Dictionnaire Rapport

2.1.1 Analyse

1. Pour les deux implémentations :

a) Expliquez brièvement vos choix d'implémentation.

b) Expliquez le principe de la fonction d'intersection, en donnant son pseudo-code dans le cas de l'arbre binaire de recherche.

Pour l'intersection, dans le cas de deux arbres binaire de recherche A et B, on parcourt l'arbre A en entier de nœud en nœud. Ensuite pour chaque noeud parcouru, on teste la condition si le nœud est contenu ou pas dans l'arbre B. Pour cela, la fonction contain() est appelé.

Pseudo code :
-Création d'un tableau pour contenir l'intersection.
-Appel de le fonction intersectionArbre() qui prend en arguments, 2 pointeurs vers un arbre et 1 pointeur vers ce tableau.
-Si un des deux arbres est NULL, l'intersection est nulle, et on ne fait rien.
-Sinon, la fonction contain() est appelé sur l'arbre B et teste la valeur de l'arbre A passé en argument, c'est-à-dire au début du programme cela correspond à la valeur de la racine.
-Si ce test est positif, alors on ajoute cet élément au tableau de String.
-Puis on parcourt l'ensemble de l'arbre A de nœud en nœud avec un parcours préfixe, et on effectue le même test d'appartenance *contain()*. Le parcours se fait grâce à la récursivité : on appelle la fonction elle-même mais dans l'ordre ; pour le fils gauche, puis pour le fils droit (parcours préfixe).
-La condition d'arrêt de cette récursivité est le fait d'appeler cette fonction sur les fils des feuilles : c'est-à-dire pour des arbres NULL. A partir de là, la fonction est dépilé.

c) Donnez et justifiez la complexité au pire cas de la fontion setIntersection en fonction des tailles nA et nB des deux ensemble (en supposant que nA <= nB)

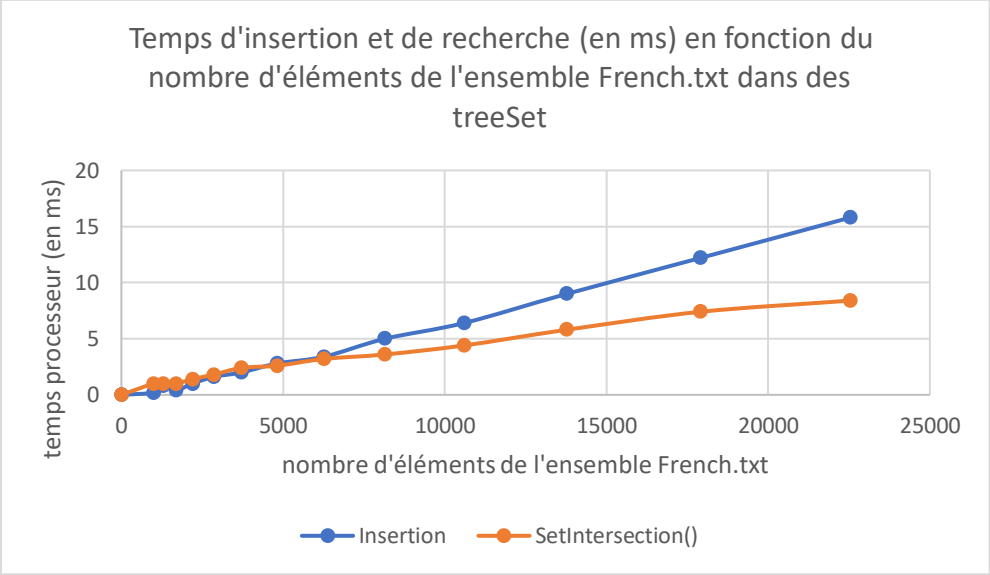
Dans le cas de la table de hachage, on parcourt les listes contenues dans chaque cases de la table de hachage, ce parcours dépend de nA, le nombre d'élément de l'arbre A. Pour chaques nœuds de cette liste, on appelle la fonction contain() qui va calculer le hashcode, ce calcul se fait en complexité constante. Puis on va effectuer la recherche dans l'arbre B, dans la liste qui se trouve à la case d'indice égal au nombre haché. Cette recherche dans le pire des cas va conduire à une complexité en O(nB) dans le cas d'une fonction de hachage qui renverrait les éléments dans la même liste.

Cependant, la fonction de hachage implémentée dans le projet assure un nombre haché unique pour chaque chaines de caractère. Ainsi, pour la recherche, on parcourt constamment dans l'arbre B, une liste de taille constante égale au nombre d'élement de l'arbre B, nB divisé par le nombre de cases de la table de hachage contentant l'ensemble B, disons m.
Dans ce cas ci, la complexité serait donc de nA*(nB/m). Mais cette complexité est encore plus faible dans le projet, car la taille de la table de hachage m dépend du nombre de l'arbre nB ce qui rend (nB/m) en une constante. Donc on obtient finalement une complexité linéaire qui ne dépend que de nA : **O(nA) .**

Dans le cas de l'arbre binaire de recherche , on parcourt chaque nœud de l'arbre A, la complexité dépend donc de nA le nombre de nœuds de l'arbre A. Ensuite pour chaque noeud parcouru, le test d'appartenance contain() est appelé. Cette fonction est de complexité O(nB) aussi car dans le pire des cas l'arbre B est un arbre dégénéré et se rapproche donc de la recherche d'un élément dans une liste chainée triée. Mais en moyenne, l'arbre sera plus ou moins équilibrée et la complexité se rapprochera plus de log(n). En effet, dans un arbre de recherche, la recherche se fait rapidement en moyenne car on va à chaque fois effectuer la recherche dans un

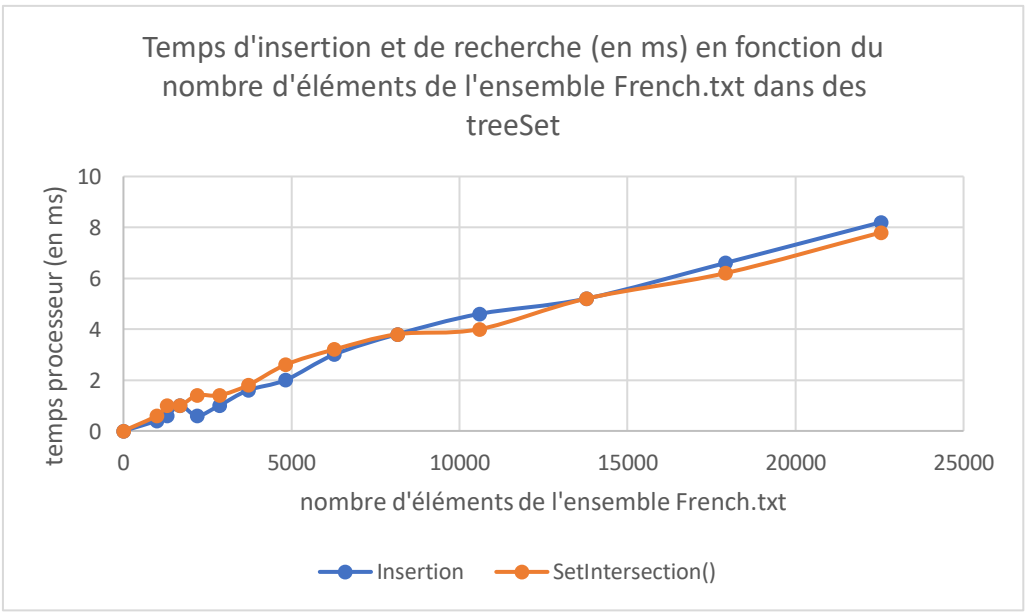
espace plus restreint, c'est une recherche dichotomique. Finalement la complexité dans le pire des cas est en **O(nA*nB).**

2) Expérimentation des temps d'insertion et de recherche en comparaison avec les temps théoriques.



Dans le cas de l'arbre binaire de recherche, les données expérimentales pour l'insertion montrent une insertion en complexité soit linéaire soit en n.log(n) (quasi linéaire) . En théorie, l'insertion des éléments dans l'ensemble A devrait se faire en moyenne en nA.log(nA).

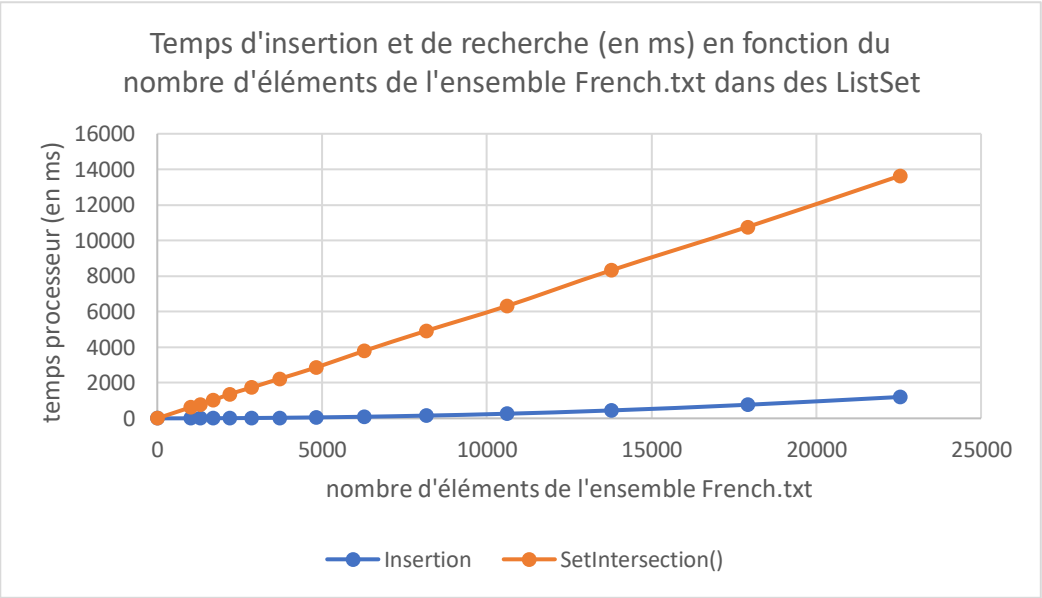
Pour l'intersection, et donc pour la recherche, à l'œil nu, cela ressemble à une complexité en log(nA), car le taux d'accroissement a l'air de baisser au fur et à mesure qu'on aggrandit l'ensemble A. En theorie, l'intersection de deux arbres doit se faire en moyenne en nB.log(nA), or nB étant constant (nombre d'élément dans l'ensemble B, English.txt) cela se fait bien en log(nA).



Dans le cas de la table de hachage l'insertion et la recherche, ont plûtot une courbe qui évolue linéairement.

Pour l'insertion, cela se fait aussi en O(nA) car l'insertion est bien constante elle aussi (calcul du hashcode, et placer l'élément en tête de liste à la position calculée), mais on fait l'opération d'insertion, nA fois ce qui fait bien une complexité en O(nA).

Puis pour l'intersection, cela doit bien se faire en O(nA), car la recherche est bien constante(quest 2.c), mais on le fait un nombre de fois égal à nA, ce qui fait une complexité en O(nA) ce qui confirme nos données.



Enfin, pour les listes chainée, la recherche est linéaire. Quant à l'insertion, la croissance a l'air quadratique (La courbe zoomé en annexe).

En theorie, l'insertion est très rapide dans une liste chainée, car on insère toujours en tête de liste. C'est donc en temps constant. Mais avant d'insérer on doit vérifier si le mot est déjà contenu dans la liste. Or la fonction contain() parcourt toute la liste, ce qui donne une complexité en O(nA). De plus on fait cette opération nA fois. Ce qui donne finalement une complexité en O(nA^2) ce qui correspond bien à la courbe quadratique.

Pour l'intersection, pour chaque nœuds de l'ensemble A on utilise contain() dans l'ensemble B. Donc cela se fait en O(nA * nB). Or nB est constante. Donc cela se fait en O(nA) ce qui correspond bien à la courbe linéaire théorique.

3.2 Analyse

1) La complexité minimale pour calculer l'intersection de A et B, deux tableaux est nA*nB, car il faut pour chaque cases de A, rechercher dans l'ensemble B, on ne peut pas raccourcir cette complexité car aucun ensemble n'est trié, il n'y a aucun moyen de faire une recherche plus rapide que O(n). Cela dépendra forcément linéairement de la taille de A et de B.

2) a) Pour la première approche, on stocke dans un arbre binaire de recherche les éléments de A. La complexité pour faire cette opération est en log(nA) en moyenne mais dans le pire des cas, c'est en O(nA).

Ensuite on parcourt les éléments de l'ensemble B et on retient que les éléments qui sont dans l'arbre donc on effectue une recherche un nombre de fois égal au nombre d'éléments de B, O(nB). De même, la recherche d'un élément se fait en log(nA) en moyenne, mais au pire des cas elle est en O(nA).

Au final on a une complexité de O(nA) + O(nA*nB).

Pour la seconde approche on stocke les deux ensembles dans des arbres. Cela se fait donc en O(nA+nB). Ensuite on utilise la fonction setIntersection(). L'intersection de deux arbres se fait dans le pire des cas en nA*nB si l'arbre dans lequel on effectue la recherche, est dégénéré.

Au final on a une complexité de O(nA+nB) + O(nA*nB)

Etudions le cas de la table de hachage. Pour stocker une valeur dans une table de hachage, cela se fait en O(1), donc pour nA valeur cela se fait en O(nA).

Ensuite on parcourt les éléments de B et on retient que les éléments qui sont dans l'arbre, donc on effectue une recherche un nombre de fois égal au nombre d'éléments de B. La recherche dans notre table de hachage se fait en temps constant car la taille du tableau est proportionel au nombre d'éléments et la fonction de hachage donne une écriture unique pour chaque chaine de caractères.

Au final on a une complexité de O(nA) + O(nB)

Pour la seconde approche on stocke les deux ensembles dans des tables de hachages. Cela se fait donc en O(nA + nB).

Puis on appelle la fonction setIntersection(). Cela se fait en O(nA) car la recherche est en temps constant et la fonction fait nA fois cette recherche.

Au final complexité de O(nA+nB) + O(nA)

b) Pour la première approche la meilleure solution est d'utiliser une table de hachage, et pour la seconde approche aussi. Mais ici on n'évalue que les pires cas, et dans le cas de notre implémentaion. Il se peut pour la table de hachage que la fonction de hachage rende la complexité plus longue. De plus, en moyenne, les arbres seront rarement aussi déséquilibrés que dans le pire cas.

3) En supposant que A et B soient triés par ordre alphabétique alors on peut utiliser directement les 2 tableaux. On parcourt un tableau en entier, et on effectue la recherche dichotomique dans l'autre. Cela fait une complexité en nA * log(nB). Il n'y a pas d'insertion à faire et on obtient une complexité quasi-linéaire. Sinon on peut utiliser le HashSet, pour une recherche plus rapide (avec fonction de hachage adaptée, et table proportionnelle au nombre d'élément). Le problème de l'arbre est que insérer des ensembles triés dans le cas de notre implémentation va créer des arbres extrêmement déséquilibrés.

ANNEXE :