SORBONNE UNIVERSITÉ

Devoir de Programmation (ALGAV)

19.12.2024

—

CIDERE Taryck 21400635

KESSAL Abdelmadjid 21402566

Introduction

Ce projet a pour objectif de représenter un dictionnaire de mot à l'aide de deux structures de données différentes les Patricia-Tries et les Tries Hybrides et de mener une étude expérimentale afin de mettre en avant les avantages et les inconvénients de chacun des modèles. Pour mener à bien ce projet, nous avons choisi d'utiliser python.

Dans les sections suivantes, nous présenterons les structures de données de même que les algorithmes que nous avons implémentés, ainsi que les mesures effectuées sur ces derniers.

Ce projet est l'occasion de mieux comprendre les concepts abordés en classe ainsi que d'approfondir notre compréhension pratique de ces structures de données.

Structure Patricia-Tries:

Définition des arbres Patricia :

- **Un Patricia-Trie** (ou Patricia Tree) est une structure de données dérivée du Trie classique, conçue pour stocker efficacement des chaînes de caractères (ou des clés). Contrairement au Trie classique, où chaque caractère est représenté par un nœud, le Patricia-Trie regroupe les caractères communs dans des nœuds, ce qui réduit la mémoire utilisée. Cela le rend particulièrement adapté lorsque les clés ont beaucoup de préfixes communs.

Pourquoi utiliser les arbres Patricia :

- Ils sont **plus rapides** pour chercher un mot, car l'arbre est moins profond.
- Ils prennent **moins de place** en mémoire que les tries classiques.

Choix du caractère pour indiquer la fin d'un mot :

Le caractère # a été choisi pour marquer la fin d'un mot dans le Patricia-Trie.

En effet, selon des analyses statistiques de textes en langue naturelle, la probabilité d'apparition du caractère # est inférieure à 0,01 %, car il est principalement utilisé dans des contextes spécifiques, comme les hashtags ou le marquage dans certains systèmes.

Cela garantit qu'il n'interfère pas avec les mots représentés. De plus, ce choix préserve la compatibilité avec les 127 autres caractères ASCII pour la construction des mots, tout en simplifiant la logique d'implémentation grâce à sa lisibilité et à son usage clair pour indiquer la fin d'un mot.

Structure et choix d'implémentation :

1. La classe PatriciaNode:
    - Chaque nœud est représenté par un dictionnaire labels, où :
        - La **clé** représente un **préfixe commun** partagé par plusieurs mots ou segments de mots, ce préfixe commun permet de regrouper plusieurs branches dans un même nœud, réduisant ainsi la profondeur de l'arbre et optimisant l'espace mémoire.
        - La **valeur** associée à chaque clé est un lien (référence) vers un autre nœud, ce nœud contient à son tour les **fils** qui continuent à partir de ce préfixe commun. En d'autres termes, il représente tous les mots ou segments de mots qui partagent ce préfixe comme point de départ.
    - Cela permet de conserver plusieurs labels dans un même nœud, améliorant ainsi la compacité de l'arbre.

2. La classe PatriciaTrie:
    - Contient un attribut root qui représente la racine de l'arbre qui est de type PatriciaNode.
    - L'attribut comparaisons est utilisé pour suivre le nombre d'opérations comparatives réalisées, utile pour l'analyse des performances (par exemple, pour mesurer l'efficacité lors des recherches ou insertions, ou pour calculer le nombre de nœuds visites lorsqu'on liste les mots).

Primitives utiles pour les arbres patricia:

Insertion (insererMot)

- **Rôle :** Ajouter un mot dans le Patricia-Trie.
- **Étapes principales :**
    1. Parcourir les nœuds à partir de la racine.
    2. Comparer le mot à chaque label pour trouver un préfixe commun.
    3. Si un préfixe commun existe, diviser le label si nécessaire et insérer le mot restant.
    4. Si aucun préfixe commun n'existe, ajouter le mot comme un nouveau label.
    5. Marquer la fin du mot (souvent avec un caractère spécial comme "#").

Longueur du préfixe commun (commonPrefixLength)

- **Rôle :** Trouver la longueur du préfixe commun entre deux chaînes.
- **Étapes principales :**
    1. Comparer les caractères des deux chaînes un par un.
    2. Compter combien de caractères sont identiques à partir du début.
    3. Retourner la longueur du préfixe commun.

Fonctions avancées :

Fonction Recherche (Arbre, mot) :

Complexité (en terme de comparaisons):

1. **Appel à common_prefix_length(label, mot):**
   Cette fonction compare les deux chaînes caractère par caractère, jusqu'à trouver une différence ou jusqu'à la fin du plus court des deux. Sa complexité en nombre de comparaisons est :
   O(min(len(label),len(mot)))
   Dans le pire cas, on peut considérer que les segments label et mot sont comparés

presque entièrement, ce qui peut aller jusqu'à O(n) si mot est encore de longueur n et que le label est d'une longueur similaire.

2. **Au niveau d'un seul nœud :**
    - Le nœud a potentiellement b enfants (où b est le nombre d'enfants par nœud).
    - On appelle common_prefix_length jusqu'à trouver un préfixe commun ou jusqu'à épuiser tous les enfants.
    - Dans le pire cas, on pourrait faire b appels à common_prefix_length, chacun pouvant faire jusqu'à O(n) comparaisons de caractères.
      Ainsi, dans le pire des cas, à un nœud donné, on peut faire jusqu'à O(b * n) comparaisons.
3. **Sur toute la hauteur de l'arbre h:**
   La recherche pourrait descendre jusqu'à h niveaux (dans le pire des cas, le mot correspond à un chemin assez long).
   À chaque niveau, dans le pire des cas, on a O(b*n) comparaisons.
   Cela donne une complexité en termes de comparaisons de caractères de :
   O(h*b*n).

Meilleur cas :
On trouve très vite le bon label (par exemple, au premier essai à chaque nœud), et common_prefix_length s'arrête rapidement. Dans ce cas, le nombre de comparaisons est proche de O(n) car on avance rapidement dans le mot et on ne multiplie pas les échecs.

Cas moyen :
Généralement, la recherche ne nécessite pas de comparer le mot à tous les enfants de tous les nœuds. On trouve souvent le bon chemin sans trop de tentatives.
De plus, à mesure qu'on trouve des préfixes communs, le mot se raccourcit, ce qui limite le nombre total de comparaisons. Ainsi, en pratique, la recherche est souvent O(n) dans le cas moyen, car chaque caractère du mot est comparé un nombre limité de fois.

Conclusion

- **Pire cas théorique (nombre de comparaisons de caractères) :**
  O(h*b*n)
- **Cas moyen/pratique :**
  Environ O(n), car chaque caractère du mot est consommé au fur et à mesure.

Fonction ComptageMots (arbre, mot):

Complexité (en terme de noeuds visités):

Cette fonction parcourt tout le Patricia-Trie à partir de la racine et compte le nombre total de mots marqués par le symbole "#".

La fonction est un parcours en profondeur de l'arbre :

1. Elle visite un nœud (opération O(1) pour vérifier "#").
2. Puis elle appelle la même fonction sur chacun de ses enfants.
3. Chacun de ces enfants fera la même chose, parcourant à leur tour leurs propres enfants, et ainsi de suite.

Finalement, chaque nœud est visité exactement une fois.

La complexité temporelle de comptageMots est donc **O(N)**, où N est le nombre total de nœuds dans le Patricia-Trie.

Fonction ListeMots (arbre) :

Complexité (en terme de noeuds visités):

Pareil que comptageMots(), notre fonction parcourt tout le Patricia-Trie à partir de la racine

La complexité temporelle de ListeMots est donc **O(N)**, où N est le nombre total de nœuds dans le Patricia-Trie.

Pareil pour les fonctions ComptageNil , Hauteur et Profondeur Moyenne, tous les nœuds de l'arbre devront être visités.

Suppression (Arbre, mot):

Complexité (en terme de comparaisons):

La complexité de la suppression d'un mot dans un Patricia-Trie peut s'analyser en fonction de la profondeur maximale de l'arbre, notée *h*, et du nombre moyen de labels par nœud, noté *b* :

- **Meilleur cas :** Le mot suit un chemin unique dans l'arbre et chaque nœud intermédiaire est supprimé sans avoir à comparer plusieurs labels. Le nombre de comparaisons est alors **O(h)**.
- **Cas moyen :** En général, la recherche du mot suit le chemin correspondant dans le Patricia-Trie, et le nombre de comparaisons reste proportionnel à la profondeur du chemin. Le coût moyen est donc également **O(h)**.
- **Pire cas :** Si, à chaque niveau, il faut comparer plusieurs labels avant de trouver le bon (ou de constater l'absence du mot), le nombre de comparaisons augmente. Dans ce scénario, le coût peut aller jusqu'à **O(h * b)**, où *b* représente le nombre moyen de labels par nœud.

En pratique, grâce à la compression du Patricia-Trie, la profondeur *h* est généralement réduite, et le nombre moyen de labels par nœud *b* reste faible. Ainsi, la complexité réelle observée se situe généralement proche de **O(h)**.

Prefixe (Arbre, mot):

Étapes principales et leur complexité

1. Recherche du préfixe (mot)

La recherche suit la logique suivante :

- À chaque nœud, on compare le préfixe restant (mot) avec les labels des enfants en utilisant common_prefix_length.
- Comparer un mot avec un label a une complexité de **O(M)** au pire (si le label est long et mot est long).
- Si un nœud a **b** enfants, on compare potentiellement **b** labels dans le pire cas.
- La recherche descend jusqu'à une profondeur **h**, correspondant à la hauteur de l'arbre.

Complexité pour cette étape :

$O(h \cdot b \cdot M)$

2. Comptage des mots (comptageMots)

Une fois que le nœud correspondant au préfixe mot est trouvé, on doit compter tous les mots dans le sous-arbre correspondant :

- La fonction comptageMots parcourt tous les nœuds descendants du nœud trouvé. Si ce nœud a **N'** nœuds descendants, alors la complexité du comptage est **O(N')**.

Complexité pour cette étape :

$O(N')$

Complexité totale

La recherche du préfixe et le comptage des mots sont exécutés l'un après l'autre. La complexité totale est donc la somme des deux :

$O(h \cdot b \cdot M + N')$

Fusion de deux arbres patricia:

Fonction fusion_nodes(self, node1, node2)

Cette fonction fusionne deux **nœuds** de Patricia-Tries (node1 et node2) ainsi que leurs sous-arbres respectifs. Elle gère la fusion de leurs labels (segments de mots) et leurs enfants.

Étapes nécessaires pour fusionner deux arbres Patricia

1. Vérifier les cas de base :
    - Si l'un des nœuds est vide (node1 ou node2), retourner l'autre :
        - Si node1 est vide, retourner node2.
        - Si node2 est vide, retourner node1.
2. Parcourir les labels de node2 :
    - Pour chaque label et son nœud enfant dans node2, essayer de le fusionner avec un label de node1.
3. Détecter les préfixes communs :
    - Pour chaque label de node2, comparer avec les labels de node1
        - Utiliser la fonction common_prefix_length pour trouver le préfixe commun entre deux labels.
4. Cas avec préfixe commun :
    - Si un préfixe commun est détecté entre un label de node2 (label2) et un label de node1 (label1) :
        - Découper les labels en trois parties :
            1. Préfixe commun : La partie commune entre les deux labels.
            2. Reste de label1 : La portion restante du label de node1 après le préfixe commun.
            3. Reste de label2 : La portion restante du label de node2 après le préfixe commun.
        - Ajouter un nœud pour le préfixe commun dans node1 si ce nœud n'existe pas déjà.
5. Traiter les restes après le préfixe commun :
    - Si le reste de label1 existe :
        - Déplacer son nœud enfant sous le nœud représentant le préfixe commun dans node1.
    - Si le reste de label2 existe :
        - Ajouter son nœud enfant sous le nœud représentant le préfixe commun.

        - Si les deux labels sont entièrement consommés (le préfixe est identique aux deux labels) :
            - Fusionner récursivement les sous-arbres associés au préfixe commun.
6. Cas sans préfixe commun :
    - Si aucun préfixe commun n'est trouvé pour un label de node2 :
        - Ajouter directement ce label et son nœud enfant à node1.
7. Fusion récursive :
    - Répéter les étapes ci-dessus pour chaque nœud enfant, en appelant la fonction de manière récursive.
8. Retourner l'arbre fusionné :
    - Après avoir traité tous les labels de node2, retourner node1 comme résultat de la fusion.

Complexité en termes de nœuds visités :

Chaque nœud des deux arbres est visité une fois, soit pour comparer ses labels, soit pour fusionner ses sous-arbres.

- Nombre total de nœuds visités : O(N1 + N2), où N1 et N2 sont les nombres de nœuds dans les deux arbres.

Complexité en termes de comparaisons

1. **À chaque niveau de l'arbre :**
    - Comparer les labels de node2 avec ceux de node1 nécessite **O(n1 * n2)** comparaisons.
    - Chaque comparaison utilise la fonction common_prefix_length, qui effectue jusqu'à **O(len(label))** comparaisons par paire.
2. **Sur tous les niveaux :**
    - Si les deux arbres contiennent un total de N1 et N2 nœuds, chaque nœud peut entraîner une comparaison proportionnelle au nombre de labels à ce niveau.
    - Dans le pire des cas (lorsque les arbres sont déséquilibrés), la complexité totale des comparaisons peut atteindre **O(N1 * N2 * L)**, où :
        - N1 : Nombre total de nœuds dans le premier arbre.
        - N2 : Nombre total de nœuds dans le second arbre.
        - L : Longueur moyenne des labels.

Trie Hybride

Classe Trie Hybride

Afin de manipuler les trie hybride dans la suite du projet, nous avons d'abord créé une classe HybridTrie représentant un tel trie.
Cette classe ne possède pas de méthode mais seulement des attributs.

Attributs :

- Value (str) : Valeur du Noeud
- position (int) : Indique si le mot est terminé
- childInf (Trie Hybride) : L'enfant inférieur
- childEq (Trie Hybride) : L'enfant égal
- childSup (Trie Hybride) : L'enfant supérieur

Primitives de bases

Sur les clefs

- prem(cle) : str –> str - Renvoie le premier caractère de la clé.
- reste(cle) : str -> str - Renvoie la clé privée de son premier caractère.
- car(cle, i) : str * i -> str - Renvoie le i-ème caractère de la clé.
- lgueur(cle) : str -> int - Renvoie le nombre de caractères de la clé.

Sur les Tries Hybrides

- TH_Vide() : -> Trie Hybride - Renvoie le trie vide.
- EstVide(A) : Trie Hybride -> bool - Renvoie vrai si le trie est vide et faux sinon.
- Val(A) : Trie Hybride -> int - Renvoie la position stockée à la racine de l'arbre.
- ValVide() : -> None - Renvoie la position d'un nœud par défaut.
- Eq(A) : Trie Hybride -> Trie Hybride - Renvoie l'enfant centrale (Eq) de A.
- Sup(A) : Trie Hybride -> Trie Hybride - Renvoie l'enfant supérieur (Sup) de A.
- Inf(A) : Trie Hybride -> Trie Hybride - Renvoie l'enfant inférieur (Inf) de A.
- Racine(A) : Trie Hybride -> str - Renvoie la valeur stockée à ce nœud.
- SousArbre(A, i) : Trie Hybride -> Trie Hybride : Renvoie le i-ème fils du nœud.
- EnfantsSauf(A, i) : Trie Hybride -> List Trie Hybride : Renvoie la liste de tous les fils direct sauf le i-ème.
- TrieH(value, inf, eq, sup, position) :
  str * Trie Hybride * Trie Hybride * Trie Hybride * int -> Trie Hybride - construit un nœud de Trie Hybride à partir des arguments et le renvoie.

Fonctions avancées

Recherche(A, c) : Trie Hybride * mot -> Booléen

Cette fonction cherche dans le trie hybride A le mot c, si ce dernier est trouvé elle renvoie True sinon False.

Fonctionnement

Dans un premier temps, elle convertit les caractères majuscules du mot en minuscule. Puis la recherche commence à travers un appelle à la fonction **lookup(A, c)**

lookup est une fonction récursive qui avant de faire quoi que ce soit vérifie si le mot recherché n'est pas le mot vide.

- Si le mot est vide alors cela signifie qu'il n'a pas été trouvé dans le trie, on renvoie alors False.
- Si le mot n'est pas vide et que le Trie ne l'est pas non plus alors on vérifie si le mot est composé d'un unique caractère, si ce dernier correspond à la valeur du nœud courant et si le nœud courant marque la fin d'un mot du trie.
    - Si tous ces tests sont évalués à vrai, on a la confirmation de la présence du mot dans le trie, on renvoie donc True.
    - Sinon on sait que si le mot existe, il n'a pas encore été totalement atteint, en fonction de la valeur du premier caractère (p) du mot on réalise les opérations suivantes :
        - p < Racine(A) : On continue rappelle lookup sur l'arbre inférieur en lui passant le mot inchangé.
        - p > Racine(A) : On continue rappelle lookup sur l'arbre supérieur en lui passant le mot inchangé.
        - p est égal à Racine(A) : On sait qu'un caractère du mot à été pu être lu, on continue récursivement la recherche au travers d'un autre appel à lookup en lui passant cette fois l'arbre égal (Eq(A)) et le mot privé de son premier caractère.
- Sinon l'arbre est vide et le mot n'y est forcément pas, on peut donc renvoyer False.

Une fois l'exécution de **lookup** terminée, **Recherche** renvoie la valeur de retour de **lookup**.

ComptageMots(A) : Trie Hybride -> int

Cette fonction compte le nombre total de mots présents dans un trie hybride.

Fonctionnement

Cette fonction utilise un compteur indiquant le nombre de mot lu dans les sous arbres de l'arbre qui lui a été passé en argument. Pour incrémenter le compteur on procède de la manière suivante :

- Si le nœud courant est vide : Alors il n'y a aucun mot à lire, la fonction renvoie alors 0.
- Sinon si le nœud courant marque la fin d'un mot, on incrémente le compteur de 1.

Que le nœud courant marque la fin du mot ou non, on procède ensuite par rappeler récursivement la fonction **ComptageMots** sur chacun des arbres enfants.
Le résultat de chacun de ces appels est alors ajouté à la valeur courante du compteur.
On peut enfin renvoyer la valeur valeur du compteur.

ListeMots(A) : Trie Hybride -> List str

Cette fonction renvoie une liste contenant chacun des mots du trie hybride qui lui a été passé en argument dans l'ordre alphabétique.

Fonctionnement

Une fois appelé, **ListeMots** appel directement la fonction **CreateWord**, qui à partir du préfixe et du trie qui lui a été passé reconstruit tous les mots du trie en ajoutant le préfixe au début de ces derniers. Cette fonction accepte deux argument :

- Trie Hybride : Correspondant au trie duquel les mots seront extraits.
- préfixe (facultatif) : le préfixe à ajouter au début de chaque mot créé.

CreateWord fonctionne ainsi :
Elle commence tout d'abord par créer une liste (res) qui contiendra tous les mots trouvés dans le trie, puis elle vérifie si le noeud courant est vide :

- Si ce dernier l'est alors la fonction retourne simplement la liste vide car aucun mot ne peut s'y trouver.
- Sinon, on reconstruit par concaténation entre le préfixe reçue et le caractère du nœud courant le potentiel mot à ajouter à la liste.
  On rappelle d'abord récursivement la fonction **CreateWord** sur le fils gauche du nœud et on ajoute le résultat obtenu à la la liste courante.
  Ensuite on vérifie si le noeud courant indique la fin d'un mot, si oui alors on ajoute le mot précédemment créé à la liste.
  Enfin on rappelle récursivement la fonction sur l'enfant égal (Eq(A)) et ajoutons le résultats à la liste et juste après on réalise la même chose sur le nœud supérieur.

Ainsi la liste contient tous les mots de cet arbre dans l'ordre alphabétique et on retourne la liste tel quel. Le résultat de **CreateWord** et immédiatement renvoyé par **ListeMots**.

ComptageNil(A) : Trie Hybride -> int

Cette fonction renvoie le nombre de nœuds vides dans un trie.

Fonctionnement

On commence par initialiser un compteur (res) à 0 et ensuite nous procédons comme suit :

- Si le nœud est vide, nous retournons 1.
- Sinon, nous rappelons récursivement la fonction sur ses trois enfants et additionnons les résultats dans res avant de le retourner.

Hauteur(A) : Trie Hybride -> int

Cette fonction renvoie la longueur de la plus longue branche de l'arbre.

Fonctionnement

On commence tout d'abord par vérifier si le noeud courant est vide

- Si oui alors nous renvoyons 0.
- Sinon nous rappelons récursivement la fonction Hauteur sur chacun de ses fils et retournons : 1 + le maximum des résultats obtenus.

ProfondeurMoyenne(A) : Trie Hybride -> int

Cette fonction calcule et renvoie la profondeur moyenne de l'arbre lui étant passé dans en argument.

Fonctionnement

On commence ici par récupérer la liste de toutes les différentes profondeurs de l'arbre en appelant la fonction ToutesProfondeurs, cette dernière prend un ou deux arguments à savoir :

- L'arbre à explorer
- La hauteur courante (facultatif)

Elle fonctionne similairement à la fonction Hauteur précédemment décrite cependant ici nous manipulons une liste et nous effectuons le traitement suivant :

- Si l’arbre est vide alors nous renvoyons la liste vide.
- Sinon nous incrémenter la hauteur courante, l’ajoutons à la liste et appelons récursivement la fonction ToutesProfondeurs sur chacun des nœuds enfants et renvoyons la liste courante concaténée avec les listes résultant de l’appelle récursifs sur les différentes enfants.

Une fois le traitement de ToutesProfondeurs terminé et le résultat stocké, nous procédons au calcul de la moyenne en sommant dans une variable res initialement égale à 0 le résultat de l’addition de tous les éléments de la liste des profondeurs.
Enfin nous renvoyons la moyenne (res/taille de la liste) si la liste n’était pas vide sinon 0.

Prefixe(A, mot) : Trie Hybride * str -> int

Cette fonction renvoie le nombre de mot du trie hybride ayant pour préfixe le mot lui étant passé en argument.

Fonctionnement

On commence par d’abord chercher le nœud marquant la fin du préfixe à l’aide de la fonction ChercherNoeudPrefixe. Cette dernière fonctionne de la même manière que la fonction Recherche à l'exception qu’ici plutôt que de renvoyer un booléen indiquant si oui ou non le mot a été trouvé, on renvoie le noeud marquant la fin du mot indépendamment du fait qu’il soit un mot terminé (avec une valeur définie : Val(A)) ou non. Si la recherche échoue, la fonction renvoie le trie hybride vide (None).
Le résultat de l’appelle à ChercherNoeudPrefixe et stocker dans une variable node qui est ensuite comparer au noeud vide :

- Si le nœud est vide alors on retourne 0 car il n’existe donc aucun mot ayant ce préfixe.
- Si le nœud n’est pas vide, on appelle alors la fonction CreateWord précédemment décrite et renvoyons la taille de la liste trouvée.

Suppression(A, mot) : Trie Hybride * str -> Trie Hybride

Cette fonction recherche dans un trie hybride un mot et s’il s’y trouve, le supprime.

Fonctionnement

On commence par s’assurer que l’arbre n’est pas vide, que le mot n’est pas vide et qu’il soit bien présent dans l’arbre.

- Si ces conditions ne sont pas vraies, alors nous retournons immédiatement l’arbre courant.

Dans le cas où le test est faux, nous savons que nous devrons procéder à une suppression. Alors nous stockons dans des variables les informations suivantes :

- p : contient le premier caractère du mot courant.
- racine : contient le caractère stocké dans le nœud courant.
- val : contient la valeur du nœud.
- inf : contient le fils inférieur du nœud.
- eq : contient le fils égal du nœud.
- sup : contient le fils supérieur du nœud.

Nous procédons ensuite comme suit :

- Si p < racine :
  Nous rappelons récursivement la fonction Suppression sur l'arbre inférieur et en lui donnant le mot inchangé et stockons son résultat dans une variable new_inf.
  Puis nous vérifions si le nœud courant marque la fin d'un mot de l'arbre et si le potentiel nouveau nœud inférieur (new_inf) ainsi que les nœuds eq et sup sont vides.
  - Si oui, alors il n'existe aucun mot dans l'arbre courant. Nous renvoyons le trie vide.
  - Sinon on reconstruit un trie en appelant TrieH avec les arguments suivants :
    - racine
    - new_inf
    - eq
    - sup
    - val
- Si p > racine :
  Nous rappelons récursivement la fonction Suppression sur l'arbre supérieur et en lui donnant le mot inchangé et stockons son résultat dans une variable new_sup.
  Puis nous vérifions si le nœud courant marque la fin d'un mot de l'arbre et si le potentiel nouveau nœud supérieur (new_sup) ainsi que les nœuds eq et inf sont vide.
  - Si oui, alors il n'existe aucun mot dans l'arbre courant. Nous renvoyons le trie vide.
  - Sinon on reconstruit un trie en appelant TrieH avec les arguments suivants :
    - racine
    - inf
    - eq
    - new_sup
    - val
- Sinon, on vérifie si le mot traité ne contient plus qu'un seul caractère.
  - Si oui, alors on stocke créer deux variables :
    - new_val : contient la future valeur du noeud.
    - new_eq : contient le nœud Eq de l'abre.
  - Sinon on sait que le mot n'est pas encore terminé et on crée de variables :
    - new_eq : contient le résultat d'un appelle récursif à Suppression sur l'arbre eq pour supprimer le reste du mot (reste(mot)).
    - new_val : la valeur courante du noeud.

  Nous effectuons ensuite un dernier test pour vérifier si plus aucun mot n'est contenu dans l'arbre courant.
  - Si oui, alors on retourne l'arbre vide.
  - Sinon on reconstruit un trie en appelant TrieH avec les arguments suivants :

- racine
- inf
- new_eq
- sup
- new_val

Fonction complexe

Idée pour réaliser un rééquilibrage

Une méthode consisterait à réaliser la chose suivante sur chacun des noeuds du trie : calculer la hauteur des sous arbres gauche et droit du Trie hybride courant, et si la différence de Hauteur dépasser un certain seuil, on effectue une rotation droite ou gauche (éventuellement une double rotation gauche droite ou droite gauche) tel que :

- Si le sous arbre gauche est plus grand alors on vérifie également le déséquilibre des sous arbres inf et sup du sous arbres.
    - Si l'enfant supérieur du sous arbre a une plus grande hauteur que l'enfant inférieur du sous arbre alors, on effectue d'abord une rotation gauche sur le sous arbre puis nous effectuons une rotation droite sur l'arbre.
    - Sinon, il est simplement nécessaire de réaliser une rotation droite sur l'arbre.
- Si le sous arbre droit est plus grand alors on vérifie également le déséquilibre des sous arbres inf et sup du sous arbres.
    - Si l'enfant inférieur du sous arbre a une plus grande hauteur que l'enfant supérieur du sous arbre alors, on effectue d'abord une rotation droite sur le sous arbre puis nous effectuons une rotation gauche sur l'arbre.
    - Sinon, il est simplement nécessaire de réaliser une rotation gauche sur l'arbre.

Une autre méthode consisterait à récupérer tous les mots du trie hybride et de les ranger dans un ordre optimal d'insertion.

Equilibrage(A) : Trie Hybride -> Trie Hybride

Cette fonction rééquilibre un trie.

Fonctionnement

On commence, on vérifie si le trie est vide et si c'est le cas alors, on retourne le trie tel quel.
Sinon on appelle récursivement la fonction Equilibrage sur chacun de ses enfants pour les mettre à jour.

Puis on évalue le déséquilibre de l'arbre courant et si la situation s'y prête on réalise une rotation gauche ou droite ou encore une double rotation gauche droite ou droite gauche.

On retourne ensuite le trie équilibré.

Ajout_Equilibrage(mot, A, v) : str * Trie Hybride * int -> Trie Hybride

Cette fonction ajoute un mot dans un trie puis rééquilibre ce trie.

Fonctionnement

On ajoute le mot dans le trie A et on stocke le résultat de cet ajout dans une nouvelle variable. Puis on lance immédiatement le rééquilibrage sur ce nouveau trie.
On retourne le trie rééquilibré.

Complexités théoriques des fonctions avancées

- Recherche :
  Mesure de complexité :
  Nombre de comparaisons.
  Pourquoi ?
  - C'est l'opération la plus fréquente dans la fonction.
  - Elle est inhérente au processus de recherche du mot dans le trie.
  - Le fonctionnement d'un trie hybride repose sur des comparaisons de caractère.

  Complexité :
  Dans le pire des cas, à chaque appelle à lookup (fonction de recherche utilisée dans Recherche(arbre, mot)) on comparera le premier caractère du mot courant au maximum un nombre constant de fois par visite de noeud.
  Soit N le nombre de mot dans le trie :
  - Trie équilibrée : la profondeur étant logarithmique on aura donc à faire au plus O(log(N)) comparaisons.
  - Trie déséquilibrée : Dans le pire des cas on aura à faire au pire O(N) comparaisons.

- ComptageMots, ListeMots, ComptageNil, Hauteur:
  Mesure de complexité : Nombre de nœuds visités.
  Pourquoi ?
  - Dans chacune des ces fonctions pour réaliser les opérations attendues, on se déplace dans les sous arbres de manière arbitraire en effectuant un nombre constant d'opérations à chaque nœud.

  Complexité :
  Soit N le nombre de nœuds dans le trie.
  Pour chacune de ces fonctions pour pouvoir effectuer les opérations attendues, on parcourt à chaque fois tous les nœuds de l'arbre une seule fois. Chacune des ces fonctions a donc une complexité en O(N).

- Prefixe :
  Mesure de complexité : Nombre de comparaisons
  Pourquoi ?
  - Car le traitement est ici divisé en deux parties. Tout d'abord, on recherche le nœud marquant la fin du préfixe passé à la fonction en comparant à chaque fois le premier caractère du préfixe courant au caractère stocké dans le nœud. Ensuite on recherche tous les mots de ce sous arbre sans effectuer aucune comparaison.

  Complexité :
  Soit N le nombre de nœuds dans le trie.
  La première étape fonctionne de manière analogue à la recherche et donc coûte : O(log(N)).

La seconde étape elle fonctionne de la même manière que ListeMots la seul différence et que ici, on récupère tous les mots depuis un sous arbre spécifique.
Soit M le nombre de noeuds contenus dans ce sous arbre, la complexité de cette étape serait en : O(M)
La complexité finale est donc en : O(M + Log(N)).

- Suppression :
  Mesure de complexité :
  Nombre de comparaisons du premier caractère du mot recherché.
  Pourquoi ?
  - Pour les mêmes raisons que Recherche.

  Complexité :
  Soit N le nombre de nœuds dans le trie.
  On commence par rechercher si le mot est présent dans le trie : O(log(N)).
  S'il ne l'est pas, l'exécution se termine.
  Si le mot existe, on recommence le parcours de l'arbre toujours de manière récursive, en se dirigeant en comparant le caractère courant avec celui contenu dans le nœud visité. Une fois le traitement de l'appelle récursif suivant terminé, on effectue à nouveau des comparaisons, cette fois-ci pour vérifier si le nœud peut être supprimé ou non.
  Au plus on effectu
  Question 3.7 : e donc un nombre constant de comparaison et effectuons deux fois la recherche du mot.
  La complexité est donc en : O(log(N))

Complexité théorique de la fonction complexe

- Ajout d'un mot suivi d'un rééquilibrage :
  Mesure de complexité :
  Nombre de nœuds visités.
  Pourquoi ?
  - Car le parcours des nœuds est l'opération la plus utilisée dans cette fonction. D'ailleurs Pour estimer si oui ou non l'arbre nécessite un rééquilibrage, il nous faut à chaque fois mesurer la hauteur de chacune de ses branches Inf et Sup afin de les comparer. Comme l'exploration de la hauteur s'évalue elle aussi en fonction du nombre de nœuds, nous utiliserons donc ici aussi cette mesure.

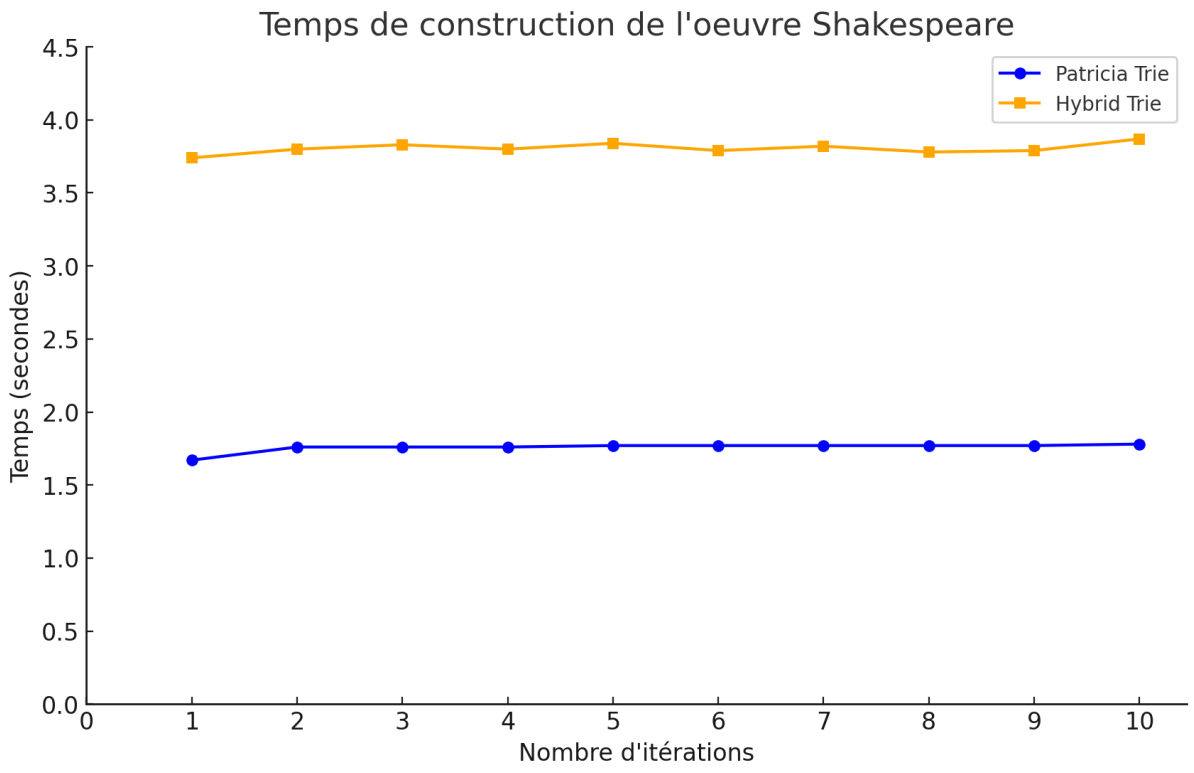
  Complexité :
  Soit N le nombre de nœuds du trie.
  On commence par insérer le mot dans le trie, cela nous coûte : O(log(N)).
  Puis on commence le rééquilibrage.
  À chaque niveau de l'arbre on essaye d'effectuer un rééquilibrage sur tous les nœuds. Et pour chaque noeud à chaque niveau, on calcule la hauteur de ses fils Inf et Sup ainsi que la hauteur des fils Inf et Sup de ses derniers.

On explore donc au moins une fois chaque nœud et plus le nœud est profond plus ce dernier est visité.
Ici, une feuille de l'arbre (le type de nœud le plus visité) ne sera donc pas visitée plus de deux fois la taille de sa branche.
Soit M la hauteur de l'arbre, on a donc une complexité en : O(M*N)

Expérimentation

Dans cette partie, si aucune précision n'est donnée concernant la version du trie hybride utilisée dans les mesures (équilibré ou non équilibré), cela signifie que la version non équilibrée a été utilisée.

3.1 Temps de construction de l'oeuvre Shakespeare:

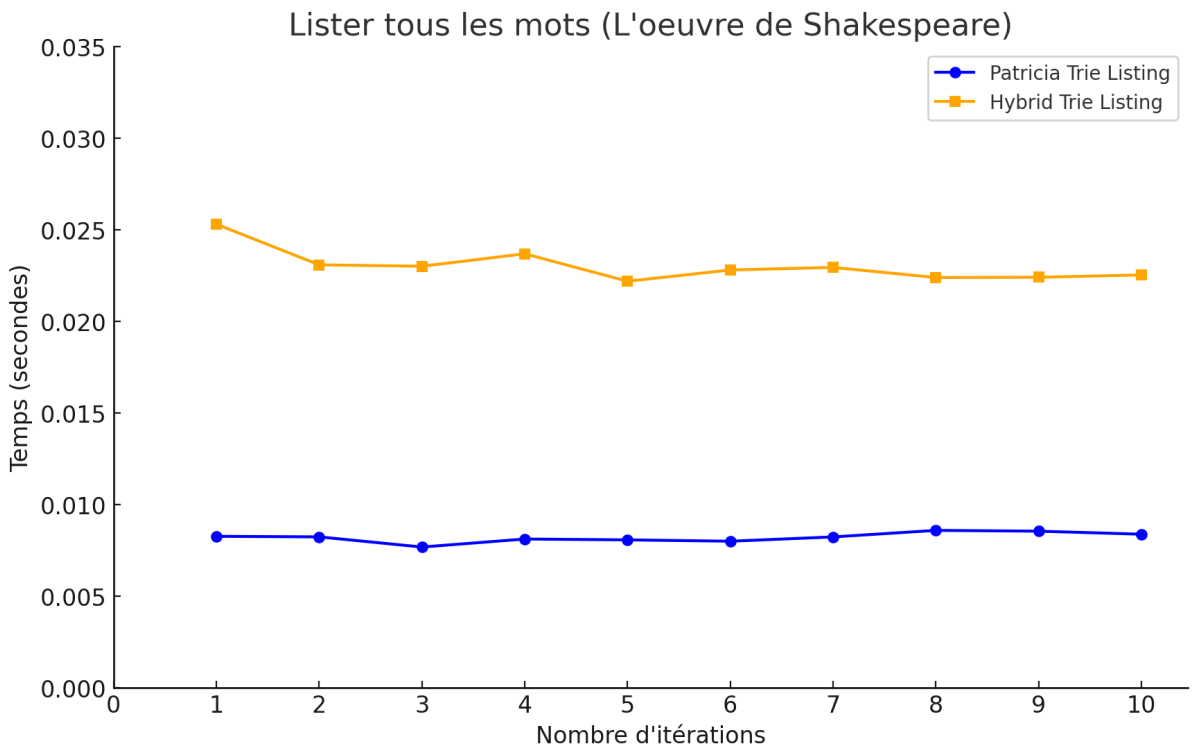


Le graphique présente une comparaison du **temps de construction** pour l'œuvre complète de **Shakespeare**, en utilisant deux structures de données différentes : **Patricia Trie** (en bleu) et **Hybrid Trie** (en orange). L'axe des abscisses représente le **nombre d'itérations** (de 1 à 10), tandis que l'axe des ordonnées indique le **temps en secondes** requis pour la construction.

Temps moyen par structure :

- **Patricia Trie :** Le temps de construction reste **constamment autour de 1.75 secondes**, avec une légère augmentation initiale avant de se stabiliser. Cette régularité montre une efficacité constante, même après plusieurs itérations.
- **Hybrid Trie :** Le temps moyen de construction est d'environ **3.8 à 4 secondes**, ce qui est nettement supérieur à celui du Patricia Trie. De plus, on observe de **légères variations** entre les itérations, sans tendance évidente à l'amélioration.

3.2 Lister tous les mots de l'oeuvre de Shakespeare :

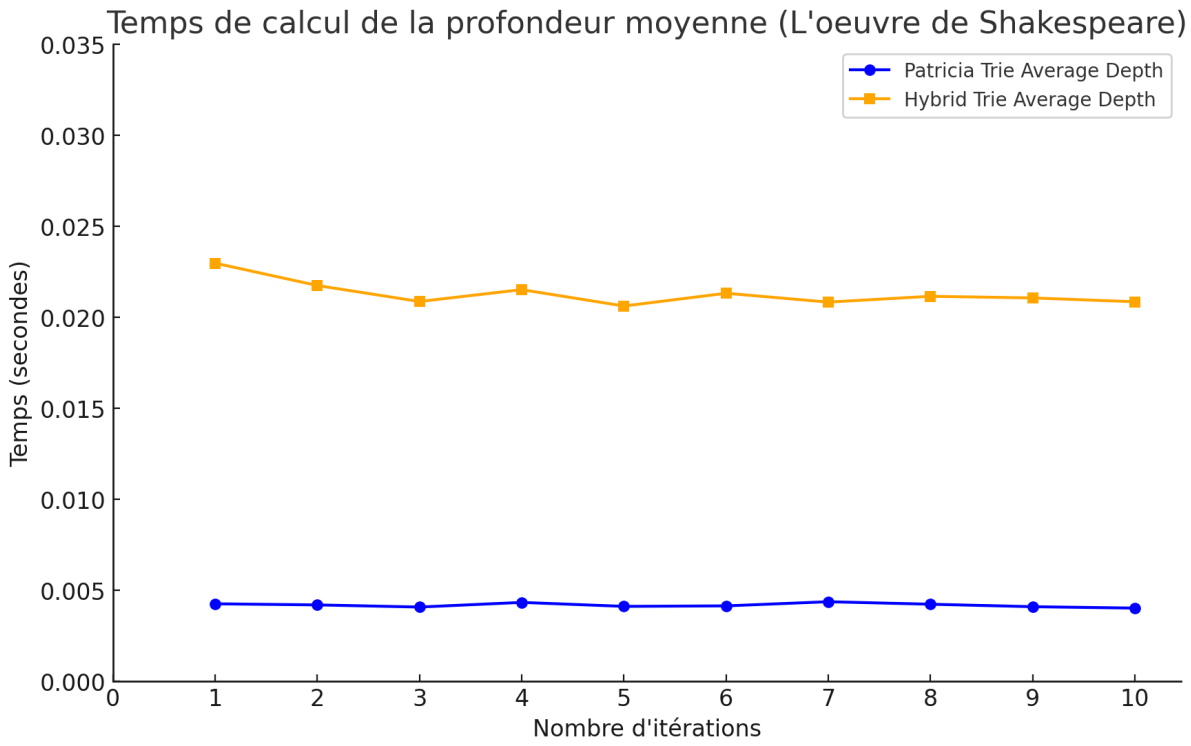


Le graphique compare le **temps nécessaire** pour lister tous les mots contenus dans l'œuvre complète de **Shakespeare**, en utilisant deux structures de données : **Patricia Trie** (en bleu) et **Hybrid Trie** (en orange).

Patricia Trie (en bleu) :
Le temps d'exécution reste très faible, autour de **0.008 secondes** en moyenne, avec une faible variation entre les itérations. Cela démontre une grande stabilité et une efficacité optimale pour lister les mots dans cette structure.

Hybrid Trie (en orange) :
Le temps moyen se situe autour de **0.022 secondes**, soit environ **2.5 fois plus lent** que le Patricia Trie. Bien qu'il reste rapide, le Hybrid Trie présente des variations légèrement plus marquées.

3.3 Temps de calcule de la profondeur moyenne de l'oeuvre de Shakespeare :



Ce graphique compare le **temps nécessaire pour calculer la profondeur moyenne** des deux structures de données, **Patricia Trie** (en bleu) et **Hybrid Trie** (en orange), appliquées à l'œuvre complète de **Shakespeare**.

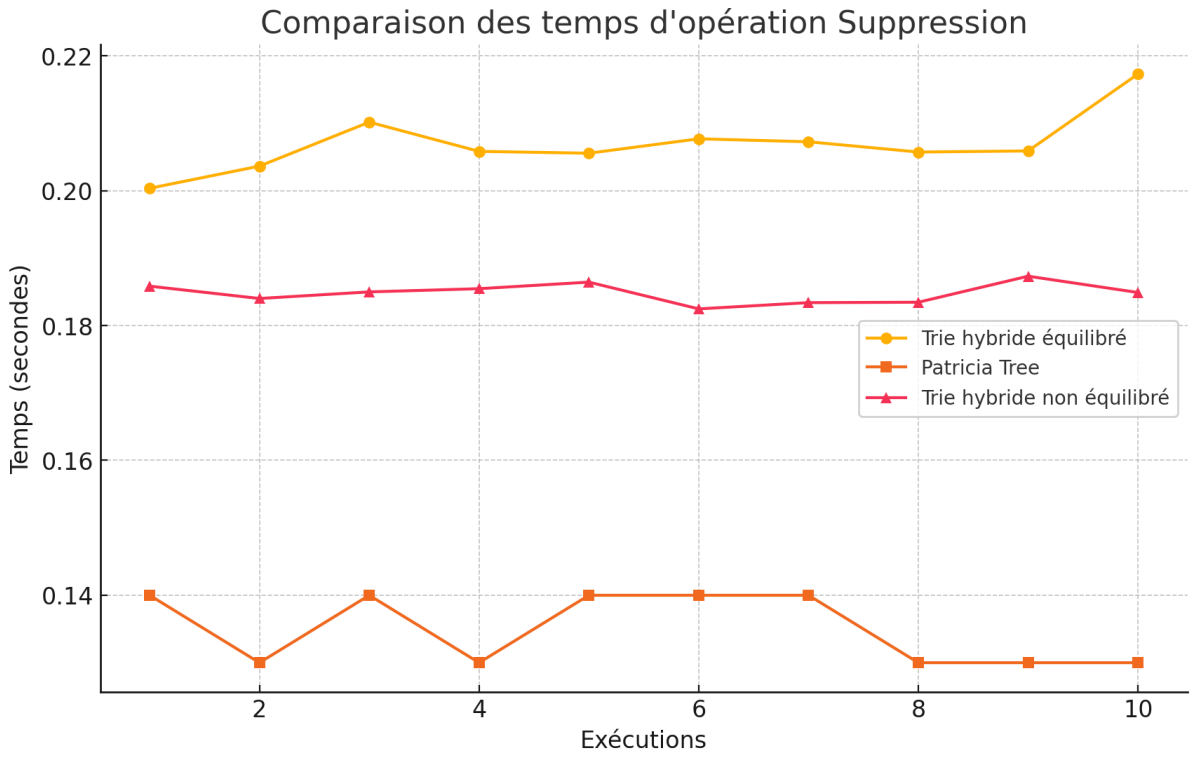
Patricia Trie (en bleu) :
Le temps de calcul reste remarquablement faible, autour de **0.004 secondes**, avec une courbe **quasi constante** sur toutes les itérations. Cela reflète l'efficacité de la structure compressée du Patricia Trie pour accéder aux données et parcourir les nœuds.

Hybrid Trie (en orange) :
Le temps de calcul est plus élevé, se situant autour de **0.021 secondes** en moyenne, soit environ **5 fois plus lent** que le Patricia Trie. Malgré des petites variations, le temps reste relativement stable au fil des itérations.

Différence notable :

- Le **Hybrid Trie** est plus coûteux en temps pour calculer la profondeur moyenne, en raison du **nombre de nœuds plus élevé** et de sa structure moins compressée. Cela nécessite un parcours plus long et plus complexe.

3.4 Temps de suppression :



Ce graphique compare le **temps nécessaire pour la suppression** d'éléments dans deux structures de données, **Patricia Trie** et **Hybrid Trie** (dans sa version équilibrée et non équilibrée), appliquées à la pièce **"1 Henry IV"** de Shakespeare.

Patricia Trie :

Le temps de suppression oscille autour de **0.13 à 0.14 secondes** avec de légères variations. Ce comportement indique une exécution relativement stable et une bonne efficacité.

Hybrid Trie :

Le temps de suppression est plus élevé, autour de **0.18 à 0.19 secondes**, montrant une constance mais avec des valeurs supérieures à celles du Patricia Trie.

Hybrid Trie équilibré :

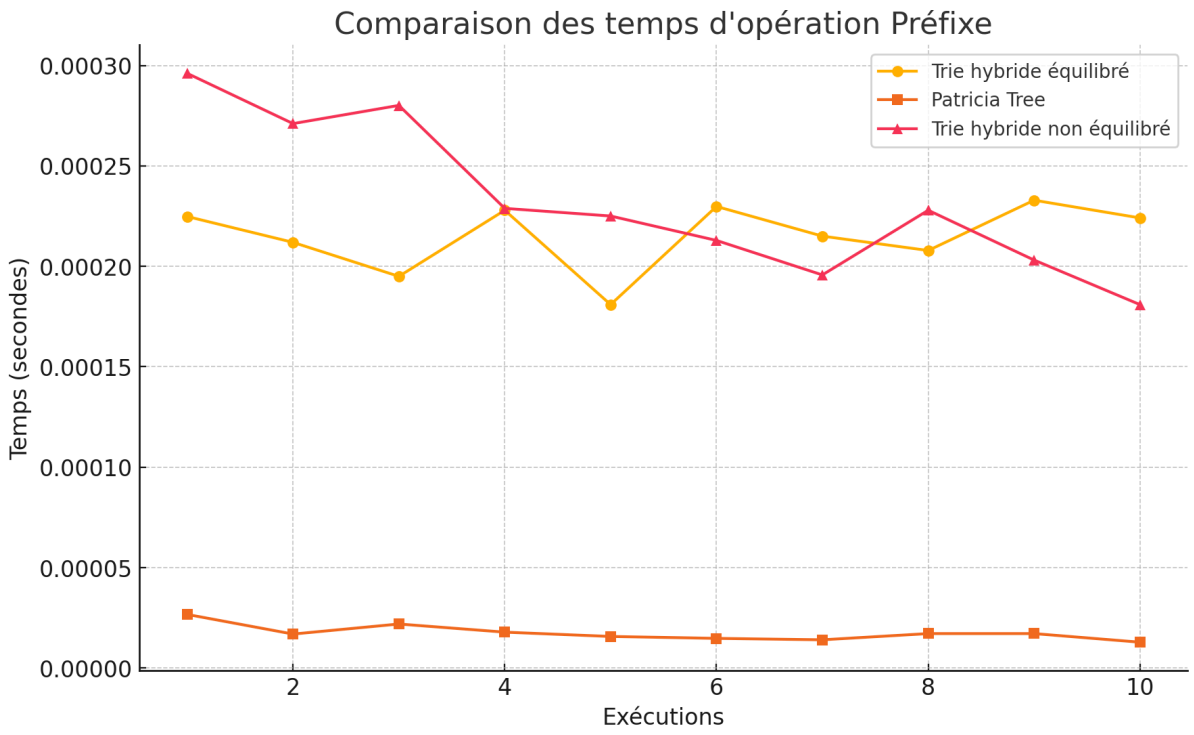
Le temps de suppression est étrangement encore plus élevé, oscillant entre 0.20 secondes à 0.22, cela indique peut-être un souci avec la méthode d'équilibrage du trie.

Efficacité du Patricia Trie :

- La compression des nœuds dans le Patricia Trie permet de traiter les suppressions de manière plus efficace, car les chemins sont plus courts et le nombre de nœuds intermédiaires est réduit.
- De plus, la vérification des préfixes et la suppression des branches vides sont optimisées grâce à cette compression.

Coût plus élevé pour le Hybrid Trie :

- Le temps plus long pour le Hybrid Trie indique que sa structure, bien que flexible, nécessite davantage de traversées pour localiser et supprimer les éléments.
- La gestion des nœuds non compressés introduit une surcharge qui se reflète dans le temps de traitement.



Ce graphique compare le **temps d'exécution** pour effectuer une **recherche de préfixe** dans deux structures de données : **Patricia Tree**, le **Trie Hybride** et sa version **équilibrée**.

Performance globale :

- **Patricia Tree** :
  Le temps d'exécution est **extrêmement faible** et stable, autour de **0.00003 secondes** (3×10^{-5} secondes) pour toutes les itérations. Cette stabilité confirme la rapidité et l'efficacité de la structure compressée.
- **Trie Hybride (ici non équilibré et équilibré)** :
  Le temps d'exécution est **nettement plus élevé**, oscillant entre **0.0003 et 0.0002 secondes** (3×10^{-4} à 2×10^{-4} secondes). On observe une **légère diminution** au fil des itérations, avec quelques fluctuations. Le temps d'exécution des deux trie hybride est relativement similaire.

Comparaison des performances :

- Le **Patricia Tree** est environ **10 fois plus rapide** que le Trie Hybride pour effectuer la recherche de préfixe.
- Cette différence s'explique par l'optimisation de la **compression des préfixes** dans le Patricia Tree, qui permet de réduire considérablement le nombre de nœuds à examiner.

Mesure expérimentales de complexité

Analyse pour le nombre de comparaison :

- **Insertion** :
  - **Patricia-Trie** : 3 657 116 comparaisons
  - **Trie Hybride** : 6 664 651 comparaisons
  - **Analyse** : Le Patricia-Trie nécessite significativement moins de comparaisons lors de l'insertion, grâce à sa structure compressée.
- **Suppression** :
  - **Patricia-Trie** : 100 550 comparaisons
  - **Trie Hybride** : 2 735 687 comparaisons
  - **Analyse** : La suppression est nettement plus efficace dans le Patricia-Trie. Cela peut s'expliquer par le fait qu'il y a moins de nœuds à parcourir en raison de la compression des données.

Analyse pour le nombre de visites :

- **Liste des mots** :
    - **Patricia-Trie** : 51 337 visites
    - **Trie Hybride** : 170 287 visites
    - **Analyse** : Lister les mots est plus rapide dans le Patricia-Trie, car sa structure réduit le nombre total de nœuds, rendant le parcours moins coûteux.
- **Profondeur moyenne** :
    - **Patricia-Trie** : 51 337 visites
    - **Trie Hybride** : 170 287 visites
    - **Analyse** : Le calcul de la profondeur moyenne dans le Trie Hybride demande plus de visites, ce qui est cohérent avec sa structure généralement plus profonde et moins optimisée.
- **Recherche de préfixes** :
    - **Patricia-Trie**: 64 visites
    - **Trie Hybride** : 178 visites
    - **Analyse** : La recherche de préfixes est beaucoup plus rapide dans le Patricia-Trie. La compression des nœuds permet de réduire significativement le nombre de comparaisons nécessaires.

Conclusion

Ce projet nous a permis de constater par nous-mêmes la différence d'efficacité marquante entre le Patricia-Trie et le Trie Hybride dans le cadre d'un dictionnaire.
Nous pouvons affirmer que les Patricia-Tries, grâce à leur structure compressée, sont plus performants que les Tries Hybrides pour les opérations implémentées ici (recherche, suppression, listage, etc.).
Cependant, les Tries Hybrides restent plus flexibles, permettant une restructuration des données stockées via des rotations, ce qui peut améliorer l'efficacité du trie dans le contexte de certaines opérations, comme la recherche.