INSA
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
ROUEN NORMANDIE

Fatiha HAMMOUCHE
Florine CHEVRIER
Loïck TOUPIN
Noé TATOUD

Correcteur Orthographique

Vous n'avez jamais aussi bien écrit !

Contents

1 Introduction

Dans le cadre de nos études dans la filière ITI à l'INSA de Rouen, nous avons réalisé un projet d'algorithmie en C. Le but de ce projet était de réaliser un correcteur orthographique. Ce projet est le premier que nous avons eu à réaliser du début à la fin en autonomie presque complète. Cela nous a permis de faire face à de nombreuses difficultés et ainsi de progresser dans de divers domaines.
En effet, ce projet a évidemment sollicité nos connaissances algorithmiques mais également notre capacité à travailler en groupe. Nous avons appris à nous organiser, mais aussi à mieux communiquer. Nous avons donc appris à nous adapter aux autres, et surtout à coder de façon claire et précise pour que nos collègues puissent comprendre ce que nous avions fait. Afin de faciliter la gestion de ce projet, nous devions utiliser Git dont nous nous étions déjà servi à d'autres occasions mais pour la plupart d'entre nous, nous ne le maîtrisions pas encore. L'utilisation de cette plateforme est donc une autre compétence essentielle que nous avons pu développer. Nous avons tous aussi progressé en C, qui est un langage que nous avons commencé à étudier au début de l'année, ainsi qu'à documenter notre code avec Doxygen. Nous avons également amélioré nos compétences en LaTeXque nous avons utilisé pour la rédaction du rapport.
Nous présentons donc dans ce rapport le résultat de notre travail, en commençant par l'analyse, puis la conception préliminaire et enfin la conception détaillée.

2 TAD

Nom: Mot
Utilise: **Chaine de caracteres,NaturelNonNul,Caractere,Booleen**
Opérations:

estUnMotValide: **Chaine de caracteres** $\nrightarrow$ **Booleen**

estUnCaractereAlphabetique: **Caractere** $\nrightarrow$ **Naturel**

copierMot: **Mot** $\rightarrow$ **Mot**

creerUnmot: **ChaineDeCaracteres** $\nrightarrow$ **Mot**

longueurMot: **Mot** $\rightarrow$ **NaturelNonNul**

obtenirChaine: **Mot** $\rightarrow$ **Chaine de Caractere**

iemeCaractere: **Mot** $\times$ **NaturelNonNul** $\nrightarrow$ **Caractere**

sontIdentiques: **Mot** $\times$ **Mot** $\rightarrow$ **Booléen**

fixerIemeCaractere: **Mot** $\times$ **NaturelNonNul** $\times$ **Caractere** $\nrightarrow$ **Mot**

fixerLongueur: **Mot** $\times$ **NaturelNonNul** $\rightarrow$ **Mot**

supprimerIemeLettre: **Mot** $\times$ **NaturelNonNul** $\nrightarrow$ **Mot**

inverserDeuxLettresConsecutives : **Mot** $\times$ **NaturelNonNul** $\nrightarrow$ **Mot**

insererLettre : **Mot** $\times$ **NaturelNonNul** $\times$ **Caractere** $\nrightarrow$ **Mot**

decomposerMot : **Mot** $\times$ **NaturelNonNul** $\nrightarrow$ **Mot** $\times$ **Mot**

reduireLaCasse : **Mot** $\nrightarrow$ **Mot**

supprimerMot : **Mot** $\rightarrow$

Sémantique:

creerUnMot : création d'un mot à partir d'une chaine de caractère.

estUnCaractereAlphabetique : verifie que le caractere est alphabetique.

estUnMotValide : renvoie un booleen qui indique si la chaine est composée de caractère alphabetique.

copierMot : permet de copier un mot.

obtenirChaine : renvoie la chaine du Mot.

longueurMot : donner la longueur d'un mot.

fixerLongueur : permet de fixer la longueur.

iemeCaractere : accéder au ieme caractere du mot.

fixerIemeCaractere : permet de fixer le ieme caractere.

sontIdentiques : vérifier si deux mots sont identiques.

remplacerIemeLettre : Remplace la ième lettre du mot par une autre lettre.

supprimerIemeLettre : Supprime la ième lettre du mot.

inverserDeuxLettresConsecutives : Inverse la lettre i et la lettre i+1.

insererLettre : Insère une lettre de l'alphabet entre la lettre i et la lettre i+1.

decomposerMot : Sépare le mot en deux parties, de part et d'autre de la lettre i.

reduireLaCasse : Change tous les caractères majuscules en minuscule.

supprimerMot : supprime le mot.

Préconditions:

creerUnMot(chaine): estUnMotValide(chaine)

estUnCaractereAlphabetique(c): longueur(c)==1

estUnMotValide(chaine): $1 \leq$ longueur(chaine)

fixerIemeCaractere(mot,i,c): $1 \leq i \leq$ longueur(mot) et estUnCaractereAlphabetique(c)

iemeCaractere(mot, i) : $1 \leq i \leq$ longueur(mot)

supprimerIemeLettre(mot, i) : $i \leq$ longueur(mot)

inverserDeuxLettresConsecutives(mot, i) : $i \leq$ longueur(mot)-1

insererLettre(mot, i) : $i \leq$ longueur(mot)+1

decomposerMot(mot, i) : $i \leq$ longueur(mot)

reduireLaCasse(chaine) : non(estVide(chaine))

Nom: Dictionnaire
Type Dictionnaire = ArbreDeLettres
Utilise : **Mot, FichierTexte, Ensemble<Mot>, Booléen**
Opérations:

genererArbreAvecEnsembleDeMot: **Ensemble<Mot>** $\nrightarrow$ **Dictionnaire**

estUnMotDuDictionnaire: **Dictionnaire** $\times$ **Mot** $\rightarrow$ **Booléen**

chargerDico : **FichierTexte**$\rightarrow$ **Dictionnaire**

sauvegarderDico: **Dictionnaire**$\rightarrow$ **FichierTexte**

Préconditions :

genererArbreAvecEnsembleDeMot(lesMots) : non estVide(lesMots)

Sémantique:

genererArbreAvecEnsembleDeMot : création d'un arbre représentant notre dictionnaire à l'aide d'un ensemble de mots

estUnMotDuDictionnaire : renvoie VRAI si le mot est dans le dictionnaire, FAUX sinon

chargerDico: recrée le dictionnaire sous forme d'arbre correspondant au fichier texte sauvegardé

sauvegarderDico : enregistre l'arbre sous forme de fichier texte

Nom: CorrecteurOrthographique
Utilise: **Mot, Dictionnaire, Ensemble<Mots>**
Opérations:

correcteur : **Dictionnaire** × **Mot** → **CorrecteurOrthographique**

obtenirMotACorriger : **CorrecteurOrthographique** → **Mot**

obtenirDictionnaire : **CorrecteurOrthographique** → **Dictionnaire**

obtenirCorrections : **CorrecteurOrthographique** → **Ensemble<Mots>**

fixerDico : **CorrecteurOrthographique** × **Dictionnaire** → **CorrecteurOrthographique**

fixerMotACorriger : **CorrecteurOrthographique** × **Mot** → **CorrecteurOrthographique**

ajouterNouvellesCorrections :
CorrecteurOrthographique × **Ensemble<Mot>** → **CorrecteurOrthographique**

trouverCorrectionsPossibles : **CorrecteurOrthographique** → **CorrecteurOrthographique**

remplacerIemeLettreEnBoucle : **Mot** × **Naturel** → **Ensemble<Mot>**

strategieRemplacerLettres : **CorrecteurOrthographique** → **CorrecteurOrthographique**

strategieSupprimerLettres : **CorrecteurOrthographique** → **CorrecteurOrthographique**

strategieInverserDeuxLettresConsecutives : **CorrecteurOrthographique** → **CorrecteurOrthographique**

insererIemeLettreEnBoucle : **Mot** × **Naturel** → **Ensemble<Mot>**

strategieInsererLettres : **CorrecteurOrthographique** → **CorrecteurOrthographique**

strategieDecomposerMot : **CorrecteurOrthographique** → **CorrecteurOrthographique**

Sémantique:

obtenirMotACorriger : Permet d'accèder au mot à corriger

obtenirDictionnaire : Permet d'accèder au dictionnaire

obtenirCorrections : Permet d'accèder aux corrections du mot

fixerDico : Donne un dictionnaire à utiliser au correcteur.

fixerMotACorriger : Donne un mot à corriger au correcteur.

ajouterNouvellesCorrections : Ajoute de nouvelles corrections du mot à corriger au correcteur.

trouverCorrectionsPossibles : Renvoie l'ensemble des corrections possibles du mot à corriger.

remplacerIemeLettreEnBoucle : Remplace une lettre du mot par toutes les autres de l'alphabet, une par une

strategieRemplacerLettres : Remplace toutes les lettres du mot par tous les caractères de l'alphabet tour à tour et ajoute les corrections valides au correcteur

strategieSupprimerLettres : Supprime les lettres du mot tour à tour et ajoute les corrections valides au correcteur

strategieInverserDeuxLettresConsecutives : Inverse les lettres du mot deux à deux, les unes après les autres et ajoute les corrections valides au correcteur

remplacerIemeLettreEnBoucle : Insère toutes les lettres de l'alphabet une par une à un endroit du mot

strategieInsererLettres : Insère un par un tous les caractères alphabétiques à tous les endroits du mot et ajoute les corrections valides au correcteur

strategieDecomposerMot : Décompose le mot en deux parties de toutes les façons possibles et ajoute les corrections valides au correcteur

Préconditions:

correcteur(unDico, unMotFaux) : non(estUnMotDuDictionnaire(unDico, unMotFaux)

fixerMotACorriger(unCorrecteur, unMotFaux) :
non(estUnMotDuDictionnaire(obtenirDictionnaire(unCorrecteur), unMotFaux))

Type Mode = {lecture,ecriture}
Nom: FichierTexte
Utilise: **Chaine de caracteres,Mode,Caractere,Booleen**
Opérations:

fichierTexte: **Chaine de caracteres** $\rightarrow$ **FichierTexte**

ouvrir: **FichierTexte** $\times$ **Mode** $\nrightarrow$ **Fichier**

fermer: **FichierTexte**$\nrightarrow$ **FichierTexte**

estOuvert: **FichierTexte**$\nrightarrow$ **Booleen**

mode: **FichierTexte** $\nrightarrow$ **Mode**

finFichier: **FichierTexte** $\nrightarrow$ **Booleen**

ecrireChaine: **FichierTexte** $\times$ **Chaine** $\nrightarrow$ **FichierTexte**

lireChaine: **FichierTexte** $\nrightarrow$ **FichierTexte** $\times$ **Chaine**

ecrireCaractere: **FichierTexte** $\times$ **Caractere** $\nrightarrow$ **FichierTexte**

lireCaractere: **FichierTexte** $\nrightarrow$ **FichierTexte** $\times$ **Caractere**

Sémantique:

fichierTexte: création d'un fichier texte à partir d'un fichier identifié par son nom.

ouvrir: ouvre un fichier texte en lecture ou écriture. Si le mode est écriture et que le fichier existe, alors ce dernier est écrasé.

fermer: ferme un fichier texte.

lireCaractere: lit un caractère à partir de la position courante du fichier.

lireChaine: lit une chaîne (jusqu'à un retour à la ligne ou la fin de fichier) à partir de la position courante du fichier.

ecrireCaractere: écrit un caractère à partir de la position courante du fichier.

ecrireChaine: écrit une chaîne suivie d'un retour à la ligne à partir de la position courante du fichier.

Préconditions:

ouvrir(f): non(estOuvert(f))

fermer(f): estOuvert(f)

finFichier(f): mode(f)=lecture

lireXX(f): estOuvert(f) et mode(f)=lecture et non finFichier(f)

ecrireXX(f): estOuvert(f) et mode(f)=ecriture

3 Analyses Descendantes

4 Conception des TAD

Mot

Conception préliminaire

Type Mot = Structure

chaine : ChaineDeCaractere

longueur : Naturel

finStructure

Signatures

fonction estUnMotValide(uneChaine : Chaine de Caractère) : Booleen

| **précondition:** 0 < longueur(chaine)

fonction estUnCaractereAlphabetique(unCaractere : Caractere) : Booleen

fonction copierMot(unMot : Mot) : Mot

fonction creerUnMot(uneChaine : Chaine de Caractère) : Mot

| **précondition:** estUnMotValide(uneChaine)

fonction longueurMot(unMot : Mot) : Naturel

fonction obtenirChaine(unMot : Mot) : Chaine de Caractère

fonction iemeCaractere(unMot : Mot, position : NaturelNonNul) : Caractere

| **précondition:** 1 ≤ position ≤ longueur(mot)

fonction sontIdentiques(unMot, unAutreMot : Mot) : Booleen

procedure fixerIemeCaractere(**E/S** unMot : Mot, **E** position : NaturelNonNul, c : Caractere)

| **précondition:** 1 ≤ position ≤ longueur(mot) et estUnCaractereAlphabetique(c)

procedure fixerLongueur(**E/S** unMot : Mot, **E** longueur : Naturel)

procedure supprimerIemeLettre(**E/S** unMot : Mot, **E** position : NaturelNonNul)

| **précondition:** position ≤ longueur(mot)

procedure inverserDeuxLettresConsecutives(**E/S** unMot : Mot, **E** position : NaturelNonNul)

| **précondition:** position ≤ longueur(mot)-1

procedure insererLettre(**E/S** unMot : Mot, **E** position : NaturelNonNul, c : Caractere)

| **précondition:** position ≤ longueur(mot)+1

procedure decomposerMot(**E/S** unMot : Mot, **E** position : NaturelNonNul)

| **précondition:** position ≤ longueur(mot)

procedure reduireLaCasse(**E/S** uneChaine : Chaine de Caractère)

fonction supprimerMot(unMot : Mot)

Conception détaillée

Fonction estUnMotValide(chaine : Chaine de Caractere):Booleen

Precondition(s) *0 < longueur(chaine)* **Declaration :** valide : Booleen
debut
| estValide ← VRAI i ← 1 **tant que** *(i ≤ longueur(chaine)) et (valide)* **faire**
| | c ← accederAuIemeCaractere(chaine,i) valide ← estUnCaractèreAlphabétique(c) i ← i+1
| **finTantQue**
| **retourner** *valide*
fin

Fonction creerUnMot(chaine : Chaine de Caractere):Mot

Precondition(s) *estUnMotValide(chaine)* **Declaration :** mot : Mot
debut
| mot.chaine ← chaine mot.longueur ← ChaineDeCaractere.longueur(chaine) **retourner** *mot*
fin

Fonction longueurMot(unMot : Mot):Naturel

Declaration :
debut
| **retourner** *unMot.longueur*
fin

Fonction iemeCaractere(unMot : Mot, position :NaturelNonNul):Caractere

Precondition(s) *1 ≤ position ≤ longueur(unMot)* **Declaration :**
debut
| **retourner** *ChaineDeCaractere.iemeCaractere(unMot.chaine,position)*
fin

Fonction sontIdentiques(unMot,unAutreMot : Mot):Booleen

Declaration : i : Naturel egaux : Booleen
debut
| **si** *longueur(unMot) ≠ longueur(unAutreMot)* **alors**
| | **retourner** *FAUX*
| **finsi**
| **sinon**
| | egaux ← VRAI i ← 1 **tant que** *(i ≤ longueur(unMot)) et (egaux)* **faire**
| | | egaux ← accederAuIemeCaractere(unMot,i) = accederAuIemeCaractere(unAutreMot,i) i ← i+1
| | **finTantQue**
| | **retourner** *egaux*
| **finsi**
fin

Procédure supprimerIemeLettre(**E/S** unMot : Mot, **E** position : NaturelNonNul)

Precondition(s) *position ≤ longueur(mot)* **Declaration :** indice : Naturel
debut
| **tant que** *indice < longueurMot(unMot)* **faire**
| | fixerIemeCaractere(unMot,indice,iemeCaractère(unMot,position+1)) indice ← indice + 1
| **finTantQue**
| fixerLongueur(unMot,longueurMot(unMot)-1)
fin

Procédure inverserDeuxLettresConsecutives(**E/S** unMot : Mot, **E** position : NaturelNonNul)

Precondition(s) *1 ≤ position ≤ longueur(mot)-1* **Declaration :** temp : Caractere
debut
| c ← iemeCaractère(unMot,position) fixerIemeCaractere(unMot,position,iemeCaractere(unMot,position+1))
| fixerIemeCaractere(unMot,position+1,temp);
fin

Procédure insererLettre(**E/S** unMot : Mot, **E** position : NaturelNonNul, c : Caractere)

Precondition(s) *1* $\leq$ *position* $\leq$ *longueur(mot)+1* **Declaration :** i : Naturel
pour *i* $\leftarrow$ *longueur(unMot.chaine) à position* **faire**
| fixerIemeCaractere(unMot,i+1,iemeCaractere(unMot,i))
finPour
fixerLongueur(unMot,longueurMot(unMot)+1) fixerIemeCaractere(unMot,position,c)

Procédure decomposerMot(**E/S** unMot : Mot, **E** position : NaturelNonNul **S** motGauche : Mot)

Precondition(s) *2* $\leq$ *position* $\leq$ *longueur(mot)* **Declaration :** chaineGauche : Chaine De Caractere i : Naturel
pour *i* $\leftarrow$ *1 à position* **faire**
| ChaineDeCaractere.fixerIemeCaractere(chaineGauche,i,iemeCaractere(unMot,1)) supprimerIemeLettre(unMot,1)
finPour
motGauche $\leftarrow$ creerUnMot(chaineGauche)

Dictionnaire

Conception préliminaire

Signatures

fonction genererArbreAvecEnsembleDeMot(lesMots : Ensemble<Mot>): ArbreDeLettres

| **précondition:** non estVide(lesMots)

fonction estUnMotDuDictionnaire(unArbre : ArbreDeLettres, unMot : Mot):Booléen

fonction chargerDico(unDictionnaire : Dictionnaire) :ArbreDeLettres

fonction sauvegarderArbreEnDictionnaire(unArbre : Arbre): Dictionnaire

Conception détaillée

Fonction genererArbreAvecEnsembleDeMot(lesMots : Ensemble<Mot>): ArbreDeLettres

Precondition(s) *non estVide(lesMots)*
Declaration :
unArbre : ArbreDeLettres
i : Entier
debut
| **pour** *i <- 1* ***allant à*** *longueur(lesMots)* **faire**
| | insererMotALaBonnePlace(unArbre, obtenirElement(lesMots, i))
| **finPour**
| **retourner** *unArbre*
fin

Fonction chargerDico(unDictionnaire : Dictionnaire) :ArbreDeLettres

Declaration :
debut
| **retourner**
fin

Fonction estUnMotDuDictionnaire(unArbre : ArbreDeLettres, unMot : Mot):Booléen

```
Declaration :
temp : ArbreDeLettres
debut
    si longueur(unMot) = 1 alors
        retourner non(estVide(unArbre) ou non(obtenirBooleen(unArbre))
    finsi
    sinon
        si non estVide(unArbre) alors
            si iemeCaractere(unMot,1) = obtenirLettre(unArbre) alors
                unMot <- supprimerIemeLettre(unMot, 1)
                temp <- obtenirFilsGauche(unArbre)
                retourner estUnMotDuDictionnaire(unMot, temp)
            finsi
            sinon
                temp <- obtenirFrere(unArbre)
                retourner estUnMotDuDictionnaire(unMot, temp)
            finsi
        finsi
        sinon
            retourner FAUX
        finsi
    finsi
fin
```

Fonction sauvegarderArbreEnDictionnaire(unArbre : Arbre): Dictionnaire

```
Declaration :
debut
    retourner
fin
```

Correcteur Orthographique

Conception préliminaire

Type CorrecteurOrthographique = Structure

motACorriger : ChaineDeCaractere

leDictionnaire : Naturel

lesCorrections : EnsembleDeMot

finStructure

Signatures

fonction correcteur(unDico : Dictionnaire, unMotFaux : unMot): CorrecteurOrthographique

| **précondition:** non(estUnMotDuDictionnaire(unDico, unMotFaux))

fonction obtenirMotACorriger(unCorrecteur : CorrecteurOrthographique): Mot

fonction obtenirDictionnaire(unCorrecteur : CorrecteurOrthographique): Dictionnaire

procedure fixerDico(**E/S** unCorrecteur : CorrecteurOrthographique,**E**)

procedure fixerMotACorriger(**E/S** unCorrecteur: CorrecteurOrthographique,**E** unMotFaux : Mot)

| **précondition:** non(estUnMotDuDictionnaire(obtenirDictionnaire(unCorrecteur), unMotFaux))

procedure fixerDico(**E/S** unCorrecteur : CorrecteurOrthographique, **E** unDico : Dictionnaire)

procedure ajouterNouvellesCorrections(**E/S** cor : CorrecteurOrthographique, **E** corrections : Ensemble<Mot>)

procedure trouverCorrectionsPossibles(**E/S** unCorrecteur : CorrecteurOrthographique)

fonction remplacerIemeLettreEnBoucle(unMot : Mot, indice : Naturel) : Ensemble<Mot>

procedure strategieRemplacerLettres(**E/S** unCorrecteur : CorrecteurOrthographique)

procedure strategieSupprimerLettres(**E/S** unCorrecteur : CorrecteurOrthographique)

procedure strategieInverserDeuxLettresConsecutives(**E/S** unCorrecteur : CorrecteurOrthographique)

fonction insererIemeLettreEnBoucle(unMot : Mot, indice : Naturel) : Ensemble<Mot>

procedure strategieInsererLettres(**E/S** unCorrecteur : CorrecteurOrthographique)

procedure strategieDecomposerMot(**E/S** unCorrecteur : CorrecteurOrthographique)

Conception détaillée

Procédure remplacerIemeLettre(**E/S** mot : Mot,**E** c : Caractere, **E** position : naturel)

Declaration :
debut
    supprimerIemeLettre(mot,position);
    insererLettre(mot,position,caractere);
fin

Procédure supprimerIemeLettre(**E/S** unMot, **E** position : Naturel)

Precondition(s) *longueur(mot)≥1 et i≤longueur*
Declaration :
debut
    ChaineDeCaractere.supprimerIemeLettre(unMot.chaine,position);
    unMot.longueur ← unMot.longueur-1 ;
fin

Procédure inverserDeuxLettresConsecutives(**E/S** unMot : Mot, **E** i : NaturelNonNul)

Precondition(s) *longueur(mot)≥1 et i≤longueur*
Declaration : c1,c2 : Caractere
debut
    c1 ← Mot.accederAuIemeCaractere(unMot,i);
    c2 ← Mot.accederAuIemeCaractere(unMot,i+1);
    remplacerIemeLettre(unMot, c2, i);
    remplacerIemeLettre(unMot, c1, i+1);
fin

Procédure insererLettre(**E/S** unMot : Mot, **E** position : Naturel, c : Caractere)

Precondition(s) *i≤ longueur(mot)*
Declaration :
debut
    ChaineDeCaratere.insererLettre(unMot.chaine,position,c);
    unMot.longueur ← unMot.longueur + 1;
fin

Fonction decomposerMot(unMot : Mot, position : NaturelNonNul) : Mot, Mot

Precondition(s) $i \leq$ *longueur(mot)*
Declaration : mot1, mot2 : Mot, c : Caractere, chaine1, chaine2 : ChaineDeCaractere
debut
  mot1 ← creerUnMot("");
  mot2 ← creerUnMot("");
  pour *i ← 1 à position-1* **faire**
    c ← Mot.accederAuIemeCaractere(unMot,i);
    insererLettre(mot1, i, c);
  finPour
  pour *i ← position à unMot.longueur* **faire**
    c ← Mot.accederAuIemeCaractere(unMot,i);
    insererLettre(mot2, i, c);
  finPour
  retourner *mot1, mot2*
fin

FichierTexte

Conception préliminaire

Structure

Type FichierTexte = Structure

fichier : Fichier

mode : Mode

finStructure

Signatures

fonction fichierTexte(chaine : Chaine de Caractère):FichierTexte

procédure ouvrir(E/S fichier:FichierTexte,E mode : Mode)

| **précondition:** non estOuvert(f)

procédure fermer(E/S fichier:FichierTexte)

| **précondition:** estOuvert(f)

fonction estOuvert(fichier:FichierTexte):Booleen

fonction mode(fichier:FichierTexte) : Mode

fonction finFichier(fichier:FichierTexte):Booleen

| **précondition:** mode(f)=lecture

procédure ecrireChaine(E/S fichier:FichierTexte, E chaine:Chaine de Caractère)

| **précondition:** estOuvert(f) et mode(f)=ecriture

procédure lireChaine(E/S fichier:FichierTexte, S chaine:Chaine de Caractère)

| **précondition:** estOuvert(f) et mode(f)=lecture et non finFichier(f)

procédure ecrireCaractere(E/S fichier:FichierTexte, E caractere:Caractère)

| **précondition:** estOuvert(f) et mode(f)=ecriture

procédure lireCaractere(E/S fichier:FichierTexte, S caractere:Caractère)

| **précondition:** estOuvert(f) et mode(f)=lecture et non finFichier(f)

Conception détaillée

5 Code C

```c
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include "ArbreDeLettres.h"

ArbreDeLettres ADL_creerADLVide(){
    errno = 0;
    return NULL;
}

int ADL_estVide(ArbreDeLettres arbre){
    errno = 0;
    return (arbre == NULL);
}

ArbreDeLettres ADL_creerADL(ArbreDeLettres fils, ArbreDeLettres frere, char lettre, int estUneFin){
    ArbreDeLettres arbre = (ArbreDeLettres)malloc(sizeof(ADL));
    arbre->fils = fils;
    arbre->frere = frere;
    arbre->estFinDeMot = estUneFin;
    arbre->lettre = lettre;
    return arbre;
}

void ADL_fixerElement(ArbreDeLettres *arbre, char c, int estUneFin){
    (*arbre)->estFinDeMot=estUneFin;
    (*arbre)->lettre=c;
}

void ADL_fixerEstFinDeMot(ArbreDeLettres *arbre, int estUneFin){
    assert(!ADL_estVide(*arbre));
    errno = 0;
    (*arbre)->estFinDeMot = estUneFin;
}

void ADL_fixerLettre(ArbreDeLettres *arbre, char lettre){
    assert(!ADL_estVide(*arbre));
    errno = 0;
    (*arbre)->lettre = lettre;
}

void ADL_fixerFrere(ArbreDeLettres *arbre, ArbreDeLettres frere){
    assert(!ADL_estVide(*arbre));
    errno = 0;
    (*arbre)->frere = frere;
}

void ADL_fixerFils(ArbreDeLettres *arbre, ArbreDeLettres fils){
    assert(!ADL_estVide(*arbre));
    errno = 0;
    (*arbre)->fils = fils;
```

```
52  }
53
54  ArbreDeLettres ADL_obtenirFils(ArbreDeLettres arbre){
55      assert(!ADL_estVide(arbre));
56      errno = 0;
57      return arbre->fils;
58  }
59
60  ArbreDeLettres ADL_obtenirFrere(ArbreDeLettres arbre){
61      assert(!ADL_estVide(arbre));
62      errno = 0;
63      return arbre->frere;
64  }
65
66  char ADL_obtenirLettre(ArbreDeLettres arbre){
67      assert(!ADL_estVide(arbre));
68      errno = 0;
69      return arbre->lettre;
70  }
71
72  int ADL_obtenirEstFinDeMot(ArbreDeLettres arbre){
73      assert(!ADL_estVide(arbre));
74      errno = 0;
75      return arbre->estFinDeMot;
76  }
77
78
79  void ADL_supprimer(ArbreDeLettres *arbre){
80      ArbreDeLettres tmp = ADL_creerADLVide();
81      if (!ADL_estVide(*arbre)){
82          tmp=ADL_obtenirFils(*arbre);
83          ADL_supprimer(&tmp);
84          tmp=ADL_obtenirFrere(*arbre);
85          ADL_supprimer(&tmp);
86      }
87      free(*arbre);
88
89  }
```

../programme/src/ArbreDeLettres.c

```
1   #include <stdlib.h>
2   #include <assert.h>
3   #include <string.h>
4   #include "CorrecteurOrthographique.h"
5   #include "Mot.h"
6   #include "EnsembleDeMot.h"
7   #include "Dictionnaire.h"
8
9   CorrecteurOrthographique CO_correcteur(Dictionnaire unDico, Mot unMotFaux){
10      assert(!D_estUnMotDuDictionnaire(unDico, M_copierMot(unMotFaux)));
11      CorrecteurOrthographique unCorrecteur;
12      unCorrecteur.leDictionnaire = unDico;
13      unCorrecteur.motACorriger = M_copierMot(unMotFaux);
```

```
14      unCorrecteur.lesCorrections = ensembleDeMot();
15      return unCorrecteur;
16  }
17
18  Mot CO_obtenirMotACorriger(CorrecteurOrthographique unCorrecteur){
19      return unCorrecteur.motACorriger;
20  }
21
22  Dictionnaire CO_obtenirDictionnaire(CorrecteurOrthographique unCorrecteur){
23      return unCorrecteur.leDictionnaire;
24  }
25
26  EnsembleDeMot CO_obtenirCorrections(CorrecteurOrthographique unCorrecteur){
27      return unCorrecteur.lesCorrections;
28  }
29
30  void CO_fixerDico(CorrecteurOrthographique* unCorrecteur, Dictionnaire unDico){
31      unCorrecteur->leDictionnaire = unDico;
32  }
33
34  void CO_fixerMotACorriger(CorrecteurOrthographique* unCorrecteur, Mot unMotFaux){
35      assert(!D_estUnMotDuDictionnaire(CO_obtenirDictionnaire(*unCorrecteur), unMotFaux));
36      unCorrecteur->motACorriger = unMotFaux;
37  }
38
39  void CO_ajouterNouvellesCorrections (CorrecteurOrthographique* unCorrecteur, EnsembleDeMot
        desCorrections){
40      EnsembleDeMot temp = unCorrecteur->lesCorrections;
41      unCorrecteur->lesCorrections = EDM_union(desCorrections, temp);
42      EDM_vider(&temp);
43  }
44
45  void CO_trouverCorrectionsPossibles(CorrecteurOrthographique* unCorrecteur){
46      CO_strategieRemplacerLettres(unCorrecteur);
47      CO_strategieSupprimerLettres(unCorrecteur);
48      CO_strategieInsererLettres(unCorrecteur);
49      CO_strategieInverserDeuxLettresConsecutives(unCorrecteur);
50      CO_strategieDecomposerMot(unCorrecteur);
51  }
52
53  EnsembleDeMot CO_remplacerIemeLettreEnBoucle(Mot motACorriger, int i){
54      EnsembleDeMot desCorrections;
55      Mot uneCorrection;
56      desCorrections = ensembleDeMot();
57      char* lettres;
58      lettres = "abcdefghijklmnopqrstuvwxyzàéèëùûêîïçôö-";
59      for(int j = 0; j < strlen(lettres);j++ ){
60          uneCorrection = M_copierMot(motACorriger);
61          M_fixerIemeCaractere(&uneCorrection, i, lettres[j]);
62          EDM_ajouter(&desCorrections, uneCorrection);
63      }
64      return desCorrections;
65  }
66
```

```
67  void CO_strategieRemplacerLettres(CorrecteurOrthographique* unCorrecteur){
68      unsigned int i, longueur;
69      Mot uneCorrection;
70
71      Mot leMotACorriger = CO_obtenirMotACorriger(*unCorrecteur);
72      Dictionnaire leDico = CO_obtenirDictionnaire(*unCorrecteur);
73      longueur = M_longueurMot(leMotACorriger);
74      EnsembleDeMot desCorrections = ensembleDeMot();
75      EnsembleDeMot correctionsCourantes = ensembleDeMot();
76      for(i = 1; i < longueur +1; i++){
77          correctionsCourantes = CO_remplacerIemeLettreEnBoucle(leMotACorriger, i);
78
79          while(EDM_cardinalite(correctionsCourantes)!=0){
80              uneCorrection = EDM_obtenirMot(correctionsCourantes);
81              EDM_retirer(&correctionsCourantes, uneCorrection);
82              if(D_estUnMotDuDictionnaire(leDico, M_copierMot(uneCorrection)) && !EDM_estPresent(
    desCorrections, uneCorrection))
83                  EDM_ajouter(&desCorrections,uneCorrection);
84              else
85                  M_supprimerMot(&uneCorrection);
86
87          }
88          CO_ajouterNouvellesCorrections(unCorrecteur, desCorrections);
89
90          EDM_vider(&correctionsCourantes);
91          EDM_vider(&desCorrections);
92      }
93  }
94
95  void CO_strategieSupprimerLettres(CorrecteurOrthographique* unCorrecteur){
96      unsigned int i, longueur;
97      Mot uneCorrection;
98
99      Mot leMotACorriger = CO_obtenirMotACorriger(*unCorrecteur);
100     Dictionnaire leDico = CO_obtenirDictionnaire(*unCorrecteur);
101     longueur = M_longueurMot(leMotACorriger);
102     EnsembleDeMot desCorrections = ensembleDeMot();
103
104     for(i = 1; i < longueur; i++){
105         uneCorrection = M_copierMot(leMotACorriger);
106         M_supprimerIemeLettre(&uneCorrection, i);
107         if(D_estUnMotDuDictionnaire(leDico, M_copierMot(uneCorrection)) && !EDM_estPresent(
    desCorrections, uneCorrection))
108             EDM_ajouter(&desCorrections, uneCorrection);
109         else
110             M_supprimerMot(&uneCorrection);
111
112     }
113     CO_ajouterNouvellesCorrections(unCorrecteur, desCorrections);
114     EDM_vider(&desCorrections);
115 }
116
117 void CO_strategieInverserDeuxLettresConsecutives(CorrecteurOrthographique* unCorrecteur){
118     unsigned int i, longueur;
```

```
119     Mot uneCorrection;
120
121     Mot leMotACorriger = CO_obtenirMotACorriger(*unCorrecteur);
122     Dictionnaire leDico = CO_obtenirDictionnaire(*unCorrecteur);
123     longueur = M_longueurMot(leMotACorriger);
124     EnsembleDeMot desCorrections = ensembleDeMot();
125
126     for(i = 1; i < longueur-1; i++){
127         uneCorrection = M_copierMot(leMotACorriger);
128         M_inverserDeuxLettresConsecutives(&uneCorrection,i);
129         if(D_estUnMotDuDictionnaire(leDico, M_copierMot(uneCorrection)) && !EDM_estPresent(
    desCorrections, uneCorrection))
130             EDM_ajouter(&desCorrections, uneCorrection);
131         else{
132             M_supprimerMot(&uneCorrection);
133         }
134     }
135     CO_ajouterNouvellesCorrections(unCorrecteur, desCorrections);
136         EDM_vider(&desCorrections);
137 }
138
139 EnsembleDeMot CO_insererIemeLettreEnBoucle(Mot motACorriger, int i){
140     EnsembleDeMot desCorrections;
141     Mot uneCorrection;
142     desCorrections = ensembleDeMot();
143     char* lettres;
144     lettres = "abcdefghijklmnopqrstuvwxyzàéèëùûêîïçôö-";
145     for(int j = 0; j < strlen(lettres);j++ ){
146         uneCorrection = M_copierMot(motACorriger);
147         M_insererLettre(&uneCorrection, i, lettres[j]);
148         EDM_ajouter(&desCorrections, uneCorrection);
149     }
150     return desCorrections;
151 }
152
153 void CO_strategieInsererLettres(CorrecteurOrthographique* unCorrecteur){
154     unsigned int i, longueur;
155     Mot uneCorrection;
156
157     Mot leMotACorriger = CO_obtenirMotACorriger(*unCorrecteur);
158     Dictionnaire leDico = CO_obtenirDictionnaire(*unCorrecteur);
159     longueur = M_longueurMot(leMotACorriger);
160     EnsembleDeMot desCorrections = ensembleDeMot();
161     EnsembleDeMot correctionsCourantes = ensembleDeMot();
162     for(i = 1; i < longueur +1; i++){
163         correctionsCourantes = CO_insererIemeLettreEnBoucle(leMotACorriger, i);
164
165         while(EDM_cardinalite(correctionsCourantes)!=0){
166             uneCorrection = EDM_obtenirMot(correctionsCourantes);
167             EDM_retirer(&correctionsCourantes, uneCorrection);
168             if(D_estUnMotDuDictionnaire(leDico, M_copierMot(uneCorrection)) && !EDM_estPresent(
    desCorrections, uneCorrection))
169                 EDM_ajouter(&desCorrections,uneCorrection);
170             else
```

```
171                     M_supprimerMot(&uneCorrection);
172             }
173             CO_ajouterNouvellesCorrections(unCorrecteur, desCorrections);
174 
175             EDM_vider(&correctionsCourantes);
176             EDM_vider(&desCorrections);
177         }
178     }
179 
180     void CO_strategieDecomposerMot(CorrecteurOrthographique* unCorrecteur){
181         unsigned int i, longueur;
182         Mot leMotACorriger = CO_obtenirMotACorriger(*unCorrecteur);
183         Mot uneCorrection;
184         Mot unMotModifiable;
185         Dictionnaire leDico = CO_obtenirDictionnaire(*unCorrecteur);
186         longueur = M_longueurMot(leMotACorriger);
187         EnsembleDeMot desCorrections = ensembleDeMot();
188 
189         for(i = 2; i < longueur-1; i++){
190             unMotModifiable = M_copierMot(leMotACorriger);
191             uneCorrection = M_decomposerMot(&unMotModifiable, i);
192             if(D_estUnMotDuDictionnaire(leDico, M_copierMot(uneCorrection)) && D_estUnMotDuDictionnaire(
        leDico, M_copierMot(unMotModifiable))
193                 && !EDM_estPresent(desCorrections, unMotModifiable) && !EDM_estPresent(desCorrections,
        uneCorrection)){
194                 EDM_ajouter(&desCorrections, uneCorrection);
195                 EDM_ajouter(&desCorrections, unMotModifiable);
196             }
197             else{
198                 M_supprimerMot(&uneCorrection);
199                 M_supprimerMot(&unMotModifiable);
200             }
201         }
202         CO_ajouterNouvellesCorrections(unCorrecteur, desCorrections);
203         EDM_vider(&desCorrections);
204     }
205 
206     void CO_supprimerCorrecteur(CorrecteurOrthographique *unCorrecteur){
207         Mot unMot;
208         M_supprimerMot(&unCorrecteur->motACorriger);
209 
210         while(EDM_cardinalite(unCorrecteur->lesCorrections) != 0){
211             unMot = EDM_obtenirMot(unCorrecteur->lesCorrections);
212             EDM_retirer(&unCorrecteur->lesCorrections, unMot);
213             M_supprimerMot(&unMot);
214         }
215 
216 
217         ADL_supprimer(&unCorrecteur->leDictionnaire);
218 
219     }
```

../programme/src/CorrecteurOrthographique.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include "Dictionnaire.h"
#include "CorrecteurOrthographique.h"
#include "Mot.h"
#include "EnsembleDeMot.h"


void corrigerTexte(char* chaine,Dictionnaire dico){
    int indiceDebutMot=0;
    for(int position=0,position<strlen(chaine),position++){
        if(CT_estUnSeparateur(chaine[position])){
            if(position>indiceDebutMot){
                CT_trouverEtAfficherCorrection(chaine, indiceDebutMot,position,dico);
            }
            else{}
            indiceDebutMot=position+1;
        }
        else{}
    }
}



int CT_estUnSeparateur(char c){
    char* apostrophe = malloc(sizeof(char));
    apostrophe = "'";
    return (c==' ')||(c==',')||(c=='.')||(c=='?')||(c==';')||(c=='!')||(c==':')||(c=='\0')||(c==
    apostrophe[0]);
}

char* CT_creerSousChaine(char* chaine,unsigned int gauche,unsigned int droite){
    assert((gauchex<=droite) && (droite<strlen(chaine)));
    char* sousChaine=malloc((droite-gauche+2)*sizeof(char));
    memcpy(sousChaine,&chaine[gauche],droite-gauche+1);
    sousChaine[droite-gauche+1] = '\0';
    return sousChaine;
}

CorrecteurOrthographique CT_trouverCorrections(Dictionnaire dico,Mot unMot){
    CorrecteurOrthographique correcteur = CO_correcteur(dico,unMot);
    CO_trouverCorrectionsPossibles(&correcteur);
    return correcteur;
}

void CT_afficherCorrection(int indiceDebutMot,CorrecteurOrthographique correcteur){
    printf("& %s %i %i :", unMot.chaine, indiceDebutMot,EDM_cardinalite(CO_obtenirCorrections(correcteur
    )));
    if(EDM_cardinalite(CO_obtenirCorrections(correcteur))==0){
        printf(" trop d'erreurs dans ce mot")
    }
```

```
52      else{
53          for(i=1, i<=EDM_cardinalite(CO_obtenirCorrections(correcteur)),i++){
54          printf(" %s", M_obtenirChaine(EDM_obtenirMot(CO_obtenirCorrections(correcteur))));
55      }
56      printf('\n');
57      }
58  }
59
60  void CT_trouverEtAfficherCorrection(char* chaine, int indiceDebutMot,int position,Dictionnaire dico){
61      char* sousChaine = malloc((position-1-indiceDebutMot));
62      sousChaine = CT_creerSousChaine(chaine,indiceDebutMot,position-1); //tester souschaine avec un seul
        car
63      Mot unMot = M_creerUnMot(sousChaine);
64      free(sousChaine);
65      if(!D_estUnMotDuDictionnaire(dico,unMot)){
66          CorrecteurOrthographique correcteur = CT_trouverCorrections(dico, unMot);
67          CT_afficherCorrection(indiceDebutMot,correcteur);
68      }
69      else{
70          printf("*\n");
71      }
72  }
```

../programme/src/corrigerTexte.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <assert.h>
4   #include <string.h>
5   #include "Dictionnaire.h"
6   #include "ArbreDeLettres.h"
7   #include "Mot.h"
8   #include "FichierTexte.h"
9   #define NB_MOTS_DICTIONNAIRE 350000
10  #define LONGUEUR_MAX_MOT 27
11
12  /* Partie privée */
13
14  void D_insererMot(Dictionnaire* unDico, Mot unMot){
15      Dictionnaire temp;
16      int enFinDeMot = 0;
17      if(M_longueurMot(unMot) == 1){
18          enFinDeMot = 1;
19          if(ADL_estVide(*unDico)){
20              D_insererLettre(unDico, M_iemeCaractere(unMot, 1), enFinDeMot);
21          }
22          else{
23              if(D_lettreEstRacine(*unDico, M_iemeCaractere(unMot, 1))){
24                  ADL_fixerEstFinDeMot(unDico, enFinDeMot);
25              }
26              else{
27                  temp = ADL_obtenirFrere(*unDico);
28                  D_insererMot(&temp, unMot);
29                  ADL_fixerFrere(unDico, temp);
```

```
30                  }
31              }
32          }
33          else{
34              if(ADL_estVide(*unDico)){
35                  D_insererLettre(unDico, M_iemeCaractere(unMot, 1), enFinDeMot);
36                  M_supprimerIemeLettre(&unMot, 1);
37                  temp = ADL_obtenirFils(*unDico);
38                  D_insererMot(&temp, unMot);
39                  ADL_fixerFils(unDico, temp);
40              }
41              else{
42                  if(D_lettreEstRacine(*unDico, M_iemeCaractere(unMot, 1))){
43                      M_supprimerIemeLettre(&unMot, 1);
44                      temp = ADL_obtenirFils(*unDico);
45                      D_insererMot(&temp, unMot);
46                      ADL_fixerFils(unDico, temp);
47                  }
48                  else{
49                      temp = ADL_obtenirFrere(*unDico);
50                      D_insererMot(&temp, unMot);
51                      ADL_fixerFrere(unDico, temp);
52                  }
53
54              }
55
56          }
57      }
58
59      void D_insererLettre(Dictionnaire* unDico, char uneLettre, int estFinDeMot){
60          assert(ADL_estVide(*unDico));
61          *unDico = ADL_creerADL(NULL, NULL, uneLettre, estFinDeMot);
62
63      }
64
65      int D_lettreEstRacine(Dictionnaire unDico, char uneLettre){
66          return ADL_obtenirLettre(unDico) ==  uneLettre;
67      }
68
69      Mot* supprimerLesMots(Mot *lesMots, int nbMots){
70          Mot* lesMotsASupprimer = lesMots;
71
72          int i = 0;
73          while(i < nbMots){
74              M_supprimerMot(&lesMotsASupprimer[i]);
75              i++;
76          }
77
78          return lesMotsASupprimer;
79      }
80
81      void supprimerTabMots(Mot **lesMots, int nbMots){
82          *lesMots = supprimerLesMots(*lesMots, nbMots);
83          free(*lesMots);
```

```
84  }
85
86  Mot* D_genererTableauDeMotAvecFichierTexte(FichierTexte ficDico, int *nbMots){
87      Mot* lesMots=(Mot*)malloc(((sizeof(char)*27)+sizeof(int))*NB_MOTS_DICTIONNAIRE);
88
89      FT_ouvrir(&ficDico, LECTURE);
90
91      char* chaine;
92
93      int tailleTab = 0;
94      while(!FT_estEnFinDeFichier(ficDico)){
95          chaine = FT_lireChaineSansLeRetourChariot(ficDico);
96
97          if(strlen(chaine)>0){
98              lesMots[tailleTab] = M_creerUnMot(chaine);
99              free(chaine);
100
101             tailleTab++;
102         }
103     }
104     FT_fermer(&ficDico);
105     *nbMots = tailleTab;
106     return lesMots;
107 }
108
109 Dictionnaire D_genererDicoAvecTableauDeMots(Mot* lesMots, int nbMots){
110     Dictionnaire unDico = ADL_creerADLVide();
111     int i;
112     Mot unMot;
113     for(i =0; i < nbMots; i++){
114         unMot = M_copierMot(lesMots[i]);
115         D_insererMot(&unDico, unMot);
116         M_supprimerMot(&unMot);
117     }
118     return unDico;
119 }
120
121 int charEnInt(char c){
122     return c - '0';
123 }
124
125 void D_chargerDicoR(Dictionnaire* unDico, FichierTexte sauvegardeDico){
126     Dictionnaire temp;
127     char lettre, estFinDeMot, aUnFils, aUnFrere;
128     char* element = FT_lireElement(sauvegardeDico);
129     lettre = element[0];
130     estFinDeMot = element[1];
131     aUnFils = element[2];
132     aUnFrere = element[3];
133     *unDico = ADL_creerADL(NULL, NULL, lettre, charEnInt(estFinDeMot));
134     if(charEnInt(aUnFils) == 1){
135         D_chargerDicoR(&temp, sauvegardeDico);
136         ADL_fixerFils(unDico, temp);
137     }
```

```
138     if(charEnInt(aUnFrere) == 1){
139         D_chargerDicoR(&temp, sauvegardeDico);
140         ADL_fixerFrere(unDico, temp);
141     }
142     /*
143     if(!charEnInt(element[3]) && !charEnInt(element[2])){
144         *unDico = ADL_creerADL(NULL, NULL, element[0], charEnInt(element[1]));
145     }
146     */
147     free(element);
148 
149 }
150 
151 void D_sauvegarderDicoR(Dictionnaire* unDico, FichierTexte fic){
152 
153     Dictionnaire tempFils, tempFrere;
154     if(!ADL_estVide(*unDico)){
155         FT_ecrireCaractere(&fic ,ADL_obtenirLettre(*unDico));
156         if(ADL_obtenirEstFinDeMot(*unDico))
157             FT_ecrireCaractere(&fic , '1');
158         else FT_ecrireCaractere(&fic , '0');
159 
160         tempFils = ADL_obtenirFils(*unDico);
161         tempFrere = ADL_obtenirFrere(*unDico);
162         if(!ADL_estVide(tempFils)){
163             FT_ecrireCaractere(&fic ,'1');
164         }
165         else FT_ecrireCaractere(&fic ,'0');
166         if(!ADL_estVide(tempFrere)){
167             FT_ecrireCaractere(&fic ,'1');
168         }
169         else FT_ecrireCaractere(&fic ,'0');
170 
171         D_sauvegarderDicoR(&tempFils, fic);
172         D_sauvegarderDicoR(&tempFrere, fic);
173     }
174 
175 }
176 
177 /* Partie publique */
178 
179 Dictionnaire D_genererDicoAvecFichierTexte(FichierTexte ficDico){
180     int nbMots;
181     Mot *lesMots = D_genererTableauDeMotAvecFichierTexte(ficDico, &nbMots);
182     Dictionnaire leDico = D_genererDicoAvecTableauDeMots(lesMots, nbMots);
183     supprimerTabMots(&lesMots, nbMots);
184     return leDico;
185 }
186 
187 int D_estUnMotDuDictionnaire(Dictionnaire unDico, Mot unMot){
188     Dictionnaire temp;
189     if(M_longueurMot(unMot) == 1){
190         if(!ADL_estVide(unDico)){
191             if(M_iemeCaractere(unMot, 1) == ADL_obtenirLettre(unDico)){
```

```
192                     M_supprimerMot(&unMot);
193                     return ADL_obtenirEstFinDeMot(unDico);
194                 }
195                 else{
196                     temp = ADL_obtenirFrere(unDico);
197                     return D_estUnMotDuDictionnaire(temp, unMot);
198                 }
199             }
200             else{
201                 M_supprimerMot(&unMot);
202                 return 0;
203             }
204         }
205         else{
206             if(!ADL_estVide(unDico)){
207                 if(M_iemeCaractere(unMot, 1) == ADL_obtenirLettre(unDico)){
208                     M_supprimerIemeLettre(&unMot, 1);
209                     temp = ADL_obtenirFils(unDico);
210                     return D_estUnMotDuDictionnaire(temp, unMot);
211                 }
212                 else{
213                     temp = ADL_obtenirFrere(unDico);
214                     return D_estUnMotDuDictionnaire(temp, unMot);
215                 }
216             }
217             else{
218                 M_supprimerMot(&unMot);
219                 return 0;
220             }
221         }
222     }
223
224     Dictionnaire D_chargerDico(FichierTexte sauvegardeDico){
225         Dictionnaire unDico;
226         FT_ouvrir(&sauvegardeDico, LECTURE);
227         D_chargerDicoR(&unDico, sauvegardeDico);
228         FT_fermer(&sauvegardeDico);
229         return unDico;
230     }
231
232     void D_sauvegarderDico(Dictionnaire* unDico, FichierTexte *sauvegardeDico){
233         FT_ouvrir(sauvegardeDico, ECRITURE);
234         D_sauvegarderDicoR(unDico, *sauvegardeDico);
235         FT_fermer(sauvegardeDico);
236     }
```

../programme/src/Dictionnaire.c

```
1   #include <stdlib.h>
2   #include <string.h>
3   #include <stdio.h>
4   #include <assert.h>
5   #include <errno.h>
6   #include "Mot.h"
```

```c
#include "ListeChaineeDeMot.h"
#include "EnsembleDeMot.h"

/* Partie privee */
void EDM_ajouterElements(EnsembleDeMot motsAAjouter, EnsembleDeMot *edmACompleter){
    errno = 0;
    ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
    l = motsAAjouter.lesMots;
    while (!LCDM_estVide(l)){
        EDM_ajouter(edmACompleter, LCDM_obtenirMot(l));
        l = LCDM_obtenirListeSuivante(l);
    }
}

/* Partie publique */
EnsembleDeMot ensembleDeMot(){
    errno = 0;
    EnsembleDeMot unEDM;
    unEDM.lesMots = LCDM_listeChaineeDeMot();
    unEDM.nbMots = 0;
    return unEDM;
}

void EDM_vider(EnsembleDeMot *unEDM){
    errno = 0;
    unEDM->nbMots = 0;
    LCDM_supprimer(&unEDM->lesMots);
    *unEDM = ensembleDeMot();
}

EnsembleDeMot EDM_copier(EnsembleDeMot unEDM){
    EnsembleDeMot copieEDM = ensembleDeMot();
    copieEDM.nbMots = unEDM.nbMots;
    copieEDM.lesMots = LCDM_copier(unEDM.lesMots);
    return copieEDM;
}

int EDM_egale(EnsembleDeMot edm_1, EnsembleDeMot edm_2){
    errno = 0;
    int sontEgales = 0;
    if (EDM_cardinalite(edm_1) == EDM_cardinalite(edm_2)){
        sontEgales = LCDM_egale(edm_1.lesMots, edm_2.lesMots);
    }
    return sontEgales;
}

void EDM_ajouter(EnsembleDeMot *unEDM, Mot unMot){
    if (!EDM_estPresent(*unEDM, unMot)){
        LCDM_ajouter(&unEDM->lesMots, unMot);
        unEDM->nbMots = unEDM->nbMots + 1;
    }
    else{
        errno = LCDM_ERREUR_MEMOIRE;
    }
```

```
61  }
62
63  void EDM_retirer(EnsembleDeMot *unEDM, Mot unMot){
64      errno = 0;
65      if (EDM_estPresent(*unEDM, unMot)){
66          LCDM_supprimerMot(&unEDM->lesMots, unMot);
67          unEDM->nbMots--;
68      }
69  }
70
71  int EDM_estPresentDansListe(ListeChaineeDeMot l, Mot unMot){
72      errno = 0;
73      if (LCDM_estVide(l)){
74          return 0;
75      }
76      else{
77          if (M_sontIdentiques(LCDM_obtenirMot(l), unMot)){
78              return 1;
79          }
80          else{
81              return EDM_estPresentDansListe(LCDM_obtenirListeSuivante(l), unMot);
82          }
83      }
84  }
85
86  int EDM_estPresent(EnsembleDeMot unEDM, Mot unMot){
87      return EDM_estPresentDansListe(unEDM.lesMots, unMot);
88  }
89
90  long int EDM_cardinalite(EnsembleDeMot unEDM){
91      errno = 0;
92      return unEDM.nbMots;
93  }
94
95  EnsembleDeMot EDM_union(EnsembleDeMot edm_1, EnsembleDeMot edm_2){
96      EnsembleDeMot unionEDM = ensembleDeMot();
97      EDM_ajouterElements(edm_1, &unionEDM);
98      EDM_ajouterElements(edm_2, &unionEDM);
99      return unionEDM;
100 }
101
102 Mot EDM_obtenirMot(EnsembleDeMot unEDM){
103     errno = 0;
104     Mot leMot;
105     ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
106     l = unEDM.lesMots;
107     leMot = LCDM_obtenirMot(l);
108     //free(l);
109     return leMot;
110 }
```

../programme/src/EnsembleDeMot.c

```
1  #include <stdlib.h>
```

```c
#include <assert.h>
#include <string.h>
#include "FichierTexte.h"
#define LONGUEUR_MAX_MOT 27

FichierTexte FT_fichierTexte(char *nomDuFichier)
{
    FichierTexte unFichier;
    unFichier.fichier = NULL;
    unFichier.nom = nomDuFichier;
    return unFichier;
}

void FT_ouvrir(FichierTexte *unFichier, Mode mode)
{
    assert(!FT_estOuvert(*unFichier));

    if (mode == ECRITURE)
    {
        unFichier->fichier = fopen(unFichier->nom, "w+");
        unFichier->mode = mode;
    }
    else if (mode == LECTURE)
    {
        unFichier->fichier = fopen(unFichier->nom, "r");
        unFichier->mode = mode;
    }
    else
        printf("ERREUR : Le mode choisi n'est pas le bon\n");
}

void FT_fermer(FichierTexte *unFichier)
{
    if (FT_estOuvert(*unFichier))
    {
        fclose(unFichier->fichier);
        unFichier->fichier = NULL;
    }
}

unsigned int FT_estOuvert(FichierTexte unFichier)
{
    return unFichier.fichier != NULL;
}

Mode FT_obtenirMode(FichierTexte unFichier)
{
    return unFichier.mode;
}

unsigned int FT_estEnFinDeFichier(FichierTexte unFichier)
{
    assert((unFichier.mode == LECTURE) && FT_estOuvert(unFichier));
    return feof(unFichier.fichier);
```

```
56  }
57  /*
58  void FT_ecrireChaine(FichierTexte *fichier, char *chaine)
59  {
60      assert(FT_estOuvert(*fichier) && (FT_mode(*fichier) == ECRITURE));
61      fputs(chaine, fichier->fichier);
62  }
63  */
64  char *FT_lireChaine(FichierTexte unFichier)
65  {
66      assert(FT_estOuvert(unFichier) && (FT_obtenirMode(unFichier) == LECTURE) && !FT_estEnFinDeFichier(
        unFichier));
67
68      char* buffer = (char*)malloc(sizeof(char)*LONGUEUR_MAX_MOT);
69      if(fgets(buffer, LONGUEUR_MAX_MOT, unFichier.fichier)){
70          return buffer;
71      }
72      else{
73          free(buffer);
74          return NULL;
75      }
76  }
77
78  void supprimerRetourChariot(char *chaine){
79      int i = 0;
80      while(chaine[i] != '\0'){
81          if(chaine[i] == '\n'){
82              chaine[i] = '\0';
83          }
84          i++;
85      }
86  }
87
88  char* FT_lireChaineSansLeRetourChariot(FichierTexte unFichier){
89      assert(FT_estOuvert(unFichier) && (FT_obtenirMode(unFichier) == LECTURE) && !FT_estEnFinDeFichier(
        unFichier));
90      char* ligne;
91      ligne = FT_lireChaine(unFichier);
92      if(ligne != NULL){
93          supprimerRetourChariot(ligne);
94          return ligne;
95      }
96      else{
97          return ""; //A voir
98      }
99  }
100
101 void FT_ecrireCaractere(FichierTexte *unFichier, char lettre)
102 {
103     assert(FT_estOuvert(*unFichier) && (FT_obtenirMode(*unFichier) == ECRITURE));
104     fputc(lettre, unFichier->fichier);
105 }
106
107
```

```
108 char FT_lireCaractere(FichierTexte unFichier)
109 {
110     assert(FT_estOuvert(unFichier) && (FT_obtenirMode(unFichier) == LECTURE) && !FT_estEnFinDeFichier(
        unFichier));
111     return fgetc(unFichier.fichier);
112 }
```

../programme/src/FichierTexte.c

```
1  #include <stdlib.h>
2  #include <assert.h>
3  #include <string.h>
4  #include "ListeChaineeDeMot.h"
5  #include "Mot.h"
6
7  ListeChaineeDeMot LCDM_listeChaineeDeMot(){
8      errno = 0;
9      return NULL;
10 }
11
12 int LCDM_estVide(ListeChaineeDeMot l){
13     errno = 0;
14     return (l == NULL);
15 }
16
17 void LCDM_ajouter(ListeChaineeDeMot *l, Mot unMot){
18    ListeChaineeDeMot pNoeud = malloc(sizeof(Noeud));
19     if (pNoeud != NULL){
20         errno = 0;
21         pNoeud->mot = unMot;
22         pNoeud->listeSuivante = *l;
23         *l = pNoeud;
24     }
25     else{
26         errno = LCDM_ERREUR_MEMOIRE;
27     }
28 }
29
30 Mot LCDM_obtenirMot(ListeChaineeDeMot l){
31     assert(!LCDM_estVide(l));
32     errno = 0;
33     return l->mot;
34 }
35
36 void LCDM_supprimerMot(ListeChaineeDeMot *l, Mot unMot){
37     assert(!LCDM_estVide(*l));
38     ListeChaineeDeMot temp = LCDM_listeChaineeDeMot();
39     if (!LCDM_estVide(*l)){
40         if (M_sontIdentiques(LCDM_obtenirMot(*l), unMot)){
41             LCDM_supprimerTete(l);
42         }else{
43             temp = LCDM_obtenirListeSuivante(*l);
44             LCDM_supprimerMot(&temp, unMot);
45             LCDM_fixerListeSuivante(l, temp);
```

```
46
47              }
48          }
49      }
50
51      ListeChaineeDeMot LCDM_obtenirListeSuivante(ListeChaineeDeMot l){
52          assert(!LCDM_estVide(l));
53          errno = 0;
54          return l->listeSuivante;
55      }
56
57      void LCDM_fixerListeSuivante(ListeChaineeDeMot *l, ListeChaineeDeMot suivant){
58          assert(!LCDM_estVide(*l));
59          errno = 0;
60          (*l)->listeSuivante = suivant;
61      }
62
63      void LCDM_fixerMot(ListeChaineeDeMot *l, Mot unMot){
64          assert(!LCDM_estVide(*l));
65          errno = 0;
66          (*l)->mot = unMot;
67      }
68
69      void LCDM_supprimerTete(ListeChaineeDeMot *l){
70          ListeChaineeDeMot temp;
71          assert(!LCDM_estVide(*l));
72          errno = 0;
73          temp = *l;
74          *l = LCDM_obtenirListeSuivante(*l);
75          free(temp);
76      }
77
78      void LCDM_supprimer(ListeChaineeDeMot *l){
79          errno = 0;
80          if (!LCDM_estVide(*l)){
81              LCDM_supprimerTete(l);
82              LCDM_supprimer(l);
83          }
84      }
85
86      ListeChaineeDeMot LCDM_copier(ListeChaineeDeMot l){
87          ListeChaineeDeMot temp;
88          errno = 0;
89          if (LCDM_estVide(l)){
90              return LCDM_listeChaineeDeMot();
91              free(temp);
92          }else{
93              temp = LCDM_copier(LCDM_obtenirListeSuivante(l));
94              LCDM_ajouter(&temp, LCDM_obtenirMot(l));
95              return temp;
96          }
97      }
98
99      int LCDM_egale(ListeChaineeDeMot l1, ListeChaineeDeMot l2){
```

```
100     errno = 0;
101     if (LCDM_estVide(l1) && LCDM_estVide(l2)){
102         return 1;
103     }else{
104         if (LCDM_estVide(l1) || LCDM_estVide(l2)){
105             return 0;
106         }else{
107             if (M_sontIdentiques(LCDM_obtenirMot(l1),LCDM_obtenirMot(l2))){
108                 return LCDM_egale(LCDM_obtenirListeSuivante(l1), LCDM_obtenirListeSuivante(l2));
109             }
110             else{
111                 return 0;
112             }
113         }
114     }
115 }
```

../programme/src/ListeChaineeDeMot.c

```
1  #include <stdlib.h>
2  #include <assert.h>
3  #include <string.h>
4
5  int main(){
6      return 0;
7  }
```

../programme/src/main.c

```
1  #include <stdlib.h>
2  #include <assert.h>
3  #include <stdio.h>
4  #include "Mot.h"
5  #include <string.h>
6  #include <ctype.h>
7
8
9  int M_estUnCaractereAlphabetique(char c){
10     char tmp = c;
11     return (isalpha(tmp) || M_estUnCaractereAAccent(c));
12 }
13
14 int M_estUnCaractereAAccent(char c){
15     return (c == '-'
16         || c == 'ù'
17         || c == 'à'
18         || c == 'é'
19         || c == 'è'
20         || c == 'ç'
21         || c == 'ï'
22         || c == 'É'
23         || c == 'É'
24         || c == 'Ç'
```

```
25            || c == 'À'
26            || c == 'â'
27            || c == 'Â'
28            || c == 'ë'
29            || c == 'Ë'
30            || c == 'ô'
31            || c == 'Ô'
32            || c == 'î'
33            || c == 'ê'
34            || c == 'û'
35            || c == 'ü'
36            || c == 'ä'
37            || c == 'ö');
38  }
39
40
41
42
43  int M_estUnMotValide(char* c){
44      assert(strlen(c)>0);
45      int longueurChaine = strlen(c);
46      int estValide = 1;
47      int i = 0;
48      while(estValide && i<longueurChaine-1){
49          if(!M_estUnCaractereAlphabetique(c[i])){
50              estValide=0;
51          }
52          i=i+1;
53      }
54      return estValide;
55  }
56
57  Mot M_copierMot(Mot unMot){
58      Mot copie;
59      copie = M_creerUnMot(unMot.chaine);
60      return copie;
61  }
62
63  Mot M_creerUnMot(char *c){
64      assert(M_estUnMotValide(c));
65      Mot unMot;
66      unMot.chaine = M_reduireLaCasseDUneChaine(c);
67      unMot.longueur = strlen(unMot.chaine);
68      return unMot;
69  }
70
71  unsigned int M_longueurMot(Mot unMot){
72      return unMot.longueur;
73  }
74
75  char* M_obtenirChaine(Mot unMot){
76      return unMot.chaine;
77  }
78
```

```c
char M_iemeCaractere(Mot unMot, unsigned int i){
    assert(i <= M_longueurMot(unMot) && i > 0);
    return unMot.chaine[i-1];
}

int M_sontIdentiques(Mot mot1, Mot mot2){
    return mot1.longueur==mot2.longueur && strcmp(mot1.chaine,mot2.chaine)==0;
}

void M_fixerIemeCaractere(Mot* unMot, unsigned int i, char c){
    assert(M_estUnCaractereAlphabetique(c)&&i<=M_longueurMot(*unMot)+1);
    unMot->chaine[i-1]=c;
}

void M_fixerLongueur(Mot* unMot, unsigned int i){
    unMot->longueur=i;
}


void M_supprimerIemeLettre(Mot* unMot, unsigned int indiceLettreASupprimer){
    assert(indiceLettreASupprimer<=M_longueurMot(*unMot));

    int indiceLettreCourante = indiceLettreASupprimer;
    while(indiceLettreCourante < M_longueurMot(*unMot)){
        M_fixerIemeCaractere(unMot,indiceLettreCourante, M_iemeCaractere(*unMot, indiceLettreCourante+1));
        indiceLettreCourante++;
    }
    unMot->chaine[indiceLettreCourante-1]='\0';
    M_fixerLongueur(unMot,M_longueurMot(*unMot)-1);
}

void M_inverserDeuxLettresConsecutives(Mot* unMot, unsigned int i){
    assert(i < M_longueurMot(*unMot) && i > 0);
    char temp;
    temp = M_iemeCaractere(*unMot,i);
    M_fixerIemeCaractere(unMot,i,M_iemeCaractere(*unMot,i+1));
    M_fixerIemeCaractere(unMot,i+1,temp);
}

void M_insererLettre(Mot* unMot, unsigned int i, char c){
    assert(i<=M_longueurMot(*unMot)+1);
    unMot->chaine = realloc(unMot->chaine,sizeof(char)*M_longueurMot(*unMot)+2);
    for (int j=strlen(unMot->chaine);j>i-1;j--){
        M_fixerIemeCaractere(unMot,j+1,M_iemeCaractere(*unMot,j));
    }
    M_fixerLongueur(unMot,M_longueurMot(*unMot)+1);
    unMot->chaine[M_longueurMot(*unMot)]='\0';
    unMot->chaine[i-1] = c;
}

Mot M_decomposerMot(Mot* unMot, unsigned int i){ //le ieme caractere est dans la deuxième partie du mot
    assert(i <= M_longueurMot(*unMot) && i > 1);
    char* chaineGauche = (char*)malloc(i+1);
```

```
132     int j;
133     for(j = 0; j<i-1;j++){
134         chaineGauche[j] =  M_iemeCaractere(*unMot, 1);
135         M_supprimerIemeLettre(unMot, 1);
136     }
137     chaineGauche[j] = '\0';
138     Mot motGauche = M_creerUnMot(chaineGauche);
139     free(chaineGauche);
140     return motGauche;
141 }
142
143 char M_reduireLaCasseDUnCaractere(char car){
144     char c = car;
145     switch (c){
146
147         case 'À':
148             c = 'à';
149             break;
150
151         case 'Â':
152             c = 'â';
153             break;
154
155         case 'É':
156             c = 'é';
157             break;
158
159         case 'È':
160             c = 'è';
161             break;
162
163         case 'Ç':
164             c = 'ç';
165             break;
166
167         case 'Ê':
168             c = 'ê';
169             break;
170
171         case 'Ô':
172             c = 'ô';
173             break;
174
175         case 'Ï':
176             c = 'à';
177             break;
178
179         case 'Î':
180             c = 'à';
181             break;
182
183         case 'Û':
184             c = 'û';
185             break;
```

```
186
187             case 'Ù':
188                 c = 'ù';
189                 break;
190
191             default:
192                 c = (tolower((unsigned char)c));
193         }
194         return c;
195     }
196
197     char* M_reduireLaCasseDUneChaine(char* chaine){
198         assert(strlen(chaine)>0);
199         char *minuscule = malloc(strlen(chaine)+2);
200         strcpy(minuscule, chaine);
201         for (int i=0;i<strlen(chaine);i++){
202             minuscule[i] = M_reduireLaCasseDUnCaractere((char)chaine[i]);
203         }
204         return minuscule;
205     }
206
207
208     void M_supprimerMot(Mot *unMot){
209         if(M_longueurMot(*unMot) > 0)
210             free(unMot->chaine);
211         unMot->longueur=0;
212     }
```

../programme/src/Mot.c

6 Tests unitaires

```c
#include <stdio.h>
#include <CUnit/Basic.h>
#include <string.h>
#include <stdbool.h>
#include "Mot.h"
#include "ArbreDeLettres.h"

int init_suite_success(void){
    return 0;
}

int clean_suite_success(void){
    return 0;
}




void test_arbre_vide(void){
    ArbreDeLettres a = ADL_creerADLVide();
    CU_ASSERT_TRUE(ADL_estVide(a));
    ADL_supprimer(&a);
}

void test_arbre_non_vide(void){
    ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
    CU_ASSERT_TRUE(!ADL_estVide(a));
    ADL_supprimer(&a);
}

void test_presence_fils_et_frere(void){
    ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
    ArbreDeLettres b = ADL_creerADL(NULL,NULL,'b',0);
    ArbreDeLettres c = ADL_creerADL(NULL,NULL,'c',0);
    ADL_fixerElement(&a,'a',0);
    ADL_fixerFrere(&a,b);
    ADL_fixerFils(&a,c);
    CU_ASSERT_TRUE(!ADL_estVide(ADL_obtenirFils(a)) && !ADL_estVide(ADL_obtenirFrere(a)));
    ADL_supprimer(&a);
}

void test_fixerElement(void){
    ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
    ADL_fixerElement(&a,'a',0);
    CU_ASSERT_TRUE('a'==ADL_obtenirLettre(a) && ADL_obtenirEstFinDeMot(a)==0);
    ADL_supprimer(&a);
}


void test_fixerEstFinDeMot(void){
    ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
```

```
52      ADL_fixerElement(&a,'a',0);
53      ADL_fixerEstFinDeMot(&a,1);
54      CU_ASSERT_TRUE(a->estFinDeMot==1);
55      ADL_supprimer(&a);
56  }
57
58  void test_fixerLettre(void){
59      ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
60      ADL_fixerLettre(&a,'a');
61      CU_ASSERT_TRUE(a->lettre=='a');
62      ADL_supprimer(&a);
63  }
64
65  void test_fixerFrere(void){
66      ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
67      ArbreDeLettres b = ADL_creerADL(NULL,NULL,'b',0);
68      ADL_fixerElement(&a,'a',0);
69      ADL_fixerElement(&b,'b',1);
70      ADL_fixerFrere(&a,b);
71      CU_ASSERT_TRUE(ADL_obtenirLettre(a->frere)=='b');
72      ADL_supprimer(&a);
73  }
74
75
76  void test_fixerFils(void){
77      ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
78      ArbreDeLettres b = ADL_creerADL(NULL,NULL,'b',0);
79      ADL_fixerElement(&a,'a',0);
80      ADL_fixerElement(&b,'b',1);
81      ADL_fixerFils(&a,b);
82      CU_ASSERT_TRUE(ADL_obtenirLettre(a->fils)=='b');
83      ADL_supprimer(&a);
84  }
85
86  void test_obtenirLettre(void){
87      ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
88      ADL_fixerLettre(&a,'a');
89      CU_ASSERT_TRUE(ADL_obtenirLettre(a)=='a');
90      ADL_supprimer(&a);
91  }
92
93  void test_obtenirFrere(void){
94      ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
95      ArbreDeLettres b = ADL_creerADL(NULL,NULL,'b',0);
96      ADL_fixerLettre(&b,'a');
97      ADL_fixerFrere(&a,b);
98      CU_ASSERT_TRUE(ADL_obtenirFrere(a)==b);
99      ADL_supprimer(&a);
100 }
101
102 void test_obtenirFils(void){
103     ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
104     ArbreDeLettres b = ADL_creerADL(NULL,NULL,'b',0);
105     ADL_fixerElement(&a,'a',0);
```

```
106     ADL_fixerElement(&b,'b',1);
107     ADL_fixerFils(&a,b);
108     CU_ASSERT_TRUE(ADL_obtenirLettre(ADL_obtenirFils(a))=='b');
109     ADL_supprimer(&a);
110 }
111
112 void test_estFinDeMot(void){
113     ArbreDeLettres a = ADL_creerADL(NULL,NULL,'a',0);
114     ADL_fixerElement(&a,'a',0);
115     ADL_fixerEstFinDeMot(&a,1);
116     CU_ASSERT_TRUE(ADL_obtenirEstFinDeMot(a)==1);
117     ADL_supprimer(&a);
118 }
119
120
121 int main(int argc, char **argv){
122     CU_pSuite pSuite = NULL;
123
124     /* initialisation du registre de tests */
125     if (CUE_SUCCESS != CU_initialize_registry()){
126         return CU_get_error();
127     }
128     /* ajout d'une suite de test */
129     pSuite = CU_add_suite("Tests boite noire", init_suite_success, clean_suite_success);
130     if (NULL == pSuite){
131         CU_cleanup_registry();
132         return CU_get_error();
133     }
134
135     /* Ajout des tests à la suite de tests boite noire */
136     if ((NULL == CU_add_test(pSuite, "1 - la creation d'une liste qui doit etre vide", test_arbre_vide))
137     || (NULL == CU_add_test(pSuite, "2 - une liste contenant un element n'est pas vide",
        test_arbre_non_vide))
138     || (NULL == CU_add_test(pSuite, "3 - creation d'un arbre avec fils et frere fonctionne",
        test_presence_fils_et_frere))
139     || (NULL == CU_add_test(pSuite, "4 - fixation d'un élément d'arbre de lettres", test_fixerElement))
140     || (NULL == CU_add_test(pSuite, "5 - fixer le parametre fin de mot d'un arbre ",
        test_fixerEstFinDeMot))
141     || (NULL == CU_add_test(pSuite, "6 - fixation de la lettre d'un élément d'arbre de lettres ",
        test_fixerLettre))
142     || (NULL == CU_add_test(pSuite, "7 - fixer le fere d'un arbre ", test_fixerFrere))
143     || (NULL == CU_add_test(pSuite, "8 - fixer le fils d'un arbre ", test_fixerFils))
144     || (NULL == CU_add_test(pSuite, "9 - test pour obtenir la lettre de l'élément dans l'arbre ",
        test_obtenirLettre))
145     || (NULL == CU_add_test(pSuite, "10 - test pour obtenir le frere de l'élément dans l'arbre ",
        test_obtenirFrere))
146     || (NULL == CU_add_test(pSuite, "11 - test pour obtenir le fils de l'élément dans l'arbre ",
        test_obtenirFils))
147     || (NULL == CU_add_test(pSuite, "12 - test pour obtenir l'etat de fin de mot ou non d'un element d'
        arbre'", test_estFinDeMot))
148
149     )
150     {
151         CU_cleanup_registry();
```

```
152             return CU_get_error();
153         }
154 
155         /* Lancement des tests */
156         CU_basic_set_mode(CU_BRM_VERBOSE);
157         CU_basic_run_tests();
158         printf("\n");
159         CU_basic_show_failures(CU_get_failure_list());
160         printf("\n\n");
161 
162         /* Nettoyage du registre */
163         CU_cleanup_registry();
164         return CU_get_error();
165     }
```

../programme/src/testArbreDeLettres.c

```
1   #include <stdio.h>
2   #include <CUnit/Basic.h>
3   #include <string.h>
4   #include <stdbool.h>
5   #include "Mot.h"
6   #include "CorrecteurOrthographique.h"
7   #include "EnsembleDeMot.h"
8   #include "Dictionnaire.h"
9   
10  
11  int init_suite_success(void){
12      return 0;
13  }
14  
15  int clean_suite_success(void){
16      return 0;
17  }
18  
19  char** creer_tableau_mot(){
20      char** lesMots=(char**)malloc((sizeof(char)*5)*50);
21      lesMots[0] = "bvec";
22      lesMots[1] = "avec";
23      lesMots[2] = "avac";
24      lesMots[3] = "avem";
25      lesMots[4] = "arride";
26      lesMots[5] = "attaque";
27      return lesMots;
28  }
29  
30  
31  
32  Dictionnaire creer_dictionnaire(){
33      char** lesChaines = creer_tableau_mot();
34      Mot* lesMots = (Mot*)malloc(((sizeof(char)*7)+sizeof(int))*54);
35      for(int i=0;i<6;i++){
36          lesMots[i]=M_creerUnMot(lesChaines[i]);
37      }
```

```
38      Dictionnaire dico = D_genererDicoAvecTableauDeMots(lesMots,6);
39      free(lesChaines);
40      supprimerTabMots(&lesMots, 6);
41      return dico;
42  }
43
44
45  char** creer_ensemble_solution(){
46      char** lesMots=(char**)malloc((sizeof(char)*5)*61);
47      lesMots[0] = "aveö";
48      lesMots[1] = "avea";
49      lesMots[2] = "aveb";
50      lesMots[3] = "avec";
51      lesMots[4] = "aved";
52      lesMots[5] = "avee";
53      lesMots[6] = "avef";
54      lesMots[7] = "aveg";
55      lesMots[8] = "aveh";
56      lesMots[9] = "avei";
57      lesMots[10] = "avej";
58      lesMots[11] = "avek";
59      lesMots[12] = "avel";
60      lesMots[13] = "avem";
61      lesMots[14] = "aven";
62      lesMots[15] = "aveo";
63      lesMots[16] = "avep";
64      lesMots[17] = "aveq";
65      lesMots[18] = "aver";
66      lesMots[19] = "aves";
67      lesMots[20] = "avet";
68      lesMots[21] = "aveu";
69      lesMots[22] = "avev";
70      lesMots[23] = "avew";
71      lesMots[24] = "avex";
72      lesMots[25] = "avey";
73      lesMots[26] = "avez";
74      lesMots[27] = "aveà";
75      lesMots[28] = "aveé";
76      lesMots[29] = "aveè";
77      lesMots[30] = "aveë";
78      lesMots[31] = "aveù";
79      lesMots[32] = "aveû";
80      lesMots[33] = "aveê";
81      lesMots[34] = "aveî";
82      lesMots[35] = "aveï";
83      lesMots[36] = "aveç";
84      lesMots[37] = "aveô";
85      return lesMots;
86  }
87
88  char** creer_ensemble_solution_inserer(){
89      char** lesMots=(char**)malloc((sizeof(char)*6)*100);
90      lesMots[0] = "aveöc";
91      lesMots[1] = "aveac";
```

```
92      lesMots[2] = "avebc";
93      lesMots[3] = "avecc";
94      lesMots[4] = "avedc";
95      lesMots[5] = "aveec";
96      lesMots[6] = "avefc";
97      lesMots[7] = "avegc";
98      lesMots[8] = "avehc";
99      lesMots[9] = "aveic";
100     lesMots[10] = "avejc";
101     lesMots[11] = "avekc";
102     lesMots[12] = "avelc";
103     lesMots[13] = "avemc";
104     lesMots[14] = "avenc";
105     lesMots[15] = "aveoc";
106     lesMots[16] = "avepc";
107     lesMots[17] = "aveqc";
108     lesMots[18] = "averc";
109     lesMots[19] = "avesc";
110     lesMots[20] = "avetc";
111     lesMots[21] = "aveuc";
112     lesMots[22] = "avevc";
113     lesMots[23] = "avewc";
114     lesMots[24] = "avexc";
115     lesMots[25] = "aveyc";
116     lesMots[26] = "avezc";
117     lesMots[27] = "aveàc";
118     lesMots[28] = "aveéc";
119     lesMots[29] = "aveèc";
120     lesMots[30] = "aveëc";
121     lesMots[31] = "aveùc";
122     lesMots[32] = "aveûc";
123     lesMots[33] = "aveêc";
124     lesMots[34] = "aveîc";
125     lesMots[35] = "aveïc";
126     lesMots[36] = "aveçc";
127     lesMots[37] = "aveôc";
128
129     return lesMots;
130
131 }
132
133 void test_remplacer_ieme_lettre(){
134     char* chaine = "avek";
135     Mot motACorriger = M_creerUnMot(chaine);
136     EnsembleDeMot ensemble = CO_remplacerIemeLettreEnBoucle(motACorriger,4);
137     char** solution = creer_ensemble_solution();
138     for(int i=0;i<38;i++){
139         Mot mot = M_creerUnMot(solution[i]);
140         CU_ASSERT_TRUE(EDM_estPresent(ensemble,mot));
141         M_supprimerMot(&mot);
142     }
143     M_supprimerMot(&motACorriger);
144     while(!EDM_cardinalite(ensemble)==0){
145         Mot tmp = EDM_obtenirMot(ensemble);
```

```
146             EDM_retirer(&ensemble,tmp);
147             M_supprimerMot(&tmp);
148         }
149         EDM_vider(&ensemble);
150         free(solution);
151     }
152
153     void test_inserer_lettre(){
154         char* chaine = "avec";
155         Mot motACorriger = M_creerUnMot(chaine);
156         EnsembleDeMot ensemble = CO_insererIemeLettreEnBoucle(motACorriger,4);
157         char **solution = creer_ensemble_solution_inserer();
158         for(int i=0;i<38;i++){
159             Mot mot = M_creerUnMot(solution[i]);
160             CU_ASSERT_TRUE(EDM_estPresent(ensemble,mot));
161             M_supprimerMot(&mot);
162         }
163         M_supprimerMot(&motACorriger);
164         while(!EDM_cardinalite(ensemble)==0){
165             Mot tmp = EDM_obtenirMot(ensemble);
166             EDM_retirer(&ensemble,tmp);
167             M_supprimerMot(&tmp);
168         }
169         EDM_vider(&ensemble);
170         free(solution);
171     }
172
173     void test_strategie_remplacer_lettre(){
174         Dictionnaire dico = creer_dictionnaire();
175         char* chaine = "abec";
176         char* chaine1 = "avec";
177         Mot motACorriger = M_creerUnMot(chaine);
178         Mot solution = M_creerUnMot(chaine1);
179         CorrecteurOrthographique correcteur = CO_correcteur(dico, motACorriger);
180         CO_strategieRemplacerLettres(&correcteur);
181         CU_ASSERT_TRUE(EDM_estPresent(correcteur.lesCorrections, solution));
182
183         M_supprimerMot(&motACorriger);
184         M_supprimerMot(&solution);
185         CO_supprimerCorrecteur(&correcteur);
186     }
187
188     void test_strategie_supprimer_lettre(){
189         Dictionnaire dico = creer_dictionnaire();
190         char* chaine = "avvec";
191         char* chaine1 = "avec";
192         Mot motACorriger = M_creerUnMot(chaine);
193         Mot solution = M_creerUnMot(chaine1);
194         CorrecteurOrthographique correcteur = CO_correcteur(dico,motACorriger);
195         CO_strategieSupprimerLettres(&correcteur);
196         CU_ASSERT_TRUE(EDM_estPresent(correcteur.lesCorrections,solution));
197         M_supprimerMot(&solution);
198         M_supprimerMot(&motACorriger);
199         CO_supprimerCO(&correcteur);
```

```
200 }
201
202 void test_strategie_inverser_lettre(){
203     Dictionnaire dico = creer_dictionnaire();
204     char* chaine = "aevc";
205     char* chaine1 = "avec";
206     Mot motACorriger = M_creerUnMot(chaine);
207     Mot solution = M_creerUnMot(chaine1);
208     CorrecteurOrthographique correcteur = CO_correcteur(dico,motACorriger);
209     CO_strategieInverserDeuxLettresConsecutives(&correcteur);
210     CU_ASSERT_TRUE(EDM_estPresent(correcteur.lesCorrections,solution));
211     M_supprimerMot(&motACorriger);
212     M_supprimerMot(&solution);
213     CO_supprimerCO(&correcteur);
214 }
215
216 void test_strategie_inserer_lettre(){
217     Dictionnaire dico = creer_dictionnaire();
218     char* chaine = "avc";
219     char* chaine1 = "avec";
220     Mot motACorriger = M_creerUnMot(chaine);
221     Mot solution = M_creerUnMot(chaine1);
222     CorrecteurOrthographique correcteur = CO_correcteur(dico,motACorriger);
223     CO_strategieInsererLettres(&correcteur);
224     CU_ASSERT_TRUE(EDM_estPresent(correcteur.lesCorrections,solution));
225     M_supprimerMot(&motACorriger);
226     M_supprimerMot(&solution);
227     CO_supprimerCO(&correcteur);
228 }
229
230 void test_strategie_decomposer_mot(){
231     Dictionnaire dico = creer_dictionnaire();
232     char* chaine = "avecarride";
233     char* chaine1 = "avec";
234     char* chaine2 = "arride";
235     Mot motACorriger = M_creerUnMot(chaine);
236     Mot solution1 = M_creerUnMot(chaine1);
237     Mot solution2 = M_creerUnMot(chaine2);
238     CorrecteurOrthographique correcteur = CO_correcteur(dico,motACorriger);
239     CO_strategieDecomposerMot(&correcteur);
240     CU_ASSERT_TRUE(EDM_estPresent(correcteur.lesCorrections, solution1) && EDM_estPresent(correcteur.
        lesCorrections, solution2));
241
242     M_supprimerMot(&motACorriger);
243     M_supprimerMot(&solution1);
244     M_supprimerMot(&solution2);
245     CO_supprimerCorrecteur(&correcteur);
246 }
247
248
249
250
251
252
```

```
253
254
255
256 int main(int argc, char **argv){
257     CU_pSuite pSuite = NULL;
258
259     /* initialisation du registre de tests */
260     if (CUE_SUCCESS != CU_initialize_registry()){
261         return CU_get_error();
262     }
263     /* ajout d'une suite de test */
264     pSuite = CU_add_suite("Tests boite noire", init_suite_success, clean_suite_success);
265     if (NULL == pSuite){
266         CU_cleanup_registry();
267         return CU_get_error();
268     }
269
270     /* Ajout des tests     la suite de tests boite noire */
271     if ((NULL == CU_add_test(pSuite, "remplacer la bonne lettre", test_remplacer_ieme_lettre))
272     ||  (NULL == CU_add_test(pSuite, "inserer a la bonne place", test_inserer_lettre))
273     //||  (NULL == CU_add_test(pSuite, "test strategie remplacer", test_strategie_remplacer_lettre))
274     //||  (NULL == CU_add_test(pSuite, "test strategie supprimer", test_strategie_supprimer_lettre))
275     //||  (NULL == CU_add_test(pSuite, "test strategie inverser", test_strategie_inverser_lettre))
276     //||  (NULL == CU_add_test(pSuite, "test strategie inserer", test_strategie_inserer_lettre))
277     //||  (NULL == CU_add_test(pSuite, "test strategie decomposer", test_strategie_decomposer_mot))
278
279     ){
280         CU_cleanup_registry();
281         return CU_get_error();
282     }
283
284     /* Lancement des tests */
285     CU_basic_set_mode(CU_BRM_VERBOSE);
286     CU_basic_run_tests();
287     printf("\n");
288     CU_basic_show_failures(CU_get_failure_list());
289     printf("\n\n");
290
291     /* Nettoyage du registre */
292     CU_cleanup_registry();
293     return CU_get_error();
294 }
```

../programme/src/testCO.c

```
1 #include <stdio.h>
2 #include <CUnit/Basic.h>
3 #include <string.h>
4 #include "EnsembleDeMot.h"
5 #include "Mot.h"
6
7 #define TRUE 1
8 #define FALSE 0
9
```

```
10  int init_suite_success(void){
11      return 0;
12  }
13
14  int clean_suite_success(void){
15      return 0;
16  }
17
18  void creer_mots_A(Mot *mot1, Mot *mot2, Mot *mot3){
19      *mot1 = M_creerUnMot("test");
20      *mot2 = M_creerUnMot("unitaires");
21      *mot3 = M_creerUnMot("ensembleDeMot");
22  }
23
24  void creer_mots_B(Mot *mot1, Mot *mot2, Mot *mot3){
25      *mot1 = M_creerUnMot("test");
26      *mot2 = M_creerUnMot("sans");
27      *mot3 = M_creerUnMot("problèmes");
28  }
29
30  void test_ensemble_vide(void){
31      EnsembleDeMot e = ensembleDeMot();
32
33      CU_ASSERT_TRUE(EDM_cardinalite(e) == 0);
34
35      EDM_vider(&e);
36  }
37
38  void test_ajouter_non_present(void){
39      int c1, c2;
40      Mot mot4 = M_creerUnMot("sans");
41      EnsembleDeMot e = ensembleDeMot();
42      Mot mot1, mot2, mot3;
43      creer_mots_A(&mot1, &mot2, &mot3);
44      EDM_ajouter(&e, mot1);
45      EDM_ajouter(&e, mot2);
46      EDM_ajouter(&e, mot3);
47
48      c1 = EDM_cardinalite(e);
49      EDM_ajouter(&e, mot4);
50      c2 = EDM_cardinalite(e);
51
52      CU_ASSERT_TRUE(c1 + 1 == c2);
53      EDM_vider(&e);
54      M_supprimerMot(&mot1);
55      M_supprimerMot(&mot2);
56      M_supprimerMot(&mot3);
57      M_supprimerMot(&mot4);
58  }
59
60  void test_ajouter_present(void){
61      int c1, c2;
62      EnsembleDeMot e = ensembleDeMot();
63      Mot mot1, mot2, mot3;
```

```
64      creer_mots_A(&mot1, &mot2, &mot3);
65      EDM_ajouter(&e, mot1);
66      EDM_ajouter(&e, mot2);
67      EDM_ajouter(&e, mot3);
68
69      c1 = EDM_cardinalite(e);
70      EDM_ajouter(&e, mot1);
71      c2 = EDM_cardinalite(e);
72
73      CU_ASSERT_TRUE(c1 == c2);
74
75      EDM_vider(&e);
76      M_supprimerMot(&mot1);
77      M_supprimerMot(&mot2);
78      M_supprimerMot(&mot3);
79  }
80
81  void test_present_apres_ajout(void){
82      Mot mot5 = M_creerUnMot("problèmes");
83      EnsembleDeMot e = ensembleDeMot();
84      Mot mot1, mot2, mot3;
85      creer_mots_A(&mot1, &mot2, &mot3);
86      EDM_ajouter(&e, mot1);
87      EDM_ajouter(&e, mot2);
88      EDM_ajouter(&e, mot3);
89
90      EDM_ajouter(&e, mot5);
91
92      CU_ASSERT_TRUE(EDM_estPresent(e, mot5));
93
94      EDM_vider(&e);
95      M_supprimerMot(&mot5);
96      M_supprimerMot(&mot1);
97      M_supprimerMot(&mot2);
98      M_supprimerMot(&mot3);
99  }
100
101 void test_retirer_present(void){
102     int c1, c2;
103     EnsembleDeMot e = ensembleDeMot();
104     Mot mot1, mot2, mot3;
105     creer_mots_A(&mot1, &mot2, &mot3);
106     EDM_ajouter(&e, mot1);
107     EDM_ajouter(&e, mot2);
108     EDM_ajouter(&e, mot3);
109
110     c1 = EDM_cardinalite(e);
111     EDM_retirer(&e, mot2);
112     c2 = EDM_cardinalite(e);
113
114     CU_ASSERT_TRUE(c1 - 1 == c2);
115
116     EDM_vider(&e);
117     M_supprimerMot(&mot2);
```

```
118     M_supprimerMot(&mot1);
119     M_supprimerMot(&mot3);
120 }
121
122 void test_retirer_non_present(void){
123     int c1, c2;
124     Mot mot4 = M_creerUnMot("sans");
125     EnsembleDeMot e = ensembleDeMot();
126     Mot mot1, mot2, mot3;
127     creer_mots_A(&mot1, &mot2, &mot3);
128     EDM_ajouter(&e, mot1);
129     EDM_ajouter(&e, mot2);
130     EDM_ajouter(&e, mot3);
131
132     c1 = EDM_cardinalite(e);
133     EDM_retirer(&e, mot4);
134     c2 = EDM_cardinalite(e);
135
136     CU_ASSERT_TRUE(c1 == c2);
137
138     EDM_vider(&e);
139     M_supprimerMot(&mot1);
140     M_supprimerMot(&mot2);
141     M_supprimerMot(&mot3);
142     M_supprimerMot(&mot4);
143 }
144
145 void test_absent_apres_retrait(void){
146     EnsembleDeMot e = ensembleDeMot();
147     Mot mot1, mot2, mot3;
148     creer_mots_A(&mot1, &mot2, &mot3);
149     EDM_ajouter(&e, mot1);
150     EDM_ajouter(&e, mot2);
151     EDM_ajouter(&e, mot3);
152
153     EDM_retirer(&e, mot2);
154     CU_ASSERT_FALSE(EDM_estPresent(e, mot2));
155
156     EDM_vider(&e);
157     M_supprimerMot(&mot1);
158     M_supprimerMot(&mot2);
159     M_supprimerMot(&mot3);
160 }
161
162 void test_union(void){
163     EnsembleDeMot e1 = ensembleDeMot();
164     Mot mot1, mot1bis, mot2, mot3, mot4, mot5;
165     creer_mots_A(&mot1, &mot2, &mot3);
166     EDM_ajouter(&e1, mot1);
167     EDM_ajouter(&e1, mot2);
168     EDM_ajouter(&e1, mot3);
169
170     EnsembleDeMot e2 = ensembleDeMot();
171     creer_mots_B(&mot1bis, &mot4, &mot5);
```

```
172     EDM_ajouter(&e2, mot1);
173     EDM_ajouter(&e2, mot4);
174     EDM_ajouter(&e2, mot5);
175
176     EnsembleDeMot e3 = EDM_union(e1, e2);
177
178     CU_ASSERT_TRUE(EDM_estPresent(e3, mot1)
179                 && EDM_estPresent(e3, mot2)
180                 && EDM_estPresent(e3, mot3)
181                 && EDM_estPresent(e3, mot4)
182                 && EDM_estPresent(e3, mot5));
183
184     EDM_vider(&e1);
185     EDM_vider(&e2);
186     EDM_vider(&e3);
187     M_supprimerMot(&mot1);
188     M_supprimerMot(&mot1bis);
189     M_supprimerMot(&mot2);
190     M_supprimerMot(&mot3);
191     M_supprimerMot(&mot4);
192     M_supprimerMot(&mot5);
193 }
194
195 void test_egalite_meme_ensemble(void){
196     EnsembleDeMot e1 = ensembleDeMot();
197     Mot mot1, mot2, mot3;
198     creer_mots_A(&mot1, &mot2, &mot3);
199     EDM_ajouter(&e1, mot1);
200     EDM_ajouter(&e1, mot2);
201     EDM_ajouter(&e1, mot3);
202
203     CU_ASSERT_TRUE(EDM_egale(e1, e1));
204
205     EDM_vider(&e1);
206     M_supprimerMot(&mot1);
207     M_supprimerMot(&mot2);
208     M_supprimerMot(&mot3);
209 }
210
211 void test_egalite_ensembles_differents(void){
212     EnsembleDeMot e1 = ensembleDeMot();
213     Mot mot1, mot1bis, mot2, mot3, mot4, mot5;
214     creer_mots_A(&mot1, &mot2, &mot3);
215     EDM_ajouter(&e1, mot1);
216     EDM_ajouter(&e1, mot2);
217     EDM_ajouter(&e1, mot3);
218
219     EnsembleDeMot e2 = ensembleDeMot();
220     creer_mots_B(&mot1bis, &mot4, &mot5);
221     EDM_ajouter(&e2, mot1);
222     EDM_ajouter(&e2, mot4);
223     EDM_ajouter(&e2, mot5);
224
225     CU_ASSERT_FALSE(EDM_egale(e1, e2));
```

```
226
227     EDM_vider(&e1);
228     EDM_vider(&e2);
229
230     M_supprimerMot(&mot1);
231     M_supprimerMot(&mot1bis);
232     M_supprimerMot(&mot2);
233     M_supprimerMot(&mot3);
234     M_supprimerMot(&mot4);
235     M_supprimerMot(&mot5);
236 }
237
238 void test_copier(void){
239     EnsembleDeMot e1 = ensembleDeMot();
240     Mot mot1, mot2, mot3;
241     creer_mots_A(&mot1, &mot2, &mot3);
242     EDM_ajouter(&e1, mot1);
243     EDM_ajouter(&e1, mot2);
244     EDM_ajouter(&e1, mot3);
245
246     EnsembleDeMot e2 = EDM_copier(e1);
247
248     CU_ASSERT_TRUE(EDM_egale(e1, e2));
249
250     EDM_vider(&e1);
251     EDM_vider(&e2);
252     M_supprimerMot(&mot1);
253     M_supprimerMot(&mot2);
254     M_supprimerMot(&mot3);
255 }
256
257 void test_obtenir_element(){
258     EnsembleDeMot e = ensembleDeMot();
259     Mot mot1, mot2, mot3, mot3test;
260     creer_mots_A(&mot1, &mot2, &mot3);
261     EDM_ajouter(&e, mot1);
262     EDM_ajouter(&e, mot2);
263     EDM_ajouter(&e, mot3);
264
265     mot3test = EDM_obtenirMot(e);
266     CU_ASSERT_TRUE(M_sontIdentiques(mot3, mot3test));
267
268     EDM_vider(&e);
269     M_supprimerMot(&mot1);
270     M_supprimerMot(&mot2);
271     M_supprimerMot(&mot3);
272 }
273
274 int main(int argc, char **argv){
275     CU_pSuite pSuite = NULL;
276
277     /* initialisation du registre de tests */
278     if (CUE_SUCCESS != CU_initialize_registry())
279         return CU_get_error();
```

```
280
281     /* ajout d'une suite de test */
282     pSuite = CU_add_suite("Tests boite noire", init_suite_success, clean_suite_success);
283     if (NULL == pSuite){
284         CU_cleanup_registry();
285         return CU_get_error();
286     }
287
288     /* Ajout des tests à la suite de tests boite noire */
289     if ((NULL == CU_add_test(pSuite, "1 - La création d'un ensemble doit etre vide", test_ensemble_vide)
        )
290     || (NULL == CU_add_test(pSuite, "2 - Ajouter un mot non present incrémente la cardinalité",
        test_ajouter_non_present))
291     || (NULL == CU_add_test(pSuite, "3 - Ajouter un mot present n'incrémente pas la cardinalité",
        test_ajouter_present))
292     || (NULL == CU_add_test(pSuite, "4 - un élément ajouté est present", test_present_apres_ajout))
293     || (NULL == CU_add_test(pSuite, "5 - retirer un élement présent décrémente la cardinalité",
        test_retirer_non_present))
294     || (NULL == CU_add_test(pSuite, "6 - retirer un élement present ne decremente pas la cardinalite",
        test_retirer_present))
295     || (NULL == CU_add_test(pSuite, "7 - un element retire n'est plus present",
        test_absent_apres_retrait))
296     || (NULL == CU_add_test(pSuite, "8 - union", test_union))
297     || (NULL == CU_add_test(pSuite, "9 - un ensemble est égal a lui meme", test_egalite_meme_ensemble))
298     || (NULL == CU_add_test(pSuite, "10 - un ensemble est different d'un autre ensemble",
        test_egalite_ensembles_differents))
299     || (NULL == CU_add_test(pSuite, "11 - un ensemble est égal a une de ses copies", test_copier))
300     || (NULL == CU_add_test(pSuite, "12 - obtenir un élément d'un ensemble renvoie le dernier élément
        ajouté", test_obtenir_element))){
301         CU_cleanup_registry();
302         return CU_get_error();
303     }
304
305     /* Lancement des tests */
306     CU_basic_set_mode(CU_BRM_VERBOSE);
307     CU_basic_run_tests();
308     printf("\n");
309     CU_basic_show_failures(CU_get_failure_list());
310     printf("\n\n");
311
312     /* Nettoyage du registre */
313     CU_cleanup_registry();
314     return CU_get_error();
315 }
```

../programme/src/testEDM.c

```
1  #include <stdio.h>
2  #include <CUnit/Basic.h>
3  #include <string.h>
4  #include <stdbool.h>
5  #include "Mot.h"
6  #include "ListeChaineeDeMot.h"
7
```

```
8  int init_suite_success(void){
9      return 0;
10 }
11
12 int clean_suite_success(void){
13     return 0;
14 }
15
16 ListeChaineeDeMot creer_liste_avec_un_mot(){
17     ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
18     char chaine1[] = "test";
19     Mot unMot = M_creerUnMot(chaine1);
20     LCDM_ajouter(&l, unMot);
21     return l;
22 }
23
24 ListeChaineeDeMot creer_liste_avec_deux_mot(){
25     ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
26     char *chaine1="chaineun";
27     char *chaine2="chainedeux";
28     Mot unMot = M_creerUnMot(chaine1);
29     Mot unAutreMot = M_creerUnMot(chaine2);
30     LCDM_ajouter(&l, unMot);
31     LCDM_ajouter(&l, unAutreMot);
32     return l;
33 }
34
35 void test_liste_vide(void){
36     ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
37     CU_ASSERT_TRUE(LCDM_estVide(l));
38     LCDM_supprimer(&l);
39 }
40
41 void test_liste_non_vide(void){
42     ListeChaineeDeMot l = creer_liste_avec_un_mot();
43     CU_ASSERT_TRUE(!LCDM_estVide(l));
44     Mot mot = LCDM_obtenirMot(l);
45     M_supprimerMot(&mot);
46     LCDM_supprimer(&l);
47 }
48
49 void test_mot_ajoute_en_tete(void){
50     ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
51     char *chaine1="chaine";
52     Mot unMot = M_creerUnMot(chaine1);
53     LCDM_ajouter(&l, unMot);
54     CU_ASSERT_EQUAL(M_sontIdentiques(LCDM_obtenirMot(l), unMot),true);
55     LCDM_supprimer(&l);
56     M_supprimerMot(&unMot);
57 }
58
59 void test_supprimer_mot(void){
60     ListeChaineeDeMot l1 = LCDM_listeChaineeDeMot();
61     ListeChaineeDeMot l2 = LCDM_listeChaineeDeMot();
```

```
62      char *chaine1="chaineun";
63      char *chaine2="chainedeux";
64      char *chaine3="chainetrois";
65      Mot unMot = M_creerUnMot(chaine1);
66      Mot unAutreMot = M_creerUnMot(chaine2);
67      Mot toujoursPlusDeMot = M_creerUnMot(chaine3);
68      LCDM_ajouter(&l1, unMot);
69      LCDM_ajouter(&l1, unAutreMot);
70      LCDM_ajouter(&l1, toujoursPlusDeMot);
71      LCDM_ajouter(&l2, unMot);
72      LCDM_ajouter(&l2, unAutreMot);
73      LCDM_supprimerMot(&l1,toujoursPlusDeMot);
74      CU_ASSERT_TRUE(LCDM_egale(l1, l2));
75      LCDM_supprimer(&l1);
76      LCDM_supprimer(&l2);
77      M_supprimerMot(&unMot);
78      M_supprimerMot(&unAutreMot);
79      M_supprimerMot(&toujoursPlusDeMot);
80  }
81
82  void test_obtenir_liste_suivante(void){
83      ListeChaineeDeMot lSuivante;
84      ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
85      char *chaine1="chaineun";
86      char *chaine2="chainedeux";
87      Mot unMot = M_creerUnMot(chaine1);
88      Mot unAutreMot = M_creerUnMot(chaine2);
89      LCDM_ajouter(&l, unMot);
90      lSuivante = l;
91      LCDM_ajouter(&l, unAutreMot);
92      CU_ASSERT_PTR_EQUAL(LCDM_obtenirListeSuivante(l), lSuivante);
93      LCDM_supprimer(&l);
94      M_supprimerMot(&unMot);
95      M_supprimerMot(&unAutreMot);
96  }
97
98  void test_fixer_liste_suivante(void){
99      ListeChaineeDeMot l1 = LCDM_listeChaineeDeMot();
100     ListeChaineeDeMot l2 = LCDM_listeChaineeDeMot();
101     ListeChaineeDeMot temp = LCDM_listeChaineeDeMot();
102     char *chaine1="chaineun";
103     char *chaine2="chainedeux";
104     char *chaine3="chainetrois";
105     Mot unMot = M_creerUnMot(chaine1);
106     Mot unAutreMot = M_creerUnMot(chaine2);
107     Mot toujoursPlusDeMot = M_creerUnMot(chaine3);
108     LCDM_ajouter(&l1, unMot);
109     temp = l1;
110     LCDM_ajouter(&l1, unAutreMot);
111     LCDM_ajouter(&l2, toujoursPlusDeMot);
112     LCDM_fixerListeSuivante(&l1, l2);
113     CU_ASSERT_PTR_EQUAL(LCDM_obtenirListeSuivante(l1), l2);
114     LCDM_supprimer(&l1);
115     LCDM_supprimer(&temp);
```

```
116     M_supprimerMot(&unMot);
117     M_supprimerMot(&unAutreMot);
118     M_supprimerMot(&toujoursPlusDeMot);
119 }
120
121 void test_copie_egale(void){
122     ListeChaineeDeMot l1 = creer_liste_avec_deux_mot();
123     ListeChaineeDeMot l2 = LCDM_copier(l1);
124     CU_ASSERT_TRUE(LCDM_egale(l1, l2));
125     Mot mot1 = LCDM_obtenirMot(LCDM_obtenirListeSuivante(l1));
126     Mot mot3 = LCDM_obtenirMot(l1);
127     M_supprimerMot(&mot1);
128     M_supprimerMot(&mot3);
129     LCDM_supprimer(&l1);
130     LCDM_supprimer(&l2);
131 }
132
133 void test_differente(void){
134     ListeChaineeDeMot l1 = creer_liste_avec_deux_mot();
135     ListeChaineeDeMot l2 = creer_liste_avec_un_mot();
136     CU_ASSERT_FALSE(LCDM_egale(l1, l2));
137     Mot mot = LCDM_obtenirMot(l1);
138     Mot mot2 = LCDM_obtenirMot(LCDM_obtenirListeSuivante(l1));
139     Mot mot4 = LCDM_obtenirMot(l2);
140     M_supprimerMot(&mot);
141     M_supprimerMot(&mot2);
142     M_supprimerMot(&mot4);
143     LCDM_supprimer(&l1);
144     LCDM_supprimer(&l2);
145 }
146
147 int main(int argc, char **argv){
148     CU_pSuite pSuite = NULL;
149
150     /* initialisation du registre de tests */
151     if (CUE_SUCCESS != CU_initialize_registry()){
152         return CU_get_error();
153     }
154     /* ajout d'une suite de test */
155     pSuite = CU_add_suite("Tests boite noire", init_suite_success, clean_suite_success);
156     if (NULL == pSuite){
157         CU_cleanup_registry();
158         return CU_get_error();
159     }
160
161     /* Ajout des tests     la suite de tests boite noire */
162     if ((NULL == CU_add_test(pSuite, "la creation d'une liste qui doit etre vide", test_liste_vide))
163     || (NULL == CU_add_test(pSuite, "une liste contenant un element n'est pas vide", test_liste_non_vide
        ))
164     || (NULL == CU_add_test(pSuite, "un element ajoute est en tete de liste", test_mot_ajoute_en_tete))
165     || (NULL == CU_add_test(pSuite, "supprimer un mot", test_supprimer_mot))
166     || (NULL == CU_add_test(pSuite, "obtenir liste suivante", test_obtenir_liste_suivante))
167     || (NULL == CU_add_test(pSuite, "fixer liste suivante", test_fixer_liste_suivante))
168     || (NULL == CU_add_test(pSuite, "une liste et sa copie sont egales", test_copie_egale))
```

```
169     || (NULL == CU_add_test(pSuite, "deux listes differentes ne sont pas egales", test_differente))){
170         CU_cleanup_registry();
171         return CU_get_error();
172     }
173 
174     /* Lancement des tests */
175     CU_basic_set_mode(CU_BRM_VERBOSE);
176     CU_basic_run_tests();
177     printf("\n");
178     CU_basic_show_failures(CU_get_failure_list());
179     printf("\n\n");
180 
181     /* Nettoyage du registre */
182     CU_cleanup_registry();
183     return CU_get_error();
184 }
```

../programme/src/testLCDM.c

```
1  #include <stdio.h>
2  #include <CUnit/Basic.h>
3  #include <string.h>
4  #include <stdbool.h>
5  #include "Mot.h"
6 
7 
8  int init_suite_success(void){
9      return 0;
10 }
11 
12 int clean_suite_success(void){
13     return 0;
14 }
15 
16 Mot creer_un_mot(){
17     Mot mot;
18     mot.chaine = "lala";
19     mot.longueur = strlen(mot.chaine);
20     return mot;
21 }
22 
23 Mot creer_mot_vide(){
24     Mot mot;
25     mot.chaine = "";
26     mot.longueur = 0;
27     return mot;
28 }
29 
30 char creer_cara_alpha_accent(){
31     return 'é';
32 }
33 
34 char creer_cara_alpha_sans_accent(){
35     return 'a';
```

```
36  }
37
38  char creer_cara_pas_alpha(){
39      return '!';
40  }
41
42  char* creer_mot_valide_avec_accent(){
43      return "fatiguée";
44  }
45
46  char* creer_mot_valide_sans_accent(){
47      return "valide";
48  }
49
50  char* creer_mot_non_valide(){
51      return "val!de";
52  }
53
54  void test_caractere_alpha_accent(){
55      char c = creer_cara_alpha_accent();
56      CU_ASSERT_TRUE(M_estUnCaractereAlphabetique(c));
57  }
58
59  void test_caractere_alpha_sans_accent(){
60      char c = creer_cara_alpha_sans_accent();
61      CU_ASSERT_TRUE(M_estUnCaractereAlphabetique(c));
62  }
63
64  void test_pas_caractere_alpha(){
65      char c = creer_cara_pas_alpha();
66      CU_ASSERT_TRUE(!M_estUnCaractereAlphabetique(c));
67  }
68
69  void test_mot_valide_avec_accent(){
70      char *c=creer_mot_valide_avec_accent();
71      CU_ASSERT_TRUE(M_estUnMotValide(c));
72  }
73
74  void test_reduire_la_casse(){
75      char *c = creer_mot_valide_avec_accent();
76      char *c1 = "FAtiguÉe";
77      Mot mot = M_creerUnMot(c);
78      Mot mot1 = M_creerUnMot(c1);
79      CU_ASSERT_TRUE(M_sontIdentiques(mot,mot1));
80      M_supprimerMot(&mot);
81      M_supprimerMot(&mot1);
82  }
83
84  void test_mot_valide_sans_accent(){
85      char *c=creer_mot_valide_sans_accent();
86      CU_ASSERT_TRUE(M_estUnMotValide(c));
87  }
88
89  void test_mot_non_valide(){
```

```
90      char *c=creer_mot_non_valide();
91      CU_ASSERT_TRUE(!M_estUnMotValide(c));
92  }
93
94
95  void test_copier_mot(){
96      Mot mot = M_creerUnMot(creer_mot_valide_avec_accent());
97      Mot mot2 = M_copierMot(mot);
98      CU_ASSERT_TRUE(M_sontIdentiques(mot,mot2));
99      M_supprimerMot(&mot);
100     M_supprimerMot(&mot2);
101 }
102
103 void test_ieme_caractere(){
104     Mot mot = M_creerUnMot(creer_mot_valide_avec_accent());
105     CU_ASSERT_TRUE(M_iemeCaractere(mot,1)=='f');
106     CU_ASSERT_TRUE(M_iemeCaractere(mot,2)=='a');
107     CU_ASSERT_TRUE(M_iemeCaractere(mot,3)=='t');
108     CU_ASSERT_TRUE(M_iemeCaractere(mot,7)=='é');
109     M_supprimerMot(&mot);
110 }
111
112 void test_sont_identiques(){
113     Mot mot = M_creerUnMot(creer_mot_valide_avec_accent());
114     Mot mot2 = M_copierMot(mot);
115     CU_ASSERT_TRUE(M_sontIdentiques(mot,mot2));
116     M_supprimerMot(&mot);
117     M_supprimerMot(&mot2);
118 }
119
120 void test_supprimer_ieme_lettre(){
121     char *c = "fatguée";
122     Mot mot = M_creerUnMot(creer_mot_valide_avec_accent());
123     Mot mot3 = M_copierMot(mot);
124     Mot mot2 = M_creerUnMot(c);
125     M_supprimerIemeLettre(&mot,4);
126     CU_ASSERT_TRUE(M_sontIdentiques(mot,mot2));
127     CU_ASSERT_TRUE(!M_sontIdentiques(mot,mot3));
128     M_supprimerMot(&mot);
129     M_supprimerMot(&mot2);
130     M_supprimerMot(&mot3);
131 }
132
133 void test_inverser_deux_lettres_consecutives(){
134     char *c = "fatiguée";
135     char *c1 = "fatgiuée";
136     Mot mot = M_creerUnMot(c);
137     Mot mot2 = M_creerUnMot(c1);
138     M_inverserDeuxLettresConsecutives(&mot,4);
139     CU_ASSERT_TRUE(M_sontIdentiques(mot,mot2));
140     M_supprimerMot(&mot);
141     M_supprimerMot(&mot2);
142 }
143
```

```
144 void test_inserer_lettre(){
145     char *c = "faiguée";
146     char *c1 = "fatiguée";
147     Mot mot = M_creerUnMot(c);
148     Mot mot2 = M_creerUnMot(c1);
149     M_insererLettre(&mot, 3,'t');
150     CU_ASSERT_TRUE(M_sontIdentiques(mot,mot2));
151     M_supprimerMot(&mot);
152     M_supprimerMot(&mot2);
153 }
154
155 void test_decomposer_mot(){
156     char *c = "fati";
157     char *c1 = "guée";
158     Mot mot = M_creerUnMot(c);
159     Mot mot1 = M_creerUnMot(c1);
160     Mot mot2 = M_creerUnMot(creer_mot_valide_avec_accent());
161     Mot mot3 = M_decomposerMot(&mot2,5);
162     CU_ASSERT_TRUE(M_sontIdentiques(mot3,mot));
163     CU_ASSERT_TRUE(M_sontIdentiques(mot2,mot1));
164     M_supprimerMot(&mot);
165     M_supprimerMot(&mot1);
166     M_supprimerMot(&mot2);
167     M_supprimerMot(&mot3);
168 }
169
170
171 int main(int argc, char **argv){
172     CU_pSuite pSuite = NULL;
173
174     /* initialisation du registre de tests */
175     if (CUE_SUCCESS != CU_initialize_registry()){
176         return CU_get_error();
177     }
178     /* ajout d'une suite de test */
179     pSuite = CU_add_suite("Tests boite noire", init_suite_success, clean_suite_success);
180     if (NULL == pSuite){
181         CU_cleanup_registry();
182         return CU_get_error();
183     }
184
185     /* Ajout des tests     la suite de tests boite noire */
186     if ((NULL == CU_add_test(pSuite, "un caractere avec accent est un caractere alphabetique",
        test_caractere_alpha_accent))
187     || (NULL == CU_add_test(pSuite, "un caractere sans accent est un caractere alphabetique",
        test_caractere_alpha_sans_accent))
188     || (NULL == CU_add_test(pSuite, "un caractere non alphabetique ne l'est pas",
        test_pas_caractere_alpha))
189     || (NULL == CU_add_test(pSuite, "un mot non valide n'est pas valide", test_mot_non_valide))
190     || (NULL == CU_add_test(pSuite, "un mot valide avec accent est valide", test_mot_valide_avec_accent)
        )
191     || (NULL == CU_add_test(pSuite, "un mot valide sans accent est valide", test_mot_valide_sans_accent)
        )
192     || (NULL == CU_add_test(pSuite, "creer un mot reduit la casse", test_reduire_la_casse))
```

```
193     || (NULL == CU_add_test(pSuite, "la copie est identique au mot", test_copier_mot))
194     || (NULL == CU_add_test(pSuite, "la copie et le mot sont identiques", test_sont_identiques))
195     || (NULL == CU_add_test(pSuite, "ieme caractere renvoie bien le ieme caractere", test_ieme_caractere
        ))
196     || (NULL == CU_add_test(pSuite, "supprimer ieme caractere supprimer bien le ieme caractere",
        test_supprimer_ieme_lettre))
197     || (NULL == CU_add_test(pSuite, "inverser deux lettres fontionne",
        test_inverser_deux_lettres_consecutives))
198     || (NULL == CU_add_test(pSuite, "inserer une lettre a la bonne place", test_inserer_lettre))
199     || (NULL == CU_add_test(pSuite, "decompo ok", test_decomposer_mot))
200
201     ){
202         CU_cleanup_registry();
203         return CU_get_error();
204     }
205
206     /* Lancement des tests */
207     CU_basic_set_mode(CU_BRM_VERBOSE);
208     CU_basic_run_tests();
209     printf("\n");
210     CU_basic_show_failures(CU_get_failure_list());
211     printf("\n\n");
212
213     /* Nettoyage du registre */
214     CU_cleanup_registry();
215     return CU_get_error();
216 }
```

../programme/src/testMot.c

7 Organisation

Du début à la fin de ce projet, nous nous sommes réparti le travail entre les différents TADs. De l'analyse aux tests, nous nous sommes efforcés de travailler chacun à notre tour sur un TAD différent afin que tous les membres du groupe aient une vision d'ensemble du projet.

Voici le tableau de répartition globale du travail :

	Fatiha	**Noé**	**Florine**	**Loïck**
TAD	Mot	Correcteur Orthographique	Dictionnaire	Dictionnaire
CP	Dictionnaire	Mot	Fichier Texte	Correcteur Orthographique
CD	Correcteur Orthographique	Dictionnaire	Mot	Rapport
Implémentation	Rapport	Correcteur Orthographique	Dictionnaire	Mot
Tests	Mot	Dictionnaire et Rapport	Correcteur Orthographique	Dictionnaire

Table 1: Répartition des tâches liées aux TAD

8 Conclusions personnelles

Noé: Ce projet aura été pour moi un mélange de deux extrêmes. En effet, d'un côté, le travail en groupe s'est très bien passé, l'entraide, la communication et la réactivité ont été présentes tout au long du projet. J'ai pu renforcer ma prise en main de git ainsi qu'apprendre à mener un projet, en étant séparé physiquement des autres membres (peu de réunions en présentiel). Il s'agit du premier projet où nous avons scrupuleusement suivi les différentes étapes d'un cycle en V et je suis très heureux de l'avoir fait, car, même si pénibles et très incertaines dans un premier temps, les étapes de spécification et conception aident énormément pour le développement. J'ai finalement pu développer de solides compétences en C et dans l'utilisation de debuggers (tels que valgrind ou ddd), ainsi qu'une plus grande rigueur dans la réalisation du code comme par exemple le nommage des fonctions ou la répartition des tâches (on ne code pas tout tout seul). Cependant, je regrette que la charge de travail induite par ce projet, en addition à celle déjà trop importante du département ITI, nous ai forcé, comme beaucoup de groupes, j'imagine, de sacrifier nos vacances de Noël, par manque de temps pour avancer convenablement sur ce travail au cours du semestre.

Loïck:

Fatiha: Pour conclure, ce projet m'a permis de découvrir plusieurs choses et d'acquérir plusieurs connaissances. En premier lieu, j'ai appris comment utiliser l'environnement git et le langage Latex. En deuxième lieu, la relation que j'ai entretenue avec l'équipe, m'a beaucoup appris sur le travail de groupe. Néanmoins, au niveau du développement, j'ai eu du mal à coder en langage C ainsi qu'à corriger les erreurs des codes. Finalement, je n'ai qu'à adresser mes vifs remerciements à mes camarades du groupe qui m'ont beaucoup aidé durant toute la période du projet et qui ont rendu ce projet possible.

Florine:
J'ai beaucoup apprécié réaliser ce projet. Travailler avec cette équipe était un véritable plaisir. C'était ma première expérience en tant que cheffe de projet, ce qui m'a permis de gagner de l'expérience en gestion de projet, mais aussi en communication. La gestion du planning était particulièrement difficile, nous avons pris un peu de retard ce qui nous a amené à beaucoup travailler pendant les vacances de Noël. J'ai pu mettre en œuvre beaucoup de mes compétences et ainsi progresser dans des domaines variés. De la rédaction du rapport au code, nous avons tous participé activement, dans une bonne ambiance et beaucoup d'entraide. Comme le C ne possède pas de ramasse-miettes, j'ai également beaucoup progressé en gestion de fuites mémoires notamment grâce à valgrind.

9 Conclusion Générale

En conclusion, ce projet a été une expérience enrichissante qui nous a tous permis de progresser et développer de nouvelles compétences.
Nous avons énormément progressé en langage C, mais aussi en gestion de projet grâce à l'outil git, ainsi qu'en LaTeXpour la rédaction du rapport. C'était aussi l'occasion de mettre en pratique les connaissances théoriques vues au cours du semestre. Pour la plupart d'entre nous, c'était la première fois que nous faisions face à un projet d'une telle ampleur, à réaliser en autonomie presque complète. Nous sommes donc fiers de pouvoir présenter un programme fini et fonctionnel. Nous avons tous profité de ce projet pour progresser en programmation et notamment en langage C, mais nous avons aussi acquis de l'expérience en gestion de projet.
Nous avons correctement réparti les charges de travail entre les différents membres de l'équipe et nous avons su communiquer et nous adapter face aux difficultés que nous avons rencontrées tout au long du projet. La gestion du temps de travail et de l'estimation du temps de résolution d'un problème étaient ce qui nous a le plus posé de souci. En effet, le temps passé sur ce projet a augmenté de façon exponentielle. Nous ne sommes pas allées assez vite les premières semaines. Pour rattraper le retard ainsi engendré, nous avons drastiquement augmenté notre temps passé sur ce projet. Grâce à notre cohésion d'équipe, nous avons pu terminer le projet à temps.