Rapport de ALGAV
Devoir de Programmation : Tries

Mengxiao LI(21304592)
Xue YANG(28705179)

UE ALGAV
Master Informatique

Tuteur(s) université :
Antoine Genitrini

décembre 2024

Table des matières

1 Sujet du Projet

1.1 Objectif du Projet

L'objectif principal de ce projet est de concevoir et d'implémenter deux structures de données, *Patricia-trie* et *Hybrid-trie*, pour représenter un dictionnaire de mots construit sur l'alphabet ASCII. L'étude vise à comparer ces modèles en termes de performances et de complexité algorithmique.

1.2 Aspects à Étudier

- **Structures des Tries** : Étude des structures *Trie Hybride* et *Patricia Trie*, avec leur construction et organisation.
- **Fonctions des Tries** : Implémentation des opérations de base (*insertion*, *suppression*, *recherche*) et des fonctions avancées.
- **Analyse de la Complexité** : Évaluation théorique des complexités des opérations et comparaison des performances.
- **Analyse Expérimentale** : Tests pour mesurer les performances (temps d'exécution, hauteur, profondeur, etc.) des deux structures.

2 Structures et Fonctions des Tries

2.1 Structure 1 : Patricia-Tries

2.1.1 La structure

Chaque nœud est représenté par : Chaque nœud dans le Patricia-Trie est représenté par les éléments suivants :

- **label** : Chaîne de caractères représentant le *plus long préfixe commun*.
- **children** : Un *dictionnaire* qui stocke les sous-nœuds. Les **clés** du dictionnaire sont les premiers caractères des étiquettes (`label`) des sous-nœuds, et les **valeurs** sont les nœuds enfants correspondants.

En dehors des nœuds individuels, le Patricia-Trie utilise également un marqueur de fin global :

- **Marqueur de fin** : Le caractère spécial `chr(0x00)` est utilisé pour marquer la *fin d'un mot*, permettant de distinguer les préfixes des mots complets. Raison du choix : `chr(0x00)` (le caractère NULL en ASCII) a été choisi car :
  - Il est le premier caractère dans la table ASCII, ce qui garantit qu'il est toujours *inférieur* à tout autre caractère lors des comparaisons ;
  - Son utilisation minimise les conflits avec les autres caractères du texte, car il n'est généralement pas utilisé dans les données textuelles.

2.1.2 Les Fonctions Auxiliaires

Dans l'implémentation de l'arbre Patricia, nous avons utilisé plusieurs fonctions auxiliaires pour prendre en charge les opérations de base :

- **json_to_patricia_trie(data)** → *arbre* : Recrée un Patricia Trie à partir d'une structure JSON.
- **find_mots_prefix(str1, str2, counter_dict=None, counter_key=None)** → *chaîne* : Détermine le plus long préfixe commun entre deux chaînes de caractères. Elle peut également incrémenter un compteur pour mesurer le nombre de comparaisons effectuées.
- **to_dict(arbre)** → *dict* : Convertit le Patricia Trie en une structure de dictionnaire pour une visualisation ou une sauvegarde simplifiée.
- **display_as_json()** → *void* : Affiche la structure de l'arbre Patricia sous forme JSON pour faciliter la lecture et la compréhension.

2.1.3 Construction du Patricia Trie

Pour construire un *Patricia Trie*, nous utilisons la méthode `inserer` afin d'insérer les mots dans la structure. Afin de garantir l'unicité des mots, un marqueur spécial `end_marker` est ajouté à la fin de chaque mot.

Algorithm 1 Construction d'un Patricia Trie (`inserer`)

```
1: function INSERER(arbre, mot)
2:   mot ← mot + end_marker          // Ajout du marqueur de fin pour garantir l'unicité
3:   current_node ← arbre.root
4:   while mot ≠ vide do
5:     if mot[0] ∈ current_node.children then
6:       child ← current_node.children[mot[0]]
7:       prefix ← FIND_MOTS_PREFIX(mot, child.label)
8:       if prefix = child.label then
9:         current_node ← child
10:        mot ← mot[len(prefix):]
11:      else                        // Partie correspondance partielle : divsier les nœuds
12:        new_node ← PatriciaTrieNode(prefix)
13:        rest ← child.label[len(prefix):]
14:        new_node.children[rest[0]] ← child
15:        child.label ← rest
16:        current_node.children[prefix[0]] ← new_node
17:        current_node← new_node
18:        mot ← mot[len(prefix):]
19:      end if
20:    else
21:      current_node.children[mot[0]] ← PatriciaTrieNode(mot)
22:      return
23:    end if
24:  end while
25: end function
```

2.1.4 Fonctions Avancées

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

Algorithm 2 Recherche d'un mot dans un Patricia Trie

```
1: function RECHERCHE(arbre, mot)
2:   mot ← mot + end_marker                    // Ajout du marqueur de fin
3:   current_node ← arbre.root
4:   while mot ≠ vide do
5:     if mot[0] ∉ current_node.children then
6:       return False                          // Pas de correspondance, mot absent
7:     end if
8:     child ← current_node.children[mot[0]]
9:     prefix ← FIND_MOTS_PREFIX(mot, child.label)
10:    if prefix ≠ child.label then
11:      return False                          // Partie correspondance partielle, échec
12:    else
13:      current_node ← child
14:      mot ← mot[len(prefix):]
15:    end if
16:  end whilereturn True                     // Mot trouvé avec succès
17: end function
```

Complexité :
- **Mesure de complexité :** Nombre de comparaisons de caractères.
- **Complexité :** $O(k)$, où k est la longueur du mot recherché.
- **Justification :** La recherche parcourt chaque caractère du mot en suivant le chemin dans l'arbre. À chaque étape, il y a une comparaison pour trouver le préfixe commun entre le mot et le label du nœud courant.

2. ComptageMots(arbre) → entier : une fonction qui compte les mots présents dans le dictionnaire.

Algorithm 3 Comptage du nombre de mots dans un Patricia Trie

```
1: function ComptageMots(arbre)
2:     return ComptageRec(arbre.root)                    // Appel à la fonction récursive
3: end function
4: function ComptageRec(node)
5:     count ← 0
6:     if node.label se termine par end_marker then
7:         count ← count + 1
8:     end if
9:     for child dans node.children do
10:        count ← count + ComptageRec(child)
11:    end for
12:    return count
13: end function
```

Complexité :
- **Mesure de complexité :** Nombre d'accès aux nœuds.
- **Complexité :** $O(N)$, où N est le nombre de nœuds dans le Patricia Trie.
- **Justification :** Pour compter les mots, il est nécessaire de parcourir l'ensemble des nœuds de l'arbre. Chaque nœud est visité une seule fois, ce qui donne une complexité linéaire $O(N)$.

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

Algorithm 4 Liste des mots dans un Patricia Trie (ordre alphabétique)

```
1: function ListeMots(arbre)
2:     mots ← []                                  // Initialisation d'une liste vide pour stocker les mots
3:     Aux(arbre.root, "", mots)
4:     return mots
5: end function
6: function Aux(node, mot_courant, mots)
7:     if node.label contient end_marker then
8:         mots.append(mot_courant + node.label sans end_marker) // Ajout du mot complet à la liste
9:     end if
10:    for chaque enfant dans node.children triés par clé do
11:        Aux(enfant, mot_courant + node.label, mots)
12:    end for
13: end function
```

Complexité :
- **Mesure de complexité :** Génération de tous les caractères des mots.
- **Complexité :** $O(M)$, où M est la somme des caractères de tous les mots stockés dans le Patricia Trie.

— **Justification :** Pour lister tous les mots, la fonction parcourt chaque nœud et reconstruit les mots en concaténant les étiquettes. L'opération nécessite de visiter chaque caractère exactement une fois, ce qui donne une complexité proportionnelle à M.

4. ComptageNil(arbre) → entier : une fonction qui compte les pointeurs vers Nil dans le Patricia Trie.

Algorithm 5 Comptage des pointeurs vers Nil dans un Patricia Trie (version simplifiée)

```
1: function ComptageNil(arbre)
2:     return CountNil(arbre.root)
3: end function
4: function CountNil(node)
5:     if node.children est vide then return 1
6:     end if
7:     return somme(CountNil(enfant) pour chaque enfant dans node.children)
8: end function
```

Variante : ComptageNilExcludeEndmarker(arbre)
Cette variante exclut explicitement les pointeurs Nil associés au marqueur de fin.

Algorithm 6 Comptage des pointeurs vers Nil en excluant le marqueur de fin (version simplifiée)

```
1: function ComptageNilExcludeEndmarker(arbre)
2:     return CountNilExclude(arbre.root)
3: end function
4: function CountNilExclude(node)
5:     if node.children est vide then return 1
6:     end if
7:     return somme(CountNilExclude(enfant) pour chaque (clé, enfant) dans
       node.children si clé ≠ end_marker)
8: end function
```

Complexité :

— **Mesure de complexité :** Nombre d'accès aux nœuds.
— **Complexité :** $O(N)$, où N est le nombre total de nœuds dans le Patricia Trie.
— **Justification :**
  — Les fonctions parcourt tous les nœuds pour vérifier s'ils sont des pointeurs vers Nil.

5. Hauteur(arbre) → entier : une fonction qui calcule la hauteur du Patricia Trie.

Algorithm 7 Calcul de la hauteur d'un Patricia Trie

```
1: function Hauteur(arbre)
2:     return CalculerHauteur(arbre.root)
3: end function
4: function CalculerHauteur(node)
5:     if node.children est vide then return 0
6:     end if
7:     return 1 + max(CalculerHauteur(enfant) pour chaque enfant dans node.children)
8: end function
```

Complexité :

— **Mesure de complexité :** Nombre d'accès aux nœuds.
— **Complexité :** $O(N)$, où N est le nombre total de nœuds dans le Patricia Trie.
— **Justification :** Pour calculer la hauteur, la fonction doit visiter chaque nœud pour trouver la profondeur maximale, ce qui nécessite de parcourir toute la structure. Cela implique un accès à N nœuds dans le pire des cas.

6. ProfondeurMoyenne(arbre) → entier : une fonction qui calcule la profondeur moyenne des feuilles d'un Patricia Trie.

Algorithm 8 Calcul de la profondeur moyenne des feuilles d'un Patricia Trie

```
1: function ProfondeurMoyenne(arbre)
2:     profondeur, feuilles ← CalculerProfondeur(arbre.root, 0)
3:     if feuilles = 0 then
4:         return 0                          // Prévenir une division par zéro si l'arbre est vide
5:     end if
6:     return profondeur / feuilles
7: end function
8: function CalculerProfondeur(node, depth)
9:     if node.children est vide then
10:        return depth, 1
11:    end if
12:    total_profondeur ← 0
13:    total_feuilles ← 0
14:    for chaque enfant dans node.children do
15:        profondeur, feuilles ← CalculerProfondeur(enfant, depth + 1)
16:        total_profondeur ← total_profondeur + profondeur
17:        total_feuilles ← total_feuilles + feuilles
18:    end for
19:    return total_profondeur, total_feuilles
20: end function
```

Complexité :

- **Mesure de complexité :** Nombre d'accès aux nœuds.
- **Complexité :** $O(N)$, où N est le nombre total de nœuds dans le Patricia Trie.
- **Justification :** La fonction doit visiter tous les nœuds pour accumuler les profondeurs et compter les feuilles, nécessitant un parcours complet de l'arbre. Dans le pire des cas, cela implique N accès.

7. Prefixe(arbre, mot) → entier : une fonction qui compte le nombre de mots dans le Patricia Trie commençant par un préfixe donné.

Algorithm 9 Recherche du nombre de mots commençant par un préfixe

```
1: function Prefixe(arbre, mot)
2:     current_node ← arbre.root
3:     while mot ≠ vide do
4:         if mot[0] ∉ current_node.children then
5:             return 0
6:         end if
7:         child ← current_node.children[mot[0]]
8:         prefix ← FIND_MOTS_PREFIX(mot, child.label)
9:         if len(prefix) < len(child.label) then
10:            if len(prefix) = len(mot) then
11:                return COMPTAGE_MOTS(child)
12:            else
13:                return 0
14:            end if
15:        end if
16:        mot ← mot[len(prefix):]
17:        current_node ← child
18:    end while
19:    return COMPTAGE_MOTS(current_node)
20: end function
```

Complexité :

- **Mesure de complexité :** Comparaisons de caractères et accès aux nœuds.
- **Complexité :** $O(k + N_{\text{subtree}})$, où k est la longueur du préfixe et N_{subtree} est le nombre de nœuds dans la sous-arbre correspondant au préfixe.
- **Justification :** La fonction localise d'abord le nœud correspondant au préfixe en $O(k)$, puis compte les mots dans la sous-arbre en $O(N_{\text{subtree}})$.

8. Suppression(arbre, mot) : une fonction qui supprime un mot du Patricia Trie.

Algorithm 10 Suppression d'un mot dans un Patricia Trie (version simplifiée)

```
1: function SUPPRESSION(arbre, mot)
2:     mot ← mot + end_marker
3:     arbre.root ← DELETEREC(arbre.root, mot)
4: end function
5: function DELETEREC(node, mot)
6:     if mot = vide then
7:         if end_marker ∈ node.children then
8:             Retirer end_marker de node.children
9:         end if
10:        if node.children est vide then
11:            return None
12:        end if
13:        return node
14:    end if
15:    if mot[0] ∉ node.children then
16:        return node
17:    end if
18:    child ← node.children[mot[0]]
19:    prefix ← FIND_MOTS_PREFIX(child.label, mot)
20:    if len(prefix) < len(mot) ou prefix ≠ child.label then
21:        return node
22:    else
23:        node.children[mot[0]] ← DELETEREC(child, mot[len(prefix):])
24:    end if
25:    if len(node.children) = 1 et node.label n'a pas de end_marker then
26:        Fusionner avec le seul enfant
27:    end if
28:    return node
29: end function
```

Complexité :

- **Mesure de complexité :** Comparaisons de caractères.
- **Complexité :** $O(k)$, où k est la longueur du mot à supprimer.
- **Justification :** La suppression implique une recherche similaire à celle de la fonction `Recherche`, suivie par un ajustement des nœuds le long du chemin. Ces opérations se limitent au chemin lié au mot, ce qui donne une complexité de $O(k)$.

9. Fusion(a, b) → arbre : une fonction qui fusionne deux Patricia Tries en un seul.

Algorithm 11 Fusion de deux Patricia Tries

```
1: function FUSION(a, b)
2:     AUX(a.root, b.root)
3: end function
4: function AUX(node_a, node_b)
5:     for key_b, child_b dans node_b.children.items() do
6:         if key_b ∈ node_a.children then
7:             child_a ← node_a.children[key_b]
8:             prefix ← FIND_MOTS_PREFIX(child_a.label, child_b.label)
9:             if prefix = child_a.label et prefix = child_b.label then
10:                AUX(child_a, child_b)      // Les deux sous-arbres sont fusionnés récursivement
11:            else if prefix ≠ vide then
12:                rest_a ← child_a.label[len(prefix):]
13:                rest_b ← child_b.label[len(prefix):]
14:                new_a ← PatriciaTrieNode(rest_a)
15:                new_b ← PatriciaTrieNode(rest_b)
16:                new_a.children ← child_a.children
17:                new_b.children ← child_b.children
18:                node_a.children[key_b] ← PatriciaTrieNode(prefix)
19:                node_a.children[key_b].children[key_b] ← AUX(new_a, new_b)
20:            end if
21:        else
22:            node_a.children[key_b] ← child_b
23:        end if
24:    end for
25:    return node_a
26: end function
```

Complexité :

- **Mesure de complexité :** Nombre de nœuds visités.
- **Complexité :** $O(N_a + N_b)$, où N_a et N_b sont respectivement les nombres de nœuds dans les arbres Patricia a et b.
- **Justification :** La fusion compare et insère chaque nœud de l'arbre b dans l'arbre a. Dans le pire des cas, tous les nœuds des deux arbres doivent être parcourus.

2.2 Structure 2 : Tries Hybrides

2.2.1 Représentation et Concept

Le Trie Hybride est une structure de données combinant les fonctionnalités des tries classiques et des arbres binaires. Chaque nœud possède trois pointeurs :

- **gauche** : Pointeur vers un nœud avec un caractère inférieur.
- **milieu** : Pointeur vers un nœud pour le prochain caractère du mot.
- **droite** : Pointeur vers un nœud avec un caractère supérieur.

2.2.2 Primitives de Base

Les primitives de base du Trie Hybride incluent les opérations essentielles suivantes :

- **Insertion**(*arbre*, *mot*) → *arbre* : Ajoute un mot dans le Trie.
- **IsEmpty**(*arbre*) → *booléen* : Vérifie si le Trie est vide.
- **to_json**(*arbre*, *chemin*) → *void* : Sauvegarde le Trie dans un fichier JSON.
- **to_dict**(*arbre*) → *dict* : Convertit le Trie en un dictionnaire.

— **from_dict**(*data*) → *arbre* : Reconstruit le Trie à partir d'un dictionnaire.
— **from_json**(*chemin*) → *arbre* : Charge un Trie depuis un fichier JSON.

2.2.3 Construction du Trie Hybride

Le Trie Hybride a été construit par insertion successive des mots issus de l'*exemple de base* :

"A quel génial professeur de dactylographie sommes-nous redevables de la superbe phrase ci-dessous, un modèle du genre, que toute dactylo connaît par cœur puisque elle fait appel à chacune des touches du clavier de la machine à écrire ?"

Voici un pseudocode pour l'insertion d'un mot dans le Trie Hybride :

Algorithm 12 Insertion(*arbre*, *mot*) → *arbre*

```
1: function Insertion(arbre, mot)
2:     arbre.root ← InsertRec(arbre.root, mot, 0)
3:     return arbre
4: end function
5: function InsertRec(node, mot, index)
6:     char ← mot[index]
7:     if node = null then
8:         node ← HybridTrieNode(char)
9:     end if
10:    if char < node.char then
11:        node.left ← InsertRec(node.left, mot, index)
12:    else if char > node.char then
13:        node.right ← InsertRec(node.right, mot, index)
14:    else
15:        if index + 1 = length(mot) then
16:            node.is_end_of_word ← true
17:        else
18:            node.middle ← InsertRec(node.middle, mot, index + 1)
19:        end if
20:    end if
21:    return node
22: end function
```

Exemple de base Représenté en JSON :

1. Les mots ont été extraits de la phrase, convertis en minuscules et insérés dans le Trie Hybride.
2. L'arbre résultant a été sauvegardé sous format JSON dans le fichier `result/exemple_base.json`.

Chaque nœud du JSON généré est décrit par :
— **le caractère qu'il contient** (`char`),
— **un marqueur indiquant s'il s'agit de la fin d'un mot** (`is_end_of_word`),
— **trois références vers ses sous-arbres gauche, milieu et droit** (`left`, `middle`, `right`).

Voici un extrait du fichier JSON généré :

```
{
    "char": "a",
    "is_end_of_word": true,
    "left": null,
```

```
"middle": {
    "char": "p",
    "is_end_of_word": false,
    "left": null,
    "middle": {
    ...
```

2.2.4 Fonctions Avancées

Les fonctions avancées offrent des outils supplémentaires pour analyser et manipuler le Trie Hybride :

1. `Recherche(arbre, mot) → booléen` : Vérifie si un mot donné existe dans le Trie. Renvoie `True` si le mot est trouvé, sinon `False`.

Algorithm 13 Recherche(arbre, mot) → booléen

```
1: function RECHERCHE(arbre, mot)
2:     return RECHERCHEREC(arbre.root, mot, 0)
3: end function
4: function RECHERCHEREC(node, mot, index)
5:     if node = null then return False
6:     end if
7:     char ← mot[index]
8:     if char < node.char then return RECHERCHEREC(node.left, mot, index)
9:     else if char > node.char then return RECHERCHEREC(node.right, mot, index)
10:    else
11:        if index + 1 = length(mot) then return node.is_end_of_word
12:        else
13:            return RECHERCHEREC(node.middle, mot, index + 1)
14:        end if
15:    end if
16: end function
```

2. `ComptageMots(arbre) → entier` : Compte et retourne le nombre total de mots stockés dans le Trie.

Algorithm 14 ComptageMots(arbre) → entier

```
1: function COMPTAGEMOTS(node)
2:     if node = null then return 0
3:     end if
4:     count ← 0
5:     if node.is_end_of_word = true then
6:         count ← count + 1
7:     end if
8:     count ← count+ COMPTAGEMOTS(node.left)
9:     count ← count+ COMPTAGEMOTS(node.middle)
10:    count ← count+ COMPTAGEMOTS(node.right)
11:    return count
12: end function
```

3. `ListeMots(arbre) → liste[mots]` : Génère une liste alphabétique de tous les mots enregistrés dans le Trie.

Algorithm 15 ListeMots(arbre) → liste[mots]

```
1: function LISTE_MOTS(self)
2:     result ← []                                  // Initialiser une liste vide pour stocker les mots
3:     _LISTE_MOTS(self.root, "", result)           // Appeler la fonction auxiliaire avec le racine
4:     return result
5: end function
6: function _LISTE_MOTS(self, node, prefix, result)
7:     if node = null then
8:         return
9:     end if
10:    _LISTE_MOTS(self, node.left, prefix, result)          // Explorer le sous-arbre gauche
11:    if node.is_end_of_word = true then
12:        APPEND(result, prefix + node.char)                // Ajouter le mot formé à la liste
13:    end if
14:    _LISTE_MOTS(self, node.middle, prefix + node.char, result)   // milieu
15:    _LISTE_MOTS(self, node.right, prefix, result)                // droit
16: end function
```

4. `ComptageNil(arbre) → entier` : Compte le nombre de pointeurs null dans la structure du Trie.

Algorithm 16 Comptage des pointeurs NULL dans un Trie Hybride

```
1: function COMPTAGE_NIL(self)
2:     function _COMPTAGE_NIL(node)
3:         if node = null then return 0
4:         end if
5:         count ← 0
6:         if node.left = null then count ← count + 1
7:         end if
8:         if node.middle = null then count ← count + 1
9:         end if
10:        if node.right = null then count ← count + 1
11:        end if
12:        count ← count+ _COMPTAGE_NIL(node.left)
13:        count ← count+ _COMPTAGE_NIL(node.middle)
14:        count ← count+ _COMPTAGE_NIL(node.right)
15:        return count
16:    end function
17:    return _COMPTAGE_NIL(self.root)
18: end function
```

5. `Hauteur(arbre) → entier` : Retourne la hauteur maximale de l'arbre.

Algorithm 17 Calcul de la hauteur d'un Trie Hybride

```
1: function HAUTEUR(self)
2:     return _HAUTEUR(self.root)
3: end function
4: function _HAUTEUR(node)
5:     if node = null then return 0
6:     end if
7:     return 1 + max(_HAUTEUR(node.left), _HAUTEUR(node.middle), _HAUTEUR(node.right))
8: end function
```

6. `ProfondeurMoyenne(arbre) → entier` : Calcule la profondeur moyenne du Trie.

Algorithm 18 Calcul de la profondeur moyenne des feuilles d'un Trie Hybride

```
1: function PROFONDEUR_MOYENNE(self)
2:     result ← {total_depth : 0, leaf_count : 0}
3:     _PROFONDEUR_MOYENNE(self.root, 0, result)
4:     if result.leaf_count = 0 then return 0
5:     end if
6:     return result.total_depth/result.leaf_count
7: end function
8: function _PROFONDEUR_MOYENNE(node, depth, result)
9:     if node = null then return
10:    end if
11:    if node.is_end_of_word = true then
12:        result.total_depth ← result.total_depth + depth
13:        result.leaf_count ← result.leaf_count + 1
14:    end if
15:    _PROFONDEUR_MOYENNE(node.left, depth + 1, result)
16:    _PROFONDEUR_MOYENNE(node.middle, depth + 1, result)
17:    _PROFONDEUR_MOYENNE(node.right, depth + 1, result)
18: end function
```

7. `Prefixe(arbre, mot) → entier` : Compte combien de mots dans le Trie commencent par un préfixe donné.

Algorithm 19 Calculer le nombre de mots commençant par un préfixe donné

```
1: function PREFIXE(self, prefix) return _PREFIXE(self.root, prefix, 0)
2: end function
3: function _PREFIXE(node, prefix, index)
4:     if node = null then return 0
5:     end if
6:     if index ≥ len(prefix) then return COMPTAGEMOTS(node)
7:     end if
8:     char ← prefix[index]
9:     if char < node.char then return _PREFIXE(node.left, prefix, index)
10:    else if char > node.char then return _PREFIXE(node.right, prefix, index)
11:    else
12:        if index + 1 = len(prefix) then return COMPTAGEMOTS(node.middle)
13:        else
14:            return _PREFIXE(node.middle, prefix, index + 1)
15:        end if
16:    end if
17: end function
```

8. `Suppression(arbre, mot) → arbre` : Supprime un mot du Trie en nettoyant les nœuds inutilisés pour optimiser la structure.

Algorithm 20 Suppression d'un mot dans un Trie Hybride

```
1: function SUPPRESSION(self, word)
2:   self.root ← _SUPPRESSION(self.root, word, 0)
3:   if self.root ≠ null and self.root.is_end_of_word = false and all children of self.root
   = null then
4:     self.root ← null                          // Nettoyage si la racine devient inutile
5:   end if
6: end function
7: function _SUPPRESSION(node, word, index)
8:   if node = null then
9:     return null                                // Retourner null si le nœud est vide
10:  end if
11:  char ← word[index]
12:  if char < node.char then node.left ← _SUPPRESSION(node.left, word, index)
13:  else if char > node.char then node.right ← _SUPPRESSION(node.right, word, index)
14:  else
15:    if index + 1 = len(word) then node.is_end_of_word ← false    // Supprimer la fin
   du mot
16:    else
17:      node.middle ← _SUPPRESSION(node.middle, word, index + 1)
18:    end if
19:    if node.is_end_of_word = false and node.left = null and node.middle = null and
   node.right = null then return null                // Supprimer le nœud inutile
20:    end if
21:  end if
22:  return node
23: end function
```

2.2.5 Fonctions complexes

Pour améliorer l'efficacité et l'équilibre du Trie hybride, plusieurs fonctions ont été implémentées :

- **is_unbalanced :** Détecte le déséquilibre de l'arbre à l'aide de deux seuils : *depth_threshold* (différence de profondeur maximale) et *balance_threshold* (rapport entre la profondeur maximale et la profondeur moyenne).
- **rebalance :** Rééquilibre l'arbre en extrayant les mots, en les triant et en les reconstruisant via une approche *par division*.
- **_build_balanced_tree :** Construit un arbre équilibré en sélectionnant l'élément central comme nœud racine et en divisant récursivement les sous-arbres gauche et droit.
- **_build_tree_from_word :** Crée une structure chaînée en insérant chaque caractère d'un mot séquentiellement et en marquant la fin du mot.
- **insert_with_balance :** Combine l'insertion conditionnelle avec la détection et la correction du déséquilibre. Lorsqu'un déséquilibre est détecté, la fonction *rebalance* est appelée automatiquement pour restaurer l'équilibre.

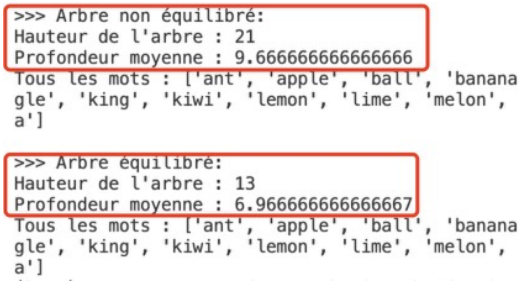

FIGURE 1 – Réduction de la profondeur moyenne après rééquilibrage.

2.2.6 Analyse des Complexités

Dans cette section, nous analysons les complexités temporelles et spatiales de chaque fonction implémentée pour le Trie Hybride.

Recherche(arbre, mot) → booléen
- **Complexité temporelle** : $O(h)$, où h est la hauteur de l'arbre. Chaque niveau de l'arbre est parcouru en fonction des caractères du mot.
- **Complexité spatiale** : $O(1)$, aucun espace auxiliaire n'est requis.

ComptageMots(arbre) → entier
- **Complexité temporelle** : $O(n)$, où n est le nombre total de nœuds dans le Trie. Chaque nœud est visité une fois.
- **Complexité spatiale** : $O(h)$, à cause de la pile d'appel récursive dont la profondeur maximale est h, la hauteur de l'arbre.

ListeMots(arbre) → liste[mots]
- **Complexité temporelle** : $O(n)$. Un parcours complet de l'arbre est effectué pour récupérer tous les mots.
- **Complexité spatiale** : $O(n)$. La liste de résultat occupe un espace proportionnel au nombre total de mots, en plus de la pile d'appel récursive ($O(h)$).

ComptageNil(arbre) → entier
- **Complexité temporelle** : $O(n)$, car tous les nœuds de l'arbre sont visités pour vérifier leurs trois pointeurs (`left`, `middle`, `right`).
- **Complexité spatiale** : $O(h)$, en raison de la pile d'appel pour la récursion.

Hauteur(arbre) → entier
- **Complexité temporelle** : $O(n)$. Chaque nœud doit être visité pour déterminer la profondeur maximale.
- **Complexité spatiale** : $O(h)$. La récursion consomme un espace proportionnel à la hauteur de l'arbre.

ProfondeurMoyenne(arbre) → entier
- **Complexité temporelle** : $O(n)$, tous les nœuds sont parcourus pour calculer les profondeurs des feuilles.
- **Complexité spatiale** : $O(h)$, dû à la profondeur maximale de la pile d'appel.

Prefixe(arbre, mot) → entier

— **Complexité temporelle** : $O(p+k)$, où p est la longueur du préfixe et k est le nombre de nœuds dans le sous-arbre du dernier caractère du préfixe.

— **Complexité spatiale** : $O(h)$, car la pile d'appel atteint une profondeur maximale de h.

Suppression(arbre, mot) → arbre

— **Complexité temporelle** : $O(h)$, car l'algorithme suit le chemin correspondant au mot à supprimer.

— **Complexité spatiale** : $O(h)$, la pile d'appel récursive nécessite un espace proportionnel à la hauteur de l'arbre.

Résumé des Complexités Nous résumons les résultats des analyses précédentes dans le tableau ci-dessous :

Fonction	Complexité temporelle	Complexité spatiale
Recherche	$O(h)$	$O(1)$
ComptageMots	$O(n)$	$O(h)$
ListeMots	$O(n)$	$O(n)$
ComptageNil	$O(n)$	$O(h)$
Hauteur	$O(n)$	$O(h)$
ProfondeurMoyenne	$O(n)$	$O(h)$
Prefixe	$O(p+k)$	$O(h)$
Suppression	$O(h)$	$O(h)$

Table 1 – Résumé des Complexités des Algorithmes

3 Étude expérimentale

Pour comparer les performances des deux structures de données (*Hybrid Trie* et *Patricia Trie*), nous avons effectué plusieurs tests expérimentaux. Les résultats obtenus sont présentés et analysés ci-dessous.

3.1 Comparaison des insertions et du temps de construction

La construction des deux structures repose sur la méthode d'insertion des mots. À cette fin, nous avons comparé le temps nécessaire à l'insertion et le nombre de comparaisons effectuées. Les résultats expérimentaux montrent que le **Hybrid Trie** nécessite un temps de construction plus court, tandis que dans la plupart des cas, le **Patricia Trie** effectue un nombre de comparaisons inférieur.

Ce phénomène peut s'expliquer par le fait que, pour le **Patricia Trie**, en raison du grand nombre de mots, le processus d'insertion exige de nombreuses opérations de comparaison de préfixes et de division des nœuds, augmentant ainsi la complexité et le coût en temps de l'insertion. En revanche, le **Hybrid Trie** utilise une stratégie d'insertion plus directe. Bien qu'il nécessite un plus grand nombre de comparaisons, son processus de construction s'avère globalement plus rapide.

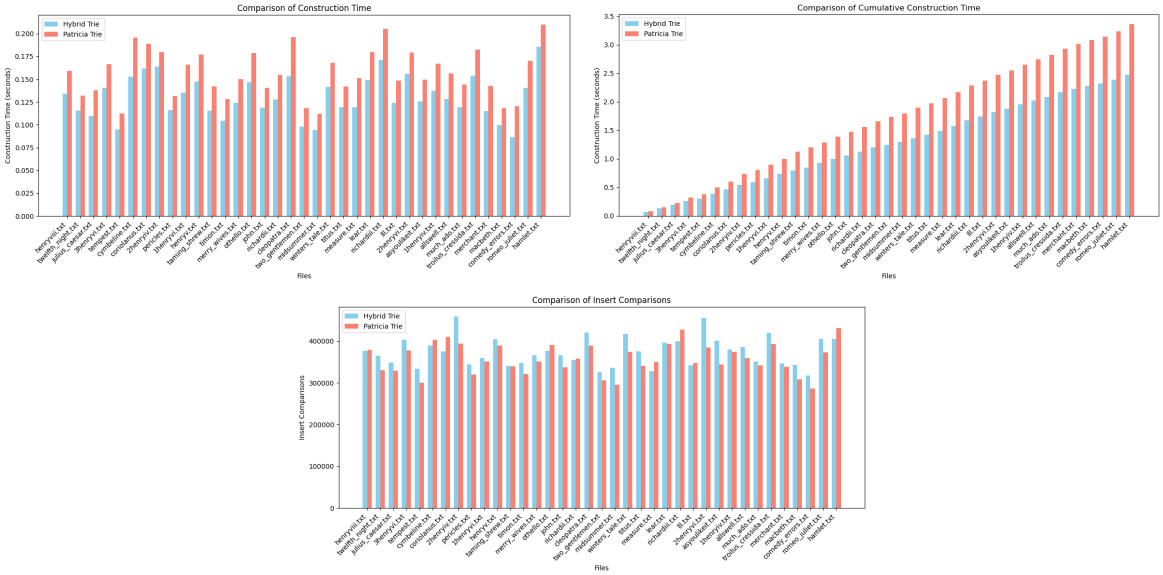

FIGURE 2 – Comparaison du temps de construction et du nombre de comparaisons d'insertion.

3.2 Insertion de nouveaux mots mélangés

Nous avons également testé l'insertion de nouveaux mots mélangés. Nous avons utilisé un outil pour mélanger un txt de Shakespeare et l'avons utilisé comme test pour insérer de nouveaux mots. Les résultats montrent que :

— **Patricia Trie** est souvent **plus efficace** dans ce cas.

— La faible similarité des préfixes entre les mots mélangés réduit le nombre d'opérations de division des nœuds dans le Patricia Trie, ce qui améliore ses performances.

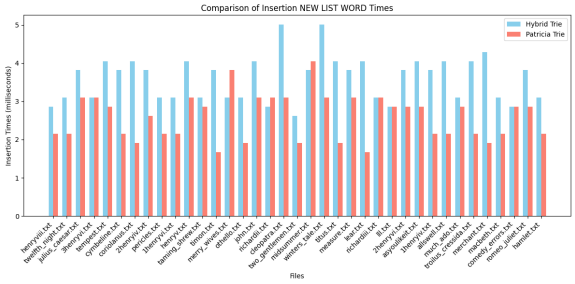

FIGURE 3 – Comparaison des performances pour l'insertion de mots mélangés.

3.3 Comparaison des hauteurs et profondeurs moyennes

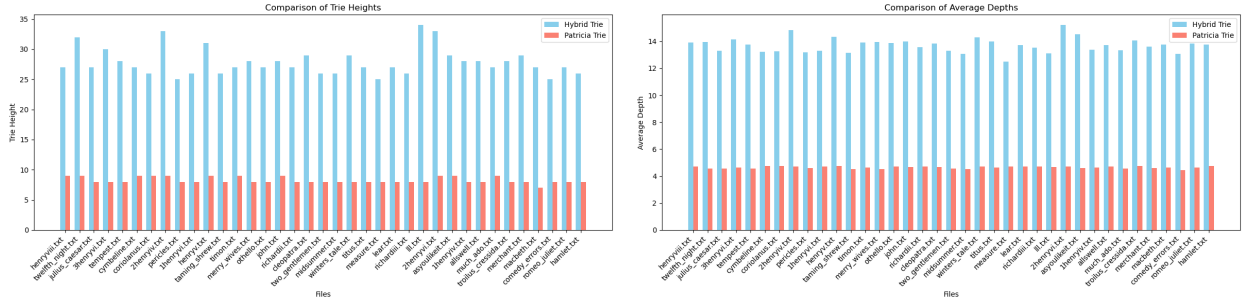

FIGURE 4 – Comparaison de la hauteur et de la profondeur moyenne (Patricia Trie vs Hybrid Trie).

Le **Patricia Trie** réduit significativement la hauteur et la profondeur moyenne en compressant les nœuds pour les préfixes communs, ce qui offre une structure **plus compacte**. En revanche, le **Hybrid Trie** est plus profond car il doit parcourir chaque caractère, augmentant ainsi la hauteur et la profondeur.

3.4 Comparaison des suppressions

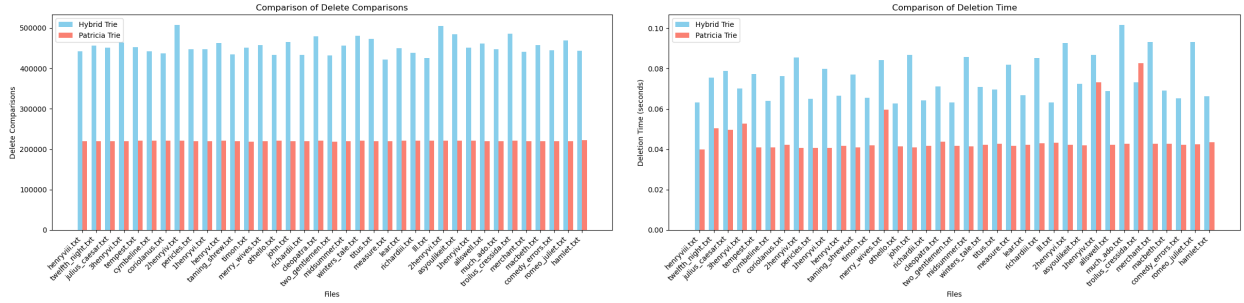

FIGURE 5 – Comparaison des comparaisons et du temps de suppression.

Le **Hybrid Trie** présente un coût en comparaisons et un temps de suppression plus élevés en raison de sa structure **plus profonde**. Le **Patricia Trie** est plus **efficace** pour la suppression grâce à sa structure **équilibrée** et **compacte**.

3.5 Comparaison des recherches

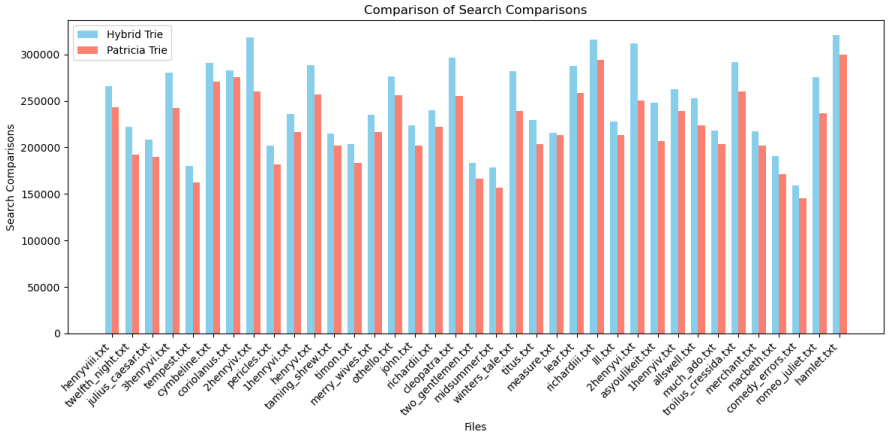

FIGURE 6 – Comparaison des comparaisons de recherche.

La **compression des préfixes** dans le Patricia Trie améliore considérablement les performances des opérations de recherche, en réduisant le nombre de comparaisons. Le **Hybrid Trie**, nécessitant un parcours jusqu'au dernier caractère, est **moins performant** pour les recherches.

3.6 Conclusion

- **Hybrid Trie :** *gestion flexible*, plus rapide à construire mais nécessite plus de **comparaisons** pour les opérations de recherche et de suppression en raison de sa **profondeur** plus importante.
- **Patricia Trie :** plus efficace pour les opérations de **recherche** et de **suppression** grâce à une structure **plus compacte** qui réduit le nombre de comparaisons. Il est également **plus économe en mémoire**.