TP1 - Création et manipulation d'un lexique

- SEC 2
- Eloi Tourangin - eloi.tourangin@eleves.ec-nantes.fr
- Thomas Verron - thomas.verron@eleves.ec-nantes.fr

Ce projet implémente deux classes en C++ :

- La classe `lexique` qui gère un dictionnaire de mots avec leur nombre d'occurrences
- La classe `lexique_ligne`, héritant de `lexique`, qui ajoute le suivi des numéros de ligne où apparaît chaque mot

Table des matières

Compilation et éxecution

```
# Compiler
make

# Exécuter
./build/main.out
```

ou executer le script bash

```
./build_and_execute.sh
```

Partie 1 : Classe lexique

Structure et implémentation

```
class lexique {
private:
    string nom;                  // Nom du lexique
    vector<string> mots;         // Liste des mots uniques
    vector<int> frequences;      // Fréquences correspondantes
};
```

La classe utilise deux vecteurs parallèles pour stocker les mots et leurs fréquences. Pour chaque mot à l'index i dans `mots`, sa fréquence est stockée à l'index i dans `frequences`.

Détail des fonctions et leurs implémentations

1. Constructeur avec fichier

```
lexique::lexique(string n, string f)
```

Ce constructeur crée un lexique à partir d'un fichier texte. Voici son fonctionnement détaillé :

```
this->nom = n;
string contenu;
if(!util::readFileIntoString(f, contenu)) {
    cerr << "Error reading file " << f << endl;
    return;
```

```
}

util::to_lower(contenu);
```

1. Initialisation :
    - Assigne le nom du lexique
    - Lit le fichier complet avec `readFileIntoString`
    - Convertit tout en minuscules

```
pch = strtok(const_cast<char*>(contenu.c_str()),
            " \"!#$%&'()*+,-./:;<=>?@[\\]^_`{|}~0123456789\n\r\t");

while (pch != NULL) {
    string tmp = string(pch);
    util::remove_punctuation(tmp);
    pch = const_cast<char*>(tmp.c_str());

    it_mots = find(ptr_mots->begin(), ptr_mots->end(), pch);

    if (it_mots != ptr_mots->end()) {
        int index = distance(ptr_mots->begin(), it_mots);
        (*ptr_frequences)[index]++;
    } else {
        ptr_mots->push_back(string(pch));
        ptr_frequences->push_back(1);
    }

    pch = strtok(NULL, " \"!#$%&'()*+,-./:;<=>?
@[\\]^_`{|}~0123456789\n\r\t");
}
```

2. Traitement du texte :
    - Découpe le texte en mots avec `strtok`
    - Pour chaque mot :
        - Nettoie la ponctuation restante
        - Cherche si le mot existe déjà
        - Si oui : incrémente sa fréquence
        - Si non : ajoute le mot avec fréquence 1

2. Constructeur lexique vide

```
lexique::lexique(string n) {
    this->nom = n;
    this->mots = vector<string>();
    this->frequences = vector<int>();
}
```

Crée un lexique vide :

- Initialise le nom
- Crée des vecteurs vides pour les mots et fréquences

3. Destructeur

```
lexique::~lexique() {
    this->mots.clear();
    this->frequences.clear();
    return;
}
```

Nettoie la mémoire en vidant les vecteurs de mots et fréquences.

4. Sauvegarde du lexique

```
void lexique::saveLexique(string f) {
    ofstream Fichier(f);
    const int largeur = 40;
    for(string mot : mots) {
        vector<string>::iterator it = find(mots.begin(), mots.end(), mot);
        int espace = 60 - mot.size();
        Fichier << mot << string(espace,' ') << ":       "
                << frequences[it-mots.begin()] << endl;
    }
    Fichier.close();
}
```

Fonctionnement :

1. Ouvre un fichier en écriture
2. Pour chaque mot :
    - Calcule l'espacement pour un alignement visuel
    - Écrit le mot, des espaces de formatage, et sa fréquence
3. Ferme le fichier

Note : L'utilisation de `find` à chaque itération n'est pas optimale puisque nous itérons déjà sur les mots dans l'ordre. Une amélioration serait d'utiliser directement l'index de la boucle.

5. Recherche de fréquence

```
int lexique::getFrequenceFromWord(string mot) const {
    vector<string>::const_iterator it = find(mots.begin(), mots.end(),
mot);
    if(it != mots.end()) {
        return frequences[it-mots.begin()];
```

```
    }
    return 0;
}
```

Fonctionnement :

1. Cherche le mot dans le vecteur avec `find`
2. Si trouvé : retourne sa fréquence en utilisant la distance depuis le début
3. Si non trouvé : retourne 0

6. Suppression d'un mot

```
void lexique::deleteWord(string mot) {
    vector<string>::iterator it = find(mots.begin(), mots.end(), mot);
    if(it != mots.end()) mots.erase(it);
}
```

Fonctionnement :

1. Cherche le mot dans le vecteur
2. Si trouvé : le supprime
3. Si non trouvé : ne fait rien

Note : Cette implémentation ne supprime que le mot, pas sa fréquence associée.

7. Affichage du nombre de mots

```
void lexique::displayNbWords() {
    cout << mots.size();
}
```

Affiche simplement la taille du vecteur de mots.

8. Opérateur de fusion (+)

```
lexique lexique::operator+(const lexique& autre) const {
    lexique nouveauLexique("Nouveau Lexique");
    nouveauLexique.mots = this->mots;
    nouveauLexique.frequences = this->frequences;

    for(string mot : autre.mots) {
        vector<string>::iterator it = find(nouveauLexique.mots.begin(),
                                           nouveauLexique.mots.end(), mot);
        if(it != nouveauLexique.mots.end()) {
            size_t index = distance(nouveauLexique.mots.begin(), it);
            nouveauLexique.frequences[index] +=
```

```
autre.getFrequenceFromWord(mot);
        } else {
            nouveauLexique.mots.push_back(mot);

nouveauLexique.frequences.push_back(autre.getFrequenceFromWord(mot));
        }
    }
    return nouveauLexique;
}
```

Fonctionnement :

1. Crée un nouveau lexique et copie les mots et fréquences du premier lexique
2. Pour chaque mot du second lexique :
    - Si le mot existe déjà : additionne les fréquences
    - Si le mot n'existe pas : l'ajoute avec sa fréquence

9. Opérateur de différence (-)

```
lexique lexique::operator-(const lexique& autre) const {
    lexique nouveauLexique("Nouveau Lexique");
    nouveauLexique.mots = this->mots;
    nouveauLexique.frequences = this->frequences;

    for(string mot : autre.mots) {
        vector<string>::iterator it = find(nouveauLexique.mots.begin(),
                                           nouveauLexique.mots.end(), mot);
        if(it != nouveauLexique.mots.end()) {
            size_t index = distance(nouveauLexique.mots.begin(), it);
            int nouvelleFrequence = nouveauLexique.frequences[index]
                                  - autre.getFrequenceFromWord(mot);
            if(nouvelleFrequence > 0) {
                nouveauLexique.frequences[index] = nouvelleFrequence;
            } else {
                nouveauLexique.mots.erase(it);
                nouveauLexique.frequences.erase(
                    nouveauLexique.frequences.begin() + index);
            }
        }
    }
    return nouveauLexique;
}
```

Fonctionnement :

1. Crée un nouveau lexique et copie les mots/fréquences du premier
2. Pour chaque mot du second lexique :
    - Si le mot existe dans le premier :
        - Soustrait la fréquence

        - Si résultat > 0 : met à jour la fréquence
        - Si résultat ≤ 0 : supprime le mot et sa fréquence
    - Si le mot n'existe pas : passe au suivant

10. Opérateur d'affichage (<<)

```
ostream& operator<<(ostream& os, const lexique& lexique) {
    for(string mot : lexique.mots) {
        vector<string>::const_iterator it = find(lexique.mots.begin(),
                                                 lexique.mots.end(), mot);
        os << mot << " : " << lexique.frequences[it-lexique.mots.begin()]
<< endl;
    }
    return os;
}
```

Fonctionnement :

1. Pour chaque mot :
    - Trouve sa position dans le vecteur
    - Affiche le mot et sa fréquence
2. Retourne le flux pour permettre le chaînage

Note : Comme pour `saveLexique`, l'utilisation de `find` pourrait être évitée.

Complexité et performances

La structure actuelle utilise deux vecteurs parallèles et `std::find` pour les recherches, ce qui donne les complexités suivantes :

- Recherche d'un mot : O(N) via `std::find`
- Ajout d'un nouveau mot : O(N) pour vérifier l'unicité + O(1) pour l'insertion
- Fusion de lexiques : O(M*N) où M et N sont les tailles des lexiques
- Différence de lexiques : O(M*N)

Une amélioration majeure serait d'utiliser `unordered_map<string,int>` pour obtenir :

- Recherche : O(1) en moyenne
- Ajout : O(1) en moyenne
- Fusion/Différence : O(M + N)

Exemple d'utilisation

```
int main() {
    // Création des lexiques
    lexique lex1("Les Miserables", "assets/lesMiserables_A.txt");
    lexique lex2("Notre Dame", "assets/notreDameDeParis_A.txt");

    // Sauvegarde
```

```
    lex1.saveLexique("assets/lexique_1.txt");
    lex2.saveLexique("assets/lexique_2.txt");

    // Opérations
    cout << "Fréquence de 'the': " << lex1.getFrequenceFromWord("the") <<
endl;

    // Fusion
    lexique fusion = lex1 + lex2;
    fusion.saveLexique("assets/fusion.txt");

    // Différence
    lexique diff = fusion - lex2;
    diff.saveLexique("assets/difference.txt");

    return 0;
}
```

Tests de la classe `lexique`

Test du constructeur vide

- Création d'un lexique vide
- Résultat : 0 mot comme attendu

Test du constructeur avec fichier (Les Misérables)

- Construction depuis le fichier texte
- Nombre de mots distincts : 22 834
- Sauvegarde réussie dans `assets/lexique_lesMiserables.txt`

Test du second constructeur (Notre Dame de Paris)

- Construction depuis un autre fichier texte
- Nombre de mots distincts : 14 013

Test de l'opérateur de fusion (+)

- Fusion de Les Misérables et Notre Dame de Paris
- Résultat : 26 852 mots distincts
- Démonstration de l'union des vocabulaires avec addition des fréquences

Test de l'opérateur de différence (-)

- Soustraction de Notre Dame de Paris du lexique fusionné
- Résultat : 22 834 mots (retour au lexique de Les Misérables)
- Vérifie la cohérence de l'opération inverse

Test de recherche et suppression

- Mot test : "the"
- Fréquence initiale : 40 917 occurrences
- Après suppression : 0 occurrence
- Démontre la suppression complète du mot

Test de l'opérateur d'affichage

- Test sur un petit lexique de test
- Exemple de sortie :

```
lorem : 8
ipsum : 6
dolor : 9
sit : 9
amet : 8
consectetur : 7
elit : 8
adipiscing : 8
```

Partie 2 : Classe lexique_ligne

La classe `lexique_ligne` hérite de la classe `lexique` et ajoute la fonctionnalité de suivi des numéros de ligne où apparaît chaque mot.

Organisation des données

```
class lexique_ligne : public lexique {
private:
    map<string, vector<int>> lexique_map;  // Stocke les numéros de ligne
pour chaque mot
};
```

En plus des structures héritées de la classe mère, cette classe ajoute une map qui associe à chaque mot un vecteur contenant les numéros des lignes où il apparaît. Cette structure permet une recherche efficace des positions d'un mot dans le texte.

Détail des fonctions

Constructeur principal

```
lexique_ligne::lexique_ligne(string n, string f) : lexique(n, f)
```

Ce constructeur étend celui de la classe mère avec les fonctionnalités suivantes :

Étape 1 : Initialisation

- Appelle le constructeur de la classe mère
- Crée une entrée vide dans la map pour chaque mot existant

Étape 2 : Traitement ligne par ligne

- Lit le fichier ligne par ligne
- Pour chaque ligne :
    - La convertit en minuscules
    - La découpe en mots
    - Pour chaque mot :
        - Nettoie la ponctuation
        - Si le mot n'est pas vide :
            - L'ajoute à la map s'il n'existe pas
            - Ajoute le numéro de ligne s'il n'est pas déjà présent

2. Destructeur

```
lexique_ligne::~lexique_ligne()
```

Nettoie la mémoire en vidant la map des lignes.

3. Affichage des lignes d'un mot

```
void lexique_ligne::displayLinesFromWord(string mot) const
```

Cette méthode affiche les positions d'un mot dans le texte :

- Recherche d'abord le mot dans la map
- Si le mot est trouvé :
    - Affiche "Le mot 'xxx' apparaît aux lignes : "
    - Liste tous les numéros de ligne où le mot apparaît
- Si le mot n'est pas trouvé :
    - Affiche un message d'erreur approprié

Utilisation détaillée

Voici un exemple complet montrant l'utilisation de la classe `lexique_ligne` :

```
int main() {
    // Création du lexique avec suivi des lignes
    lexique_ligne lex("Les Miserables", "assets/lesMiserables_A.txt");

    // Affichage des lignes où apparaît un mot
    lex.displayLinesFromWord("jean");  // Affiche les numéros de ligne où
```

```
    "jean" apparaît

        // On peut toujours utiliser les méthodes de la classe mère
        cout << "Fréquence du mot 'jean': " << lex.getFrequenceFromWord("jean")
    << endl;

        return 0;
    }
```

Tests de la classe `lexique_ligne`

Test du constructeur

- Construction depuis Les Misérables
- Vérification de l'héritage : même nombre de mots (22 834)

Tests de displayLinesFromWord

Test avec mot fréquent :

- Mot "included"
- Apparaît aux lignes : 5, 2198, 14921, 39847, 53007, 63416
- Total : 6 occurrences

Test avec autre mot fréquent :

- Mot "unique"
- Apparaît aux lignes : 17529, 22988, 23432, 45471, 48117, 48185, 53702, 61758
- Total : 8 occurrences

Test avec mot inexistant :

- Test avec "xyzabc123"
- Message d'erreur approprié affiché

Test de l'héritage

- Vérification des méthodes héritées
- `displayNbWords()` : 22 834 mots
- `getFrequenceFromWord("included")` : 6 occurrences
- Démontre le bon fonctionnement de l'héritage