Projet LZW 2021 - 2022

Compression de fichier

Université de Strasbourg - UFR Mathématique & Informatique - L2S4A - Structure de données et algorithmes 2

Modalités

Le TP est à rendre au plus tard le 19 décembre 2021 à 23h59 via la plateforme Moodle. Il est à réaliser en binôme exclusivement, durant le temps libre.

Travail à rendre

Il vous est demandé un document synthétique tenant sur au maximum trois feuilles recto verso, soit six pages, où vous ferez figurer les réponses aux questions ainsi que tout élément que vous jugerez bon de préciser. Vous déposerez sur Moodle, dans une archive au format *.tar.gz* :

- le document synthétique au format électronique *.pdf* ;
- les sources dûment commentées de votre réalisation en langage C, ce qui inclut notamment un fichier *Makefile* mais pas les fichiers objets *.o*. **Un code qui ne compile pas ne sera pas accepté.**

L'ensemble des programmes ayant servi à l'élaboration du projet doit être dûment commenté. Essayez d'attacher une grande importance à la lisibilité du code : il est indispensable de respecter les conventions d'écriture (*coding style*) du langage C et de bien présenter le code (indentation, saut de ligne, etc.) ; les identificateurs (nom de types, variables, fonctions, etc.) doivent avoir des noms cohérents documentant leur rôle ; indiquez les pré-conditions, post-conditions, paramètres, la sortie et un descriptif de chaque fonction. Un commentaire se doit d'être sobre, concis et aussi précis que possible. Pensez à utiliser des outils comme *gdb* ou *valgrind* pour faciliter votre développement et optimiser votre programme.

Barème indicatif

- Code (programmation et algorithmique) : 12 points
  - Structures de données : 7 points
  - Mise en application : 5 points
- Code (forme, commentaires) : 2 points
- Rapport : 6 points
- Bonus/Malus rendu des TP : de +2 à -2 points

Problème

On se propose d'étudier dans ce projet une méthode de compression, et de décompression, d'un fichier au format texte en utilisant plusieurs structures de données différentes.

L'objectif principal de cette étude sera notamment de juger des impacts relatifs au choix des structures de données dans une application.

Approche proposée

La méthode à analyser sera l'algorithme de compression *LZW* (*Lempel-Ziv-Welch*, du nom de ses auteurs). En plus du bon fonctionnement de la compression des fichiers votre analyse de cette algorithme s'axera sur la structure de données qu'il utilise , à savoir un dictionnaire. Il vous sera demandé de trouver une méthode de comparaison entre chacune des structures proposées. Voici quelques pistes à explorer, mais celles-ci ne sont bien évidement pas exhaustives :

- Complexité en temps
- Complexité en espace
- Facilité de compréhension et temps d'implémentation de la structure

Dans l'optique de délimiter et simplifier le projet nous considérerons uniquement les fichiers textes rédigés au format *ASCII*, donc sans accents et correspondant au type *char* en *C*. Une autre alternative serait d'utiliser le format *Extended ASCII* correspondant au type *unsigned char* en *C*.

Algorithme

Vous trouverez ci-dessous le pseudo-code, issu de *GeeksforGeeks*, permettant l'implémentation de l'algorithme *LZW*.

Fonction de compression

```
Initialize table with single character strings
P = first input character
WHILE not end of input stream
  C = next input character
  IF P + C is in the string table
    P = P + C
  ELSE
    output the code for P
    add P + C to the string table
    P = C
  END IF
END WHILE
output code for P
```

Fonction de décompression

```
Initialize table with single character strings
OLD = first input code
output translation of OLD
WHILE not end of input stream
  NEW = next input code
  IF NEW is not in the string table
    S = translation of OLD
    S = S + C
  ELSE
    S = translation of NEW
  END IF
  output S
  C = first character of S
  OLD + C to the string table
  OLD = NEW
END WHILE
```

Encodage des fichiers binaires

L'écriture des fichiers compressés par l'algorithme *LZW* se fera sous forme binaire. Les fichiers sous un tel format finiront, par convention, avec l'extension *.lzw*. L'ouverture du fichier en binaire se fera grâce aux options *'wb'* et *'rb'* de la fonction *fopen*.

Les *codes* (*output code* dans l'algorithme de compression et *input code* dans la décompression) contiennent les données à écrire au format binaires. Pour faciliter le bon développement du projet, les *codes* seront représentés au format hexadécimal dans les dictionnaires. Ils seront écrits / lus grâce aux fonctions suivantes :

- *wb_hex_as_short*
- *wb_hex_as_int*
- *wb_hex_as_long*
- *rb_next_short_as_hex*
- *rb_next_int_as_hex*
- *rb_next_long_as_hex*

Stocker les *codes* dans les fichiers *.lzw* est crucial au bon fonctionnement de l'algorithme, car sinon les fichiers se retrouveront plus lourd une fois enregistrés (par exemple si tous les codes sont enregistrés sous le type *unsigned long*).

Voici plusieurs stratégies que vous pouvez adopter pour pallier à ce problème :

- Tout écrire grâce au type *unsigned short* et prier 🙏 que la taille du dictionnaire n'excède pas la taille maximale définie pour le type *unsigned short* (**option fortement déconseillée**).

- Définir une taille maximum pour votre dictionnaire (par exemple la taille maximale du type *unsigned short* ou *unsigned int*) et une fois le dictionnaire plein, ne plus rajouter de *codes* supplémentaires.
- Définir une taille maximum pour votre dictionnaire (par exemple la taille maximale du type *unsigned short* ou *unsigned int*), définir un *clear code* et une fois le dictionnaire plein écrire ce *clear code* dans le fichier et vider le dictionnaire (= réinitialiser le dictionnaire) pour pouvoir continuer à y ajouter de tous nouveaux codes.
- Utiliser le bon type en fonction de la taille actuelle du dictionnaire. Par exemple, si la taille du dictionnaire est inférieure à la taille maximale du type *unsigned short*, écrire dans le fichier grâce au type *unsigned short*.

Implémentation du dictionnaire

Pour permettre le bon fonctionnement de l'algorithme *LZW* il est nécessaire de remplir un dictionnaire dont la fonction principale est de pouvoir fournir une association sous la forme **clé/valeur**. L'objectif est de récupérer une valeur en fonction d'une chaîne de caractères (*i.e.* la clé) donnée.

Dans l'algorithme de *LZW* deux dictionnaires sont à gérer, un pour la partie compression et un autre pour la partie décompression. Voici la représentation des paires clé/valeur pour chacun de ces dictionnaires :

Méthode	Clé	Valeur
Compression	*w*	entrée à écrire sous forme hexadécimale
Décompression	entrée des bits lus sous forme hexadécimale	*w*

Liste chaînée

La première implémentation du dictionnaire prendra la forme d'une liste chaînée. La structure des maillons devra comprendre au minimum les éléments suivants :

- clé
- valeur
- pointeur vers le maillon suivant

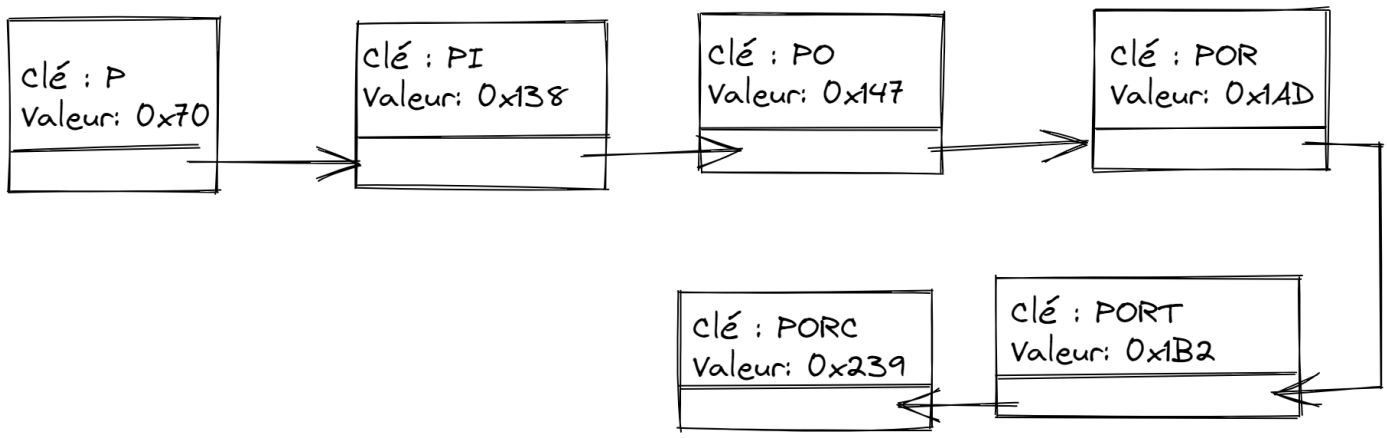

Figure 1 : Exemple d'une liste ayant comme clés p, pi, po, por, porc et port

Trie

Une autre forme que peut prendre une table associative (un des autres noms qu'un dictionnaire peut avoir) est celle d'un **trie** qui permet sa représentation sous forme d'arbre. La structure permettant de désigner un noeud de l'arbre est composée des éléments suivants :

- Un tableau de pointeurs vers ses fils, de taille 128 (un pour chaque caractères *ASCII*, ou alors de taille 256 pour *Extended ASCII*). Une case i de ce tableau est un pointeur vers son noeud fils si il est possible de continuer un mot (dans notre cas la clé) avec la lettre d'indice *i* à partir du préfixe encodé par le noeud courant, ou un pointeur NULL sinon.
- Un attribut permettant de savoir si le mot formé en partant de la racine et en allant jusqu'au noeud courant est une clé valide du dictionnaire.
- Un attribut représentant la valeur à stocker, si la clé courante appartient bien à notre tableau associatif.

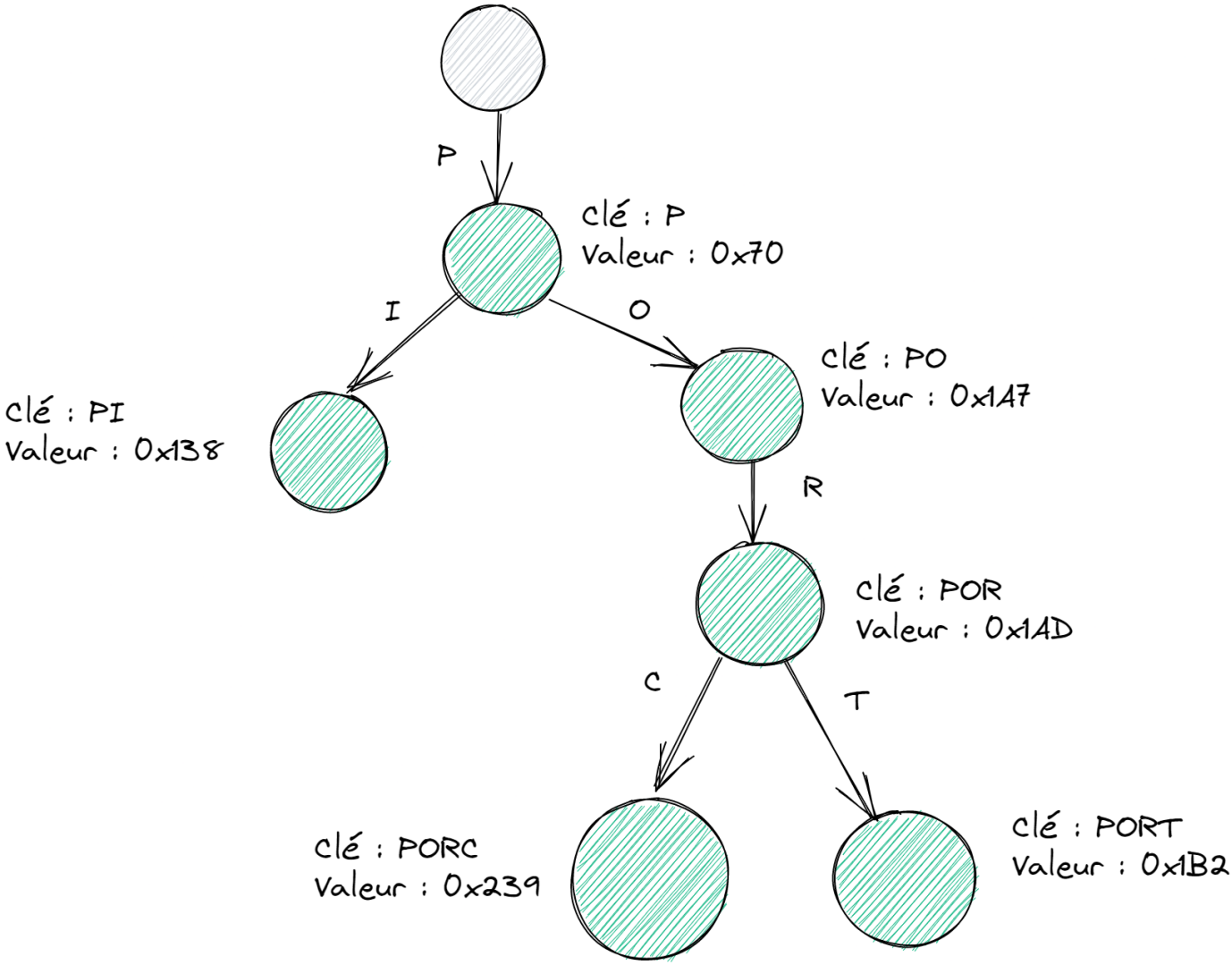

Figure 2 : Exemple d'un trie ayant comme clés p, pi, po, por, porc et port. Les noeuds verts sont des noeuds comprenant une clé valide ; les noeuds gris, l'inverse.

Une version modifiée du *trie* pourra être développée pour permettre un stockage plus performant des codes hexadécimaux dans l'algorithme de décompression.

Hashmap

Une hashmap est l'une des implémentations les plus optimisées d'un tableau associatif. L'idée ici n'est pas de comprendre tous les rouages de son fonctionnement interne mais d'intégrer une structure de données existante sous forme de librairie et d'apprendre à l'utiliser.

La librairie à utiliser sera la suivante https://github.com/sheredom/hashmap.h et sa documentation est disponible dans le *README.md* du répertoire GitHub.

Pour les plus curieux souhaitant comprendre plus en détails la réalisation pratique d'une table de hachage, l'article https://www.andreinc.net/2021/10/02/implementing-hash-tables-in-c-part-1 de Andrei Ciobanu y constitue une bonne porte d'entrée.

Ressources et aides

Vous avez à votre disposition les fichiers suivants :

- L'implémentation d'une hashmap (*hashmap.h*).
- Des fonctions pour la manipulation de caractères et de lecture et écriture (*utils.h*, *utils.c*)
- Des fonctions de conversion de codes hexadécimaux (*hex.h*, *hex.c*)

Vous aurez à utiliser des fonctions de la librairie standard. Vous pourrez inclure <*stdbool.h*> pour utiliser les booléens en C. Pour la manipulation de chaînes de caractères, vous utiliserez <*string.h*>.

- hashmap.h :
  - hashmap_create() qui permet d'initialiser une hashmap avec une taille qui est une puissance de 2;
  - hashmap_get() qui permet de récupérer la valeur pour une clé donnée si la clé existe *NULL* sinon;
  - hashmap_put() qui permet d'enregistrer la valeur pour une certaine clé;
  - hashmap_iterate() qui permet, à l'aide d'un pointeur de fonction, d'itérer à travers chacun des éléments contenus dans le dictionnaire;
  - hashmap_destroy() qui permet de libérer l'espace mémoire alloué à la hashmap. 1ttention l'espace mémoire des éléments qui ont été insérés dans la hashmap ne sont quant à eux pas libérés;
- utils.c :
  - concat() qui permet de créer une nouvelle chaîne de caractères en concaténant deux chaînes de caractères;
  - char2str() qui permet de transformer un caractère en chaîne de caractères;
  - wb_hex_as_short qui permet d'écrire une chaîne hexadécimale sous forme binaire sur une taille de *sizeof(unsigned short)*, le plus souvent 2 octets;
  - wb_hex_as_int qui permet d'écrire une chaîne hexadécimale sous forme binaire sur une taille de *sizeof(unsigned int)*, le plus souvent 4 octets;
  - wb_hex_as_long qui permet d'écrire une chaîne hexadécimale sous forme binaire sur une taille de *sizeof(unsigned long)*, le plus souvent 8 octets;
  - rb_next_short_as_hex qui permet de lire la prochaine chaîne hexadécimale sous forme binaire sur une taille de *sizeof(unsigned short)*, le plus souvent 2 octets;
  - rb_next_int_as_hex qui permet de lire la prochaine chaîne hexadécimale sous forme binaire sur une taille de *sizeof(unsigned short)*, le plus souvent 4 octets;
  - rb_next_long_as_hex qui permet de lire la prochaine chaîne hexadécimale sous forme binaire sur une taille de *sizeof(unsigned short)*, le plus souvent 8 octets;
- hex.c :
  - hex2dec() qui permet de convertir une chaîne hexadécimale en entier de type long
  - dec2hex() qui pemet de convertir un entier de type long en chaîne hexadécimale

Des exemples concrets d'utilisation de la hashmap sont disponibles à l'adresse suivante : https://github.com/sheredom/hashmap.h

Questions

Question 1

Implémentation de la liste chaînée : vous devez proposer une implémentation complète de la structure contenant les fonctions d'**initialisation**, d'**insertion d'une paire clé/valeur**, et de **recherche**.

Question 2

Implémentation du trie : vous devez proposer une implémentation complète de la structure contenant les fonctions d'**initialisation**, d'**insertion d'une paire clé/valeur**, et de **recherche**.

En complément (**non obligatoire**) vous pouvez également fournir une implémentation du trie spécifique à un stockage de clés de codes hexadécimaux.

Question 3

De la même manière que les deux questions précédentes, intégrer la librairie *hashmap.h* pour permettre son utilisation dans votre algorithme *LZW*.

Question 4

Proposer une mise en application de l'algorithme *LZW*. Celle-ci doit être capable de compresser et de décompresser un fichier texte encodé en *ASCII*.

Question 5

Proposer des tests permettant de valider le bon fonctionnement de votre approche et de l'utilisation des différentes structures. Une partie de vos tests doit comprendre une évaluation des performances. L'ensemble du déroulement de vos tests ainsi que leurs résultats doivent être expliqués et détaillés dans votre rapport.

Question 6

Dans votre rapport, vous proposerez une analyse théorique et pratique sur les différences d'utilisation des structures utilisées.