Algorithmique et Programmation
Avril 2023

---

# Projet : Saisie Prédictive

---

Théo Gachet
Kawtar El Mamoun

# Table des matières

# 1 Présentation du sujet

## 1.1 Saisie intuitive

Le but du projet est de réaliser une interface homme-machine permettant de proposer des suggestions à une saisie incomplète par l'utilisateur. L'application mise en place doit aussi être capable d'acquérir et de stocker les mots saisis ainsi que les mots les plus fréquemment utilisés. L'orthographe sera systématiquement vérifiée et l'utilisateur aura la possibilité d'ajouter des noms et acronymes.

## 1.2 Consignes

L'application doit être en mesure d'assurer l'interaction entre l'utilisateur et le programme au moyen d'une interface conviviale en langage C, permettant de :

— Activer/désactiver la prédiction de n = 3 mots les plus fréquemment/récemment utilisés avant d'avoir tapé toutes les lettres.
Ces prédictions proviennent du dictionnaire de prédiction fourni et peuvent être complétées par le dictionnaire français.

— Mettre à jour le dictionnaire de prédiction après chaque rédaction de mot.
Le dictionnaire de prédiction étant sensible à l'orthographe, on le complètera uniquement avec des mots correctement orthographiés et provenant du dictionnaire français.

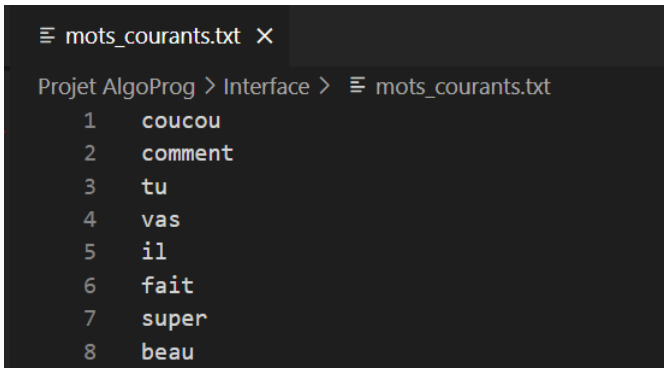
— Ajouter des noms ou acronymes fréquemment utilisés dans le dictionnaire de prédiction de mots pour les retrouver facilement.

— Modifier ou supprimer des mots dans le dictionnaire de prédiction.

— Gérer les tables de hachage, les afficher, etc.

## 1.3 Ressources

Un dictionnaire de prédiction initial est disponible sur la plateforme CAMPUS et nous avons également importé un dictionnaire de tous les mots de la langue française.



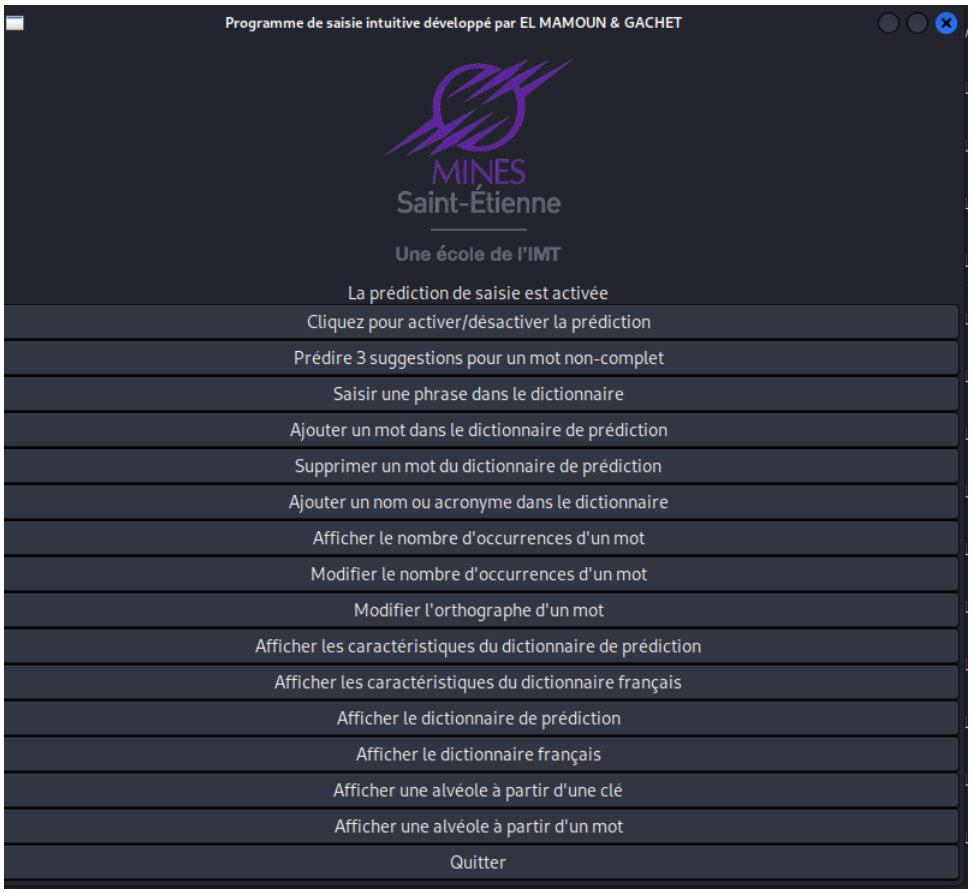
## 1.4 Interprétation

Afin de réaliser un programme convivial pour l'utilisateur, nous avons choisi de réaliser deux types d'interface :

— Une interface interactive qui s'affiche sur le terminal à l'éxecution du programme :

```
| Bienvenue ! Ce programme a été développé par Théo GACHET et Kawtar EL MAMOUN |
|  0  : Quitter l'application                                                  |
|  1  : Activer/Désactiver la prédiction de saisie (statut actuel : OFF)       |
|  2  : Prédire 3 suggestions pour un mot non-complet                          |
|  3  : Saisir une phrase dans le dictionnaire                                 |
|  4  : Ajouter un mot dans le dictionnaire de prédiction                      |
|  5  : Supprimer un mot du dictionnaire de prédiction                         |
|  6  : Ajouter un nom ou acronyme dans le dictionnaire                        |
|  7  : Afficher le nombre d'occurrences d'un mot                              |
|  8  : Modifier le nombre d'occurrences d'un mot                              |
|  9  : Modifier l'orthographe d'un mot                                        |
|  10 : Afficher les caractéristiques du dictionnaire de prédiction            |
|  11 : Afficher les caractéristiques du dictionnaire français                 |
|  12 : Afficher le dictionnaire de prédiction                                 |
|  13 : Afficher le dictionnaire français                                      |
|  14 : Afficher une alvéole à partir d'une clé                                |
|  15 : Afficher une alvéole à partir d'un mot                                 |
```

— Une interface graphique développée avec la librairie GTK3 :



# 2 Structures de données utilisées

## 2.1 Tables de hachage

Nous allons extraire les mots fournis par le fichier `mots_courants.txt` et les stocker dans une table de hachage afin de pouvoir y accéder rapidement et efficacement. En effet, les tables de hachage permettent un accès en $O(1)$ en moyenne, quel que soit le nombre de paires clé–valeur dans la table.

Une table de hachage est une structure de données qui permet de stocker des éléments de manière efficace et de les retrouver rapidement. Elle est utilisée dans de nombreuses applications, notamment les compilateurs, les bases de données et les algorithmes de recherche. Le fonctionnement d'une table de hachage en C peut être résumé en quelques étapes :

— Tout d'abord, une fonction de hachage est définie. Cette fonction prend en entrée une clé (un identifiant unique associé à chaque élément de la table de hachage) et renvoie un index dans la table de hachage. L'objectif est de distribuer les éléments de manière uniforme dans la table de hachage afin d'optimiser le temps d'accès aux données.

— Ensuite, une table de hachage est créée. Il s'agit d'un tableau de taille fixe, dans lequel chaque case contient soit un élément, soit une référence à une liste chaînée d'éléments ayant le même index de hachage.

— Lorsque des éléments sont ajoutés à la table de hachage, la fonction de hachage est utilisée pour déterminer leur index dans la table. Si la case correspondante est libre, l'élément y est directement stocké. Sinon, une nouvelle entrée est ajoutée à la liste chaînée correspondante.

— Pour retrouver un élément dans la table de hachage, la fonction de hachage est à nouveau utilisée pour déterminer son index. Si la case correspondante contient l'élément recherché, il est renvoyé. Sinon, la liste chaînée correspondante est parcourue pour chercher l'élément.

— Enfin, lorsque des éléments sont supprimés de la table de hachage, la case correspondante est vidée ou la référence à l'élément est retirée de la liste chaînée.

Il est important de noter que le choix d'une bonne fonction de hachage est crucial pour optimiser les performances d'une table de hachage en termes de temps d'accès aux données. De plus, il est souvent nécessaire de redimensionner la table de hachage en fonction du nombre d'éléments stockés afin de maintenir une distribution uniforme des données.

## 2.2 Listes chaînées

Une liste chaînée est une structure de données qui permet de stocker une séquence d'éléments de manière dynamique et efficace. Contrairement aux tableaux, les listes chaînées peuvent être redimensionnées facilement et sans avoir à déplacer tous les éléments en mémoire.

Une liste chaînée est constituée de nœuds (ou éléments) qui contiennent chacun une donnée et un pointeur vers le nœud suivant de la liste. Le premier nœud de la liste est appelé tête de liste, et le dernier nœud est généralement terminé par un pointeur `NULL`.

Il est important de noter que la structure des listes chaînées en C peut être utilisée pour implémenter d'autres structures de données, telles que les piles, les files, les arbres binaires, etc.

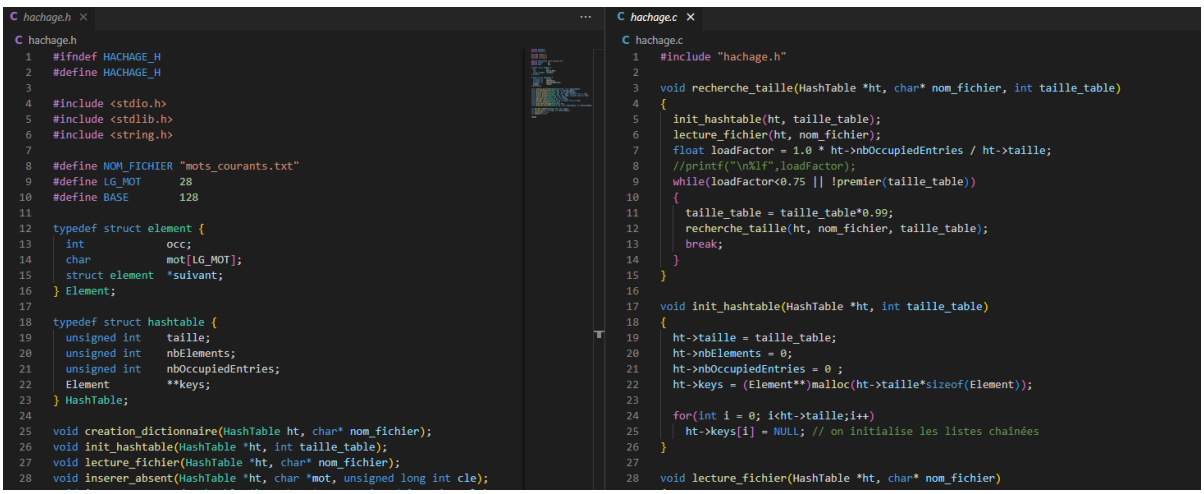
La structure d'un nœud pour nos listes chaînées sera définie comme suit :

```c
typedef struct element {
  int             occ;
  char            mot[LG_MOT];
  struct element  *suivant;
} Element;
```

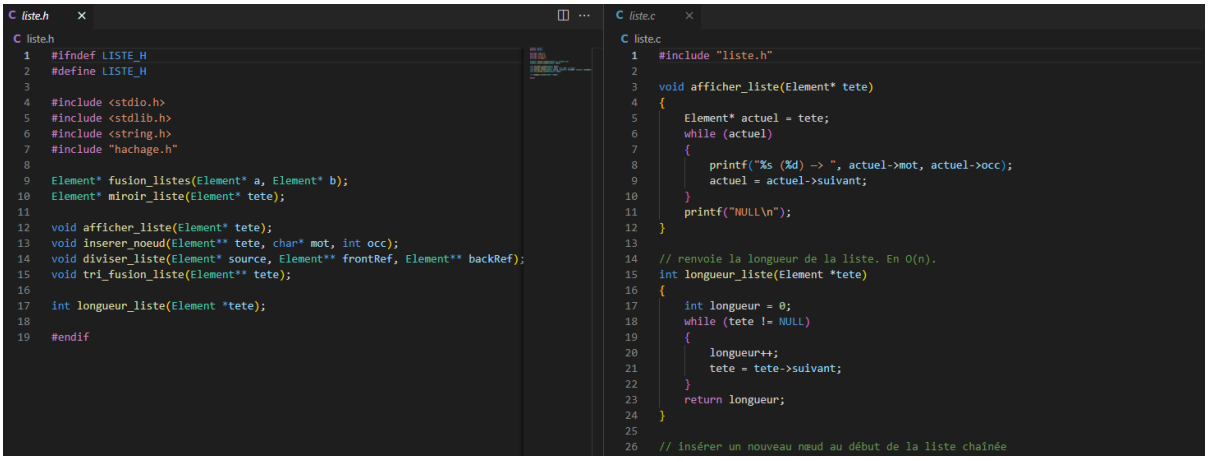
# 3 MODULARITÉ ET STRUCTURE DU CODE

La modularisation du projet est détaillée sur le fichier `README`. Notre programme est découpé en 3 modules ainsi qu'un fichier `main.c`.

Le premier module (`hachage.h` | `hachage.c`) concerne la strucure de la table de hachage utilisée pour ce programme, les prototypes, et les fonctions associées.



Le deuxième module, (`liste.h` | `liste.c`), détaille la structure de nos listes chaînées ainsi que les fonctions utiles à leur manipulation.



```c
// liste.h
#ifndef LISTE_H
#define LISTE_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hachage.h"

Element* fusion_listes(Element* a, Element* b);
Element* miroir_liste(Element* tete);

void afficher_liste(Element* tete);
void inserer_noeud(Element** tete, char* mot, int occ);
void diviser_liste(Element* source, Element** frontRef, Element** backRef);
void tri_fusion_liste(Element** tete);

int longueur_liste(Element *tete);

#endif
```

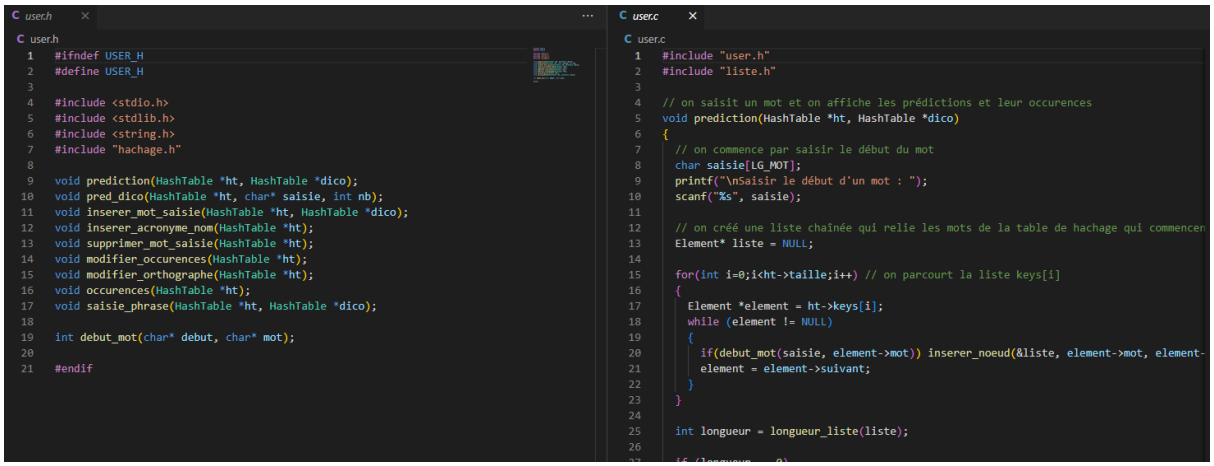
```c
// liste.c
#include "liste.h"

void afficher_liste(Element* tete)
{
    Element* actuel = tete;
    while (actuel)
    {
        printf("%s (%d) -> ", actuel->mot, actuel->occ);
        actuel = actuel->suivant;
    }
    printf("NULL\n");
}

// renvoie la longueur de la liste. En O(n).
int longueur_liste(Element *tete)
{
    int longueur = 0;
    while (tete != NULL)
    {
        longueur++;
        tete = tete->suivant;
    }
    return longueur;
}

// insérer un nouveau nœud au début de la liste chaînée
```

Enfin, le module (`user.h` | `user.c`) est composé des fonctions constituant le menu principal et consigne les fonctions auxquelles l'utilisateur fera directement appel lorsqu'il interagira avec l'interface.



```c
// user.h
#ifndef USER_H
#define USER_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hachage.h"

void prediction(HashTable *ht, HashTable *dico);
void pred_dico(HashTable *ht, char* saisie, int nb);
void inserer_mot_saisie(HashTable *ht, HashTable *dico);
void inserer_acronyme_nom(HashTable *ht);
void supprimer_mot_saisie(HashTable *ht);
void modifier_occurences(HashTable *ht);
void modifier_orthographe(HashTable *ht);
void occurences(HashTable *ht);
void saisie_phrase(HashTable *ht, HashTable *dico);

int debut_mot(char* debut, char* mot);

#endif
```

```c
// user.c
#include "user.h"
#include "liste.h"

// on saisit un mot et on affiche les prédictions et leur occurences
void prediction(HashTable *ht, HashTable *dico)
{
  // on commence par saisir le début du mot
  char saisie[LG_MOT];
  printf("\nSaisir le début d'un mot : ");
  scanf("%s", saisie);

  // on créé une liste chaînée qui relie les mots de la table de hachage qui commencen
  Element* liste = NULL;

  for(int i=0;i<ht->taille;i++) // on parcourt la liste keys[i]
  {
    Element *element = ht->keys[i];
    while (element != NULL)
    {
      if(debut_mot(saisie, element->mot)) inserer_noeud(&liste, element->mot, element-
      element = element->suivant;
    }
  }

  int longueur = longueur_liste(liste);
```

# 4 ALGORITHMES PROPOSÉS

NB : Vous trouverez dans le chapitre 7 (page 12) la liste des fonctions implémentées ainsi qu'un rapide descriptif pour chacune d'elles. De plus, leur fonctionnement est détaillé dans le code via les commentaires.

## 4.1 Gestion des tables de hachage

D'abord, la table de hachage est initialisée comme un tableau d'alvéoles vides (contenant des listes chaînées vides, i.e. `NULL`). Ensuite, `lecture_fichier()` permet d'aller chercher les mots que l'on souhaite importer, et les ajoute dans la table de hachage en fonction de leur clé calculée selon la fonction de hachage. En parallèle, la fonction `recherche_taille()` réitère le processus jusqu'à obtenir des paramètres de hachage qui permettent un facteur de charge optimal, i.e. proche de 0,75.

## 4.2 Prédiction de saisie

La fonction `prediction()` propose à l'utilisateur de saisir un mot et renvoie 3 suggestions pour le compléter. Les suggestions viennent du dictionnaire de prédiction construit à partir du fichier `mots_courants.txt` mais peut être complété par le dictionnaire français dans le cas où le fichier initial ne parvient pas à fournir assez de suggestions.

L'utilisateur saisit le début d'un mot et on créé une liste chaînée vide destinée à recevoir les suggestions correspondantes. On parcourt ensuite la table de hachage à la recherche des mots pour qui la fonction `debut_mot()` renvoie 1 (si les deux mots ont le même début) et on ajoute les mots concernés dans la liste.

La structure de noeud d'une liste chaînée contient l'orthographe du mot mais également son occurrence, i.e. son nombre d'apparition(s). Ainsi, diverses manipulations sur les listes chaînées (fusion, miroir, tri fusion) permettent d'ordonner la liste chaînée contenant toutes les prédictions possibles de manière à la trier selon l'ordre décroissant des occurrences dans la table de hachage. On peut ensuite, en sélectionnant les 3 premiers noeuds de la liste chaînée, en déduire les 3 suggestions ayant le plus grand nombre d'apparitions dans le dictionnaire de prédiction. La fonction `pred_dico()` complète ensuite ces suggestions dans le cas où il n'y en aurait pas assez.

## 4.3 Ajout et suppression d'un mot ou d'un acronyme

Pour ajouter un mot dans la liste de hachage via `inserer_mot()`, on commence par calculer sa clé avec `calcul_cle()`. Ensuite, la fonction `mot_dans_table()` nous indique si il est déjà présent ou non dans le dictionnaire de prédiction. Deux options sont alors possibles :

- Le mot existe déjà : dans ce cas, on incrémente simplement son nombre d'occurrences via `inserer_present()`
- Le mot n'existe pas : dans ce cas, on l'ajoute à l'alvéole qui correspond à sa clé via `inserer_absent()`

Pour supprimer un mot, on vérifie qu'il est présent dans le dictionnaire de prédiction et, si c'est le cas, on supprime son noeud dans l'alvéole qui le contient (accessible via la clé).

## 4.4 Saisie d'une phrase

Lorsque l'utilisateur saisit une phrase, il saisit des mots séparés par un espace. Le programme va donc segmenter la phrase en entrée sous la forme d'une matrice (i.e. d'un tableau de tableaux puisqu'une chaîne de caractères n'est autre qu'un tableau) de manière à pouvoir appliquer la fonction `inserer_mot()` à chacun des éléments de la phrase.

## 4.5 Modifications orthographiques

L'énoncé évoquait la "modification" d'un mot. Nous avons donc interprété cela comme le fait qu'un mot puisse changer d'orthographe. Concrètement, si `mot1` doit être modifié en `mot2`, alors on ajoute `mot2` dans la table de hachage, on initialise son nombre d'occurrences comme étant égal à celui de `mot1`, puis on supprime `mot1`.

## 4.6 Caractéristiques des tables de hachage

Pour des raisons diverses, souvent d'optimisation, l'utilisateur peut vouloir consulter les caractéristiques d'une table de hachage, que ce soit le dictionnaire de prédiction ou le dictionnaire français. Ainsi, on affiche le facteur de charge, le nombre de mots, le nombre total d'alvéoles ainsi que le nombre d'alvéoles effectivement occupées.

## 4.7 Affichage des alvéoles

L'utilisateur a la possibilité de visualiser une alvéole en saisissant un mot ou directement en entrant une clé. Ainsi, on affiche tous les mots ayant la même clé, ainsi que leur fréquence d'apparition sous le format :

[clé] -> $\text{mot}_1$ ($\text{occ}_1$) -> ... -> $\text{mot}_n$ ($\text{occ}_n$)

## 4.8 Affichage des tables de hachage

L'utilisateur a la possibilité de visualiser l'ensemble de la table de hachage. Cela revient en réalité à afficher toutes ses alvéoles sous le format :

[0] -> $\text{mot}_{0,1}$ ($\text{occ}_{0,1}$) -> ... -> $\text{mot}_{n,x}$ ($\text{occ}_{0,x}$)
[1] -> $\text{mot}_{1,1}$ ($\text{occ}_{1,1}$) -> ... -> $\text{mot}_{n,y}$ ($\text{occ}_{1,y}$)
...
[n] -> $\text{mot}_{n,1}$ ($\text{occ}_{n,1}$) -> ... -> $\text{mot}_{n,z}$ ($\text{occ}_{n,z}$)

## 4.9 Création d'une interface graphique via GTK3

GTK est une bibliothèque logicielle permettant de réaliser, entre autres, des interfaces graphiques comme celle que nous souhaitons développer.

En utilisant GTK, nous avons créé une interface interactive simple à utiliser pour l'utilisateur. Pour ce faire, nous avons implémenté plusieurs widgets à l'affichage, dont la fenêtre elle-même, le logo de l'Ecole, un bouton permettant à l'utilisateur d'activer ou de désactiver la saisie intuitive, associé à un texte dynamique pour afficher cette information. Enfin, nous avons implémenté un menu interactif permettant de lancer les fonctions préalablement définies selon le choix de l'utilisateur.

# 5 Limites du projet et solutions proposées

## 5.1 Recherche d'un mot

Pour prédire la suite d'un mot et afficher les 3 suggestions, il faut d'abord parcourir l'entièreté de la table de hachage et stocker les mots dans une liste chaînée. Bien que nous avons implémenté un algorithme de tri fusion pour trier efficacement cette dernière, le parcours de la table de hachage pourrait être encore amélioré.

## 5.2 Arbres de préfixes

Pour résoudre la question posée par le parcours de la table de hachage, on pourrait par exemple choisir d'implémenter un tas. En particulier, un arbre de préfixe semble être adapté à notre problème.

## 5.3 Réalisme de la prédiction de texte

Pour des algorithmes de saisie prédictive plus poussés tels que ceux de Google, d'autres fonctionnalités sont mises en place afin de compléter la saisie en temps réel.

Les moteurs de recherche utilisent l'historique et les habitudes afin de prédire la suite de sa saisie, chose qu'on a réussi à implémenter dans notre programme. Ceci n'est néanmois pas suffisant pour prédire correctement et précisement les attentes de l'utilisateur.

En effet, les moteurs de recherche analysent le contexte de la requête émise par celui-ci, et basés sur les recherches populaires associées à cette requête, plusieurs choix sont proposés. Des modèles de langage (LLM) permettent à la machine de faire cette analyse en se basant sur la fréquence d'apparition ou d'utilisation d'un très grand nombre de mots. Dans notre programme, une version simplifiée est proposée, qui consiste à incrémenter la fréquence d'apparition et le nombre d'occurrences d'un mot selon la saisie de l'utilisateur.

# 6 Analyse des résultats obtenus

## 6.1 Optimisation de la table de hachage

### 6.1.1 Facteur de charge (load factor)

Nous avons programmé la création du dictionnaire de prédiction sous la forme d'une table de hachage dynamique. En effet, la taille de la table influence la clé de hachage et celle-ci influence le remplissage des alvéoles. De plus, ce remplissage ainsi que le nombre total d'alvéoles participent au calcul du facteur de charge. Les paramètres étant co-dépendants, nous avons choisi de développer la fonction `recherche_taille()` qui permet d'obtenir une valeur optimale pour la taille de la table de hachage, et donc un facteur de charge proche de 0,75.

### 6.1.2 Fonction de hachage

Dans le code, on calcule la clé d'un mot comme suit :

```
49  int calcul_cle(char *string, int taille_table) // *string est un pointeur de string
50  {
51    unsigned long long int cle = 0 ;
52    int i = 0 ;
53
54    while ((*string) != '\0')   // \0 est le caractère de fin de chaîne
55    {
56      cle += cle % taille_table + ((*string) * (int) puissance(i)) % taille_table;
57      i++;
58      string++;
59    }
60    return cle % taille_table ;
61  }
```

Ainsi, la fonction de hachage permet d'obtenir la clé de `mot` de la manière suivante :

On initialise nb_lettres = n et cle(0) = 0 et $\forall i \in [1, n], i \in N$ :

cle(i) = cle(i-1) + cle(i-1) % taille_table + (mot[i] $* 128^i$) % taille_table

Et finalement : cle_finale = cle(n) % taille_table

## 6.2 Utilisation du programme sur le terminal

Pour lancer le code avec l'interface du terminal, exécutez les commandes suivantes :

```
make all_without_interface
./all_without_interface
```

```
| Bienvenue ! Ce programme a été développé par Théo GACHET et Kawtar EL MAMOUN |

|  0  : Quitter l'application                                                  |
|  1  : Activer/Désactiver la prédiction de saisie (statut actuel : OFF)       |
|  2  : Prédire 3 suggestions pour un mot non-complet                          |
|  3  : Saisir une phrase dans le dictionnaire                                 |
|  4  : Ajouter un mot dans le dictionnaire de prédiction                      |
|  5  : Supprimer un mot du dictionnaire de prédiction                         |
|  6  : Ajouter un nom ou acronyme dans le dictionnaire                        |
|  7  : Afficher le nombre d'occurrences d'un mot                              |
|  8  : Modifier le nombre d'occurrences d'un mot                              |
|  9  : Modifier l'orthographe d'un mot                                        |
|  10 : Afficher les caractéristiques du dictionnaire de prédiction            |
|  11 : Afficher les caractéristiques du dictionnaire français                 |
|  12 : Afficher le dictionnaire de prédiction                                 |
|  13 : Afficher le dictionnaire français                                      |
|  14 : Afficher une alvéole à partir d'une clé                                |
|  15 : Afficher une alvéole à partir d'un mot                                 |
```

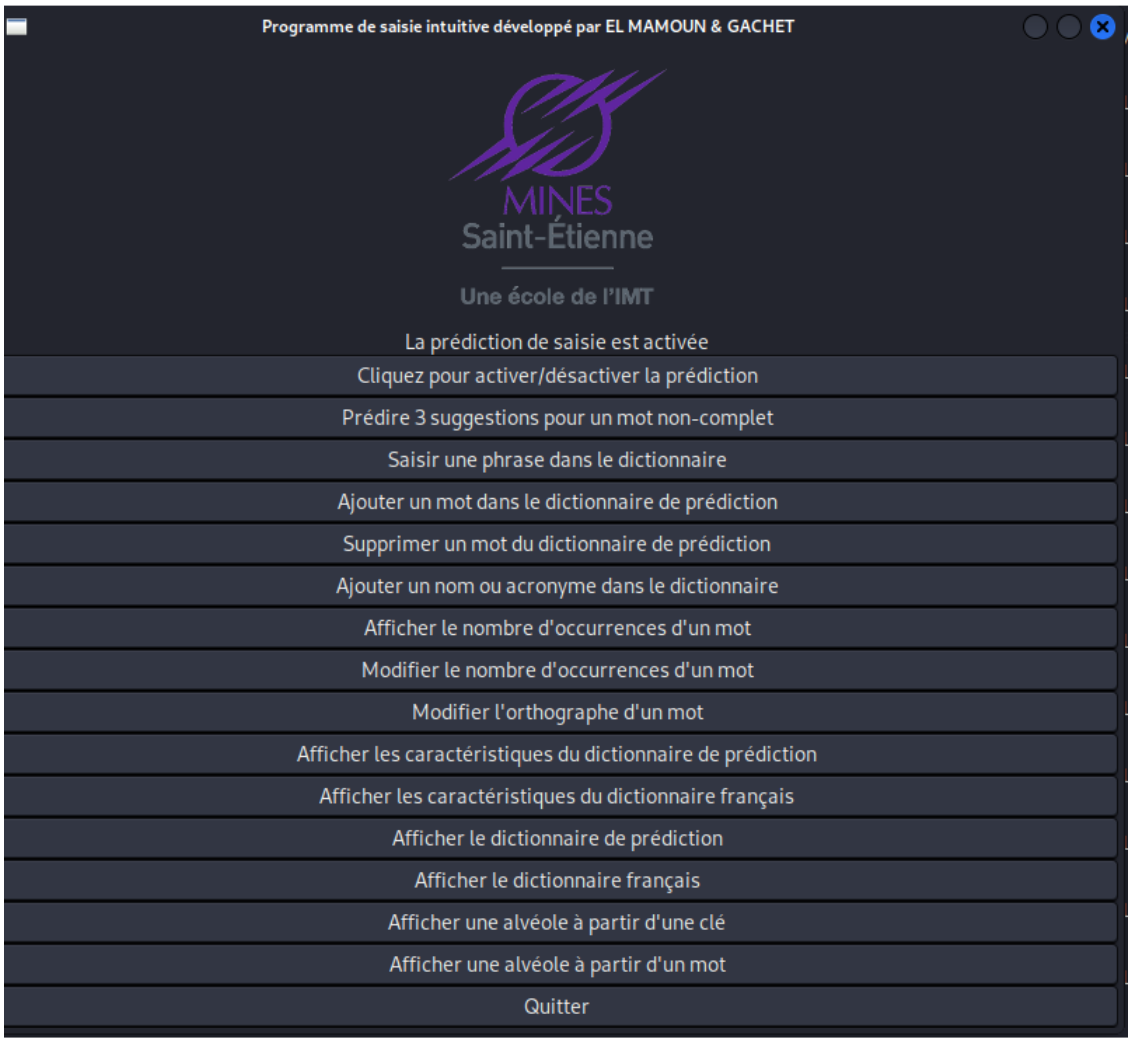
## 6.3 Utilisation du programme sur l'interface GTK3

Pour lancer le code avec l'interface graphique développée via GTK, il vous faudra installer la librairie `gtk` :

```
sudo apt-get update
apt-get install libgtk-3-dev
```

Une fois la librairie installée, vous pouvez afficher l'interface interactive en exécutant les commandes suivantes :

```
make all_w_interface
./all_w_interface
```




## 6.4 Toutes nos fonctionnalités sont... fonctionnelles !

Pas besoin d'en dire plus.

# 7 DÉTAIL DES FONCTIONS PROGRAMMÉES

## 7.1 Pour la table de hachage

```
void init_hashtable(HashTable *ht, int taille_table);
```
Alloue la mémoire et initialise la table de hachage comme un tableau de taille prédéfinie que l'on remplit de listes chaînées vides (NULL).

```
int calcul_cle(char *string, int taille_table);
```
Calcule la clé d'un mot.

```
void recherche_taille(HashTable *ht, char* nom_fichier, int taille_table);
```
Fonction d'optimisation qui cherche la taille optimale d'une table de hachage afin de répondre aux exigences concernant le facteur de charge.

```
int mot_dans_table(HashTable *ht, char *mot);
```
Renvoie 1 si un mot est dans la table, 0 sinon.

```
void inserer_absent(HashTable *ht, char *mot, unsigned long int cle);
```
Insert la première apparition d'un mot dans la table de hachage.

```
void inserer_present(HashTable *ht, char *mot, unsigned long int cle);
```
Augmente l'occurrence d'un mot dans la table de hachage.

```
void inserer_mot(HashTable *ht, char *mot);
```
Insert un mot dans la table de hachage.

```
void supprimer_mot(HashTable* ht, char* mot);
```
Supprime un mot de la table de hachage.

```
void lecture_fichier(HashTable *ht, char* nom_fichier);
```
Lit le fichier en argument et insert chaque mot dans la table de hachage.

```
void afficher_alveole(HashTable* ht, unsigned long int cle);
```
Affiche une alvéole de la table de hachage selon la clé fournie en argument.

```
void afficher_table(HashTable* ht);
```
Affiche la table de hachage.

```
void param_table_hachage(HashTable* ht);
```
Affiche les caractéristiques d'une table de hachage.

```
int puissance(int i);
```
Renvoie la base de calcul à la puissance i (pour la fonction de hachage).

```
int premier(int n);
```
Renvoie 1 si n est premier, 0 sinon.

## 7.2 Pour les listes chaînées

```
void diviser_liste(Element* source, Element** frontRef, Element** backRef);
```
Divise une liste chaînée en deux parties à peu près égales et met les nœuds de chaque partie dans des pointeurs de pointeurs différents.

```
Element* fusion_listes(Element* a, Element* b);
```
Prend deux listes triées par ordre croissant et fusionne leurs nœuds pour former une plus grande liste triée (et retournée).

```
Element* miroir_liste(Element* tete);
```
Renvoie le miroir d'une liste chaînée. En O(n).

```
void tri_fusion_liste(Element** tete);
```
Trie une liste chaînée à l'aide du tri par fusion.

```
void inserer_noeud(Element** tete, char* mot, int occ);
```
Insert un nouveau nœud au début de la liste chaînée.

```
void afficher_liste(Element* tete);
```
Affiche une liste chaînée.

```
int longueur_liste(Element *tete);
```
Renvoie la longueur de la liste. En O(n).

## 7.3 Pour l'utilisateur

```
int debut_mot(char* debut, char* mot);
```
Renvoie 1 si « mot » commence par « debut », et 0 sinon.

```
void prediction(HashTable *ht, HashTable *dico);
```
L'utilisateur saisit un mot et les 3 prédictions ayant les plus grandes occurrences dans le dictionnaire de prédiction sont proposées. Si le nombre de suggestions du dictionnaire de prédiction est inférieur à 3, alors on complète avec des suggestions venant du dictionnaire français.

```
void pred_dico(HashTable *ht, char* saisie, int nb);
```
Affiche nb suggestions commençant par « saisie » et provenant du dictionnaire français.

```
void inserer_mot_saisie(HashTable *ht, HashTable *dico);
```
L'utilisateur saisit un mot qui est ajouté à la table de hachage. Si le mot est déjà dans la table de hachage, alors sa valeur « occ » est incrémentée. L'ajout d'un mot mal orthographié (absent du dictionnaire français) est impossible.

```
void inserer_acronyme_nom(HashTable *ht);
```
L'utilisateur saisit un mot qui est ajouté à la table de hachage. Si le mot est déjà dans

la table de hachage, alors sa valeur « occ » est incrémentée. L'ajout d'un mot mal orthographié (absent du dictionnaire français) est possible.

```
void saisie_phrase(HashTable *ht, HashTable *dico);
```
L'utilisateur saisit une phrase dont les mots, séparés par des espaces, sont insérés dans le dictionnaire de prédiction.

```
void supprimer_mot_saisie(HashTable *ht);
```
L'utilisateur saisit un mot et celui-ci est supprimé de la table de hachage.

```
void occurences(HashTable *ht);
```
L'utilisateur saisit un mot et la fonction affiche son nombre d'occurrences.

```
void modifier_occurences(HashTable *ht);
```
L'utilisateur saisit le mot dont il veut modifier le nombre d'occurrences ainsi que la nouvelle valeur, les modifications sont ensuite effectuées.

```
void modifier_orthographe(HashTable *ht);
```
L'utilisateur saisit le mot1 dont il veut changer l'orthographe, représentée par mot2. Ainsi, mot2 est ajouté à la table de hachage et son nombre d'occurrences est celui de mot1. Puis mot1 est supprimé de la table de hachage. Le mot1 a en quelque sorte transmis ses occurrences à mot2 avant de disparaître.

## 7.4 Pour l'interface graphique

Les fonctions suivantes ont permis le développement de l'interface graphique interactive, leur nom étant assez explicite et leur rôle étant principalement de structurer le design de l'interface, il est inutile de s'attarder sur leur fonctionnement :

```
void option_quitter(GtkButton *button, gpointer data);
void option_activer(GtkButton *button, gpointer data);
void option_predire(GtkButton *button, gpointer data);
void option_saisir_phrase(GtkButton *button, gpointer data);
void option_ajouter_mot(GtkButton *button, gpointer data);
void option_supprimer_mot(GtkButton *button, gpointer data);
void option_ajouter_acronyme(GtkButton *button, gpointer data);
void option_afficher_occurrences(GtkButton *button, gpointer data);
void option_modifier_occurences(GtkButton *button, gpointer data);
void option_modifier_orthographe(GtkButton *button, gpointer data);
void option_caract_dico_pred(GtkButton *button, gpointer data);
void option_caract_dico_fr(GtkButton *button, gpointer data);
void option_afficher_dico_pred(GtkButton *button, gpointer data);
void option_afficher_dico_fr(GtkButton * button, gpointer data);
void option_alveole_par_cle(GtkButton *button, gpointer data);
void option_alveole_par_mot(GtkButton *button, gpointer data);
```