Big Data | TP 2

> Présenté par Omar ALLOUCH

Point de départ

On va se servir du TP 1 pour prendre la fonction de hachage et le code responsable d'ouvirir les fichiers, lire les mots et appliquer la fonction de hachage.

```
# include <stdio.h>
# include <string.h>
# include <stdlib.h>

# define TABLE_SIZE 786307 // From the previous exercise

int hash_function(char *word, int table_size) {
  int hash = 4291;
  for (int i = 0; i < strlen(word); i++) {
    hash ^= ((hash << 5) + (hash >> 2) + word[i]) % table_size;
  }
  return hash % table_size;
}

int main(int argc, char *argv[]) {
  if (argc != 3) {
    printf("Usage: %s file1 file2\n", argv[0]);
    exit(1);
  }

  FILE *file1 = fopen(argv[1], "r");
  if (file1 == NULL) {
```

```
    fprintf(stderr, "Error: Cannot open file %s\n", argv[1]);
    exit(1);
  }

  FILE *file2 = fopen(argv[2], "r");
  if (file2 == NULL) {
    fprintf(stderr, "Error: Cannot open file %s\n", argv[2]);
    fclose(file1);
    exit(1);
  }

  char word[100];
  while (fscanf(file1, "%s", word) != EOF) {
    ...
  }

  // Close files
  fclose(file1);
  fclose(file2);

  return 0;
}
```

Plan d’actions

Dans la suite, on va présenter 2 méthodes de résolution de collisions qui vont nous aider à bien garder en mémoire les mots du premier fichier pour qu’on puisse savoir les intersections avec le deuxième. Ces 2 méthodes sont: **Adressage ouvert**, et **adressage fermé**. Pour chacune de ces méthodes, on présente les fonctions de recherche et d’insertion et leur utilisation.

Adressage ouvert

Dans une table de hachage à adressage ouvert tous les éléments sont gardés dans la table. On ne dispose pas d’une seule fonction de hachage, mais d’une famille de fonctions de hachage:

$$h(x,i) = (h'(x) + i) \; mod \; M$$

Ca se traduit par le suivant :

Fonction de recherche :

```
// Search for a word in the hash table using open addressing
int open_addressing_search(char *table[], char *word, int table_
  int hash = hash_function(word, table_size);
  int initial_hash = hash;

  // Linear probing
  while (table[hash] != NULL) {
    // Check if the word is found
    if (strcmp(table[hash], word) == 0) {
      return 1; // Word found
    }
    hash = (hash + 1) % table_size;

    // Check if we have looped around to the starting point
    if (hash == initial_hash) {
      break; // Word not found, break the loop
    }
  }
  return 0; // Word not found
}
```

Fonction d’insertion :

```
// Insert a word into the hash table using open addressing
void open_addressing_insert(char *table[], char *word, int table
  int hash = hash_function(word, table_size);
  // Linear probing
  while (table[hash] != NULL) {
    if (strcmp(table[hash], word) == 0) {
```

```
      return; // Word already exists, no need to insert again
    }
    hash = (hash + 1) % table_size;
  }
  table[hash] = strdup(word);
}
```

Utilisation :

```
int main(int argc, char *argv[]) {
    ...
  // Open addressing ----------------------------------------
  printf("Open addressing:\n");
  // Data structure for open addressing hash table
  char *open_addressing_table[TABLE_SIZE] = {0}; // Initialize a

  // Insert words from file1 into the hash table
  // Use open addressing to handle collisions
  char word[100];
  while (fscanf(file1, "%s", word) != EOF) {
    open_addressing_insert(open_addressing_table, word, TABLE_SI
  }

  // Search for intersections
  int nb_intersections = 0;
  char *intersections[TABLE_SIZE] = {0};
  while (fscanf(file2, "%s", word) != EOF) {
    if (open_addressing_search(open_addressing_table, word, TABL
      printf("%s\n", word);
      nb_intersections++;
    }
  }
  printf("Total intersections: %d\n", nb_intersections);
  // ------------------------------------------------------------
```

```
  ...
  // Free memory allocated for hash table entries
  for (int i = 0; i < TABLE_SIZE; i++) {
    free(open_addressing_table[i]);
  }
  ...
}
```

Nombre d'intersections obtenus: **15760**.

Adressage fermé

Dans une table de hachage à adressage fermé une alvéole contient non pas un élément mais un
pointeur vers une liste chainée.

Ca se traduit par le suivant :

Structures nécessaires :

```
typedef struct Node {
  char *word;
  struct Node *next;
} Node;

typedef struct HashTable {
  Node *words[TABLE_SIZE];
} HashTable;
```

Fonction de recherche :

```
// Search for a word in the hash table using chaining
int chaining_search(HashTable *table, char *word, int table_size
```

```
  int hash = hash_function(word, table_size);
  Node *current = table->words[hash];
  while (current != NULL) {
    if (strcmp(current->word, word) == 0) {
      return 1; // Word found
    }
    current = current->next;
  }
  return 0; // Word not found
}
```

Fonction d’insertion :

```
// Insert a word into the hash table using chaining
void chaining_insert(HashTable *table, char *word, int table_siz
  int hash = hash_function(word, table_size);
  Node *new_node = malloc(sizeof(Node));
  new_node->word = strdup(word);
  new_node->next = NULL;
  if (table->words[hash] == NULL) {
    table->words[hash] = new_node;
  } else {
    new_node->next = table->words[hash];
    table->words[hash] = new_node;
  }
}
```

Utilisation :

```
int main(int argc, char *argv[]) {
    ...
  // Chaining ------------------------------------------------
  printf("Chaining:\n");
```

```
// Data structure for chaining hash table
HashTable chaining_table;
for (int i = 0; i < TABLE_SIZE; i++) {
  chaining_table.words[i] = NULL;
}

// Insert words from file1 into the hash table
// Use chaining to handle collisions
char word[100];
while (fscanf(file1, "%s", word) != EOF) {
  chaining_insert(&chaining_table, word, TABLE_SIZE);
}

// Search for intersections
int nb_intersections = 0;
char *intersections[TABLE_SIZE] = {0};
while (fscanf(file2, "%s", word) != EOF) {
  if (chaining_search(&chaining_table, word, TABLE_SIZE)) {
    printf("%s\n", word);
    nb_intersections++;
  }
}
printf("Total intersections: %d\n", nb_intersections);
// ------------------------------------------------------------
...
// Free memory allocated for hash table entries
for (int i = 0; i < TABLE_SIZE; i++) {
  Node *current = chaining_table.words[i];
  while (current != NULL) {
    Node *temp = current;
    current = current->next;
    free(temp->word);
    free(temp);
  }
}
```

```
    ...
}
```

Nombre d’intersections obtenus: **15760**.