Alexis MATHEY
Célestin VALLOIS

Introduction

Objectifs

Il nous a été demandé d'écrire un programme utilisant les propriétés des arbres binaires de recherche pour implémenter un exemple d'indexation et de recherche sur un fichier contenant un texte quelconque.

Dans ce programme, l'arbre binaire de cherche contiendra tous les mots présents dans le texte à indexer. Chaque nœud de l'arbre contiendra un mot ainsi qu'une liste des positions du mot dans le texte. Une position contiendra un numéro de ligne, un ordre de ligne et un numéro de phrase.

Notre programme devra pouvoir créer un ABR, charger un fichier dans cet ABR, afficher les caractéristiques de l'ABR, afficher tous les mots distincts par ordre alphabétique, recherche un mot et enfin afficher les phrases contenant deux mots entrés par l'utilisateur.

Réalisation

L'organisation de notre projet codé en C est la suivante :

- Trois fichiers d'entête arbre.h, liste.h et outils.h contenant les déclarations des structures et fonctions de bases ;

- Trois fichier source arbre.c, liste.c et outils.c contenant la définition de chaque fonction ;

- Un fichier source main.c contenant le programme principal.

Problèmes rencontrés

Au cours de notre TP, nous n'avons pas rencontré de problème majeur/spécifique. Il a fallu essayer de prendre en compte tous les cas possibles pour chaque fonction. Comme par exemple les espaces et la casse des données dans le texte. Nous avons alloué dynamiquement toute la mémoire utilisée qui est libérée lorsqu'on quitte le programme.

Nos choix de programmation

Les structures de données

Dans nos fichiers arbre.h, liste.h et outils.h, nous avons déclaré 4 types à partir de 4 structures de données que nous avons créés :

- **Position** qui contient les caractéristiques d'une position : 3 entiers indiquant le numéro de ligne, l'ordre et le numéro de phrase. Ainsi qu'un pointeur vers la position suivante.

- **ListePosition** qui pointe vers une position de début et qui possède un entier indiquant le nombre de positions qu'elle possède.

- **NoeudABR** qui reprend sa propre structure : ce type contient un pointeur vers 2 structures nœud : le fils gauche et le droit. Il possède aussi la clé du nœud qui est un mot (char*) et une liste de position.

- **ArbreBR** qui est les caractéristiques de l'arbreBR : son nœud racine, son nombre de mots total et différents.

Ce type de structure implique évidemment une gestion dynamique de la mémoire à l'aide des fonctions **malloc()** lors de la création d'un élément et **free()** lors de la suppression.

Les algorithmes utilisés

Voici un rappel des algorithmes vus en cours et que nous avons réutilisés dans notre programme.

ABR_Insérer (T : arbre, z : noeud)
/ int ajouter_noeud(ArbreBR *arbre, NoeudABR *noeud) */*

```
y := nil
x := racine[T]
Tant_que x <> nil faire
        y := x
        Si clé[z] < clé[x]
                alors x := gauche[x]
        sinon x := droit[x]
Pere[z] := y
Si y = nil alors racine[T] := z
sinon si clé[z] < clé[y]
        alors gauche[y] := z
sinon droit[y] := z
```

⇒ Nous ajoutons à cet algorithme une condition si le mot est déjà dans l'ABR et dans ce cas, nous faisons appel à la fonction ajouter_position() qui ajoute une nouvelle position à la liste de positions du nœud concerné.

ABR_Rechercher_Itératif (x : noeud, k : mot)
/ NoeudABR* rechercher_noeud(ArbreBR *arbre, char *mot) */*

```
Tant_que x <> nil et k <> clé[x] faire
        Si k < clé[x] alors x := gauche[x]
                sinon x := droit[x]
Retourner (x)
```

Inserer_ trier_liste (L : liste , z : élément de L)
/ int ajouter_position(ListePosition* listeP, int ligne, int ordre, int num_phrase) */*

```
x := tete[L]
si (cle[z] < cle[x]) ou (x = NIL)
        alors Inserer_tete_liste( L, z)
sinon
        Tant que succ[x] ≠ NIL et cle[z] > cle[succ[x]]
                faire x := succ[x]
        succ[z] := succ[x]
        succ[x] := z
```

Est_equilibre (x : noeud)
/ int est_equilibre(NoeudABR* nœud) */*

Retourner $|\text{hauteur}(\text{droit}(x)) - \text{hauteur}(\text{gauche}(x))| \leq 1$

Recherche_liste (L : liste, k : clé)
/ Utilisé dans void afficher_position(NoeudABR* nœud) */*

```
x := tete[L]
Tant que x  ≠ NIL et cle[x] ≠ k
                faire x := succ[x]
Retourner(x)
```

Hauteur_noeud (x : nœud)
/ int hauteur (NoeudABR * nœud) */*

```
si (x = NIL) alors retourner -1
sinon retourner 1+max(hauteur(gauche[x]), hauteur(droit[x]))
```

Parcours_Infixe (x : noeud)
/* void afficher_noeud (NoeudABR* x) */

–Si x <> nil alors
Parcours_Infixe(gauche[x])
Afficher clé[x]
Parcours_Infixe(droit[x])

Les fonctions

Le menu

Pour accéder aux fonctions, nous avons créé un menu proposant les choix suivants :

- Créer un arbre binaire de recherche (**creer_abr()**)
- Charger un fichier dans l'ABR (**charger_fichier ()**)
- Afficher les caractéristiques de l'ABR (**est_equilibre(),hauteur_arbre()** et **structure de l'arbre**)
- Afficher tous les mots distincts de l'ABR par ordre alphabétique (**afficher_arbre ()**)
- Rechercher un mot dans l'ABR (**rechercher_noeud ()**)
- Afficher les phrases contenant les 2 mots saisis (**afficher_phrases ()**)
- Quitter (**free_arbre()**)

Ce menu s'affiche en boucle tant que l'utilisateur ne sélectionne pas l'option « Quitter ».

Création de l'ABR - creer_ABR()

Cette fonction crée un ABR vide (donc *nb_mots_differents=0*, *nb_mots_total=0* et *racine=NULL*) et renvoie l'adresse de cet ABR (un pointeur sur le type **ArbreBR**) ou bien NULL si la fonction a échoué.

L'ABR crée est alloué dynamiquement à l'aide de la fonction **malloc()**. Nous testons ensuite si l'allocation a bien marché à l'aide d'un IF, si ce n'est pas bon on renvoie NULL sinon on renvoie l'adresse de l'arbre créé. Tout ceci se fait en temps constant.

Complexité : *0(1)*

Affichage des caractériqtiques de l'ABR - est_equilibre(),hauteur_arbre()

Cette fonction affiche le nombre de mots (différents et total) dans l'ABR, sa hauteur et s'il est équilibré.

Pour déterminer l'équilibre de l'arbre, nous utilisons la fonction récursive **hauteur()**, qui retourne la hauteur du nœud passé en paramètre et qui fonctionne selon l'algorithme que nous avons vu en cours qui est de complexité *O(n).*

Complexité : 0(n) et 0(h)

L'affichage du nombre de mots dans l'arbre est ensuite direct car il est implémenter dans la structure AbreBR.

Affichage des mots de l'ABR - afficher_arbre()

Cette fonction affiche tous les mots de l'arbre dans l'ordre alphabétique. Elle fait appel à la fonction **afficher_noeuds()** qui effectue un parcours infixe de l'arbre à partir du nœud passé en paramètre.

Nous parcourons tous les nœuds de l'arbre donc : (avec n le nombre total de nœuds)

Complexité : 0(n)

Chargement d'un fichier dans l'ABR - charger_fichier()

Cette fonction permet de lire un fichier dont le nom est saisi par l'utilisateur et de placer tous les mots qu'il contient dans l'ABR (c'est-à-dire, classés par ordre alphabétique avec leurs positions renseignées dans une **ListePosition**). Pour cela nous lisons le fichier caractère par caractère et stockons les caractères dans une string et dès que nous rencontrons un caractère non alphanumérique (c'est-à-dire la fin du mot actuel) nous ajoutons le mot lu dans l'ABR avec la fonction **ajouter_nœud()**. En même temps nous incrémentons les compteurs d'ordre des mots, de numéro de ligne et de numéro de phrase.

Nous parcourons tout le fichier caractère par caractère donc nous répétons la boucle principale n fois (avec n la taille du fichier) et dès qu'un mot est formé nous faisons appel à **ajouter_noeud()** qui est en *O(h)* (avec h la hauteur de l'ABR qui augmente au fur et à mesure de l'ajout de mots).

Complexité : 0(n) où n : nb de caractères du fichier

Recherche d'un noeud – rechercher_noeud()

Cette fonction recherche un nœud dans l'ABR et retourne un pointeur vers le nœud s'il est présent dans l'arbre et NULL sinon. Nous utilisons le modèle de l'algorithme ABR_Rechercher_Itératif() vu en cours. Ce qui nous donne :

Complexité : *0(h)*

Affichage des phrases contenant 2 mots recherchés - afficher_phrases()

Cette fonction recherche dans l'ABR toutes les phrases dans lesquelles se trouvent deux mots saisis en paramètres. Nous faisons donc appel à la fonction **rechercher_noeud()** pour chacun des mots et si les deux mots sont présents dans l'arbre, nous cherchons s'ils sont présents dans une (ou des) même(s) phrase(s) en parcourant leur liste de positions et nous affichons les phrases concernées avec la fonction **afficher_phrase_occurrence()**.

Rechercher_noeud() est en *O(h)*. Dans le meilleur des cas, un des mots n'est pas présent dans le texte donc nous ne faisons que la recherche et donc *Ω(h).* Dans le pire des cas, nous parcourons ensuite les deux listes de postions (de longueurs p et q) jusqu'à la fin de la plus petite donc nous effectuons la boucle *min(p, q)* fois. Et à chaque itération de cette boucle nous affichons une phrase avec **afficher_phrase_occurrence()** qui est de complexité en *O(t)* avec t la taille du fichier contenant le texte.

*Complexité : Ω(h) et O(h+min(p,q) * t)*

Fonctions supplémentaires (dans outils.c)

StringToLower()

Cette fonction permet de rendre notre programme insensible à la casse car elle transforme la chaîne de caractères passée en paramètre afin qu'elle ne contienne plus que des minuscules.

Nous faisons appel à cette fonction lors de l'ajout de mot dans l'ABR ou lors de la recherche d'un mot pour standardiser la casse de tout notre ABR et ainsi rendre le programme plus robuste (on peut taper « BonJoUR » pour trouver les occurrences de « bonjour »).

Conclusion : suggestions, améliorations

Pour conclure, ce TP nous aura permis de découvrir la gestion de fichier dans un programme. On aura plu manipuler les notions vu sur les ABR et le équilibres en cours avec un exemple concret et voir la complexité, sans mauvais jeu de mots, d'optimiser la complexité de nos fonctions.
Nous n'avons malheureusement pas eu le temps de faire le Bonus qui permettait d'équilibré l'arbre.
Pour conclure, nous sommes sûrs que notre programme n'est pas optimisé à 100% vis-à-vis de la compléxité des fonctions malgré les efforts et le temps que nous y avons passé.

PS : Vraiment désolé pour le retard. Nous pensions que nous avions jusqu'à lundi soir minuit et non dimanche soir minuit pour rendre le projet. On aurait dû vérifier la date et ne pas faire cette erreur. Nous inspirons à devenir ingénieur, nous savons que nous aurons des deadlines à tenir et que nous devrions les rendre en temps et en heure. Nous ne reproduirons pas cette erreur.