Rapport du projet - SDA2
Louis-César PAGES et Chloé RICHE

1. Ensemble ordonné
    a. La structure ensemble, “Ens” est représenté à l’aide d’une liste chaînée d’éléments “el” constitués d’un entier : “val” et d’un pointeur vers l’élément “el” suivant. Cette structure rend les opérations d’insertion, d’accès et de suppression plus complexes mais une liste chaînée nous permet d’allouer de la mémoire pour des ensembles de taille variable et de trier les éléments par ordre croissant de leur valeur. Le fait que les éléments soient triés par ordre croissant nous permet de réduire la complexité de recherche de valeur. En effet si un élément est plus petit que l’une des valeurs d’une structure représentant un ensemble, alors cet élément n’appartient pas à la partie supérieur de cette structure.
    b. La fonction insertion vérifie dans un premier temps que les ensembles donnés en paramètre sont implémentés. On crée des pointeurs sur ces deux ensemble afin de pouvoir manipuler la structure et comparer les valeurs. On initialise un nouvel ensemble qui servira à stocker le résultat de l’intersection. On parcours dans un premier temps l’ensemble 1 dans la première boucle “while” , puis dans la seconde boucle, on compare toute les valeurs de l’ensemble 2 avec l’ensemble 1. Si l’égalité est vérifiée, la deuxième boucle “while” est arrêté après avoir stocké la valeur dans “res” , le nouvel ensemble. Sinon, on vérifie ne pas être arrivé à la fin de la deuxième structure avant de déplacer le pointeur de l’ensemble à l’élément suivant. Ainsi de suite jusqu’à être arrivé à la fin du second ensemble. Si lors de la recherche, l’élément de l’ensemble 1 comparé devient plus petit qu’un élément de l’ensemble 2, alors on stop la comparaison et on passe à l’élément suivant. On réinitialise le pointeur de support du deuxième ensemble afin de redémarrer au début de l’ensemble.
    On retourne le pointeur sur l’ensemble d’intersection.

    c. Complexité des opérations
        i. Si le nombre à insérer doit être placé tout à la fin de l’ensemble, la complexité au pire cas est O(n).
        ii. Si le nombre à trouver est tout à la fin de l’ensemble, la complexité au pire cas est O(n).
        iii. Si l’ensemble 1 et l’ensemble 2 contient tout deux les mêmes éléments la complexité au pire cas est O(m).

2. Arbre binaire de recherche

a. Nous avons choisi d'implémenter l'arbre avec une structure pour chaque noeud : elle contient sa valeur (mot en chaîne de caractères), un pointeur vers l'arbre gauche et un pointeur vers l'arbre droit, ces deux arbres étant de la même structure. Cette implémentation permet d'utiliser la récursivité et donc la simplicité et lisibilité du code pour les fonctions d'insertion et de recherche.

b. Pour la fonction isBalanced, nous avons choisi d'utiliser des fonctions auxiliaires nous indiquant la différence d'un noeud, selon les hauteurs de ses fils droit et gauche. Si la différence est comprise entre -1 et 1, isBalanced renvoie vrai, sinon faux. La différence d'un noeud est la hauteur de son fils droit moins celle de son fils gauche.
Pour getAverageDepth, on fait appelle à une fonction auxiliaire "getAverageDepthAux" dans un premier temps pour pouvoir récupérer la somme des hauteurs de chaque noeud. On calcule par récursion la somme des sommets du fils gauche et du fils droit. On divise ensuite cette somme dans la fonction getAverageDepth par le nombre d'éléments dans l'arbre obtenu à l'aide de la fonction getNumberString.

c. Etude de la complexité de findCoocurrences :
   i. Dans le cas d'un arbre non équilibré, la complexité au pire est O(n). Pour l'arbre équilibré, la complexité est de O(log(n)).
   ii. La complexité de recherche de mots dans un arbre non trié est O(n) pour le pire cas.
   iii. Pour un arbre de recherche équilibré, la complexité est de O(log(n)).