Devoir de Programmation
UE Algorithmique Avancée

AYED Fatima, KONÉ Daba

Table des matières

1.1 Structure 1 : Patricia Trie

La structure utilisé pour nos Patricia trie (retrouvable dans le fichier Patricia Trie) se repose sur la classe Patricia_Trie qui est essentiellement une liste d'Elements, Elements qui est une classe interne à Patricia_Trie qui est composé de 2 attributs :

- String val : Représente une chaîne de caractère dans la patricia trie
- Patricia_Trie next : Pointeur vers une sous-liste de Patricia_Trie

Les fonctions de la classe Elements sont des accesseurs et des poseurs pour les deux attributs de la classe.

Les attributs de la classe Patricia_Trie sont :

- char fin_chaine : caractère de fin de mot (qui est le caractère ascii 0 dans notre cas)
- Elements liste[] : liste d'Elements qui représente la structure d'un arbre Patricia avec la liste qui représente le noeud principal qui stocke des Elements qui ont les chaînes de caractère qui compose l'arbre Patricia et les pointeurs vers les sous-noeuds
- Boolean mot_vide : un booléen qui détermine si le mot vide est présent dans l'arbre Patricia
- int nb_nil : compte le nombre de case null de liste
- int comparaisonCpt : attribut qui compte le nombre de comparaison pour la questions 6.12 du projet

1.1.1 Primitives de base

- String plusGrandPrefixe(String mot1,String mot2) : renvoie le plus grand préfixe entre les 2 mots
- Elements getElm_At(String m) : Renvoie l'Elements du tableau "liste" auquel la conversion du premier caractère ascii de m -1 (car le caractère ascii 0 n'appartient pas au mot de l'arbre Patricia) correspond
- void setElm_At(String at,String val,Patricia_Trie next) : crée un nouveau Elements dans la liste à la position indiqué par le premier caractère de at avec comme attribut val et next
- Boolean isMotVide() : Détermine si le caractère vide appartient à l'arbre de patricia

1.1.2 Insertion dans l'arbre

(Aussi pour le reste des fonctions le mot clé fera référence au éléments de la liste "liste")

Afin de créer un arbre de Patricia par ajout successif de mot, nous avons crée la fonction PatInsertion(String mot) qui va appeler la fonction auxiliaire PatInsertionBis(String mot,Elements[] first) qui va elle réaliser l'insertion du mot (la raison de l'existence de cette fonction est pour tout simplement avoir la référence du noeud racine afin d'ajouter le mot vide à l'arbre que si on s'y trouve dessus).
La fonction PatInsertionBis fonctionne de la manière suivante :

- Si le mot a inséré est le mot vide alors on vérifie si on se trouve dans bien dans la racine si oui on met à jour le booléen mot_vide à true sinon on fait rien
- Sinon si le mot a insérée doit être placé dans une case vide de liste alors on crée un nouveau Elements à cette case qui contiendra le mot avec le caractère de fin de mot ajouté et un pointeur vers un sous-noeud vide

- Sinon un mot se trouve déjà sur la case où notre mot doit être inserer et alors plusieurs cas sont possibles :

    - Soit on insère un mot suffixe ou égal au mot de la clé trouvé et alors on insère le mot privée du mot trouvé (qui est préfixe du mot à insérer) dans le sous-noeud de la clé avec PatInsertionBis sur ce sous-noeud avec le nouveau mot et first qui reste pareil .

    -Soit on insère un mot qui à un un préfixe commun avec la valeur de la clé et y'a deux options possibles, la première c'est que le préfixe commun est différent du mot a inséré et alors on prend le mot a inséré et la valeur dans la clé qu'on prive du préfixe commun obtenue puis on crée un nouveau noeud où on on insère les deux mots et on fait en sorte que l'ancien sous-noeud de la clé soit maintenant le sous noeud de la valeur de la clé privé du préfixe inséré dans le nouveau noeud et que ce nouveau noeud créé soit le sous noeud de la clé qui a maintenant comme mot le préfixe commun obtenu.
    La deuxième est que si le préfixe commun obtenu vaut le mot qu'on cherche à insérer, alors on a encore 2 possibilités
    1. la valeur de la clé a le caractère de fin de fin de mot , dans ce cas on va insérer le suffixe du mot dans la clé dans un nouveau sous-noeud créé puis on va placer le mot à insérer dans la clé avec le caractère de fin de mot et faire pointé la clé à ce nouveau sous-noeud.
    2. la valeur de la clé ne l'a pas et ainsi on ajoute le suffixe de la valeur de la clé au début de tous les mots du sous-noeud et on met le mot à insérer dans la clé avec le caractère de fin de mot (pour éviter l'ajout de sous-noeud alors que le mot n'appartient pas à l'arbre).

1.1.3 Fonctions avancées

1.1.3.1 Rechercher_mot(String mot)

Tout comme la fonction d'insertion, la fonction rechercher_mot va naviguer récursivement dans l'arbre de nœud en nœud pour vérifier si le mot existe bien dans l'arbre ou pas. À la différence de l'insertion, il n'y aura pas de manipulation de clé ou même de cas dans des cas pour s'assurer de l'intégrité de la fonction mais juste 3 vérifications :

- la première est que si on tombe sur une clé qui a le même mot que le mot rencontrer on vérifie si la valeur de la clé a le caractère de fin de mot si oui on renvoie vrai sinon on renvoie faux
- la deuxième est la taille du mot recherché car si on tombe sur un mot différent de la valeur de la clé mais le mot recherché est plus petit, on saura que le mot ne sera pas dans l'arbre
- la troisième s'opère juste après la taille et c'est si le mot qui sera plus long ou égaux alors on vérifie si la valeur de la clé est préfixe si oui c'est qu'on doit vérifier dans les sous-noeuds si le mot se trouve bien dans l'arbre sinon c'est que le mot n'est pas dans l'arbre

Et pour la récursion, on prend que la partie du mot recherché dont la longueur dans la chaîne est supérieur à la longueur de la clé.

1.1.3.1.2 Complexité théorique

Pour la complexité de rechercher_mot ou même des autres fonctions je vais supposer que les fonctions propre au module de java ont une complexité de O(1) sur les mots qu'on manipule.

Pour rechercher_mot, on voit que notre seule opération coûteuse est le parcours de l'arbre qui est au pire cas en O(h) avec h la hauteur de l'arbre.

1.1.3.1.2 Complexité expérimentale

Nous avons construit l'arbre de Patricia de l'ensemble des œuvres de Shakespeare grâce au fonction de lecture disponible sur Java..
Nous avons ajouté un compteur comparaisonCpt qui compte le nombre d'appels récursifs lors de la recherche du mot "floccinaucinihilipilification" qui est un vrai mot anglais, l'un des plus longs et on arrive à un total de 1 appel récursif ce qui est bien sûr dans la complexité en pire cas de O(h) avec dans cet exemple un arbre de hauteur 11. Si on prend plutôt un mot du dictionnaire de Shakespeare comme "strength" on a alors 5 appels récursif ce qui respectable vu la complexité pire cas.

1.1.3.2 ComptageMots()

ComptageMots est une fonction qui va parcourir chaque clé de chaque noeud de l'arbre en comptant à chaque fois s'il trouve un mot qui se termine avec le caractère de fin de mot et en appliquant ComptageMots() à tous les sous-noeuds des clés non vides qu'il trouve.

1.1.3.2.2 Complexité théorique

Ce qui donne comme complexité à ComptageMots, une complexité en O(n) avec n le nombre de clé de l'arbre vu qu'on devra tous les parcourir dans tous les cas. Le nombre de clés a été pris sur le nombre de mots car pour chaque insertion qui crée un préfixe ou un suffixe d'un mot déjà existant , un nouveau nœud est créé et ainsi une nouvelle liste de clés se suivent.

1.1.3.2.2 Complexité expérimentale

En plaçant un compteur sur le nombre de clé qu'on parcours lors qu'on utilise ComptageMots on s'aperçoit dans l'exemple de Shakespeare pour 22305 mots différents insérés dans l'arbre on parcours un total de 2628392 cases de noeuds de l'arbre, ce qui nous donne 2628392/127=20696 noeuds créé au total, qui couplé avec un hauteur de 11 nous informe que l'arbre est très étalé sur la largeur et non pas sur la longueur. Tout ça nous montre que pour un temps d'accès d'insertions,recherche,suppression et préfixe très rapide avec une hauteur basse, on a sacrifié un temps de parcours de nœuds de l'arbre énorme, et aurait dû nous faire soulever plus la question de la complexité spatiale que temporelle vu que pour une différence que de 2000 entre le nombre de mots et de clé on a autant de case mémoire non utilisé.

1.1.3.3 ListeMots()

listeMots() est une fonction qui va appeler une fonction auxiliaire listeMots_sub() qui va ajouter à une liste (arraylist) tous les mots de l'arbre en parcourant récursivement tous les nœuds de l'arbre. En faisant un parcours sur le tableau "liste" de chaque noeud il ajoute tout les mots qui ont le caractère de fin de mot à la liste et s'il existe un sous-noeud appelle récursivement listeMots_sub() sur le sous-noeud et ajoute à sa liste tous les mots de la liste produite l'appel sur le sous-noeud concaténer avec le mot de la clé .
Après avoir obtenu la liste des mots de l'arbre, pour ranger la liste dans l'ordre alphabétique on utilise la fonction sort de java pour trier la liste obenue.

1.1.3.3.2 Complexité théorique

Pour la complexité de ListeMots(),de par la fonction de trie on sait que la complexité de cette fonction sera au moins en O(mlog(m)) m valant le nombre de mots de l'arbre sinon elle dépend aussi de listeMots_sub() qui a elle une complexité minimum en O(n) vu qu'elle parcourt toutes les clés de tous les noeuds sauf qu'en plus pour chaque noeud non vide si

elle a un sous-noeuds va faire une boucle sur tous les mots du sous-noeuds ce qui donne une complexité en O(m) pour cette opération ce qui donne à la fonction ListeMots_sub() une complexité totale pire cas en O(m)+O(n)(ce n'est pas une multiplication car la récursion dans l'arbre et le parcours de la liste des mots du sous arbres ne sont pas directement lié tout en supposant que la concaténation des mots à un temps constant en moyenne).
Ainsi ListeMots a une complexité en O(m)+O(n)+O(mlog(m)) .

1.1.3.3.2 Complexité expérimentale

Pour listeMots, on va subir le même destin qu'avec ComptageMots vu qu'on va parcourir toutes les clés de tous les noeuds de l'arbre ce qui va donner les mêmes complexité (au minimum vu qu'on fait aussi des opérations de concaténations de mots sur la liste des sous-noeuds) que ComptageMots sauf qu'on fait aussi un tri sur ses 22305 mots ce qui nous donne 96 991 = (m*log(m)) ce qui reste négligeable par rapport au nombre de cases qu'on traverse (case du tableau liste sont les clés des noeuds). Du coup, on observe qu'à mesure que le dictionnaire grandis le tri a de moins en moins d'impact comparé au parcours de l'arbre.

1.1.3.4 ComptageNil()

ComptageNil() est comme comptageMots() c'est à dire qu'elle va parcourir tous les noeuds de l'arbre et incrémenter le résultat grâce à la variable nb_nil de la classe Patricia_Trie qui est initialisé à 127 sur la construction d'un Patricia_Trie , décrémenté à chaque utilisations de setElm_At et incrémenté à chaque suppression réussite d'un élément.

1.1.3.4.2 Complexité théorique

La complexité de ComptageNil() est la même que Comptage mots (en mieux même vu qu'on a pas de comparaison à faire) c'est à dire O(n)

1.1.3.4.2 Complexité expérimentale

ComptageNil a exactement la même complexité temps que Comptage mots vu qu'il parcourt toutes les clés des noeuds de l'arbre afin d'obtenir le nombre de cases nil des tableaux liste des patricia_tries. Le nombre de cases null de l'arbre Patricia est de 2600226 cases.

1.1.3.5 Hauteur()

Hauteur est une fonction qui va appeler une fonction intermédiaire profondeurList qui a comme paramètre une liste d'entier (arraylist d'entier) et un entier qui va donner la profondeur. Cette fonction va parcourir tous les noeuds de l'arbre et pour chaque clé qui contient une mot qui se finit par le caractère de fin de mot et qui n'a pas de sous-noeud, va ajouter la profondeur actuelle dans la liste et va aussi faire une récursion si la clé a un pointeur vers un sous-noeud en augmentant la profondeur grâce à l'entier en paramètre.

Après avoir obtenu la liste des profondeurs des mots avec le caractère de fin de mot de l'arbre, hauteur va faire une boucle sur cette liste afin d'obtenir la profondeur maximale et ainsi donner la hauteur de l'arbre.

1.1.3.5.2 Complexité théorique

La complexité de hauteur dépend de profondeurList qui a elle une complexité en O(n) vu qu'elle parcourt tous les nœuds de l'arbre pour avoir les profondeurs des feuilles. Plus la boucle en fonction du nombre de feuille, la fonction hauteur a une complexité moyenne en O(n+m) (vu que les feuilles indique un mot de l'arbre ce qui est assuré même avec la suppression).

1.1.3.5.2 Complexité expérimentale

Pour la hauteur, on se retrouve encore une fois avec une complexité très proche de ComptageMots vu qu'on doit parcourir toutes les clé de l'arbre. Tout ça pour avoir une liste de 7471 éléments (ce qui indique aussi le nombre de feuilles de l'arbre de Patricia et aussi que seulement ⅓ des mots de l'arbre sont des feuilles) dont le parcours de la liste des profondeurs des feuilles est négligeable par rapport au parcours des clés de l'arbre.

1.1.3.6 ProfondeurMoyenne()

ProfondeurMoyenne fonctionne comme hauteur sauf qu'au lieu de prendre la plus grande profondeur de la liste obtenue par profondeurList, elle va faire la somme de ces profondeurs puis la diviser par la taille de la liste pour donner la profondeur moyenne.

1.1.3.6.2 Complexité théorique

La complexité de profondeurMoyenne est donc la même que celle de Hauteur c'est-à-dire O(n+m). La somme et la division sont négligeables ici vu qu'on travaille sur des nombres pas très grands.

1.1.3.6.2 Complexité expérimentale

La même chose que pour hauteur car même parcours des clés de l'arbre avec profondeurList et même boucle sur la liste des profondeurs reçu par profondeurList.

1.1.3.7 Prefixe(String mot)

Prefixe est une fonction qui va parcourir les hauteurs de l'arbre à la recherche d'une clé dont il est préfixe de son mot afin de compter le nombre de mots contenus par les sous-noeuds

de la clé avec comptageMots(). Plus précisément, elle va en premier lieu (si mot donne bien un Elements avec getElm_At sinon 0 mots dont il est préfixe) comparer la longueur de la valeur de la clé et du mot et comparer la taille :

- Si mot est plus que la valeur de la clé, alors si la valeur est préfixe de mot et que le sous-noeud n'est pas vide va faire une récursion sur le suffixe de mot. Sinon 0 préfixe .

- Sinon on vérifie si le mot est préfixe de la valeur de la clé, si oui alors si le sous-noeud n'est pas vide, on va lancer ComptageMots sur le sous-noeud +1 si la valeur à le caractère de fin de chaîne. Sinon si le sous-noeud est vide, on regarde si la valeur à le caractère de fin de chaîne, si oui on renvoie 1 sinon 0. Et si le mot n'est pas préfixe, on renvoie 0.

1.1.3.7.2 Complexité théorique

Dans le pire des cas, pour le préfixe, on a un mot très long qui doit passer pour beaucoup de sous-noeuds avant d'être préfixe d'un autre mot ce qui fait que comme la recherche où on doit parcourir l'arbre verticalement suivant et quand on est enfin arriver à une clé dont on est préfixe de la valeur on appel à la méthode de ComptageMots qui à elle une complexité en O(n') pour n' le nombre de clé alloué sur les sous-noeuds de la clé ce qui donne une complexité en O(h+n') au pire cas.

1.1.3.7.2 Complexité expérimentale

nombre de prefixe de s = 2510 + nb de comparaison = 292990
nombre de prefixe de give = 7 + nb de comparaison = 766
Avec ces deux exemples, on voit que la hauteur ne joue pas un rôle crucial dans la complexité de Prefixe mais dans la complexité de comptageMots qui va être plus importante si le mot dont on cherche le nombre de mots dont il est préfixe est proche de la racine (ce qui confirme aussi l'hypothèse annoncé à la compléxité expérimentale de ComptageMots). Du coup, la hauteur en elle-même est négligeable dans le calcul de la complexité du préfixe.

1.2.3.8 Suppression(String mot)

Suppression va fonctionner comme recherche au niveau de la récursion pour la recherche du mot. Sauf que la différence va se jouer quand on va trouver le mot, avec Recherche on renvoie vrai et on s'arrête alors que pour Suppression, on va faire une vérification suivante :

- Si la clé n'a pas de sous-noeud alors, on va libérer la case de la clé et aussi incrémenter la variable nb_nil
- Sinon, on retire le caractère de fin de mot

Aussi après les étapes de vérifications indiqué dans la déscription de recherche, on va vérifier s'il n'y a plus de case alloué dans le sous-noeud :

- Si oui, on libère le sous-noeud ce qui va nous économiser du temps de calcul sur des fonctions qui doivent regarder tous les noeuds

1.1.3.8.2 Complexité théorique

La complexité de suppression est comme celle de recherche c'est-à-dire dans le pire des cas en O(h).

1.1.3.8.2 Complexité expérimentale

Suppression à la même complexité que la recherche c'est-à-dire une bonne complexité vu qu'on dépend que de la hauteur de l'arbre qui est courte et de l'accès au clé à partir des mots ce qui est un accès direct en O(1).

1.2 Structure 2 : Tries Hybrides

La structure de notre classe Tries Hybrides repose sur la classe Hybridetries dans le fichier source Hybridetries.java .Nous pouvons y trouver une classe interne nommé Racine avec les attributs suivant:

- char ra: représente le caractère de la racine.
- int cpt : représente un entier nul si le nœud ne correspond pas au caractère defin d'un mot existant dans le dictionnaire,s'il est non nul, alors cpt représente l'ordre d'insertion du mot dans l'arbre.
- boolean visited: servira de flag pour certaines fonctions comme Prefixe(arbre, mot).

Il y a notamment plusieurs Getter et Setter pour ces attributs puisqu'ils sont déclarés en private.
Le constructeur initialise tous les champs de la classe Racine.

Les attributs de la classe Hybridetries sont:

- Racine racine : instance de la classe racine, initialisée avec '/' comme caractère qui marque le nœud vide, cpt=0 et visited = false pour indiquer que le nœud n'a pas encore été visité.
- Hybridetries inf: initialisé à null, représente le sous-arbre gauche du nœud.
- Hybridetries eq : initialisé à null, représente le sous-arbre du milieu du nœud.
- Hybridetries sup: initialisé à null, représente le sous-arbre droit du nœud.
- int end_word : un compteur qui sera incrémenté à chaque fois qu'on ajoute un mot dans l'arbre.

Le constructeur crée une instance d'un arbre avec un nœud vide et trois sous arbres nuls.
Nous avons notamment implémenté des Getters et des setters pour certains attributs, notamment:

- setRac() qui nous permettra d'affecter un caractère à un nœud.
- SetCpt() qui nous permettra d'incrémenter end_word
- set_end_of_word() qui permet d'affecter en word à l'attribut cpt de la racine afin de marquer la fin et l'existence d'un mot.
- resetCpt() qui nous permettra de remettre à 0 end_word.

D'autres attributs seront ajoutés au cours du projet, nous les présenterons tout au long du rapport.

1.2.1 Primitives de base

Nous avons implémenter les primitives suivantes:

- char firstchar() : renvoie le premier caractère du mot.
- String remaining() : renvoie le mot sans le premier caractère.
- int length(): renvoie la longueur du mot
- isEmpty(): vérifie si le noeud de l'arbre est vide ou pas (si son attribut ra est à '/');

1.2.2 Insertion dans l'arbre

Afin de construire un arbre de trie hybride par ajout successif de mots, nous avons implémenté une fonction récursive insertKey(String key). voici les grandes étapes de son fonctionnement :

- Si le noeud de l'arbre actuel est vide :
    - Si le mot à insérer correspond à une seule lettre, alors nous considérons que c'est la fin du mot.
      Nous attribuons ce caractère à la racine , incrémentons le end_word et l'attribuons au champ cpt de la racine. Ainsi lors de la recherche, ce mot sera reconnu comme existant dans le dictionnaire et nous pourrons avoir accès à son ordre d'insertion.
      Nous finissons par créer une instance de Hybridetries pour chaque attribut in,eq,sup, ce qui nous servira plus tard à calculer le nombre de pointeurs nuls et nous renvoyons l'arbre.

    - Si la longueur du mot est supérieure à 1, cela signifie que nous n'avons pas encore fini d'insérer toutes les lettres du mot. Nous effectuons les mêmes étapes que précédemment néanmoins nous n'incrémentons pas end_world et nous faisons un appel récursif sur l'arbre du milieu eq avec le reste des caractères du mot au lieu de retourner l'arbre.
- Si le noeud de l'arbre actuel n' est pas vide :
    - Nous récupérons la première lettre du mot et la comparons avec le caractère contenu dans le nœud, si elle est supérieure ou inférieure nous effectuons un appel récursif sur le sous arbre correspondant.Notons que nous n' affectons pas la lettre au noeud car elle y est déjà (c'était donc un mot préfixe d'un autre mot déjà dans l'arbre), il suffit d'ajouter le marqueur de fin de mot pour ajouter le mot dans le dictionnaire.
      On retourne l'arbre.
    - Si la longueur du mot est de 1 mais cette fois le champ cpt de la racine n'est pas nul, cela signifie que le mot que nous cherchons à ajouter au dictionnaire est déjà dedans.
    - On retourne l'arbre.
    - Si la longueur du mot n'est pas de 1 alors ce n'est pas la dernière lettre du mot donc on fait un appel récursif sur le sous arbre du milieu eq avec le reste des lettres du mot.

1.2.2.1 Insertion dans l'arbre

1.2.3 Fonctions avancées

1.2.3.1 Recherche(arbre, mot) → booléen

1.2.3.1.1 Implémentation

Notre fonction Recherche suis la même logique d'appel récursif et de comparaison que notre fonction d'insertion néanmoins ici, on ne cherche pas à manipuler le champs cpt pour ajouter des mots, on teste seulement s'il est non nul lorsque le caractère est égale à celui du noeud courant et que et que ce caractère correspond bien à la dernière lettre du mot que nous recherchons. Dans ce cas là nous renvoyons True. De plus si lors du parcours on rencontre un nœud vide alors le mot n'existe pas donc nous renvoyons False.

1.2.3.1.2 Complexité théorique

Pour la complexité de la fonction de recherche, nous allons négliger le coup de la fonction toLowerCase() car nous ne la considérons pas comme représentative de la complexité de notre algorithme.
A chaque comparaison, notre algorithme effectue un appel récursif lorsqu'il n'est pas sur une condition d'arrêt. Le pire cas correspond à enchaîner les appels récursif sur les sous arbre droit et gauche, ce qui signifie que le mot n'est soit pas dans l'arbre, soit il n'est préfixe d'aucun mot. Cela veut dire qu'il est possible de faire un nombre d'appels récursif égale à la hauteur h de l'arbre.
De plus, pour chaque appel récursif, le pire cas serait de faire 4 comparaison avant d'arriver au prochain appel récursif. Ceci vaudrait une complexité de O(4h), or nous n'allons pas prendre en compte les comparaisons car elle ne varie pas, elle sont ainsi négligeable.

La complexité en pire est donc en O(h) avec h la hauteur de l'arbre.

1.2.3.1.2 Complexité expérimentale

Nous avons construit l'arbre de trie hybride de l'ensemble des œuvres de Shakespeare grâce à la fonction read() disponible dans notre classe de trie hybride.
Nous avons ajouté un compteur cpt_re qui compte le nombre d'appels récursifs lors de la recherche du mot "floccinaucinihilipilification" qui est un vrai mot anglais, l'un des plus long de la langue, et dont nous avons vérifié qu'il n'existait pas auparavant et inséré.
Nous obtenons un nombre d'appels récursifs de 34 et une hauteur de 35.
Essayons avec un nouveau mot long "pneumonoultramicroscopicsilicovolcanoconiosis" qui est le plus long mot de la langue anglaise.
Nous avons une complexité de 50 pour la Recherche() et une hauteur de 50 pour l'arbre.
Ce qui respecte bien, la complexité théorique en O(h) en pire cas.

Essayons maintenant avec un mot directement tiré de l'œuvre de Shakespear “of” et qui ne devrait pas être un pire cas. On obtient un résultat de 4 appels récursifs pour une hauteur de l’arbre qui est de 35 (nous avons enlever les autres tests).
Notre complexité expérimentale est bien cohérente avec notre complexité théorique

1.2.3.2 ComptageMots(arbre) → entier

1.2.3.2.1 Implémentation

Notre fonction repose sur l’implémentation de la fonction Compter_bis(arbre).
Son implémentation suit le même principe de parcours récursif de nos fonctions précédentes.Afin de compter les mots nous effectuons un parcours en profondeur de l’arbre. A chaque fois que le champ cpt du nœud est non nul, nous additionnons 1 avec les trois autres appels récursif. De plus, nous changeons le flag visited à True afin de ne pas prendre en compte ce nœud si on repasse dessus.
Notons que nous commençons la fonction ComptageMots(arbre) par un appel à reset_flag() qui parcours en profondeur tous les nœuds de l’arbre et remets tous les flags à false afin de pouvoir récupérer tous les mots même lorsque lorsque les flags ont été positionné par d’autre fonction.

1.2.3.2.2 Complexité théorique

Peu importe la hauteur de l’arbre, notre fonction passera par tous les nœuds de l’arbre afin de compter tous les mots. Supposons que notre arbre de recherche possède N nœuds, alors , la complexité du comptage du nombre de mots est en O(N). Le coût de la fonction reset_fleg() peut être négligé.

1.2.3.2.3 Complexité expérimentale

Pour mesurer la complexité expérimentalement, nous avons besoin d’avoir le nombre de nœuds de notre arbre, pour ce faire, nous pouvons prendre la taille de chaque mot et les additionner puisqu’une lettre représente un nœud. Nous avons juste à ajouter une variable globale cpt_noeuds et l‘incrémenter lorsque ComptageMots() passe par un nœud afin d’avoir le nombre de nœuds. Nous obtenons 56797 noeuds ce qui parait raisonnable par rapport au nombre de mots de l’arbre qui est de 23087.La hauteur reste de 44, si elle est vrai, cela voudrait dire que c’est un arbre qui s'étend en largeur plutôt qu'en longueur, ce qui est bien l'intérêt du trie hybride afin de pouvoir faire une insertion et une recherche sans aller en profondeur dans l’arbre.
Notre complexité expérimentale à donc l’air d'être cohérente avec notre complexité théorique.

1.2.3.3 ListeMots(arbre) → liste[mots]

1.2.3.3.1 Implémentation

L'implémentation de notre fonction repose sur une autre fonction ajout_mot() qui prend un arbre, un mot courant ainsi qu'un ensemble de chaîne de caractères.
A chaque fois que nous croisons un nœud non vide, nous effectuons d'abord un appel récursif sur les sous arbre inf et sup que nous allons traiter en premier. Lorsque ce traitement sera fini, le programme traitera les sous arbres eq qui nous permettra d'ajouter les mots dans notre ensemble.
Lorsqu'on croise un noeud non vide sur un sous arbre eq,nous rajoutons son caractère au mot courant et si son champ cpt est non nul, nous ajoutons le mot à notre ensemble en construction.Nous positionnons notamment le flag visited qui sera testé à chaque fois que nous passons par un noeud qui représente le caractère de fin d'un mot afin de ne pas ajouter le même mot deux fois dans notre ensemble de mots et de ne pas avoir de doublons.
Utiliser un ensemble Set<String> nous permet de garantire qu'il n'y aura pas de doublons et rend l'algorithme plus efficace car nous pouvons facilement y ajouter des éléments.

Ensuite , la fonction ListeMots(arbre) récupère l'ensemble de mots renvoyé par la fonction ajout_mot() et les retranscrit dans une ArrayListe qui sera triée par ordre alphabétique par la fonction Collection.sort. Utiliser une ArrayListe nous permet de ne pas dépendre d'une taille fixe et d'indice pour parcourir cette liste.
Notons que nous commençons la fonction ajout_mot() par un appel à reset_flag().
Dans l'exemple de notre Main.java, ListeMots(arbre) ne fonctionne pas lorsqu'elle est invoquée après ComptageMots() (elle ne donne pas tous les mots de l'arbre), nous n'avons malheureusement pas réussit à la debugger.Néanmoins, nous pensons que cela vient du fait que certain flags n'ont pas été remis à false malgré l'appel de la fonction reset_flag(), ce qui est assez curieux car la fonction passe bien par tous les noeuds et nous permet d'avoir une fonction CoptageMots() qui renvoie le bon nombre de mots après la suppression de "génial". Elle fonctionne lorsqu'elle est appelée avant la fonction Comptage_mot().

1.2.3.3.2 Complexité théorique

La complexité de notre fonction dépend de plusieurs parametre:

- la fonction ajout_mot() passe par tous les noeuds donc sa complexité est de O(N) avec N le nombre de noeuds dans l'arbre
- retranscrire les mots dans une ArrayListe a une complexité de O(n) avec n le nombre de mots dans l'ensemble.
- la fonction collection.sort() à une complexité en O(nlog(n)) dans le pire cas.

Cela nous fait une complexité en O(N+n+nlogn).
Nous pouvons négliger la complexité de la retranscription car le nombre de mots est généralement plus petit que le nombre de nœuds dans un arbre donc la complexité de notre fonction est en O(N+nlog(n)).
Notre N est beaucoup plus grand que n donc il domine. On a une complexité de O(N).
Finalement après avoir fait quelques expérimentations, nous avons trouvé que nlog(n) domine largement N.
Nous avons donc une complexité en O(nlog(n)).

1.2.3.3.3 Complexité expérimentale

Nous avons 56797 noeuds dans notre arbre ainsi que 23087 mots ce qui correspond au N, Pour la partie que nous avons négligé, nous avons 23087*log(23087)=100737 environ, ce qui dépasse notre N alors que nous avions considérer N comme dominant car nlog(n) croît moins vite et ne dépasse pas N si celui-ci est très grand par rapport à lui. Nous allons donc changer notre complexité théorique.

1.2.3.4 ComptageNil(arbre) → entier

1.2.3.4.1 Implémentation

Nous considérons les pointeurs nuls comme étant des arbres avec une racine vide(nœud vide).
Notre fonction parcourt l'arbre en profondeur et ajoute 1 au résultat lorsqu'elle tombe sur un nœud vide.

1.2.3.4.2 Complexité théorique

Notre fonction parcourt tous les nœuds de l'arbre pour trouver les nœuds vides.Supposons que nous avons N nœuds dans notre arbre, le pire cas serait que chaque nœud de notre arbre ai deux sous arbres vides en plus des feuilles qui elles auront 3 arbres vides.
Si N est notre nombre de nœuds, alors le nombre de nœuds vide (d'arbre vide/pointeurs null) ne devrait pas dépasser 2N, nous allons metre de coté les arbres vide de chaque feuille.
Nous pouvons ignorer la constante et considérer que la complexité est en O(N).

1.2.3.4.3 Complexité expérimentale

Nous avons 56806 nœuds (nous avons changé le mot ajouté pour les tests d'où le fait que la valeur n'est pas la même que précédemment) et la fonction Comptage nous donne un résultat de 113613 pointeurs nuls dans l'arbre. On dépasse largement le nombre de noeuds dans l'arbre mais si nous n'ignorons pas la constante de multiplication, cela nous ferais un pire cas en 56806*2
=113612 ce qui est très proche de notre résultat.
Il est difficile de prédire la complexité en pire cas car la forme de l'arbre peut énormément varier en fonction de s'il y a plusieurs mots qui sont préfixes l'un de l'autre ou pas .
Cela pourrait être dû aussi au fait que nous avons mis de côté les arbres vides des feuilles.

1.2.3.5 Hauteur(arbre) → entier

1.2.3.5.1 Implémentation

Notre implémentation repose en partie sur celle de hauteur_bis() qui prend un arbre hybride, un compteur et un ensemble d'entiers et renvoie l'ensemble des profondeur des feuilles de l'arbre. Elle va donc parcourir en profondeur tout l'arbre et ajouter 1 à chaque fois qu'elle passe sur un nœud jusqu'à arriver à une feuille. Elle ajoutera ainsi la profondeur dans notre ensemble "profondeur" et recommencera pour les autres feuilles de l'arbre.

La fonction Hauteur() récupère l'ensemble des profondeur de l'arbre et renvoie le maximum de cet ensemble, ce qui correspond à la hauteur de l'arbre.

1.2.3.5.2 Complexité théorique

- La fonction hauteur_bis() parcours tous les nœuds. La complexité est donc en O(N) avec N le nombre de nœuds de l'arbre.
- La boucle qui parcourt l'ensemble de profondeurs de l'arbre pour récupérer la hauteur fera va itérer sur les n auteurs de l'arbre. La complexité est donc de O(n).

Nous avons donc une complexité de O(n+ N) mais nous suggérons de négliger n car dans un arbre il y a plus de nœuds que de feuille(chaque feuille représente une profondeur et le max de ces profondeur est la hauteur) .
La complexité est donc en O(N).

1.2.3.5.3 Complexité expérimentale

La complexité expérimentale est le nombre de nœuds 56806. Nous nous sommes rendu compte au dernier moment que cette mesure de complexité ne voulait pas dire grand chose, peut-être aurait-il fallut compter le nombre d'appels récursif afin de parcourir tous les nœuds, or nous avons l'impression que cela revient au même .

1.2.3.6 ProfondeurMoyenne(arbre) → entier

1.2.3.6.1 Implémentation

Il s'agit du même algorithme que la Hauteur. ProfondeurMoyenne() récupère l'ensemble des profondeurs puis itère sur cet ensemble pour faire la somme des profondeur de l'arbre qui sera ensuite divisée par le nombre de profondeur dans l 'arbre.
L'appel de cette fonction sur l'exemple de base est disponible dans Main.java.

1.2.3.6.2 Complexité théorique

Il s'agit ici de la même complexité que Hauteur(), O(N).

1.2.3.6.3 Complexité expérimentale

La mesure de notre complexus paraît assez discutable car nous n'avons pas énormément de choses à débattre dessus.

1.2.3.7 Prefixe(arbre, mot) → entier

1.2.3.7.1 Implémentation

Il s'agit du même algorithme que insert Key(), or ici on ajoute 1 au résultat des appels récursifs à chaque fois qu'on croise un noeud avec un champ cpt non nul car l'algorithme nous met directement sur le chemin du mot, donc lorsqu'on atteint la dernière lettre du mot,

tous le sous arbres eq de ce noeud ainsi que tous les sous arbres droits et gauches du noeuds suivant aurons ce mot en préfixe.

1.2.3.7.2 Complexité théorique

Le mot avec lequel nous effectuons la recherche nous donne directement le chemin à parcourir, il ne parcours pas tout l'arbre, or notre mot peut être préfixe de plusieurs mots dans l'arbre.IL pourrait donc y avoir plus d'appels récursifs que le nombre d'appels fait pour la fonction de recherche par exemple.
Nous allons donc considérer que la complexité en pire est en O(N) avec N le nombre de nœuds dans l'arbre tout en sachant que cela ne devrait jamais arriver car la fonction ne va pas parcourir l'arbre entier.

1.2.3.7.3 Complexité expérimentale

Nous allons compter le nombre d'appels récursifs dont on a besoin pour trouver tous les mots dont notre argument est préfixe.Nous avons appliqué la fonction Prefixe() avec le mot tempestuous qui appartie au fichier tempest.txt.Nous savons qu'elle est préfixe d'au moins deux mots or nous avons toujours un résultat de 1, pareil pour la complexité. Notre fonction est fonctionnelle pour l'exemple de base mais ne semble pas fonctionner correctement pour l'œuvre de Shakespeare, nous n'avons pas pu déterminer la complexité expérimentale.

1.2.3.8 Suppression(arbre, mot) → arbre

1.2.3.8.1 Implémentation

L'implémentation repose sur celle de ajout_mot_bis() qui est exactement la même que ajout mot() avec un détail près: au lieu d'ajouter les mots dans un ensemble, nous ajoutons un tableau de deux chaînes de caractères dans un ensemble, la première case correspond au mot ajouté et la deuxième correspond au champ cpt de la dernière lettre du mot , ce qui correspond à son ordre d'insertion. Ceci nous servira par la suite.
Nous avons notamment implémenté une fonction reset() afin de remettre à vide tous les nœuds de notre arbre.De plus nous avons implanté l'algorithme de tri à bulle bubblesort() qui nous servira à trier les mots selon l'ordre d'insertion.

Stratégie de la fonction suppression:

- On récupère l'ensemble des couple mot-cpt renvoyé par ajout_mots_bis et on les transcrit dans un tableau de tableau de 2.
- On les tri en fonction dans leur ordre d'insertion qui est le deuxième élément des couples (avec bubblesort()).
- On le parcourt en retranscrivant tous les mots dans un tableau simple de chaine de caractère sauf celui que nous voulons supprimer .

- Les mots sont retranscrit dans leur ordre d'insertion, ainsi lorsque nous appelons reset() pour remettre tous les nœuds à vide et réinsérer tous les mots sauf celui que nous avons supprimé, l'ordre n'est pas altéré.

Nous pouvons voir dans Main.java que le premier appel à Suppression fonctionne mais pas le deuxième malgré le fait que nous avons mis tous nos nœuds à jour entre les deux appels.

1.2.3.8.2 Complexité théorique

- ajout_mot passe par tous les nœuds donc sa complexité est de O(N) avec N le nombre de nœuds dans dans l'arbre.
- Pour le tri à bulle, le pire cas est que les ordres soient inversés donc la complexité est de O(n^2) avec n le nombre de mots dans l'arbre
- La transcription a un coup de O(n) mais nous pouvons la négliger
- Nous négligeons le coût de reset() car nous ne la trouvons pas représentative de la complexité de l'algorithme.
- insertKey() appelé à la fin pour réinsérer les mots a une complexité de O(h) car au pire cas aucun mot ne peut dépasser la hauteur de l'arbre précédent, il y à n-1 mots dans l'arbre donc O((n-1)*h)

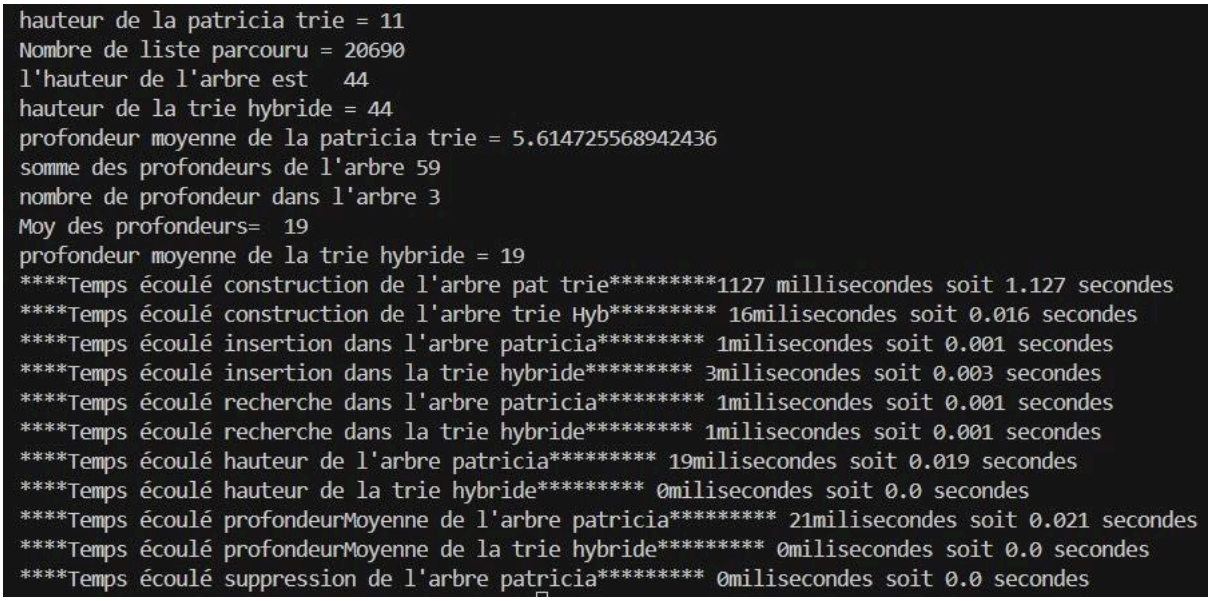
La complexité finale est de O(N+n^2+(n-1)*h).
Les termes qui domines sont n^2 et n*h or le nombre de mot n est largement plus grand que la hauteur donc n^2 domine.
Nous avons donc une complexité en O(n^2).

1.2.3.8.3 Complexité expérimentale

Nous allons nous contenter de mettre un compteur cpt_bulle 19 974 086 et on a n^2=23087^2 = 533 009 569
Nous trouvons une complexité bien inférieure au pire cas et cela même si on ajoute les valeurs négligées. Notre valeur théorique est donc bien respectée.

NOTE
Malheureusement en raison d'un malentendu lors de l'organisation du projet, nous n'avons pas pu implémenter le format json pour le trie hybride. Nous avons seulement pu le faire pour le patricia tries car nous pensions tous les deux que l'autre s'en occupait.

1.2.4 Comparaison des temps d'execution



Pour le temps de construction , le tries hybride est le plus rapide car il y a beaucoup plus de comparaison faite dans le Patricia trie en raison de son implémentation.
Mais une fois construit, l'arbre de Particia est beaucoup plus rapide pour l'insertion et la suppression comme on peut le remarquer sur nos résultats , cela est sûrement dû au fait que sa hauteur soit beaucoup plus courte que celle du trie hybride.Cependant, pour les opérations qui demande de parcourir tout l'arbre, le trie hybride sera plus rapide que la Patricia trie vu que pour la patricia trie on doit parcourir toute le tableau d'Elements qu'il possède (taille 127) pour chaque noeud ce qui réduit considérablement la vitesse de ces fonctions..