvre la réflexion sur l'optimisation du processus de rééquilibrage et la prise en compte de la structure des données dans des applications pratiques. En définitive, le choix entre un trie équilibré ou non dépendra largement des besoins spécifiques en termes de performance et de la nature des données à traiter.