AREZKI Benjamin

SLAM Zineb

Rapport TP4 (ABR)

**Structures implémentées:**

**Arbre :** un pointeur sur Nœud et un nombre de nœuds , ou chaque nœud est un mot

🡪**Nœud :** contient un fils gauche (inférieur) et un fils Droite (supérieur), et une liste Chainée

**🡪Liste Chainée :** un pointeur sur Cellule et contient aussi le nombre d’occurrences du mot

🡪**Cellule :** contient le numéro de phrase, le numéro de ligne et la position dans la phrase du mot.

**Fonctions Principales**

# Liste\_pos \* creerListePositions():

**Début**

Liste\_pos\* listeP=(Liste\_pos\*) malloc(sizeof(Liste\_pos));

Si(listeP<>NULL)

listeP->debut:=NULL;

listeP->nb\_elements:=0;

retourner listeP;

Si non

retourner NULL;

**Fin**

**Complexite:** **O(1)**

# int ajouter position (Liste\_pos \*listeP, int ligne, int ordre):

**ajouter position (Liste\_pos \*listeP, int ligne, int ordre): Entier**

**Début**

Position \*pos =(Position\*)malloc(sizeof(Position));

Position\* temp= listeP->debut;

Si(pos<>NULL)

pos->numero\_ligne:=ligne;

pos->ordre:=ordre;

pos->suivant:=NULL

Si (listeP->debut==NULL) *// liste vide ajout au debut*

listeP->debut=pos;

retourner 1;

Sinon *//ajout en fin de liste*

**Tant que (temp->suivant <>NULL)**

temp:=temp->suivant;

**ftq**

temp->suivant:=pos;

listeP->nb\_elements:= listeP->nb\_elements+1;

retourner 1;

Sinon

retourner 0;

**Fin**

**Complexite:** **O(n) (n:** le nombre d’élements dans la liste chainéee)

# Créer Arbre :

**Début**

t\_ArbreBR\* Arbre;

Arbre:=(t\_ArbreBR\*) malloc(sizeof(Arbre));

Si(Arbre<>NULL)

Arbre->racine:=NULL

Arbre->nb\_mots\_differents:=0

Arbre->nb\_mots\_total:=0

retourner Arbre

Sinon

retourner NULL;

**Fin**

**Complexite:** **O(1)**

# Créer Noeud

**creer\_noeud( char \*mot, int ligne, int ordre) : t\_NoeudABR\***

**Début**

t\_NoeudABR\* New= (t\_NoeudABR\*)malloc(sizeof(t\_NoeudABR));

Si(New<>NULL)

New->mot:=mot;

Liste\_pos \*New\_liste:= creerListePositions();

ajouter\_position (New\_liste,ligne,ordre);

New->positions:= New\_liste;

New->filsGauche:=NULL;

New->filsDroit:=NULL;

retourner New;

Sinon

retourner NULL;

**Fin**

**Complexite:** **O(1)**

# Ajouter Nœud

Comme il fallait garder le père pour le relier au nouveau nœud on a procédé ainsi : On fait un premier appel de la fonction ajout ou on fait une comparaison du noeud avec le mot à insérer puis on lance la fonction ajout2 dans le sous arbre gauche (si le mot est inférieur à la racine )ou le sous arbre droit (si le mot est supérieur)

**ajouter\_noeud (t\_ArbreBR \*arbre, char \*mot, int ligne, int ordre) : Entier**

**Début**

**Si(arbre==NULL)** *// Pour éviter que ça bloque si on lance la fonction avant d’avoir crée l’arbre*

t\_ArbreBR\* arbre :=creer\_abr()

**Si(arbre->racine==NULL)** *//arbre vide donc on cree l’arbre*

t\_NoeudABR\* New =creer\_noeud(mot,ligne,ordre)

arbre->racine:=New;

retourner 1;

**Sinon**

comp:=strcasecmp(arbre->racine->mot,mot): entier

Si(comp==0)

ajouter\_position (arbre->racine->positions,ligne,ordre)

retourner 1;

Sinon Si (comp<0)

retourner ajout2(arbre->racine->filsGauche,arbr>racine,mot,ligne,ordre)

Sinon *// (comp>0)*

retourner ajout2(arbre->racine->filsDroit,arbre->racine,mot,ligne,ordre)

**Fin**

La fonction ajout2 prend en parametre le père du noeud pour le relier avec le noeud (qui sera le fils)

**int ajout2 (t\_NoeudABR\* Noeud, t\_NoeudABR\* pere, char\* mot, int ligne, int ordre)**

**Début**

Compar: entier

**Si(Noeud==NULL)** *// La condition d'arrêt de recursivité*

t\_NoeudABR\* New =creer\_noeud(mot,ligne,ordre); //Crée le nouveau Noeud

compar=strcasecmp(pere->mot, mot);

Si(compar<0)

pere->filsDroit:=New *// Fils Droit du père*

Sinon Si(compar>0)

pere->filsGauche:=New *//Fils Gauche du père*

retourner 1

**Sinon**

compar:=strcasecmp(Noeud->mot,mot);

**Si(compar==0)**

ajouter\_position(Noeud->positions,ligne, ordre);

retourner 1

**Si (compar<0)**

ajout2(Noeud->filsDroit,Noeud,mot,ligne,ordre);

**Si (compar>0)**

ajout2(Noeud->filsGauche,Noeud,mot, ligne,ordre);

**Fin**

**Complexite totale d’ajout:** **O(h) ( h = la hateur de l’arbre )**

# Rechercher Nœud :

**t\_NoeudABR\* Rechercher\_noeud(t\_NoeudABR\* racine, char\* mot)**

**Début**

**Si (racine==NULL)**

retourner NULL;

**Sinon**

compar: entier

compar:=strcasecmp(mot,racine->mot)

Si(compar==0) *// mot trouve*

retourner racine

Si(compar<0)

retourner Rechercher\_noeud(racine->filsGauche,mot)

Sinon

retourner Rechercher\_noeud(racine->filsDroit,mot)

**Fin**

**Complexite:** **O(h) :** h = la hateur de l’arbre ; dans le pire des cas le mot est une feuille

# Afficher Arbre : Tous les mots

**Procédure : afficher\_arbre(t\_NoeudABR\* racine)**

**Début**

Si (racine <> NULL)

**Si (racine->filsGauche <> NULL)**

afficher\_arbre(racine->filsGauche)

**Affichage(racine->mot)**

**Si (racine->filsDroit <> NULL)**

afficher\_arbre(racine->filsDroit)

**Fin**

**Complexite:** **O(h) :** h = la hateur de l’arbre

# Charger Fichier :

**int charger\_fichier(t\_ArbreBR \*arbre, char \*filename)**

**Début**

FILE\* f= NULL;

Entier: ordre, ligne, num\_phrase,i

Char current;

ligne=1, ordre=1, num\_phrase=1

Si ((f=fopen(filename,"r"))==NULL) *//Erreur d’ouverture de fichier*

Sortie

current=fgetc(f); *// recupère un caractère du fichier*

**Tant que(!feof(f))** *// Tant qu’on n’est pas dans la fin de fichier*

char \*mot\_ajout:=malloc(sizeof(char)\*20);

i:=0

**Tant que (current)**

Si(current==' ' || current=='.' || current=='\n') break; // word's end

mot\_ajout[i]:=current;

i:= i+1

current=fgetc(f);

**ftq**

ajouter\_noeud(arbre,mot\_ajout,ligne,ordre);

ordre:= ordre+1

Si(current=='\n') *// si saut de ligne*

Ligne:=ligne+1

Si(current=='.') *// Si fin de phrase*

num\_phrase:= num\_phrase +1

current=fgetc(f*); // character suivant*

**ftq**

Fermer(f) *//fermer fichier*

**Fin**

**Complexite:** **O(h) :** h = la hateur de l’arbre

**Caractéristiques de l’ABR :**

# Hauteur du Nœud :

Par définition la hauteur d’un nœud est le maximum entre la hauteur du sous  arbre Gauche et le sous arbre droit. Donc pour calculer la hauteur nous avons mis en place une fonction récursive qui s’incrémente, jusqu’à ce qu’on arrive au Nœud (*racine==NULL)*

**hauteur (t\_NoeudABR\* racine ) : Entier**

**Début**

Si ( racine == NULL )

retourner 0;

Sinon

retourner (1 + MAX[ hauteur(racine->filsDroit) , hauteur(racine->filsGauche) )]

**Fin**

Et donc à pour trouver la hauteur , on appelera la fcontion de la sortie :

**Retourner (1 + MAX( hauteur(racine->filsDroit) , hauteur(racine->filsGauche)**

**Complexite:** **O(h) (h la hauteur de l’arbre)**

# Est Equilibré

int estEquilibre(t\_NoeudABR\* r,int nb ): Entier

**début**

Si racine != NULL

Si (valeurAbsolue(hauteur(r->filsGauche)- hauteur(r->filsDroit))<=1)

Si EstEquilibré(r->filsGauche) && EstEquilibré(r->filsDroit)

Return vrai

Sinon

Return Faux

Sinon

Return Faux

FinSi

Sinon

Return vrai

Finsi

**Fin**

**Complexite:** **O(h)** *(h = la hateur de l’arbre)* de par l’appel récursif de la fonction hauteur.

**Afficher Phrase avec 2 mots:**

On copie tous les mots de la ligne demandée (numPhrase) dans un tableau:

**checkPositionList(t\_NoeudABR\* racine, int numPhrase,char \*\* file): Entier**

**Début**

Entier: nb:=0;

**Si (racine <> NULL)**

Position \*tmp = racine->positions->debut;

**Tant que (tmp<> NULL)**

Si (tmp->numero\_phrase==numPhrase)

nb = nb +1; // le nombre de mots dans la ligne

strcpy(file[tmp->ordre],racine->mot); // on copie le mot tmp=tmp->suivant; // noeud suivant🡪 Mot suivant

**ftq**

retourner nb;

**Fin**

**Complexite:** **O(n)** *(n le nombre d’elements dans la liste Chainée)*

**afficherPhrase (t\_NoeudABR\* racine, int numPhrase,char \*\* file,int nb) : Entier**

**début**

Si (racine <> NULL)

nb = nb + checkPositionList(racine,numPhrase,file)

Si(racine->filsGauche <> NULL )

nb = afficherPhrase(racine->filsGauche,numPhrase,file,nb);

Si(racine->filsDroit <>NULL)

nb = afficherPhrase(racine->filsDroit,numPhrase,file,nb)

return nb;

**Fin**

**Complexite:** **O(n+h)** *h=la hauteur de l’arbre et n= le nombre d’élements de la liste Chainée Position*

# Algorithme principal d’affichage :

Pour afficher une phrase on crée un tableau de chaine de caractère ou chaque case correspondra à un mot. On considère qu’une phrase contient au maximum 50 mots.

**Procédure: AfficherPhrase2(t\_NoeudABR\* racine, int numPhrase)**

**Début**

char \*\* file = malloc(50\*sizeof(char\*));*// tableau de mots🡪 phrase*

int i= 0;

int nb=0;

if (file != NULL)

for(i=0 ; i < 50 ; i++)

file[i] = malloc(20 \* sizeof(char\*\*)

nb= afficherPhrase(racine,numPhrase,file,nb)

for(i=1 ; i < nb +1 ; i++)

printf(" %s ",file[i])

**Fin**

**Arbre AVL**

Dans cette partie il va falloir ajouter les champs suivants caractérisant chaque nœud : La hauteur (h) , l’équilibre (eq) , le père (père qu’on a choisi de noter père[x] pour que le code reste clair) .

Nous avons choisi de détailler au mieux la fonction de rotation Droite avec les changements de hauteur . On réalise d’abord tous les changements dans la rotation puis on sauvegarde toutes les hauteurs des nœud ou sous arbres. Vous remarquerez que si le nœud est nul nous avons affecté à la hauteur une valeur de (-1) ceci pour que notre programme ne bloque pas si le sous arbre est vide. En ce qui concerne le changement de la hauteur nous avons fait une boucle qui remonte du nœud ou il y’a eu changement à la racine , cette boucle s’arrête quand on atteint la racine (père est nul) ou lorsque le Delta=0 c’est à dire il n’y a plus de changement de hauteur et donc tout ce qui est au-dessus ne changera pas de hauteur.

Pour faire un arbre equilibré nous avons choisi de réequilbrer l’arbre depuis le début , c’est à dire à chaque insertion, et donc comme l’arbre était déjà equilbré à l’insertion on n’aura pas à remonter toute la racine mais juste à metre à jour les équilibres du noeud concerné.

# Rotation Droite:

B

C

B

C

D

D

**Rotation\_D ( ABR: A, Noeud: x) : Arbre, Entier**

**Début**

**Si(X<>NUL et filsGauche[x]<>NIL)**

Pere[y]:=Pere[x]

**Si (y🡪filsGauche<>NIL)** hB := y🡪filsGauche🡪h

**Sinon** hB := -1

**Si (y🡪filsDroit <>NIL)** hC := y🡪filsDroit🡪h

**Sinon** hC := -1

**Si (x🡪filsDroit <> NIL)** hD := x🡪filsDroit🡪h

**Sinon** hD := 1

**Si(x==filsGauche[pere[x]]** filsGauche[pere[x]]🡨y

**Sinon** filsDroit[pere[x]]🡨y

**Si (x==pere[x]🡪filsGauche** Pere[x]🡪filsGauche:= y

**Sinon** Pere[x]🡪filsDroit:= y

pere[x]:=y

x🡪filsGauche:= y🡪filsDroit

y🡪filsDroit:=x

h:=x🡪h

x🡪h:= 1+MAX(hC,hD)

y🡪h:=1+MAX(hB, x🡪h)

Δ:= (y🡪h) – h

V:= y

P:= pere[y]

**Tant que p<>NIL et Delta<>0 faire**

**Si (v==p🡪filsGauche)** u:= p🡪filsDroit

**Sinon** u:=p🡪filsGauche

**Si (u<>NIL)** hu:=u🡪p

**Sinon** hu:= -1

**Si (v🡪h >hu)**

p🡪h:= p🡪h + Δ

y:=p

p:= pere[p] //on remonte dans l’arbre

**Sinon**

Δ:=0 *//Sortie de la boucle = il n’y’a plus de modifications*

Retourner y, Δ

**Sinon**

Retourner x, Δ

**Fin**

**Rééquilibrer ( ABR: A, Noeud: x) : Arbre, Entier**

**Début**

**Si eq(x)>1**

Alors Y🡨filsGauche[x]

Si eq(y)>=0

Alors retourner Rot\_D(x, Δ)

Sinon

Retourner Rot\_G\_D(x, Δ)

**fsi**

**Si eq(y)<-1**

Alors Y🡨filsDroite[x]

Si eq(y)<=0

Alors retourner Rot\_D\_G(x,Δ)

Sinon

Retourner Rot\_D\_G(x,Δ)

**Finsi**

**Fin**

**Equilibre\_RotD ( ABR: A, Noeud: x) : Entier**

**Début**

Eqx := x🡪eq

Eqy := y🡪eq

x🡪eq := eqx-1-max(0,eqy)

**Si x🡪eq >=0**

**Alors** y🡪eqy := eqy-1

**Sinon**

Eqy🡪 eqx-2+MIN(0,eqy)

**Fin**

**Delta\_Mise\_a\_jour\_RotD (Noeud: x)**

**Début**

**Si x🡪eq<0**

Alors Δ :=1

**Sinon Si y🡪eq>0**

Alors Δ :=-1

**Sinon**

Δ :=0

**Fin**

**Rot\_G\_D (A : Arbre , Noeud: x) : Arbre, Entier**

**Début**

X := A🡪filsGauche

(A🡪filsGauche, Δ0) := Rot\_G\_D(x,0)

**Si( x🡪eq>0 ou eq🡪x==0 et Δ0==0)**

Δ:= Δ0

**Sinon**

Δ :=0

A🡪eq := a🡪eq + Δ

(A, Δ) := Rot\_D(A, Δ)

retourner (A, Δ)

**Fin**

**Insérer\_AVL ( ABR: A, Noeud: x) : Arbre, Entier**

**Début**

Si A=NULL // Condition d’arrêt = nœud Nul

Δinsert:=1

Retourner (x, Δinsert)

**Si (x🡪cle < A🡪cle)**

Alors (A🡪filsGauche, Δfg):= insérer\_AVL(A🡪filsGauche,x)

Si A🡪eq>=0

Alors ΔA:= Δfg

Sinon ΔA := 0

**Sinon** *//par symétrie*

(A🡪filsDroit, Δfd):= insérer\_AVL(A🡪filsDroit,x)

Si A🡪eq<=0

Alors ΔA:= Δfd

Sinon ΔA := 0

**Finsi**

Si |A🡪eq|>2 // Arbre non equilibre

Alors (A, Δreeq):= Reequilibrer(A, ΔA)

Retourner (A, ΔA)

**Fin**

**Difficultés rencontrées et solutions :**

|  |  |
| --- | --- |
| **Difficultés** | **Solutions** |
| **L’ordre : Lors de la présentation, nous avons remarque qu’un décalage était créé dans l’ordre après la seconde phrase.** | **Nous ajoutions +1 dans l’ordre lors du changement de phrase, nous avons supprimé ce +1 et on obtient l’ordre correct** |
| **La hauteur : La hauteur était toujours supérieure de 1** | **Nous avons soustrait 1 au retour de la fonction, cela est dut qu’un arbre à un seul nœud est de hauteur 0** |
| **estEquilibre : Nous avions compris qu’il s’agissait de testait si un arbre était parfait** | **Finalement nous avons corrigé cette fonction pour répondre à la consigne, le texte1.txt est équilibré et pas le texte2.txt** |