Première partie

Présentation du projet

L'objectif de ce TP est de coder un système capable de gérer et de parcourir des arborescences structurées par lien vertical et horizontal. La structure de donnée sera codée en `C` et le code contiendra les fonctions de recherche et suppression (avec élagage), ainsi que la possibilité de charger un arbre en mémoire à partir de son écriture algébrique dans un fichier. Le code pourra aussi afficher à l'écran le nombre de nœuds, le nombre de feuilles, et la hauteur d'un arbre.

1 Description du projet

1.1 Schéma de la structure de donnée

Chaque nœud d'un arbre est représenté par un mot de 3 bloc. Ce bloc contient :

- La valeur du nœud (ici, un seul caractère).
- Le lien horizontal du nœud, c'est à dire un pointeur vers son frère suivant (s'il en a un).
- Le lien vertical du nœud, c'est à dire un pointeur vers son premier fils (s'il en a).

Le premier nœud est appelé *racine* de l'arbre.

Le parcours d'un arbre se fait grâce à une pile. Une pile est représentée par un tableau d'éléments. L'un d'entre eux est un désigné comme étant la tête. L'indice de l'élément de tête est stockée en mémoire.

Dans le code, les piles et les arborescences utilisées sont des `structs`. De plus, les éléments de la pile sont déclarés avec un `typedef` afin d'en garantir la généricité et sont, par défaut, des `void*`.

Quant au valeur des nœuds d'un arbre, ce sont des `char`.
`(cf. DDD_arbre_exemple.png)`.

1.2 Organisation du code source

Le code source est divisé en trois parties : Le noyau (toutes les fonctions répondant au besoins du TP), la section « tests » pour les fonctions de test, le système de gestion des tests (indépendant du TP), et enfin le programme principal, `main.c`, contenant la fonction `main()` qui charge un fichier dont le nom est donné en ligne de commande et affiche la structure de donnée

générée. Le fichier doit contenir la notation algébrique d'un arbre avec un seul caractère par élément.

Une exécution du programme avec le fichier `exemple.arbre` vu par `ddd` est donné en annexe, ainsi que son rapport `valgrind`.

dans `main.c`, qui permet soit d'exécuter les tests de toutes les fonctions du noyau, soit de charger un fichier à traiter (le nom du fichier doit être donné en commande).

Les macros et types généraux sont déclarés dans `lib.h`.

Les fonctions relatives à la pile sont codés dans `pile.c`.

- `initPile()`
- `empiler()`
- `depiler()`
- `vide()`
- `pleine()`
- `libererPile()`

Les fonctions relatives aux arbres sont codées dans `arbre.c`.

- `initArbre()`
- `arbreSupprimer()`
- `arbreSupprimerValeur()`
- `arbreRecherche()`
- `compterNoeuds()`
- `mesurerHauteur()`
- `compterFeuilles()`
- `libererArbre()`

La fonction `initArbre()` utilise trois sous-programmes :

- `obtenirValeur()`
- `obtenirOperation()`
- `obtenirSuivant()`

La documentation des ces fonctions se trouve respectivement dans `pile.h` et `arbre.h`. Ces fichiers contiennent aussi des fonctions `toString()` pour convertir les structures de données associées en chaine de caractères, utilisées uniquement par le système de test.

Les fonctions qui gèrent l'exécution et l'affichage des sorties des test sont

déclarées dans `testlib.h` et codées dans `testlib.c`.

Chaque fonction du noyau est testée individuellement dans `tests.c` selon le principe des tests unitaires (c'est-à-dire que chaque fonction est testée dans un contexte qui lui est propre). La liste des tests à exécuter est déclarée dans `tests.h`.

Pour lancer les tests, ajoutez `test` à la ligne d'exécution du programme.

Le résultat d'une exécution de ces tests est donné en annexe, ainsi que son rapport `valgrind`.

2 Méthodologie

2.1 Langage de programmation

Il était imposé d'utiliser le langage C dans le cadre de ce TD.

2.2 Gestionnaire de version

Afin de pouvoir travailler à deux simplement nous avons décidé d'utiliser le gestionnaire de version GIT ainsi que GitHub.com pour centraliser le code.