

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



Table des matières

1	Présentation	2
1.1	Structure de données employée	2
1.2	Organisation du code source	3
1.2.1	Gestion de la pile	3
1.2.2	Gestion des listes chaînées	3
1.2.3	Gestion de l'arbre	3
1.2.4	Programme principal	3
2	Détails du programme	4
2.1	Gestion de la pile	4
2.2	Gestion des listes chaînées	8
2.3	Gestion de l'arbre	11
2.4	Programme principal	18
3	Principes et lexiques des fonctions	21
3.1	Gestion de la pile	21
3.2	Gestion des listes chaînées	21
3.3	Gestion de l'arbre	21
3.3.1	creerArbre	22
3.3.2	creerNoeud	23
3.3.3	afficherArbrePref	23
3.3.4	afficherArbre	24
3.3.5	afficherPoint	24
3.3.6	libererArbre	24
3.3.7	rech_mot	25
3.3.8	insererMot	26
3.3.9	rech_motif	27
4	Compte rendu d'exécution	28
4.1	Makefile	28
4.2	Jeux de tests	28
4.2.1	Fichier de tests	28
4.2.2	Bonne utilisation de la mémoire	42

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 :

- Créer 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 ordre alphabétique en lecture préfixe. La fin d'un mot est signalée par une lettre majuscule.

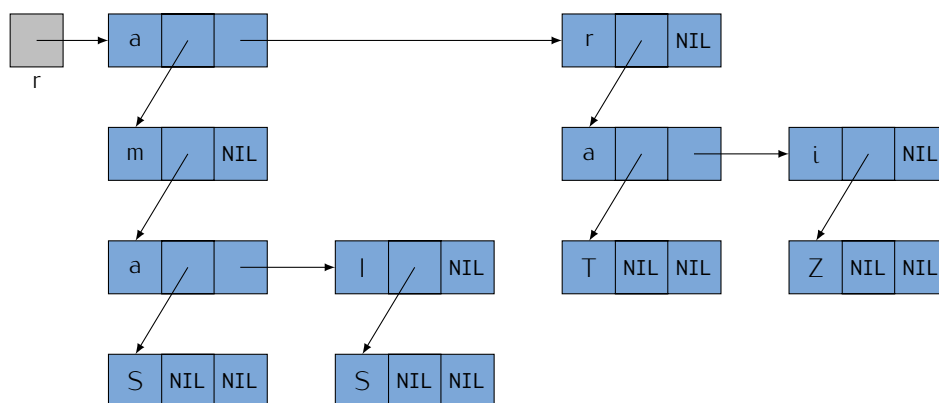


FIGURE 1.1 – Représentation en liens vertical et horizontal

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 après 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

```
1  /* stack.h
2  Header
3
4  -----| PILE |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11
12 #ifndef __STACK__H
13 #define __STACK__H
14
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include "tree.h"
18
19 typedef tree_t* datatype;
20
21 typedef struct _stack_t {
22     int      max; /* nombre max d'elements dans la pile */
23     int      top; /* position de l'element en tete de pile */
24     datatype *val; /* tableau des valeurs de la pile */
25 } stack_t;
26
27 int init(stack_t *,int);
28 void supp(stack_t *);
29 int empty(stack_t);
30 int full(stack_t);
31 int pop(stack_t *, datatype *);
32 int top(stack_t *, datatype *);
33 int push(stack_t *, datatype);
34 void dump(stack_t, void (*)(datatype));
35
36 #endif
```

```

1  /* stack.c
2  Fonctions de gestion de la structure de pile
3
4  -----| PILE |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11 #include "stack.h"
12
13 /* int init(stack_t *p, int n)
14 Fonction d'initialisation de la pile avec une taille max
15
16 Entrees :
17     *p : pointeur sur la pile
18     n : taille maximum de la pile
19
20 Sortie :
21     int : code d'erreur
22         1 si aucune erreur
23         0 si erreur de creation de la pile
24 */
25 int init(stack_t *p, int n) {
26     int ret = 1;
27     p->max = n;
28     p->top = -1;
29     p->val = (datatype*) malloc(n*sizeof(datatype));
30     if (p->val == NULL) {
31         ret = 0;
32     }
33     return ret;
34 }
35
36 /* void supp(stack_t *p)
37 Fonction de suppression de la pile
38
39 Entree :
40     *p : pointeur sur la tete de la pile
41
42 Sortie :
43     Aucune
44 */
45 void supp(stack_t *p) {
46     free(p->val);
47     p->top = -1; /* Empeche de depiler */
48     p->max = 0; /* Empeche d'empiler */
49 }
50
51 /* int empty(stack_t *p)
52 Teste si la pile est vide ou non
53
54 Entree :
55     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

```

```

112     0 si rien n'est retourne
113     1 si on recupere l'element en tete
114 */
115 int top(stack_t *p, datatype *v) {
116     int ok = 0;
117     if (!empty(*p)) {
118         *v = p->val[p->top];
119         ok = 1;
120     }
121     return ok;
122 }
123
124 /* int push(stack_t *p, datatype v)
125 Insere un element en tete de la pile
126
127 Entree :
128     *p : pointeur sur la tete de la pile
129     v : element a inserer dans la pile
130
131 Sortie :
132     int : code d'erreur
133         0 si l'element n'est pas ajoute dans la pile
134         1 si l'element est ajoute dans la pile
135 */
136 int push(stack_t *p, datatype v) {
137     int ok = 0;
138     if (!full(*p)) {
139         p->top++;
140         p->val[p->top] = v;
141         ok = 1;
142     }
143     return ok;
144 }
145
146 /* void dump(stack_t p, void (*afficherData)(datatype))
147 Affiche le contenu de la pile
148
149 Entree :
150     p : tete de la pile
151     *afficherData : pointeur de fonction permettant l'affichage des elements
152
153 Sortie :
154     Aucune
155 */
156 void dump(stack_t p, void (*afficherData)(datatype)) {
157     int i;
158     if (!empty(p)) {
159         for (i = 0; i <= p.top; i++) {
160             afficherData(p.val[i]);
161         }
162     }
163 }

```


2.2 Gestion des listes chaînées

Code C

```
1  /* list.h
2  Header
3
4  -----| LISTE CHAINEE |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11
12 #ifndef __LISTE_H__
13 #define __LISTE_H__
14
15 #include <string.h>
16 #include <ctype.h>
17
18 typedef struct _node_t {
19     char letter;
20     struct _node_t *lv;
21     struct _node_t *lh;
22 } node_t;
23
24 typedef node_t cell_t;
25
26 void adj_cell(cell_t **, cell_t *);
27 cell_t ** rech_prec(cell_t **, char, short int*);
28 void supp_cell(cell_t **);
29 void liberer_liste(cell_t **);
30 void ins_cell(cell_t **, cell_t *);
31 cell_t * creer_cell(char);
32
33 #endif
```

Code C

```
1  /* list.c
2  Fonctions de gestion de la liste chaine
3
4  -----| LISTE CHAINEE |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include "list.h"
15
16 /* void adj_cell(cell_t **prec, cell_t *elt)
17 Ajoute une cellule apres un element partir d'un pointeur sur l'element
18 et d'un pointeur sur le pointeur de l'element apres lequel ajouter
```

```

19
20 Entrees :
21     cell_t **prec : pointeur sur le pointeur de l'element apres lequel ajouter
22     cell_t *elt : pointeur sur l'element a ajouter a la liste chaine
23
24 Sortie :
25     Aucune
26
27 /*
28 void adj_cell(cell_t **prec, cell_t *elt) {
29     elt->lh = (*prec);
30     (*prec) = elt;
31 }
32
33 /* cell_t ** rech_prec(cell_t **liste, char letter, short int *existe)
34 Recherche le precedent d'un element dans la liste chaine
35
36 Entrees :
37     cell_t **liste : pointeur sur le pointeur du premier element de la liste chaine
38     char lettre : caractere a chercher dans la liste
39     short int *existe : variable en entree/sortie indiquant la présence de la lettre
40     0 : absence
41     1 : presence
42
43 Sortie :
44     cell_t ** : pointeur sur le pointeur de l'element precedent
45
46 /*
47 cell_t ** rech_prec(cell_t **liste, char letter, short int *existe) {
48     cell_t **prec = liste;
49     while ((*prec) && tolower((*prec)->letter) < tolower(letter)) {
50         prec = &((*prec)->lh);
51     }
52     /* Booleen de presence      */
53     /* 1 : present              */
54     /* 0 : absent               */
55     *existe = (*prec && tolower((*prec)->letter) == tolower(letter))?1:0;
56     return prec;
57 }
58
59 /* void supp_cell(cell_t **prec)
60 Permet de supprimer un element dans la liste chaine a partir
61 de son precedent
62
63 Entrees :
64     cell_t **prec : pointeur sur le pointeur de l'element precedent l'element a supprimer
65
66 Sortie :
67     Aucune
68
69 /*
70 void supp_cell(cell_t **prec) {
71     cell_t *elt = *prec;
72     *prec = elt->lh;
73     free(elt);
74 }
75
76 /* void liberer_liste(cell_t **liste)
77 Libere les allocations memoires de la liste

```

```

75
76  Entrees :
77      cell_t **liste : pointeur sur le pointeur du premier element de la liste chaine
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'insérer une cellule a la bonne place dans la liste chaine
91
92  Entrees :
93      cell_t **liste : pointeur sur le pointeur du premier element de la liste chaine
94      cell_t *elt : pointeur sur l'element a insérer dans la liste chaine
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 créer un element de la liste chaine 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

```
1  /* tree.h
2  Header
3
4  -----| GESTION DU DICTIONNAIRE |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11
12 #ifndef __TREE__H
13 #define __TREE__H
14
15     #include <stdio.h>
16     #include <stdlib.h>
17     #include <string.h>
18     #include <ctype.h>
19     #include "list.h"
20     typedef node_t tree_t;
21     #include "stack.h"
22
23     #define DEBUG 0
24     #define STACKSIZE 200
25
26     int  creerArbre(char *, tree_t **);
27     void libererArbre(tree_t **);
28     void afficherArbrePref(tree_t *, char *);
29     void afficherArbre(tree_t *);
30     void afficherPoint(tree_t *);
31     int  insererMot(tree_t **, char *);
32     void rech_motif(tree_t **, char *);
33
34 #endif
```

Code C

```
1  /* tree.c
2  Fonction de gestion de l'arbre
3
4  -----| GESTION DU DICTIONNAIRE |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11
12 #include "tree.h"
13
14 tree_t *creerNoeud(char);
15
16 /* int creerArbre(char *ch, tree_t **r)
17 Cree un arbre a partir d'une chaine de caracteres representant la notation parenthesee
```

```

18 et en prenant l'adresse du pointeur sur la tete de l'arbre
19
20 Entrees :
21     char *ch : chaine de representation de l'arbre
22     tree_t **r : pointeur double de tête de l'arbre
23
24 Sortie :
25     int : code de retour sur la creation
26         0 : probleme d'alloc d'element
27         1 : aucun probleme
28 */
29 int creerArbre(char *ch, tree_t **r) {
30     stack_t p; /* Pile */
31     tree_t **prec = r; /* Pointeur de parcours de l'arbre */
32     tree_t *tmp; /* Pointeur temporaire */
33     char *cour = ch; /* Caractere courant */
34     int ret = 0; /* Variable de retour */
35
36     cour++; /* On consomme la premiere parenthese */
37     if (init(&p, STACKSIZE)) {
38         ret = 1; /* Allocation ok */
39         while (ret && (!empty(p) || *cour != ')')) { /* Aucun souci et chaine non-finie */
40             if (*cour == '(') { /* Ajout de fils */
41                 push(&p, *prec); /* Sauvegarde de l'adresse courant */
42                 prec = &((*prec)->lv); /* Deplacement sur le lien vertical */
43                 cour++; /* Accelération, passe au prochain caractere */
44             } else if (*cour == ',') { /* Ajout de frere */
45                 prec = &((*prec)->lh); /* Deplacement sur le lien horizontal */
46                 cour++;
47             }
48             *prec = creerNoeud(*cour);
49             if (!(*prec)) {
50                 ret = 0; /* Problème allocation */
51             } else {
52                 cour++; /* Passage caractere suivant */
53             }
54             while (ret && !empty(p) && *cour == ')') {
55                 pop(&p, &tmp); /* Recuperation du lien horizontal parent */
56                 prec = &(tmp->lh);
57                 cour++;
58             }
59         }
60         supp(&p); /* Liberation pile */
61     }
62     return ret;
63 }
64
65 /* tree_t *creerNoeud(char v)
66 Cree un noeud ayant pour valeur le caractere entre
67
68 Entrees :
69     char v : valeur du nouveau noeud cree
70
71 Sortie :
72     tree_t* : pointeur sur le nouvel element cree
73 */

```

```

74 tree_t *creerNoeud(char v) {
75     tree_t *r = (tree_t*) malloc (sizeof(tree_t));
76     if (r) {          /* Allocation OK */
77         r->lv = NULL;   /* Initialisation lien vertical */
78         r->lh = NULL;   /* Initialisation lien horizontal */
79         r->letter = v;  /* Initialisation valeur */
80     }
81     return r;
82 }
83
84 /* void afficherArbrePref(tree_t *t, char *prefixe)
85 Affiche les mots contenus dans l'arbre avec un prefixe donne en entre
86
87 Entrees :
88     tree_t *t : pointeur sur la tete de l'arbre
89     char *prefixe : prefixe a ecrire avant chaque mot de l'arbre
90
91 Sortie :
92     Aucune
93 */
94 void afficherArbrePref(tree_t *t, char *prefixe) {
95     stack_t p;          /* Pile */
96     tree_t *cour = t;   /* Pointeur de parcours de l'arbre */
97
98     if (cour != NULL && init(&p, STACKSIZE)) {
99         do {
100             while (cour != NULL) {
101                 push(&p, cour);          /* Sauvegarde du point courant */
102                 if (isupper(cour->letter)) { /* Detection fin de mot */
103                     printf("%s", prefixe); /* Affiche le prefixe */
104                     dump(p, afficherPoint); /* Affichage du mot (lecture pile) */
105                     printf("\n");
106                 }
107                 cour = cour->lv;          /* Deplacement sur le lien vertical */
108             }
109             /* On a atteint une feuille */
110             while (!empty(p) && cour == NULL) { /* Recherche du premier frere des ascendants */
111                 pop(&p, &cour);          /* Recuperation du parent */
112                 cour = cour->lh;          /* Deplacement sur le lien horizontal */
113             }
114         } while (!empty(p) || cour != NULL);
115         supp(&p);
116     }
117 }
118
119 /* void afficherArbrePref(tree_t *t)
120 Affiche les mots contenus dans l'arbre
121
122 Entrees :
123     tree_t *t : pointeur sur la tete de l'arbre
124
125 Sortie :
126     Aucune
127 */
128 void afficherArbre(tree_t *t) {
129     afficherArbrePref(t, "");

```

```

130 }
131
132 /* void afficherPoint(tree_t *t)
133 Affiche la valeur d'un noeud, connaissant son adresse
134
135 Entrees :
136     tree_t *t : pointeur sur le noeud
137
138 Sortie :
139     Aucune
140 */
141 void afficherPoint(tree_t *t) {
142     printf("%c", tolower(t->letter));
143 }
144
145 /* void libererArbre(tree_t **t)
146 Libere la memoire occupee par l'arbre
147
148 Entrees :
149     tree_t **t : adresse du pointeur sur la tete de l'arbre
150
151 Sortie :
152     Aucune
153 */
154 void libererArbre(tree_t **t) {
155     stack_t p; /* Pile */
156     tree_t *cour = *t; /* Pointeur de parcours de l'arbre */
157     tree_t *tmp; /* Pointeur temporaire pour conserver l'adresse du point a ←
    ↪ supprimer */
158
159     if (cour != NULL && init(&p, STACKSIZE)) {
160         do {
161             while (cour != NULL) {
162                 tmp = cour; /* Sauvegarde du courant */
163                 if (cour->lh != NULL) {
164                     push(&p, cour->lh); /* Sauvegarde du frere */
165                 }
166                 cour = cour->lv; /* Deplacement sur le lien vertical */
167                 free(tmp); /* Suppression du point courant */
168             }
169             if (!empty(p)) {
170                 pop(&p, &cour); /* Recuperation du premier lien horizontal parmi les ←
    ↪ parents */
171             }
172         } while (!empty(p) || cour != NULL);
173         supp(&p);
174     }
175     *t = NULL;
176 }
177
178 /* tree_t **rech_mot(tree_t **t, char **w)
179
180 Entrees :
181     **t : adresse du pointeur de tete de l'arbre
182     **w : pointeur sur le mot a chercher
183

```

```

184  Sortie :
185      tree_t ** : adresse du pointeur dans l'arbre ou on a trouve la derniere lettre
186                  possible du mot
187  */
188  tree_t **rech_mot(tree_t **t, char **w) {
189      char *cour = *w;          /* Pointeur parcour du mot */
190      tree_t **arbre = t;       /* Pointeur parcour de l'arbre */
191      short int existe = 1;      /* Booleen d'existence de lettre */
192
193      /* Avance dans l'arbre tant que le debut du mot y est present */
194      while (existe && *arbre && !isupper(*cour)) {
195          arbre = rech_prec(arbre, *cour, &existe);
196          if (existe) {
197              arbre = &((*arbre)->lv); /* va sur l'adresse du fils */
198              cour++;                    /* Consommation du caractere */
199          }
200      }
201      /* Test derniere lettre sensible a la casse pour indiquer la presence */
202      if (*arbre && isupper(*cour)) {
203          /* Recherche d'un hypothetique point d'insertion */
204          arbre = rech_prec(arbre, *cour, &existe);
205          if ((*arbre)->letter == *cour) {
206              cour++;                /* Consommation du caractere */
207          }
208      }
209      *w = cour; /* Mise a jour de la position des caracteres non encore presents dans l'arbre */
210
211      return arbre;
212  }
213
214  /* int insererMot(tree_t **t, char *w)
215  Insere un mot dans le dictionnaire a la bonne place
216
217  Entrees :
218      tree_t **t : adresse du pointeur de tete du dictionnaire (arbre)
219      char *w : chaine de caracteres (mot) a inserer
220
221  Sortie :
222      res : code d'erreur
223              0 : probleme d'allocation ou d'insertion
224              1 : aucun souci d'insertion
225  */
226  int insererMot(tree_t **t, char *w) {
227      int len;          /* Longueur du mot */
228      int i;            /* Indice de parcour pour copie */
229      int res = 1;      /* Code de retour */
230      char *cour;        /* Copie du mot */
231      tree_t *tmp;       /* Noeud temporaire de creation */
232      tree_t **arbre = t; /* Pointeur de parcour de l'arbre */
233
234      if (*w != '\0') { /* Mot non vide */
235          /* Traitement du mot */
236          len = strlen(w); /* Calcul longueur */
237          cour = (char*) malloc ((len+1)*sizeof(char));
238          if (cour) {      /* Allocation ok */
239              i = 0;

```



```

240 while (w[i+1] != '\0') {
241     cour[i] = tolower(w[i]); /* Passage en minuscules */
242     ++i;
243 }
244 cour[i] = toupper(w[i]); /* Derniere lettre majuscule */
245 cour[++i] = '\0';
246
247 /* Recherche d'un debut deja present dans l'arbre */
248 arbre = rech_mot(t,&cour);
249
250 if (*cour != '\0') { /* Mot non deja present dans l'arbre */
251     /* Insertion dans la liste chaine horizontale */
252     if (*arbre && (*arbre)->letter == tolower(*cour)) {
253         /* Derniere lettre deja existante, necessite de changer la casse */
254         (*arbre)->letter = *cour; /* Passage en majuscule pour ajouter le mot */
255         cour++; /* Consommation du dernier caractere */
256     } else {
257         /* Insertions necessaires */
258         /* Ajout de lien horizontal */
259         tmp = creerNoeud(*cour);
260         if (tmp) { /* Noeud cree */
261             adj_cell(arbre,tmp); /* Insertion lien horizontal */
262             arbre = &((*arbre)->lv); /* Pointeur sur noeud fils */
263             cour++; /* Lettre suivante */
264
265             /* Insertion des lettres restantes selon des liens verticaux */
266             while (res && *cour != '\0') {
267                 tmp = creerNoeud(*cour);
268                 if (tmp) { /* Noeud cree */
269                     *arbre = tmp; /* Implantation du nouveau noeud */
270                     arbre = &((*arbre)->lv); /* Pointeur sur noeud fils */
271                     cour++; /* Lettre suivante */
272                 } else { /* Noeud non cree */
273                     res = 0;
274                 }
275             }
276             } else { /* Noeud non cree */
277                 res = 0;
278             }
279         }
280     }
281     free(cour-len); /* Liberation a partir du pointeur sur le debut du mot */
282 } else { /* Allocation ratee */
283     res = 0;
284 }
285 }
286 return res;
287 }
288
289 /* void rech_motif(tree_t **t, char *w)
290 Affiche tous les mots commençant par un certain motif dans l'arbre
291
292 Entrees :
293     tree_t **t : adresse du pointeur sur l'arbre
294     char *w : chaine de caracteres representant le motif a rechercher
295

```

```

296     Sortie :
297     Aucune
298 */
299 void rech_motif(tree_t **t, char *w) {
300     tree_t **arbre = t;
301     char *cour = w;
302
303     arbre = rech_mot(t, &cour); /* Recherche jusqu'a la fin du motif */
304     if (*cour == '\\0') {        /* On a trouve tout le motif */
305         afficherArbrePref(*arbre, w);
306     }
307 }

```

2.4 Programme principal

Code C

```
1  /* main.c
2  Fonction principale du programme, pour les tests
3
4  -----| ARBRES |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11 #include "tree.h"
12 #define COLUMNS 401 /* Nombre max de caracteres par ligne + 1 */
13
14 void test();
15
16
17 int main(int argc, char *argv[]) {
18     FILE *f;
19     char buf[COLUMNS];
20     char text[COLUMNS];
21     tree_t *arbre = NULL;
22     if (argc > 1) {
23         /* Lecture de fichier de commandes */
24         f = fopen(argv[1], "r");
25         if (f) {
26             while (!feof(f)) {
27                 buf[0] = '\0';
28                 text[0] = '\0';
29                 fgets(buf, COLUMNS, f);
30                 switch (buf[0]) {
31                     case 'C': /* creer */
32                         sscanf(&buf[1], "%s", text);
33                         creerArbre(text, &arbre);
34                         break;
35                     case 'I': /* inserer */
36                         sscanf(&buf[1], "%s", text);
37                         insererMot(&arbre, text);
38                         break;
39                     case 'M': /* motif */
40                         sscanf(&buf[1], "%s", text);
41                         rech_motif(&arbre, text);
42                         break;
43                     case 'L': /* liberer */
44                         libererArbre(&arbre);
45                         break;
46                     case 'A': /* afficher */
47                         afficherArbre(arbre);
48                         break;
49                     case '#': /* texte */
50                         printf("%s", &buf[1]);
51                         break;
52                     default:
53                         puts("\n");
```

```

54     }
55 }
56 if (arbre) {
57     libererArbre(&arbre);
58 }
59 fclose(f);
60 } else {
61     fprintf(stderr, "Fichier invalide\n");
62 }
63 } else {
64     /* Fonctions de tests de base */
65     test();
66 }
67 return 0;
68 }

```

```

70 void test() {
71     tree_t *monArbre = NULL;
72     int i, nbMotifs = 4;
73     char *motif[] = {"a", "", "az", "x"};
74     printf("#####\nDEBUT DU PROGRAMME DE TEST\n#####\n");
75
76     if (creerArbre
77         ↪ ("(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,
78         ↪ &monArbre))) {
79         printf("*****\nAffichage avant insertion\n");
80         afficherArbre(monArbre);
81
82         insererMot(&monArbre, "ALPHABET"); /* Debut deja present */
83         insererMot(&monArbre, "foxtrot"); /* Aucune Lettre deja presente */
84         insererMot(&monArbre, "echo"); /* Mot deja present */
85         insererMot(&monArbre, "epi"); /* Mot inclus dans un mot deja present */
86         insererMot(&monArbre, ""); /* Mot vide */
87
88         printf("*****\nAffichage apres insertion\n");
89         afficherArbre(monArbre);
90
91         for (i = 0; i < nbMotifs; ++i) {
92             printf("*****\nRecherche du motif \"%s\"\n", motif[i]);
93             rech_motif(&monArbre, motif[i]);
94         }
95
96         printf("*****\nInsertion dans l'arbre vide\n");
97         libererArbre(&monArbre);
98         insererMot(&monArbre, "alpha");
99
100        printf("*****\nAffichage apres insertion\n");
101        afficherArbre(monArbre);
102
103        printf("*****\nLiberation de l'arbre\n");
104        libererArbre(&monArbre);
105
106        printf("*****\nAffichage apres liberation\n");
107        afficherArbre(monArbre);
108
109        printf("#####\nFIN DU PROGRAMME DE TEST\n#####\n");
110    } else {
111        fprintf(stderr, "Probleme creation arbre\n");
112    }
113 }

```

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 précédemment 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 fournis 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 caractères)
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 insérerMot

Algorithme insérerMot (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

```
1 #Compilateur et options de compilation
2 CC=gcc
3 CFLAGS=-Wall -ansi -pedantic -Wextra -g
4
5 #Fichiers du projet
6 SOURCES=main.c stack.c tree.c list.c
7 OBJECTS=$(SOURCES:.c=.o)
8
9 EXEC=prog
10
11 $(EXEC): $(OBJECTS)
12     $(CC) $(CFLAGS) $^ -o $(EXEC)
13
14 .c.o:
15     $(CC) -c $(CFLAGS) *.c
16
17 clean:
18     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 commençant par ba
M ba

#Recherche de motif: cas de mots inexistants: mots commençant 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 :

Test de creation	Resultat
Affichage apres creation	
abat	faire
arbre	jouet
art	joueur
arts	kimono
bailler	kiwi
bar	mas
barre	masse
bien	vain
bosse	yack
bossu	zebu
faim	
	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 → commençant par ba	Recherche de motif: recherche du motif vide → (= contenu de l'arbre)
bail	abat
bailler	arbore
bar	arbre
barre	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.

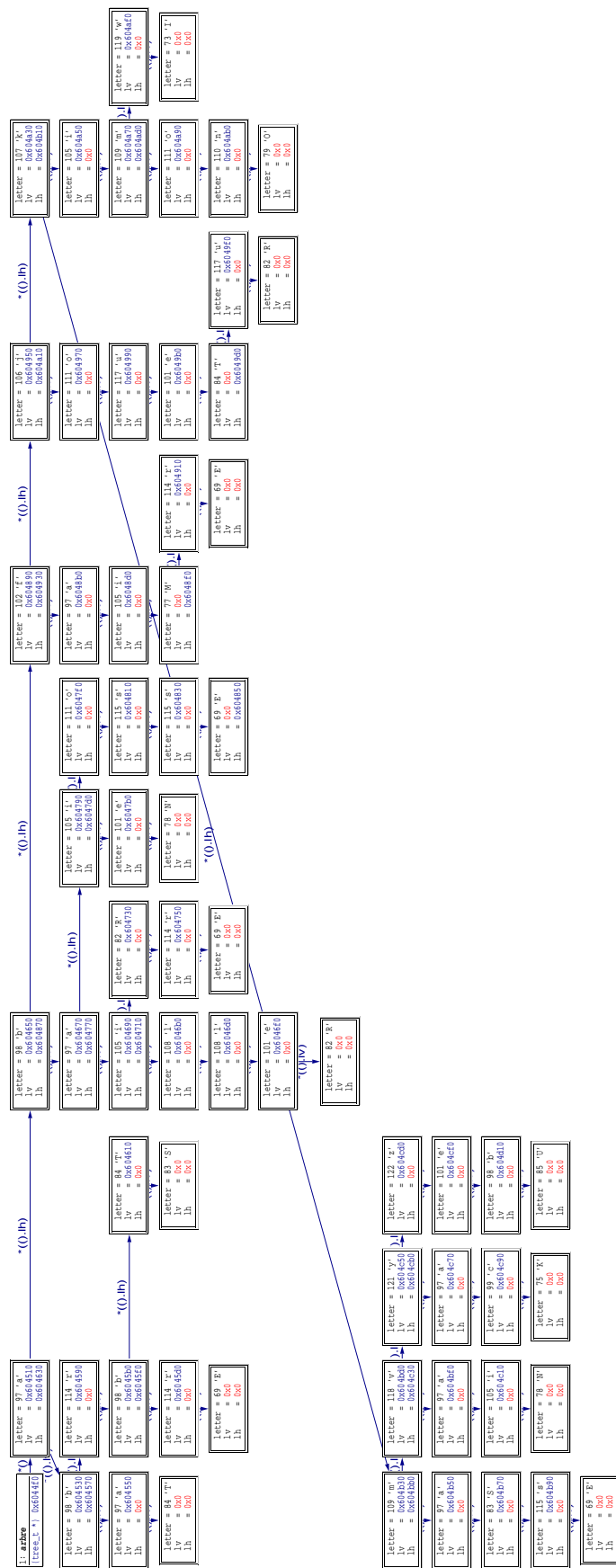


FIGURE 4.1 – Création de l'arbre

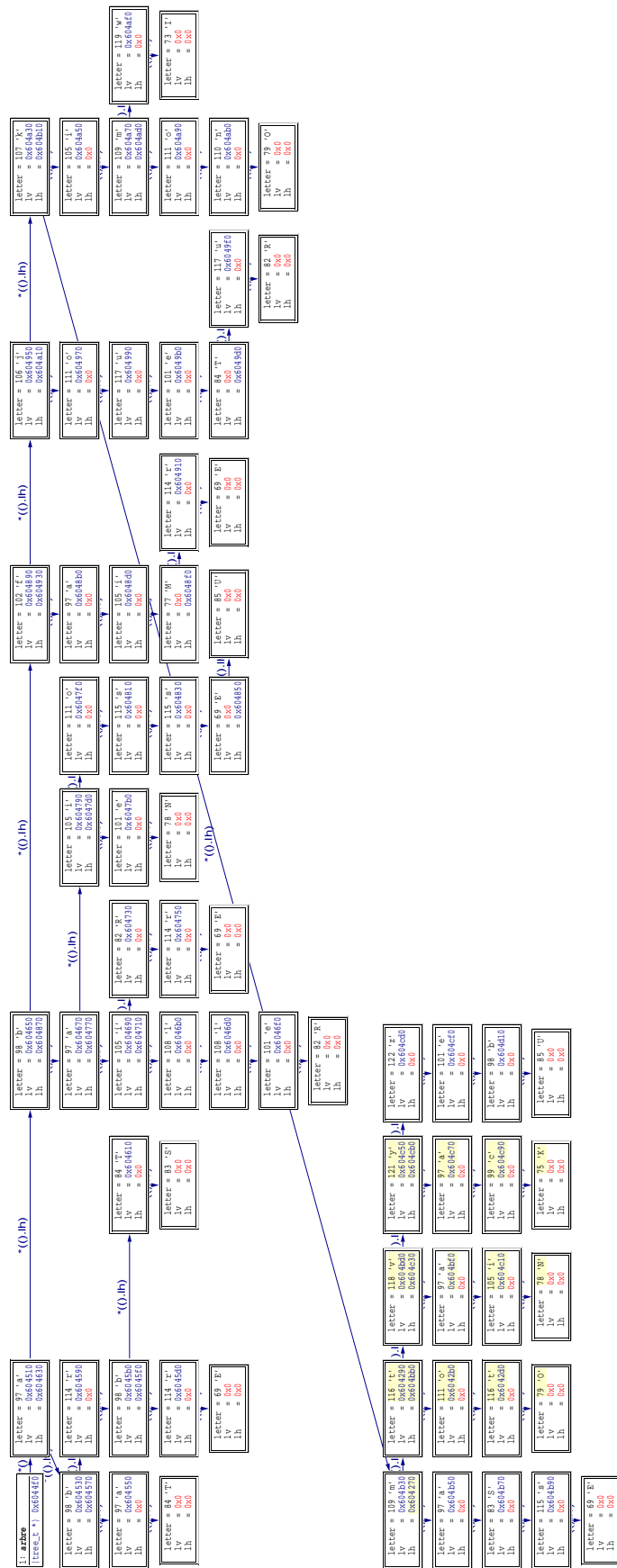


FIGURE 4.2 – Insertion de "toto"

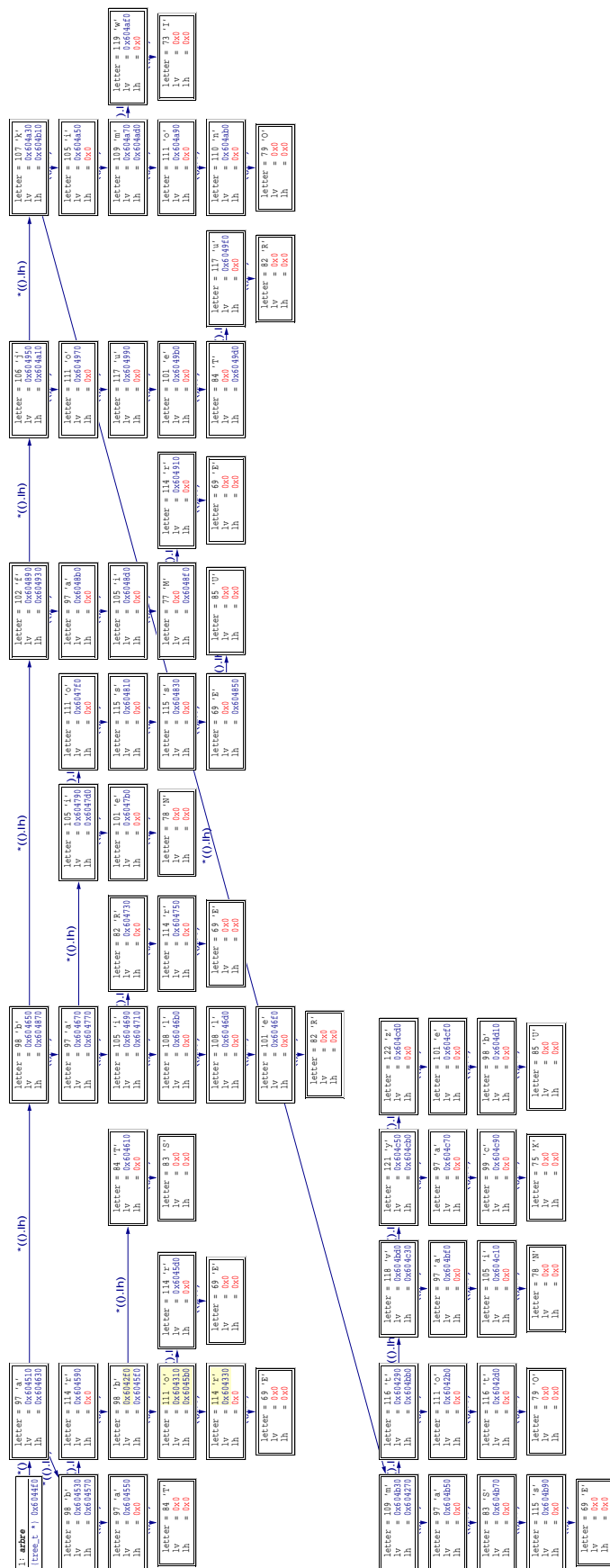


FIGURE 4.3 – Insertion de "arbre"

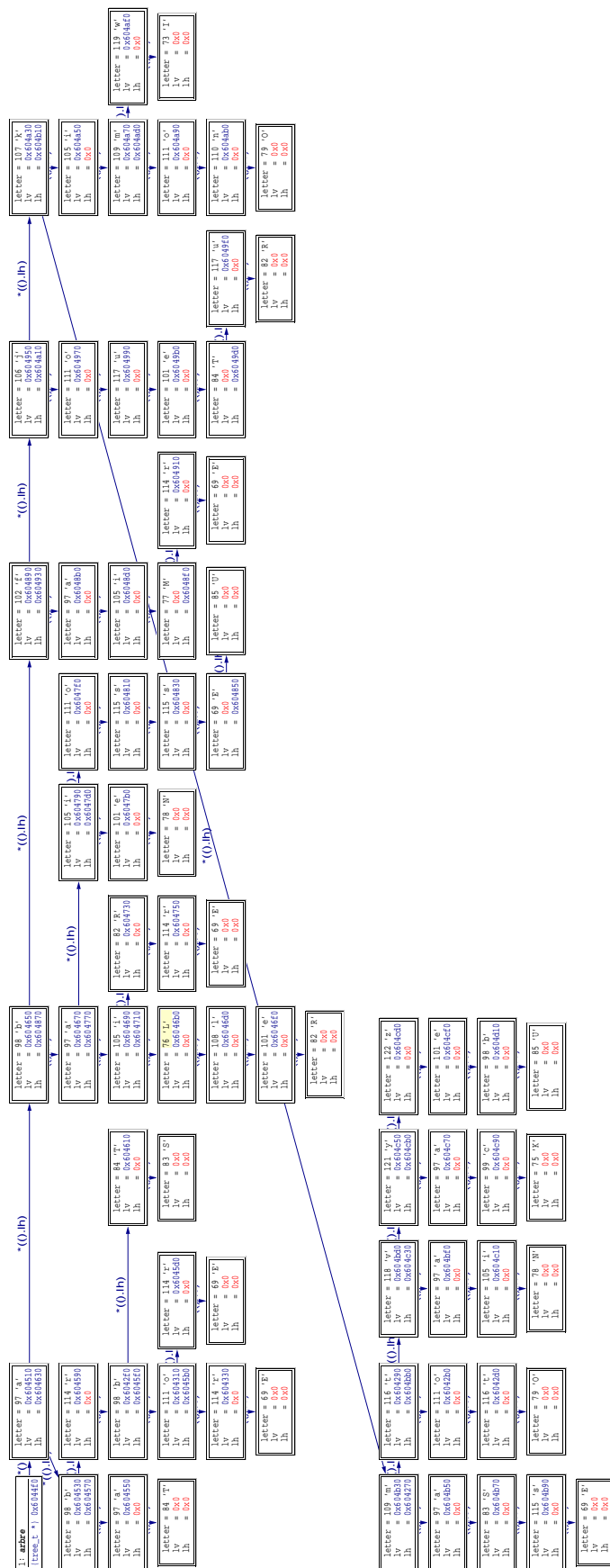


FIGURE 4.4 – Insertion de "baïl"

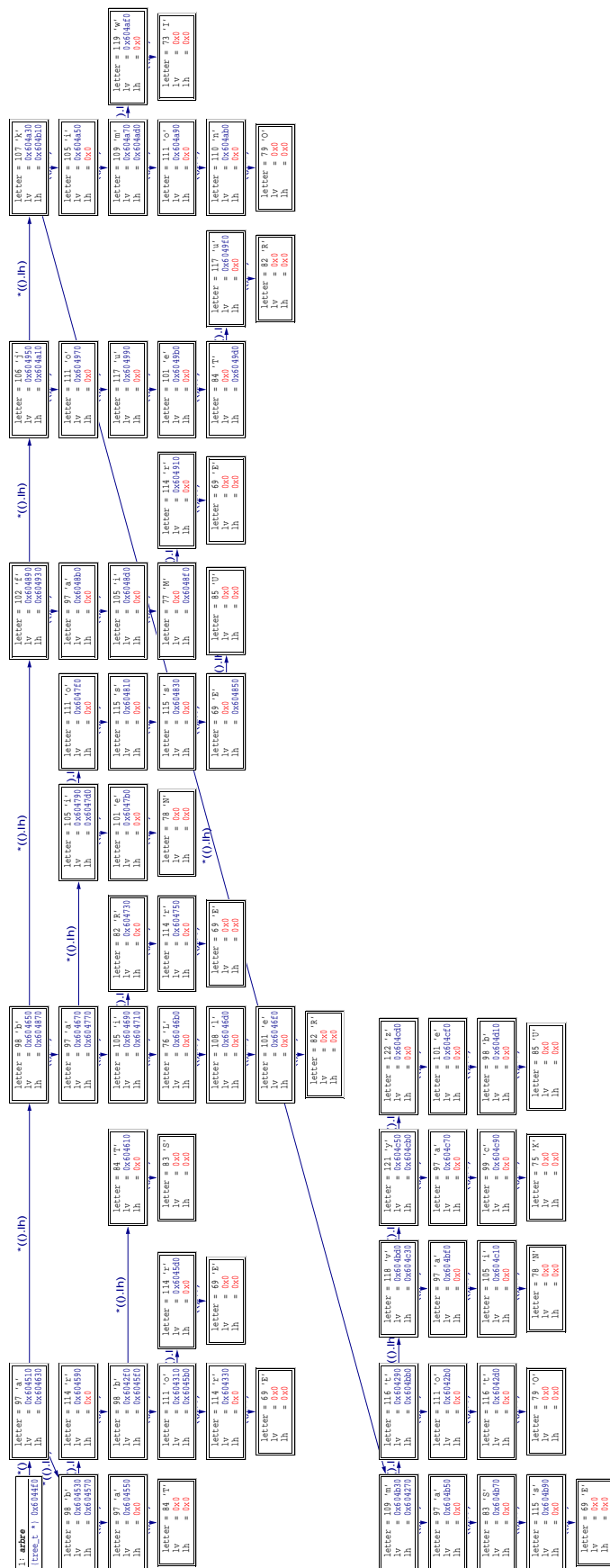


FIGURE 4.5 – Insertion de "barre"

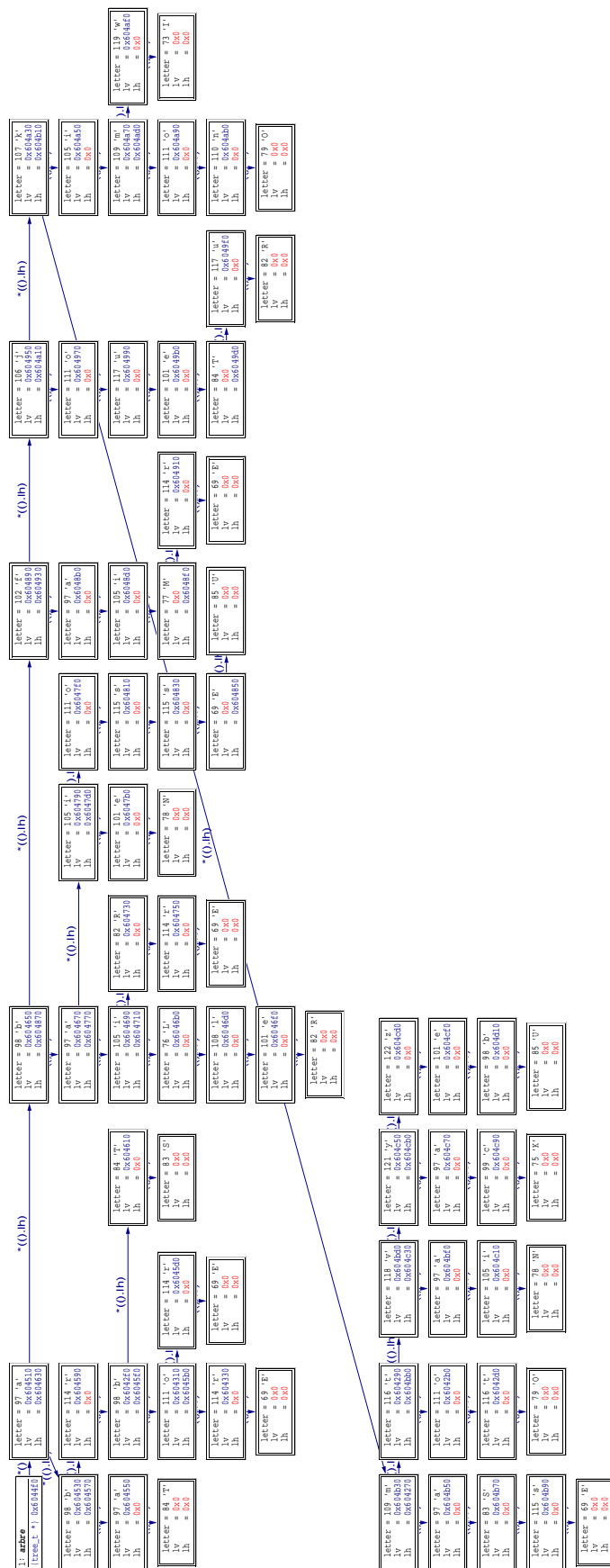


FIGURE 4.6 – Insertion du mot vide

94: arbre
(tree_t *) 0x0

FIGURE 4.7 – Suppression de l'arbre

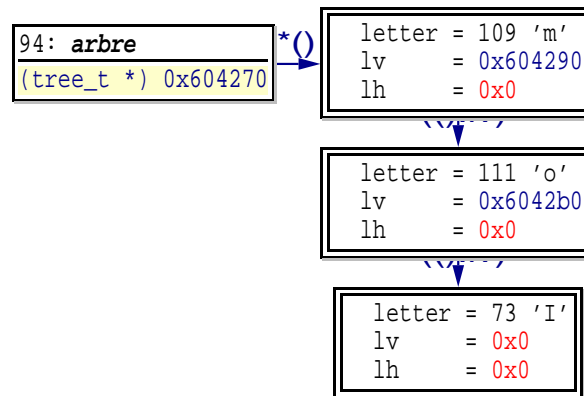


FIGURE 4.8 – Insertion de "moi"

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 HORIZONTALS 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` :

```

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