Compte rendu du TP2 de Structure de Données

Laurent Valentin, Malrin Vincent

6 juin 2013

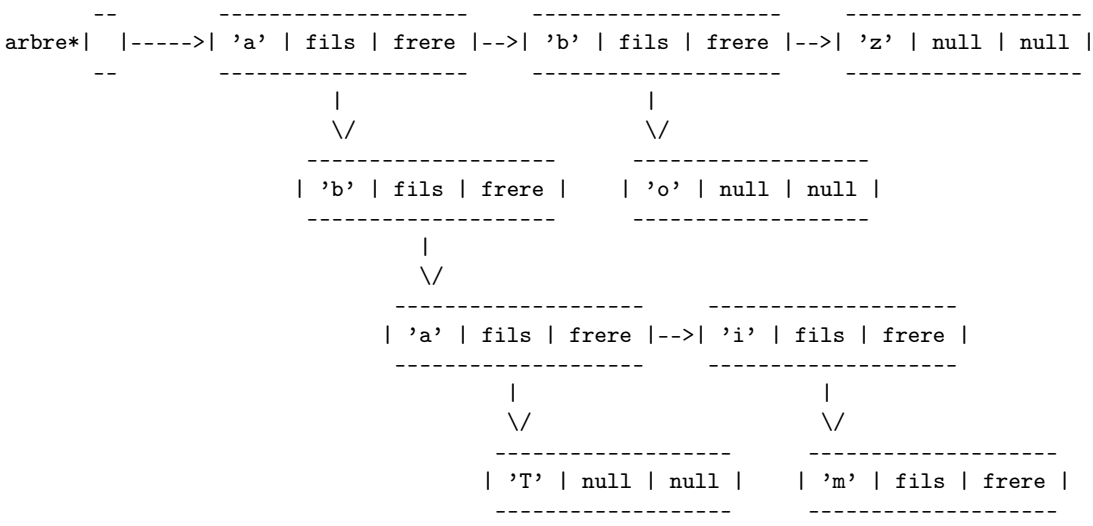
Table des matières

1.1 Objet du TP

Représentation d'un dictionnaire par une information arborescente (stockage lvh).

1.2 Structure de données

Chaque cellule de la structure arborescente contient une valeur (lettre), un pointeur vers le fils, et un pointeur vers le frère. Chaque valeur n'apparait qu'une seule fois parmi ses frères, les lettres appartenant à une même liste chaînée par le lien horizontal sont ordonnées par ordre alphabétique. De plus, si une valeur est une majuscule, la suite des lettres depuis la racine jusqu'à cet lettre est un mot et si un mot est dans le dictionnaire, il existe un chemin depuis une racine jusqu'à une lettre en majuscule dont les valeurs des cellules forment ce mot.



1.3 Organisation du code source

arbre.h

C'est le fichier d'entête des fonctions sur les arbres.

arbre.c

Ce fichier contient les fonctions :

- construction
- lecture_abr
- liberation_abr
- affichage_abr
- insertion_abr

lifo.h

C'est le fichier d'entête des fonctions sur les piles.

lifo.c

Ce fichier contient les fonctions :

- empiler
- depiler
- init_lifo
- lifo_vide
- lifo_pleine
- free_lifo
- afficher_lifo
- inversion_pile

fifo.h

C'est le fichier d'entête des fonctions sur les files.

fifo.c

Ce fichier contient les fonctions :

- emfiler
- defiler
- init_fifo
- fifo_vide
- fifo_pleine
- free_fifo
- afficher_fifo

chaine.h

C'est le fichier d'entête des fonctions sur les chaines de caractères.

chaine.c

Ce fichier contient les fonctions :

- verif_chaine
- verif_arguments

1.4 Codes Sources

1.4.1 Question 1 : Inversion d'une pile

EN TÊTE lifo.c : lifo.h

```c
/* Malrin Vincent
 * Laurent Valentin
 * 4/06/2013
 * LIFO : lifo.h
 */

/*GARDIEN*/
#ifndef LIFO_H
#define LIFO_H

/*INCLUDE*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fifo.h"

typedef struct lifo
{
    int t_max;
    int taille;
    type_t * tab;
} lifo_t;

/* FONCTION : empiler
 * Empile un element dans la lifo (lifo_t).
 *
 * entree : le pointeur de la lifo (lifo_t)
 *          la valeur a inserer (type_t *)
 * sortie : le code d'erreur (int)
 */
int empiler(lifo_t * p, type_t val);

/* FONCTION : depiler
 * Depile un element dans la lifo (lifo_t).
 *
 * entree : le pointeur de la lifo (lifo_t)
 *          la valeur a inserer (type_t *)
 * sortie : le code d'erreur (int)
 */
type_t depiler(lifo_t * p);

/* FONCTION : init_lifo
 * Initialise la lifo (lifo_t).
 *
 * entree : le pointeur de la lifo (lifo_t)
 *          la taille maximale (int)
 * sortie : le code d'erreur (int)
*/
int init_lifo(lifo_t * p, int t_max);

/* FONCTION : est_vide
 * Vérifie si la lifo est vide (lifo_t).
 *
```

```
 * entree : le pointeur de la lifo (lifo_t)
 * sortie : le code d'erreur (int)
*/
int lifo_vide(lifo_t* p);

/* FONCTION : est_pleine
 * Vérifie si la lifo est pleine (lifo_t).
 *
 * entree : le pointeur de la lifo (lifo_t)
 * sortie : le code d'erreur (int)
*/
int lifo_pleine(lifo_t* p);

/* FONCTION : free_lifo
 * Libère la lifo (lifo_t).
 *
 * entree : le pointeur de la lifo (lifo_t)
 * sortie : le code d'erreur (int)
*/
void free_lifo(lifo_t* p);

/* FONCTION : afficher_lifo
 * Affiche la pile.
 *
 * entree : le pointeur de la lifo (lifo_t)
 */

void afficher_lifo(lifo_t * p);

/*FONCTION : inversion_pile
 * Porte bien son nom.
 *
 * entree : le pointeur de la pile (lifo_t).
 */

void inversion_pile(lifo_t * p);

/*FONCTION : lecture_pile
 * Lit une chaine de caractère et empile chaque caractère.
 *
 * entree : la pile
 *          une chaine de caractère
 */

void lecture_pile(lifo_t * p, char * ch);

#endif
```

GESTION PILE : lifo.c

```
/* Malrin Vincent
 * Laurent Valentin
 * 4/06/2013
 * INVERSION LIFO : lifo.h
 */

/*INCLUDE*/
#include "lifo.h"
#include "fifo.h"

```

```c
/*DEFINE*/
#define TRUE 1
#define FALSE 0

int empiler(lifo_t * p, type_t val)
{
    /*variables locales*/
    int        err = FALSE;   /*code erreur*/

    /*teste si la lifo est pleine*/
    if (p->t_max > (p->taille + 1))
    {
        p->taille++;  /*on incremente la taille*/
        p->tab[p->taille]=val; /*insertion de la valeur dans la pile*/
        err=TRUE;  /*on valide le code erreur*/
    }
    return err;
}

type_t depiler(lifo_t * p)
{
    /*variables locales*/
    type_t       val;

    if (!lifo_vide(p)) /*si pile non vide*/
    {
        val=p->tab[p->taille]; /*on recupere la valeur*/
        p->taille--;  /*on decremente la taille*/

    }
    return val;
}

int init_lifo(lifo_t * p, int t_max)
{
    /*variables locales*/
    int        err = TRUE;

    /*initialisation de la lifo*/
    p->t_max = t_max; /*initialisation de la taille max*/
    p->taille = -1;   /*la taille allouee vaut -1 -> pile vide*/
    p->tab = (type_t*)malloc(t_max*sizeof(type_t));

    if (p->tab)    /*on verifie l'allocation*/
    {
        err = FALSE;
    }
    return err;
}

int lifo_vide(lifo_t* p)
{
    /*variables locales*/
    int ret = FALSE;

    if (p->taille < 0) /*on verifie l'indice de pile*/
    {
        ret = TRUE;
    }
```

```
70      return ret;
71  }
72
73  int lifo_pleine(lifo_t* p)
74  {
75      /*variables locales*/
76      int ret = FALSE;
77
78      if (p->taille == p->t_max) /*on verifie l'indice de pile*/
79      {
80          ret = TRUE;
81      }
82      return ret;
83  }
84
85  void free_lifo(lifo_t* p)
86  {
87
88      free(p->tab); /*libere l'espace d'empilage*/
89      free(p);
90  }
91
92  void afficher_lifo(lifo_t * p)
93  {
94      /*variables locales*/
95      int         cour = 0;             /*parcours de pile*/
96      int         taille = p->taille;   /*parcours de l'espace d'empilage*/
97
98      while (cour <= taille) /*boucle de parcours*/
99      {
100         printf("%c\n",(p->tab[cour]));
101         fflush(stdout);
102         cour++;
103     }
104     printf("\n");
105 }
106
107 void inversion_pile(lifo_t * p)
108 {
109     /*variables locales*/
110     fifo_t *      f = NULL;  /*file*/
111     type_t        tmp;       /*valeurs a afficher*/
112     int           err;       /*erreur*/
113
114     /*initialisation*/
115     f = (fifo_t *)malloc(sizeof(fifo_t));
116     err = init_fifo(f,p->t_max);
117
118     if (! err)
119     {
120         printf("Pile a inverser :\n");
121         afficher_lifo(p);
122
123         /*parcours de la pile*/
124         while (!lifo_vide(p))
125         {
126             tmp = depiler(p); /*passage des valeurs*/
127             enfiler(f,tmp);
128         }
```

```
129
130         printf("Etat de la file :\n");
131         afficher_fifo(f);
132
133         while (! fifo_vide(f))
134         {
135             tmp = defiler(f); /*passage des valeurs*/
136             empiler(p,tmp);
137         }
138
139         /*liberation de la file*/
140         free_fifo(f);
141     }
142
143     printf("Pile inversee :\n");
144     afficher_lifo(p);
145 }
146
147 void lecture_pile(lifo_t * p, char * ch)
148 {
149     /*variables locales*/
150     int         compt = 0;
151
152     while (ch[compt] != '\0')
153     {
154         empiler(p,ch[compt]);
155         compt++;
156     }
157 }
```

EN TÊTE fifo.c : fifo.h

```
1  /* Malrin Vincent
2   * Laurent Valentin
3   * 4/06/2013
4   * FIFO : fifo.h
5   */
6
7  /*GARDIEN*/
8  #ifndef FIFO_H
9  #define FIFO_H
10
11 /*INCLUDE*/
12 #include <stdio.h>
13 #include <stdlib.h>
14
15 typedef char type_t;
16
17 typedef struct fifo
18 {
19     int t_max;
20     int in;
21     int out;
22     type_t * tab;
23 } fifo_t;
24
25 /* FONCTION : enfiler
26  * Enfile un element dans la file (fifo_t).
27  *
```

```
 * entree : le pointeur de la file (fifo_t)
 *         la valeur a inserer (type_t)
 */

int enfiler(fifo_t * f, type_t val);

/* FONCTION : defiler
 * Defile un element dans la file (fifo_t).
 *
 * entree : le pointeur de la file (fifo_t)
 *         la valeur a inserer (type_t)
 */

type_t defiler(fifo_t * f);

/* FONCTION : init_fifo
 * Initialise la file (fifo_t).
 *
 * entree : le pointeur de la file (fifo_t)
 *         la taille maximale (int)
 */

int init_fifo(fifo_t * f, int t_max);

/* FONCTION : est_vide
 * Vérifie si la fifo est vide (fifo_t).
 *
 * entree : le pointeur de la fifo (fifo_t)
 * sortie : le code d'erreur (int)
 */

int fifo_vide(fifo_t * f);

/* FONCTION : est_pleine
 * Vérifie si la fifo est pleine (fifo_t).
 *
 * entree : le pointeur de la fifo (fifo_t)
 * sortie : le code d'erreur (int)
 */

int fifo_pleine(fifo_t* f);

/* FONCTION : free_fifo
 * Libère la fifo (fifo_t).
 *
 * entree : le pointeur de la fifo (fifo_t)
 * sortie : le code d'erreur (int)
 */

void free_fifo(fifo_t* f);

/* FONCTION : afficher_fifo
 * Affiche la fifo (fifo_t).
 *
 * entree : le pointeur de la fifo (fifo_t)
 */

void afficher_fifo(fifo_t * f);

```

```
87 #endif
```

GESTION FILE : fifo.c

```
1  /* Malrin Vincent
2   * Laurent Valentin
3   * 4/06/2013
4   * FIFO : fifo.c
5   */
6
7
8  /*INCLUDE*/
9  #include "fifo.h"
10
11 /*DEFINE*/
12 #define TRUE 1
13 #define FALSE 0
14
15 int enfiler(fifo_t * f, type_t val)
16 {
17     /*variables locale*/
18     int        err = TRUE; /*code erreur*/
19
20     /*teste si la fifo est pleine*/
21     if (! fifo_pleine(f))
22     {
23         f->in++;   /*incremente l'indice d'entree*/
24         f->tab[f->in % f->t_max] = val; /*on enfile la valeur*/
25         err = FALSE;
26     }
27     return err;
28 }
29
30 type_t defiler(fifo_t * f)
31 {
32     /*variable locale*/
33     type_t       val; /*valeur cherchee*/
34
35     if (!fifo_vide(f))
36     {
37         f->out++;  /*incremente l'indice de sortie*/
38         val = f->tab[f->out % f->t_max]; /*on defile la valeur*/
39     }
40     return val;
41 }
42
43 int init_fifo(fifo_t * f, int t_max)
44 {
45     /*variables locales*/
46     int        err = TRUE;
47
48     /*initialisation de la fifo*/
49     f->t_max = t_max; /*initialise la taille max*/
50     f->tab = (type_t*)malloc(t_max*sizeof(type_t));
51
52     if (f->tab)
53     {
54         f->in = 0; /*initialise les indices*/
55         f->out = 0;
56         err = FALSE;
```

```
    }
    return err;
}

int fifo_vide(fifo_t * f)
{
    /*variables locales*/
    int        ret = FALSE;

    if ((f->in - f->out) == 0) /*file vide -> indice aux meme endroits*/
    {
        ret = TRUE;
    }

    return ret;
}

int fifo_pleine(fifo_t * f)
{
    /*variables locales*/
    int ret = FALSE;

    if ((f->in - f->out) == f->t_max) /*file pleine -> la distance des indice*/
    {                                 /* vaut taille max*/
        ret = TRUE;
    }
    return ret;
}

void free_fifo(fifo_t* f)
{
    free(f->tab); /*libere l'espace d'enfilage*/
    free(f);
}

void afficher_fifo(fifo_t * f)
{
    /*variables locales*/
    int               cour = f->out + 1;    /*parcours*/
    int               nb = f->in - f->out;  /*nb d'elements*/

    while (nb > 0)
    {
        printf("%c\n",f->tab[cour]);
        cour++;    /*avance dans la file*/
        cour = cour % f->t_max;
        nb--;
    }
    printf("\n");
}
```

MAIN INVERSION : main.c

```
/* Malrin Vincent
 * Laurent Valentin
 * 4/06/2013
 * INVERSION MAIN : main.c
 */

/*INCLUDE*/
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lifo.h"
#include "fifo.h"

/*DEFINE*/
#define TMAX 100
#define TRUE 1
#define FALSE 0

int main(int argc, char ** argv)
{
    /*variables locales*/
    lifo_t *   p = NULL;      /*pile*/
    int        err = TRUE;    /*erreur d'allocation*/

    /*test des arguments*/
    if (argc < 2)
    {
        printf("USAGE %s <chaine a inverser>\n",argv[0]);
    }
    else
    {
        /*initialisation*/
        p = (lifo_t *)malloc(sizeof(lifo_t));
        err = init_lifo(p,TMAX);

        if (!err && p)
        {
            lecture_pile(p,argv[1]);
            inversion_pile(p);

            /*liberation des ressources*/
            free_lifo(p);
            p = NULL;
        }
    }
    return 0;
}
```

MAKEFILE

```
# compilateur
CC = gcc

# options de compilation
CFLAGS = -Wall -ansi -pedantic -Wextra -g

# options de l'edition des liens
LDFLAGS = -g

# nom de l'executable a generer
EXEC = inversion

# liste des fichiers objets
OBJ = main.o lifo.o fifo.o

all : $(EXEC)

```

```
18 # regle de production finale
19 inversion : $(OBJ)
20   $(CC) $(OBJ) -o $@ $(LDFLAGS)
21
22 # regle de production pour chaque fichier
23 main.o : main.c lifo.h fifo.h
24   $(CC) -c $< $(CFLAGS)
25
26 lifo.o : lifo.c lifo.h fifo.h
27   $(CC) -c $< $(CFLAGS)
28
29 fifo.o : fifo.c fifo.h lifo.h
30   $(CC) -c $< $(CFLAGS)
31
32 # regle de suppression des fichiers
33 clean :
34   rm *.o $(EXEC)
```

1.4.2 Question 2,3 et 4 : Gestion du dictionnaire arborescent

EN TÊTE arbre.c : arbre.h

```c
/* Malrin Vincent
 * Laurent Valentin
 * 4/06/2013
 * ARBRE : arbre.h
 */

/*GARDIEN*/
#ifndef ARBRE_H
#define ARBRE_H

/*INCLUDE*/
#include <stdio.h>
#include <stdlib.h>
#include "ctype.h"
#include "string.h"

typedef struct abr
{
    char          val;
    struct abr *  fils;
    struct abr *  frere;
} abr_t;

/* FONCTION : construction
 * Construit un noeud à partir de sa valeur (noeud_t).
 *
 * entree : la valeur (char)
 * sortie : le code erreur (int)
 */

int construction(char val, abr_t ** n);

/* FONCTION : lecture_abr
 * Lit la chaine de caractere (ecriture parenthesee) et construit l'arbre.
 *
 * entree : chaine de caractere (char *)
 *          le pointeur sur l'arbre (noeud_t *)
 * sortie : erreur (int).
 */

int lecture_abr(char * ch, abr_t ** r);

/* FONCTION : liberation_abr
 * Libere les ressources.
 *
 * entree : la racine (abr_t ** r)
 * sortie : erreur (int)
 */

int liberation_abr(abr_t ** r);

/* FONCTION : affichage_abr
 * Affiche les mots contenus dans l'arbre.
 *
 * entree : la racine (abr_t ** r)
```

```
 * sortie : erreur (int)
 */

int affichage_abr(abr_t ** r);

/* FONCTION : insertion_abr
 * Insère une chaine de caractères dans l'arbre trié.
 *
 * entrée : la racine (abr_t ** r)
 *          la chaine a inserer (char * ch)
 * sortie : erreur (int)
 */

int insertion_abr(abr_t ** r, char * ch);

#endif
```

GESTION ARBRE : arbre.c

```
/* Malrin Vincent
 * Laurent Valentin
 * 4/06/2013
 * ARBRE : arbre.c
 */

/*INCLUDE*/
#include "lifo.h"
#include "chaine.h"

/*DEFINE*/
#define TRUE 1
#define FALSE 0
#define TMAX 50

/* FONCTION : construction
 * Construit un noeud à partir de sa valeur (noeud_t).
 *
 * entree : la valeur (char)
 * sortie : le code erreur (int)
 */

int construction(char val, abr_t ** a)
{
    /*variables locales*/
    int        err = TRUE;

    /*allocation memoire*/
    *a = (abr_t *)malloc(sizeof(abr_t));

    /*verification*/
    if (*a)
    {
        err = FALSE;
        (*a)->val = val;      /*insertion de la valeur*/
        (*a)->fils = NULL;    /*initialisation du fils*/
        (*a)->frere = NULL;   /*initialisation du frere*/
    }
    return err;
}

```

```
42  int lecture_abr(char * ch, abr_t ** r)
43  {
44      /*variables locales*/
45      lifo_t *      p = (lifo_t *)malloc(sizeof(lifo_t)); /*pile*/
46      abr_t **      cour = r;                             /*pointeur de parcours*/
47      int           compt = 0;                            /*compteur sur la chaine*/
48      int           compt_parenthese = 0;                 /*compteur de parentheses*/
49      int           err = FALSE;
50
51      /*initialisation de la pile*/
52      err = init_lifo(p,TMAX);
53
54      if (err)
55      {
56          printf("Erreur : pile non initialisee !\n");
57      }
58
59      err = verif_chaine(ch);
60
61      if (ch[compt] != '(' && ! err) /*verification 1ere parenthese*/
62      {
63          printf("Parametre incorrect : il manque la 1ere parenthese !\n");
64          err = TRUE;
65      }
66      else if (! err)
67      {
68          compt_parenthese++; /*on compte la 1ere parenthese*/
69          compt++; /*on avance dans la chaine*/
70
71          if (ch[compt] != ')') /*arbre NON vide*/
72          {
73              construction(ch[compt],cour); /*initialisation de la racine*/
74              compt++;
75              *r = *cour; /*initialisation des pointeurs*/
76          }
77          else   /*arbre vide*/
78          {
79              cour = NULL;
80              compt_parenthese--;
81          }
82
83          /*boucle*/
84          while ((compt_parenthese > 0) && (! err))
85          {
86              switch(ch[compt])
87              {
88                  case '(':
89                      empiler(p,cour); /*on empile l'adr*/
90                      compt++; /*on avance dans la chaine*/
91                      compt_parenthese++; /*on incremente le compteur de parentheses*/
92                      cour = &(*cour)->fils; /*on deplace cour sur le fils*/
93                      construction(ch[compt],cour); /*on cree le fils*/
94                      compt++; /*on avance dans la chaine*/
95                      break;
96
97                  case ')':
98                      if (compt_parenthese > 0)
99                      cour = depiler(p); /*on depile l'adr*/
100                     compt++; /*on avance dans la chaine*/
```

```
                    compt_parenthese--; /*on decremente le compteur de parentheses*/
                    break;

                case ',':
                    compt++; /*on avance dans la chaine*/
                    break;

                default:
                    cour = &(*cour)->frere; /*on deplace cour sur le frere*/
                    construction(ch[compt],cour); /*on construit une nouvelle cellule*/
                    compt++; /*on avance dans la chaine*/
            }
        }
    }
    free_lifo(p);  /*libération des ressources*/

    return err;
}

int liberation_abr(abr_t ** r)
{
    lifo_t *      p = (lifo_t *)malloc(sizeof(lifo_t)); /*pile*/
    abr_t **      cour = r;                             /*pointeur de parcours*/
    abr_t **      tmp = NULL;                           /*pointeur de liberation*/
    int           fin = FALSE;                          /*fin de parcours (pile vide)*/
    int           err = FALSE;                          /*erreur d'allocation*/

    err = init_lifo(p,TMAX); /*initialisation de la pile*/

    if (err)   /*verification d'allocation*/
    {
        printf("Erreur : pile non initialisee !");
    }

    while ((! fin))   /*boucle de parcours*/
    {
        while (*cour) /*parcours postfixe*/
        {
            empiler(p,cour);  /*si il a un fils : on empile l'adr*/
            cour = &(*cour)->fils;
        }
        if (! lifo_vide(p))
        {
            cour = depiler(p);    /*feuille : on depile les cellule parcourues*/
            tmp = cour;

            if ((*cour)->frere)   /*si il a un frere : on empile l'adr*/
            {
                empiler(p,cour);
                cour = &(*cour)->frere;
            }
            else
            {
                cour = &(*cour)->frere;  /*sinon on libere la memoire*/
                printf("free(%c)\n",(*tmp)->val); /*affichage pour tests*/
                fflush(stdout);
                free(*tmp);
                *tmp = NULL;
            }
```

```
160         }
161         else
162         {
163             fin = TRUE;    /*fin de parcours : pile vide*/
164         }
165     }
166 
167     free_lifo(p); /*liberation des ressources*/
168 
169     return err;
170 }
171 
172 int affichage_abr(abr_t ** r)
173 {
174     lifo_t *       p = (lifo_t *)malloc(sizeof(lifo_t)); /*pile*/
175     abr_t **       cour = r;                             /*pointeur de parcours*/
176     int            fin = FALSE;                          /*fin de parcours*/
177     int            err = FALSE;                          /*erreur d'allocation*/
178 
179     err = init_lifo(p,TMAX); /*initialisation de la pile*/
180 
181     if (err) /*verification de l'allocation*/
182     {
183         printf("Erreur : pile non initialisee !");
184     }
185 
186     while ((! fin) && (! err)) /*boucle de parcours*/
187     {
188         while (*cour) /*parcours en profondeur*/
189         {
190             empiler(p,cour);
191 
192             /*verifie si on parcours une majuscule*/
193             if (((*cour)->val > 64) && ((*cour)->val < 91))
194             {
195                 afficher_lifo(p); /*si majuscule : on affiche le mot parcouru*/
196             }
197             cour = &(*cour)->fils; /*on avance dans la structure (mot suivant)*/
198         }
199         if (! lifo_vide(p))   /*on depile jusqu'au frere suivant*/
200         {
201             cour = depiler(p);
202             cour = &(*cour)->frere;
203         }
204         else    /*fin de parcours : pile vide*/
205         {
206             fin = TRUE;
207         }
208     }
209     free_lifo(p);  /*liberation des ressources*/
210 
211     return err;
212 }
213 
214 int insertion_abr(abr_t ** r, char * ch)
215 {
216     abr_t **       prec = r;      /*pointeur sur precedent (parcours)*/
217     int            err = FALSE;   /*erreur d'allocation*/
218     int            i = 0;
```

```
219     abr_t *        tmp = NULL;    /*valeur temporaire (insertion)*/
220     char           char_ch;       /*carctere pointe dans l'abr*/
221     char           char_abr;      /*caractere pointe dans la chaine*/
222
223     while(ch[i] != '\0')
224     {
225         if (*prec)
226         {
227             char_ch = ch[i] % 32; /*initialisation des caracteres parcourus*/
228             char_abr = (*prec)->val % 32;
229
230             if (char_abr == char_ch) /*si les caracteres sont les mêmes...*/
231             {
232                 if(ch[i+1] == '\0')   /*gestion dernière lettre*/
233                 {
234                     (*prec)->val=toupper(ch[i]);
235                 }
236                 else
237                 {
238                     prec = &(*prec)->fils; /*...on continue le parcours (fils)*/
239                 }
240                 i++;
241             }
242             else if (char_abr < char_ch) /* si le car mot > car abr...*/
243             {
244                 prec = &(*prec)->frere;  /*...on continue le parcours (frere)*/
245             }
246             else
247             {
248                 tmp = *prec;  /*...a la place de la cellule courante...*/
249
250                 if(ch[i+1] == '\0') /*gestion dernière lettre*/
251                 {
252                     construction(toupper(ch[i]),prec);
253                 }
254                 else
255                 {
256                     construction(ch[i],prec);
257                 }
258
259                 (*prec)->frere = tmp; /*...qui devient le frere*/
260                 prec = &(*prec)->fils;
261                 i++;
262             }
263         }
264         else
265         {
266             if(ch[i+1] == '\0') /*gestion dernière lettre*/
267             {
268                 construction(toupper(ch[i]),prec);
269             }
270             else
271             {
272                 construction(ch[i],prec); /*on insere le reste du mot*/
273             }
274
275             prec = &(*prec)->fils;
276             i++;
277         }
```

```
278     }
279     return err;
280 }
```

EN TÊTE chaine.c : chaine.h

```
1  /* Malrin Vincent
2   * Laurent Valentin
3   * 4/06/2013
4   * CHAINE : chaine.c
5   */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include "ctype.h"
10 #include "string.h"
11
12 /* FONCTION : verif_chaine
13  * Verifie si la chaine est une notation parenthesee.
14  *
15  * entree : la chaine (char *)
16  * sortie : le code erreur (int)
17  */
18
19 int verif_chaine(char * ch);
20
21 /* FONCTION : verif_argument
22  * Verifie si la chaine est un mot.
23  *
24  * entree : la chaine (char *)
25  * sortie : le code erreur (int)
26  */
27
28 int verif_argument(char * ch);
```

GESTION CHAINE : chaine.c

```
1  /* Malrin Vincent
2   * Laurent Valentin
3   * 4/06/2013
4   * CHAINE : chaine.c
5   */
6
7  /*INCLUDE*/
8  #include "chaine.h"
9
10 /*DEFINE*/
11 #define FALSE 0
12 #define TRUE 1
13
14 int verif_chaine(char * ch)
15 {
16     /*variables locales*/
17     int        compt = 0;            /*compteur dans la chaine*/
18     int        compt_parenthese = 0; /*compteur de parenthese*/
19     int        err = FALSE;          /*erreur*/
20     int        abr_vide = FALSE;     /*cas particuler : l'abr vide*/
21
22     /*initialisation de la chaine*/
23     ch[strlen(ch) - 1] = '\0';
```

```
24
25      if (ch[compt] == '(' && ch[compt + 1] == ')' && ch[compt + 2] == '\0')
26      {
27          abr_vide = TRUE; /*verifie si c'est l'arbre vide*/
28      }
29
30      while (ch[compt + 1] != '\0' && ! err && ! abr_vide)
31      {
32          switch(ch[compt])
33          {
34              case '(': /*si le caractere est '(' -> une lettre ou '('*/
35                  if(isalpha(ch[compt + 1]) || ch[compt + 1] == '(')
36                  {
37                      compt_parenthese++;
38                      compt++;
39                  }
40                  else
41                  {
42                      err = TRUE;
43                  }
44                  break;
45              case ')': /*si le caractere est ')' -> une lettre ou ')'*/
46                  if(isalpha(ch[compt + 1]) || ch[compt + 1] == ')')
47                  {
48                      compt_parenthese--;
49                      compt++;
50                  }
51                  else
52                  {
53                      err = TRUE;
54                  }
55                  break;
56              case ',': /*si le caractere est '(' -> une lettre*/
57                  if(isalpha(ch[compt + 1]))
58                  {
59                      compt++;
60                  }
61                  else
62                  {
63                      err = TRUE;
64                  }
65                  break;
66              default: /*sinon c'est une lettre non suivie d'une autre lettre*/
67                  if(isalpha(ch[compt]))
68                  {
69                      if(ch[compt + 1] == ')' || ch[compt + 1] == '(' || ch[compt + 1] == ',')
70                      {
71                          compt++;
72                      }
73                      else
74                      {
75                          err = TRUE;
76                      }
77                  }
78                  else /*n'est pas une lettre*/
79                  {
80                      err = TRUE;
81                  }
82          }
```

```
    }

    if (ch[compt] == ')')
    {
        compt_parenthese--;
    }

    if (compt_parenthese != 0) /*les parentheses '(' et ')' doivent etre de nbr egal*/
    {
        err = TRUE;
    }

    if (err)
    {
        printf("Ecriture parenthesee incorrecte !\n");
    }

    return err;
}

int verif_argument(char * ch)
{
    /*variables locales*/
    int        compt = 0;           /*compteur dans la chaine*/
    int        err = FALSE;         /*erreur*/

    while(ch[compt] != '\0' && ! err)
    {
        if (! isalpha(ch[compt]))
        {
            err = TRUE;
        }
        compt++;
    }
    return err;
}
```

EN TÊTE lifo.c : lifo.h

```
/* lifo.h
*/

/*GARDIEN*/
#ifndef LIFO_H
#define LIFO_H

/*INCLUDE*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "arbre.h"

typedef abr_t ** type_t;

typedef struct lifo
{
    int t_max;
    int taille;
    type_t * tab;
```

```
21 } lifo_t;
22
23 /* FONCTION : empiler
24  * Empile un element dans la lifo (lifo_t).
25  *
26  * entree : le pointeur de la lifo (lifo_t)
27  *          la valeur a inserer (type_t *)
28  * sortie : le code d'erreur (int)
29  */
30 int empiler(lifo_t * p, type_t val);
31
32 /* FONCTION : depiler
33  * Depile un element dans la lifo (lifo_t).
34  *
35  * entree : le pointeur de la lifo (lifo_t)
36  *          la valeur a inserer (type_t *)
37  * sortie : le code d'erreur (int)
38  */
39 type_t depiler(lifo_t * p);
40
41 /* FONCTION : init_lifo
42  * Initialise la lifo (lifo_t).
43  *
44  * entree : le pointeur de la lifo (lifo_t)
45  *          la taille maximale (int)
46  * sortie : le code d'erreur (int)
47 */
48 int init_lifo(lifo_t * p, int t_max);
49
50 /* FONCTION : est_vide
51  * Vérifie si la lifo est vide (lifo_t).
52  *
53  * entree : le pointeur de la lifo (lifo_t)
54  * sortie : le code d'erreur (int)
55 */
56 int lifo_vide(lifo_t* p);
57
58 /* FONCTION : est_pleine
59  * Vérifie si la lifo est pleine (lifo_t).
60  *
61  * entree : le pointeur de la lifo (lifo_t)
62  * sortie : le code d'erreur (int)
63 */
64 int lifo_pleine(lifo_t* p);
65
66 /* FONCTION : free_lifo
67  * Libère la lifo (lifo_t).
68  *
69  * entree : le pointeur de la lifo (lifo_t)
70  * sortie : le code d'erreur (int)
71 */
72 void free_lifo(lifo_t* p);
73
74 /* FONCTION : afficher_lifo
75  * Affiche la pile.
76  *
77  * entree : le pointeur de la lifo (lifo_t)
78  */
79
```

```
void afficher_lifo(lifo_t * p);

/*FONCTION : inversion_pile
 * Porte bien son nom.
 *
 * entree : le pointeur de la pile (lifo_t).
 */

void inversion_pile(lifo_t * p);
#endif
```

GESTION PILE : lifo.c

```
/* lifo.c
*/

#include "lifo.h"

/*DEFINE*/
#define TRUE 1
#define FALSE 0

/* FONCTION : empiler
 * Empile un element dans la lifo (lifo_t).
 *
 * entree : le pointeur de la lifo (lifo_t)
 *          la valeur a inserer (type_t *)
 * sortie : le code d'erreur (int)
 */
int empiler(lifo_t * p, type_t val)
{
    /*variables locales*/
    int        err=FALSE; /*code erreur*/

    /*teste si la lifo est pleine*/
    if (p->t_max > (p->taille + 1))
    {
        p->taille++;
        p->tab[p->taille]=val;
        err=TRUE;
    }
    return err;
}

/* FONCTION : empiler
 * Empile un element dans la lifo (lifo_t).
 *
 * entree : le pointeur de la lifo (lifo_t)
 *          la valeur a inserer (type_t *)
 * sortie : le code d'erreur (int)
 */
type_t depiler(lifo_t * p)
{
    /*variables locales*/
    type_t       val;

    if (!lifo_vide(p))
    {
        val=p->tab[p->taille];
        p->taille--;
```

```
48
49      }
50      return val;
51  }
52
53  /* FONCTION : init_lifo
54   * Initialise la lifo (lifo_t).
55   *
56   * entree : le pointeur de la lifo (lifo_t)
57   *          la taille maximale (int)
58   * sortie : le code d'erreur (int)
59  */
60  int init_lifo(lifo_t * p, int t_max)
61  {
62      /*variables locales*/
63      int         err = TRUE;
64
65      /*initialisation de la lifo*/
66      p->t_max = t_max;
67      p->taille = -1;
68      p->tab = (type_t*)malloc(t_max*sizeof(type_t));
69
70      if (p->tab)
71      {
72          err = FALSE;
73      }
74      return err;
75  }
76
77  /* FONCTION : est_vide
78   * Vrifie si la lifo est vide (lifo_t).
79   *
80   * entree : le pointeur de la lifo (lifo_t)
81   * sortie : le code d'erreur (int)
82  */
83  int lifo_vide(lifo_t* p)
84  {
85      /*variables locales*/
86      int ret = FALSE;
87
88      if (p->taille < 0)
89      {
90          ret = TRUE;
91      }
92      return ret;
93  }
94
95  /* FONCTION : est_pleine
96   * Vrifie si la lifo est pleine (lifo_t).
97   *
98   * entree : le pointeur de la lifo (lifo_t)
99   * sortie : le code d'erreur (int)
100 */
101 int lifo_pleine(lifo_t* p)
102 {
103     /*variables locales*/
104     int ret = FALSE;
105
106     if (p->taille == p->t_max)
```

```
    {
        ret = TRUE;
    }
    return ret;
}

/* FONCTION : free_lifo
 * Libre la lifo (lifo_t).
 *
 * entree : le pointeur de la lifo (lifo_t)
 * sortie : le code d'erreur (int)
 */
void free_lifo(lifo_t* p)
{
    type_t     tmp;

    while (! lifo_vide(p))
    {
        tmp = depiler(p);
        free(*tmp);
        *tmp = NULL;
    }

    free(p->tab);
    free(p);
    p = NULL;
}

/* FONCTION : afficher_lifo
 * Affiche la pile.
 *
 * entree : le pointeur de la lifo (lifo_t)
 */
void afficher_lifo(lifo_t * p)
{
    /*variables locales*/
    int        cour = 0;
    int        taille = p->taille;

    while (cour <= taille)
    {
        printf("%c",(*p->tab[cour])->val);
        fflush(stdout);
        cour++;
    }
    printf("\n");
}
```

MAIN DICO : main.c

```
/* Malrin Vincent
 * Laurent Valentin
 * 4/06/2013
 * DICO MAIN : main.c
 */


/*INCLUDE*/
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include "lifo.h"
#include "arbre.h"
#include "chaine.h"

/*DEFINE*/
#define TMAX 100
#define TRUE 1
#define FALSE 0

int main(int argc, char * argv[])
{
    /*variables globales*/
    abr_t **   arbre = NULL;
    FILE *     f_lecture = NULL;
    char *     chaine = NULL;
    int        compt_ch = 2;
    int        err = FALSE;

    if (argc < 2)
    {
        printf("USAGE : %s <fichier de lecture> <chaines d'insertions>\n",argv[0]);
        fflush(stdout);
    }
    else
    {
        arbre = (abr_t **)malloc(sizeof(abr_t *));
        f_lecture = fopen(argv[1],"r");
        chaine = (char *)malloc(sizeof(char)*TMAX);

        if (! f_lecture)
        {
            printf("Fichier invalide !\n");
        }
        else if (! arbre || ! chaine)
        {
            printf("Allocations impossibles");
        }

        if (arbre && chaine && f_lecture)
        {
            fgets(chaine,TMAX,f_lecture); /*lecture du fichier texte*/
            printf("\n# Affichage de la notation parenthesee :\n%s\n",chaine);
            err = lecture_abr(chaine,arbre);  /*creation de l'arbre*/

            if (! err)
            {
                if (argc > 2)
                {
                    while (compt_ch < argc)
                    {
                        err = verif_argument(argv[compt_ch]);

                        if (err)
                        {
                            printf("L'argument '%s' n'est pas un mot valide !\n",argv[compt_ch]);
                        }
                        else
                        {
```

```
                        insertion_abr(arbre,argv[compt_ch]); /*insertion*/
                    }
                    compt_ch++;
                }
            }

            printf("\n# Affichage des mots du dictionnaire :\n");
            affichage_abr(arbre); /*affichage*/

            /*liberation des ressources*/
            printf("\n# Liberation de l'arbre :\n");
            liberation_abr(arbre); /*liberation arbre*/
        }
        free(arbre);
        arbre = NULL;
        free(chaine);
        fclose(f_lecture);
    }
  }
  return 0;
}
```

MAKEFILE

```
# compilateur
CC = gcc

# options de compilation
CFLAGS = -Wall -ansi -pedantic -Wextra -g

# options de l'edition des liens
LDFLAGS = -g

# nom de l'executable a generer
EXEC = dico

# liste des fichiers objets
OBJ = main.o lifo.o arbre.o chaine.o

all : $(EXEC)

# regle de production finale
dico : $(OBJ)
  $(CC) $(OBJ) -o $@ $(LDFLAGS)

# regle de production pour chaque fichier
main.o : main.c lifo.h arbre.h chaine.h
  $(CC) -c $< $(CFLAGS)

lifo.o : lifo.c lifo.h arbre.h
  $(CC) -c $< $(CFLAGS)

arbre.o : arbre.c arbre.h lifo.h chaine.h
  $(CC) -c $< $(CFLAGS)

chaine.o : chaine.c chaine.h
  $(CC) -c $< $(CFLAGS)

# regle de suppression des fichiers
clean :
```

```
rm *.o $(EXEC)
```

1.4.3 Script de tests : tests_sdd.sh

```
#!/bin/bash

# Malrin Vincent
# Laurent Valentin
# Script shell de tests : tests_sdd.sh

#inversion
#1)chaine vide
./inversion/inversion > ./tests/tests_inversion1.txt;
#2)un caractere
./inversion/inversion a > ./tests/tests_inversion2.txt;
#3)cas general
./inversion/inversion dcba > ./tests/tests_inversion3.txt;

#validite des arguments
#1) validite du fichier
./dico/dico inconnu.txt > ./tests/tests_finvalide.txt;
#2) validite de lecriture parenthese
./dico/dico dico/abr_err1.txt > ./tests/tests_err1.txt
./dico/dico dico/abr_err2.txt > ./tests/tests_err2.txt
./dico/dico dico/abr_err3.txt > ./tests/tests_err3.txt
./dico/dico dico/abr_err4.txt > ./tests/tests_err4.txt
./dico/dico dico/abr_err5.txt > ./tests/tests_err5.txt
#3) validite des chaines a inserrer
./dico/dico dico/abr_vide.txt rien arbre7 @rbre arbré > ./tests/tests_err6.txt

#lecture et affichage
#1)arbre vide
./dico/dico dico/abr_vide.txt > ./tests/tests_aff0.txt;
#2)arbre atomique
./dico/dico dico/abr_atome.txt > ./tests/tests_aff1.txt;
#3)cas general
./dico/dico dico/abr_verbe.txt > ./tests/tests_aff2.txt;

#insersion
#arbre vide
./dico/dico dico/abr_vide.txt rien > ./tests/tests_vide.txt;

#arbre atomique
#1) insertion frere avant
./dico/dico dico/abr_atome.txt ah > ./tests/tests_atome1.txt;
#2) insertion frere apres
./dico/dico dico/abr_atome.txt oh > ./tests/tests_atome2.txt
#3) insertion fils
./dico/dico dico/abr_atome.txt eh > ./tests/tests_atome3.txt

#cas general
#1) insertion avant les freres
./dico/dico dico/abr_verbe.txt chanta > ./tests/tests_insert1.txt;
#1bis) insertion avant les freres (sans fils)
./dico/dico dico/abr_verbe.txt chantea > ./tests/tests_insert1bis.txt;
#2) insertion entre les freres
./dico/dico dico/abr_verbe.txt chantier > ./tests/tests_insert2.txt;
#2bis) insertion entre les freres (sans fils)
./dico/dico dico/abr_verbe.txt chanter > ./tests/tests_insert2bis.txt;
```

```
#3) insertion apres les freres
./dico/dico dico/abr_verbe.txt chans > ./tests/tests_insert3.txt;
#3bis) insertion apres les freres (sans fils)
./dico/dico dico/abr_verbe.txt chantez > ./tests/tests_insert3bis.txt;
#4) insertion en fils
./dico/dico dico/abr_verbe.txt chanson > ./tests/tests_insert4.txt;
#5) insertion d'un mot inclu
./dico/dico dico/abr_verbe.txt chant > ./tests/tests_insert5.txt;
#6) insertion d'un mot deja ecrit
./dico/dico dico/abr_verbe.txt chante > ./tests/tests_insert6.txt;
#7) insertion sur une cellule sans frere (apres)
./dico/dico dico/abr_verbe.txt chartes > ./tests/tests_insert7.txt;
#8) insertion sur une cellule sans frere (avant)
./dico/dico dico/abr_verbe.txt charles > ./tests/tests_insert8.txt;
#9) insertion avant racine
./dico/dico dico/abr_verbe.txt bavarde > ./tests/tests_insert9.txt;
#10) insertion apres racine
./dico/dico dico/abr_verbe.txt discute > ./tests/tests_insert10.txt;
#11) insertion fils racine
./dico/dico dico/abr_verbe.txt calvitie > ./tests/tests_insert11.txt;

#liberation memoire
#1)dico sans argument
valgrind ./dico/dico 2> ./tests/tests_free0.txt
#2)dico
valgrind ./dico/dico dico/abr_verbe.txt zorro 2> ./tests/tests_free1.txt
#3)inversion
valgrind ./inversion/inversion dcba 2> ./tests/tests_free2.txt
```

1.5 Jeux de tests

1.5.1 Cas à tester

Inversion de pile (Question 1)

Cas à tester :

1. Cas de chaine non valide (pour la lecture)
2. Cas caractère unique
3. Cas général

Validité des arguments du programme

Cas à tester :

1. Validité du fichier
2. Validité de l'écriture parenthésée
3. Validité des mots à insérer

Lecture et affichage (Question 2 et 3)

Cas à tester :

1. Cas arbre vide
2. Cas arbre atomique
3. Cas général

Insertion (Question 4)

Cas à tester :

1. Cas arbre vide
2. Cas arbre atomique
   - Cas insertion avant les frères
   - Cas insertion apres les frères
   - Cas insertion fils
3. Cas général
   - Cas insertion avant les frères
   - Cas insertion avant les frères (sans fils)
   - Cas insertion entre les frères
   - Cas insertion entre les frères (sans fils)
   - Cas insertion apres les frères
   - Cas insertion apres les frères (sans fils)
   - Cas insertion fils
   - Cas insertion d'un mot inclu
   - Cas insertion d'un mot déjà présent
   - Cas insertion sur une cellule sans frère (après)
   - Cas insertion sur une cellule sans frère (après)
   - Cas insertion avant racine
   - Cas insertion apres racine

1.5.2 Fichiers en entrée

Test de listes vides : abr_vide.txt

```
1 || ()
```

Test de cellule unique : abr_atome.txt

```
1 || (E)
```

Test du cas général : abr_verbe.txt

```
1 || (c(h(a(n(c(E)s,t(E(E,S)o(n(S)))R(m(E))T))))
```

1.5.3 Execution

Inversion (Question 1)

- Cas de chaine non valide : ./inversion/inversion

```
USAGE ./inversion/inversion <chaine a inverser>
```

- Cas caractère unique : ./inversion/inversion a

```
Pile a inverser :
a

Etat de la file :
a

Pile inversee :
a
```

- Cas général : ./inversion/inversion dcba

```
Pile a inverser :
d
c
b
a

Etat de la file :
a
b
c
d

Pile inversee :
a
b
c
d
```

Validité des arguments

- Validité du fichier : ./dico/dico inconnu.txt

```
Fichier invalide !
```

- Validité de l'écriture parenthésée 1 : ./dico/dico dico/abr_err1.txt

```

# Affichage de la notation parenthesee :
a(b(t)))

Ecriture parenthesee incorrecte !
```

- Validité de l'écriture parenthésée 2 : ./dico/dico dico/abr_err2.txt

```

# Affichage de la notation parenthesee :
(()(((()R))

Ecriture parenthesee incorrecte !
```

- Validité de l'écriture parenthésée 3 : ./dico/dico dico/abr_err3.txt

```

# Affichage de la notation parenthesee :
(a(b((b)(d))))

Ecriture parenthesee incorrecte !
```

- Validité de l'écriture parenthésée 4 : ./dico/dico dico/abr_err4.txt

```

# Affichage de la notation parenthesee :
(a(b,cd))

Ecriture parenthesee incorrecte !
```

- Validité de l'écriture parenthésée 5 : ./dico/dico dico/abr_err5.txt

```

# Affichage de la notation parenthesee :
(a(b,c,,d))

Ecriture parenthesee incorrecte !
```

- Validité des chaines à insérer : ./dico/dico dico/abr_vide.txt rien arbre7 @rbre arbré

```
1
2  # Affichage de la notation parenthesee :
3  ()
4
5  L'argument 'arbre7' n'est pas un mot valide !
6  L'argument '@rbre' n'est pas un mot valide !
7  L'argument 'arbré' n'est pas un mot valide !
8
9  # Affichage des mots du dictionnaire :
10 rieN
11
12 # Liberation de l'arbre :
13 free(N)
14 free(e)
15 free(i)
16 free(r)
```

Lecture et Affichage (Question 3)

- Lecture -> Cas arbre vide : ./dico/dico dico/abr_vide.txt

```
# Affichage de la notation parenthesee :
()


# Affichage des mots du dictionnaire :

# Liberation de l'arbre :
```

- Lecture -> Cas arbre atomique : ./dico/dico dico/abr_atome.txt

```
# Affichage de la notation parenthesee :
(E)


# Affichage des mots du dictionnaire :
E

# Liberation de l'arbre :
free(E)
```

- Lecture -> Cas général : ./dico/dico dico/abr_verbe.txt

```
# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chantE
chantEE
chantES
chantonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(E)
```

```
28 free(m)
29 free(T)
30 free(R)
31 free(n)
32 free(a)
33 free(h)
34 free(c)
```

FIGURE 1.1 – Structure crée à partir de abr_vide.txt

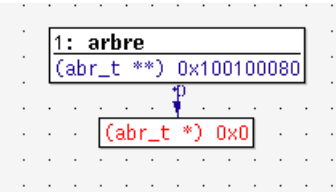


FIGURE 1.2 – Structure crée à partir de abr_atome.txt

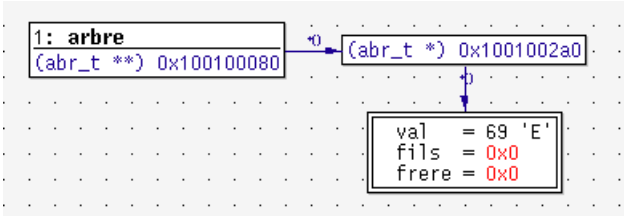
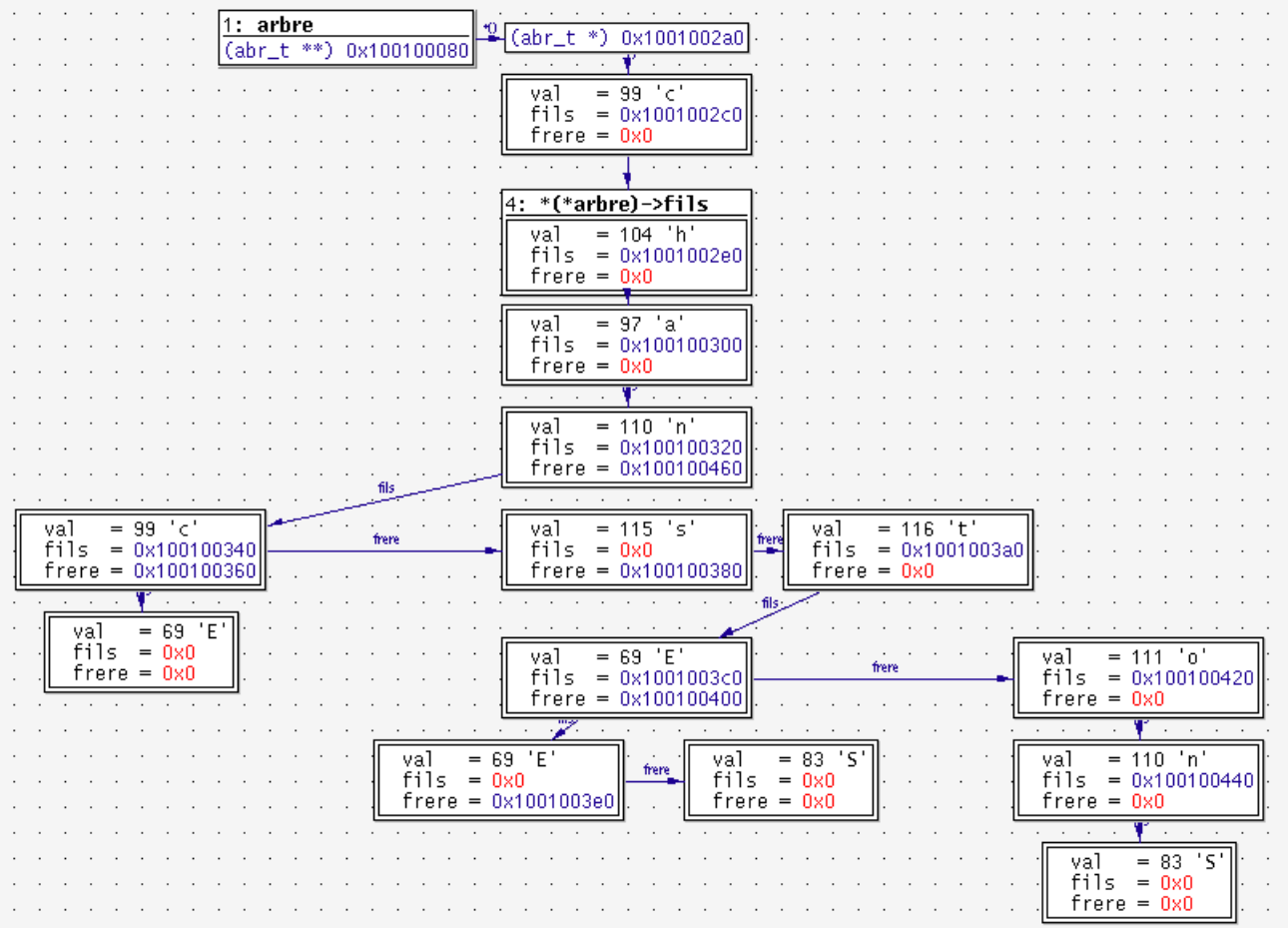


FIGURE 1.3 – Structure crée à partir de abr_verbe.txt



Insertion (Question 4)

- Insertion -> Cas arbre vide : ./dico/dico dico/abr_vide.txt abricoT

```

# Affichage de la notation parenthesee :
()


# Affichage des mots du dictionnaire :
rieN

# Liberation de l'arbre :
free(N)
free(e)
free(i)
free(r)
```

- Insertion arbre atomique -> frere avant : ./dico/dico dico/abr_atome.txt ah

```

# Affichage de la notation parenthesee :
(E)


# Affichage des mots du dictionnaire :
aH
E

# Liberation de l'arbre :
free(H)
free(E)
free(a)
```

- Insertion arbre atomique -> frere apres : ./dico/dico dico/abr_atome.txt oh

```

# Affichage de la notation parenthesee :
(E)


# Affichage des mots du dictionnaire :
E
oH

# Liberation de l'arbre :
free(H)
free(o)
free(E)
```

- Insertion arbre atomique -> fil : ./dico/dico dico/abr_atome.txt eh

```

```

```
# Affichage de la notation parenthesee :
(E)


# Affichage des mots du dictionnaire :
E
EH

# Liberation de l'arbre :
free(H)
free(E)
```

1. Insertion -> Avant les freres : ./dico/dico dico/abr_verbe.txt chanta

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chantA
chantE
chantEE
chantES
chantonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(A)
free(t)
free(s)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
```

2. Insertion -> Avant les freres (sans fils) : ./dico/dico dico/abr_verbe.txt chantea Ce mot ne veut rien dire, c'est juste pour le test.

```

```

```
# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chantE
chantEA
chantEE
chantES
chantonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(S)
free(E)
free(A)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
```

3. Insertion -> Entre les freres : ./dico/dico dico/abr_verbe.txt chantier

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chantE
chantEE
chantES
chantieR
chantonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(S)
```

```
free(E)
free(R)
free(e)
free(S)
free(n)
free(o)
free(i)
free(E)
free(t)
free(s)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
```

4. Insertion -> Entre les freres (sans fils) : ./dico/dico dico/abr_verbe.txt chanter

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chantE
chantEE
chantER
chantES
chantonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(S)
free(R)
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
```

```
free(c)
```

5. Insertion -> Apres les freres : ./dico/dico dico/abr_verbe.txt chans

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chanS
chantE
chantEE
chantES
chantonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(S)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
```

6. Insertion -> Apres les freres (sans fils) : ./dico/dico dico/abr_verbe.txt chant

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chantE
chantEE
chantES
chantEZ
chantonS
chaR
```

```
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(Z)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
```

7. Insertion -> fils : ./dico/dico dico/abr_verbe.txt chansons

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chansoN
chantE
chantEE
chantES
chantonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(N)
free(o)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(E)
free(m)
```

```
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
```

8. Insertion -> Mot inclu : ./dico/dico dico/abr_verbe.txt chant

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chanT
chanTE
chanTEE
chanTES
chanTonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(T)
free(s)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
```

9. Insertion -> Mot déjà présent : ./dico/dico dico/abr_verbe.txt chantons

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chantE
```

```
chantEE
chantES
chantonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
```

10. Insertion -> Cellule sans frère (après) : ./dico/dico dico/abr_verbe.txt chartres

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chantE
chantEE
chantES
chantonS
chaR
chaRmE
chaRteS
chaT

# Liberation de l'arbre :
free(E)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(E)
```

```
free(S)
free(e)
free(t)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
```

11. Insertion -> Cellule sans frère (avant) : ./dico/dico dico/abr_verbe.txt charles

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chantE
chantEE
chantES
chantonS
chaR
chaRleS
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(S)
free(e)
free(E)
free(m)
free(l)
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
```

12. Insertion -> Avant racine : ./dico/dico dico/abr_verbe.txt bavarde

```

```

```
# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
bavardE
chancE
chantE
chantEE
chantES
chantonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(d)
free(r)
free(a)
free(v)
free(a)
free(E)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(c)
free(b)
```

13. Insertion -> Après racine : ./dico/dico dico/abr_verbe.txt discute

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
chancE
chantE
chantEE
chantES
chantonS
chaR
chaRmE
```

```
chaT
discutE

# Liberation de l'arbre :
free(E)
free(S)
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(E)
free(t)
free(u)
free(c)
free(s)
free(i)
free(d)
free(c)
```

14. Insertion -> Fils racine : ./dico/dico dico/abr_verbe.txt calvitie

```

# Affichage de la notation parenthesee :
(c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))


# Affichage des mots du dictionnaire :
calvitiE
chancE
chantE
chantEE
chantES
chantonS
chaR
chaRmE
chaT

# Liberation de l'arbre :
free(E)
free(i)
free(t)
free(i)
free(v)
free(l)
free(E)
free(S)
```

```
free(E)
free(S)
free(n)
free(o)
free(E)
free(t)
free(s)
free(c)
free(E)
free(m)
free(T)
free(R)
free(n)
free(a)
free(h)
free(a)
free(c)
```

Libération mémoire

- Valgrind -> DICO sans argument : valgrind ./dico/dico

```
==30969== Memcheck, a memory error detector
==30969== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==30969== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==30969== Command: ./dico/dico
==30969==
==30969==
==30969== HEAP SUMMARY:
==30969==    in use at exit: 0 bytes in 0 blocks
==30969==  total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==30969==
==30969== All heap blocks were freed -- no leaks are possible
==30969==
==30969== For counts of detected and suppressed errors, rerun with: -v
==30969== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

- Valgrind -> DICO : valgrind ./dico/dico dico/abr_verbe.txt zorro

```
==31301== Memcheck, a memory error detector
==31301== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==31301== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==31301== Command: ./dico/dico dico/abr_verbe.txt zorro
==31301==
==31301== Invalid read of size 8
==31301==   at 0x400F94: liberation_abr (arbre.c:137)
==31301==   by 0x400A32: main (main.c:81)
==31301== Address 0x4c2b7d0 is 16 bytes inside a block of size 24 free'd
==31301==   at 0x4A0595D: free (vg_replace_malloc.c:366)
==31301==   by 0x401049: liberation_abr (arbre.c:157)
==31301==   by 0x400A32: main (main.c:81)
==31301==
==31301==
==31301== HEAP SUMMARY:
==31301==    in use at exit: 0 bytes in 0 blocks
==31301==  total heap usage: 32 allocs, 32 frees, 2,476 bytes allocated
==31301==
==31301== All heap blocks were freed -- no leaks are possible
==31301==
==31301== For counts of detected and suppressed errors, rerun with: -v
==31301== ERROR SUMMARY: 23 errors from 1 contexts (suppressed: 6 from 6)
```

- Valgrind -> INVERSION : valgrind ./inversion/inversion dcba

```
==31682== Memcheck, a memory error detector
==31682== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==31682== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==31682== Command: ./inversion/inversion dcba
==31682==
==31682==
==31682== HEAP SUMMARY:
==31682==    in use at exit: 0 bytes in 0 blocks
==31682==  total heap usage: 4 allocs, 4 frees, 240 bytes allocated
```

```
==31682==
==31682== All heap blocks were freed -- no leaks are possible
==31682==
==31682== For counts of detected and suppressed errors, rerun with: -v
==31682== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```