CPS : compte rendu du projet

Voici le compte rendu du projet de CPS sur le dictionnaire, realisé par :

- Matthieu NOGUERON
- Aymeric VIAL-GRELIER

1) Comment est fragmentée l'application?

Liste des différents répertoires et utilitée(s) de chacun

Lorsque vous avez décompressé l'archive, vous avez alors pû constater une structure précise. Voici la description de cette architecture.

- Examples : contient les fichiers de tests au format .txt. Ces fichiers peuvent être utilisés lors de l'utilisation de l'application.
- doc : contient les différents fichiers de documentation du projet (sujet + compte rendu).
- include : contient les headers (.h) du projet:
  - *dictionnaire.h* contient le header concernant le dictionnaire
  - *maillon.h* contient le header concernant les maillons
  - *mot.h* contient le header concernant les mots
  - *read_word.h* contient le header concernant la lecture des fichiers afin de les transformer en mots
  - *utils.h* contient le header concernant des fonctions utiles sur différents plans.
- lib : contient les librairies fournies pour le déroulement du projet au format *.so* et *.dylib*
- src : contient les sources du projet à savoir
  - *dictionnaire.c* contient les différentes fonctions en rapport avec le dictionnaire
  - *maillon.c* contient les différentes fonctions en rapport avec les maillons
  - *main.c* contient le programme principal et les différents traitements proposés par l'application.
  - *mot.c* contient les différentes fonctions en rapport avec les mot
  - *test.c* contient des fonctions de tests utiles pour tester les fonctionnalitées sur des maillons de différentes tailles
  - *utils.c* contient les différentes fonctions d'utilitées diverses

2) Comment utiliser l'application?

Compiler les sources

Rien de plus simple! Deux makefiles sont fournis. *Makefile* est un makefile permettant de compiler l'application sous macOSX et donc utiliser la librairie au format *.dylib*. Si vous souhaitez utiliser l'application sous Ubuntu, il suffit de renomer les deux makefiles

```
mv Makefile MakefileMac
```

```
mv MakefileUbuntu Makefile
```

Puis il suffit de compiler

```
make
```

Tout devrait bien se dérouler. Si vous souhaitez effacer les fichiers générés à la compilation :

```
make clean
```

Utiliser l'application

Plusieurs choix s'offrent à vous et celà se traduit dans notre cas par deux aspects.

- le premier aspect est la présence de plusieurs exécutables : *target/dictionarizeX*, *target/test_functionX* ou X peut prendre la valeur : 8, 16, 32, 64 et qui fixe la taille des maillons utilisés.
  - *target/dictionarizeX* permet d'utiliser l'application "normalement"
  - *target/test_functionX* permet d'utiliser des programmes de tests générant des mots de manière aléatoire et ensuite tester plusieurs fonctionnalités comme affichage des maillons, comparaisons de mots.

Afin d'utiliser l'application de manière normale trois choix s'offrent à vous:

- `./target/dictionarizeX` vous permet d'utiliser l'application avec pour entrée *stdin*
- `./target/dictionarizeX [PATH_TO_FILE]` vous permet d'utiliser l'application avec pour entrée un fichier
- `./target/dictionarizeX [PATH_TO_FILE]*` vous permet d'utiliser l'application avec pour entrée plusieurs fichiers

Dans chacun des cas ci-dessus un menu s'affichera et vous proposera soit d'afficher le dictionnaire en tappant 1 dans votre console et en validant avec un retour chariot, soit d'afficher le dictionnaire AVEC les maillons associés à chaque mot du dit dictionnaire en tappant 2. Tapper autre chose que 1 ou 2 entrainera la fermeture de l'application.

Attention dans le cas ou vous utilisez l'application avec *stdin* il vous sera impossible d'accéder au menu et seul l'affichage du dictionnaire sera effectué. De plus après avoir saisi votre texte vous devrez appuyer sur `ctrl+D` pour afficher le résultat d'éxécution.

3) Détails des différentes fonctions

- *dictionnaire.h*

  - *pDictionnaire* **add_to_head_dictionnaire** *(pDictionnaire d, Mot m);*

  prends en entrée un pointeur vers un dictionnaire et un Mot, puis ajoute ce mot au dictionnaire en tête de dictionnaire, et enfin retourne le pointeur vers le dictionnaire.

- *pDictionnaire* **add_to_tail_dictionnaire** *(pDictionnaire d, Mot m)*;

  prends en entrée un pointeur vers un dictionnaire et un Mot, puis ajoute ce mot au dictionnaire en queue de dictionnaire, et enfin retourne le pointeur vers l'ancien dictionnaire avec le Mot en plus.

- *pDictionnaire* **add_inside_dictionnaire** *(pDictionnaire pred, pDictionnaire succ, Mot m)*;

  prends en entrée un mot deux pointeurs vers

- *maillon.h*

- *mot.h*
- *read_word.h*
- *utils.h*

4) Choix et justifications

Globalement la structure choisie est celle proposée dans le sujet. Néanmoins afin d'aider dans le développement du projet nous avons ajouté des fonctions de débugage comme les fonctions d'affichage binaires.

De plus nous avons rajouté des fonctions de libérations de mémoires.

Nous avons commencé par gérer le type *uint32_t*, pour cette structure de donnée l'élément que l'on devait ajouter à notre maillon était de type *uint8_t* ce qui correspond à un caractèr, cependant quand nous avons voulu passer à la version 64bits, nous avons remarqué que faire un décalage vers la gauche supérieur à 31 d'un uint8_t le faisait faire un cycle. Ainsi au lieu d'écrire à la place 0 dans le cas d'un *uint64_t* on écrivait à la place 5 d'où un problème et d'où le passage a une variable à ajouter de type Storage (identique à celui de notre maillon). Enfin dans la fonction get_charnum, nous avons du faire la même chose pour la valeur 31 que l'on décallait pour obtenir un masque. De base un entier est codé sur 32 bits et oppérer un décalage donc supérieur à 31 le faire revenir au début. Nous avons donc du assigner une variable mask de type Storage sur laquelle on exécute le décalage. Enfin la paramétrisation s'est faite par l'intermédiaire de 3 define *SIZE*, *NBL* et *Storage*, qui est le type de notre maillon. Ces define sont définis lors de la compilation grâce à l'option `-D -DSIZE=x ... -DNBL=x`

Nous avons fait ce choix pour simplifier la compilation dans de multiples version (64, 32, 16 et 8 bits) de notre programme

Enfin nous avons décidé de donner la possibilité à l'utilisateur de choisir s'il veut afficher simplement le dictionnaire ou bien le dictionnaire avec l'affichage des maillons

Nous avons aussi choisi de supporter un nombre supérieur à 1 de fichiers à analyser dans les arguments passés au programme

Cependant si un fichier n'est pas ouvrable, celà génére une segfault (retour code 1).

5) Possibilités d'évolution

Dans le futur afin de perfectionner l'application il serait possible d'inclure la gestion des chiffres et nombres, ainsi que les caractères spéciaux et ainsi permettre une plus vaste et plus libre utilisation de l'application. Nous avons préféré ne pas s'attarder sur ces fonctionnalités, n'étant pas un des points majeurs de ce projet, néanmoins ce pourrait être utile.