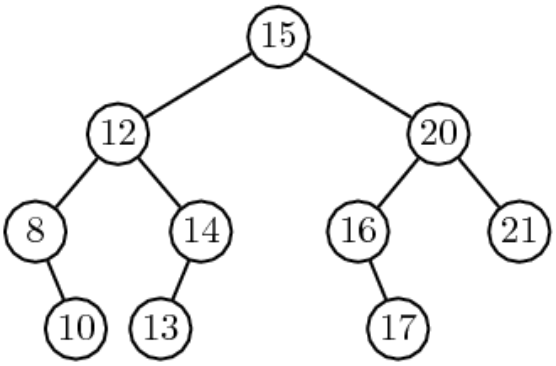
09/01/2015

TP n°4 : Arbres Binaires de Recherche

[NF16]



Lefebvre Clémence & Laviolette Etienne

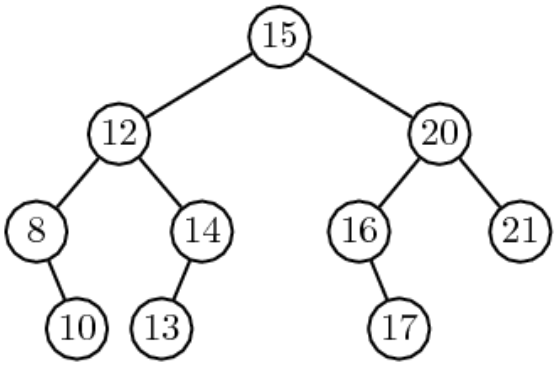
Table des matières :

Introduction :

Lors de ce dernier TP qui nous a été proposé en NF16, on nous a imposé l'utilisation d'une structure de données fondamentale essentielle : l'arbre binaire de recherche. Après avoir développé notre TP numéro 3 à l'aide de listes chainées, nous voici ici confronté à la mise en pratique de cette notion importante de deuxième partie de semestre de l'UV d'algorithmique et structure de données.

L'intérêt primaire qui réside dans l'utilisation de cette structure est tout d'abord de se familiariser avec. En effet, pour chacun d'entre nous, nous n'avions jamais programmé en langage C une structure d'une telle importance. Guidés par les instructions contenues dans l'énoncé de ce TP nous avons essayé d'exploiter l'intérêt primordial des ABR : la rapidité de recherche d'un élément.

En effet, comme il est visible sur l'exemple ci-dessous issu de l'ABR présenté en page de garde : tous les éléments à gauche d'un nœud ont une clé strictement plus petite que leur père. A contrario, ceux situés à droite ont une clé strictement plus grande.



L'application qui nous est présenté dans le TP ci présent est la suivante : nous allons mettre en place un programme capable d'analyser, grâce à un ABR, un fichier texte simplifié. Ainsi, nous pourrons mettre en place des fonctions d'études de ce texte par l'intermédiaire de l'application des algorithmes sur les ABR vus lors du cours de NF16.

La mise en pratique des fonctions de traitement de fichier en langage C, l'application, une fois de plus, des pointeurs, des structures ainsi que de l'allocation dynamique constituent les thématiques sous-jacentes à ce dernier TP du semestre.

I. Mise en place des structures :

Cette première étape, sans doute la plus importante, a été réalisée en accord avec l'énoncé de TP. Nous avons spécifié notre problème et implanté les structures nécessaires à notre problème.

- Nous avons d'abord introduit la structure arbre. Structure constituée de 3 champs différents : un premier étant un pointeur vers le nœud appelé nœud racine de l'ABR. Nous ajoutons un champ contenant le nombre de mots différents puis un dernier contenant le nombre de mot total de cet ABR.

```
struct arbreBR {
    NoeudABR* racine;
    int nb_mots_differents;
    int nb_mots_total;
};
typedef struct arbreBR ArbreBr;
```

```
struct noeudABR {
    char* mot;
    ListePosition* positions;
    struct noeudABR* filsGauche;
    struct noeudABR* filsDroit;
};
typedef struct noeudABR NoeudABR;
```

- Un nœud de notre ABR étant défini comme suit : un mot, une liste des positions de ce mot, un fils gauche qui est un pointeur vers un nœud et enfin un fils droit. Nous constatons ici que l'énonce nous propose de coupler les acquis du TP numéro 3 sur les listes chaînées avec nos récentes connaissances sur les ABR pour créer des structure encore plus importantes.

- La liste est position qui est mentionnée dans le nœud défini ci-dessus est une structure contenant un pointeur vers une première position et un nombre d'élément de cette structure. On s'aperçoit que cette liste de positions comme son nom l'évoque, nous permettra d'utiliser nombreux algorithmes de listes chainées.

```
struct listePosition {
    Position* debut;
    int nb_elements;
};
typedef struct listePosition ListePosition;
```

- Enfin une position, élément de base de liste chainée des positions se décompose en plusieurs champs. Un numéro de ligne de cette position, un entier correspondant à l'ordre et un dernier entier correspondant au numéro de phrase de cette position de mot. Bien sur le dernier champ constituant le chainage est présent : un pointeur vers la position suivante.

```
typedef struct position{
    int numero_ligne;
    int ordre;
    int numero_phrase;
    struct position *suivant;
}Position;
```

Les structures qui nous sont présentées ici se suffisent à elles-mêmes. Nous n'avons pas ressenti le besoin d'en ajouter. Les deux premières relatives strictement à la structure d'arbre sont définies

dans le fichier d'en-tête nommé : *arbre.h*. Quant aux deux dernières, qui sont relatives aux listes ont été placées dans le fichier *liste.h.*

II. Fonctions développées et complexité

Afin de manipuler les structures présentées précédemment, nous avons été guidées quant au développement des fonctions de services nécessaires qui sont les suivantes :

- Créer_liste_positions :

```
ListePosition *creer_liste_positions(){
    // Fonction qui crée une liste de positions vide (renvoie NULL si echec)

    ListePosition* new_liste = malloc(sizeof (ListePosition));
    if(new_liste){

        new_liste->nb_elements=0;
        new_liste->debut=NULL;
        return new_liste;
    }
    else{
        perror("malloc");
        return NULL;
    }
}
```

But: Créer une nouvelle liste de positions Creer_liste_position()

Valeur retournée: NULL ou le pointeur vers la liste de positions créée

Paramètre : Aucun

Complexité: O(1), Ω(1) :θ(1)

- Créer_position :

But: Créer une position

Valeur retournée: NULL ou le pointeur vers la position créée

Paramètre : Aucun

Complexité: O(1), Ω(1) : θ(1) (opérations triviales d'affectations)

```
Position* creer_position(){
    // Fonction qui crée une liste de positions vide (renvoie NULL si echec)

    Position * new_position = (Position*) malloc(sizeof(Position));

    if(new_position){

        new_position->numero_ligne=0;
        new_position->ordre=0;
        new_position->numero_phrase=0;
        new_position->suivant=NULL;
        return new_position;
    }
    else {
        perror("malloc");
        return NULL;

    }
}
```

Ces deux fonctions sont primordiales quant à l'appel de la suivante qui ajoute une position. Nous avons décidé de créer nous-même la seconde, qui n'était pas demandée pour simplifier le code.

- Ajouter_position :

```c
int ajouter_position(ListePosition *listeP, int ligne, int ordre, int num_phrase) {
    // Fonction qui ajoute un élément dans une liste de positions en respectant l'ordre du mot dans le texte
    Position* new_pos;              // Créer la position

    new_pos = creer_position();
    new_pos->numero_ligne = ligne;
    new_pos->ordre = ordre;
    new_pos->numero_phrase = num_phrase;
    Position* pos_temp = listeP->debut;  // L'insérer dans la liste

    if(pos_temp==NULL){             // Si liste vide
        listeP->debut=new_pos;
        new_pos->suivant=NULL;
        listeP->nb_elements++;
        return 1;
    }

    if((pos_temp->numero_ligne>ligne)&&(pos_temp->ordre>ordre)){   // Si inférieur à toutes les pos de la liste
        new_pos->suivant=listeP->debut;
        listeP->debut=new_pos;
        listeP->nb_elements++;
        return 1;
    }

    while ((pos_temp->suivant!=NULL)&&(pos_temp->numero_ligne<ligne)){ // Recherche de l'endroit où insérer
        pos_temp=pos_temp->suivant;
    }

    while ((pos_temp->suivant!=NULL)&&(pos_temp->ordre<ordre)){
        pos_temp=pos_temp->suivant;
    }
    new_pos->suivant = pos_temp->suivant;       // Insertion
    pos_temp->suivant=new_pos;
    listeP->nb_elements++;

    return 1;
```

But: Ajouter une position à une liste de position

Valeur retournée: 1 en cas de succès, 0 en cas d'échec

Paramètre : Un pointeur vers une liste de positions, le numéro de ligne de cette position ainsi que son ordre et son numéro de phrase

Complexité: 0(n) : où n est le nombre de positions de la liste. En complexité on parcourt dans le pire des cas la liste entière. Dans le meilleur 0(1) : on ajoute en première position. Pas de θ possible.

Nous avons donc nos trois fonctions associées aux structures et aux manipulations de listes chainées. Elles sont toutes les trois stockées dans le fichier liste.c.

Voici les déclarations et commentaires des fonctions du fichier arbre.c manipulant l'arbre grâce à ses structures et les fonctions de services sur les listes vues précédemment.

- Créer_abr :

```
ArbreBr *creer_abr(){
    // Fonction qui crée un ABR vide (renvoie NULL en cas d'échec)
    ArbreBr* new_arbre = malloc(sizeof (ArbreBr));
    if(new_arbre){

        new_arbre->racine = NULL;
        new_arbre->nb_mots_differents = 0;
        new_arbre->nb_mots_total = 0;
        return new_arbre;
    }
    else{
        perror("malloc");    //Commentaire de l'échec du malloc
        return NULL;
    }

}
```

But: Créer d'un nouvel arbre

Valeur retournée: NULL ou le pointeur vers l'arbre crée,

Paramètre : Aucun

Complexité: O(1), Ω(1) :θ(1)

- Ajouter_mot (alternative à ajouter_noeud)

Le code de cette fonction étant long nous l'avons ajouté en annexe 1. En voici ses caractéristiques :

But: Ajouter un nœud (par ses caractéristiques) dans un arbre en respectant les règles d'insertions d'un ABR

Remarque : Nous avons corrigé le problème que nous avions rencontré lors de la présentation en TP et ainsi nous ne prenons plus en compte la casse.

Valeur retournée: 1 en cas de succès, 0 pour l'échec

Paramètre : Un pointeur vers un arbre et les caractéristiques du nœud à ajouter : le mot, la liste des positions et les deux fils.

Complexité: O(h) où h est la hauteur de l'arbre

Algorithme : ABR_Inserer (Pointeur vers arbre : T et Z : nœud à insérer)

y :=nil

x := racine[T]

TantQue x <> nil faire

y :=x

Si cle[z] < cle [x] alors x := gauche[x]

Sinon x := droit[x]

Pere[z] := y

Si y = NIL alors racine[T] := z

Sinon Si cle[z] < cle[y] alors gauche[y] := z

Sinon droit[y] :=z

- Charger_fichier :

  Cette fonction est disponible en annexe 2. En effet vu sa longueur nous avons préféré l'ajouter à la fin plutôt que de nombreuses pages de codes ici. De plus cette fonction a été la fonction centrale de notre programme :

But : Nous chargeons un fichier texte, que nous analysons par la même occasion : c'est-à-dire que nous créons l'arbre associé (il faut que l'ABR ait été créé avant..).

Il faut toujours appeler son fichier .txt « fichier_test.txt » pour l'analyser, nous n'avons pas prévu d'interface pour récupérer son nom. Sinon il faut modifier le fopen de cette fonction

Valeur retournée: 1 en cas de succès, 0 pour l'échec

Paramètre : Un pointeur vers un arbre et le pointeur

Complexité : Parcourt le fichier une première fois ligne par ligne, caractère par caractère : O(l * c).

Grâce à ce premier parcourt on va compter le nombre de mot par ligne. Les compteurs incrémentés nous permettrons lors du second parcours, de savoir exactement dans quelle ligne et à quel ordre de chaque ligne nous sommes sur le nœud traité à l'instant t .

Remarque : Nous aurions pu gérer à l'aide d'un fgetc ce problème et ne parcourir qu'une fois le fichier

Puis une seconde fois mot par mot O(m) * l'appel de ajouter_mot qui est de complexité O(h).

Donc au total une fonction couteuse de complexite : O(l*c) + O(m*h)

- Rechercher_noeud :

```
NoeudABR *rechercher_noeud(ArbreBr *arbre, char *mot){
    // Fonction de service pour rechercher un noeud de l'arbre par le mot qu'il contient

    NoeudABR * noeud_temp= arbre->racine;

    while((noeud_temp!=NULL)&&(strcmp(mot, noeud_temp->mot)!=0)){    //Parcourt l'arbre selon l'ordre alphabétique
        if(strcmp(mot, noeud_temp->mot)<0){                           // des mots des noeuds recontrés
            noeud_temp=noeud_temp->filsGauche;
        }
        else{
            noeud_temp=noeud_temp->filsDroit;
        }

    }
    return noeud_temp;
}
```

But: Rechercher le nœud contenant le mot passé en paramètre

Valeur retournée: un pointeur vers le nœud recherché

Paramètre : un pointeur vers l'arbre ainsi qu'une chaine de caractère contenant le mot à rechercher

Complexité: 0(h) : où h est la hauteur de l'arbre. En effet le pire cas est de rechercher un mot contenu dans la feuille de profondeur la plus importante. C'est-à-dire parcourir toute la hauteur de l'arbre

Ω(1) si l'on recherche le mot « racine ». Donc pas de θ exactement.

Algorithme : ABR_Recherche_itératif (X : nœud racine ABR, k : cle rechechée)

TantQue X <> NIL et k<>cle[X] Faire

Si k<cle[X] Alors x := gauche[X]

Sinon x := droit[X]

- Afficher_mot :

```
void afficher_mot (NoeudABR* noeud){
    //Fonction de service pour afficher un noeud et non un arbre

    NoeudABR * mot_temp= noeud;
    if(mot_temp!=NULL){
        afficher_mot(mot_temp->filsGauche);
        printf("%s\n", mot_temp->mot);
        afficher_mot(mot_temp->filsDroit);
    }
}
```

But: Afficher le mots de l'arbre par parcourt infixe.

Valeur retournée: rien

Paramètre : Un pointeur vers un noeud

Complexité: O(n) où n est le nombre de nœuds. K appels pour la procédure récursive afficher_mot pour le fils gauche. Puis n-k -1 pour le fils droit. Donc de l'autre de n.

Algorithme : Parcours_infixe (T : ABR, racine r)

Si r est différent de NIL

Parcours_infixe (arbre de racine fils_gauche[r]

Afficher cle[r]

Parcours_infixe (arbre de racine fils_droit[r]

FniSi

- Afficher_arbre :

```
void afficher_arbre(ArbreBr* arbre){
    // Fonction qui affiche les mots classés par ordre alphabétique
    // Par parcours infixe grâce à l'appel de afficher_mot

    NoeudABR * mot_temp= arbre->racine;
    afficher_mot(mot_temp);

}
```

But: afficher les nœuds d'un arbre

Valeur retournée : rien

Paramètre : Pointeur vers racine

Complexité: La même que la fonction afficher_mot ci-dessus puisqu'elle ne fait qu'une affectation et un appel de cette fonction.

III. Fonctions tierces utilisées dans le menu :

Les fonctions décrites ici sont appelées par les « case » du menu qui demande d'aller encore plus loin dans l'analyse de notre arbre. Elles sont stockées dans le fichier arbre.c

- Profondeur_arbre :

```
int* ProfondeurArbre (NoeudABR* racine, int prof, int* profMax){ //Calcul la profondeur de l'arbre

    if (racine!=NULL){
    prof++;
    ProfondeurArbre(racine->filsGauche,prof, profMax);
    if(prof>*profMax){
        *profMax=prof;
    }
    ProfondeurArbre(racine->filsDroit,prof, profMax);
    if(prof>*profMax){
        *profMax=prof;
    }
    prof--;
    }
    return profMax;

}
```

But: Calculer la profondeur de l'arbre utilisé

Valeur retournée: la profondeur en question

Paramètres : 3 paramètres : le nœud dont on cherche la profondeur, la profondeur de ce nœud et la profondeur maximale de l'arbre rencontrée jusqu'à ce nœud.

Complexité: Cette fonction est récursive et provoque un appel sur chaque nœud de l'arbre. Donc : O(j) avec j le nombre de nœuds

Algorithme :

si nœud <>NIL
    on incrémente profondeur
    appel recursif de la profondeur sur le fils gauche
    si la profondeur actuelle > profondeur max stockée en mémoire
        *profMax :=prof
    appel recursif de la profondeur sur le fils droit
    si la profondeur actuelle > profondeur max stockée en mémoire
        *profMax :=prof
    on sort d'un appel de la fonction sur un nœud (on a déjà la profondeur des fils gauches et droits, on veut maintenant remonter dans l'arbre) donc on décrémente prof

on retourne profMax

- Equilibre() :

```
int equilibre (ArbreBr* arb){   //Renvoie l'equilibre de l'arbre
//hauteur gauche
    int* profGauche=malloc(sizeof(int));
    *profGauche=0;
    profGauche= ProfondeurArbre( arb->racine->filsGauche, 0, profGauche);

//hauteur droit
    int* profDroit=malloc(sizeof(int));
    *profDroit=0;
    profDroit= ProfondeurArbre ( arb->racine->filsDroit, 0, profDroit);
//egal?

    if(*profGauche==*profDroit){
        free(profGauche);
        free(profDroit);
        return 1;
    }
    else{
        free(profGauche);
        free(profDroit);
        return 0;
        }
}
```

But: Calculer l 'équilibre de l'arbre utilisé. Pour cela on calcul la profondeur du fils gauche et du fils droit de la racine et on compare si elles sont égales.

Valeur retournée: Si l'arbre est équilibré ou pas

Paramètre : un pointeur vers l'arbre en question

Complexité: O(p + q) avec p le nombre de nœuds à gauche de la racine et q le nombre de nœuds à droite. →O(j-1) avec j le nombre de nœuds de l'arbre.

- ABR_minimum :

```
NoeudABR* ABR_Minimum (NoeudABR* x){ // Renvoie le noeud de cle minimum d'un arbre

    while(x->filsGauche!=NULL){
        x = x->filsGauche;
    }

    return x;
}
```

But: Retourner le minimum d'un arbre

Valeur retournée : Le pointeur vers le minimum

Paramètre : un pointeur vers un noeud

Complexité: O(h) car au pire des cas le nœud minimum est au plus profond de l'arbre

Algorithme : ABR_Minimum (pointeur vers un nœud : x)

TantQue gauche[x] <> NIL Faire

X := gauche[x]

Retourner(x)

- ABR_Pere :

But: Retourner le père d'un noeud
Valeur retournée: le pointeur vers le père
Paramètre : un pointeur vers un arbre et le pointeur vers le nœud dont on cherche le père
Complexité: O(h) au pire on cherche le père du nœud le plus profond

```
NoeudABR* Pere (ArbreBr* arb, NoeudABR* n){ // Renvoie le pere d'un noeud
    NoeudABR * noeud_temp= arb->racine;
    NoeudABR * noeud_pere=NULL;
      while((noeud_temp!=NULL)&&(strcmp(n->mot, noeud_temp->mot)!=0)){
          noeud_pere=noeud_temp;
            if(strcmp(n->mot, noeud_temp->mot)<0){
                noeud_temp=noeud_temp->filsGauche;
            }
            else{
                noeud_temp=noeud_temp->filsDroit;
            }

        }
        return noeud_pere;
}
```

- ABR_Successeur :

```
NoeudABR* Successeur(ArbreBr* arb, NoeudABR* n){ //Renvoie le successeur d'un noeud
    if (n->filsDroit!=NULL){
        n=ABR_Minimum(n->filsDroit);
        return n;
    }
    else{
        NoeudABR* noeud2=Pere(arb, n);
        while((noeud2!=NULL)&&(noeud2->filsDroit!=NULL)&&(strcmp(noeud2->filsDroit->mot,n->mot)==0)){
            n=noeud2;
            noeud2=Pere(arb, noeud2);
        }

        return noeud2;
    }
}
```

But: Retourner le successeur d'un noeud
Valeur retournée: le pointeur vers le successeur
Paramètre : un pointeur vers un arbre et le pointeur vers le nœud dont on cherche le successeur
Complexité: O(h) au pire on cherche le successeur du nœud le plus profond

Algorithme : ABR_Successeur(pointeur vers la racine : x)

```
Si droit[x] <> NIL retourner ABR_Minimum(droit[x])
y := père[x]
TantQue y <> nil et x = droit[y] faire
    x := y
    y : = père[y]
Retourner(y)
```

- Effacer_arbre() :

Le code de cette fonction qui nous sert dans le case 7 de notre menu est disponible en annexe 3.

But: Vider les nœuds d'un arbre de manière correcte (en vidant aussi les pointeurs qui sont contenus dans la structure nœud. Notamment les listes de positions). Nous allons donc : Parcourir l'arbre en postfixé puis pour chaque nœud vider la liste chainée des positions, le pointeur vers le début de cette liste, le mot du nœud, et le nœud lui-même. Un affichage est prévu pour donner le déroulement

Valeur retournée: rien
Paramètre : un pointeur vers le nœud racine d'un arbre

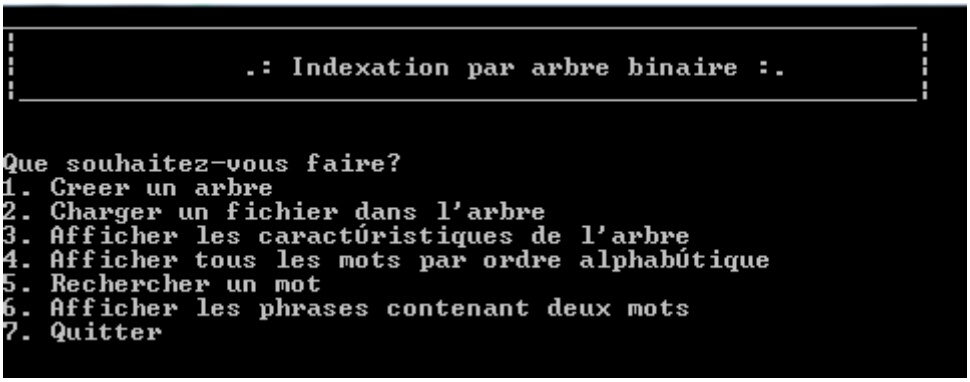
Complexité: O(n*liste). En effet on parcourt l'ensemble des nœuds puis on vide la liste chainée qui est associée. Liste est alors la longueur de la liste.

Algorithme de parcourt postfixé : Parcours_Postfixe(pointeur vers la racine d'un arbre : x)

```
Si x <> nil alors
    Parcours_Postfixe(gauche[x])
    Parcours_Postfixe(droit[x])
    Affiche cle[x]
```

IV. Limites et remarques :

Le main.c de notre projet fait simplement appel à outil.c qui contient le menu de notre programme :



- Nous aurions pu une nouvelle fois découper notre menu encore plus pour fournir une structure switch de sélection plus épurée et claire,

- La fonction de chargement de fichier est très couteuse en complexité. Elle a été codée en dure et une optimisation est sans doute possible,

- Nous n'avons pas eu le temps de coder le bonus en AVL,

- Pas assez de contrôles sont présents au niveau des interactions avec l'utilisateur,

- Un graphisme plus poussé et un menu amélioré aurait pu être réalisé

Enfin, nous avons aussi était contraint à quelques modifications après la présentation de notre travail durant la dernière heure de TP :

- En effet nous avons amélioré notre programme afin de ne pas prendre en compte la casse dans l'ordre lexicographique (c.f « case 6 de notre menu).
- Aussi un problème dans l'affichage des phrases contenant deux mots oubliait de citer une phrase (un parcours de la liste manquait).
- Quelques modifications d'affichage ont été réalisées pour rendre notre travail plus lisible
- Aussi nous avons décidé de ne traiter que les fichiers nommés « fichier_test.txt » afin d'éviter une modification du code au cas où on changerait le fichier à traiter. Seule une modification de son nom est nécessaire.

Conclusion :

Le développement de cet ultime TP de NF16 lors de ce semestre nous a permis d'appliquer nos connaissances sur les arbres, de coder « en dur », sur machine une représentation de ces arbres et de les manipuler. Nous avons pu consolider les bases acquises lors de ce semestre mais des progrès sont encore à réaliser sur la qualité sur code que ce soit au niveau de la complexité des fonctions (notamment spatiale) ou au niveau des indentations.

Les TP, en général, durant ce semestre de NF16 ont constitué un excellent exercice d'application du cours et des techniques abordées sur papier en TD. Nous avons pu aussi mettre chacun d'entre nous à l'épreuve du travail en binôme, ce qui fut une expérience enrichissante.

Annexes :

1. Code de ajouter_mot (alternative à ajouter_noeud)

```
int ajouterMot(ArbreBr* a, char* mot, int ligne, int ordre, int numphrase){
    // Chercher si le mot est déjà présent dans l'arbre (fonciton existante)
    NoeudABR* noeud_temp = rechercher_noeud(a, mot);


    if (noeud_temp != NULL){
        // Si oui, il existe ajouter position (fonction existante)
        if(ajouter_position(noeud_temp->positions, ligne, ordre, numphrase))
        return 1;
    }

    else{
        // Sinon créer nouveau noeud grâce à la fonction creer_noeud et incrementer nomre de mots differents

        a->nb_mots_differents = a->nb_mots_differents+1;
        noeud_temp= creer_noeud();

        strcpy(noeud_temp->mot,mot);  // Lui attribue le mot actuellement traité

        ajouter_position(noeud_temp->positions, ligne, ordre, numphrase);
        noeud_temp->filsGauche=NULL;
        noeud_temp->filsDroit=NULL;


        //l'insérer dans l'arbre
        if (a->racine==NULL){  //en tant que racine de l'arbre
            a->racine=noeud_temp;
            return 1;
        }

        else{
            //insérer dans arbre en tant que noeud different de la racine
            NoeudABR* noeud_arbre = a->racine;
            NoeudABR* noeud_y = NULL;
```

```
                while(noeud_arbre!=NULL){
                    noeud_y=noeud_arbre;
                    if(strcmp(noeud_temp->mot,noeud_arbre->mot)<0){
                        noeud_arbre=noeud_arbre->filsGauche;
                    }
                    else{
                        noeud_arbre=noeud_arbre->filsDroit;
                    }
                }

               // Pere[z]:=y?????

                if(noeud_y==NULL){
                    a->racine=noeud_temp;
                }
                else{
                    if(strcmp(noeud_temp->mot,noeud_y->mot)<0){
                        noeud_y->filsGauche=noeud_temp;
                    }
                    else{
                        noeud_y->filsDroit=noeud_temp;
                    }
                }

                //noeud_temp->filsGauche
                //noeud_temp->filsDroit
                return 1;
            }

        }

    }
```

2. Charger_fichier :

```
int charger_fichier(ArbreBr *arbre){

    FILE* fichier = NULL;
    int nb_mots=0;
    int ordre=1;
    int num_phrase=1;

    fichier = fopen("fichier_test.txt", "r"); // Toujours changer le nom du fichier text en fichier_test.txt pour le
     if (fichier != NULL)
        {

            char* mot=malloc(20*sizeof(char));
            char buff[1000];
            int i=0 ;
            int longueur_chaine;
            int compteur_mot = 0;
            int compteur_ligne=0;
            int tableau_mot[1000];
            int max_ligne;
            int numero_mot = 1;
            int numero_ligne = 1;
            int test_phrase=0;

            while (fgets(buff, 1000, fichier)!=NULL){ //Premier parcourt du fichier pour récupérer nombre de ligne et
                compteur_ligne++;                     // Boucle while qui est effectuée à chaque ligne récupérée par
                longueur_chaine = strlen(buff);

                for (i=0; i<longueur_chaine; i++){

                    if (buff[i]== ' '){
                        compteur_mot++;
                    }
                }

                compteur mot++;
                tableau_mot[compteur_ligne]= compteur_mot;  //Tableau stockant : en indice le numero de la ligne et le
                printf("\nLa ligne numero : %d contient %d mots\n", compteur_ligne, tableau_mot[compteur_ligne]);
                compteur_mot = 0;
                max_ligne = compteur_ligne;

            }
            rewind(fichier);  // On revient au début et on analyse mot par mot le fichier désormais

            while(fscanf(fichier, "%s",mot)==1){ // Analyse mot par mot du fichier

                longueur_chaine = strlen (mot);  // Récupère la longueur du mot

                if (mot[longueur_chaine - 1]=='.'){ // Vérifie si ce mot n'est pas le dernier du phrase. Dans ce cas i

                    mot[longueur_chaine - 1]= mot[longueur_chaine];
                    test_phrase = 1;
                    }

                if (numero_mot<=tableau_mot[numero_ligne]){ // Test si on est toujours à la ligne actuelle ou pas (grâ

                    if(ajouterMot(arbre, mot, numero_ligne, ordre, num_phrase)==1){
                        numero_mot++;
                        nb_mots++;
                        arbre->nb_mots_total= arbre->nb_mots_total+1;
                        }
                    if (test_phrase)
                        num_phrase++;
                    ordre++;
                }


                else{ // Si non, on a sauté à la ligne suivant
                    numero mot=1;
```

```
                else{ // Si non, on a sauté à la ligne suivant
                    numero_mot=1;
                    ordre = 1;
                    numero_ligne++;
                    if(ajouterMot(arbre, mot, numero_ligne, ordre, num_phrase)==1){
                        nb_mots++;
                        arbre->nb_mots_total= arbre->nb_mots_total+1;
                        numero_mot++;
                    }
                    ordre++;
                }
            test_phrase=0;
            }
        }

    return nb_mots;
}
```

3. Effacer_arb :

```
void effacer_arbre(NoeudABR* racine){ // Parcourt l'arbre binaire en postfixe pour efface chaque noeud

    ListePosition* list_pos;
    Position *pos_temp;
    Position *memoire;
    int noeud_efface = 0;
    int i =1;


    if (racine != NULL) // Arbre non vide
        {
            effacer_arbre(racine->filsGauche); // Parcourt
            effacer_arbre(racine->filsDroit); // Postfixe

            printf("Pour le mot : \"%s\", on a ", racine->mot);
            // Liberer d'abord les positions de chaque mot s'ils sont apparus a plusieurs endroits
            pos_temp =((racine->positions)->debut);
            list_pos = racine->positions;
            printf("%d occurence\n", list_pos->nb_elements);
            if (list_pos->nb_elements != 1) { // Condition pour savoir si le mot existe plusieurs fois dans s'il faut y

                while (pos_temp != NULL){
                    memoire = pos_temp;
                    pos_temp=pos_temp->suivant;   //Liberer les positions de la liste de position s'il y a
                    free(memoire);
                    printf("Free l'occurrence numero : %d\n", i); //Informe de l'occurence supprimee
                    i++;
                    }
            printf("Liste des positions de ce mot a ete videe\n");
            }
            free(list_pos->debut); //Liberer le premier element de a liste de position qui existe toujours
            free(list_pos); // Liberation du pointeur de liste de position
            printf("Liberation du pointeur de liste des positions effectuee\n");
```

```
            free(racine->mot); //Liberer le mot de chaque noeud
            printf("Liberation du mot effectue\n");
            free(racine);    //Liberer le noeud actuel (noeud cle)
            printf("Liberation du noeud\n\n");
            racine = NULL;  //Remise à NULL du noeud actuel apres liberation
            noeud_efface++;

        }

}
```