TP3
Dictionnaire arborescent

Mathieu ARQUILLIERE

Table des matières

I – Présentation

L'objectif de ce TP est de créer une bibliothèque de gestion de dictionnaire arborescent en utilisant les notions d'arbre vu en cours et la bibliothèque de la pile créée au TP 2.

II – Description des structures de données

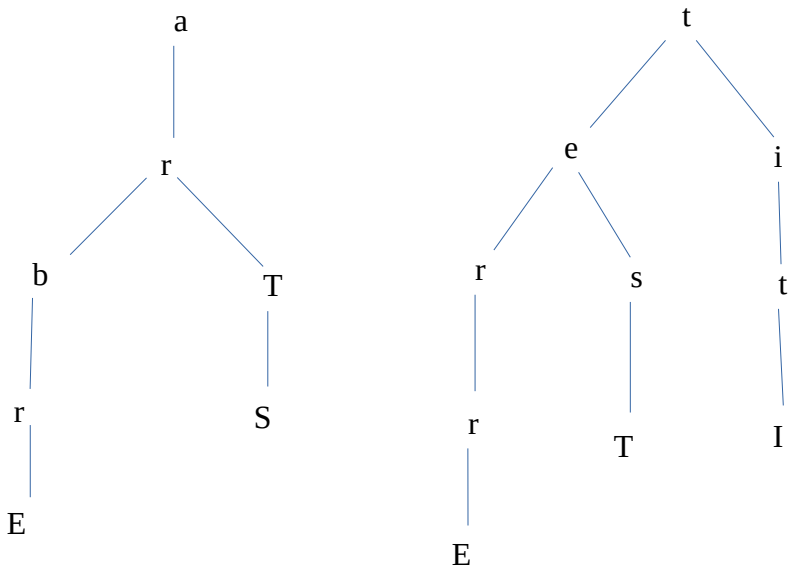
Les arbres seront utilisés avec la structure des liens horizontaux et liens verticaux. Chaque point de l'arbre contient donc une valeur (en l'occurrence un caractère), un pointeur sur son frère le plus proche et un pointeur sur son premier fils. On a donc des gestions de listes chaînées.

Les arbres seront utilisés pour contenir des dictionnaires. Il y a donc les contraintes suivantes :

- Chaque point de l'arbre est une lettre
- Une lettre ne figure qu'une seule fois dans la liste des successeurs d'un point
- Les lettres appartenant à une même liste chaînée en lien horizontal sont triées par ordre alphabétique
- Si un point est une majuscule, la suite de lettres depuis la racine jusqu'à ce point forme un mot
- Si un mot existe dans le dictionnaire, il existe un chemin depuis la racine jusqu'à une lettre en majuscule et dont les caractères des points forment le mot

Exemple de dictionnaire sous forme d'arbre contenant les mots :

arbre ; arts ; art ; test ; terre ; titi



III – Organisation du code

1) Module Pile

Ce programme utilise des arbres, avec des algorithmes itératifs. On a donc besoin d'une pile pour certains parcours d'arbre. Ainsi on réutilise le module pile réalisé dans le TP 3 (Pile, File et dérécursification). Pour pouvoir l'utiliser, on modifie juste le type des éléments de la pile (On met le type d'un pointeur de nœud d'arbre) tout en ayant préalablement indiqué au compilateur que le type nœud existe. En effet on ne peut pas inclure le fichier « arbre.h » car on inclut dans celui-ci le fichier « pile.h » et la double inclusion ne fonctionne pas.

```c
struct noeud;

#define FORMAT_PILE "%p"
typedef struct noeud* T;
```

2) Module Arbre

```c
struct noeud
{
    char            lettre;
    struct noeud*   lv;
    struct noeud*   lh;
};
typedef struct noeud noeud_t;
```

Un nœud d'arbre est représenté par une structure contenant un caractère (la valeur du nœud), un pointeur sur sa propre structure pour le lien vertical et un autre pointeur sur sa propre structure pour le lien horizontal.

```c
noeud_t*    creer_cell      (char l);
void        adj_cell        (noeud_t** prec, noeud_t* nouv);
int         rech_prec       (noeud_t** liste, char l, noeud_t*** prec);
int         recherche       (noeud_t** racine, char* mot, int tailleMot, noeud_t** derCell);
void        ajouter_mot     (noeud_t** racine, char* mot, int tailleMot);
void        affichage_arbre (noeud_t* a, char* prefixe);
void        affichage_motif (noeud_t* racine, char* motif, int tailleMotif);
void        liberer_arbre   (noeud_t** racine);
void        debug_arbre     (noeud_t* racine);
```

L'arbre a plusieurs fonctions associées :

- **creer_cell** permet d'allouer et initialisé un nœud d'arbre
- **adj_cell** permet de relier un nœud grâce à un précédent
- **rech_prec** permet de rechercher le précédent dans la liste chaînée des liens horizontaux
- **recherche** permet de parcourir l'arbre pour trouver un mot donné (ou du moins le début)
- **ajouter_mot** permet d'ajouter un mot dans l'arbre
- **affichage_arbre** permet d'afficher tous les mots contenus dans l'arbre
- **affichage_motif** permet d'afficher les mots de l'arbre commençants par un motif donné
- **liberer_arbre** permet de supprimer l'arbre et de libérer toute la mémoire associée
- **debug_arbre** permet d'afficher le contenu de l'arbre avec les niveaux des nœuds

IV – Présentation du programme

1) Module Pile

cf. code de pile.h et pile.c en annexe.

2) Module Arbre

cf. code de arbre.h et arbre.c en annexe.

3) Programme de tests

Le programme du code du fichier « test.c » consiste à tester toutes les fonctionnalités et les limites du module arbre. C'est donc une seule fonction (main) qui créer un arbre et effectue des opérations dessus afin de détecter les différents comportements. Pour plus de précision sur les tests effectués, cf. compte rendu d'exécution du programme de tests.

V – Compte-rendu d'exécution

1) Tests de création d'arbre

Liste des cas à traiter sur la création de l'arbre :

- ajouter un mot à un arbre vide
- ajouter un mot à un arbre avec une racine non existante
- ajouter un mot vide
- ajouter un mot qui existe déjà à cause d'un autre mot existant
- ajouter un mot avec une racine déjà existante
- ajouter un mot déjà existant dans l'arbre

On teste tous ces cas dans le programme « test.c » :

```
noeud_t* arbre = NULL;

ajouter_mot(&arbre, "test", 4);
ajouter_mot(&arbre, "arbre", 5);
ajouter_mot(&arbre, "arts", 4);
ajouter_mot(&arbre, "art", 3);
ajouter_mot(&arbre, "terre", 5);
ajouter_mot(&arbre, "titi", 4);
ajouter_mot(&arbre, "titiar", 6);
ajouter_mot(&arbre, "", 0);
ajouter_mot(&arbre, "arbre", 5);
ajouter_mot(&arbre, "arbre", 5);
ajouter_mot(&arbre, "abouti", 6);
```

Initialiser un arbre vide

Ajouter un mot à un arbre vide

Ajouter un mot avec une racine inexistante

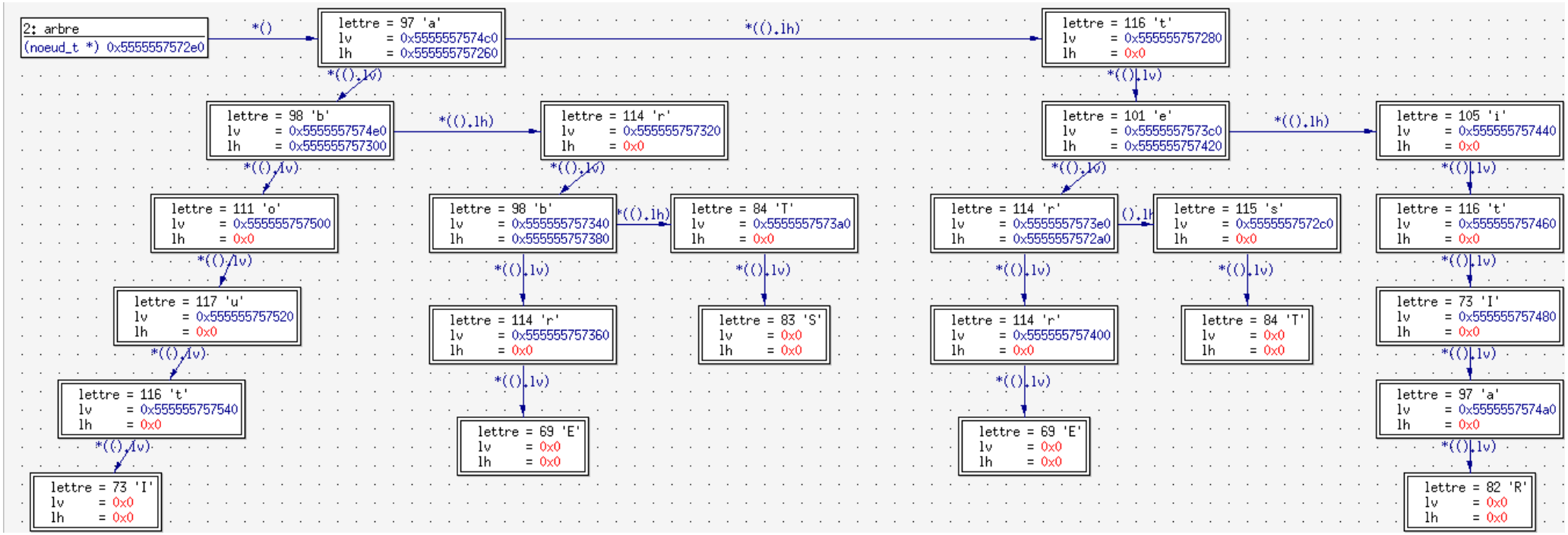
Ajouter un mot qui existe déjà (art) à cause d'un mot existant (arts)

Ajouter un mot avec une racine existante (titiar a la racine titi)

Ajouter un mot vide

Ajouter un mot déjà existant

On a le résultat suivant (réalisé avec ddd) :



La fonction ajouter_mot gère donc parfaitement le traitement de tous les cas listés ci-dessus.

2) Tests de l’affichage

L’affichage global de l’arbre affiche tous les mots par ordre alphabétique :

```c
printf("=== Arbre ===\n");
affichage_arbre(arbre, "");
```

```
=== Arbre ===
abouti
arbre
art
arts
terre
test
titi
titiar
```

Liste des cas à traiter pour l’affichage par motif :

- un motif vide
- un motif avec une seule lettre (dans les racines de l’arbre)
- un motif avec des mots existants
- un motif avec aucun mot existant
- un motif représentant un mot complet
- un motif plus long que les mots de l’arbre

On teste tous ces cas dans le programme « test.c » :

```c
printf("=== Motif vide ===\n");
affichage_motif(arbre, "", 0);

printf("=== Motif a ===\n");
affichage_motif(arbre, "a", 1);

printf("=== Motif ar ===\n");
affichage_motif(arbre, "ar", 2);

printf("=== Motif ba ===\n");
affichage_motif(arbre, "ba", 2);

printf("=== Motif test ===\n");
affichage_motif(arbre, "test", 4);

printf("=== Motif tests ===\n");
affichage_motif(arbre, "testsss", 7);
```

```
=== Motif vide ===
=== Motif a ===
abouti
arbre
art
arts
=== Motif ar ===
arbre
art
arts
=== Motif ba ===
=== Motif test ===
test
=== Motif tests ===
```

VI – Makefile

Afin de compiler les programmes, on utilise un makefile :

```
OPT = -g -Wall -Wextra
LOG = @echo "\#MAKE"
TEST = test

all: $(TEST)

$(TEST): test.c pile.o arbre.o
    gcc -o $(TEST) test.c pile.o arbre.o $(OPT)
    $(LOG) "executable $(TEST) généré"

pile.o: pile.h pile.c
    gcc -o pile.o -c pile.c $(OPT)
    $(LOG) "lib pile généré"

arbre.o: arbre.h arbre.c
    gcc -o arbre.o -c arbre.c $(OPT)
    $(LOG) "lib arbre généré"

clean:
    rm *.o
    rm $(TEST)
```

Ce makefile génère l'exécutable des tests sur les arbres.

Il a besoin du module d'arbre et du module de pile (donc arbre.o et pile.o) et du code source de test (test.c).

On a donc une règle pour chaque .o et une règle pour l'exécutable.

Commandes :

- "make" pour faire l'exécutable « test »
- "make clean" pour effacer l'exécutable et les .o
- "./test" pour lancer le programme