École Centrale de Lille

AAP - Algorithmique avancée et programmation

BARBOZA DE DEUS FONSECA Fernando

BASTOS DUNLEY Rafael

SILVEIRA LARRUBIA Eduardo

Compte Rendu - Fil Rouge

Responsable :

Monsieur THOMAS BOURDEAUD HUY

Villeneuve-d'Ascq

Décembre 2021

Table des matières

1 Introduction **3**

2 displayAVL.exe **3**
2.1 Développement 3
2.2 Résultat 9

3 indexation.exe **10**
3.1 Développement 10
3.2 Résultat 14

4 anagrammes.exe **14**
4.1 Développement 14
4.2 Résultat 18

5 Conclusion **18**

6 Réferences **18**

Table des figures

1 Fonction *newNode* et *insertAVL* 4
2 Fonction *rotateLeft / rotateRight* 5
3 Fonction *balance* 6
4 Fonction *genDot* 7
5 *main* Partie 1 8
6 Résultat Partie 1 9
7 Fonction *newNode* et *insertAVL* (changements en évidence) 10
8 Fonction *eltSig* 11
9 Fonction *depthNodeAVL* 11
10 Fonctions *lenWords et nbWords* 12
11 Fonction *print_Search* 12
12 *main* Partie 2 13
13 Résultat à partir du dictionnaire (Dico_03.txt) 14
14 Fonction *sortAddNodeAna* 15
15 Fonction *getAnagrams* 15
16 Fonction *printAnagrams* 16
17 *main* Partie 3 17
18 Résultat à partir du dictionnaire (Dico_03.txt) 18

1 Introduction

L'objectif du Fil Rouge est de travailler avec la théorie des arbres binaires, en effectuant l'équilibrage, l'affichage AVL, l'assemblage de listes (mots de même signature) et le scanning de tous les éléments.

La activité est divisé en 3 parties :

— Programme 1 (displayAVL.exe) : Implémenter arbre binaire avec index du balance et affichage en PNG ;
— Programme 2 (indexation.exe) : Implémenter la distribution des mots d'un dictionnaire dans un arbre binaire au moyen de la signature de chaque mot ;
— Programme 3 (anagrammes.exe) : Implémenter l'analyse de l'ensemble de la structure afin de retrouver les mots de même signature. ;

Le dossier Fil_Rouge_LesBrésiliens est organisé comme suit :

— Fichier ***README*** avec des informations sur makefile e la structure ;
— Fichier ***makefile*** pour toutes les parties ;
— Dossier ***include*** avec les fichiers utilisés dans tous les exercices ;
— Dossiers ***Partie**** avec la ***main.c*** et les fichiers utilisés dans chaque exercice ;

2 displayAVL.exe

2.1 Développement

Cette partie est composée de six fonctions principales :

— ***newNode*** : Création d'un nouveau nœud d'un arbre AVL contenant 4 données : le contenu (e), 2 pointeurs et un facteur d'équilibrage.
— ***insertAVL*** : Insertion de la donnée (e) dans l'arbre binaire.Après insertion, appel de la fonction d'équilibrage.

```c
static T_avl newNodeAVL(T_elt e) {

    T_avlNode * pAux;
    pAux = (T_avlNode *) malloc(sizeof (T_avlNode));
    CHECK_IF(pAux, NULL,"erreur malloc dans newNode");
    pAux->val = eltdup(e);
    pAux->l = NULL;
    pAux->r = NULL;
    pAux->bal = BALANCED;

    return pAux;
}


int insertAVL (T_avlNode ** pRoot, T_elt e) {

    // cas de base
    if ((*pRoot)== NULL){
        (*pRoot) = newNodeAVL(e);
        return 1;
    }

    // cas général
    int deltaH;
    // On compare l'élément et la racine
    if (eltcmp(e, (*pRoot)->val) <= 0) {
    // On insère dans le sous-arbre gauche
        deltaH = insertAVL(&(*pRoot)->l, e);
        (*pRoot)->bal += deltaH;
    } else {
    // On insère dans le sous-arbre droit
        deltaH = insertAVL(&(*pRoot)->r, e);
        (*pRoot)->bal -= deltaH;
    }

    // Si l'arbre ne croît pas, on quitte
    if(deltaH ==0) return 0;
    // Sinon on balance
    else (*pRoot) = balanceAVL(*pRoot);

    //Renvoie 1 si le nœud est déséquilibré, 0 sinon.
    return (*pRoot)->bal != BALANCED;

}
```

FIGURE 1 – Fonction *newNode* et *insertAVL*

— ***rotateLeft / rotateRight*** : Fait pivoter le nœud par rapport à ses nœuds adjacents. Outil pour équilibrer l'arbre.

```c
static T_avlNode * rotateLeftAVL (T_avlNode * B) {
    // rotation gauche

    T_avlNode *A = B->r;

    T_bal b_ = heightAVL(B->l) - heightAVL(A);
    T_bal a_ = heightAVL(A->l) - heightAVL(A->r);

    B->r = A->l;
    A->l = B;

    // Calcul des nouvelles balances
    T_bal b = b_ + 1 - MIN2(0,a_);
    T_bal a = a_ + 1 + MAX2(0,b);

    A->bal = a;
    B->bal = b;

    return A;
}


static T_avlNode * rotateRightAVL (T_avlNode * A) {
    // rotation droite

    T_avlNode *B = A->l;

    T_bal a = heightAVL(B) - heightAVL(A->r);
    T_bal b = heightAVL(B->l) - heightAVL(B->r);

    A->l = B->r;
    B->r = A;

    // Calcul des nouvelles balances
    T_bal a_ = a - 1 - MAX2(0,b);
    T_bal b_ = b - 1 + MIN2(0,a_);

    A->bal = a_;
    B->bal = b_;

    return B;

}
```

FIGURE 2 – Fonction *rotateLeft / rotateRight*

— ***balance*** : Détection du déséquilibre de l'arbre AVL et appel des fonctions de rotation vers la gauche ou vers la droite.

```c
static T_avlNode * balanceAVL(T_avlNode * node) {
    // rééquilibrage de A

    if( node->bal == DOUBLE_LEFT){
        if( node->l->bal == RIGHT){
            // leftRightCase
            node->l = rotateLeftAVL(node->l);
            return rotateRightAVL(node);
        }
        else{
            // leftLeftCase
            return rotateRightAVL(node);
        }
    }

    if(node->bal == DOUBLE_RIGHT){
        if( node->r->bal == LEFT){
            // rightLeftCase
            node->r = rotateRightAVL(node->r);
            return rotateLeftAVL(node);
        }
        else{
            // rightRightCase
            return rotateLeftAVL(node);
        }
    }

    return node;

}
```

FIGURE 3 – Fonction *balance*

— ***genDotAVL*** : Création d'un fichier .dot pour la représentation graphique de l'arbre.

```c
static void  genDotAVL(T_avl root, FILE *fp) {
    // Attention : les fonction toString utilisent un buffer alloué comme une variable statique
    // => elles renvoient toujours la même adresse
    // => on ne peut pas faire deux appels à toString dans le même printf()

    fprintf(fp, "\t\"%s\"",toString(root->val));
    fprintf(fp, " [label = \"{{<c> %s | <b> %d} | { <g> | <d>}}\"];\n",toString(root->val),root->bal);
    if (root->r == NULL && root->l == NULL)
    {
        fprintf(fp, "\t\"%s\"", toString(root->val));
        fprintf(fp, " [label = \"{{<c> %s | <b> %d} | { <g> NULL | <d> NULL}}\"];\n", toString(root->val),root->bal);
    }
    else if (root->r == NULL)
    {
        fprintf(fp, "\t\"%s\"", toString(root->val));
        fprintf(fp, " [label = \"{{<c> %s | <b> %d} | { <g> | <d> NULL}}\"];\n", toString(root->val),root->bal);
    }
    else if (root->l == NULL)
    {
        fprintf(fp, "\t\"%s\"",toString(root->val));
        fprintf(fp, " [label = \"{{<c> %s | <b> %d} | { <g> NULL | <d> }}\"];\n", toString(root->val),root->bal);
    }

    if (root->l)
    {
        fprintf(fp, "\t\"%s\"",toString(root->val));
        fprintf(fp, ":g -> \"%s\";\n", toString(root->l->val));
        genDotAVL(root->l, fp);
    }

    if (root->r)
    {
        fprintf(fp, "\t\"%s\"",toString(root->val));
        fprintf(fp,":d -> \"%s\";\n", toString(root->r->val));
        genDotAVL(root->r, fp);
    }
}
```

FIGURE 4 – Fonction *genDot*

— ***main*** : Vérification des entrées, lecture des fichiers (Prenoms.txt) et création de l'arbre, générant à chaque itération une représentation graphique.

```c
int main(int argc, char ** argv) {

    //Vérifie si l'argument est valide
    if(argc < 3){
        printf("Not a valid input.\n");
        printf("Please insert file with names path and the number of names with de comand.\n");
        printf("Exemple: ./displayAVL.exe PrenomsV1.txt 10\n");
        NL();
        return 1;
    }

    FILE *fpNames;

    fpNames = fopen(argv[1], "r");
    CHECK_IF(fpNames, NULL, "File not found");

    int n = atoi(argv[2]);

    T_avl root = NULL;

    CLRSCR();
    WHOAMI();

    ////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////

    outputPath = "Partie1/output";

    char name[30];
    //Boucle qui insère un nouveau nom dans l'arbre
    // et crée une représentation graphique de l'arbre
    // actualisée pour chaque nouveau mot
    for(int i = 0;i<n;i++){

        fgets(name, sizeof(name), fpNames);
        insertAVL(&root, name);
        createDotAVL(root, "displayAVL");

    }

    //Ferme le fichier et libère la mémoire
    fclose(fpNames);
    freeAVL(root);

    return 0;
}
```

FIGURE 5 – *main* Partie 1

2.2 Résultat

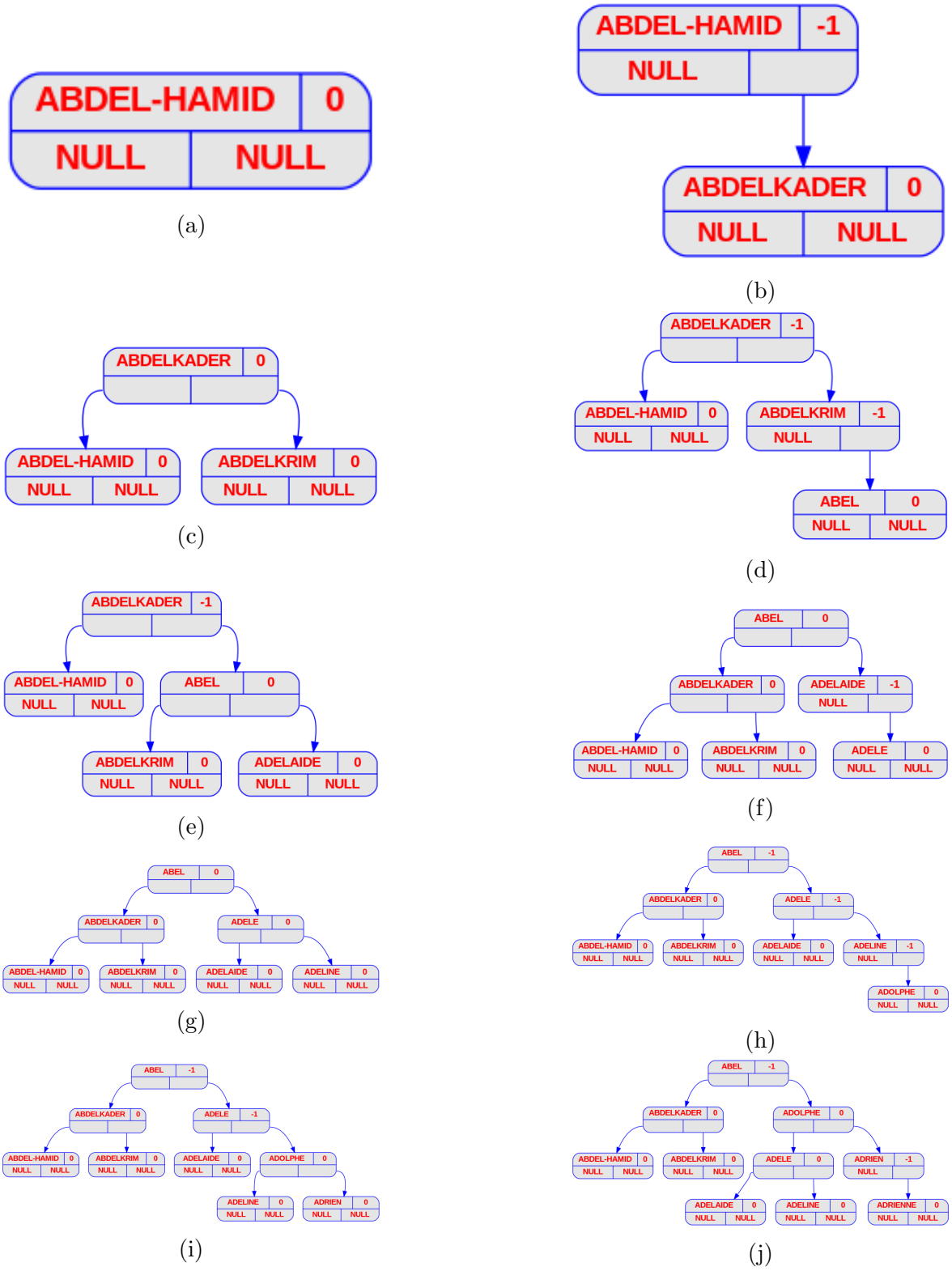

FIGURE 6 – Résultat Partie 1

3 indexation.exe

3.1 Développement

Cette partie est composée de huit fonctionnes principales :

— ***newNode*** : Création d'un nouveau nœud d'un arbre AVL contenant 5 données : une liste chaînée avec le contenu (e), la signature de (e), 2 pointeurs et un facteur d'équilibrage.

— ***insertAVL*** : Insertion de la donnée (e) dans l'arbre binaire. Après insertion, appel de la fonction d'équilibrage. Si (e) a la même signature que celle du nœud, il est inséré dans la liste liée du nœud.

```
static T_avl newNodeAVL(T_elt e) {

    T_avlNode * pAux;
    pAux = (T_avlNode *) malloc(sizeof (T_avlNode));
    CHECK_IF(pAux, NULL,"erreur malloc dans newNode");
    pAux->words = addNode(e, pAux->words);
    pAux->sig = eltSig(e);
    pAux->l = NULL;
    pAux->r = NULL;
    pAux->bal = BALANCED;

    return pAux;
}

int insertAVL (T_avlNode ** pRoot, T_elt e) {

    // cas de base
    if ((*pRoot)== NULL){
        (*pRoot) = newNodeAVL(e);
        return 1;
    }

    // cas général
    int deltaH = 0;
    T_elt sig_e = eltSig(e);
    // On compare l'élément et la racine
    if (eltcmp(sig_e, (*pRoot)->sig) < 0) {
    // On insère dans le sous-arbre gauche
        deltaH = insertAVL(&(*pRoot)->l, e);
        (*pRoot)->bal += deltaH;
    } else  if(eltcmp(sig_e, (*pRoot)->sig) > 0) {
    // On insère dans le sous-arbre droit
        deltaH = insertAVL(&(*pRoot)->r, e);
        (*pRoot)->bal -= deltaH;
    } else if (eltcmp(sig_e, (*pRoot)->sig) == 0){
    // On insère dans la list de mots
        if(!inListIte(e, (*pRoot)->words))
            (*pRoot)->words = addNode(e, (*pRoot)->words);
    }

    // Si l'arbre ne croît pas, on quitte
    if(deltaH ==0) return 0;
    // Sinon on balance
    else (*pRoot) = balanceAVL(*pRoot);
    //Renvoie 1 si le nœud est déséquilibré, 0 sinon.
    return (*pRoot)->bal != BALANCED;
}
```

FIGURE 7 – Fonction *newNode* et *insertAVL* (changements en évidence)

— ***eltSig*** : Créer une signature en un mot en mettant les lettres dans l'ordre alphabétique.

```c
T_elt eltSig(T_elt e) {

    T_elt sig = NULL;
    sig = strdup(e);
    char temp;

    int i, j;
    int n = strlen(sig);
    //Bubble sort
    for (i = 0; i < n-1; i++) {
        for (j = i+1; j < n; j++) {
            if (sig[i] > sig[j]) {
                temp = sig[i];
                sig[i] = sig[j];
                sig[j] = temp;
            }
        }
    }

    return sig;
}
```

FIGURE 8 – Fonction *eltSig*

— ***depthNodeAVL*** : Calcul de la profondeur de la branche d'un nœud.

```c
int depthNodeAVL(T_avl root, T_elt e){
    // profondeur d'un nœud
    int depth = -1;

    if(root == NULL) return depth;

    if(eltcmp(e, root->sig) == 0) return depth + 1;

    depth = depthNodeAVL(root->l, e);
    if (depth >= 0) return depth +1;

    depth = depthNodeAVL(root->r, e);
    if (depth >= 0) return depth +1;

    return depth;

}
```

FIGURE 9 – Fonction *depthNodeAVL*

— ***lenWords_AVL*** : Calcul de la taille des mots dans les nœuds de l'arbre.

— ***nbWords_AVL*** : Calcul du nombre de mots dans l'arbre entier.

```c
int lenWords_AVL(T_avl root){
    //Trouver la taille du plus grand mot de l'arbre
    if(root ==NULL) return 0;

    return MAX3(strlen(root->sig), lenWords_AVL(root->l), lenWords_AVL(root->r));
}
int nbWords_AVL(T_avl root){
    // nb de mot d'un arbre (y compris les feuilles)
    if (root == NULL)
        return 0;

    return getSizeRec(root->words) + nbWords_AVL(root->l) + nbWords_AVL(root->r);

}
```

FIGURE 10 – Fonctions *lenWords et nbWords*

— ***print_SearchAVL*** : On en cherche un dans l'arbre. S'il est trouvé, il affiche les anagrammes de ce mot, la profondeur du nœud et le temps nécessaire pour le trouver.

```c
void print_SearchAVL(T_avl root, T_elt word){
    //Fonction permettant de rechercher un certain mot dans l'arbre
    //et d'afficher les informations correspondantes

    T_elt sig = eltSig(word);
    printf("À la recherche de %s avec signature: %s", toString(word), sig); NL();
    clock_t begin = clock();
    T_avlNode *node = searchAVL_rec(root, sig);
    clock_t end = clock();

    //Teste si le nœud est nul ou s'il n'y a pas le mot
    //(même s'il a sa signature dans l'arbre)
    if(node == NULL || !thereIsWord(node, word)){
        printf("Mot pas trouvé :("); NL();
        return;
    }

    printf("Liste des mots de même signature:"); NL();
    showList(node->words); NL();

    printf("La profondeur du noeud est: %d", depthNodeAVL(root, sig)); NL();

    double time_spent = 1000.0*(end-begin)/CLOCKS_PER_SEC;
    printf("Temp nécessaire pour trouver ce mot: %.3f ms", time_spent); NL();

}
```

FIGURE 11 – Fonction *print_Search*

— ***main*** : Vérification des entrées, lecture des fichiers (Dico.txt) et création de l'arbre et affiche les statistiques et paramètres (Taille des mots, Nombre de mots, Temps pour construire l'arbre, nombre de nœuds et hauteur minimale d'arbre pour un même nombre de nœuds). Après, disponibilité de la recherche de mots dans les arbres à l'aide de la fonction print_searchAVL pour afficher les résultat.

```
//Construction de l'arbre
while(fgets(name, sizeof(name), fpNames)){

    j = 0;
    while(name[j] != '\n'){
        if(name[j] == '\0') break;
        j++;
    }
    name[j] = '\0';
    insertAVL(&root, name);
    i++;

}

fclose(fpNames);

//Afficher des informations sur l'arbre
clock_t end = clock();
double time_spent = 1000.0*(end-begin)/CLOCKS_PER_SEC;

printf("Taille des mots du dictionnaire: %d", lenWords_AVL(root)); NL();
printf("Nombre de mots du dictionnaire: %d", nbWords_AVL(root)); NL();
printf("Durée de construction de l'arbre: %.3f ms", time_spent); NL();
printf("Nombre de noeuds: %d\nHauteur de l'abre: %d", nbNodesAVL(root), heightAVL(root)); NL();
int minHeight = (int) log2(nbNodesAVL(root));
printf("Hauteur minimale d'un arbre de %d noeuds: %d", nbNodesAVL(root), minHeight); NL();

//Recherche de mots dans la liste
char word[30];
printf("--------------\n");
printf("Saisir un mot dans l'arbre:"); NL();
while(1){
    printf("Entrez le mot: ");
    scanf("%s", word);
    if(word[0] == '0') break;

    print_SearchAVL(root, word);

    printf("\n--------------\n");
    printf("Type 0 to end the program"); NL();

}

freeAVL(root);
return 0;
```

FIGURE 12 – *main* Partie 2

3.2 Résultat

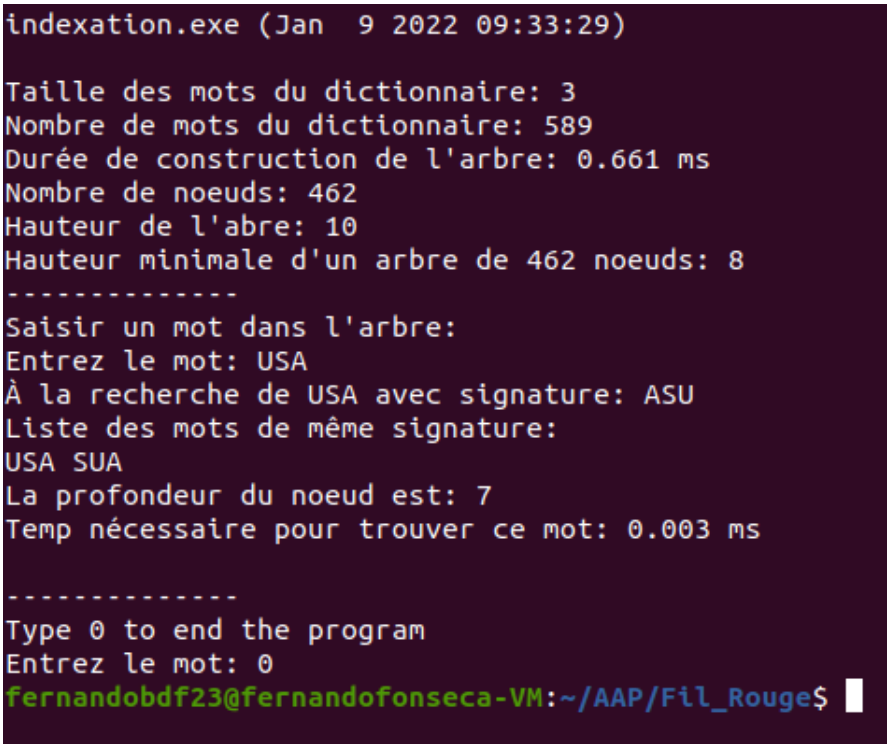

FIGURE 13 – Résultat à partir du dictionnaire (Dico_03.txt)

4 anagrammes.exe

4.1 Développement

Cette partie est composée de quatre fonctionnes principales :

— ***sortAddNodeAna*** : insertion d'une liste de mots par ordre alphabétique d'anagrammes.

```
T_anagramList sortAddNodeAna(T_list e, T_anagramList l){
    //Insère une liste de mots par ordre croissant dans
    //la liste d'anagrammes

    if(l!=NULL){
        T_anagramList next, aux = l;

        while(aux && aux->pNext && getSizeRec(e) > getSizeRec(aux->pNext->words)){
            aux = aux->pNext;
        }

        next = aux->pNext;
        aux->pNext = newNode(e);
        aux->pNext->pNext = next;

        return l;
    }
    else return newNode(e);
}
```

FIGURE 14 – Fonction *sortAddNodeAna*

— ***getAnagrams*** : Analyse de l'arbre et, si elle trouve des anagrammes (liste de mots avec plus de 2 éléments), un pointeur (dans la liste chaînée des anagrammes) est créé vers la liste respective. La liste de pointeurs est un ordre croissant (premièrement,liste 2 mots, après, liste de 3 mots, ...).

```
void getAnagrams(T_avl root, T_anagramList *pL){

    //cas de base
    if(root == NULL) return;

    //Si on a des anagrammes dans la liste de mots
    if(getSizeRec(root->words) > 1){
        *pL = sortAddNodeAna(root->words, *pL);
    }

    getAnagrams(root->l, pL);
    getAnagrams(root->r, pL);

    return;
}
```

FIGURE 15 – Fonction *getAnagrams*

— ***printAnagrams*** : affiche la liste pointeurs dans un ordre decroissant.

```
void printAnagrams(T_anagramList anagramsList){
    //Affiche les anagrammes par ordre décroissant

    if(anagramsList == NULL) return;

    printAnagrams(anagramsList->pNext);
    showList_rec(anagramsList->words); NL();

    return;
}

void freeAnagrams(T_anagramList l){

    if(l == NULL) return;

    freeAnagrams(l->pNext);

    free(l);

    return;
}
```

FIGURE 16 – Fonction *printAnagrams*

— ***main*** :
Vérification des entrées, lecture des fichiers (Dico.txt) et création de l'arbre, création de la liste d'anagrammes et affiche le nombre d'anagrammes.

```
char name[30];
int j;
//Construction de l'arbre
while(fgets(name, sizeof(name), fpNames)){

    j = 0;
    while(name[j] != '\n'){
        if(name[j] == '\0') break;
        j++;
    }
    name[j] = '\0';
    insertAVL(&root, name);
}

fclose(fpNames);


int nb = nbAnagrams(root);
printf("Nombre de mots du dictionnaire disposant d'anagrammes = %d", nb); NL(); NL();

//Crée une liste chaînée auxiliaire contenant
//tous les anagrammes de l'arbre
T_anagramList l = NULL;
getAnagrams(root, &l);
printAnagrams(l);

freeAnagrams(l);
freeAVL(root);

return 0;
```

FIGURE 17 – *main* Partie 3

4.2 Résultat

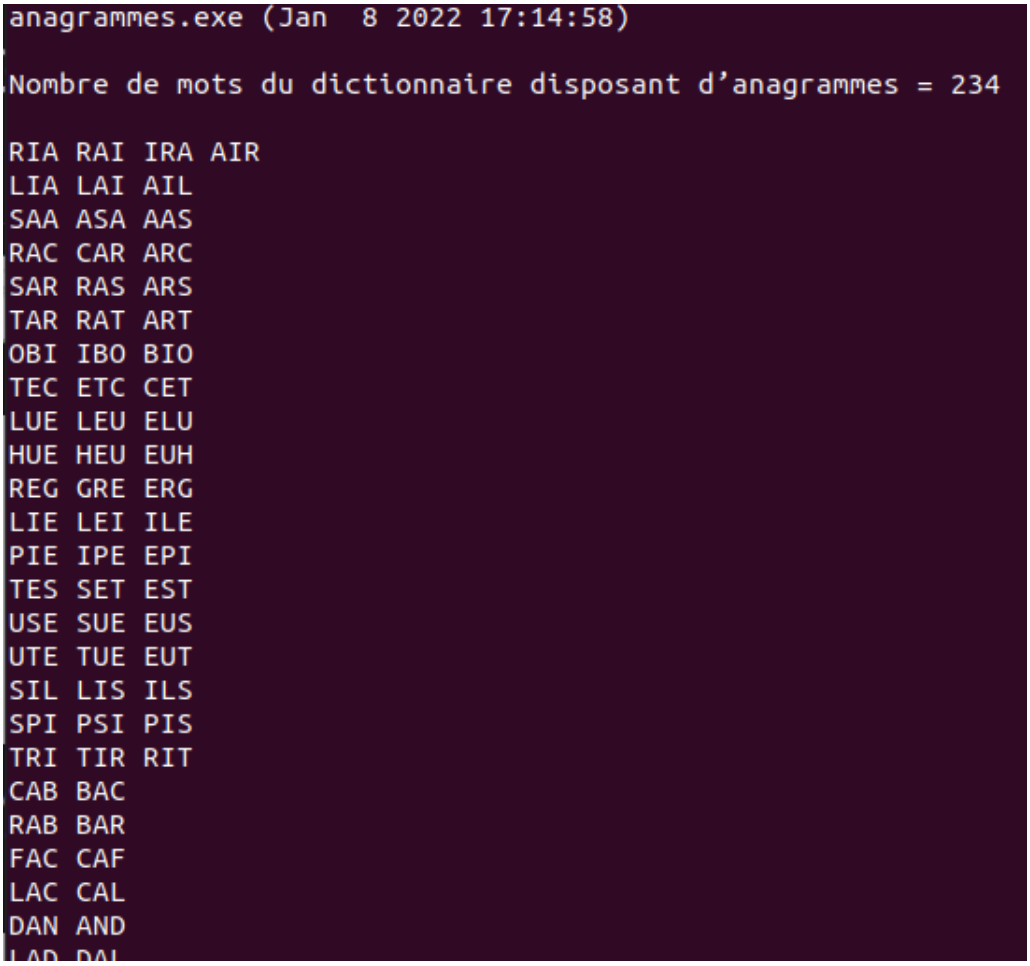

FIGURE 18 – Résultat à partir du dictionnaire (Dico_03.txt)

5 Conclusion

Grâce à ce travail, il a été possible de vérifier plusieurs applications et itérations entre différentes structures de données, en particulier l'arbre AVL. Il a été possible de se rendre compte de son avantage par rapport à un arbre binaire standard ou une liste chaînée, par exemple. Avec cela, nous avons élaboré un code succinct et robuste, permettant la création de structures pour différents types de données dans la même logique.

6 Réferences

— AAP - Séance 4
— Fil Rouge 2021
— Arbre AVL