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 nous sommes adaptés aux autres, notamment en codant 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, et nous avons appris 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'accéder au mot à corriger

obtenirDictionnaire : Permet d'accéder au dictionnaire

obtenirCorrections : Permet d'accéder 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)

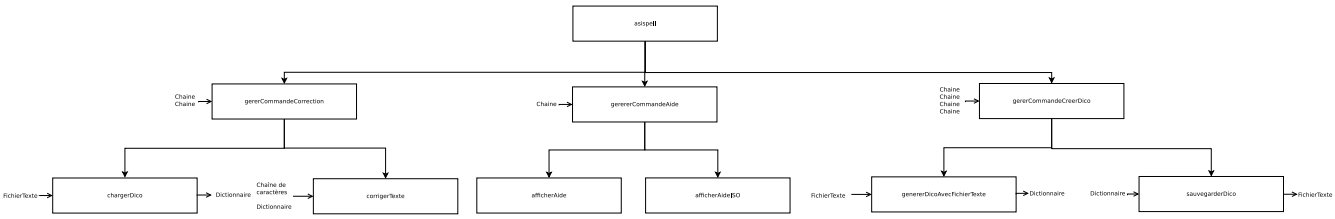
finFichier(f): mode(f)=lecture

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

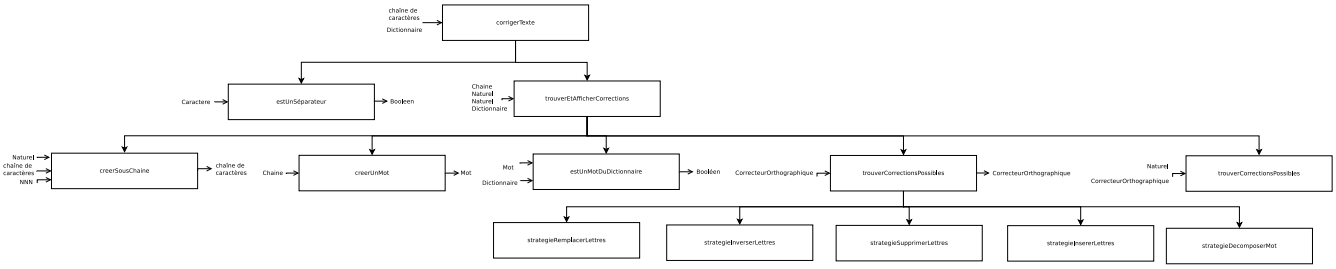
ecrireXX(f): estOuvert(f) et mode(f)=ecriture

3 Analyses Descendantes

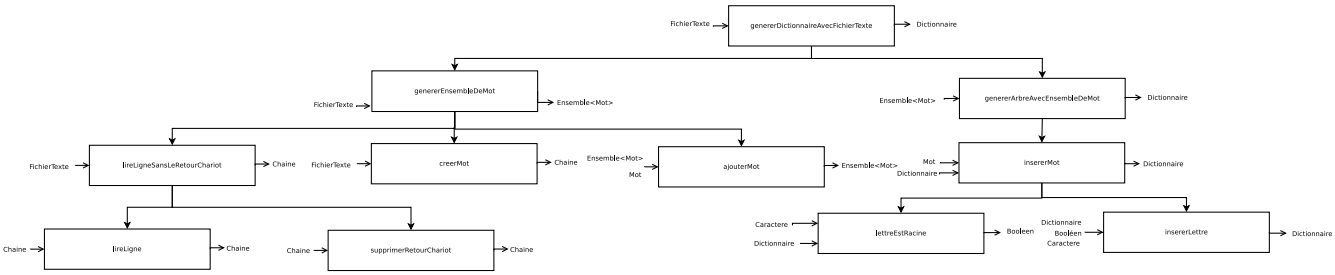
3.1 Analyse Descendante Générale



3.2 Analyse Descendante de corrigerTexte



3.3 Analyse Descendante de genererDictionnaireAvecFichierTexte



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)

procedure supprimerMot(**E/S** 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

debut
  retourner *unMot.longueur*;
fin

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

Precondition(s) $1 \leq position \leq longueur(unMot)$;
Declaration :
debut
  retourner *ChaineDeCaractere.iemeCaractere(unMot.chaine,position)*;
fin

Fonction sontIdentiques(unMot,unAutreMot : Mot):Booleen

Declaration : i : Naturel, egaux : Booleen
debut
  si $longueur(unMot) \neq longueur(unAutreMot)$ **alors**
    retourner *FAUX*
  finsi
  sinon
    egaux ← VRAI ;
    i ← 1 ;
    tant que $(i \leq longueur(unMot))$ *et (egaux)* **faire**
      egaux ← accederAuIemeCaractere(unAutreMot,i);
      i ← i+1 ;
    finTantQue
    retourner *egaux*;
  finsi
fin

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

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

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

Precondition(s) *2 ≤ position ≤ longueur(mot)*;
Declaration : chaineGauche : Chaine De Caractere, i : Naturel
pour *i ← 1 à position* **faire**
    ChaineDeCaractere.fixerIemeCaractere(chaineGauche,i,iemeCaractere(unMot,1));
    supprimerIemeLettre(unMot,1);
finPour
motGauche ← creerUnMot(chaineGauche)

Arbre de lettres

Conception préliminaire

Structure

Type ArbreDeLettres = Structure

fils, frere : ArbreDeLettres

lettre : Caractere

estFinDeMot : Booleen

finStructure

Signatures

fonction creerADLVide() : ArbreDeLettres

fonction creerADL(fils : ArbreDeLettres,frere : ArbreDeLettres,lettre : Caractere,estFinDeMot : Booleen) : ArbreDeLettres

procedure fixerEstFinDeMot(**E/S** arbre : ArbreDeLettres, **E** estFinDeMot : Booleen)

procedure fixerLettre(**E/S** arbre : ArbreDeLettres, **E** lettre : Caractere)

procedure fixerElement(**E/S** arbre : ArbreDeLettres, **E** lettre : Caractere, estFinDeMot : Booleen)

procedure fixerFrere(**E/S** arbre : ArbreDeLettres, **E** frere : ArbreDeLettres)

procedure fixerFils(**E/S** arbre : ArbreDeLettres, **E** fils : ArbreDeLettres)

fonction obtenirFils(arbre : ArbreDeLettres) : ArbreDeLettres

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

fonction obtenirFrere(arbre : ArbreDeLettres) : ArbreDeLettres

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

fonction obtenirLettre(arbre : ArbreDeLettres) : Caractere

fonction obtenirEstFinDeMot(arbre : ArbreDeLettres) : Booleen

procedure supprimer(**E/S** arbre : ArbreDeLettres)

Dictionnaire

Conception préliminaire

Dictionnaire = ArbreDeLettres

Signatures

fonction genererDicoAvecEnsembleDeMot(lesMots : Ensemble<Mot>): Dictionnaire

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

fonction estUnMotDuDictionnaire(unDico : Dictionnaire, unMot : Mot) : Booléen

fonction chargerDico(unFichier : FichierTexte) :Dictionnaire

fonction sauvegarderDico(unDico : Dictionnaire) : FichierTexte

Signatures des sous fonctions

fonction lettreEstRacine(unDico : Dictionnaire uneLettre : Caractere):Booléen

procedure insererLettre (E/S : unDico : Dictionnaire, E : uneLettre : Caractere, estFinDeMot : Booleen)

procedure insererMot(E/S : unDico : Dictionnaire, E : unMot : Mot)

procedure chargerDicoR(E/S, unDico : Dictionnaire, E : unFichier : FichierTexte)

procedure sauvegarderDicoR(E/S, unDico : Dictionnaire,: unFichier : FichierTexte)

Conception détaillée

Procédure chargerDicoR(E/S, unDico : Dictionnaire, E : unFichier : FichierTexte)

```
Declaration :
temp : Dictionnaire
element : Chaine
lettre, estFinDeMot, aFils, aFrere : Caractere
debut
    element ← lireElement(unFichier)
    lettre ← element[0]
    estFinDeMot ← element[1]
    aFils ← element[2]
    aFrere ← element[3]
    unDico ← creerADL(creerADLVide(), creerADLVide(), lettre, caractereEnBooleen(estFinDeMot))
    si caractereEnBooleen(aFils) alors
        chargerDicoR(temp, unFichier)
        fixerFils(unDico, temp)
    finsi
    si caractereEnBooleen(aFrere) alors
        chargerDicoR(temp, unFichier)
        fixerFrere(unDico, temp)
    finsi
fin
```

Procédure sauvegarderDicoR(E/S, unDico : Dictionnaire,: unFichier : FichierTexte)

```
Declaration :
tempFils, tempFrere : Dictionnaire
debut
    si non(estVide(unDico)) alors
        ecrireCaractere(unFichier, obtenirLettre(unDico)
        ecrireCaractere(unFichier, booleenEnCaractere(obtenirEstFinDeMot(unDico))
        tempFils ← obtenirFils(unDico)
        tempFrere ← obtenirFrere(unDico)
        ecrireCaractere(unFichier, booleenEnCaractere(non(estVide(tempFils))))
        ecrireCaractere(unFichier, booleenEnCaractere(non(estVide(tempFils))))
        sauvegarderDicoR(tempFils, unFichier)
        sauvegarderDicoR(tempFrere, unFichier)
    finsi
fin
```

```
Fonction estUnMotDuDictionnaire(unDico : Dictionnaire, unMot : Mot) : Booléen
    Declaration :
    temp : Dictionnaire
    debut
        si longueurMot(unMot) = 1 alors
            si non(estVide(unDico)) alors
                si iemeCaractere(unMot, 1) = obtenirLettre(unDico) alors
                    retourner obtenirEstFinDeMot(unDico)
                finsi
                sinon
                    temp ← obtenirFrere(unDico)
                    retourner estUnMotDuDictionnaire(temp, unMot)
                finsi
            finsi
            sinon
                retourner FAUX
            finsi
        finsi
        sinon
            si non(estVide(unDico)) alors
                si iemeCaractere(unMot, 1) = obtenirLettre(unDico) alors
                    supprimerIemeLettre(unMot, 1)
                    temp ← obtenirFils(unDico)
                    retourner estUnMotDuDictionnaire(temp, unMot
                finsi
                sinon
                    temp ← obtenirFrere(unDico)
                    retourner estUnMotDuDictionnaire(temp, unMot
                finsi
            finsi
            sinon
                retourner FAUX
            finsi
        finsi
    fin
```

```
Procédure insererMot(E/S : unDico : Dictionnaire, E : unMot : Mot)
    Declaration :
    temp : Dictionnaire
    estFinDeMot : Booleen
    debut
        estFinDeMot ← FAUX
        si longueurMot(unMot) = 1 alors
            estFinDeMot ← VRAI
            si estVide(unDico) alors
                insererLettre(unDico, iemeCaractere(unMot, 1), enFinDeMot)
            finsi
            sinon
                si lettreEstRacine(unDico, iemeCaractere(unMot, 1)) alors
                    fixerEstFinDeMot(unDico, estFinDeMot)
                finsi
                sinon
                    temp ← obtenirFrere(unDico)
                    insererMot(temp, unMot)
                    fixerFrere(unDico, temp)
                finsi
            finsi
        finsi
        sinon
            si estVide(unDico) alors
                insererLettre(unDico, iemeCaractere(unMot, 1), enFinDeMot)
                supprimerIemeLettre(unMot, 1)
                temp ← obtenirFils(unDico)
                insererMot(temp, unMot)
                fixerFils(unDico, temp)
            finsi
            sinon
                si lettreEstRacine(unDico, iemeCaractere(unMot, 1)) alors
                    supprimerIemeLettre(unMot, 1)
                    temp ← obtenirFils(unDico)
                    insererMot(temp, unMot)
                    fixerFils(unDico, temp)
                finsi
                sinon
                    temp ← obtenirFrere(unDico)
                    insererMot(temp, unMot) fixerFrere(unDico, temp)
                finsi
            finsi
        finsi
    fin
```

Correcteur Orthographique

Conception préliminaire

Type CorrecteurOrthographique = Structure

motACorriger : Mot

leDictionnaire : Naturel

lesCorrections : EnsembleDeMot

finStructure

Signatures

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

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

fonction obtenirMotACorriger(unCorrecteur : CorrecteurOrthographique) : Mot

fonction obtenirDictionnaire(unCorrecteur : CorrecteurOrthographique) : Dictionnaire

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

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

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

procedure ajouterNouvellesCorrections(**E/S** unCorrecteur : CorrecteurOrthographique,**E** desCorrections : EnsembleDeMot)

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

fonction remplacerIemeLettreEnBoucle(unMot : Mot, indice : Naturel) : EnsembleDeMot

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

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

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

fonction insererIemeLettreEnBoucle(unMot : Mot, indice : Naturel) : EnsembleDeMot

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

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

procedure supprimerCorrecteur(**E** unCorrecteur : CorrecteurOrthographique)

Conception détaillée

Fonction correcteur(Dictionnaire : unDico, Mot : unMotFaux):CorrecteurOrthographique

Precondition(s) *non(Dictionnaire.estUnMotDuDictionnaire(unDico,Mot.copierMot(unMotFaux)))*;
Declaration : unCorrecteur : CorrecteurOrthographique
debut
    unCorrecteur.leDictionnaire ← unDico;
    unCorrecteur.motACorriger ← Mot.copierMot(unMotFaux);
    unCorrecteur.lesCorrections ← EnsembleDeMot.ensembleDeMot();
    retourner *unCorrecteur*;
fin

Procédure ajouterNouvellesCorrections(**E/S** unCorrecteur : CorrecteurOrthographique, **E** desCorrections : EnsembleDeMot)

Declaration : temp : EnsembleDeMot
debut
    temp ← unCorrecteur.lesCorrections;
    unCorrecteur.lesCorrections ← EnsembleDeMot.union(desCorrections,temp);
fin

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

debut
    strategieRemplacerLettres(unCorrecteur);
    strategieInverserDeuxLettresConsecutives(unCorrecteur);
    strategieSupprimerLettres(unCorrecteur);
    strategieInsererLettres(unCorrecteur);
    strategieDecomposerMot(unCorrecteur);
fin

Fonction remplacerIemeLettreEnBoucle(mot : Mot, position : Naturel):EnsembleDeMot

Declaration : desCorrections : EnsembleDeMot, uneCorrection : Mot, lettres : Chaine de caractere
debut
    desCorrections=EnsembleDeMot.ensembleDeMot() lettres = "abcdefghijklmnopqrstuvwxyzàéèëùûêîïçôö-";
    pour *j ← 0 à longueur(lettres)-1* **faire**
        uneCorrection ← Mot.copierMot(mot) Mot.fixierIemeCaractere(uneCorrection,i,lettres[j]);
        EnsembleDeMot.ajouter(desCorrections,uneCorrection);
        Mot.insererLettre(mot2, i, c);
    finPour
    retourner *desCorrections*
fin

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

Declaration : i,longueur : Naturel, uneCorrection, leMotACorriger : Mot, desCorrections, correctionsCourantes : EnsembleDeMot, leDico : Dictionnaire

debut

```
leMotACorriger ← CorrecteurOrthographique.obtenirMotACorriger(unCorrecteur);
leDico ← CorrecteurOrthographique.obtenirDictionnaire(unCorrecteur);
longueur ← Mot.longueur(leMotACorriger);
desCorrections ← EnsembleDeMot.ensembleDeMot();
correctionsCourantes ← EnsembleDeMot.ensembleDeMot();
pour i ← 1 à longueur faire
    correctionsCourantes = CorrecteurOrthographique.remplacerIemeLettreEnBoucle(leMotACorriger,i);
    tant que EnsembleDeMot.cardinalité(correctionsCourantes) ≠ 0 faire
        uneCorrection ← EnsembleDeMot.obtenirMot(correctionsCourantes);
        EnsembleDeMot.retirer(correctionsCourantes,uneCorrection);
        si Dictionnaire.estUnMotDuDictionnaire(leDico,Mot.copierMot(uneCorrection)) et
          non(EnsembleDeMot.estPresent(desCorrections,uneCorrection)) alors
            EnsembleDeMot.ajouter(desCorrections,uneCorrection);
        finsi
    finTantQue
    CorrecteurOrthographique.ajouterNouvellesCorrections(uneCorrection);
finPour
```

fin

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

Declaration : i,longueur : Naturel, uneCorrection, leMotACorriger : Mot, desCorrections : EnsembleDeMot, leDico : Dictionnaire

debut

```
leMotACorriger ← CorrecteurOrthographique.obtenirMotACorriger(unCorrecteur);
leDico ← CorrecteurOrthographique.obtenirDictionnaire(unCorrecteur);
longueur ← Mot.longueur(leMotACorriger);
desCorrections ← EnsembleDeMot.ensembleDeMot();
pour i ← 1 à longueur-1 faire
    uneCorrection ← Mot.copierMot(mot);
    Mot.supprimerIemeLettre(uneCorrection,i);
    si Dictionnaire.estUnMotDuDictionnaire(leDico,Mot.copierMot(uneCorrection)) et
      non(EnsembleDeMot.estPresent(desCorrections,uneCorrection)) alors
        EnsembleDeMot.ajouter(desCorrections,uneCorrection);
    finsi
finPour
CorrecteurOrthographique.ajouterNouvellesCorrections(uneCorrection);
```

fin

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

Declaration : i,longueur : Naturel, uneCorrection, leMotACorriger : Mot, desCorrections : EnsembleDeMot, leDico : Dictionnaire

debut

```
leMotACorriger ← CorrecteurOrthographique.obtenirMotACorriger(unCorrecteur);
leDico ← CorrecteurOrthographique.obtenirDictionnaire(unCorrecteur);
longueur ← Mot.longueur(leMotACorriger);
desCorrections ← EnsembleDeMot.ensembleDeMot();
pour i ← 1 à longueur faire
    uneCorrection ← Mot.copierMot(mot);
    Mot.inverserDeuxLettresConsecutives(uneCorrection,i);
    si Dictionnaire.estUnMotDuDictionnaire(leDico,Mot.copierMot(uneCorrection)) et
     non(EnsembleDeMot.estPresent(desCorrections,uneCorrection)) alors
        EnsembleDeMot.ajouter(desCorrections,uneCorrection);
    finsi
finPour
CorrecteurOrthographique.ajouterNouvellesCorrections(uneCorrection);
```

fin

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

Declaration : i,longueur : Naturel, uneCorrection, leMotACorriger : Mot,correctionsCourantes, desCorrections : EnsembleDeMot, leDico : Dictionnaire

debut

```
leMotACorriger ← CorrecteurOrthographique.obtenirMotACorriger(unCorrecteur);
leDico ← CorrecteurOrthographique.obtenirDictionnaire(unCorrecteur);
longueur ← Mot.longueur(leMotACorriger);
desCorrections ← EnsembleDeMot.ensembleDeMot();
correctionsCourantes ← EnsembleDeMot.ensembleDeMot();
pour i ← 1 à longueur) faire
    correctionsCourantes ← CorrecteurOrthographique.insererIemeLettreEnBoucle(leMotACorriger,i);
    tant que EnsembleDeMot.cardinalité(correctionsCourantes)≠ 0 faire
        uneCorrection ← EnsembleDeMot.obtenirMot(correctionsCourantes);
        EnsembleDeMot.retirer(correctionsCourantes,uneCorrection);
        si Dictionnaire.estUnMotDuDictionnaire(leDico,Mot.copierMot(uneCorrection)) et
         non(EnsembleDeMot.estPresent(desCorrections,uneCorrection)) alors
            EnsembleDeMot.ajouter(desCorrections,uneCorrection);
        finsi
    finTantQue
    CorrecteurOrthographique.ajouterNouvellesCorrections(uneCorrection);
finPour
```

fin

```
Procédure strategieDecomposerMot(E/S unCorrecteur : CorrecteurOrthographique)
  Declaration : i,longueur : Naturel, uneCorrection, unMotModifiable, leMotACorriger : Mot, desCorrections :
                EnsembleDeMot, leDico : Dictionnaire
  debut
    leMotACorriger ← CorrecteurOrthographique.obtenirMotACorriger(unCorrecteur);
    leDico ← CorrecteurOrthographique.obtenirDictionnaire(unCorrecteur);
    longueur ← Mot.longueur(leMotACorriger);
    desCorrections ← EnsembleDeMot.ensembleDeMot();
    pour i ← 2 à longueur-2 faire
      unMotModifiable = Mot.copierMot(leMotACorriger);
      uneCorrection ← Mot.decomposerMot(unMotModifiable,i);
      si Dictionnaire.estUnMotDuDictionnaire(leDico,Mot.copierMot(uneCorrection)) et
       Dictionnaire.estUnMotDuDictionnaire(leDico,Mot.copierMot(unMotModifiable)) alors
        EnsembleDeMot.ajouter(desCorrections,uneCorrection);
        EnsembleDeMot.ajouter(desCorrections,unMotModifiable);
      finsi
    finPour
    CorrecteurOrthographique.ajouterNouvellesCorrections(uneCorrection);
  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)

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;
```

```
52      (*arbre)->frere = frere;
53  }
54
55  void ADL_fixerFils(ArbreDeLettres *arbre, ArbreDeLettres fils)
56  {
57      assert(!ADL_estVide(*arbre));
58      errno = 0;
59      (*arbre)->fils = fils;
60  }
61
62  ArbreDeLettres ADL_obtenirFils(ArbreDeLettres arbre)
63  {
64      assert(!ADL_estVide(arbre));
65      errno = 0;
66      return arbre->fils;
67  }
68
69  ArbreDeLettres ADL_obtenirFrere(ArbreDeLettres arbre)
70  {
71      assert(!ADL_estVide(arbre));
72      errno = 0;
73      return arbre->frere;
74  }
75
76  char ADL_obtenirLettre(ArbreDeLettres arbre)
77  {
78      assert(!ADL_estVide(arbre));
79      errno = 0;
80      return arbre->lettre;
81  }
82
83  int ADL_obtenirEstFinDeMot(ArbreDeLettres arbre)
84  {
85      assert(!ADL_estVide(arbre));
86      errno = 0;
87      return arbre->estFinDeMot;
88  }
89
90  void ADL_supprimer(ArbreDeLettres *arbre)
91  {
92      ArbreDeLettres tmp = ADL_creerADLVide();
93      if (!ADL_estVide(*arbre))
94      {
95          tmp = ADL_obtenirFils(*arbre);
96          ADL_supprimer(&tmp);
97          tmp = ADL_obtenirFrere(*arbre);
98          ADL_supprimer(&tmp);
99      }
100     free(*arbre);
101 }
```

../programme/src/ArbreDeLettres.c

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

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

CorrecteurOrthographique CO_correcteur(Dictionnaire unDico, Mot unMotFaux)
{
    assert(!D_estUnMotDuDictionnaire(unDico, M_copierMot(unMotFaux)));
    CorrecteurOrthographique unCorrecteur;
    unCorrecteur.leDictionnaire = unDico;
    unCorrecteur.motACorriger = M_copierMot(unMotFaux);
    unCorrecteur.lesCorrections = ensembleDeMot();
    return unCorrecteur;
}

Mot CO_obtenirMotACorriger(CorrecteurOrthographique unCorrecteur)
{
    return unCorrecteur.motACorriger;
}

Dictionnaire CO_obtenirDictionnaire(CorrecteurOrthographique unCorrecteur)
{
    return unCorrecteur.leDictionnaire;
}

EnsembleDeMot CO_obtenirCorrections(CorrecteurOrthographique unCorrecteur)
{
    return unCorrecteur.lesCorrections;
}

void CO_fixerDico(CorrecteurOrthographique *unCorrecteur, Dictionnaire unDico)
{
    unCorrecteur->leDictionnaire = unDico;
}

void CO_fixerMotACorriger(CorrecteurOrthographique *unCorrecteur, Mot unMotFaux)
{
    assert(!D_estUnMotDuDictionnaire(CO_obtenirDictionnaire(*unCorrecteur), unMotFaux));
    unCorrecteur->motACorriger = unMotFaux;
}

void CO_ajouterNouvellesCorrections(CorrecteurOrthographique *unCorrecteur, EnsembleDeMot desCorrections
    )
{
    EnsembleDeMot temp = unCorrecteur->lesCorrections;
    unCorrecteur->lesCorrections = EDM_union(desCorrections, temp);
    EDM_vider(&temp);
}

void CO_trouverCorrectionsPossibles(CorrecteurOrthographique *unCorrecteur)
{
    CO_strategieRemplacerLettres(unCorrecteur);
```

```
55      CO_strategieSupprimerLettres(unCorrecteur);
56      CO_strategieInsererLettres(unCorrecteur);
57      CO_strategieInverserDeuxLettresConsecutives(unCorrecteur);
58      CO_strategieDecomposerMot(unCorrecteur);
59  }
60
61  EnsembleDeMot CO_remplacerIemeLettreEnBoucle(Mot motACorriger, int i)
62  {
63      EnsembleDeMot desCorrections;
64      Mot uneCorrection;
65      desCorrections = ensembleDeMot();
66      char *lettres;
67      lettres = "abcdefghijklmnopqrstuvwxyzàéèëùûêîïçôö–";
68      for (int j = 0; j < strlen(lettres); j++)
69      {
70          uneCorrection = M_copierMot(motACorriger);
71          M_fixerIemeCaractere(&uneCorrection, i, lettres[j]);
72          EDM_ajouter(&desCorrections, uneCorrection);
73      }
74      return desCorrections;
75  }
76
77  void CO_strategieRemplacerLettres(CorrecteurOrthographique *unCorrecteur)
78  {
79      unsigned int i, longueur;
80      Mot uneCorrection;
81
82      Mot leMotACorriger = CO_obtenirMotACorriger(*unCorrecteur);
83      Dictionnaire leDico = CO_obtenirDictionnaire(*unCorrecteur);
84      longueur = M_longueurMot(leMotACorriger);
85      EnsembleDeMot desCorrections = ensembleDeMot();
86      EnsembleDeMot correctionsCourantes = ensembleDeMot();
87      for (i = 1; i < longueur + 1; i++)
88      {
89          correctionsCourantes = CO_remplacerIemeLettreEnBoucle(leMotACorriger, i);
90
91          while (EDM_cardinalite(correctionsCourantes) != 0)
92          {
93              uneCorrection = EDM_obtenirMot(correctionsCourantes);
94              EDM_retirer(&correctionsCourantes, uneCorrection);
95              if (D_estUnMotDuDictionnaire(leDico, M_copierMot(uneCorrection)) && !EDM_estPresent(
    desCorrections, uneCorrection))
96                  EDM_ajouter(&desCorrections, uneCorrection);
97              else
98                  M_supprimerMot(&uneCorrection);
99          }
100         CO_ajouterNouvellesCorrections(unCorrecteur, desCorrections);
101
102         EDM_vider(&correctionsCourantes);
103         EDM_vider(&desCorrections);
104     }
105 }
106
107 void CO_strategieSupprimerLettres(CorrecteurOrthographique *unCorrecteur)
```

```
108 {
109     unsigned int i, longueur;
110     Mot uneCorrection;
111 
112     Mot leMotACorriger = CO_obtenirMotACorriger(*unCorrecteur);
113     Dictionnaire leDico = CO_obtenirDictionnaire(*unCorrecteur);
114     longueur = M_longueurMot(leMotACorriger);
115     EnsembleDeMot desCorrections = ensembleDeMot();
116 
117     for (i = 1; i <= longueur; i++)
118     {
119         uneCorrection = M_copierMot(leMotACorriger);
120         M_supprimerIemeLettre(&uneCorrection, i);
121         if (D_estUnMotDuDictionnaire(leDico, M_copierMot(uneCorrection)) && !EDM_estPresent(
    desCorrections, uneCorrection))
122             EDM_ajouter(&desCorrections, uneCorrection);
123         else
124             M_supprimerMot(&uneCorrection);
125     }
126     CO_ajouterNouvellesCorrections(unCorrecteur, desCorrections);
127     EDM_vider(&desCorrections);
128 }
129 
130 void CO_strategieInverserDeuxLettresConsecutives(CorrecteurOrthographique *unCorrecteur)
131 {
132     unsigned int i, longueur;
133     Mot uneCorrection;
134 
135     Mot leMotACorriger = CO_obtenirMotACorriger(*unCorrecteur);
136     Dictionnaire leDico = CO_obtenirDictionnaire(*unCorrecteur);
137     longueur = M_longueurMot(leMotACorriger);
138     EnsembleDeMot desCorrections = ensembleDeMot();
139 
140     for (i = 1; i < longueur; i++)
141     {
142         uneCorrection = M_copierMot(leMotACorriger);
143         M_inverserDeuxLettresConsecutives(&uneCorrection, i);
144         if (D_estUnMotDuDictionnaire(leDico, M_copierMot(uneCorrection)) && !EDM_estPresent(
    desCorrections, uneCorrection))
145             EDM_ajouter(&desCorrections, uneCorrection);
146         else
147         {
148             M_supprimerMot(&uneCorrection);
149         }
150     }
151     CO_ajouterNouvellesCorrections(unCorrecteur, desCorrections);
152     EDM_vider(&desCorrections);
153 }
154 
155 EnsembleDeMot CO_insererIemeLettreEnBoucle(Mot motACorriger, int i)
156 {
157     EnsembleDeMot desCorrections;
158     Mot uneCorrection;
159     desCorrections = ensembleDeMot();
```

```
160     char *lettres;
161     lettres = "abcdefghijklmnopqrstuvwxyzàéèëùûêîïçôö–";
162     for (int j = 0; j < strlen(lettres); j++)
163     {
164         uneCorrection = M_copierMot(motACorriger);
165         M_insererLettre(&uneCorrection, i, lettres[j]);
166         EDM_ajouter(&desCorrections, uneCorrection);
167     }
168     return desCorrections;
169 }
170
171 void CO_strategieInsererLettres(CorrecteurOrthographique *unCorrecteur)
172 {
173     unsigned int i, longueur;
174     Mot uneCorrection;
175
176     Mot leMotACorriger = CO_obtenirMotACorriger(*unCorrecteur);
177     Dictionnaire leDico = CO_obtenirDictionnaire(*unCorrecteur);
178     longueur = M_longueurMot(leMotACorriger);
179     EnsembleDeMot desCorrections = ensembleDeMot();
180     EnsembleDeMot correctionsCourantes = ensembleDeMot();
181     for (i = 1; i <= longueur + 1; i++)
182     {
183         correctionsCourantes = CO_insererIemeLettreEnBoucle(leMotACorriger, i);
184
185         while (EDM_cardinalite(correctionsCourantes) != 0)
186         {
187             uneCorrection = EDM_obtenirMot(correctionsCourantes);
188             EDM_retirer(&correctionsCourantes, uneCorrection);
189             if (D_estUnMotDuDictionnaire(leDico, M_copierMot(uneCorrection)) && !EDM_estPresent(
    desCorrections, uneCorrection))
190                 EDM_ajouter(&desCorrections, uneCorrection);
191             else
192                 M_supprimerMot(&uneCorrection);
193         }
194         CO_ajouterNouvellesCorrections(unCorrecteur, desCorrections);
195
196         EDM_vider(&correctionsCourantes);
197         EDM_vider(&desCorrections);
198     }
199 }
200
201 void CO_strategieDecomposerMot(CorrecteurOrthographique *unCorrecteur)
202 {
203     unsigned int i, longueur;
204     Mot leMotACorriger = CO_obtenirMotACorriger(*unCorrecteur);
205     Mot uneCorrection;
206     Mot unMotModifiable;
207     Dictionnaire leDico = CO_obtenirDictionnaire(*unCorrecteur);
208     longueur = M_longueurMot(leMotACorriger);
209     EnsembleDeMot desCorrections = ensembleDeMot();
210
211     for (i = 2; i < longueur + 1; i++)
212     {
```

```
213             unMotModifiable = M_copierMot(leMotACorriger);
214             uneCorrection = M_decomposerMot(&unMotModifiable, i);
215             if (D_estUnMotDuDictionnaire(leDico, M_copierMot(uneCorrection)) && D_estUnMotDuDictionnaire(
        leDico, M_copierMot(unMotModifiable)) && !EDM_estPresent(desCorrections, unMotModifiable) && !
        EDM_estPresent(desCorrections, uneCorrection))
216             {
217                 EDM_ajouter(&desCorrections, uneCorrection);
218                 EDM_ajouter(&desCorrections, unMotModifiable);
219             }
220             else
221             {
222                 M_supprimerMot(&uneCorrection);
223                 M_supprimerMot(&unMotModifiable);
224             }
225         }
226         CO_ajouterNouvellesCorrections(unCorrecteur, desCorrections);
227         EDM_vider(&desCorrections);
228     }
229
230     void CO_supprimerCorrecteur(CorrecteurOrthographique *unCorrecteur)
231     {
232         Mot unMot;
233         M_supprimerMot(&unCorrecteur->motACorriger);
234
235         while (EDM_cardinalite(unCorrecteur->lesCorrections) != 0)
236         {
237             unMot = EDM_obtenirMot(unCorrecteur->lesCorrections);
238             EDM_retirer(&unCorrecteur->lesCorrections, unMot);
239             M_supprimerMot(&unMot);
240         }
241
242         ADL_supprimer(&unCorrecteur->leDictionnaire);
243     }
```

../programme/src/CorrecteurOrthographique.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <assert.h>
4   #include <string.h>
5   #include "Dictionnaire.h"
6   #include "CorrecteurOrthographique.h"
7   #include "Mot.h"
8   #include "EnsembleDeMot.h"
9   #include "corrigerTexte.h"
10
11  void CT_corrigerTexte(char *chaine, Dictionnaire dico)
12  {
13      int indiceAvantMot = 0;
14      for (int position = 0; position < strlen(chaine); position++)
15      {
16          if (CT_estUnSeparateur(chaine[position]))
17          {
18              if (position > indiceAvantMot)
```

```
19              {
20                  CT_trouverEtAfficherCorrection(chaine, indiceAvantMot, position, dico);
21              }
22              indiceAvantMot = position + 1;
23          }
24          else
25          {
26          }
27      }
28      if (strlen(chaine) > indiceAvantMot)
29      {
30          CT_trouverEtAfficherCorrection(chaine, indiceAvantMot, strlen(chaine), dico);
31      }
32      printf("\n");
33  }
34
35  int CT_estUnSeparateur(char c)
36  {
37      char *apostrophe = "'";
38      return (c == ' ') || (c == ',') || (c == '.') || (c == '?') || (c == ';') || (c == '!') || (c == ':'
        ) || (c == '\0') || (c == apostrophe[0]) || (c == '\n');
39  }
40
41  char *CT_creerSousChaine(char *chaine, unsigned int gauche, unsigned int droite)
42  {
43      assert((gauche <= droite) && (droite < strlen(chaine)));
44      char *sousChaine = malloc((droite - gauche + 2) * sizeof(char));
45      memcpy(sousChaine, &chaine[gauche], droite - gauche + 1);
46      sousChaine[droite - gauche + 1] = '\0';
47      return sousChaine;
48  }
49
50  CorrecteurOrthographique CT_trouverCorrections(Dictionnaire dico, Mot unMot)
51  {
52      CorrecteurOrthographique correcteur = CO_correcteur(dico, unMot);
53      CO_trouverCorrectionsPossibles(&correcteur);
54      return correcteur;
55  }
56
57  void CT_afficherCorrection(int indiceDebutMot, CorrecteurOrthographique correcteur)
58  {
59      printf("& %s %i %ld :", CO_obtenirMotACorriger(correcteur).chaine, indiceDebutMot, EDM_cardinalite(
        CO_obtenirCorrections(correcteur)));
60      if (EDM_cardinalite(CO_obtenirCorrections(correcteur)) == 0)
61      {
62          printf(" Trop d'erreurs dans ce mot\n");
63      }
64      else
65      {
66          int cardinalite = EDM_cardinalite(CO_obtenirCorrections(correcteur));
67          for (int i = 0; i < cardinalite; i++)
68          {
69              Mot tmp = EDM_obtenirMot(CO_obtenirCorrections(correcteur));
70              printf(" %s", M_obtenirChaine(tmp));
```

```
71              EDM_retirer(&correcteur.lesCorrections, tmp);
72              M_supprimerMot(&tmp);
73          }
74          printf("\n");
75      }
76  }
77
78  void CT_trouverEtAfficherCorrection(char *chaine, int indiceDebutMot, int position, Dictionnaire dico)
79  {
80      char *sousChaine = CT_creerSousChaine(chaine, indiceDebutMot, position - 1);
81      Mot unMot = M_creerUnMot(sousChaine);
82
83      if (!D_estUnMotDuDictionnaire(dico, M_copierMot(unMot)))
84      {
85          CorrecteurOrthographique correcteur = CT_trouverCorrections(dico, unMot);
86          CT_afficherCorrection(indiceDebutMot, correcteur);
87          M_supprimerMot(&correcteur.motACorriger);
88      }
89      else
90      {
91          M_supprimerMot(&unMot);
92          printf("*\n");
93      }
94      free(sousChaine);
95      M_supprimerMot(&unMot);
96  }
```

../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  {
16      Dictionnaire temp;
17      int enFinDeMot = 0;
18      if (M_longueurMot(unMot) == 1)
19      {
20          enFinDeMot = 1;
21          if (ADL_estVide(*unDico))
22          {
23              D_insererLettre(unDico, M_iemeCaractere(unMot, 1), enFinDeMot);
24          }
25          else
```

```
26          {
27              if (D_lettreEstRacine(*unDico, M_iemeCaractere(unMot, 1)))
28              {
29                  ADL_fixerEstFinDeMot(unDico, enFinDeMot);
30              }
31              else
32              {
33                  temp = ADL_obtenirFrere(*unDico);
34                  D_inserer Mot(&temp, unMot);
35                  ADL_fixerFrere(unDico, temp);
36              }
37          }
38      }
39      else
40      {
41          if (ADL_estVide(*unDico))
42          {
43              D_insererLettre(unDico, M_iemeCaractere(unMot, 1), enFinDeMot);
44              M_supprimerIemeLettre(&unMot, 1);
45              temp = ADL_obtenirFils(*unDico);
46              D_inserer Mot(&temp, unMot);
47              ADL_fixerFils(unDico, temp);
48          }
49          else
50          {
51              if (D_lettreEstRacine(*unDico, M_iemeCaractere(unMot, 1)))
52              {
53                  M_supprimerIemeLettre(&unMot, 1);
54                  temp = ADL_obtenirFils(*unDico);
55                  D_inserer Mot(&temp, unMot);
56                  ADL_fixerFils(unDico, temp);
57              }
58              else
59              {
60                  temp = ADL_obtenirFrere(*unDico);
61                  D_inserer Mot(&temp, unMot);
62                  ADL_fixerFrere(unDico, temp);
63              }
64          }
65      }
66  }
67
68  void D_insererLettre(Dictionnaire *unDico, char uneLettre, int estFinDeMot)
69  {
70      assert(ADL_estVide(*unDico));
71      *unDico = ADL_creerADL(NULL, NULL, uneLettre, estFinDeMot);
72  }
73
74  int D_lettreEstRacine(Dictionnaire unDico, char uneLettre)
75  {
76      return ADL_obtenirLettre(unDico) == uneLettre;
77  }
78
79  Mot *supprimerLesMots(Mot *lesMots, int nbMots)
```

```
80  {
81      Mot *lesMotsASupprimer = lesMots;
82
83      int i = 0;
84      while (i < nbMots)
85      {
86          M_supprimerMot(&lesMotsASupprimer[i]);
87          i++;
88      }
89
90      return lesMotsASupprimer;
91  }
92
93  void supprimerTabMots(Mot **lesMots, int nbMots)
94  {
95      *lesMots = supprimerLesMots(*lesMots, nbMots);
96      free(*lesMots);
97  }
98
99  Mot *D_genererTableauDeMotAvecFichierTexte(FichierTexte ficDico, int *nbMots)
100 {
101     Mot *lesMots = (Mot *)malloc(((sizeof(char) * 27) + sizeof(int)) * NB_MOTS_DICTIONNAIRE);
102
103     FT_ouvrir(&ficDico, LECTURE);
104
105     char *chaine;
106
107     int tailleTab = 0;
108     while (!FT_estEnFinDeFichier(ficDico))
109     {
110         chaine = FT_lireChaineSansLeRetourChariot(ficDico);
111
112         if (strlen(chaine) > 0)
113         {
114             lesMots[tailleTab] = M_creerUnMot(chaine);
115             free(chaine);
116
117             tailleTab++;
118         }
119     }
120     FT_fermer(&ficDico);
121     *nbMots = tailleTab;
122     return lesMots;
123 }
124
125 Dictionnaire D_genererDicoAvecTableauDeMots(Mot *lesMots, int nbMots)
126 {
127     Dictionnaire unDico = ADL_creerADLVide();
128     int i;
129     Mot unMot;
130     for (i = 0; i < nbMots; i++)
131     {
132         unMot = M_copierMot(lesMots[i]);
133         D_insererMot(&unDico, unMot);
```

```
134             M_supprimerMot(&unMot);
135         }
136         return unDico;
137     }
138
139     int charEnInt(char c)
140     {
141         return c - '0';
142     }
143
144     void D_chargerDicoR(Dictionnaire *unDico, FichierTexte sauvegardeDico)
145     {
146         Dictionnaire temp;
147         char lettre, estFinDeMot, aUnFils, aUnFrere;
148         char *element = FT_lireElement(sauvegardeDico);
149         lettre = element[0];
150         estFinDeMot = element[1];
151         aUnFils = element[2];
152         aUnFrere = element[3];
153         *unDico = ADL_creerADL(NULL, NULL, lettre, charEnInt(estFinDeMot));
154         if (charEnInt(aUnFils) == 1)
155         {
156             D_chargerDicoR(&temp, sauvegardeDico);
157             ADL_fixerFils(unDico, temp);
158         }
159         if (charEnInt(aUnFrere) == 1)
160         {
161             D_chargerDicoR(&temp, sauvegardeDico);
162             ADL_fixerFrere(unDico, temp);
163         }
164
165         free(element);
166     }
167
168     void D_sauvegarderDicoR(Dictionnaire *unDico, FichierTexte fic)
169     {
170
171         Dictionnaire tempFils, tempFrere;
172         if (!ADL_estVide(*unDico))
173         {
174             FT_ecrireCaractere(&fic, ADL_obtenirLettre(*unDico));
175             if (ADL_obtenirEstFinDeMot(*unDico))
176                 FT_ecrireCaractere(&fic, '1');
177             else
178                 FT_ecrireCaractere(&fic, '0');
179
180             tempFils = ADL_obtenirFils(*unDico);
181             tempFrere = ADL_obtenirFrere(*unDico);
182             if (!ADL_estVide(tempFils))
183             {
184                 FT_ecrireCaractere(&fic, '1');
185             }
186             else
187                 FT_ecrireCaractere(&fic, '0');
```

```
188             if (!ADL_estVide(tempFrere))
189             {
190                 FT_ecrireCaractere(&fic, '1');
191             }
192             else
193                 FT_ecrireCaractere(&fic, '0');
194 
195             D_sauvegarderDicoR(&tempFils, fic);
196             D_sauvegarderDicoR(&tempFrere, fic);
197         }
198     }
199 
200     /* Partie publique */
201 
202     Dictionnaire D_genererDicoAvecFichierTexte(FichierTexte ficDico)
203     {
204         int nbMots;
205         Mot *lesMots = D_genererTableauDeMotAvecFichierTexte(ficDico, &nbMots);
206         Dictionnaire leDico = D_genererDicoAvecTableauDeMots(lesMots, nbMots);
207         supprimerTabMots(&lesMots, nbMots);
208         return leDico;
209     }
210 
211     int D_estUnMotDuDictionnaire(Dictionnaire unDico, Mot unMot)
212     {
213         Dictionnaire temp;
214         if (M_longueurMot(unMot) == 1)
215         {
216             if (!ADL_estVide(unDico))
217             {
218                 if (M_iemeCaractere(unMot, 1) == ADL_obtenirLettre(unDico))
219                 {
220                     M_supprimerMot(&unMot);
221                     return ADL_obtenirEstFinDeMot(unDico);
222                 }
223                 else
224                 {
225                     temp = ADL_obtenirFrere(unDico);
226                     return D_estUnMotDuDictionnaire(temp, unMot);
227                 }
228             }
229             else
230             {
231                 M_supprimerMot(&unMot);
232                 return 0;
233             }
234         }
235         else
236         {
237             if (!ADL_estVide(unDico))
238             {
239                 if (M_iemeCaractere(unMot, 1) == ADL_obtenirLettre(unDico))
240                 {
241                     M_supprimerIemeLettre(&unMot, 1);
```

```
242                     temp = ADL_obtenirFils(unDico);
243                     return D_estUnMotDuDictionnaire(temp, unMot);
244                 }
245                 else
246                 {
247                     temp = ADL_obtenirFrere(unDico);
248                     return D_estUnMotDuDictionnaire(temp, unMot);
249                 }
250             }
251             else
252             {
253                 M_supprimerMot(&unMot);
254                 return 0;
255             }
256         }
257     }
258
259     Dictionnaire D_chargerDico(FichierTexte sauvegardeDico)
260     {
261         Dictionnaire unDico;
262         FT_ouvrir(&sauvegardeDico, LECTURE);
263         if (FT_verifierIdDico(sauvegardeDico))
264             D_chargerDicoR(&unDico, sauvegardeDico);
265         else
266             printf("Le fichier que vous essayez d'utiliser n'est pas compatible");
267         FT_fermer(&sauvegardeDico);
268         return unDico;
269     }
270
271     void D_sauvegarderDico(Dictionnaire *unDico, FichierTexte *sauvegardeDico)
272     {
273         FT_ouvrir(sauvegardeDico, ECRITURE);
274         FT_ecrireID(sauvegardeDico);
275         D_sauvegarderDicoR(unDico, *sauvegardeDico);
276         FT_fermer(sauvegardeDico);
277     }
```

../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"
7   #include "ListeChaineeDeMot.h"
8   #include "EnsembleDeMot.h"
9
10  /* Partie privee */
11  void EDM_ajouterElements(EnsembleDeMot motsAAjouter, EnsembleDeMot *edmACompleter)
12  {
13      errno = 0;
14      ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
15      l = motsAAjouter.lesMots;
```

```
16      while (!LCDM_estVide(l))
17      {
18          EDM_ajouter(edmACompleter, LCDM_obtenirMot(l));
19          l = LCDM_obtenirListeSuivante(l);
20      }
21  }
22
23  /* Partie publique */
24  EnsembleDeMot ensembleDeMot()
25  {
26      errno = 0;
27      EnsembleDeMot unEDM;
28      unEDM.lesMots = LCDM_listeChaineeDeMot();
29      unEDM.nbMots = 0;
30      return unEDM;
31  }
32
33  void EDM_vider(EnsembleDeMot *unEDM)
34  {
35      errno = 0;
36      unEDM->nbMots = 0;
37      LCDM_supprimer(&unEDM->lesMots);
38      *unEDM = ensembleDeMot();
39  }
40
41  EnsembleDeMot EDM_copier(EnsembleDeMot unEDM)
42  {
43      EnsembleDeMot copieEDM = ensembleDeMot();
44      copieEDM.nbMots = unEDM.nbMots;
45      copieEDM.lesMots = LCDM_copier(unEDM.lesMots);
46      return copieEDM;
47  }
48
49  int EDM_egale(EnsembleDeMot edm_1, EnsembleDeMot edm_2)
50  {
51      errno = 0;
52      int sontEgales = 0;
53      if (EDM_cardinalite(edm_1) == EDM_cardinalite(edm_2))
54      {
55          sontEgales = LCDM_egale(edm_1.lesMots, edm_2.lesMots);
56      }
57      return sontEgales;
58  }
59
60  void EDM_ajouter(EnsembleDeMot *unEDM, Mot unMot)
61  {
62      if (!EDM_estPresent(*unEDM, unMot))
63      {
64          LCDM_ajouter(&unEDM->lesMots, unMot);
65          unEDM->nbMots = unEDM->nbMots + 1;
66      }
67      else
68      {
69          errno = LCDM_ERREUR_MEMOIRE;
```

```
70      }
71  }
72
73  void EDM_retirer(EnsembleDeMot *unEDM, Mot unMot)
74  {
75      errno = 0;
76      if (EDM_estPresent(*unEDM, unMot))
77      {
78          LCDM_supprimerMot(&unEDM->lesMots, unMot);
79          unEDM->nbMots--;
80      }
81  }
82
83  int EDM_estPresentDansListe(ListeChaineeDeMot l, Mot unMot)
84  {
85      errno = 0;
86      if (LCDM_estVide(l))
87      {
88          return 0;
89      }
90      else
91      {
92          if (M_sontIdentiques(LCDM_obtenirMot(l), unMot))
93          {
94              return 1;
95          }
96          else
97          {
98              return EDM_estPresentDansListe(LCDM_obtenirListeSuivante(l), unMot);
99          }
100     }
101 }
102
103 int EDM_estPresent(EnsembleDeMot unEDM, Mot unMot)
104 {
105     return EDM_estPresentDansListe(unEDM.lesMots, unMot);
106 }
107
108 long int EDM_cardinalite(EnsembleDeMot unEDM)
109 {
110     errno = 0;
111     return unEDM.nbMots;
112 }
113
114 EnsembleDeMot EDM_union(EnsembleDeMot edm_1, EnsembleDeMot edm_2)
115 {
116     EnsembleDeMot unionEDM = ensembleDeMot();
117     EDM_ajouterElements(edm_1, &unionEDM);
118     EDM_ajouterElements(edm_2, &unionEDM);
119     return unionEDM;
120 }
121
122 Mot EDM_obtenirMot(EnsembleDeMot unEDM)
123 {
```

```
124     errno = 0;
125     Mot leMot;
126     ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
127     l = unEDM.lesMots;
128     leMot = LCDM_obtenirMot(l);
129     //free(l);
130     return leMot;
131 }
```

../programme/src/EnsembleDeMot.c

```
1  #include <stdlib.h>
2  #include <assert.h>
3  #include <string.h>
4  #include "FichierTexte.h"
5  #define LONGUEUR_MAX_CHAINE 100
6  #define ID_DICO "DICO GROUPE DD"
7
8  FichierTexte FT_fichierTexte(char *nomDuFichier)
9  {
10     FichierTexte unFichier;
11     unFichier.fichier = NULL;
12     unFichier.nom = nomDuFichier;
13     return unFichier;
14 }
15
16 void FT_ouvrir(FichierTexte *unFichier, Mode mode)
17 {
18     assert(!FT_estOuvert(*unFichier));
19
20     if (mode == ECRITURE)
21     {
22         unFichier->fichier = fopen(unFichier->nom, "w+");
23         unFichier->mode = mode;
24     }
25     else if (mode == LECTURE)
26     {
27         unFichier->fichier = fopen(unFichier->nom, "r");
28         unFichier->mode = mode;
29     }
30     else
31         printf("ERREUR : Le mode choisi n'est pas le bon\n");
32 }
33
34 void FT_fermer(FichierTexte *unFichier)
35 {
36     if (FT_estOuvert(*unFichier))
37     {
38         fclose(unFichier->fichier);
39         unFichier->fichier = NULL;
40     }
41 }
42
43 unsigned int FT_estOuvert(FichierTexte unFichier)
```

```
44  {
45      return unFichier.fichier != NULL;
46  }
47
48  Mode FT_obtenirMode(FichierTexte unFichier)
49  {
50      return unFichier.mode;
51  }
52
53  unsigned int FT_estEnFinDeFichier(FichierTexte unFichier)
54  {
55      assert((unFichier.mode == LECTURE) && FT_estOuvert(unFichier));
56      return feof(unFichier.fichier);
57  }
58
59  char *FT_lireChaine(FichierTexte unFichier)
60  {
61      assert(FT_estOuvert(unFichier) && (FT_obtenirMode(unFichier) == LECTURE) && !FT_estEnFinDeFichier(
        unFichier));
62
63      char *buffer = (char *)malloc(sizeof(char) * LONGUEUR_MAX_CHAINE);
64      if (fgets(buffer, LONGUEUR_MAX_CHAINE, unFichier.fichier))
65      {
66          return buffer;
67      }
68      else
69      {
70          free(buffer);
71          return NULL;
72      }
73  }
74
75  void FT_afficherContenuFichier(FichierTexte unFichier)
76  {
77      FT_ouvrir(&unFichier, LECTURE);
78      char *chaine;
79
80      while (!FT_estEnFinDeFichier(unFichier))
81      {
82          chaine = FT_lireChaineSansLeRetourChariot(unFichier);
83          if (strlen(chaine) > 0)
84          {
85              puts(chaine);
86              free(chaine);
87          }
88      }
89
90      FT_fermer(&unFichier);
91  }
92
93  void supprimerRetourChariot(char *chaine)
94  {
95      int i = 0;
96      while (chaine[i] != '\0')
```

```
97      {
98          if (chaine[i] == '\n')
99          {
100             chaine[i] = '\0';
101         }
102         i++;
103     }
104 }
105
106 char *FT_lireChaineSansLeRetourChariot(FichierTexte unFichier)
107 {
108     assert(FT_estOuvert(unFichier) && (FT_obtenirMode(unFichier) == LECTURE) && !FT_estEnFinDeFichier(
        unFichier));
109     char *ligne;
110     ligne = FT_lireChaine(unFichier);
111     if (ligne != NULL)
112     {
113         supprimerRetourChariot(ligne);
114         return ligne;
115     }
116     else
117     {
118         free(ligne);
119         return "";
120     }
121 }
122
123 void FT_ecrireCaractere(FichierTexte *unFichier, char lettre)
124 {
125     assert(FT_estOuvert(*unFichier) && (FT_obtenirMode(*unFichier) == ECRITURE));
126     fputc(lettre, unFichier->fichier);
127 }
128
129 char FT_lireCaractere(FichierTexte unFichier)
130 {
131     assert(FT_estOuvert(unFichier) && (FT_obtenirMode(unFichier) == LECTURE) && !FT_estEnFinDeFichier(
        unFichier));
132     return fgetc(unFichier.fichier);
133 }
134
135 char *FT_lireElement(FichierTexte unFichier)
136 {
137     char *element = malloc(4);
138     element[0] = FT_lireCaractere(unFichier);
139     element[1] = FT_lireCaractere(unFichier);
140     element[2] = FT_lireCaractere(unFichier);
141     element[3] = FT_lireCaractere(unFichier);
142     return element;
143 }
144
145 int FT_verifierIdDico(FichierTexte unFichier)
146 {
147     assert(FT_estOuvert(unFichier) && (FT_obtenirMode(unFichier) == LECTURE));
148     char *id = FT_lireChaineSansLeRetourChariot(unFichier);
```

```
149     int idCorrect = strcmp(id, ID_DICO);
150     free(id);
151     return (idCorrect == 0);
152 }
153 void FT_ecrireChaine(FichierTexte *unFichier, char *chaine)
154 {
155     assert(FT_estOuvert(*unFichier) && (FT_obtenirMode(*unFichier) == ECRITURE));
156     fputs(chaine, unFichier->fichier);
157 }
158
159 void FT_ecrireID(FichierTexte *unFichier)
160 {
161     FT_ecrireChaine(unFichier, ID_DICO);
162     FT_ecrireCaractere(unFichier, '\n');
163 }
```

../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  {
9      errno = 0;
10     return NULL;
11 }
12
13 int LCDM_estVide(ListeChaineeDeMot l)
14 {
15     errno = 0;
16     return (l == NULL);
17 }
18
19 void LCDM_ajouter(ListeChaineeDeMot *l, Mot unMot)
20 {
21     ListeChaineeDeMot pNoeud = malloc(sizeof(Noeud));
22     if (pNoeud != NULL)
23     {
24         errno = 0;
25         pNoeud->mot = unMot;
26         pNoeud->listeSuivante = *l;
27         *l = pNoeud;
28     }
29     else
30     {
31         errno = LCDM_ERREUR_MEMOIRE;
32     }
33 }
34
35 Mot LCDM_obtenirMot(ListeChaineeDeMot l)
36 {
```

```
37      assert(!LCDM_estVide(l));
38      errno = 0;
39      return l->mot;
40  }
41
42  void LCDM_supprimerMot(ListeChaineeDeMot *l, Mot unMot)
43  {
44      assert(!LCDM_estVide(*l));
45      ListeChaineeDeMot temp = LCDM_listeChaineeDeMot();
46      if (!LCDM_estVide(*l))
47      {
48          if (M_sontIdentiques(LCDM_obtenirMot(*l), unMot))
49          {
50              LCDM_supprimerTete(l);
51          }
52          else
53          {
54              temp = LCDM_obtenirListeSuivante(*l);
55              LCDM_supprimerMot(&temp, unMot);
56              LCDM_fixerListeSuivante(l, temp);
57          }
58      }
59  }
60
61  ListeChaineeDeMot LCDM_obtenirListeSuivante(ListeChaineeDeMot l)
62  {
63      assert(!LCDM_estVide(l));
64      errno = 0;
65      return l->listeSuivante;
66  }
67
68  void LCDM_fixerListeSuivante(ListeChaineeDeMot *l, ListeChaineeDeMot suivant)
69  {
70      assert(!LCDM_estVide(*l));
71      errno = 0;
72      (*l)->listeSuivante = suivant;
73  }
74
75  void LCDM_fixerMot(ListeChaineeDeMot *l, Mot unMot)
76  {
77      assert(!LCDM_estVide(*l));
78      errno = 0;
79      (*l)->mot = unMot;
80  }
81
82  void LCDM_supprimerTete(ListeChaineeDeMot *l)
83  {
84      ListeChaineeDeMot temp;
85      assert(!LCDM_estVide(*l));
86      errno = 0;
87      temp = *l;
88      *l = LCDM_obtenirListeSuivante(*l);
89      free(temp);
90  }
```

```
91
92  void LCDM_supprimer(ListeChaineeDeMot *l)
93  {
94      errno = 0;
95      if (!LCDM_estVide(*l))
96      {
97          LCDM_supprimerTete(l);
98          LCDM_supprimer(l);
99      }
100 }
101
102 ListeChaineeDeMot LCDM_copier(ListeChaineeDeMot l)
103 {
104     ListeChaineeDeMot temp;
105     errno = 0;
106     if (LCDM_estVide(l))
107     {
108         return LCDM_listeChaineeDeMot();
109         free(temp);
110     }
111     else
112     {
113         temp = LCDM_copier(LCDM_obtenirListeSuivante(l));
114         LCDM_ajouter(&temp, LCDM_obtenirMot(l));
115         return temp;
116     }
117 }
118
119 int LCDM_egale(ListeChaineeDeMot l1, ListeChaineeDeMot l2)
120 {
121     errno = 0;
122     if (LCDM_estVide(l1) && LCDM_estVide(l2))
123     {
124         return 1;
125     }
126     else
127     {
128         if (LCDM_estVide(l1) || LCDM_estVide(l2))
129         {
130             return 0;
131         }
132         else
133         {
134             if (M_sontIdentiques(LCDM_obtenirMot(l1), LCDM_obtenirMot(l2)))
135             {
136                 return LCDM_egale(LCDM_obtenirListeSuivante(l1), LCDM_obtenirListeSuivante(l2));
137             }
138             else
139             {
140                 return 0;
141             }
142         }
143     }
144 }
```

../programme/src/ListeChaineeDeMot.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "IHM.h"

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

    if (argc <= 2 || argc > 5)
    {
        if (argc == 2)
            gererCommandeAide(argv[1]);
        else
            gererCommandeAide("");
    }

    else if (argc == 3)
    {
        gererCommandeCorrection(argv[1], argv[2]);
    }

    else if (argc == 5)
    {
        gererCommandeCreerDico(argv[1], argv[2], argv[3], argv[4]);
    }

    return EXIT_SUCCESS;
}
```

../programme/src/main.c

```c
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include "Mot.h"
#include <string.h>
#include <ctype.h>

int M_estUnCaractereAlphabetique(char c)
{
    char tmp = c;
    return (isalpha(tmp) || M_estUnCaractereAAccent(c));
}

int M_estUnCaractereAAccent(char c)
{
    return (c == '-' || c == 'ù' || c == 'à' || c == 'é' || c == 'è' || c == 'ç' || c == 'ï' || c == 'É'
     || c == 'É' || c == 'Ç' || c == 'À' || c == 'â' || c == 'Â' || c == 'ë' || c == 'Ë' || c == 'ô' ||
    c == 'Ô' || c == 'î' || c == 'ê' || c == 'û' || c == 'ü' || c == 'ä' || c == 'ö');
}
```

```
18
19  int M_estUnMotValide(char *c)
20  {
21      assert(strlen(c) > 0);
22      int longueurChaine = strlen(c);
23      int estValide = 1;
24      int i = 0;
25      while (estValide && i < longueurChaine - 1)
26      {
27          if (!M_estUnCaractereAlphabetique(c[i]))
28          {
29              estValide = 0;
30          }
31          i = i + 1;
32      }
33      return estValide;
34  }
35
36  Mot M_copierMot(Mot unMot)
37  {
38      Mot copie;
39      copie = M_creerUnMot(unMot.chaine);
40      return copie;
41  }
42
43  Mot M_creerUnMot(char *c)
44  {
45      assert(M_estUnMotValide(c));
46      Mot unMot;
47      unMot.chaine = M_reduireLaCasseDUneChaine(c);
48      unMot.longueur = strlen(unMot.chaine);
49      return unMot;
50  }
51
52  unsigned int M_longueurMot(Mot unMot)
53  {
54      return unMot.longueur;
55  }
56
57  char *M_obtenirChaine(Mot unMot)
58  {
59      return unMot.chaine;
60  }
61
62  char M_iemeCaractere(Mot unMot, unsigned int i)
63  {
64      assert(i <= M_longueurMot(unMot) && i > 0);
65      return unMot.chaine[i - 1];
66  }
67
68  int M_sontIdentiques(Mot mot1, Mot mot2)
69  {
70      return mot1.longueur == mot2.longueur && strcmp(mot1.chaine, mot2.chaine) == 0;
71  }
```

```
72
73  void M_fixerIemeCaractere(Mot *unMot, unsigned int i, char c)
74  {
75      assert(M_estUnCaractereAlphabetique(c) && i <= M_longueurMot(*unMot) + 1);
76      unMot->chaine[i - 1] = c;
77  }
78
79  void M_fixerLongueur(Mot *unMot, unsigned int i)
80  {
81      unMot->longueur = i;
82  }
83
84  void M_supprimerIemeLettre(Mot *unMot, unsigned int indiceLettreASupprimer)
85  {
86      assert(indiceLettreASupprimer <= M_longueurMot(*unMot));
87
88      int indiceLettreCourante = indiceLettreASupprimer;
89      while (indiceLettreCourante < M_longueurMot(*unMot))
90      {
91          M_fixerIemeCaractere(unMot, indiceLettreCourante, M_iemeCaractere(*unMot, indiceLettreCourante +
        1));
92          indiceLettreCourante++;
93      }
94      unMot->chaine[indiceLettreCourante - 1] = '\0';
95      M_fixerLongueur(unMot, M_longueurMot(*unMot) - 1);
96  }
97
98  void M_inverserDeuxLettresConsecutives(Mot *unMot, unsigned int i)
99  {
100     assert(i < M_longueurMot(*unMot) && i > 0);
101     char temp;
102     temp = M_iemeCaractere(*unMot, i);
103     M_fixerIemeCaractere(unMot, i, M_iemeCaractere(*unMot, i + 1));
104     M_fixerIemeCaractere(unMot, i + 1, temp);
105 }
106
107 void M_insererLettre(Mot *unMot, unsigned int i, char c)
108 {
109     assert(i <= M_longueurMot(*unMot) + 1);
110     unMot->chaine = realloc(unMot->chaine, sizeof(char) * M_longueurMot(*unMot) + 2);
111     for (int j = strlen(unMot->chaine); j > i - 1; j--)
112     {
113         M_fixerIemeCaractere(unMot, j + 1, M_iemeCaractere(*unMot, j));
114     }
115     M_fixerLongueur(unMot, M_longueurMot(*unMot) + 1);
116     unMot->chaine[M_longueurMot(*unMot)] = '\0';
117     unMot->chaine[i - 1] = c;
118 }
119
120 Mot M_decomposerMot(Mot *unMot, unsigned int i)
121 { //le ieme caractere est dans la deuxième partie du mot
122     assert(i <= M_longueurMot(*unMot) && i > 1);
123     char *chaineGauche = (char *)malloc(i + 1);
124     int j;
```

```
125     for (j = 0; j < i - 1; j++)
126     {
127         chaineGauche[j] = M_iemeCaractere(*unMot, 1);
128         M_supprimerIemeLettre(unMot, 1);
129     }
130     chaineGauche[j] = '\0';
131     Mot motGauche = M_creerUnMot(chaineGauche);
132     free(chaineGauche);
133     return motGauche;
134 }
135
136 char M_reduireLaCasseDUnCaractere(char car)
137 {
138     char c = car;
139     switch (c)
140     {
141
142     case 'À':
143         c = 'à';
144         break;
145
146     case 'Â':
147         c = 'â';
148         break;
149
150     case 'É':
151         c = 'é';
152         break;
153
154     case 'È':
155         c = 'è';
156         break;
157
158     case 'Ç':
159         c = 'ç';
160         break;
161
162     case 'Ê':
163         c = 'ê';
164         break;
165
166     case 'Ô':
167         c = 'ô';
168         break;
169
170     case 'Ï':
171         c = 'à';
172         break;
173
174     case 'Î':
175         c = 'à';
176         break;
177
178     case 'Û':
```

```
179             c = 'û';
180             break;
181
182         case 'Ù':
183             c = 'ù';
184             break;
185
186         default:
187             c = (tolower((unsigned char)c));
188         }
189         return c;
190     }
191
192     char *M_reduireLaCasseDUneChaine(char *chaine)
193     {
194         assert(strlen(chaine) > 0);
195         char *minuscule = malloc(strlen(chaine) + 2);
196         strcpy(minuscule, chaine);
197         for (int i = 0; i < strlen(chaine); i++)
198         {
199             minuscule[i] = M_reduireLaCasseDUnCaractere((char)chaine[i]);
200         }
201         return minuscule;
202     }
203
204     void M_supprimerMot(Mot *unMot)
205     {
206         if (M_longueurMot(*unMot) > 0)
207             free(unMot->chaine);
208         unMot->longueur = 0;
209     }
```

../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);
}

```

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

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

```
        test_estFinDeMot))

    )
    {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Lancement des tests */
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    printf("\n");
    CU_basic_show_failures(CU_get_failure_list());
    printf("\n\n");

    /* Nettoyage du registre */
    CU_cleanup_registry();
    return CU_get_error();
}
```

../programme/src/testArbreDeLettres.c

```
#include <stdio.h>
#include <CUnit/Basic.h>
#include <string.h>
#include <stdbool.h>
#include "Mot.h"
#include "CorrecteurOrthographique.h"
#include "EnsembleDeMot.h"
#include "Dictionnaire.h"

int init_suite_success(void)
{
    return 0;
}

int clean_suite_success(void)
{
    return 0;
}

char **creer_tableau_mot()
{
    char **lesMots = (char **)malloc((sizeof(char) * 5) * 50);
    lesMots[0] = "bvec";
    lesMots[1] = "avec";
    lesMots[2] = "avac";
    lesMots[3] = "avem";
    lesMots[4] = "arride";
    lesMots[5] = "attaque";
    return lesMots;
}

Dictionnaire creer_dictionnaire()
```

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

```
87      return lesMots;
88  }
89
90  char **creer_ensemble_solution_inserer()
91  {
92      char **lesMots = (char **)malloc((sizeof(char) * 6) * 100);
93      lesMots[0] = "aveöc";
94      lesMots[1] = "aveac";
95      lesMots[2] = "avebc";
96      lesMots[3] = "avecc";
97      lesMots[4] = "avedc";
98      lesMots[5] = "aveec";
99      lesMots[6] = "avefc";
100     lesMots[7] = "avegc";
101     lesMots[8] = "avehc";
102     lesMots[9] = "aveic";
103     lesMots[10] = "avejc";
104     lesMots[11] = "avekc";
105     lesMots[12] = "avelc";
106     lesMots[13] = "avemc";
107     lesMots[14] = "avenc";
108     lesMots[15] = "aveoc";
109     lesMots[16] = "avepc";
110     lesMots[17] = "aveqc";
111     lesMots[18] = "averc";
112     lesMots[19] = "avesc";
113     lesMots[20] = "avetc";
114     lesMots[21] = "aveuc";
115     lesMots[22] = "avevc";
116     lesMots[23] = "avewc";
117     lesMots[24] = "avexc";
118     lesMots[25] = "aveyc";
119     lesMots[26] = "avezc";
120     lesMots[27] = "aveàc";
121     lesMots[28] = "aveéc";
122     lesMots[29] = "aveèc";
123     lesMots[30] = "aveëc";
124     lesMots[31] = "aveùc";
125     lesMots[32] = "aveûc";
126     lesMots[33] = "aveêc";
127     lesMots[34] = "aveîc";
128     lesMots[35] = "aveïc";
129     lesMots[36] = "aveçc";
130     lesMots[37] = "aveôc";
131
132     return lesMots;
133 }
134
135 void test_remplacer_ieme_lettre()
136 {
137     char *chaine = "avek";
138     Mot motACorriger = M_creerUnMot(chaine);
139     EnsembleDeMot ensemble = CO_remplacerIemeLettreEnBoucle(motACorriger, 4);
140     char **solution = creer_ensemble_solution();
```

```
141     for (int i = 0; i < 38; i++)
142     {
143         Mot mot = M_creerUnMot(solution[i]);
144         CU_ASSERT_TRUE(EDM_estPresent(ensemble, mot));
145         M_supprimerMot(&mot);
146     }
147     M_supprimerMot(&motACorriger);
148     while (!EDM_cardinalite(ensemble) == 0)
149     {
150         Mot tmp = EDM_obtenirMot(ensemble);
151         EDM_retirer(&ensemble, tmp);
152         M_supprimerMot(&tmp);
153     }
154     EDM_vider(&ensemble);
155     free(solution);
156 }
157
158 void test_inserer_lettre()
159 {
160     char *chaine = "avec";
161     Mot motACorriger = M_creerUnMot(chaine);
162     EnsembleDeMot ensemble = CO_insererIemeLettreEnBoucle(motACorriger, 4);
163     char **solution = creer_ensemble_solution_inserer();
164     for (int i = 0; i < 38; i++)
165     {
166         Mot mot = M_creerUnMot(solution[i]);
167         CU_ASSERT_TRUE(EDM_estPresent(ensemble, mot));
168         M_supprimerMot(&mot);
169     }
170     M_supprimerMot(&motACorriger);
171     while (!EDM_cardinalite(ensemble) == 0)
172     {
173         Mot tmp = EDM_obtenirMot(ensemble);
174         EDM_retirer(&ensemble, tmp);
175         M_supprimerMot(&tmp);
176     }
177     EDM_vider(&ensemble);
178     free(solution);
179 }
180
181 void test_strategie_remplacer_lettre()
182 {
183     Dictionnaire dico = creer_dictionnaire();
184     char *chaine = "abec";
185     char *chaine1 = "avec";
186     Mot motACorriger = M_creerUnMot(chaine);
187     Mot solution = M_creerUnMot(chaine1);
188     CorrecteurOrthographique correcteur = CO_correcteur(dico, motACorriger);
189     CO_strategieRemplacerLettres(&correcteur);
190     CU_ASSERT_TRUE(EDM_estPresent(correcteur.lesCorrections, solution));
191
192     M_supprimerMot(&motACorriger);
193     M_supprimerMot(&solution);
194     CO_supprimerCorrecteur(&correcteur);
```

```
195 }
196
197 void test_strategie_supprimer_lettre()
198 {
199     Dictionnaire dico = creer_dictionnaire();
200     char *chaine = "avvec";
201     char *chaine1 = "avec";
202     Mot motACorriger = M_creerUnMot(chaine);
203     Mot solution = M_creerUnMot(chaine1);
204     CorrecteurOrthographique correcteur = CO_correcteur(dico, motACorriger);
205     CO_strategieSupprimerLettres(&correcteur);
206     CU_ASSERT_TRUE(EDM_estPresent(correcteur.lesCorrections, solution));
207     M_supprimerMot(&solution);
208     M_supprimerMot(&motACorriger);
209     CO_supprimerCorrecteur(&correcteur);
210 }
211
212 void test_strategie_inverser_lettre()
213 {
214     Dictionnaire dico = creer_dictionnaire();
215     char *chaine = "aevc";
216     char *chaine1 = "avec";
217     Mot motACorriger = M_creerUnMot(chaine);
218     Mot solution = M_creerUnMot(chaine1);
219     CorrecteurOrthographique correcteur = CO_correcteur(dico, motACorriger);
220     CO_strategieInverserDeuxLettresConsecutives(&correcteur);
221     CU_ASSERT_TRUE(EDM_estPresent(correcteur.lesCorrections, solution));
222     M_supprimerMot(&motACorriger);
223     M_supprimerMot(&solution);
224     CO_supprimerCorrecteur(&correcteur);
225 }
226
227 void test_strategie_inserer_lettre()
228 {
229     Dictionnaire dico = creer_dictionnaire();
230     char *chaine = "avc";
231     char *chaine1 = "avec";
232     Mot motACorriger = M_creerUnMot(chaine);
233     Mot solution = M_creerUnMot(chaine1);
234     CorrecteurOrthographique correcteur = CO_correcteur(dico, motACorriger);
235     CO_strategieInsererLettres(&correcteur);
236     CU_ASSERT_TRUE(EDM_estPresent(correcteur.lesCorrections, solution));
237     M_supprimerMot(&motACorriger);
238     M_supprimerMot(&solution);
239     CO_supprimerCorrecteur(&correcteur);
240 }
241
242 void test_strategie_decomposer_mot()
243 {
244     Dictionnaire dico = creer_dictionnaire();
245     char *chaine = "avecarride";
246     char *chaine1 = "avec";
247     char *chaine2 = "arride";
248     Mot motACorriger = M_creerUnMot(chaine);
```

```
249     Mot solution1 = M_creerUnMot(chaine1);
250     Mot solution2 = M_creerUnMot(chaine2);
251     CorrecteurOrthographique correcteur = CO_correcteur(dico, motACorriger);
252     CO_strategieDecomposerMot(&correcteur);
253     CU_ASSERT_TRUE(EDM_estPresent(correcteur.lesCorrections, solution1) && EDM_estPresent(correcteur.
        lesCorrections, solution2));
254
255     M_supprimerMot(&motACorriger);
256     M_supprimerMot(&solution1);
257     M_supprimerMot(&solution2);
258     CO_supprimerCorrecteur(&correcteur);
259 }
260
261 int main(int argc, char **argv)
262 {
263     CU_pSuite pSuite = NULL;
264
265     /* initialisation du registre de tests */
266     if (CUE_SUCCESS != CU_initialize_registry())
267     {
268         return CU_get_error();
269     }
270     /* ajout d'une suite de test */
271     pSuite = CU_add_suite("Tests boite noire", init_suite_success, clean_suite_success);
272     if (NULL == pSuite)
273     {
274         CU_cleanup_registry();
275         return CU_get_error();
276     }
277
278     /* Ajout des tests     la suite de tests boite noire */
279     if ((NULL == CU_add_test(pSuite, "remplacer la bonne lettre", test_remplacer_ieme_lettre)) || (NULL
        == CU_add_test(pSuite, "inserer a la bonne place", test_inserer_lettre)) || (NULL == CU_add_test(
        pSuite, "test strategie remplacer", test_strategie_remplacer_lettre)) || (NULL == CU_add_test(pSuite
        , "test strategie supprimer", test_strategie_supprimer_lettre)) || (NULL == CU_add_test(pSuite, "
        test strategie inverser", test_strategie_inverser_lettre)) || (NULL == CU_add_test(pSuite, "test
        strategie inserer", test_strategie_inserer_lettre)) || (NULL == CU_add_test(pSuite, "test strategie
        decomposer", test_strategie_decomposer_mot))
280
281     )
282     {
283         CU_cleanup_registry();
284         return CU_get_error();
285     }
286
287     /* Lancement des tests */
288     CU_basic_set_mode(CU_BRM_VERBOSE);
289     CU_basic_run_tests();
290     printf("\n");
291     CU_basic_show_failures(CU_get_failure_list());
292     printf("\n\n");
293
294     /* Nettoyage du registre */
295     CU_cleanup_registry();
```

```
296     return CU_get_error();
297 }
```

../programme/src/testCO.c

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

#define TRUE 1
#define FALSE 0

int init_suite_success(void)
{
    return 0;
}

int clean_suite_success(void)
{
    return 0;
}

void creer_mots_A(Mot *mot1, Mot *mot2, Mot *mot3)
{
    *mot1 = M_creerUnMot("test");
    *mot2 = M_creerUnMot("unitaires");
    *mot3 = M_creerUnMot("ensembleDeMot");
}

void creer_mots_B(Mot *mot1, Mot *mot2, Mot *mot3)
{
    *mot1 = M_creerUnMot("test");
    *mot2 = M_creerUnMot("sans");
    *mot3 = M_creerUnMot("problèmes");
}

void test_ensemble_vide(void)
{
    EnsembleDeMot e = ensembleDeMot();

    CU_ASSERT_TRUE(EDM_cardinalite(e) == 0);

    EDM_vider(&e);
}

void test_ajouter_non_present(void)
{
    int c1, c2;
    Mot mot4 = M_creerUnMot("sans");
    EnsembleDeMot e = ensembleDeMot();
    Mot mot1, mot2, mot3;
    creer_mots_A(&mot1, &mot2, &mot3);
```

```
50      EDM_ajouter(&e, mot1);
51      EDM_ajouter(&e, mot2);
52      EDM_ajouter(&e, mot3);
53
54      c1 = EDM_cardinalite(e);
55      EDM_ajouter(&e, mot4);
56      c2 = EDM_cardinalite(e);
57
58      CU_ASSERT_TRUE(c1 + 1 == c2);
59      EDM_vider(&e);
60      M_supprimerMot(&mot1);
61      M_supprimerMot(&mot2);
62      M_supprimerMot(&mot3);
63      M_supprimerMot(&mot4);
64  }
65
66  void test_ajouter_present(void)
67  {
68      int c1, c2;
69      EnsembleDeMot e = ensembleDeMot();
70      Mot mot1, mot2, mot3;
71      creer_mots_A(&mot1, &mot2, &mot3);
72      EDM_ajouter(&e, mot1);
73      EDM_ajouter(&e, mot2);
74      EDM_ajouter(&e, mot3);
75
76      c1 = EDM_cardinalite(e);
77      EDM_ajouter(&e, mot1);
78      c2 = EDM_cardinalite(e);
79
80      CU_ASSERT_TRUE(c1 == c2);
81
82      EDM_vider(&e);
83      M_supprimerMot(&mot1);
84      M_supprimerMot(&mot2);
85      M_supprimerMot(&mot3);
86  }
87
88  void test_present_apres_ajout(void)
89  {
90      Mot mot5 = M_creerUnMot("problèmes");
91      EnsembleDeMot e = ensembleDeMot();
92      Mot mot1, mot2, mot3;
93      creer_mots_A(&mot1, &mot2, &mot3);
94      EDM_ajouter(&e, mot1);
95      EDM_ajouter(&e, mot2);
96      EDM_ajouter(&e, mot3);
97
98      EDM_ajouter(&e, mot5);
99
100     CU_ASSERT_TRUE(EDM_estPresent(e, mot5));
101
102     EDM_vider(&e);
103     M_supprimerMot(&mot5);
```

```
104     M_supprimerMot(&mot1);
105     M_supprimerMot(&mot2);
106     M_supprimerMot(&mot3);
107 }
108
109 void test_retirer_present(void)
110 {
111     int c1, c2;
112     EnsembleDeMot e = ensembleDeMot();
113     Mot mot1, mot2, mot3;
114     creer_mots_A(&mot1, &mot2, &mot3);
115     EDM_ajouter(&e, mot1);
116     EDM_ajouter(&e, mot2);
117     EDM_ajouter(&e, mot3);
118
119     c1 = EDM_cardinalite(e);
120     EDM_retirer(&e, mot2);
121     c2 = EDM_cardinalite(e);
122
123     CU_ASSERT_TRUE(c1 - 1 == c2);
124
125     EDM_vider(&e);
126     M_supprimerMot(&mot2);
127     M_supprimerMot(&mot1);
128     M_supprimerMot(&mot3);
129 }
130
131 void test_retirer_non_present(void)
132 {
133     int c1, c2;
134     Mot mot4 = M_creerUnMot("sans");
135     EnsembleDeMot e = ensembleDeMot();
136     Mot mot1, mot2, mot3;
137     creer_mots_A(&mot1, &mot2, &mot3);
138     EDM_ajouter(&e, mot1);
139     EDM_ajouter(&e, mot2);
140     EDM_ajouter(&e, mot3);
141
142     c1 = EDM_cardinalite(e);
143     EDM_retirer(&e, mot4);
144     c2 = EDM_cardinalite(e);
145
146     CU_ASSERT_TRUE(c1 == c2);
147
148     EDM_vider(&e);
149     M_supprimerMot(&mot1);
150     M_supprimerMot(&mot2);
151     M_supprimerMot(&mot3);
152     M_supprimerMot(&mot4);
153 }
154
155 void test_absent_apres_retrait(void)
156 {
157     EnsembleDeMot e = ensembleDeMot();
```

```
158     Mot mot1, mot2, mot3;
159     creer_mots_A(&mot1, &mot2, &mot3);
160     EDM_ajouter(&e, mot1);
161     EDM_ajouter(&e, mot2);
162     EDM_ajouter(&e, mot3);
163
164     EDM_retirer(&e, mot2);
165     CU_ASSERT_FALSE(EDM_estPresent(e, mot2));
166
167     EDM_vider(&e);
168     M_supprimerMot(&mot1);
169     M_supprimerMot(&mot2);
170     M_supprimerMot(&mot3);
171 }
172
173 void test_union(void)
174 {
175     EnsembleDeMot e1 = ensembleDeMot();
176     Mot mot1, mot1bis, mot2, mot3, mot4, mot5;
177     creer_mots_A(&mot1, &mot2, &mot3);
178     EDM_ajouter(&e1, mot1);
179     EDM_ajouter(&e1, mot2);
180     EDM_ajouter(&e1, mot3);
181
182     EnsembleDeMot e2 = ensembleDeMot();
183     creer_mots_B(&mot1bis, &mot4, &mot5);
184     EDM_ajouter(&e2, mot1);
185     EDM_ajouter(&e2, mot4);
186     EDM_ajouter(&e2, mot5);
187
188     EnsembleDeMot e3 = EDM_union(e1, e2);
189
190     CU_ASSERT_TRUE(EDM_estPresent(e3, mot1) && EDM_estPresent(e3, mot2) && EDM_estPresent(e3, mot3) &&
        EDM_estPresent(e3, mot4) && EDM_estPresent(e3, mot5));
191
192     EDM_vider(&e1);
193     EDM_vider(&e2);
194     EDM_vider(&e3);
195     M_supprimerMot(&mot1);
196     M_supprimerMot(&mot1bis);
197     M_supprimerMot(&mot2);
198     M_supprimerMot(&mot3);
199     M_supprimerMot(&mot4);
200     M_supprimerMot(&mot5);
201 }
202
203 void test_egalite_meme_ensemble(void)
204 {
205     EnsembleDeMot e1 = ensembleDeMot();
206     Mot mot1, mot2, mot3;
207     creer_mots_A(&mot1, &mot2, &mot3);
208     EDM_ajouter(&e1, mot1);
209     EDM_ajouter(&e1, mot2);
210     EDM_ajouter(&e1, mot3);
```

```
211
212     CU_ASSERT_TRUE(EDM_egale(e1, e1));
213
214     EDM_vider(&e1);
215     M_supprimerMot(&mot1);
216     M_supprimerMot(&mot2);
217     M_supprimerMot(&mot3);
218 }
219
220 void test_egalite_ensembles_differents(void)
221 {
222     EnsembleDeMot e1 = ensembleDeMot();
223     Mot mot1, mot1bis, mot2, mot3, mot4, mot5;
224     creer_mots_A(&mot1, &mot2, &mot3);
225     EDM_ajouter(&e1, mot1);
226     EDM_ajouter(&e1, mot2);
227     EDM_ajouter(&e1, mot3);
228
229     EnsembleDeMot e2 = ensembleDeMot();
230     creer_mots_B(&mot1bis, &mot4, &mot5);
231     EDM_ajouter(&e2, mot1);
232     EDM_ajouter(&e2, mot4);
233     EDM_ajouter(&e2, mot5);
234
235     CU_ASSERT_FALSE(EDM_egale(e1, e2));
236
237     EDM_vider(&e1);
238     EDM_vider(&e2);
239
240     M_supprimerMot(&mot1);
241     M_supprimerMot(&mot1bis);
242     M_supprimerMot(&mot2);
243     M_supprimerMot(&mot3);
244     M_supprimerMot(&mot4);
245     M_supprimerMot(&mot5);
246 }
247
248 void test_copier(void)
249 {
250     EnsembleDeMot e1 = ensembleDeMot();
251     Mot mot1, mot2, mot3;
252     creer_mots_A(&mot1, &mot2, &mot3);
253     EDM_ajouter(&e1, mot1);
254     EDM_ajouter(&e1, mot2);
255     EDM_ajouter(&e1, mot3);
256
257     EnsembleDeMot e2 = EDM_copier(e1);
258
259     CU_ASSERT_TRUE(EDM_egale(e1, e2));
260
261     EDM_vider(&e1);
262     EDM_vider(&e2);
263     M_supprimerMot(&mot1);
264     M_supprimerMot(&mot2);
```

```
265     M_supprimerMot(&mot3);
266 }
267
268 void test_obtenir_element()
269 {
270     EnsembleDeMot e = ensembleDeMot();
271     Mot mot1, mot2, mot3, mot3test;
272     creer_mots_A(&mot1, &mot2, &mot3);
273     EDM_ajouter(&e, mot1);
274     EDM_ajouter(&e, mot2);
275     EDM_ajouter(&e, mot3);
276
277     mot3test = EDM_obtenirMot(e);
278     CU_ASSERT_TRUE(M_sontIdentiques(mot3, mot3test));
279
280     EDM_vider(&e);
281     M_supprimerMot(&mot1);
282     M_supprimerMot(&mot2);
283     M_supprimerMot(&mot3);
284 }
285
286 int main(int argc, char **argv)
287 {
288     CU_pSuite pSuite = NULL;
289
290     /* initialisation du registre de tests */
291     if (CUE_SUCCESS != CU_initialize_registry())
292         return CU_get_error();
293
294     /* ajout d'une suite de test */
295     pSuite = CU_add_suite("Tests boite noire", init_suite_success, clean_suite_success);
296     if (NULL == pSuite)
297     {
298         CU_cleanup_registry();
299         return CU_get_error();
300     }
301
302     /* Ajout des tests à la suite de tests boite noire */
303     if ((NULL == CU_add_test(pSuite, "1 - La création d'un ensemble doit etre vide", test_ensemble_vide)
        ) || (NULL == CU_add_test(pSuite, "2 - Ajouter un mot non present incrémente la cardinalité",
        test_ajouter_non_present)) || (NULL == CU_add_test(pSuite, "3 - Ajouter un mot present n'incrémente
        pas la cardinalité", test_ajouter_present)) || (NULL == CU_add_test(pSuite, "4 - un élément ajouté
        est present", test_present_apres_ajout)) || (NULL == CU_add_test(pSuite, "5 - retirer un élement pré
        sent décrémente la cardinalité", test_retirer_non_present)) || (NULL == CU_add_test(pSuite, "6 -
        retirer un élement present ne decremente pas la cardinalite", test_retirer_present)) || (NULL ==
        CU_add_test(pSuite, "7 - un element retire n'est plus present", test_absent_apres_retrait)) || (NULL
         == CU_add_test(pSuite, "8 - union", test_union)) || (NULL == CU_add_test(pSuite, "9 - un ensemble
        est égal a lui meme", test_egalite_meme_ensemble)) || (NULL == CU_add_test(pSuite, "10 - un ensemble
         est different d'un autre ensemble", test_egalite_ensembles_differents)) || (NULL == CU_add_test(
        pSuite, "11 - un ensemble est égal a une de ses copies", test_copier)) || (NULL == CU_add_test(
        pSuite, "12 - obtenir un élément d'un ensemble renvoie le dernier élément ajouté",
        test_obtenir_element)))
304     {
305         CU_cleanup_registry();
```

```
306             return CU_get_error();
307         }
308 
309         /* Lancement des tests */
310         CU_basic_set_mode(CU_BRM_VERBOSE);
311         CU_basic_run_tests();
312         printf("\n");
313         CU_basic_show_failures(CU_get_failure_list());
314         printf("\n\n");
315 
316         /* Nettoyage du registre */
317         CU_cleanup_registry();
318         return CU_get_error();
319     }
```

../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   {
10      return 0;
11  }
12 
13  int clean_suite_success(void)
14  {
15      return 0;
16  }
17 
18  ListeChaineeDeMot creer_liste_avec_un_mot()
19  {
20      ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
21      char chaine1[] = "test";
22      Mot unMot = M_creerUnMot(chaine1);
23      LCDM_ajouter(&l, unMot);
24      return l;
25  }
26 
27  ListeChaineeDeMot creer_liste_avec_deux_mot()
28  {
29      ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
30      char *chaine1 = "chaineun";
31      char *chaine2 = "chainedeux";
32      Mot unMot = M_creerUnMot(chaine1);
33      Mot unAutreMot = M_creerUnMot(chaine2);
34      LCDM_ajouter(&l, unMot);
35      LCDM_ajouter(&l, unAutreMot);
36      return l;
37  }
```

```
38
39  void test_liste_vide(void)
40  {
41      ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
42      CU_ASSERT_TRUE(LCDM_estVide(l));
43      LCDM_supprimer(&l);
44  }
45
46  void test_liste_non_vide(void)
47  {
48      ListeChaineeDeMot l = creer_liste_avec_un_mot();
49      CU_ASSERT_TRUE(!LCDM_estVide(l));
50      Mot mot = LCDM_obtenirMot(l);
51      M_supprimerMot(&mot);
52      LCDM_supprimer(&l);
53  }
54
55  void test_mot_ajoute_en_tete(void)
56  {
57      ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
58      char *chaine1 = "chaine";
59      Mot unMot = M_creerUnMot(chaine1);
60      LCDM_ajouter(&l, unMot);
61      CU_ASSERT_EQUAL(M_sontIdentiques(LCDM_obtenirMot(l), unMot), true);
62      LCDM_supprimer(&l);
63      M_supprimerMot(&unMot);
64  }
65
66  void test_supprimer_mot(void)
67  {
68      ListeChaineeDeMot l1 = LCDM_listeChaineeDeMot();
69      ListeChaineeDeMot l2 = LCDM_listeChaineeDeMot();
70      char *chaine1 = "chaineun";
71      char *chaine2 = "chainedeux";
72      char *chaine3 = "chainetrois";
73      Mot unMot = M_creerUnMot(chaine1);
74      Mot unAutreMot = M_creerUnMot(chaine2);
75      Mot toujoursPlusDeMot = M_creerUnMot(chaine3);
76      LCDM_ajouter(&l1, unMot);
77      LCDM_ajouter(&l1, unAutreMot);
78      LCDM_ajouter(&l1, toujoursPlusDeMot);
79      LCDM_ajouter(&l2, unMot);
80      LCDM_ajouter(&l2, unAutreMot);
81      LCDM_supprimerMot(&l1, toujoursPlusDeMot);
82      CU_ASSERT_TRUE(LCDM_egale(l1, l2));
83      LCDM_supprimer(&l1);
84      LCDM_supprimer(&l2);
85      M_supprimerMot(&unMot);
86      M_supprimerMot(&unAutreMot);
87      M_supprimerMot(&toujoursPlusDeMot);
88  }
89
90  void test_obtenir_liste_suivante(void)
91  {
```

```
92      ListeChaineeDeMot lSuivante;
93      ListeChaineeDeMot l = LCDM_listeChaineeDeMot();
94      char *chaine1 = "chaineun";
95      char *chaine2 = "chainedeux";
96      Mot unMot = M_creerUnMot(chaine1);
97      Mot unAutreMot = M_creerUnMot(chaine2);
98      LCDM_ajouter(&l, unMot);
99      lSuivante = l;
100     LCDM_ajouter(&l, unAutreMot);
101     CU_ASSERT_PTR_EQUAL(LCDM_obtenirListeSuivante(l), lSuivante);
102     LCDM_supprimer(&l);
103     M_supprimerMot(&unMot);
104     M_supprimerMot(&unAutreMot);
105 }
106
107 void test_fixer_liste_suivante(void)
108 {
109     ListeChaineeDeMot l1 = LCDM_listeChaineeDeMot();
110     ListeChaineeDeMot l2 = LCDM_listeChaineeDeMot();
111     ListeChaineeDeMot temp = LCDM_listeChaineeDeMot();
112     char *chaine1 = "chaineun";
113     char *chaine2 = "chainedeux";
114     char *chaine3 = "chainetrois";
115     Mot unMot = M_creerUnMot(chaine1);
116     Mot unAutreMot = M_creerUnMot(chaine2);
117     Mot toujoursPlusDeMot = M_creerUnMot(chaine3);
118     LCDM_ajouter(&l1, unMot);
119     temp = l1;
120     LCDM_ajouter(&l1, unAutreMot);
121     LCDM_ajouter(&l2, toujoursPlusDeMot);
122     LCDM_fixerListeSuivante(&l1, l2);
123     CU_ASSERT_PTR_EQUAL(LCDM_obtenirListeSuivante(l1), l2);
124     LCDM_supprimer(&l1);
125     LCDM_supprimer(&temp);
126     M_supprimerMot(&unMot);
127     M_supprimerMot(&unAutreMot);
128     M_supprimerMot(&toujoursPlusDeMot);
129 }
130
131 void test_copie_egale(void)
132 {
133     ListeChaineeDeMot l1 = creer_liste_avec_deux_mot();
134     ListeChaineeDeMot l2 = LCDM_copier(l1);
135     CU_ASSERT_TRUE(LCDM_egale(l1, l2));
136     Mot mot1 = LCDM_obtenirMot(LCDM_obtenirListeSuivante(l1));
137     Mot mot3 = LCDM_obtenirMot(l1);
138     M_supprimerMot(&mot1);
139     M_supprimerMot(&mot3);
140     LCDM_supprimer(&l1);
141     LCDM_supprimer(&l2);
142 }
143
144 void test_differente(void)
145 {
```

```
146     ListeChaineeDeMot l1 = creer_liste_avec_deux_mot();
147     ListeChaineeDeMot l2 = creer_liste_avec_un_mot();
148     CU_ASSERT_FALSE(LCDM_egale(l1, l2));
149     Mot mot = LCDM_obtenirMot(l1);
150     Mot mot2 = LCDM_obtenirMot(LCDM_obtenirListeSuivante(l1));
151     Mot mot4 = LCDM_obtenirMot(l2);
152     M_supprimerMot(&mot);
153     M_supprimerMot(&mot2);
154     M_supprimerMot(&mot4);
155     LCDM_supprimer(&l1);
156     LCDM_supprimer(&l2);
157 }
158
159 int main(int argc, char **argv)
160 {
161     CU_pSuite pSuite = NULL;
162
163     /* initialisation du registre de tests */
164     if (CUE_SUCCESS != CU_initialize_registry())
165     {
166         return CU_get_error();
167     }
168     /* ajout d'une suite de test */
169     pSuite = CU_add_suite("Tests boite noire", init_suite_success, clean_suite_success);
170     if (NULL == pSuite)
171     {
172         CU_cleanup_registry();
173         return CU_get_error();
174     }
175
176     /* Ajout des tests     la suite de tests boite noire */
177     if ((NULL == CU_add_test(pSuite, "la creation d'une liste qui doit etre vide", test_liste_vide)) ||
        (NULL == CU_add_test(pSuite, "une liste contenant un element n'est pas vide", test_liste_non_vide))
        || (NULL == CU_add_test(pSuite, "un element ajoute est en tete de liste", test_mot_ajoute_en_tete))
        || (NULL == CU_add_test(pSuite, "supprimer un mot", test_supprimer_mot)) || (NULL == CU_add_test(
        pSuite, "obtenir liste suivante", test_obtenir_liste_suivante)) || (NULL == CU_add_test(pSuite, "
        fixer liste suivante", test_fixer_liste_suivante)) || (NULL == CU_add_test(pSuite, "une liste et sa
        copie sont egales", test_copie_egale)) || (NULL == CU_add_test(pSuite, "deux listes differentes ne
        sont pas egales", test_differente)))
178     {
179         CU_cleanup_registry();
180         return CU_get_error();
181     }
182
183     /* Lancement des tests */
184     CU_basic_set_mode(CU_BRM_VERBOSE);
185     CU_basic_run_tests();
186     printf("\n");
187     CU_basic_show_failures(CU_get_failure_list());
188     printf("\n\n");
189
190     /* Nettoyage du registre */
191     CU_cleanup_registry();
192     return CU_get_error();
```

```
193 }
```

../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  int init_suite_success(void)
8  {
9      return 0;
10 }
11
12 int clean_suite_success(void)
13 {
14     return 0;
15 }
16
17 Mot creer_un_mot()
18 {
19     Mot mot;
20     mot.chaine = "lala";
21     mot.longueur = strlen(mot.chaine);
22     return mot;
23 }
24
25 Mot creer_mot_vide()
26 {
27     Mot mot;
28     mot.chaine = "";
29     mot.longueur = 0;
30     return mot;
31 }
32
33 char creer_cara_alpha_accent()
34 {
35     return 'é';
36 }
37
38 char creer_cara_alpha_sans_accent()
39 {
40     return 'a';
41 }
42
43 char creer_cara_pas_alpha()
44 {
45     return '!';
46 }
47
48 char *creer_mot_valide_avec_accent()
49 {
50     return "fatiguée";
```

```
51  }
52
53  char *creer_mot_valide_sans_accent()
54  {
55      return "valide";
56  }
57
58  char *creer_mot_non_valide()
59  {
60      return "val!de";
61  }
62
63  void test_caractere_alpha_accent()
64  {
65      char c = creer_cara_alpha_accent();
66      CU_ASSERT_TRUE(M_estUnCaractereAlphabetique(c));
67  }
68
69  void test_caractere_alpha_sans_accent()
70  {
71      char c = creer_cara_alpha_sans_accent();
72      CU_ASSERT_TRUE(M_estUnCaractereAlphabetique(c));
73  }
74
75  void test_pas_caractere_alpha()
76  {
77      char c = creer_cara_pas_alpha();
78      CU_ASSERT_TRUE(!M_estUnCaractereAlphabetique(c));
79  }
80
81  void test_mot_valide_avec_accent()
82  {
83      char *c = creer_mot_valide_avec_accent();
84      CU_ASSERT_TRUE(M_estUnMotValide(c));
85  }
86
87  void test_reduire_la_casse()
88  {
89      char *c = creer_mot_valide_avec_accent();
90      char *c1 = "FAtiguÉe";
91      Mot mot = M_creerUnMot(c);
92      Mot mot1 = M_creerUnMot(c1);
93      CU_ASSERT_TRUE(M_sontIdentiques(mot, mot1));
94      M_supprimerMot(&mot);
95      M_supprimerMot(&mot1);
96  }
97
98  void test_mot_valide_sans_accent()
99  {
100     char *c = creer_mot_valide_sans_accent();
101     CU_ASSERT_TRUE(M_estUnMotValide(c));
102 }
103
104 void test_mot_non_valide()
```

```
105 {
106     char *c = creer_mot_non_valide();
107     CU_ASSERT_TRUE(!M_estUnMotValide(c));
108 }
109
110 void test_copier_mot()
111 {
112     Mot mot = M_creerUnMot(creer_mot_valide_avec_accent());
113     Mot mot2 = M_copierMot(mot);
114     CU_ASSERT_TRUE(M_sontIdentiques(mot, mot2));
115     M_supprimerMot(&mot);
116     M_supprimerMot(&mot2);
117 }
118
119 void test_ieme_caractere()
120 {
121     Mot mot = M_creerUnMot(creer_mot_valide_avec_accent());
122     CU_ASSERT_TRUE(M_iemeCaractere(mot, 1) == 'f');
123     CU_ASSERT_TRUE(M_iemeCaractere(mot, 2) == 'a');
124     CU_ASSERT_TRUE(M_iemeCaractere(mot, 3) == 't');
125     CU_ASSERT_TRUE(M_iemeCaractere(mot, 7) == 'é');
126     M_supprimerMot(&mot);
127 }
128
129 void test_sont_identiques()
130 {
131     Mot mot = M_creerUnMot(creer_mot_valide_avec_accent());
132     Mot mot2 = M_copierMot(mot);
133     CU_ASSERT_TRUE(M_sontIdentiques(mot, mot2));
134     M_supprimerMot(&mot);
135     M_supprimerMot(&mot2);
136 }
137
138 void test_supprimer_ieme_lettre()
139 {
140     char *c = "fatguée";
141     Mot mot = M_creerUnMot(creer_mot_valide_avec_accent());
142     Mot mot3 = M_copierMot(mot);
143     Mot mot2 = M_creerUnMot(c);
144     M_supprimerIemeLettre(&mot, 4);
145     CU_ASSERT_TRUE(M_sontIdentiques(mot, mot2));
146     CU_ASSERT_TRUE(!M_sontIdentiques(mot, mot3));
147     M_supprimerMot(&mot);
148     M_supprimerMot(&mot2);
149     M_supprimerMot(&mot3);
150 }
151
152 void test_inverser_deux_lettres_consecutives()
153 {
154     char *c = "fatiguée";
155     char *c1 = "fatgiuée";
156     Mot mot = M_creerUnMot(c);
157     Mot mot2 = M_creerUnMot(c1);
158     M_inverserDeuxLettresConsecutives(&mot, 4);
```

```
159     CU_ASSERT_TRUE(M_sontIdentiques(mot, mot2));
160     M_supprimerMot(&mot);
161     M_supprimerMot(&mot2);
162 }
163
164 void test_inserer_lettre()
165 {
166     char *c = "faiguée";
167     char *c1 = "fatiguée";
168     Mot mot = M_creerUnMot(c);
169     Mot mot2 = M_creerUnMot(c1);
170     M_insererLettre(&mot, 3, 't');
171     CU_ASSERT_TRUE(M_sontIdentiques(mot, mot2));
172     M_supprimerMot(&mot);
173     M_supprimerMot(&mot2);
174 }
175
176 void test_decomposer_mot()
177 {
178     char *c = "fati";
179     char *c1 = "guée";
180     Mot mot = M_creerUnMot(c);
181     Mot mot1 = M_creerUnMot(c1);
182     Mot mot2 = M_creerUnMot(creer_mot_valide_avec_accent());
183     Mot mot3 = M_decomposerMot(&mot2, 5);
184     CU_ASSERT_TRUE(M_sontIdentiques(mot3, mot));
185     CU_ASSERT_TRUE(M_sontIdentiques(mot2, mot1));
186     M_supprimerMot(&mot);
187     M_supprimerMot(&mot1);
188     M_supprimerMot(&mot2);
189     M_supprimerMot(&mot3);
190 }
191
192 int main(int argc, char **argv)
193 {
194     CU_pSuite pSuite = NULL;
195
196     /* initialisation du registre de tests */
197     if (CUE_SUCCESS != CU_initialize_registry())
198     {
199         return CU_get_error();
200     }
201     /* ajout d'une suite de test */
202     pSuite = CU_add_suite("Tests boite noire", init_suite_success, clean_suite_success);
203     if (NULL == pSuite)
204     {
205         CU_cleanup_registry();
206         return CU_get_error();
207     }
208
209     /* Ajout des tests     la suite de tests boite noire */
210     if ((NULL == CU_add_test(pSuite, "un caractere avec accent est un caractere alphabetique",
        test_caractere_alpha_accent)) || (NULL == CU_add_test(pSuite, "un caractere sans accent est un
        caractere alphabetique", test_caractere_alpha_sans_accent)) || (NULL == CU_add_test(pSuite, "un
```

```
        caractere non alphabetique ne l'est pas", test_pas_caractere_alpha)) || (NULL == CU_add_test(pSuite,
         "un mot non valide n'est pas valide", test_mot_non_valide)) || (NULL == CU_add_test(pSuite, "un mot
         valide avec accent est valide", test_mot_valide_avec_accent)) || (NULL == CU_add_test(pSuite, "un
        mot valide sans accent est valide", test_mot_valide_sans_accent)) || (NULL == CU_add_test(pSuite, "
        creer un mot reduit la casse", test_reduire_la_casse)) || (NULL == CU_add_test(pSuite, "la copie est
         identique au mot", test_copier_mot)) || (NULL == CU_add_test(pSuite, "la copie et le mot sont
        identiques", test_sont_identiques)) || (NULL == CU_add_test(pSuite, "ieme caractere renvoie bien le
        ieme caractere", test_ieme_caractere)) || (NULL == CU_add_test(pSuite, "supprimer ieme caractere
        supprimer bien le ieme caractere", test_supprimer_ieme_lettre)) || (NULL == CU_add_test(pSuite, "
        inverser deux lettres fontionne", test_inverser_deux_lettres_consecutives)) || (NULL == CU_add_test(
        pSuite, "inserer une lettre a la bonne place", test_inserer_lettre)) || (NULL == CU_add_test(pSuite,
         "decompo ok", test_decomposer_mot))
211
212     )
213     {
214         CU_cleanup_registry();
215         return CU_get_error();
216     }
217
218     /* Lancement des tests */
219     CU_basic_set_mode(CU_BRM_VERBOSE);
220     CU_basic_run_tests();
221     printf("\n");
222     CU_basic_show_failures(CU_get_failure_list());
223     printf("\n\n");
224
225     /* Nettoyage du registre */
226     CU_cleanup_registry();
227     return CU_get_error();
228 }
```

../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 et Dictionnaire	Rapport
Implémentation	Rapport et FichierTexte	CorrecteurOrthographique, Dictionnaire, EnsembleDeMot et IHM	Dictionnaire, ArbreDeLettres et ListeChaineeDeMot	Mot et CorrigerTexte
Tests	Mot	Dictionnaire et EnsembleDeMot	CorrecteurOrthographique et ListeChaineeDeMot	Dictionnaire et ArbreDeLettres

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 ait 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: Ce projet a été pour moi une vraie expérience. Je n'avais jamais eu de travail de groupe de cette ampleur. Beaucoup de nouveautés, comme l'utilisation de GIT, du LaTeXet globalement l'organisation au sein du groupe. La communication dans notre groupe s'est très bien passée (beaucoup de réunions à distance) ce qui nous permettait constamment de s'entraider sur nos parties. Le groupe était motivé et efficace. En réalisant les efforts au fur et à mesure (comme la documentation ou encore le nommage clair des fonctions et variables) nous avons évité de nombreuses difficultés. Nous avons cependant pris du retard sur le planning, et avons dû consacrer une partie non-négligeable de nos vacances sur le projet. J'ai également appris à utiliser l'outil Valgrind, qui nous a été d'une grande aide pour améliorer notre code. Enfin, j'ai très largement perfectionné mes connaissances en C en me confrontant directement aux différentes erreurs et j'ai développé une rigueur de développement en suivant les étapes et en testant régulièrement les modifications ajoutées au projet. Je suis très satisfait du résultat obtenu et ce projet m'a motivé à utiliser le C pour d'autres applications.

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 l'estimation du temps de résolution d'un problème ont représenté nos plus grandes difficultés. 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.