Université de Liège

Structure de données et algorithmes

Projet 2 : Mots mêlés

BOUSTANI Mehdi
ALBASHITYALSHAIER Abdelkader

Table des matières

1. Complexité : Recherche et Insertion

Dans cette section, nous décrivons les complexités en temps de la recherche [1], ainsi que de l'insertion des 3 implémentations d'ensemble, en fonction du nombre de mots m dans l'ensemble et de la longueur l du mot recherché. Nous utilisons quelques opérations qui assisteront à la recherche ou/et à l'insertion.

1. `duplicate_string` : Cette fonction sera fréquemment appelée sur le mot à insérer, avec un coût total de 2l $\in \Theta(l)$: l pour `strlen` et l pour `memcpy`.
2. `strcmp` : Sa complexité est $\Theta(l)$, car elle compare le mot à insérer avec d'autres chaînes de caractères.

Arbre binaire de recherche

Recherche

En supposant que les mots du lexique sont ordonnés de manière aléatoire, l'arbre résultant sera équilibré, garantissant ainsi un parcours de complexité $\Theta(\log m)$.
À chaque nœud visité, une comparaison est effectuée entre le mot recherché et la clé du nœud, engendrant un coût de $\Theta(l)$.

Par conséquent, la complexité totale de la recherche est $\Theta(l \log m)$.

Insertion

Pour l'insertion, le processus est similaire à la recherche. Le parcours des nœuds a une complexité de $\Theta(\log m)$, avec une comparaison entre le nœud visité et le mot recherché, coûtant $\Theta(l)$.

Après le parcours, si le mot n'est pas déjà présent, une copie du mot est créée en $\Theta(l)$ (coût de 2l au total) et insérée dans l'ensemble.

Le coût total de l'insertion est donc $l \log m + 2l$, appartenant à une complexité de $\Theta(l \log m)$.

Table de hachage

Recherche

En supposant un hachage uniforme et un facteur de charge constant, la complexité devient $\Theta(1)$ pour repérer le mot recherché, car les tailles des listes sont très petites et donc le temps et peuvent donc être négligeable. Ainsi, le coût total est de $2l$: l pour le hachage et l pour la comparaison entre le mot recherché et la clé de l'élément, aboutissant à une complexité de $\Theta(l)$ pour la recherche.

Insertion

Pour insérer un mot dans l'ensemble, en plus des opérations de recherche, nous devons également prendre en compte une opération supplémentaire (en supposant que malloc $\in \Theta(1)$) :

- La copie du mot à insérer (`duplicate_string`) avec un coût de $2l$.

Le coût total devient donc $l + 2l = 3l$, étant $\Theta(l)$. Ainsi, la complexité de l'insertion est $\Theta(l)$.

1. Dans ce rapport, nous analysons la complexité (avec Θ) uniquement dans le pire cas.

Arbre radix

Recherche

Pour rechercher un mot dans un arbre radix, voici les opérations impliquées :

- **duplicate_string** : Réalisée sur la racine de l'arbre (qui est principalement un noeud de clé vide), ce qui prend $\Theta(1)$. Ceci nous permet d'initialiser nos chaînes plus facilement.
- **Parcours des nœuds** :
  - **Comparaison** : À chaque nœud, nous comparons sa clé avec le mot à insérer, ce qui prend $\Theta(l)$.
  - **Parcours des arêtes sortantes** : Concaténation du préfixe actuel avec l'étiquette de l'arête, coûtant $2l$ en tout (l pour realloc et l pour strcat), donc $\in \Theta(l)$.
  - **isPrefix** coûte $\Theta(l)$.
  - **copy_string** (dans les 2 cas de `isPrefix`) coûte $\Theta(l)$.

    Ainsi, pour chaque arête, le coût total est $2l + l + l = 4l \in \Theta(l)$.

Cela pourrait nous mener à un pire cas correspondant à $\Theta(l)$, où nous devons suivre chaque préfixe du mot recherché (lorsque tous les préfixes du mot sont présents dans l'ensemble). Ainsi, la complexité de la recherche correspond à $\Theta(l)$.

Insertion

En plus des opérations de recherche, plusieurs opérations s'ajoutent, telles que la recherche d'un préfixe commun entre le reste du mot et l'étiquette de l'arête rencontrée, l'appel à `isPrefix`, et l'ajout d'une arête. Toutes ces opérations ont une complexité $\in \Theta(l)$. Cependant, la complexité totale reste de l'ordre de $\Theta(l)$.

2. Complexité : setGetAllStringPrefixes

Arbre binaire de recherche

Pour récupérer tous les préfixes d'une clé, nous définissons d'abord une borne minimale et maximale des préfixes [2].

Ensuite, nous faisons appel à une fonction auxiliaire `fillPrefixes` qui ajoute les préfixes trouvés à la liste. Cette fonction cherche les préfixes de manière récursive en en une seule descente dans l'arbre, en choisissant à chaque étape la bonne branche (le bon sous-arbre) pour continuer la recherche.

Cette fonction commence la recherche à partir de la racine de l'arbre, où à chaque noeud rencontré, nous effectuons 2 comparaisons :

1. Une comparaison entre la clé du noeud courant et le préfixe minimum du mot, nous donnant un coût au pire cas de $l \in \Theta(l)$ lorsque le préfixe minimum est égal au mot.
2. Une comparaison entre la clé du noeud courant et le préfixe maximum du mot, avec un coût de $l \in \Theta(l)$

Le coût de ces deux comparaisons est donc de $2l \in \Theta(l)$.

Ensuite, nous appelons la fonction `isPrefix` pour déterminer si la clé du nœud est un préfixe du mot, avec un coût de $l \in \Theta(l)$. Si la clé du nœud est un préfixe, nous l'insérons dans la liste des préfixes et choisissons le/les sous-arbre(s) en fonction des cas suivants :

2. Dans ce projet, nous avons supposé que des mots d'une lettre peuvent être présents

- La clé du nœud est à la fois le préfixe minimum et maximum : nous arrêtons la recherche.
- La clé correspond au préfixe minimum : nous prenons le sous-arbre de droite.
- La clé correspond au préfixe maximum : nous prenons le sous-arbre de gauche.
- Si la clé est un autre préfixe, nous choisissons les deux sous-arbres, avec un coût supplémentaire de $l \in \Theta(l)$ pour la détermination de nouvelles bornes préfixes.

Si la clé du nœud n'est pas un préfixe, nous avons les choix suivants :

- La clé est supérieure au préfixe maximum : sous-arbre de gauche.
- La clé est inférieure au préfixe minimum : sous-arbre de droite.
- La clé est comprise entre le préfixe minimum et maximum : nous choisissons les deux sous-arbres, avec un coût supplémentaire de $l \in \Theta(l)$ pour déterminer de nouvelles bornes.

En résumé, le parcours de l'arbre a un coût de $\log m$. Compte tenu des opérations supplémentaires ayant une complexité $\Theta(l)$, la complexité totale est $\Theta(l \log m)$.

Table de hachage

Pour chercher tous les préfixes d'un mot dans l'ensemble, nous utilisons la même stratégie de hachage tout en parcourant le mot une seule fois : $\Theta(l)$, en mettant à jour un compteur à chaque itération et en appelant `isPrefix` (étant $\Theta(l)$) tout en évitant d'appeler `hashFunction` sur chaque préfixe individuellement. Cela réduit notre complexité à $\Theta(l^2)$

Arbre radix

Dans cette implémentation, nous utilisons 2 chaînes de caractères : `pref` pour le préfixe courant et `tmp` pour stocker le dernier préfixe. Leur création est en temps constant, partant de la racine, qui est initialement une chaîne vide.

Nous parcourons l'arbre en suivant les préfixes du mot. À chaque nœud rencontré, nous comparons sa clé avec le mot pour détecter le préfixe maximum. Cette opération a un coût de $l \in \Theta(l)$, sans compter l'insertion dans la liste et la duplication de la clé.

Si la clé du nœud ne correspond pas au mot, nous vérifions si elle est un préfixe du mot avec `isPrefix` ($l \in \Theta(l)$). Si tel est le cas, nous insérons la clé dans la liste. Si le nœud est une feuille, la recherche des préfixes est terminée. Sinon, nous poursuivons la recherche.

À partir du nœud courant, nous parcourons les arêtes sortantes en concaténant le préfixe courant avec l'étiquette de l'arête ($\in \Theta(l)$). Nous vérifions si le résultat est un préfixe (l de coût). Si oui, nous mettons à jour `tmp` et passons au nœud suivant. Sinon, nous revenons au préfixe précédent et passons à l'arête suivante. Dans les 2 cas, le coût est au pire l. En résumé, le coût total est de $3l \in \Theta(l)$ pour chaque nœud parcouru, plus $3l \in \Theta(l)$ pour chaque arête de ce nœud, menant à une complexité $\in \Theta(l)$.

3. Recherche dirigée par le lexique et par la grille

Supposons que la longueur des mots du dictionnaire peut atteindre N.

3.1. Recherche par la grille

En parcourant la grille ($N \times N \in \Theta(N^2)$), nous appelons `addFoundPrefixes` sur chaque case. Grâce à `getWord` ($\Theta(N)$), nous récupérons les mots dans les 8 directions [3]

Arbre binaire de recherche

Lorsqu'une ligne est récupérée de la grille, nous appliquons `setGetAllStringPrefixes` pour remplir la liste [4] des mots trouvés dans l'ensemble : $\Theta(N\ log\ m)$ au pire cas. Jusqu'à présent, le remplissage des préfixes dans la liste à partir d'une case nous donne un coût de $N + N\ log\ m \in \Theta(N\ log\ m)$. Ceci donne au total une complexité totale de $\Theta(N^2 \times N\ log\ m) = \Theta(N^3\ log\ m)$.

Table de hachage

Lorsqu'une ligne est récupérée, nous appliquons `setGetAllStringPrefixes` : $\Theta(N^2)$ au pire cas, donc `addFoundPrefixes` aura un coût de $N + N^2 \in \Theta(N^2)$. Ainsi, la complexité totale est $\Theta(N^2 \times N^2) = \Theta(N^4)$.

Arbre radix

De la même manière, en appliquant `setGetAllStringPrefixes` : $\Theta(N)$, `addFoundPrefixes` aura un coût de $N + N = 2N \in \Theta(N)$. Ainsi, la complexité totale est $\Theta(N^2 \times N) = \Theta(N^3)$.

3.2. Recherche par le lexique

Cette recherche commence par parcourir la liste des mots du lexique avec un parcours de $m \in \Theta(m)$, et pour chaque mot, nous appelons **boardContainsWord**, qui doit parcourir toute la grille ($N \times N$). Depuis chaque case, nous recherchons le mot dans les 8 directions (voir `searchDirection`) : $\Theta(N)$ au pire cas.

Cela nous mène à une complexité totale de $\Theta(N^2 \times N \times m) = \Theta(N^3\ m)$.

4. Évaluations empiriques des temps de calcul

Dans cette section, nous mesurons empiriquement les temps de calcul [5] des quatre solutions pour deux tailles de grille croissantes : 100 et 300.

Recherche dirigée par le lexique

100	300
8.621	76.127

Recherche dirigée par la grille

Dictionnaire	100	300
Arbre binaire de recherche	0.223	2.311
Table de hachage	0.145	3.133
Arbre radix	0.153	1.760

D'après ces résultats, nous remarquons que le temps de calcul de la recherche dirigée par le lexique est beaucoup plus élevé par rapport à la recherche dirigée par la grille, ce qui est

3. Le cas où le mot atteint une taille N n'arrive pas dans toutes les 8 directions, cette fonction peut être bornée par $O(8N)$.
4. Nous ne prenons pas en compte la construction de cette liste
5. Les temps de calcul sont mesurés en secondes.

cohérent avec les complexités de chaque solution. En effet, la recherche par le lexique dépend principalement de la taille du lexique, qui est souvent beaucoup plus grande que la taille de la grille, ce qui joue un rôle significatif.

Dans la recherche par la grille, pour une taille de 100, nous observons une légère différence entre les trois solutions. Cependant, pour une taille de 300, nous commençons à remarquer un écart plus important. Le temps de calcul dans la table de hachage devient plus élevé que celui dans l'arbre binaire de recherche. La table de hachage dépend principalement de la taille de la grille ($\Theta(N^4)$), tandis que l'arbre binaire de recherche dépend à la fois de la taille de la grille et du nombre de mots dans le lexique ($\Theta(N^3\ log\ m)$). Ainsi, lorsque la taille de la grille augmente, la différence entre les deux devient plus importante en faveur de l'arbre binaire de recherche.

Pour l'arbre radix, sa solution est donc la plus efficace, car elle ne dépend que de la taille de la grille ($\Theta(N^3)$).

5. Analyse critique des différentes solutions

En conclusion, nous avons pu observer l'efficacité de la recherche par la grille par rapport à la recherche par le lexique. La recherche par le lexique dépend en partie du nombre de mots dans le lexique, ce qui joue un rôle important, surtout lorsque le dictionnaire est volumineux. En revanche, la recherche par la grille n'est pas fortement influencée par le nombre **total** de mots dans le lexique.

En ce qui concerne les solutions de la recherche par la grille, nous avons constaté que l'utilisation d'un arbre radix est la plus efficace, suivie par l'arbre binaire de recherche, qui est légèrement moins efficace [6]. Enfin, la table de hachage est la moins efficace pour la recherche par la grille, surtout lorsque la taille de la grille devient plus importante.

6. Avec une différence de $log\ m$ au niveau de la complexité par rapport à l'arbre radix