ISIMA Première Année

Compte-Rendu de TP

Structures de Données

Gestion d'un dictionnaire arborescent

Benjamin BARBESANGE
Pierre-Loup PISSAVY
Groupe G21

Enseignant :
Michelle CHABROL

mars 2015

ISIMA

Table des matières

1 | Présentation

Le but de ce TP est de créer une structure d'arbre permettant de gérer des mots d'un dictionnaire. Chaque liste des liens horizontaux est rangée par ordre alphabétique.

Les opérations suivantes sont permises avec l'arbre :

- Creer l'arbre à partir de la notation parenthésée,
- Insérer un mot à la bonne place dans l'arbre,
- Afficher le contenu de l'arbre,
- Rechercher des mots commençant par un certain motif,
- Libérer la mémoire occupée par l'arbre.

1.1 Structure de données employée

Les mots du dictionnaire sont rangés dans un arbre à liens horizontaux et verticaux, par orbre alphabétique en lecture préfixe. La fin d'un mot est signalée par une lettre majuscule.

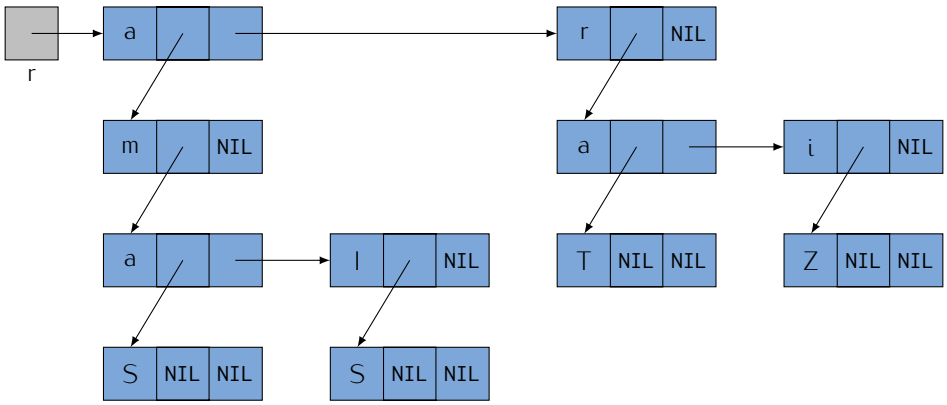

FIGURE 1.1 – Représentation en liens vertical et horizontal

Afin d'effectuer les tests, nous proposons une fonction basique effectuant des tests sommaires, ainsi que l'interprétation d'un fichier qui peut être donné comme premier paramètre.

Le cas échéant, la structure de ce fichier doit respecter les règles suivantes :

- 400 caractères au maximum par ligne,
- Les caractères suivants sont acceptés en début de ligne :
  C : Création d'arbre, doit contenir ensuite une représentation parenthésée,
  I : Insertion, peut contenir un mot ensuite,
  M : Recherche de motif, peut contenir un motif ensuite (chaîne de caractères),
  L : Libérer l'arbre,
  A : Afficher l'arbre,
  # : Provoque l'affichage du texte qui suit (commentaire affiché).
- Pour l'insertion, la casse n'a pas d'importance,
- Si l'on souhaite créer un nouvel arbre apres en avoir créé un premier, il est nécessaire de libérer ce dernier,
- Tout autre caractère ou bien une ligne vide provoqueront l'affichage d'une ligne vide.

1.2 Organisation du code source

Nous avons découpé le TP en 3 parties. Une partie permet la gestion de pile, une autre la gestion de listes chaînées (qui sont utilisées dans la définition de l'arbre) et la dernière gère la structure d'arbre que nous avons créée.

1.2.1 Gestion de la pile

- `src/stack.h`
- `src/stack.c`

1.2.2 Gestion des listes chaînées

- `src/list.h`
- `src/list.c`

1.2.3 Gestion de l'arbre

- `src/tree.h`
- `src/tree.c`

1.2.4 Programme principal

- `src/main.c`

2 | Détails du programme

2.1 Gestion de la pile

Nous avons toutefois ajouté une fonction d'affichage du contenu de la pile (dump).

Code C

```c
/*  stack.h
  Header

  -----| PILE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#ifndef __STACK__H
#define __STACK__H

  #include <stdio.h>
  #include <stdlib.h>
  #include "tree.h"

  typedef tree_t* datatype;

  typedef struct _stack_t {
    int       max;  /* nombre max d'elements dans la pile */
    int       top;  /* position de l'element en tete de pile */
    datatype *val;  /* tableau des valeurs de la pile */
  } stack_t;

  int init(stack_t *,int);
  void supp(stack_t *);
  int empty(stack_t);
  int full(stack_t);
  int pop(stack_t *, datatype *);
  int top(stack_t *, datatype *);
  int push(stack_t *, datatype);
  void dump(stack_t, void (*)(datatype));

#endif
```

Code C

```c
/*  stack.c
  Fonctions de gestion de la structure de pile

  -----| PILE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/
#include "stack.h"

/*  int init(stack_t *p, int n)
Fonction d'initialisation de la pile avec une taille max

  Entrees :
    *p : pointeur sur la pile
    n : taille maximum de la pile

  Sortie :
    int : code d'erreur
      1 si aucune erreur
      0 si erreur de creation de la pile
*/
int init(stack_t *p, int n) {
  int ret = 1;
  p->max = n;
  p->top = -1;
  p->val = (datatype*) malloc(n*sizeof(datatype));
  if (p->val == NULL) {
    ret = 0;
  }
  return ret;
}

/*  void supp(stack_t *p)
Fonction de suppression de la pile

  Entree :
    *p : pointeur sur la tete de la pile

  Sortie :
    Aucune
*/
void supp(stack_t *p) {
  free(p->val);
  p->top = -1; /* Empeche de depiler */
  p->max = 0; /* Empeche d'empiler */
}

/*  int empty(stack_t *p)
Teste si la pile est vide ou non

  Entree :
    p : tete de la pile
```

```
56
57    Sortie :
58      int : booleen
59        0 si la pile n'est pas vide
60        1 si la pile est vide
61  */
62  int empty(stack_t p) {
63    return (p.top == -1)?1:0;
64  }
65
66  /*  int full(stack_t p)
67  Teste si la pile est pleine ou non
68
69    Entree :
70      p : tete de la pile
71
72    Sortie :
73      int : booleen
74        0 si la pile n'est pas pleine
75        1 si la pile est pleine
76  */
77  int full(stack_t p) {
78    return (p.top == p.max-1)?1:0;
79  }
80
81  /*  int pop(stack_t *p, datatype *v)
82  Recupere le premier element de la pile (et l'enleve) et retourne un code d'erreur
83
84    Entree :
85      *p : pointeur sur la tete de la pile
86      *v : pointeur sur un element du type de la pile, variable en I/O
87
88    Sortie :
89      int : code d'erreur
90        0 si rien n'est retourne dans la variable v
91        1 si on a recupere l'element en tete
92  */
93  int pop(stack_t *p, datatype *v) {
94    int ok = 0;
95    if (!empty(*p)) {
96      *v = p->val[p->top];
97      ok = 1;
98      p->top--;
99    }
100   return ok;
101 }
102
103 /*  int top(stack_t *p, datatype *v)
104 Retourne l'element en tete de la pile (sans l'enlever) et retourne un code d'erreur
105
106   Entree :
107     *p : pointeur sur la tete de la pile
108     *v : pointeur sur un element du type de la pile, variable en I/O
109
110   Sortie :
111     int : code d'erreur
```

```
      0 si rien n'est retourne
      1 si on recupere l'element en tete
*/
int top(stack_t *p, datatype *v) {
  int ok = 0;
  if (!empty(*p)) {
    *v = p->val[p->top];
    ok = 1;
  }
  return ok;
}

/*  int push(stack_t *p, datatype v)
Insere un element en tete de la pile

  Entree :
    *p : pointeur sur la tete de la pile
    v : element a inserer dans la pile

  Sortie :
    int : code d'erreur
      0 si l'element n'est pas ajoute dans la pile
      1 si l'element est ajoute dans la pile
*/
int push(stack_t *p, datatype v) {
  int ok = 0;
  if (!full(*p)) {
    p->top++;
    p->val[p->top] = v;
    ok = 1;
  }
  return ok;
}

/*  void dump(stack_t p, void (*afficherData)(datatype))
Affiche le contenu de la pile

  Entree :
    p : tete de la pile
    *afficherData : pointeur de fonction permettant l'affichage des elements

  Sortie :
    Aucune
*/
void dump(stack_t p, void (*afficherData)(datatype)) {
  int i;
  if (!empty(p)) {
    for (i = 0; i <= p.top; i++) {
      afficherData(p.val[i]);
    }
  }
}
```

2.2 Gestion des listes chaînées

Code C

```c
/*  list.h
  Header

  -----| LISTE CHAINEE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#ifndef __LISTE_H__
#define __LISTE_H__

  #include <string.h>
  #include <ctype.h>

  typedef struct _node_t {
    char letter;
    struct _node_t *lv;
    struct _node_t *lh;
  } node_t;

  typedef node_t cell_t;

  void adj_cell(cell_t **, cell_t *);
  cell_t ** rech_prec(cell_t **, char, short int*);
  void supp_cell(cell_t **);
  void liberer_liste(cell_t **);
  void ins_cell(cell_t **, cell_t *);
  cell_t * creer_cell(char);

#endif
```

Code C

```c
/*  list.c
  Fonctions de gestion de la liste chainee

  -----| LISTE CHAINEE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#include <stdio.h>
#include <stdlib.h>
#include "list.h"

/*  void adj_cell(cell_t **prec, cell_t *elt)
  Ajoute une cellule apres un element partir d'un pointeur sur l'element
  et d'un pointeur sur le pointeur de l'element apres lequel ajouter
```

```c

  Entrees :
    cell_t **prec : pointeur sur le pointeur de l'element apres lequel ajouter
    cell_t *elt : pointeur sur l'element a ajouter a la liste chainee

  Sortie :
    Aucune
*/
void adj_cell(cell_t **prec, cell_t *elt) {
  elt->lh = (*prec);
  (*prec) = elt;
}

/*  cell_t ** rech_prec(cell_t **liste, char letter, short int *existe)
  Recherche le precedent d'un element dans la liste chainee

  Entrees :
    cell_t **liste : pointeur sur le pointeur du premier element de la liste chainee
    char lettre : caractere a chercher dans la liste
    short int *existe : variable en entree/sortie indiquant la présence de la lettre
      0 : absence
      1 : presence

  Sortie :
    cell_t ** : pointeur sur le pointeur de l'element precedent
*/
cell_t ** rech_prec(cell_t **liste, char letter, short int *existe) {
  cell_t **prec = liste;
  while ((*prec) && tolower((*prec)->letter) < tolower(letter)) {
    prec = &((*prec)->lh);
  }
  /* Booleen de presence     */
  /* 1 : present             */
  /* 0 : absent              */
  *existe = (*prec && tolower((*prec)->letter) == tolower(letter))?1:0;
  return prec;
}

/*  void supp_cell(cell_t **prec)
  Permet de supprimer un element dans la liste chainee a partir
  de son precedent

  Entrees :
    cell_t **prec : pointeur sur le pointeur de l'element precedent l'element a supprimer

  Sortie :
    Aucune
*/
void supp_cell(cell_t **prec) {
  cell_t *elt = *prec;
  *prec = elt->lh;
  free(elt);
}

/*  void liberer_liste(cell_t **liste)
  Libere les allocations memoires de la liste
```

```
75
76    Entrees :
77      cell_t **liste : pointeur sur le pointeur du premier element de la liste chainee
78
79    Sortie :
80      Aucune
81  */
82  void liberer_liste(cell_t **liste) {
83    while (*liste) {
84      supp_cell(liste);
85    }
86    *liste = NULL;
87  }
88
89  /*  void ins_cell(cell_t **liste, cell_t *elt)
90    Permet d'inserer une cellule a la bonne place dans la liste chainee
91
92    Entrees :
93      cell_t **liste : pointeur sur le pointeur du premier element de la liste chainee
94      cell_t *elt : pointeur sur l'element a inserer dans la liste chainee
95
96    Sortie :
97      Aucune
98  */
99  void ins_cell(cell_t **liste, cell_t *elt) {
100   short int existe;
101   cell_t **prec = rech_prec(liste,elt->letter,&existe);
102   adj_cell(prec,elt);
103 }
104
105 /*  node_t * creer_cell(char letter)
106   Permet de creer un element de la liste chainee a partir du
107   caractere donne en parametre
108
109   Entrees :
110     char letter : lettre a mettre dans l'element
111
112   Sortie :
113     node_t* : pointeur sur l'element cree
114 */
115 node_t * creer_cell(char letter) {
116   node_t *elt = (node_t*) malloc(sizeof(node_t));
117   if (elt) {
118     elt->letter = letter;
119   }
120   return elt;
121 }
```

2.3 Gestion de l'arbre

Code C

```c
/*  tree.h
  Header

  -----| GESTION DU DICTIONNAIRE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#ifndef __TREE__H
#define __TREE__H

  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include <ctype.h>
  #include "list.h"
  typedef node_t tree_t;
  #include "stack.h"

  #define DEBUG 0
  #define STACKSIZE 200

  int  creerArbre(char *, tree_t **);
  void libererArbre(tree_t **);
  void afficherArbrePref(tree_t *, char *);
  void afficherArbre(tree_t *);
  void afficherPoint(tree_t *);
  int  insererMot(tree_t **, char *);
  void rech_motif(tree_t **, char *);

#endif
```

Code C

```c
/*  tree.c
  Fonction de gestion de l'arbre

  -----| GESTION DU DICTIONNAIRE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#include "tree.h"

tree_t *creerNoeud(char);

/*  int creerArbre(char *ch, tree_t **r)
Cree un arbre a partir d'une chaine de caracteres representant la notation parenthesee
```

```c
et en prenant l'adresse du pointeur sur la tete de l'arbre

  Entrees :
    char *ch : chaine de representation de l'arbre
    tree_t **r : pointeur double de tête de l'arbre

  Sortie :
    int : code de retour sur la creation
          0 : probleme d'alloc d'element
          1 : aucun probleme
*/
int creerArbre(char *ch, tree_t **r) {
  stack_t p;                    /* Pile */
  tree_t **prec = r;            /* Pointeur de parcours de l'arbre */
  tree_t *tmp;                  /* Pointeur temporaire */
  char *cour = ch;              /* Caractere courant */
  int ret = 0;                  /* Variable de retour */

  cour++;                       /* On consomme la premiere parenthese */
  if (init(&p,STACKSIZE)) {
    ret = 1;                    /* Allocation ok */
    while (ret && (!empty(p) || *cour != ')')) {   /* Aucun souci et chaine non-finie */
      if (*cour == '(') {         /* Ajout de fils */
        push(&p,*prec);           /* Sauvegarde de l'adresse courant */
        prec = &((*prec)->lv);    /* Deplacement sur le lien vertical */
        cour++;                   /* Acceleration, passe au prochain caractere */
      } else if (*cour == ',') {  /* Ajout de frere */
        prec = &((*prec)->lh);    /* Deplacement sur le lien horizontal */
        cour++;
      }
      *prec = creerNoeud(*cour);
      if (! (*prec)) {
        ret = 0;                  /* Problème allocation */
      } else {
        cour++;                   /* Passage caractere suivant */
      }
      while (ret && !empty(p) && *cour == ')') {
        pop(&p,&tmp);             /* Recuperation du lien horizontal parent */
        prec = &(tmp->lh);
        cour++;
      }
    }
    supp(&p);                     /* Liberation pile */
  }
  return ret;
}

/*  tree_t *creerNoeud(char v)
Cree un noeud ayant pour valeur le caractere entre

  Entrees :
    char v : valeur du nouveau noeud cree

  Sortie :
    tree_t* : pointeur sur le nouvel element cree
*/
```

```c
tree_t *creerNoeud(char v) {
  tree_t *r = (tree_t*) malloc (sizeof(tree_t));
  if (r) {          /* Allocation OK */
    r->lv = NULL;   /* Initialisation lien vertical */
    r->lh = NULL;   /* Initialisation lien horizontal */
    r->letter = v;  /* Initialisation valeur */
  }
  return r;
}

/*  void afficherArbrePref(tree_t *t, char *prefixe)
Affiche les mots contenus dans l'arbre avec un prefixe donne en entre

  Entrees :
    tree_t *t : pointeur sur la tete de l'arbre
    char *prefixe : prefixe a ecrire avant chaque mot de l'arbre

  Sortie :
    Aucune
*/
void afficherArbrePref(tree_t *t, char *prefixe) {
  stack_t p;                    /* Pile */
  tree_t *cour = t;             /* Pointeur de parcours de l'arbre */

  if (cour != NULL && init(&p,STACKSIZE)) {
    do {
      while (cour != NULL) {
        push(&p,cour);              /* Sauvegarde du point courant */
        if (isupper(cour->letter)) { /* Detection fin de mot */
          printf("%s", prefixe);    /* Affiche le prefixe */
          dump(p,afficherPoint);    /* Affichage du mot (lecture pile) */
          printf("\n");
        }
        cour = cour->lv;            /* Deplacement sur le lien vertical */
      }
      /* On a atteint une feuille */
      while (!empty(p) && cour == NULL) {  /* Recherche du premier frere des ascendants */
        pop(&p,&cour);              /* Recuperation du parent */
        cour = cour->lh;            /* Deplacement sur le lien horizontal */
      }
    } while (!empty(p) || cour != NULL);
    supp(&p);
  }
}

/*  void afficherArbrePref(tree_t *t)
Affiche les mots contenus dans l'arbre

  Entrees :
    tree_t *t : pointeur sur la tete de l'arbre

  Sortie :
    Aucune
*/
void afficherArbre(tree_t *t) {
  afficherArbrePref(t, "");
```

```c
}

/*  void afficherPoint(tree_t *t)
Affiche la valeur d'un noeud, connaissant son adresse

  Entrees :
    tree_t *t : pointeur sur le noeud

  Sortie :
    Aucune
*/
void afficherPoint(tree_t *t) {
  printf("%c",tolower(t->letter));
}

/*  void libererArbre(tree_t **t)
Libere la memoire occupee par l'arbre

  Entrees :
    tree_t **t : adresse du pointeur sur la tete de l'arbre

  Sortie :
    Aucune
*/
void libererArbre(tree_t **t) {
  stack_t p;                    /* Pile */
  tree_t *cour = *t;            /* Pointeur de parcours de l'arbre */
  tree_t *tmp;                  /* Pointeur temporaire pour conserver l'adresse du point a supprimer */

  if (cour != NULL && init(&p,STACKSIZE)) {
    do {
      while (cour != NULL) {
        tmp = cour;                 /* Sauvegarde du courant */
        if (cour->lh != NULL) {
          push(&p,cour->lh);        /* Sauvegarde du frere */
        }
        cour = cour->lv;            /* Deplacement sur le lien vertical */
        free(tmp);                  /* Suppression du point courant */
      }
      if(!empty(p)) {
        pop(&p,&cour);              /* Recuperation du premier lien horizontal parmi les parents */
      }
    } while (!empty(p) || cour != NULL);
    supp(&p);
  }
  *t = NULL;
}

/*  tree_t **rech_mot(tree_t **t, char **w)

  Entrees :
    **t : adresse du pointeur de tete de l'arbre
    **w : pointeur sur le mot a chercher

```

```
  Sortie :
    tree_t ** : adresse du pointeur dans l'arbre ou on a trouve la derniere lettre
                possible du mot
*/
tree_t **rech_mot(tree_t **t, char **w) {
  char *cour = *w;        /* Pointeur parcours du mot */
  tree_t **arbre = t;     /* Pointeur parcours de l'arbre */
  short int existe = 1;   /* Booleen d'existence de lettre */

  /* Avance dans l'arbre tant que le debut du mot y est present */
  while (existe && *arbre && !isupper(*cour)) {
    arbre = rech_prec(arbre,*cour,&existe);
    if (existe) {
      arbre = &((*arbre)->lv);  /* va sur l'adresse du fils */
      cour++;                   /* Consommation du caractere */
    }
  }
  /* Test derniere lettre sensible a la casse pour indiquer la presence */
  if (*arbre && isupper(*cour)) {
    /* Recherche d'un hypothetique point d'insertion */
    arbre = rech_prec(arbre,*cour,&existe);
    if ((*arbre)->letter == *cour) {
      cour++;       /* Consommation du caractere */
    }
  }
  *w = cour; /* Mise a jour de la position des caracteres non encore presents dans l'arbre */

  return arbre;
}

/*  int insererMot(tree_t **t, char *w)
Insere un mot dans le dictionnaire a la bonne place

  Entrees :
    tree_t **t : adresse du pointeur de tete du dictionnaire (arbre)
    char *w : chaine de caracteres (mot) a inserer

  Sortie :
    res : code d'erreur
          0 : probleme d'allocation ou d'insertion
          1 : aucun souci d'insertion
*/
int insererMot(tree_t **t, char *w) {
  int len;              /* Longueur du mot */
  int i;                /* Indice de parcours pour copie */
  int res = 1;          /* Code de retour */
  char *cour;           /* Copie du mot */
  tree_t *tmp;          /* Noeud temporaire de creation */
  tree_t **arbre = t;  /* Pointeur de parcours de l'arbre */

  if (*w != '\0') { /* Mot non vide */
    /* Traitement du mot */
    len = strlen(w);   /* Calcul longueur */
    cour = (char*) malloc ((len+1)*sizeof(char));
    if (cour) {        /* Allocation ok */
      i = 0;
```

```
      while (w[i+1] != '\0') {
        cour[i] = tolower(w[i]);  /* Passage en minuscules */
        ++i;
      }
      cour[i] = toupper(w[i]);    /* Derniere lettre majuscule */
      cour[++i] = '\0';

      /* Recherche d'un debut deja present dans l'arbre */
      arbre = rech_mot(t,&cour);

      if (*cour != '\0') { /* Mot non deja present dans l'arbre */
        /* Insertion dans la liste chainee horizontale */
        if (*arbre && (*arbre)->letter == tolower(*cour)) {
          /* Derniere lettre deja existante, necessite de changer la casse */
          (*arbre)->letter = *cour;  /* Passage en majuscule pour ajouter le mot */
          cour++;                    /* Consommation du dernier caractere */
        } else {
          /* Insertions necessaires */
          /* Ajout de lien horizontal */
          tmp = creerNoeud(*cour);
          if (tmp) {                 /* Noeud cree */
            adj_cell(arbre,tmp);     /* Insertion lien horizontal */
            arbre = &((*arbre)->lv); /* Pointeur sur noeud fils */
            cour++;                  /* Lettre suivante */

            /* Insertion des lettres restantes selon des liens verticaux */
            while (res && *cour != '\0') {
              tmp = creerNoeud(*cour);
              if (tmp) {                 /* Noeud cree */
                *arbre = tmp;            /* Implantation du nouveau noeud */
                arbre = &((*arbre)->lv); /* Pointeur sur noeud fils */
                cour++;                  /* Lettre suivante */
              } else {                   /* Noeud non cree */
                res = 0;
              }
            }
          } else {                   /* Noeud non cree */
            res = 0;
          }
        }
      }
      free(cour-len); /* Liberation a partir du pointeur sur le debut du mot */
    } else {          /* Allocation ratee */
      res = 0;
    }
  }
  return res;
}

/*  void rech_motif(tree_t **t, char *w)
Affiche tous les mots commencant par un certain motif dans l'arbre

  Entrees :
    tree_t **t : adresse du pointeur sur l'arbre
    char *w : chaine de caracteres representant le motif a rechercher
```

```
  Sortie :
    Aucune
*/
void rech_motif(tree_t **t, char *w) {
    tree_t **arbre = t;
    char *cour = w;

    arbre = rech_mot(t, &cour); /* Recherche jusqu'a la fin du motif */
    if (*cour == '\0') {        /* On a trouve tout le motif */
      afficherArbrePref(*arbre, w);
    }
}
```

2.4 Programme principal

Code C

```c
/*  main.c
  Fonction principale du programme, pour les tests

  -----| ARBRES |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/
#include "tree.h"
#define COLUMNS 401 /* Nombre max de caracteres par ligne + 1 */

void test();


int main(int argc, char *argv[]) {
  FILE *f;
  char buf[COLUMNS];
  char text[COLUMNS];
  tree_t *arbre = NULL;
  if (argc > 1) {
    /* Lecture de fichier de commandes */
    f = fopen(argv[1],"r");
    if (f) {
      while (!feof(f)) {
        buf[0] = '\0';
        text[0] = '\0';
        fgets(buf,COLUMNS,f);
        switch (buf[0]) {
          case 'C': /* creer */
            sscanf(&buf[1],"%s",text);
            creerArbre(text,&arbre);
            break;
          case 'I': /* inserer */
            sscanf(&buf[1],"%s",text);
            insererMot(&arbre,text);
            break;
          case 'M': /* motif */
            sscanf(&buf[1],"%s",text);
            rech_motif(&arbre,text);
            break;
          case 'L': /* liberer */
            libererArbre(&arbre);
            break;
          case 'A': /* afficher */
            afficherArbre(arbre);
            break;
          case '#': /* texte */
            printf("%s",&buf[1]);
            break;
          default:
            puts("\n");
```

```
        }
      }
      if (arbre) {
        libererArbre(&arbre);
      }
      fclose(f);
    } else {
      fprintf(stderr, "Fichier invalide\n");
    }
  } else {
    /* Fonctions de tests de base */
    test();
  }
  return 0;
}
```

Code C

```c
void test() {
  tree_t *monArbre = NULL;
  int i, nbMotifs = 4;
  char *motif[] = {"a","","az","x"};
  printf("##########\nDEBUT DU PROGRAMME DE TEST\n##########\n");

  if (creerArbre("(a(l(p(h(A)))b(r(a(v(E,O))))c(h(a(r(l(i(E))))))d(e(l(t(A(p(l(a(n(E)))))))))e(c(h(O))p(i(n(E))))g(o(l(F)))h(o(t(e(L))))i(n(d(i(A,e(N))))))", &monArbre)) {
    printf("**********\nAffichage avant insertion\n");
    afficherArbre(monArbre);

    insererMot(&monArbre,"ALPHABET");   /* Debut deja present */
    insererMot(&monArbre,"foxtrot");   /* Aucune lettre deja presente */
    insererMot(&monArbre,"echo");      /* Mot deja present */
    insererMot(&monArbre,"epi");     /* Mot inclus dans un mot deja present */
    insererMot(&monArbre,"");         /* Mot vide */

    printf("**********\nAffichage apres insertion\n");
    afficherArbre(monArbre);

        for (i = 0; i < nbMotifs; ++i) {
      printf("**********\nRecherche du motif \"%s\"\n", motif[i]);
      rech_motif(&monArbre, motif[i]);
        }

    printf("**********\nInsertion dans l'arbre vide\n");
    libererArbre(&monArbre);
    insererMot(&monArbre,"alpha");

    printf("**********\nAffichage apres insertion\n");
    afficherArbre(monArbre);

    printf("**********\nLiberation de l'arbre\n");
    libererArbre(&monArbre);

    printf("**********\nAffichage apres liberation\n");
    afficherArbre(monArbre);

    printf("##########\nFIN DU PROGRAMME DE TEST\n##########\n");
  } else {
    fprintf(stderr, "Probleme creation arbre\n");
  }
}
```

3 | Principes et lexiques des fonctions

Dans cette partie, sont décrits les algorithmes de principe associés aux fonctions écrites en langage C, ainsi qu'un lexique concernant les variables intermédiaires des fonctions.

Les lexiques des variables d'entrée, sortie et entrée/sortie sont disponibles dans le code source directement.

3.1 Gestion de la pile

La gestion de la pile s'effectue grâce aux fichiers `stack.c` et `stack.h`. Les algorithmes de principe des différentes fonctions ont été précédement détaillés dans le TP2, nous ne les détaillerons donc pas.

3.2 Gestion des listes chaînées

Les fonctions de gestion des listes chaînées peuvent être trouvées dans les fichiers `list.c` et `list.h`. Les algorithmes de principe de ces fonctions ont également été fourni dans le TP1. Ils ne seront donc pas inclus ici.

3.3 Gestion de l'arbre

La gestion de l'arbre s'effectue avec les fonctions contenues dans `tree.c` et `tree.h`.

3.3.1 creerArbre

```
Algorithme creerArbre (Principe)
Début
    Initialise le code d'erreur à 0;
    Initialise caractère cour, au début de la chaîne;
    Initialise pointeur prec, de parcours à la racine;
    Initialisation de la pile;
    Si l'initialisation de la pile est réussie Alors
        Code d'erreur passe à 1;
        TantQue Code d'erreur = 1 Et  (Pile non vide Ou caractere courant ≠ ')') Faire
            Si cour = ')' Alors
                Empiler l'adresse du pointeur de parcours;
                prec passe sur le lien vertical;
                Avance d'un caractère dans la chaîne;
            Sinon
                Si cour = ',' Alors
                    prec passe sur le lien horizontal;
                    Avance d'un caractère dans la chaîne;
                FinSi;
                On crée un nœud à l'adresse prec, avec le caractère courant;
                Si l'allocation a échoué Alors
                    Code d'erreur passe à 0;
                Sinon
                    Avance au caractère suivant dans la chaîne;
                FinSi;
                TantQue Code d'erreur = 1 Et  Pile non vide Et  cour = ')' Faire
                    On dépile dans un pointeur temporaire;
                    prec devient pointeur sur l'adresse du lien horizontal de ce que l'on vient de dépiler;
                    Avance au caractère suivant dans la chaîne;
                FinTantQue;
            FinTantQue;
            Libération de la pile;
        FinSi;
        Retourner Code d'erreur;
    Fin
```

Lexique :

- p : *pile*
- **prec : *adresse du pointeur de parcours de l'arbre*
- tmp : *pointeur temporaire lorsque l'on dépile*
- cour : *pointeur sur le caractère courant dans la chaîne*
- taille : *taille max de la pile (taille de la chaîne de catactères)*
- ret : *code d'erreur (1 si tout va bien, 0 sinon)*

3.3.2 creerNoeud

```
Algorithme creerNoeud (Principe)
Début
    Allocation d'un nouvel élément;
    Si allocation réussie Alors
        Le lien vertical de l'élément est NIL;
        Le lien horizontal de l'élément est NIL;
        La valeur de l'élément prend la valeur du paramètre;
    FinSi;
    Retourner le nouvel élément créé;
Fin
```

Lexique :

- *r : *nouvel élément créé*

3.3.3 afficherArbrePref

```
Algorithme afficherArbrePref (Principe)
Début
    Initialisation de la pile;
    Initialisation d'un pointeur cour, de parcours de l'arbre;
    Si cour ≠ NIL Et  pile allouée Alors
        Répéter
            TantQue cour ≠ NIL Faire
                Empiler cour;
                Si la lettre dans cour est majuscule Alors   [ fin de mot ]
                    Affiche le préfixe donné en paramètre;
                    Affiche le contenu de la pile;
                    Affiche un retour à la ligne;
                FinSi;
            FinTantQue;
            TantQue pile non vide Et  cour = NIL Faire
                Dépiler dans cour;
                cour passe sur son lien horizontal;
            FinTantQue;
        TantQue pile non vide Ou cour ≠ NIL fait;
        Libération de la pile;
    FinSi;
Fin
```

Lexique :

- p : *pile*
- *cour : *pointeur de parcours de l'arbre*

3.3.4 afficherArbre

Ici, on appelle simplement la fonction précédente avec un préfixe valant la chaîne vide.

3.3.5 afficherPoint

Cette fonction affiche simplement la valeur d'un élément en convertissant le caractère en minuscules

3.3.6 libererArbre

```
Algorithme libererArbre (Principe)
Début
    Initialisation d'une pile;
    Initialisation d'un pointeur cour, sur la tête de l'arbre;
    Si cour ≠ NIL Et  Pile initialisée Alors
        Répéter
            TantQue cour ≠ NIL Faire
                Place la valeur cour dans un pointeur temporaire;
                Si Lien horizontal de cour ≠ NIL Alors
                    Empile l'adresse dans la pile;   [ Sauvegarde pour y revenir ]
                FinSi;
                Passe au lien vertical de cour;
                Libération de l'élément contenu dans le pointeur temporaire;
            FinTantQue;
            Si Pile non vide Alors   [ Il reste des éléments à libérer ]
                Dépile dans cour;
            FinSi;
        TantQue Pile non vide Ou cour ≠ NIL fait;
        Libération de la pile;
    FinSi;
    Mise du pointeur de tête de l'arbre sur NIL;
Fin
```

```
Lexique :
    p : pile
    *cour : pointeur de parcours de l'arbre
    *tmp : pointeur temporaire servant à libérer les éléments
```

3.3.7 rech_mot

```
Algorithme rech_mot (Principe)
Début
    Initialisation d'un pointeur cour, sur le caractère courant du mot;
    Initialisation d'un pointeur arbre, de parcours de l'arbre;
    Initialisation d'un booléen existe;   [ Initialisé à Vrai ]
    TantQue existe = Vrai Et  arbre ≠ NIL Et Alors  cour est en minuscule Faire   [ Recherche du début du mot ]
        Lance rech_prec() et stocke les résultats dans arbre et existe;
        Si existe = Vrai Alors
            On passe arbre sur l'adresse de son fils;
            Avance d'un caractère dans le mot;
        FinSi;
    FinTantQue;
    Si arbre ≠ NIL Et  cour en majuscule Alors
        Lance rech_prec() et stocke les résultats dans arbre et existe;
        Si la valeur du nœud courant = cour Alors
            Avance d'un caractère dans le mot;
        FinSi;
    FinSi;
    Le mot en paramètre prend la valeur de cour;   [ Ne contiendra que les lettres du mot non traitées ]
    Retourner arbre;
Fin
```

Lexique :

- *cour : *pointeur de parcours du mot en paramètre*
- **arbre : *double pointeur de parcours de l'arbre*
- existe : *booléen d'existence du caractère courant du mot dans l'arbre*

3.3.8 insererMot

```
Algorithme insererMot (Principe)
Début
    Initialisation d'un pointeur arbre à la tête;
    Initialisation d'un code d'erreur à 1;
    Si Le mot n'est pas vide Alors
        On crée une copie du mot entré;
        Mise de chaque lettre de la copie en minuscule, sauf la dernière (en majuscule);
        [ Avec un simple Tant Que ]
        Lance rech_mot() et stocke les résultats dans arbre et cour;
        Si Non fin de chaîne Alors   [ Le mot n'est pas présent ]
            Si arbre non vide Et Alors  valeur du nœud = cour(en minuscule) Alors
                On passe la lettre dans arbre en majuscule;
                Avance d'un caractère dans le mot;
            Sinon
                Création d'un nouveau nœud contenant la lettre courante;
                Si élément correctement créé Alors
                    Ajout de l'élément dans le lien horizontal de arbre;
                    On passe arbre sur l'adresse du pointeur de son fils;
                    Avance d'un caractère dans le mot;
                    TantQue code d'erreur ≠ 0 Et  mot non fini Faire   [ Insère les lettres restantes ]
                        Création d'un nouveau nœud avec cour comme valeur;
                        Si élément correctement créé Alors
                            Ajout de l'élément dans le lien vertical de arbre;
                            On passe arbre sur l'adresse du pointeur de son fils;
                            On avance d'un caractère dans le mot;
                        Sinon
                            Code d'erreur passe à 0;   [ Problème d'allocation nœud ]
                        FinSi;
                    FinTantQue;
                Sinon
                    Code d'erreur passe à 0;   [ Problème d'allocation nœud ]
                FinSi;
            FinSi;
        FinSi;
    FinSi;
    Retourner Code d'erreur;
Fin
```

Lexique :

- len : *taille du mot en paramètre*
- i : *indice de boucle pour copie du mot*
- res : *code d'erreur (1 si tout s'est bien passé, 0 sinon)*
- *cour : *pointeur sur caractère courant du mot (copié)*
- *tmp : *pointeur temporaire pour la création de nouveaux nœuds*
- **arbre : *pointeur de parcours de l'arbre*

3.3.9 rech_motif

Algorithme rech_motif (Principe)
Début

- Initialisation d'un pointeur arbre, sur la tête;
- Initialisation d'un pointeur cour, sur le caractère courant du mot;
- Lance `rech_mot()` et stocke les résultats dans arbre et cour;
- Si le motif est présent dans l'arbre Alors
  - Affichage de l'arbre avec pour préfixe le motif entré;
- FinSi;

Fin

Lexique :

- **arbre : *pointeur de parcours de l'arbre*
- *cour : *pointeur sur le caractère courant de l'arbre*

4 | Compte rendu d'exécution

4.1 Makefile

Makefile

```
#Compilateur et options de compilation
CC=gcc
CFLAGS=-Wall -ansi -pedantic -Wextra -g

#Fichiers du projet
SOURCES=main.c stack.c tree.c list.c
OBJECTS=$(SOURCES:.c=.o)

EXEC=prog

$(EXEC): $(OBJECTS)
	$(CC) $(CFLAGS) $^ -o $(EXEC)

.c.o:
	$(CC) -c $(CFLAGS) $*.c

clean:
	rm -rf $(OBJECTS) $(EXEC)
```

4.2 Jeux de tests

4.2.1 Fichier de tests

Le programme est exécuté avec le fichier `tests/test_complet` :

Test

```
#Test de creation
C (a(b(a(T))r(b(r(E))T(S)))b(a(i(l(l(e(R))))R(r(E)))i(e(N))o(s(s(E,U))))f(a(i(M,r(E))))j(o(u(e(T,u(R))))k(
#Affichage apres creation
A

#Insertion d'un mot dont aucune partie n'est presente dans le dictionnaire: toto
I toto
#Affichage suite a insertion
A

#Insertion d'un mot dont le debut est deja present: arbore
```

```
I arbore
#Affichage suite a insertion
A

#Insertion d'un mot inclus dans un mot deja present: bail
I bail
#Affichage suite a insertion
A

#Insertion d'un mot deja existant: barre
I barre
#Affichage suite a insertion
A

#Insertion du mot vide (doit etre sans effet)
I
#Affichage suite a insertion
A

#Recherche de motif: cas mots existants: mots commencant par ba
M ba

#Recherche de motif: cas de mots inexistants: mots commencant par x
M x

#Recherche de motif: recherche du motif vide (= contenu de l'arbre)
M

#Suppression de l'arbre
L
#Affichage suite a suppression (arbre vide)
A

#Insertion dans un arbre vide
I moi
#Affichage suite a insertion
A
L
```

On obtient alors le résultat suivant :

Resultat

```
Test de creation
Affichage apres creation
abat
arbre
art
arts
bailler
bar
barre
bien
bosse
bossu
faim
faire
jouet
joueur
kimono
kiwi
mas
masse
vain
yack
zebu


Insertion d'un mot dont aucune partie n'est
 ↪  presente dans le dictionnaire: toto
```

```
Affichage suite a insertion
abat
arbre
art
arts
bailler
bar
barre
bien
bosse
bossu
faim
faire
jouet
joueur
kimono
kiwi
mas
masse
toto
vain
yack
zebu


Insertion d'un mot dont le debut est deja
 ↪  present: arbore
Affichage suite a insertion
abat
arbore
arbre
art
arts
bailler
bar
barre
bien
bosse
bossu
faim
faire
jouet
joueur
kimono
kiwi
mas
masse
toto
vain
yack
zebu


Insertion d'un mot inclus dans un mot deja
 ↪  present: bail
Affichage suite a insertion
abat
arbore
arbre
art
arts
bail
bailler
bar
barre
bien
bosse
bossu
faim
faire
jouet
joueur
kimono
kiwi
mas
masse
toto
vain
yack
zebu


Insertion d'un mot deja existant: barre
Affichage suite a insertion
abat
arbore
arbre
art
arts
bail
bailler
bar
barre
bien
bosse
bossu
faim
faire
jouet
joueur
kimono
kiwi
mas
masse
toto
vain
yack
zebu


Insertion du mot vide (doit etre sans effet)
Affichage suite a insertion
```

```
abat
arbore
arbre
art
arts
bail
bailler
bar
barre
bien
bosse
bossu
faim
faire
jouet
joueur
kimono
kiwi
mas
masse
toto
vain
yack
zebu


Recherche de motif: cas mots existants: mots
 ↪  commencant par ba
bail
bailler
bar
barre


Recherche de motif: cas de mots inexistants:
 ↪  mots commencant par x
```

```

Recherche de motif: recherche du motif vide
 ↪  (= contenu de l'arbre)
abat
arbore
arbre
art
arts
bail
bailler
bar
barre
bien
bosse
bossu
faim
faire
jouet
joueur
kimono
kiwi
mas
masse
toto
vain
yack
zebu


Suppression de l'arbre
Affichage suite a suppression (arbre vide)


Insertion dans un arbre vide
Affichage suite a insertion
moi
```

Pour ce fichier de tests, nous avons effectué un suivi avec l'outil ddd.

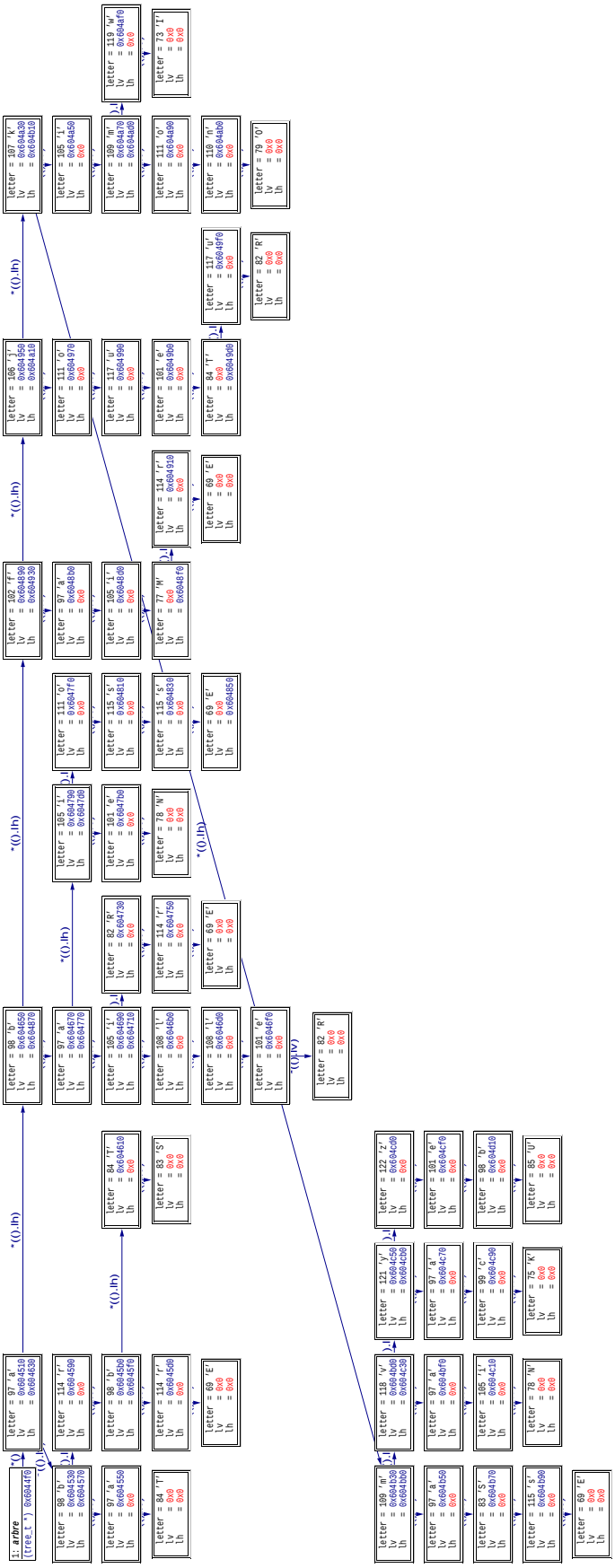

FIGURE 4.1 – Création de l'arbre

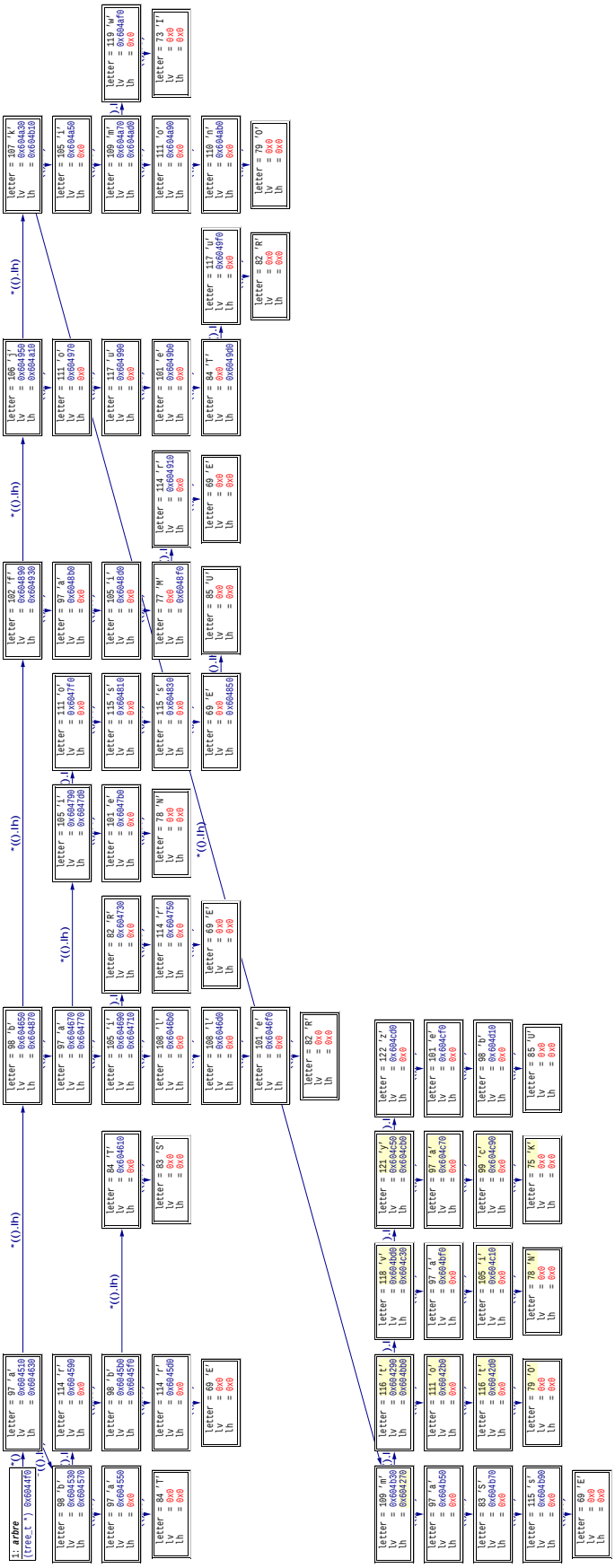

FIGURE 4.2 – Insertion de "toto"

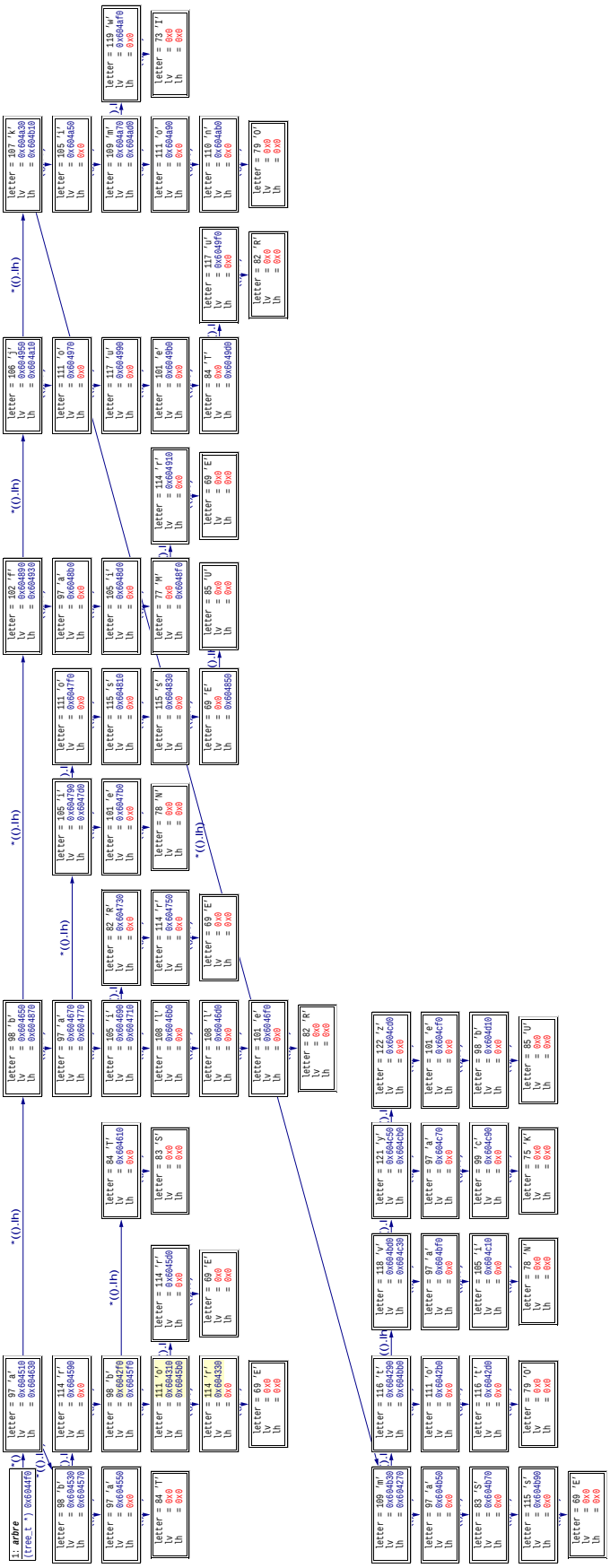

FIGURE 4.3 – Insertion de "arbore"

FIGURE 4.4 – Insertion de "bail"

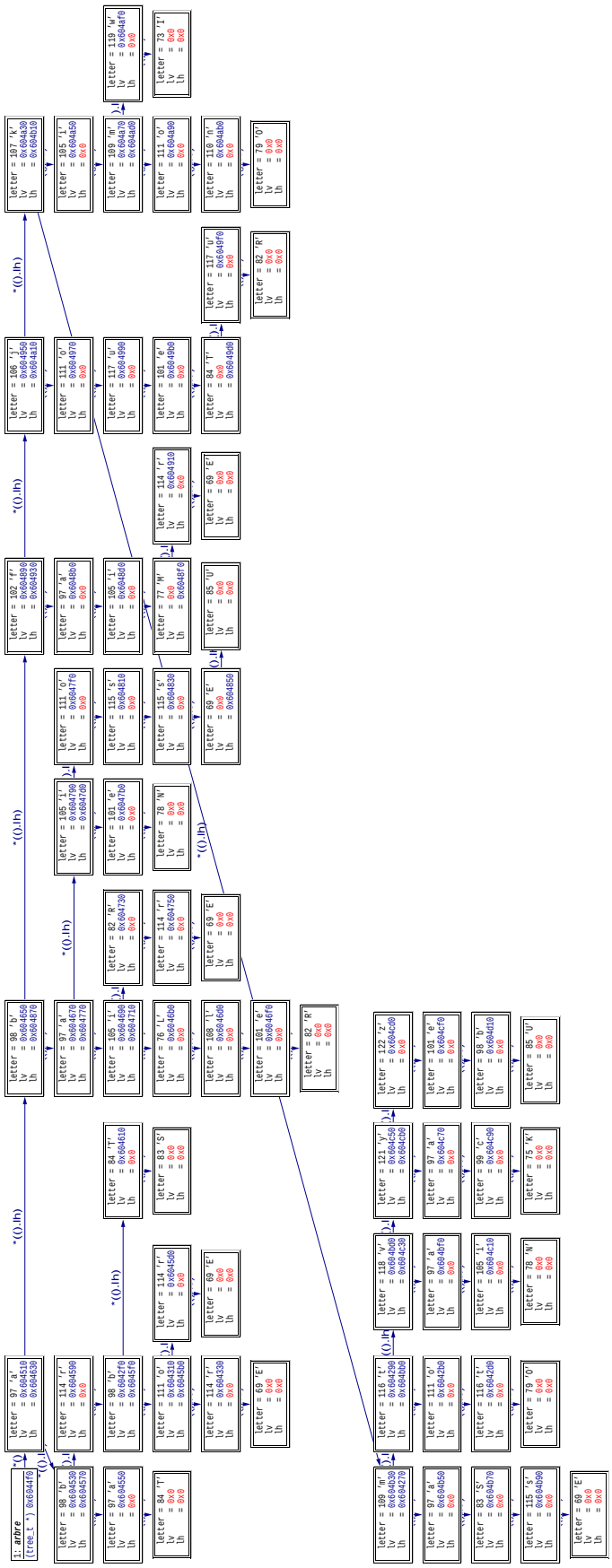

FIGURE 4.5 – Insertion de "barre"

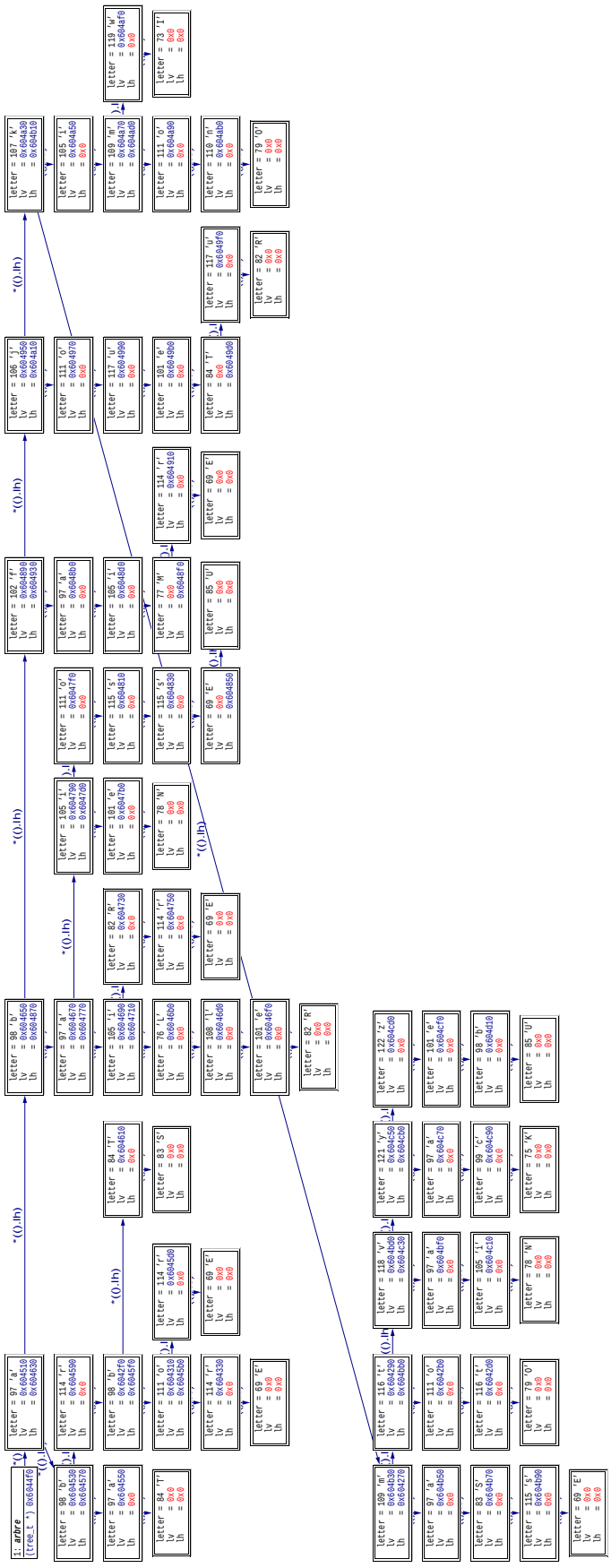

FIGURE 4.6 – Insertion du mot vide

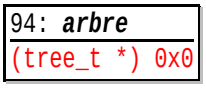

FIGURE 4.7 – Suppression de l'arbre

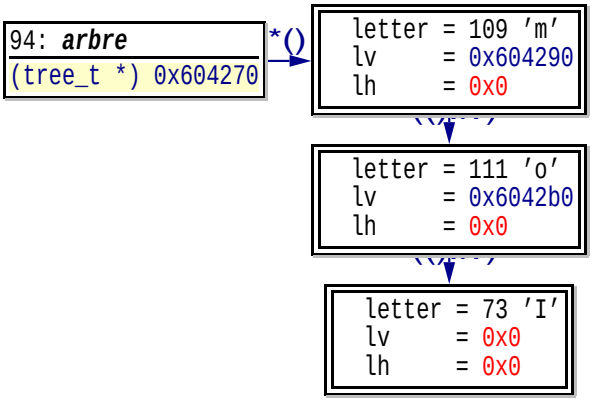

FIGURE 4.8 – Insertion de "moi"

Le programme est exécuté avec le fichier `tests/test_creation` :

Test

```
#CREATION ARBRE
C (a(b(a(T))r(b(r(E))T(S)))b(a(i(L(l(e(R))))R(r(e)))i(e(N))o(s(s(E,U))))f(a(i(M,r(E))))j(o(u(e(T,u(R))))k(
A
L
#CREATION ARBRE VIDE
C ()
A
L
#CREATION ARBRE A 1 SEUL POINT
C (A)
A
L
#CREATION ARBRE A LIENS VERTICAUX SEULEMENT (= 1 MOT)
C (m(a(n(g(e(R))))))
A
L
#CREATION ARBRE A LIENS HORIZONTAUX SEULEMENT (MOTS D'UNE SEULE LETTRE)
C (A,Y)
A
L
```

On obtient alors le résultat suivant :

Resultat

```
CREATION ARBRE
abat
arbre
art
arts
bail
bailler
bar
bien
bosse
bossu
faim
faire
jouet
joueur
kimono
kiwi
mas
masse
vain
yack
zebu
CREATION ARBRE VIDE
CREATION ARBRE A 1 SEUL POINT
a
CREATION ARBRE A LIENS VERTICAUX SEULEMENT (= 1 MOT)
manger
CREATION ARBRE A LIENS HORIZONTAUX SEULEMENT (MOTS D'UNE SEULE LETTRE)
a
y
```

Le programme est exécuté avec le fichier `tests/test_insertion` :

Test

```
#ARBRE VIDE
A
#INSERTION DANS L'ARBRE VIDE: toto
I toto
A
#INSERTION 2: barre
I barre
A
#INSERTION MOT EXISTANT: toto
I toto
A
#INSERTION MOT VIDE:
I
A
#INSERTION MOT INCLUS: bar
I bar
A
#INSERTION MOT A DEBUT COMMUN 1: barreau
I barreau
A
#INSERTION MOT A DEBUT COMMUN 2: barriere
I barriere
A
L
```

On obtient alors le résultat suivant :

Resultat

```
ARBRE VIDE
INSERTION DANS L'ARBRE VIDE: toto
toto
INSERTION 2: barre
barre
toto
INSERTION MOT EXISTANT: toto
barre
toto
INSERTION MOT VIDE:
barre
toto
INSERTION MOT INCLUS: bar
bar
barre
toto
INSERTION MOT A DEBUT COMMUN 1: barreau
bar
barre
barreau
toto
INSERTION MOT A DEBUT COMMUN 2: barriere
bar
barre
barreau
barriere
toto
```

Le programme est exécuté avec le fichier `tests/test_motif` :

Test

```
C (a(b(a(T))r(b(r(E))T(S)))b(a(i(L(l(e(R))))R(r(e)))i(e(N))o(s(s(E,U))))f(a(i(M,r(E))))j(o(u(e(T,u(R))))k(
#ARBRE UTILISE
A
#
#MOTIF: ba
M ba
#
#MOTIF VIDE: ""
M
#
#MOTIF MOT: "bail"
M bail
#
#MOTIF ABSENT: "kh"
M kh
L
```

On obtient alors le résultat suivant :

Resultat

```
ARBRE UTILISE
abat
arbre
art
arts
bail
bailler
bar
bien
bosse
bossu
faim
faire
jouet
joueur
kimono
kiwi
mas
masse
vain
yack
zebu

MOTIF: ba
bail
bailler
bar

MOTIF VIDE: ""
abat
arbre
art
arts
bail
bailler
bar
bien
bosse
bossu
faim
faire
jouet
joueur
kimono
kiwi
mas
masse
vain
yack
zebu

MOTIF MOT: "bail"
bailler

MOTIF ABSENT: "kh"
```

4.2.2 Bonne utilisation de la mémoire

Pour vérifier la bonne libération de la mémoire, nous avons utilisé `valgrind` avec le programme seul et le fichier de test complet. Aucun bloc de mémoire n'est perdu, le retour texte de valgrind est présenté en figure 4.9.

Resultat Valgrind

```
==13196== Memcheck, a memory error detector
==13196== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==13196== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==13196== Command: ../src/prog test_complet
==13196==
==13196==
==13196== HEAP SUMMARY:
==13196==     in use at exit: 0 bytes in 0 blocks
==13196==   total heap usage: 94 allocs, 94 frees, 21,619 bytes allocated
==13196==
==13196== All heap blocks were freed -- no leaks are possible
==13196==
==13196== For counts of detected and suppressed errors, rerun with: -v
==13196== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

FIGURE 4.9 – Passage dans l'outil `valgrind`