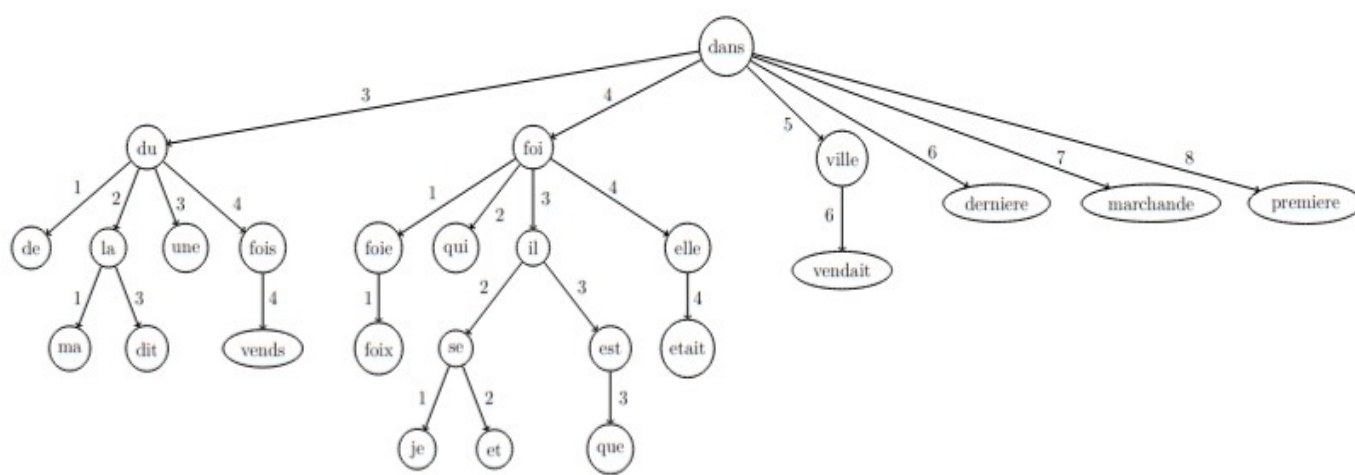


Rapport sur le rendu de l'étape 3 :
Algorithmique des arbres



Fait par :
MABROUK Fayez
&
BASHEER AHAMED Nafis

TP1

Introduction

Dans cette troisième étape de projet, l'objectif est de proposer des corrections pour les listes des erreurs obtenu dans l'étape précédente en parcourant un arbreBK de recherche.

Démarche de programmation

int inserer_dans_ArbreBK(ArbreBK * A, char * mot) :

Nous avons implémenté la fonction d'une manière itérative :
Si l'arbre est vide nous allouons un nœud pour la racine par la fonction **ArbreBK alloue_noeud(char *mot, int val)** , sinon nous parcourons l'arbre en calculant les distances Levenshtein et en vérifiant si un nœud possède la même distance de Levenshtein par rapport à la racine si c'est vrai alors nous allons dans son fils gauche sinon nous allons dans son frère droit. De plus, pour afficher l'arbre d'une manière croissante nous avons ajoutés le condition **if ((*fils)->valeur > d)** ça veut dire que aucun nœud n'a la même distance de Lenvenshtein que le mot que l'on souhaite ajouter.

Liste rechercher_dans_ArbreBK(ArbreBK A, char * mot) :

Pour cette fonction nous n'avons pas réussi à faire le même prototype à cause de variable *seuil qui nous a fait des problèmes après avoir faire plein des tests alors nous avons développé la fonction **void rechercher_dans_ArbreBK(ArbreBK A, char* mot, Liste* corrections, int *seuil)**.
Ensuite, Nous avons implémenter presque le même algorithme vu en cours en utilisant le principe d'inégalité triangulaire pour enregistrer les meilleurs propositions pour le mot mal orthographié et elle est très proche de l'algorithme 2 que nous avons implémenté dans le deuxième étape de projet.

ArbreBK creer_ArbreBK(File * dico) :

Nous avons déclaré les variables nécessaires pour insérer les mots de fichier passé en paramètre dans un arbreBK en appelant la fonction **int inserer_dans_ArbreBK(ArbreBK * A, char * mot)** .

void liberer_ArbreBK(ArbreBK * A) :

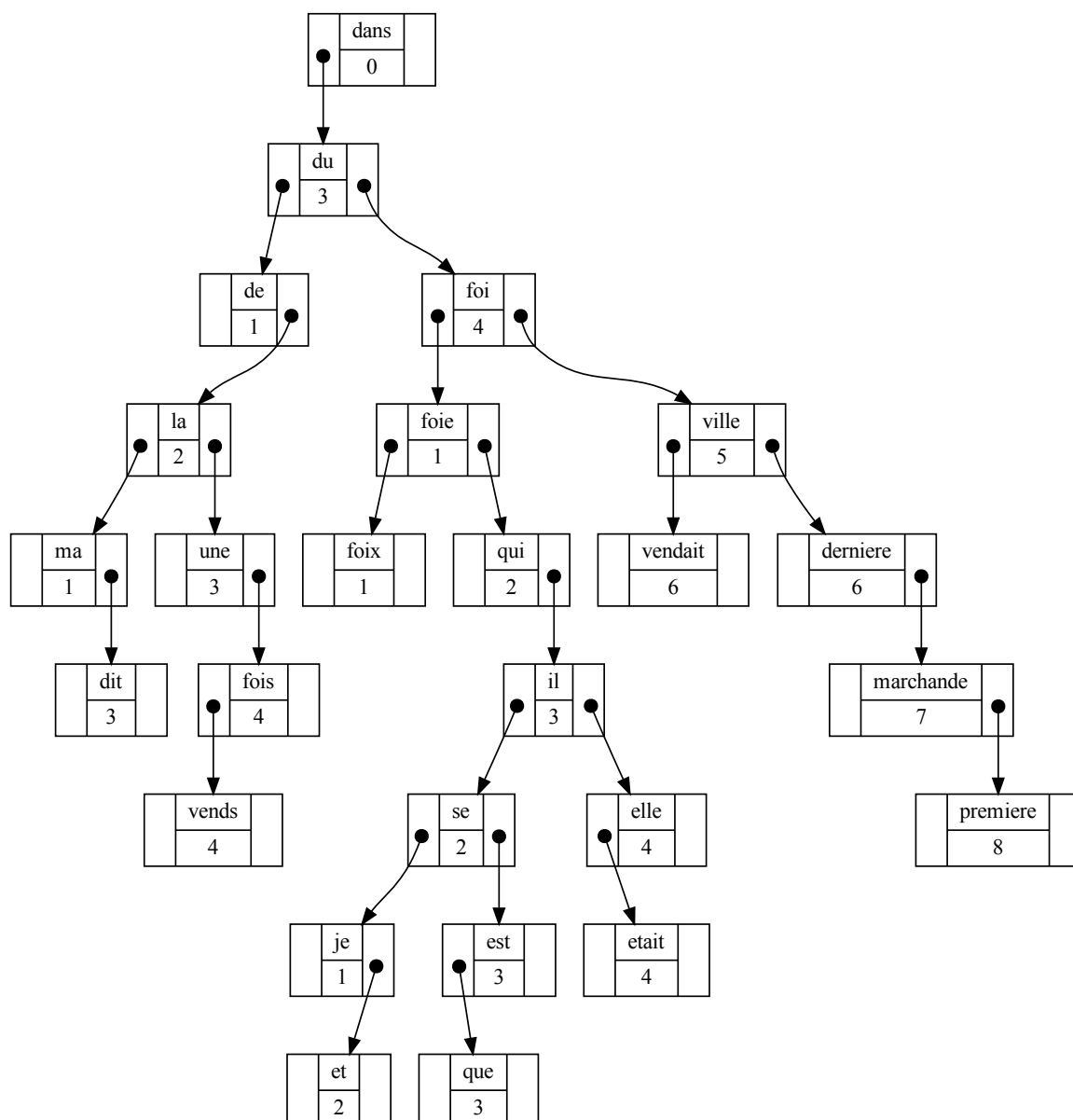
Nous avons implémenté la fonction récursivement :

Si le fils gauche est non nul nous faisons l'appel à la fonction. Si le frère droit est non nul nous faisons l'appel à la fonction, Ensuite nous libérons la racine par la fonction **free** et nous affectons **A** à la valeur nul.

_void afficher_ArbreBK(ArbreBK A) :

Nous avons affiché l'arbre par la représentation fils gauche-frère droit et en utilisant en utilisant **GraphViz** et le langage **.dot**.

On obtient l'arbre ceci :



NB : l'icône gauche représente le fils gauche et l'icône droite représente le frère droit de l'arbre .De plus le valeur apparue dans la case représente la valuation de l'arête liant le nœud courant à son parent.

Pour assurer si notre fonction d'insertion marche bien nous avons testé notre affichage sur une lexique plus petit donné dans l'énoncé de projet.

Lexique = {une, fois, foie, qui, ville, foix, elle, foi, est, que}

Et nous avons obtenu la même représentation fils gauche-frère droit

Arbre dans l'énoncé :

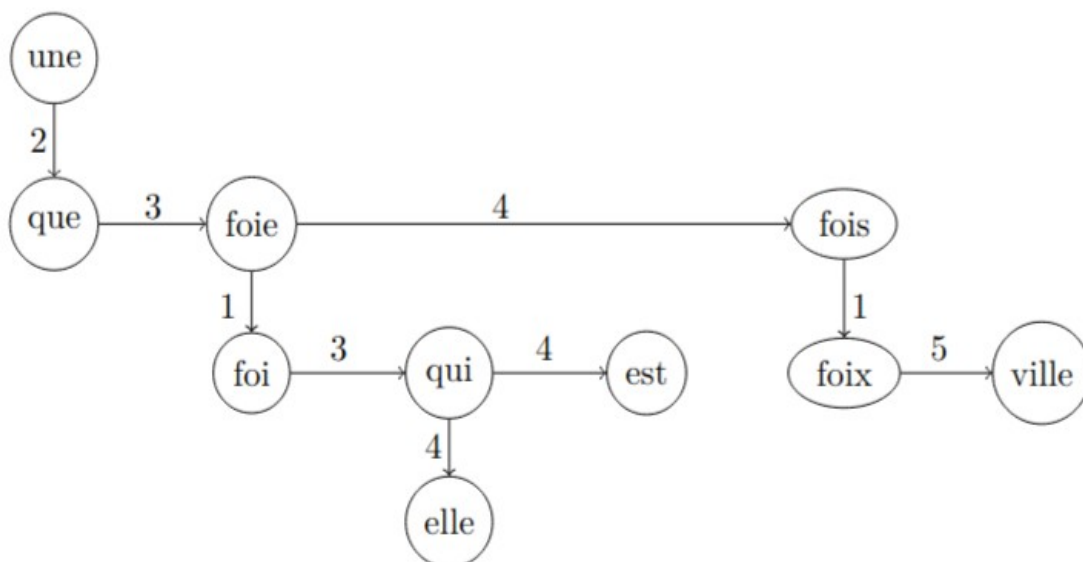
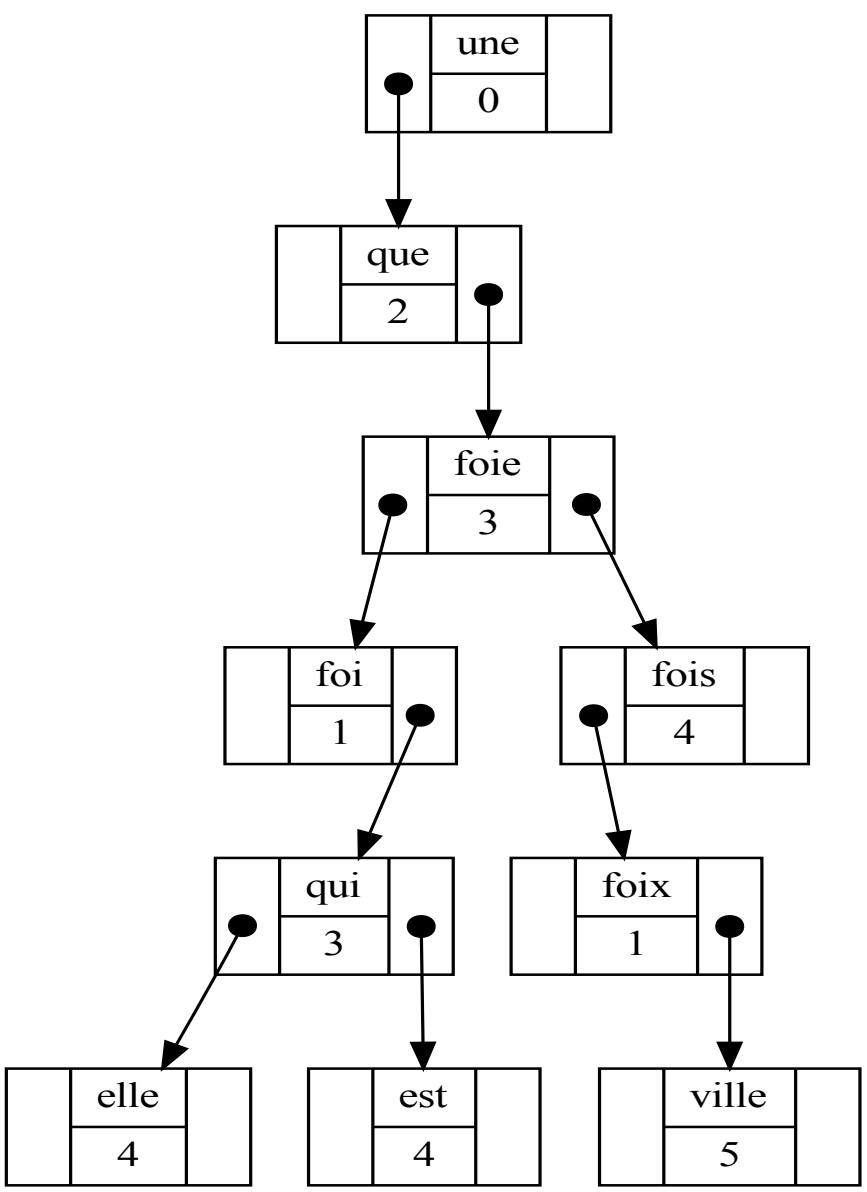


FIGURE 2 – L'arbre BK de la figure 1 sous forme d'arbre fils gauche / frères droits

Et notre représentation de l'arbre :



Explication des fonctions ajoutées:

`_int appartient_BK(ArbreBK A, char* mot) :`

Nous avons implémenté cette fonction pour afficher les mots qui ne sont pas dans l'arbre(mots mal orthographié) en calculant les distances Levenshtein entre les mots.

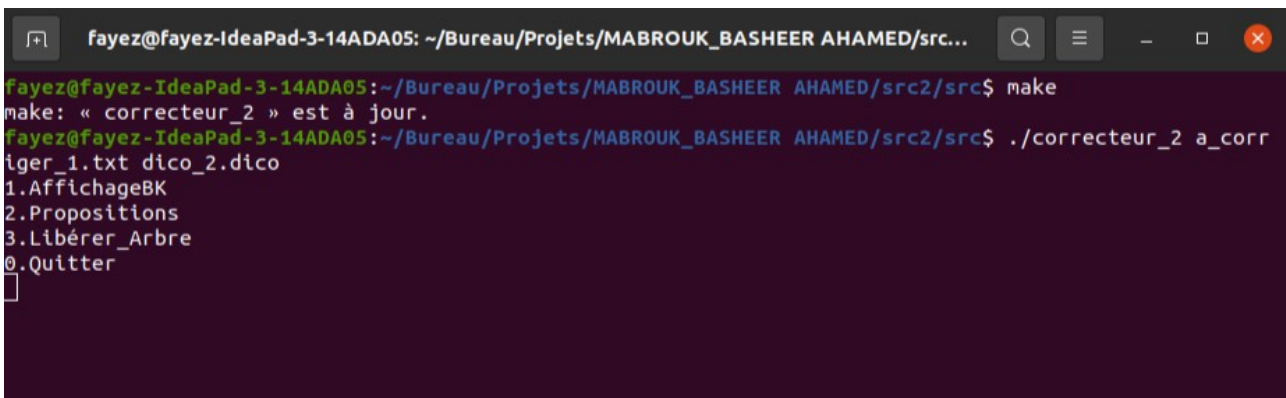
Si la distance de Levenshtein est nul alors le mot est déjà dans l'arbre sinon nous allons dans son fils gauche et si le fils gauche est nul alors le mot n'est pas dans l'arbre.

`_void affichage_proposition(ArbreBK dico, FILE* a_corriger, char *mot, Liste erreurs, Liste corrections) :`

Nous avons implémenté cette fonction pour afficher à chaque fois le mot mal orthographié (Liste erreurs) (en utilisant la fonction `_int appartient_BK(ArbreBK A, char* mot)`) et proposer les solutions les plus proches a ce mot en appelant la fonction `Liste rechercher_dans_ArbreBK(ArbreBK A, char * mot)` qui renvoie une liste de corrections Ensuite, nous libérons les deux listes pour avoir à chaque fois une seule mot mal orthographié et proposer les corrections associés .

Améliorations pour correcteur_2.c :

Nous avons implémenté des améliorations pour le fichier correcteur_2.c ,après avoir remplir l'arbre qui se fait automatiquement on aura 4 choix affichées dans le terminal :



```
fayez@fayez-IdeaPad-3-14ADA05: ~/Bureau/Projets/MABROUK_BASHEER AHAMED/src...
fayez@fayez-IdeaPad-3-14ADA05:~/Bureau/Projets/MABROUK_BASHEER AHAMED/src2/src$ make
make: « correcteur_2 » est à jour.
fayez@fayez-IdeaPad-3-14ADA05:~/Bureau/Projets/MABROUK_BASHEER AHAMED/src2/src$ ./correcteur_2 a_corr
iger_1.txt dico_2.dico
1.AffichageBK
2.Propositions
3.Libérer_Arbre
0.Quitter
□
```

En appuyant sur 2 ça affiche une fois les propositions :

Problème résolu : Nous avons réussi à implémenter l'affichage demandées que nous avons pas réussi pendant la deuxième étape (développer la fonction `_void afficher_Liste_corr(Liste L)` itérativement (qui affiche un virgule après le mot)


```
fayez@fayez-IdeaPad-3-14ADA05: ~/Bureau/Projets/MABROUK_BASHEER AHAMED/src...
fayez@fayez-IdeaPad-3-14ADA05:~/Bureau/Projets/MABROUK_BASHEER AHAMED/src2/src$ make
make: « correcteur_2 » est à jour.
fayez@fayez-IdeaPad-3-14ADA05:~/Bureau/Projets/MABROUK_BASHEER AHAMED/src2/src$ ./correcteur_2 a_corr
iger_1.txt dico_2.dico
1.AffichageBK
2.Propositions
3.Libérer_Arbre
0.Quitter
2
choix :2
mots mal orthographié : etais
Proposition(s) : etait
mots mal orthographié : foit
Proposition(s) : foix, foie, foi, fois
mots mal orthographié : doigt
Proposition(s) : dit
mots mal orthographié : vil
Proposition(s) : il
mots mal orthographié : faix
Proposition(s) : foix
mots mal orthographié : c
Proposition(s) : et, je, se, il, ma, la, de, du
mots mal orthographié : faix
Proposition(s) : foix
1.AffichageBK
2.Propositions
3.Libérer_Arbre
0.Quitter
```

Et que le 1 ça affiche l'arbreBK ,Le 3 ça le vide ,Et après avoir appuyé la 3 on peut éventuellement ré-appuyer 1 pour voir l'arbre vide ,Et après on est obligé de quitter en appuyant sur 0.

Guide d'utilisation

Pour pouvoir compiler le programme, il suffit de se déplacer au dossier **src** du programme et taper la commande **make** dans le terminal ce qui produira le fichier exécutable projet « correcteur_2 ».

```
fayez@fayez-IdeaPad-3-14ADA05:~$ make
```

Ensuite on exécute le programme principal avec le fichier texte contenant les mots à corriger et le dictionnaire .Comme ceci :

```
fayez@fayez-IdeaPad-3-14ADA05:~$ ./correcteur_2 a_corriger_1.txt dico_2.dico
```

Conclusion

Durant le développement de ce projet, nous avons beaucoup ré-appris sur l'arbreBK (les arbre n-aires de recherche)et sur la représentation fils gauche – frère droit.

Nous avons trouvé que l'étape 2 (arbre ternaire de recherche) est plus rapide que l'étape 3 (ArbreBK) vu que nous avons implémenté l'arbre ternaire de recherche

*d'une façon récursive et l'arbre **BK** d'une façon itérativement (la récursion c'est plus rapide pour les arbres). De plus, nous avons rencontré un problème quand nous testons avec le dico3 vu que nous avons implémenté la fonction insertion itérativement la complexité est très grande dans la version itérative par exemple pour 100 mots ça prend 100 secondes.*