Projet: Anagrammes
Algorithmique et structures de données
Licence 2 Informatique

Julien Bernard

Table des matières

Le but de ce projet est de réaliser un outil interactif en mode texte permettant de trouver des anagrammes à partir d'un ensemble de lettres. Les mots à trouver feront partie d'un dictionnaire qu'on lira à partir d'un fichier fourni sur MOODLE (`dictionnaire.txt`). Les mots ne contiennent que des lettres minuscules, sans signe diacritique (accents, cédille, etc). Il est conseillé de faire la dernière partie en même temps que les autres de manière à pouvoir vérifier votre programme au fur et à mesure.

Partie 1 : Fonctions sur les chaînes de caractères

Dans cette première partie, vous devez écrire un ensemble de fonctions relatives aux chaînes de caractères qui vous seront utiles pour la suite.

Question 1.1 Écrire une fonction qui dit si deux chaînes sont des anagrammes l'une de l'autre. On pourra, par exemple, compter le nombre de chaque lettre dans la première chaîne puis dans la deuxième et vérifier que les comptes sont égaux.

```
bool string_are_anagrams(const char *str1, const char *str2);
```

Question 1.2 Écrire une fonction qui renvoie une copie de la chaîne passée en paramètre, allouée grâce à `malloc(3)`.

```
char *string_duplicate(const char *str);
```

Question 1.3 Écrire une fonction qui trie les lettres de la chaîne passée en paramètre dans l'ordre alphabétique. Vous utiliserez un tri par insertion pour cette fonction.

```
void string_sort_letters(char *str);
```

Partie 2 : Tableau de mots

Dans cette partie, nous allons écrire un ensemble de fonctions pour manipuler un tableau de mots. Le tableau sera responsable de la mémoire utilisée pour stocker les mots. La structure utilisée pour le tableau de mots sera la suivante :

```
struct word_array {
  char **data;
  size_t size;
  size_t capacity;
};
```

Question 2.1 Écrire une fonction qui initialise un tableau de mots.

```
void word_array_create(struct word_array *self);
```

Question 2.2 Écrire une fonction qui détruit un tableau de mots. On prendra garde à libérer la mémoire utilisée pour les mots.

```
void word_array_destroy(struct word_array *self);
```

Question 2.3 Écrire une fonction qui ajoute un mot passé en paramètre au tableau. On fera une copie du mot passé en paramètre pour l'insérer dans le tableau.

```
void word_array_add(struct word_array *self, const char *word);
```

Question 2.4 Écrire une fonction qui cherche des anagrammes d'un mot dans un tableau de mots. Les anagrammes trouvées seront placées dans un autre tableau de mots.

```
void word_array_search_anagrams(const struct word_array *self,
  const char *word, struct word_array *result);
```

Question 2.5 Écrire une fonction qui trie les mots d'un tableau de mots. On utilisera un algorithme optimal pour effectuer le tri.

```
void word_array_sort(struct word_array *self);
```

Question 2.6 Écrire une fonction qui affiche tous les mots d'un tableau de mots (un par ligne).

```
void word_array_print(const struct word_array *self);
```

Partie 3 : Dictionnaire

Dans cette partie, nous allons écrire un ensemble de fonctions pour stocker les mots dans un dictionnaire, sous forme d'une table de hachage.

Une table de hachage est une structure de données qui permet une implémentation d'un tableau associatif, c'est-à-dire un tableau dans lequel les éléments sont représentés par une clef. Dans notre cas, la clef sera le mot dont les lettres sont triées.

Concrètement, une table de hachage est un tableau de k listes chaînées. Pour insérer un élément dans l'ensemble, on utilise une fonction de hachage qui prend un élément et renvoie un entier h, l'élément est alors inséré dans la liste chaînée qui se trouve à l'indice ($h \mod k$).

Si deux éléments ont le même indice, ils se retrouvent dans la même liste chaînée, on parle alors de *collision*. On essaie de choisir une fonction de hachage qui permet d'éviter les collisions. On appelle *facteur de compression* le rapport $\frac{n}{k}$, c'est-à-dire le nombre d'éléments de la table divisé par le nombre de cases du tableau.

Quand le facteur de compression dépasse une certaine valeur pour laquelle le risque de collision devient important (0.5 par exemple), alors on effectue un *rehash*, c'est-à-dire qu'on va doubler la taille du tableau et recalculer tous les indices des éléments déjà présents.

Dans notre cas, la table de hachage ne sera pas responsable de la mémoire utilisée pour stocker les mots (ils seront déjà dans un tableau de mots), elle manipulera donc uniquement des pointeurs.

La figure 1 montre un schéma d'une table de hachage où des collisions ont eu lieu pour les indices 1 et 4 mais pas pour l'indice 0.

—

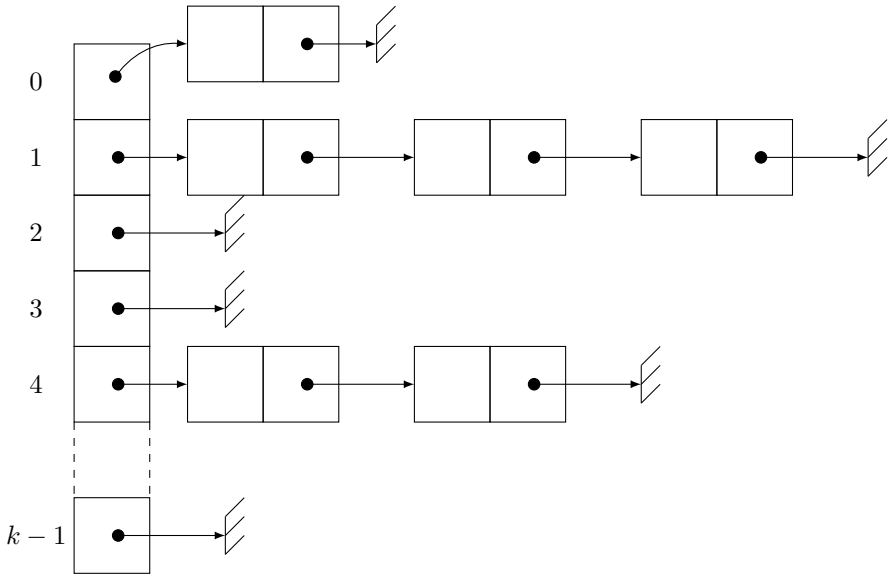

FIGURE 1 – Schéma d'une table de hachage

On s'intéresse tout d'abord à la structure de liste chaînée (qu'on appelle généralement *bucket* dans le cadre des tables de hachage).

```
struct word_dict_bucket {
  const char *word;
  struct word_dict_bucket *next;
};
```

Question 3.1 Écrire une fonction qui ajoute un élément à une liste chaînée éventuellement vide et renvoie la nouvelle liste chaînée.

```
struct word_dict_bucket *word_dict_bucket_add(
  struct word_dict_bucket *self, const char *word);
```

Question 3.2 Écrire une fonction qui détruit une liste chaînée éventuellement vide.

```
void word_dict_bucket_destroy(struct word_dict_bucket *self);
```

—

La table de hachage est représentée par la structure suivante où `count` désigne le nombre d'éléments dans la table et `size` désigne la taille du tableau `buckets`.

```
struct word_dict {
  struct word_dict_bucket **buckets;
  size_t count;
  size_t size;
};
```

Question 3.3 Écrire une fonction qui crée un dictionnaire vide.

```
void word_dict_create(struct word_dict *self);
```

Question 3.4 Écrire une fonction qui détruit un dictionnaire. On détruira toutes les listes chaînées mais pas les mots.

```
void word_dict_destroy(struct word_dict *self);
```

Question 3.5 Écrire une fonction qui calcule un hash FNV-1a [1] 64 bits. On fera une copie de la clef puis on triera les lettres avant de calculer le hash sur cette chaîne avec lettres triées.

```
size_t fnv_hash(const char *key);
```

Question 3.6 Écrire une fonction qui effectue un rehash.

```
void word_dict_rehash(struct word_dict *self);
```

Question 3.7 Écrire une fonction qui ajoute une entrée dans le dictionnaire. On effectuera un rehash si nécessaire.

```
void word_dict_add(struct word_dict *self, const char *word);
```

Question 3.8 Écrire une fonction qui remplit un dictionnaire avec les mots contenus dans un tableau de mots.

```
void word_dict_fill_with_array(struct word_dict *self,
  const struct word_array *array);
```

Question 3.9 Écrire une fonction qui cherche des anagrammes d'un mot dans un dictionnaire. Les anagrammes trouvées seront placées dans un tableau de mots.

```
void word_dict_search_anagrams(const struct word_dict *self,
  const char *word, struct word_array *result);
```

Partie 4 : Utilisation d'un caractère joker

Dans cette partie, nous allons ajouter une difficulté : la possibilité pour l'utilisateur de mettre un caractère joker dans les lettres qu'il soumet pour trouver une anagramme. Le caractère joker pourra être remplacé par n'importe quel autre caractère. Le caractère joker sera le caractère `'*'`. On limitera le nombre de jokers à 4 par recherche.

Pour chercher les jokers, on utilisera la structure suivante où `count` est le nombre de jokers et `index` est un tableau contenant les indices dans la chaîne de caractères où se trouvent les jokers.

1. `https://en.wikipedia.org/wiki/Fowler-Noll-Vo_hash_function`

```
#define WILDCARDS_MAX 4

struct wildcard {
  size_t count;
  size_t index[WILDCARDS_MAX];
};
```

Question 4.1 Écrire une fonction qui initialise la structure des jokers.

```
void wildcard_create(struct wildcard *self)
```

Question 4.2 Écrire une fonction qui recherche les jokers dans une chaîne de caractères.

```
void wildcard_search(struct wildcard *self, const char *word)
```

Question 4.3 Écrire une fonction qui recherche les anagrammes dans un tableau de mots en prenant en compte les jokers.

```
void word_array_search_anagrams_wildcard(
  const struct word_array *self, const char *word,
  struct word_array *result);
```

Question 4.4 Écrire une fonction qui recherche les anagrammes dans un dictionnaire en prenant en compte les jokers.

```
void word_dict_search_anagrams_wildcard(
  const struct word_dict *self, const char *word,
  struct word_array *result);
```

Partie 5 : Application interactive

Dans cette partie, nous allons créer une application interactive permettant d'utiliser toutes les fonctions que nous venons de coder. Pour cela, nous allons utiliser la fonction `fgets(3)` qui permet de charger un fichier ligne par ligne. Cette fonction copie chaque ligne du fichier dans un buffer, il est donc nécessaire de supprimer le caractère de changement de ligne avant de traiter le buffer.

Question 5.1 Écrire une fonction qui supprime le caractère de changement de ligne final et le remplace par `\0`.

```
void clean_newline(char *buf, size_t size)
```

—

La boucle de saisie peut alors s'écrire de la manière suivante :

```
char buf[BUFSIZE];

for (;;) {
  printf("> ");
  fgets(buf, BUFSIZE, stdin);
  clean_newline(buf, BUFSIZE);
}
```

Question 5.2 Avant la boucle de saisie, lire le fichier `dictionnaire.txt` pour initialiser un tableau de mots, puis créer un dictionnaire à partir de ce tableau de mots. Vérifier que vous avez lu le bon nombre de mots. On pourra utiliser la fonction `word_array_read_file` dont l'implémentation vous est donné et qui permet de lire un fichier contenant des mots et de les stocker dans un tableau de mots.

Question 5.3 Dans la boucle de saisie, demander un mot (ou un ensemble de lettres) et afficher toutes ses anagrammes dans l'ordre alphabétique. On prendra en compte les jokers. On indiquera, à la fin, le nombre d'anagrammes trouvées et le temps mis pour les trouver en microsecondes (indice : `gettimeofday(3)`). On réalisera cette opération deux fois, une fois avec le tableau de mots et une fois avec le dictionnaire, de manière à vérifier que les deux donnent les mêmes résultats et comparer leur temps d'exécution.

Question 5.4 Si le mot saisi est vide, quitter la boucle. Après la boucle de saisie, libérer correctement la mémoire.

Question 5.5 Vérifier que vos algorithmes fonctionnent correctement. Par exemple, «crane» doit vous renvoyer dix anagrammes et «argent» doit vous renvoyer huit anagrammes. Avec un joker, «ecran*» doit renvoyer 52 anagrammes.

Considérations générales et évaluation

Le projet est à faire en binôme. Vous disposez sur MOODLE d'une archive avec un squelette de code que vous aurez à remplir. Vous pouvez construire l'exécutable `anagrammes-query` en tapant la commande `make`.

```
tar zxvf anagrammes.tar.gz
cd anagrammes
make
./anagrammes-query
```

Pour sortir du programme, vous pouvez utiliser la combinaison de touches `CTRL+C`.

Une fois votre projet terminé, vous aurez à remplir le fichier `README` avec les noms des deux binômes, puis à créer une archive à déposer sur MOODLE avant la date indiquée (aucun retard ne sera autorisé).

```
cd ..
tar zcvf anagrammes-result.tar.gz anagrammes
```

La note du projet tiendra compte des points suivants (liste non-exhaustive) :
— la correction des algorithmes proposés ;
— la complexité optimale des algorithmes proposés ;
— l'absence de fuites mémoire ;
— les commentaires dans le code source ;
— la propreté du code.