Martin CREUZE
Lilian VOLTIGEUR

NF16 TP4

Compte rendu

Objectif : Utiliser les arbres binaires de recherche pour implémenter un exemple d'indexation et de recherche sur un fichier contenant un texte quelconque. Chaque nœud de l'arbre contient un mot, ainsi que la liste de ses positions dans le texte

Tout d'abord, nous avons créé les fonctions liées à la création de la liste des positions d'un mot. La fonction **t_ListePositions* creer_liste_positions()** crée cette liste et correspond à une suite d'instructions élémentaires pour l'allocation dynamique de la mémoire et l'initialisation des paramètres de la structure à 0. C'est donc une complexité en O(1). Ensuite, on ajoute une position dans la liste avec la fonction **int ajouter_position(t_ListePositions *listeP, int ligne, int ordre, int num_phrase)**. Elle l'ajoute dans la liste d'abord selon son numéro de ligne puis selon son ordre. Nous avons ajouté la fonction **void ajoutertete(t_ListePositions *listeP, int ligne, int ordre, int num_phrase)** pour simplifier l'écriture dans la fonction. (la complexité d'ajoutertete est O(1) et la fonction **void afficheliste(t_ListePositions *p)** pour pouvoir afficher cette liste de position. Elles nous ont permis de vérifier que les fonctions précédentes marchent correctement. Mais dans le problème du TP, nous sommes jamais dans un cas où on ajoute au début ou en milieu de liste étant donné qu'on incrémente à chaque fois la position pour chaque mot, on l'ajoute toujours en fin de liste. La complexité d'ajouter_position est de O(n), on ajoute la nouvelle position à la fin en parcourant toute la liste. n correspond ici à la longueur de la liste, ce qui dans le TP équivaut au nombre d'occurence du mot.

Ensuite, pour la Q9 et la Q10, nous avons décidé de rajouter/modifier les structures ci-dessous. L'index a maintenant un attribut de type liste de listes qui contient la liste des phrases, eux même étant une liste de mots :

La liste de mots:

```
typedef struct Pmot
{
   char* mot;
   struct Pmot* suivant;
}Pmot;
```

La liste de phrases:

```
typedef struct Phrase
{
   int numero_phrase;
   Pmot* premier_mot;
   struct Phrase* phrase_suivante;
}Phrase;

typedef Phrase* t_Phrase;
```

L'index modifié :

```
typedef struct t_index
{
   t_Noeud* racine;
   t_Phrase liste_phrases;
   int nb_mots_differents;
   int nb_mots_total;
}t_Index
```

La fonction **t_Index* creer_index()** crée un index. Elle est faite avec une suite d'instructions élémentaires pour l'allocation dynamique de la mémoire et l'initialisation des paramètres de la structure à 0 soit une complexité en O(1). La fonction **t_Noeud* rechercher_mot(t_Index *, char *)** effectue une recherche dans l'ABR de l'index en partant

de la racine et en descendant soit à gauche soit à droite de l'arbre pour, dans le pire des cas, arriver à une feuille. Soit une complexité en O(h) (h étant la hauteur de l'arbre). Mais avant tout, il faut ajouter des noeuds avec **int ajouter_noeud(t_Index*, t_Noeud*)**. On recherche le mot (même fonctionnement que pour la fonction de recherche). S'il est présent, on ajoute l'occurrence du mot avec ajouter_position(). Sinon, on utilise la fonction **t_Noeud* creer_noeud()** de complexité O(1) pour en créer un et on l'ajoute à l'emplacement adapté. Dans le pire des cas, on ajoute le nœud au niveau des feuilles soit une complexité en O(h). Pour qu'on se déplace dans l'ABR sans faire attention aux majuscules des mots, nous avons créé la fonction **char* min(char*)** qui renvoie la chaîne de caractère mis en paramètre en minuscule. Les comparaisons indiquant si on descend vers le fils gauche ou droit se font sur les chaînes en minuscule. Soit une complexité en O(nbre_caractere)

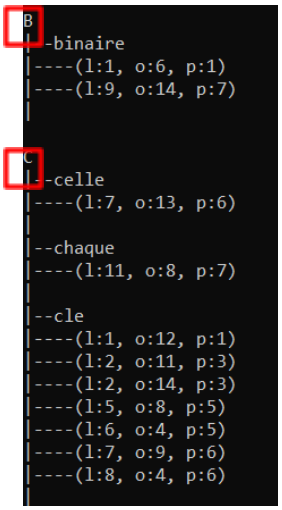
Nous avons vu précédemment que nous avions modifié la structure de l'index, lui attribuant une liste de liste. Nous avons donc également dû ajouter les fonctions **t_Phrase creer_phrase() - Pmot* creer_pmot() - void ajouter_Pmot (t_Phrase phrase, char* str) - void ajouter_phrase(t_Phrase phrase)** qui nous seront utile lors de l'indexation.

En effet, dans **int indexer_fichier(t_Index *, char *)**, on lit chaque caractère du fichier mis en paramètre. Dès qu'on a une lettre, on l'ajoute à une chaîne de caractères. On crée pour chaque mot un nœud avec creernoeud(), on lui attribue cette chaine de caractère, et sa position avec ajouter_position() (qui est en O(nbre_occurence)) et on ajoute également ce mot dans la liste de liste avec **creer_pmot()** et **ajouter_Pmot()**, ajouté dans la phrase initialement créée avec **creer_phrase()**. Dès qu'on lit autre chose qu'une lettre, on change les caractéristiques de position et dans le cas d'un point, on crée une phrase suivante avec **ajouter_phrase()**.

Sachant que **creer_phrase()** et **creer_pmot()** sont en O(1) : suite d'instructions élémentaires pour l'allocation dynamique de la mémoire et l'initialisation des paramètres de la structure à 0 ; et que **ajouter_Pmot()** est en O(nbre_mot) et **ajouter_phrase()** en O(nbre_phrase) : elles ajoutent le mot ou la phrase toujours en fin de liste, on a donc **indexer_fichier()** de complexité O(taille du fichier).

La deuxième fonctionnalité du menu nous demande d'indiquer si cet arbre est équilibré ou non. Pour ce faire, nous avons ajouté les fonctions **int maximum(int, int)**, **int hauteur(t_Noeud *)**, et **int estequilibre(t_Noeud *)**. Maximum() est de complexité O(1), il renvoie le plus grand nombre entre deux entiers mis en paramètre. hauteur() est en O(h) et renvoie la hauteur de l'arbre et estquilibre() vérifie l'équilibre pour chaque nœud. Il appelle donc hauteur() pour son fils droit et son fils gauche().
Complexité : En partant de la racine, la fonction **int estequilibre(t_Noeud *)** nécessite la fonction **int hauteur(t_Noeud *)** (O(h)) et appelle récursivement **int estequilibre(t_Noeud *)** pour les fils de la racine (hauteur h-1). le nombre d'appels récursifs est d'ordre de o(n) et chaque appel requiert **int hauteur(t_Noeud *).**
On a finalement une complexité d'ordre de O(somme (pour i allant de 1 à h) de n x h-i) n étant le nb de nœuds sur l'étage i. Dans le pire des cas, l'arbre est un arbre binaire complet. Au premier niveau, il y a 1 noeud, au second 2, puis 4 … au dernier 2^h. La somme vaudrait donc 2x(h-1)+ 4x(h-2) + ….. + 2^(h-1)x 1 donc une complexité de l'ordre de 2^h

Pour l'affichage de l'index, nous avons décidé de faire un parcours infixe de l'ABR en version récursive. Pour chaque nœud, on affiche le mot affecté et la liste de ses positions avec **afficheliste()** créée dans la première partie, de complexité O(nbre_occurences_mot). On parcourt tous les nœuds de l'ABR soit une complexité de O(n).



Nous avons également rajouté un **abécédaire** pour permettre de présenter un affichage comme celui demandé dans le TP. L'abécédaire permet de n'afficher qu'une fois la première lettre (encadrée en rouge) des mots qui commencent tous par la même lettre.

L'abécédaire est une chaîne de caractères qui contient la première lettre des mots rencontrés en début d'affichage. Lorsqu'on affiche un mot, on verifie d'abord que sa première lettre n'est pas dans l'abécédaire. Si elle ne l'est pas, on la rajoute et on affiche la première lettre et le mot. Sinon, on affiche uniquement le mot sans la première lettre (déjà affichée auparavant).

La 5eme fonctionnalité du menu demande d'afficher le mot avec le maximum d'apparitions. Nous avons ajouté la fonction **int trouver_max_apparition(t_Index *index, t_Noeud* noeudmax)** qui est un parcours infixe récursif de l'arbre soit une complexité de O(n) et qui renvoie le noeud du mot avec nombre d'occurrences maximum. Ce noeud est ensuite exploité dans **void afficher_max_apparition(t_Index *index)** pour afficher le mot ayant un nombre maximal d'occurrences, aussi en O(n) par conséquent.

Ensuite, la 6ème fonctionnalité demandait d'afficher les phrases où le mot saisi apparaissait. **Void afficher_occurences_mot(t_Index *, char *)** fait appel à la fonction **rechercher()** pour connaître toutes les occurrences du mot en question et ses positions. Il suffisait donc d'afficher sa ligne, son ordre mais également la phrase où le mot apparaît. Dans la liste de liste, les phrases sont numérotées, on affiche les phrases qui coïncident avec les positions du mot. Nous avons créé **void afficher_i_ligne(t_Phrase , int)** pour afficher la ième phrase du texte.
Dans afficher_i_ligne(), tant qu'on est pas la i ème phrase, on passe à la suivante. Puis on affiche tous les mots de la phrase. Soit une complexité de O(i+nombre_mot_phrase_i). Dans afficher_occurrences() on affiche la phrase pour chaque position soit O(nombre_occurences x (i+nombre_mot_phrase_i))

Enfin, la dernière fonctionnalité nous demandait de construire le texte sur un nouveau fichier. **Void construire_texte(t_Index *, char *)** crée un nouveau fichier, prend mot par mot de la liste de liste et l'écrit dedans avec un saut de ligne en fin de phrase. Il fallait également afficher le texte : pendant le parcours mot par mot, nous les affichons également.
On affiche autant de mot qu'il y en a dans notre liste de liste. Ce nombre de mots dépend du fichier indexé dans la fonctionnalité, soit une complexité en O(nb_mot(fichier_1)).