

# Table des matières

1.1	Objet du TP	2
1.2	Structure de données	2
1.3	Organisation du code source	3
1.4	Codes Sources	4
1.4.1	Question 1 : Inversion d'une pile	4
1.4.2	Question 2,3 et 4 : Gestion du dictionnaire arborescent	14
1.4.3	Script de tests : tests_sdd.sh	29
1.5	Jeux de tests	31
1.5.1	Cas à tester	31
1.5.2	Fichiers en entrée	32
1.5.3	Execution	33

## Compte rendu du TP2 de Structure de Données

LAURENT Valentin, MALRIN Vincent

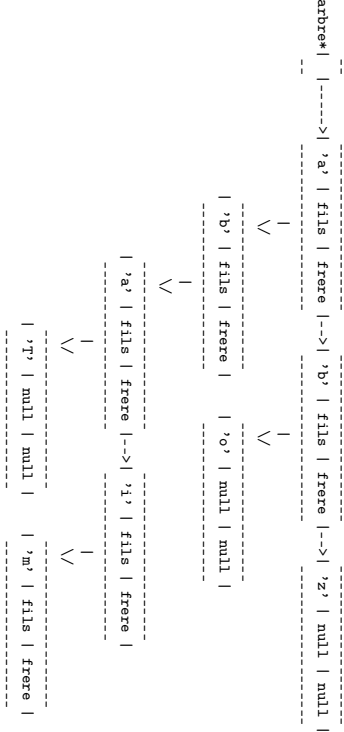
6 juin 2013

### 1.1 Objet du TP

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

### 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'apparaît 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 :

- enfiler
- defiler
- init\_fifo
- fifo\_vide
- fifo\_pleine
- free\_fifo
- afficher\_fifo

**chaîne.h**

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

**chaîne.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 iffo.c : iffo.h

```
1  /* Malrén Vincent
2  * Laurent Valentin
3  * 4/06/2013
4  * LIFO : tifo.h
5  */
6
7  /*GABRIEL*/
8  #ifndef LIFO_H
9  #define LIFO_H
10
11  /*INCLUDE*/
12  #include <stdio.h>
13  #include <stdlib.h>
14  #include <string.h>
15  #include "iffo.h"
16
17  typedef struct lifo
18  {
19      int t_max;
20      int taille;
21      type_t * tab;
22  } lifo_t;
23
24  /* FONCTION : empiler
25   * Empile un element dans la lifo (tifo_t).
26   *
27   * entree : le pointeur de la lifo (tifo_t).
28   * la valeur a insérer (type_t *)
29   * sortie : le code d'erreur (int)
30   */
31  int empiler(lifo_t * p, type_t val);
32
33  /* FONCTION : depiler
34   * Depile un element dans la lifo (tifo_t).
35   *
36   * entree : le pointeur de la lifo (tifo_t)
37   * la valeur a insérer (type_t *)
38   * sortie : le code d'erreur (int)
39   */
40  type_t depiler(lifo_t * p);
41
42  /* FONCTION : init_lifo
43   * Initialise la lifo (tifo_t).
44   *
45   * entree : le pointeur de la lifo (tifo_t)
46   * la taille maximale (int)
47   * sortie : le code d'erreur (int)
48   */
49  int init_lifo(lifo_t * p, int t_max);
50
51  /* FONCTION : est_vide
52   * Vérifie si la lifo est vide (tifo_t).
53   *
```

```
54  * entree : le pointeur de la lifo (tifo_t)
55  * sortie : le code d'erreur (int)
56  */
57  int lifo_vide(lifo_t * p);
58
59  /* FONCTION : est_pleine
60   * Vérifie si la lifo est pleine (tifo_t).
61   *
62   * entree : le pointeur de la lifo (tifo_t)
63   * sortie : le code d'erreur (int)
64   */
65  int lifo_pleine(lifo_t * p);
66
67  /* FONCTION : free_lifo
68   * Libère la lifo (tifo_t).
69   *
70   * entree : le pointeur de la lifo (tifo_t)
71   * sortie : le code d'erreur (int)
72   */
73  void free_lifo(lifo_t * p);
74
75  /* FONCTION : afficher_lifo
76   * Affiche la pile.
77   *
78   * entree : le pointeur de la lifo (tifo_t)
79   */
80  void afficher_lifo(lifo_t * p);
81
82  /*FONCTION : inversion_pile
83   * Porte bien son nom.
84   *
85   * entree : le pointeur de la pile (tifo_t).
86   */
87
88  void inversion_pile(lifo_t * p);
89
90  /*FONCTION : lecture_pile
91   * Lit une chaîne de caractère et empile chaque caractère.
92   *
93   * entree : la pile
94   * une chaîne de caractère
95   */
96
97  void lecture_pile(lifo_t * p, char * ch);
98
99
100 #endif
```

GESTION PILE : iffo.c

```
1  /* Malrén Vincent
2  * Laurent Valentin
3  * 4/06/2013
4  * INVERSION LIFO : tifo.h
5  */
6
7  /*INCLUDE*/
8  #include "iffo.h"
9  #include "iffo.h"
10
```

```

11  /*DEFINE*/
12  #define TRUE 1
13  #define FALSE 0
14
15  int empiler(lifo_t * p, type_t val)
16  {
17      /*variables locales*/
18      int err = FALSE; /*code erreur*/
19
20      /*teste si la lifo est pleine*/
21      if (p->t_max > (p->taille + 1))
22      {
23          p->taille++; /*on incremente la taille*/
24          p->tab[p->taille]=val; /*insertion de la valeur dans la pile*/
25          err=TRUE; /*on valide le code erreur*/
26      }
27      return err;
28  }
29
30  type_t depiler(lifo_t * p)
31  {
32      /*variables locales*/
33      type_t val;
34
35      if (!lifo_vide(p) /*si pile non vide*/
36      {
37          val=p->tab[p->taille]; /*on recupere la valeur*/
38          p->taille--; /*on decremente la taille*/
39      }
40      return val;
41  }
42
43
44  int init_lifo(lifo_t * p, int t_max)
45  {
46      /*variables locales*/
47      int err = TRUE;
48
49      /*initialisation de la lifo*/
50      p->t_max = t_max; /*initialisation de la taille max*/
51      p->taille = -1; /*la taille allouee vaut -1 -> pile vide*/
52      p->tab = (type_t*)malloc(t_max*sizeof(type_t));
53
54      if (p->tab) /*on verifie l'allocation*/
55      {
56          err = FALSE;
57      }
58      return err;
59  }
60
61  int lifo_vide(lifo_t* p)
62  {
63      /*variables locales*/
64      int ret = FALSE;
65
66      if (p->taille < 0) /*on verifie l'indice de pile*/
67      {
68          ret = TRUE;
69      }

```

```

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      free(p->tab); /*libere l'espace d'empilage*/
88      free(p);
89  }
90
91  void afficher_lifo(lifo_t * p)
92  {
93      /*variables locales*/
94      int cour = 0; /*parcours de pile*/
95      int taille = p->taille; /*parcours de l'espace d'empilage*/
96
97      while (cour <= taille) /*boucle de parcours*/
98      {
99          printf("%c\n", (p->tab[cour]));
100          fflush(stdout);
101          cour++;
102      }
103      printf("\n");
104  }
105
106  void inversion_pile(lifo_t * p)
107  {
108      /*variables locales*/
109      lifo_t * f = NULL; /*file*/
110      type_t tmp; /*valeurs a afficher*/
111      int err; /*erreur*/
112
113      /*initialisation*/
114      f = (lifo_t *)malloc(sizeof(lifo_t));
115      err = init_lifo(f, p->t_max);
116
117      if (! err)
118      {
119          printf("Pile a inverser : \n");
120          afficher_lifo(p);
121
122          /*parcours de la pile*/
123          while (!lifo_vide(p))
124          {
125              tmp = depiler(p); /*passage des valeurs*/
126              enfiler(f, tmp);
127          }
128

```

```

1290      printf("Etat de la file :\n");
1291      afficher_fifo(f);
1292
1293      while (! fifo_vide(f))
1294      {
1295          tmp = defiler(f); //passage des valeurs*/
1296          empiler(p,tmp);
1297      }
1298      /*liberation de la file*/
1299      free_fifo(f);
1300  }
1301
1302  printf("Pile inversee :\n");
1303  afficher_fifo(p);
1304  }
1305
1306  void lecture_pile(lifo_t * p, char * ch)
1307  {
1308      /*variables locales*/
1309      int      compt = 0;
1310
1311      while (ch[compt] != '\0')
1312      {
1313          empiler(p,ch[compt]);
1314          compt++;
1315      }
1316  }

```

## EN TETE 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 : empiler
26  * Empile un element dans la file (fifo_t).
27  */

```

```

28  * entree : le pointeur de la file (fifo_t)
29  *          la valeur a inserer (type_t)
30  */
31
32  int enfiler(fifo_t * f, type_t val);
33
34  /* FONCTION : defiler
35  * Defile un element dans la file (fifo_t).
36  * entree : le pointeur de la file (fifo_t)
37  *          la valeur a inserer (type_t)
38  */
39
40  type_t defiler(fifo_t * f);
41
42  /* FONCTION : init_fifo
43  * Initialise la file (fifo_t).
44  * entree : le pointeur de la file (fifo_t)
45  *          la taille maximale (int)
46  */
47
48  int init_fifo(fifo_t * f, int t_max);
49
50  /* FONCTION : est_vide
51  * Verifie si la fifo est vide (fifo_t).
52  * entree : le pointeur de la fifo (fifo_t)
53  * sortie : le code d'erreur (int)
54  */
55
56  int fifo_vide(fifo_t * f);
57
58  /* FONCTION : est_pleine
59  * Verifie si la fifo est pleine (fifo_t).
60  * entree : le pointeur de la fifo (fifo_t)
61  * sortie : le code d'erreur (int)
62  */
63
64  int fifo_pleine(fifo_t * f);
65
66  /* FONCTION : free_fifo
67  * Libere la fifo (fifo_t).
68  * entree : le pointeur de la fifo (fifo_t)
69  * sortie : le code d'erreur (int)
70  */
71
72  void free_fifo(fifo_t * f);
73
74  /* FONCTION : afficher_fifo
75  * Affiche la fifo (fifo_t).
76  * entree : le pointeur de la fifo (fifo_t)
77  */
78
79  void afficher_fifo(fifo_t * f);
80
81
82
83
84
85
86

```

```

87 //endif
GESTION FILE : fifo.c

1  /* Halryn 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 locales*/
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 locales*/
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;

```

10

```

57     }
58     return err;
59 }
60
61 int fifo_vide(fifo_t * f)
62 {
63     /*variables locales*/
64     int ret = FALSE;
65
66     if ((f->in - f->out) == 0) /*file vide -> indice aux meme endroits*/
67     {
68         ret = TRUE;
69     }
70     return ret;
71 }
72
73 int fifo_pleine(fifo_t * f)
74 {
75     /*variables locales*/
76     int ret = FALSE;
77
78     if ((f->in - f->out) == f->t_max) /*file pleine -> la distance des indices*/
79     {
80         ret = TRUE;
81     }
82     return ret;
83 }
84
85 void free_fifo(fifo_t* f)
86 {
87     free(f->tab); /*libere l'espace d'enfilage*/
88     free(f);
89 }
90
91 void afficher_fifo(fifo_t * f)
92 {
93     /*variables locales*/
94     int cour = f->out + 1; /*parcours*/
95     int nb = f->in - f->out; /*nb d'elements*/
96
97     while (nb > 0)
98     {
99         printf("%c\n",f->tab[cour]);
100         cour++; /*avance dans la file*/
101         cour = cour % f->t_max;
102         nb--;
103     }
104     printf("\n");
105 }
106
MAIN INVERSION : main.c
1  /* Halryn Vincent
2  * Laurent Valentin
3  * 4/06/2013
4  * INVERSION MAIN : main.c
5  */
6
7  /*INCLUDE*/

```

11

```

8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11 #include "lifo.h"
12 #include "fifo.h"
13
14 /*DEFINE*/
15 #define TMAX 100
16 #define TRUE 1
17 #define FALSE 0
18
19 int main(int argc, char ** argv)
20 {
21     /*variables locales*/
22     lifo_t * p = NULL;
23     int err = TRUE; /*erreur d'allocation*/
24
25     /*test des arguments*/
26     if (argc < 2)
27     {
28         printf("USAGE %s <chaîne a inverser>\n", argv[0]);
29     }
30     else
31     {
32         /*initialisation*/
33         p = (lifo_t *)malloc(sizeof(lifo_t));
34         err = init_lifo(p, TMAX);
35
36         if (!err && p)
37         {
38             lecture_pile(p, argv[1]);
39             inversion_pile(p);
40
41             /*libération des ressources*/
42             free_lifo(p);
43             p = NULL;
44         }
45         return 0;
46     }
47 }

```

# MAKEFILE

```

1  # compilateur
2  CC = gcc
3
4  # options de compilation
5  CFLAGS = -Wall -ansi -pedantic -Wextra -g
6
7  # options de l'edition des liens
8  LDFLAGS = -g
9
10 # nom de l'executable a generer
11 EXEC = inversion
12
13 # liste des fichiers objets
14 OBJ = main.o lifo.o fifo.o
15
16 all : $(EXEC)
17

```

```

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)
35

```

1.4.2 Question 2,3 et 4 : Gestion du dictionnaire arborescent  
EN TÊTE arbre.c : arbre.h

```
1  /* Maîtr'n Vincent
2   * Laurent Valentin
3   * 4/06/2013
4   * ARBRE : arbre.h
5   */
6
7  /*GARDIEN*/
8  #ifndef ARBRE_H
9  #define ARBRE_H
10
11 /*INCLUDE*/
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include "ctype.h"
15 #include "string.h"
16
17 typedef struct abr
18 {
19     char      val;
20     struct abr * fils;
21     struct abr * frere;
22 } abr_t;
23
24 /* FONCTION : construction
25  * Construit un noeud à partir de sa valeur (noeud_t).
26  */
27 * entree : la valeur (char)
28 * sortie : le code erreur (int)
29  */
30
31 int construction(char val, abr_t ** n);
32
33 /* FONCTION : lecture_abr
34  * Lit la chaîne de caractère (écriture parenthésée) et construit l'arbre.
35  */
36 * entree : chaîne de caractère (char *)
37  * le pointeur sur l'arbre (noeud_t *)
38 * sortie : erreur (int).
39  */
40
41 int lecture_abr(char * ch, abr_t ** r);
42
43 /* FONCTION : liberation_abr
44  * Libère les ressources.
45  */
46 * entree : la racine (abr_t ** r)
47 * sortie : erreur (int)
48  */
49
50 int liberation_abr(abr_t ** r);
51
52 /* FONCTION : affichage_abr
53  * Affiche les mots contenus dans l'arbre.
54  */
55 * entree : la racine (abr_t ** r)
```

```
56 * sortie : erreur (int)
57  */
58
59 int affichage_abr(abr_t ** r);
60
61 /* FONCTION : insertion_abr
62  * Insère une chaîne de caractères dans l'arbre trié.
63  */
64 * entree : la racine (abr_t ** r)
65  * la chaîne à insérer (char * ch)
66 * sortie : erreur (int)
67  */
68
69 int insertion_abr(abr_t ** r, char * ch);
70
71 #endif
```

GESTION ARBRE : arbre.c

```
1  /* Maîtr'n Vincent
2   * Laurent Valentin
3   * 4/06/2013
4   * ARBRE : arbre.c
5   */
6
7  /*INCLUDE*/
8  #include "lifo.h"
9  #include "chaîne.h"
10
11 /*DEFINE*/
12 #define TRUE 1
13 #define FALSE 0
14 #define TMAX 50
15
16 /* FONCTION : construction
17  * Construit un noeud à partir de sa valeur (noeud_t).
18  */
19 * entree : la valeur (char)
20 * sortie : le code erreur (int)
21  */
22
23 int construction(char val, abr_t ** a)
24 {
25     /*variables locales*/
26     int      err = TRUE;
27
28     /*allocation memoire*/
29     *a = (abr_t *)malloc(sizeof(abr_t));
30
31     /*verification*/
32     if (*a)
33     {
34         err = FALSE;
35         (*a)>val = val;
36         (*a)>fils = NULL;
37         (*a)>frere = NULL; /*initialisation du frere*/
38
39         return err;
40     }
41 }
```



```

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 chaîne*/
48     int compt_parenthese = 0; /*compteur de parenthèses*/
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 initialisée !\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 chaîne*/
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 chaîne*/
91                     compt_parenthese++; /*on incremente le compteur de parenthèses*/
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 chaîne*/
95                     break;
96
97                     case ')':
98                         if (compt_parenthese > 0)
99                             cour = depiler(p); /*on depile l'adr*/
100                             compt++; /*on avance dans la chaîne*/
101
102     compt_parenthese--; /*on decremente le compteur de parenthèses*/
103     break;
104
105     case ',':
106         compt++; /*on avance dans la chaîne*/
107         break;
108
109     default:
110         cour = &(*cour)->frere; /*on deplace cour sur le frere*/
111         construction(ch[compt],cour); /*on construit une nouvelle cellule*/
112         compt++; /*on avance dans la chaîne*/
113     }
114 }
115 free_lifo(p); /*libération des ressources*/
116
117 return err;
118 }
119
120 int liberation_abr(abr_t **r)
121 {
122     lifo_t * p = (lifo_t*)malloc(sizeof(lifo_t)); /*pile*/
123     abr_t ** cour = r; /*pointeur de parcours*/
124     int tmp = NULL; /*pointeur de libérations*/
125     int fin = FALSE; /*fin de parcours (pile vide)*/
126     int err = FALSE; /*erreur d'allocation*/
127
128     err = init_lifo(p,TMAX); /*initialisation de la pile*/
129
130     if (err) /*verification d'allocation*/
131     {
132         printf("Erreur : pile non initialisée !");
133     }
134
135     while ((! fin)) /*boucle de parcours*/
136     {
137         while (*cour) /*parcours positif*/
138         {
139             empiler(p,cour); /*si il a un fils : on empile l'adr*/
140             cour = &(*cour)->fils;
141         }
142         if (! lifo_vide(p))
143         {
144             cour = depiler(p); /*feuille : on depile les cellule parcourues*/
145             tmp = cour;
146
147             if ((*cour)->frere) /*si il a un frere : on empile l'adr*/
148             {
149                 empiler(p,cour);
150                 cour = &(*cour)->frere;
151             }
152             else
153             {
154                 cour = &(*cour)->frere; /*si on on libere la memoire*/
155                 printf("Free(%c)\n",(*tmp)->val); /*affichage pour tests*/
156                 fflush(stdout);
157                 free(*tmp);
158                 *tmp = NULL;
159             }
160         }
161     }

```

```

1601     }
1602     else
1603     {
1604         fin = TRUE;      /*fin de parcours : pile vide*/
1605     }
1606 }
1607
1608 free_lifo(p); /*libération des ressources*/
1609
1610 return err;
1611 }
1612
1613 int affiche_abr(abr_t ** r)
1614 {
1615     lifo_t * p = (lifo_t *)malloc(sizeof(lifo_t)); /*pile*/
1616     abr_t ** cour = r;      /*pointeur de parcours*/
1617     int fin = FALSE;      /*fin de parcours*/
1618     int err = FALSE;      /*erreur d'allocation*/
1619
1620     err = init_lifo(p,TMAX); /*initialisation de la pile*/
1621
1622     if (err) /*erreur d'allocation*/
1623     {
1624         printf("Erreur : pile non initialisée\n");
1625     }
1626
1627     while ((i fin) && (i err)) /*boucle de parcours*/
1628     {
1629         while (*cour) /*parcours en profondeur*/
1630         {
1631             empiler(p,cour);
1632
1633             /*verifie si on parcourt une majuscule*/
1634             if (((*cour)->val > 64) && ((*cour)->val < 91))
1635             {
1636                 afficher_lifo(p); /*si majuscule : on affiche le mot parcouru*/
1637             }
1638             cour = &(*cour)->fils; /*on avance dans la structure (mot suivant)*/
1639         }
1640         if (! lifo_vide(p)) /*on depile jusqu'au frere suivant*/
1641         {
1642             cour = depiler(p);
1643             cour = &(*cour)->frere;
1644         }
1645         else /*fin de parcours : pile vide*/
1646         {
1647             fin = TRUE;
1648         }
1649     }
1650
1651     free_lifo(p); /*libération des ressources*/
1652
1653     return err;
1654 }
1655
1656 int insertion_abr(abr_t ** r, char * ch)
1657 {
1658     abr_t ** prec = r;      /*pointeur sur precedent (parcours)*/
1659     int err = FALSE;      /*erreur d'allocation*/
1660     int i = 0;
1661 }

```

```

219     abr_t * tmp = NULL;      /*valeur temporaire (insertion)*/
220     char char_ch;      /*caractere 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 memes...*/
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                 if(ch[i+1] == '\0') /*gestion dernière lettre*/
250                 {
251                     construction(toupper(ch[i]),prec);
252                 }
253                 else
254                 {
255                     construction(ch[i],prec);
256                 }
257                 (*prec)->frere = tmp; /*...qui devient le frere*/
258                 prec = &(*prec)->fils;
259                 i++;
260             }
261             else
262             {
263                 if(ch[i+1] == '\0') /*gestion dernière lettre*/
264                 {
265                     construction(toupper(ch[i]),prec);
266                 }
267                 else
268                 {
269                     construction(ch[i],prec); /*on insere le reste du mot*/
270                 }
271                 prec = &(*prec)->fils;
272                 i++;
273             }
274         }
275     }
276 }
277

```

```

278     }
279     return err;
280 }

```

# EN TÊTE chaine.c : chaine.h

```

1  /* Matrin 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 parenthesesee.
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  /* Matrin 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;
18     int compt_parenthese = 0; /*compteur de parentheses*/
19     int err = FALSE;
20     int abr_vide = FALSE; /*cas particulier : 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                     compt_parenthese++;
37                 compt++;
38             else
39             {
40                 err = TRUE;
41             }
42             case ')': /*si le caractere est ')' -> une lettre ou ')'*/
43                 if (isalpha(ch[compt + 1]) || ch[compt + 1] == '?')
44                     break;
45                 if (isalpha(ch[compt + 1]) || ch[compt + 1] == '?')
46                 {
47                     compt_parenthese--;
48                     compt++;
49                 }
50                 else
51                 {
52                     err = TRUE;
53                 }
54                 break;
55                 case ',': /*si le caractere est ',' -> une lettre ou ','*/
56                     if (isalpha(ch[compt + 1]) || ch[compt + 1] == '?')
57                     {
58                         compt_parenthese--;
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                     }

```

```

83     }
84
85     if (ch[compt] == ')')
86     {
87         compt_parenthese--;
88     }
89
90     if (compt_parenthese != 0) /*les parentheses '(' et ')' doivent etre de nbr egal*/
91     {
92         err = TRUE;
93     }
94
95     if (err)
96     {
97         printf("Ecriture parentheses incorrecte !\n");
98     }
99
100     return err;
101 }
102
103 int verif_argument(char * ch)
104 {
105     /*variables locales*/
106     int compt = 0;
107     int err = FALSE;
108
109     /*compteur dans la chaine*/
110     while(ch[compt] != '\0' && ! err)
111     {
112         if (! isalpha(ch[compt]))
113         {
114             err = TRUE;
115         }
116         compt++;
117     }
118     return err;
119 }

```

## EN TETE lifo.c : lifo.h

```

1  /* lifo.h
2  */
3
4  /*GABRIEL*/
5  #ifndef LIFO_H
6  #define LIFO_H
7
8  /*INCLUDE*/
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include "arbre.h"
13
14 typedef abr_t ** type_t;
15
16 typedef struct lifo
17 {
18     int t_max;
19     int taille;
20     type_t * tab;

```

```

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

```

```

80 void afficher_lifo(lifo_t * p);
81
82 /*FONCTION : inversion_pile
83  * Porte bien son nom.
84  *
85  * entree : le pointeur de la pile (lifo_t).
86  */
87
88 void inversion_pile(lifo_t * p);
89 #endif

```

## GESTION PILE : lifo.c

```

1  /* lifo.c
2  */
3
4  #include "lifo.h"
5
6  /*DEFINE*/
7  #define TRUE 1
8  #define FALSE 0
9
10 /* FONCTION : empiler
11  * Empile un element dans la lifo (lifo_t).
12  *
13  * entree : le pointeur de la lifo (lifo_t)
14  *          la valeur a insérer (type_t *)
15  * sortie : le code d'erreur (int)
16  */
17 int empiler(lifo_t * p, type_t val)
18 {
19     /*variables locales*/
20     int err=FALSE; /*code erreur*/
21
22     /*teste si la lifo est pleine*/
23     if (p->t_max > (p->taille + 1))
24     {
25         p->taille++;
26         p->tab[p->taille]=val;
27         err=TRUE;
28     }
29     return err;
30 }
31
32 /* FONCTION : empiler
33  * Empile un element dans la lifo (lifo_t).
34  *
35  * entree : le pointeur de la lifo (lifo_t)
36  *          la valeur a insérer (type_t *)
37  * sortie : le code d'erreur (int)
38  */
39 type_t depiler(lifo_t * p)
40 {
41     /*variables locales*/
42     type_t val;
43
44     if (!lifo_vide(p))
45     {
46         val=p->tab[p->taille];
47         p->taille--;

```

```

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

```

```

107 {
108     ret = TRUE;
109 }
110 return ret;
111 }
112
113 /* FONCTION : free_lifo
114    * Libère la lifo (lifo_t).
115
116    * entree : le pointeur de la lifo (lifo_t)
117    * sortie : le code d'erreur (int)
118    */
119 void free_lifo(lifo_t* p)
120 {
121     type_t tmp;
122
123     while (! lifo_vide(p))
124     {
125         tmp = depiler(p);
126         free(*tmp);
127         *tmp = NULL;
128     }
129
130     free(p->tab);
131     free(p);
132     p = NULL;
133 }
134
135 /* FONCTION : afficher_lifo
136    * Affiche la pile.
137
138    * entree : le pointeur de la lifo (lifo_t)
139    */
140 void afficher_lifo(lifo_t * p)
141 {
142     /*variables locales*/
143     int cour = 0;
144     int taille = p->taille;
145
146     while (cour <= taille)
147     {
148         printf("%c", (*p->tab[cour])>>val);
149         fflush(stdout);
150         cour++;
151     }
152     printf("\n");
153 }
154
155 MAIN DICO : main.c
156
157 1 /* Malrin Vincent
158 2  * Laurent Valentin
159 3  * 4/06/2013
160 4  * DICO MAIN : main.c
161 5  */
162
163 6
164 7
165 8 /*INCLUDE*/
166 9 #include <stdio.h>
167 10 #include <stdlib.h>

```

```

11 #include <string.h>
12 #include "lifo.h"
13 #include "arbre.h"
14 #include "chaîne.h"
15
16 /*DEFINE*/
17 #define TMAX 100
18 #define TRUE 1
19 #define FALSE 0
20
21 int main(int argc, char * argv[])
22 {
23     /*variables globales*/
24     abr_t ** arbre = NULL;
25     FILE * f_lecture = NULL;
26     char * chaîne = NULL;
27     int compt_ch = 2;
28     int err = FALSE;
29
30     if (argc < 2)
31     {
32         printf("USAGE : %s <fichier de lecture> <chaines d'insertions>\n", argv[0]);
33         fflush(stdout);
34     }
35     else
36     {
37         arbre = (abr_t**)malloc(sizeof(abr_t *));
38         f_lecture = fopen(argv[1], "r");
39         chaîne = (char *)malloc(sizeof(char)*TMAX);
40
41         if (! f_lecture)
42         {
43             printf("Fichier invalide !\n");
44         }
45         else if (! arbre || ! chaîne)
46         {
47             printf("Allocations impossibles");
48         }
49
50         if (arbre && chaîne && f_lecture)
51         {
52             fgets(chaîne, TMAX, f_lecture); /*lecture du fichier texte*/
53             printf("\nAffichage de la notation parenthésée :\n%s\n", chaîne);
54             err = lecture_abr(chaîne, arbre); /*creation de l'arbre*/
55
56             if (! err)
57             {
58                 if (argc > 2)
59                 {
60                     while (compt_ch < argc)
61                     {
62                         err = verif_argument(argv[compt_ch]);
63
64                         if (err)
65                         {
66                             printf("L'argument '%s' n'est pas un mot valide !\n", argv[compt_ch]);
67                         }
68                         else
69                         {

```

```

70         insertion_abr(arbre,argv[comp_ch]); /*insertion*/
71     }
72     compt_ch++;
73 }
74
75 printf("\n# Affichage des mots du dictionnaire :\n");
76 affichage_abr(arbre); /*affichage*/
77
78 /*liberation des ressources*/
79 printf("\n# Liberation de l'arbre :\n");
80 liberation_abr(arbre); /*liberation arbre*/
81
82 }
83 free(arbre);
84 arbre = NULL;
85 free(chaine);
86 fclose(f_lecture);
87
88 }
89 return 0;
90 }

```

## MAKEFILE

```

1  # compilateur
2  CC = gcc
3
4  # options de compilation
5  CFLAGS = -Wall -ansi -pedantic -Wextra -g
6
7  # options de l'edition des liens
8  LDPLAGS = -g
9
10 # nom de l'executable a generer
11 EXEC = dico
12
13 # liste des fichiers objets
14 OBJ = main.o lifo.o arbre.o chaine.o
15
16 all : $(EXEC)
17
18 # regle de production finale
19 dico : $(OBJ)
20 $(CC) $(OBJ) -o $@ $(LDPLAGS)
21
22 # regle de production pour chaque fichier
23 main.o : main.c lifo.h arbre.h chaine.h
24 $(CC) -c $(CFLAGS)
25
26 lifo.o : lifo.c lifo.h arbre.h
27 $(CC) -c $(CFLAGS)
28
29 arbre.o : arbre.c arbre.h lifo.h chaine.h
30 $(CC) -c $(CFLAGS)
31
32 chaine.o : chaine.c chaine.h
33 $(CC) -c $(CFLAGS)
34
35 # regle de suppression des fichiers
36 clean :

```

```

37 rm *.o $(EXEC)

```

## 1.4.3 Script de tests : tests\_sdd.sh

```

1  #!/bin/bash
2
3  # Matryn Vincent
4  # Laurent Valentin
5  # Script shell de tests : tests_sdd.sh
6
7  #inversion
8
9  ./inversion/inversion > ./tests/tests_inversion1.txt;
10 #2) un caractere
11 ./inversion/inversion a > ./tests/tests_inversion2.txt;
12 #3) cas general
13 ./inversion/inversion dcba > ./tests/tests_inversion3.txt;
14
15 #validee des arguments
16
17 ./dico/dico incommu.txt > ./tests/tests_finvalide.txt;
18 #2) validee de l'écriture parenthese
19 ./dico/dico dico/abr_err1.txt > ./tests/tests_err1.txt
20 ./dico/dico dico/abr_err2.txt > ./tests/tests_err2.txt
21 ./dico/dico dico/abr_err3.txt > ./tests/tests_err3.txt
22 ./dico/dico dico/abr_err4.txt > ./tests/tests_err4.txt
23 ./dico/dico dico/abr_err5.txt > ./tests/tests_err5.txt
24 #3) validee des chaines a inserer
25 ./dico/dico dico/abr_vide.txt rien arbre7 @bre arbre > ./tests/tests_err6.txt
26
27 #lecture et affichage
28
29 ./dico/dico dico/abr_vide.txt > ./tests/tests_af10.txt;
30 #1) arbre vide
31
32 ./dico/dico dico/abr_atome.txt > ./tests/tests_aff1.txt;
33 #3) cas general
34 ./dico/dico dico/abr_verbe.txt > ./tests/tests_aff2.txt;
35
36 #inversion
37
38 ./dico/dico dico/abr_vide.txt rien > ./tests/tests_vide.txt;
39 #arbre vide
40
41 ./dico/dico dico/abr_vide.txt ah > ./tests/tests_atome1.txt;
42 #1) insertion frere avant
43 ./dico/dico dico/abr_atome.txt ah > ./tests/tests_atome2.txt
44 #2) insertion frere apres
45 ./dico/dico dico/abr_atome.txt eh > ./tests/tests_atome3.txt
46 #3) insertion fils
47
48 ./dico/dico dico/abr_atome.txt eh > ./tests/tests_atome3.txt
49 #cas general
50
51 ./dico/dico dico/abr_verbe.txt chanta > ./tests/tests_insert1.txt;
52 #1bis) insertion avant les freres (sans fils)
53 ./dico/dico dico/abr_verbe.txt chantea > ./tests/tests_insert1bis.txt;
54 #2) insertion entre les freres
55 ./dico/dico dico/abr_verbe.txt chantier > ./tests/tests_insert2.txt;
56 #2bis) insertion entre les freres (sans fils)
57 ./dico/dico dico/abr_verbe.txt chanter > ./tests/tests_insert2bis.txt;

```

```

56 #3) insertion apres les freres
57 ./dico/dico dico/abr_verbe.txt chams > ./tests/tests_insert3.txt;
58 #3bis) insertion apres les freres (sans fils)
59 ./dico/dico dico/abr_verbe.txt chantez > ./tests/tests_insert3bis.txt;
60 #4) insertion en fils
61 ./dico/dico dico/abr_verbe.txt chanson > ./tests/tests_insert4.txt;
62 #5) insertion d'un mot inclu
63 ./dico/dico dico/abr_verbe.txt chant > ./tests/tests_insert5.txt;
64 #6) insertion d'un mot déjà écrit
65 ./dico/dico dico/abr_verbe.txt chante > ./tests/tests_insert6.txt;
66 #7) insertion sur une cellule sans frere (apres)
67 ./dico/dico dico/abr_verbe.txt charles > ./tests/tests_insert7.txt;
68 #8) insertion sur une cellule sans frere (avant)
69 ./dico/dico dico/abr_verbe.txt charles > ./tests/tests_insert8.txt;
70 #9) insertion avant racine
71 ./dico/dico dico/abr_verbe.txt bavarde > ./tests/tests_insert9.txt;
72 #10) insertion apres racine
73 ./dico/dico dico/abr_verbe.txt discute > ./tests/tests_insert10.txt;
74 #11) insertion fils racine
75 ./dico/dico dico/abr_verbe.txt calvitie > ./tests/tests_insert11.txt;
76
77 #liberation memoire
78 #1) dico sans argument
79 valgrind ./dico/dico > ./tests/tests_free0.txt
80 #2) dico
81 valgrind ./dico/dico dico/abr_verbe.txt zorro > ./tests/tests_free1.txt
82 #3) inversion
83 valgrind ./inversion/inversion dcba > ./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 chaîne 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 freres
  - Cas insertion avant les freres (sans fils)
  - Cas insertion entre les freres
  - Cas insertion entre les freres (sans fils)
  - Cas insertion apres les freres
  - Cas insertion apres les freres (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\_vides.txt

1 || ()

Test de cellule unique : abr\_atome.txt

1 || (E)

Test du cas général : abr\_verbe.txt

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

1.5.3 Execution

Inversion (Question 1)

- Cas de chaîne non valide : /inversion/inversion

1 || USAGE ./inversion/inversion <chaîne a inverser>

- Cas caractère unique : /inversion/inversion a

1 || Pile a inverser :  
2 || a  
3 ||  
4 || Etat de la file :  
5 || a  
6 ||  
7 || Pile inversee :  
8 || a

- Cas général : /inversion/inversion dcba

1 || Pile a inverser :  
2 || d  
3 || c  
4 || b  
5 || a  
6 ||  
7 || Etat de la file :  
8 || a  
9 || b  
10 || c  
11 || d  
12 ||  
13 || Pile inversee :  
14 || a  
15 || b  
16 || c  
17 || d

Validité des arguments

- Validité du fichier : /dico/dico\_inconnu.txt

1 || Fichier invalide !

- Validité de l'écriture parenthésée 1 : /dico/dico dico/abr\_err1.txt

```
1 ||
2 # Affichage de la notation parenthesee :
3 a(b(t))
4
5 || Ecriture parenthesee incorrecte !
```

- Validité de l'écriture parenthésée 2 : /dico/dico dico/abr\_err2.txt

```
1 ||
2 # Affichage de la notation parenthesee :
3 ((()((()R))
4
5 || Ecriture parenthesee incorrecte !
```

- Validité de l'écriture parenthésée 3 : /dico/dico dico/abr\_err3.txt

```
1 ||
2 # Affichage de la notation parenthesee :
3 (a(b(b(d))))
4
5 || Ecriture parenthesee incorrecte !
```

- Validité de l'écriture parenthésée 4 : /dico/dico dico/abr\_err4.txt

```
1 ||
2 # Affichage de la notation parenthesee :
3 (a(b,cd))
4
5 || Ecriture parenthesee incorrecte !
```

- Validité de l'écriture parenthésée 5 : /dico/dico dico/abr\_err5.txt

```
1 ||
2 # Affichage de la notation parenthesee :
3 (a(b,c,,d))
4
5 || Ecriture parenthesee incorrecte !
```

- Validité des chaînes à insérer : /dico/dico dico/abr\_vide.txt rien arbre? @pre arbre

```
1
2 # Affichage de la notation parenthesee :
3 ()
4
5 L'argument 'arbre?' n'est pas un mot valide !
6 L'argument '?@pre' n'est pas un mot valide !
7 L'argument 'arbre' 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(l)
16 free(r)
```

Lecture et Affichage (Question 3)

- Lecture -> Cas arbre vide : /dico/dico dico/abr\_vide.txt

```
1
2 # Affichage de la notation parenthesee :
3 ()
4
5
6 # Affichage des mots du dictionnaire :
7
8 # Liberation de l'arbre :
```

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

- Lecture -> Cas arbre atomique : /dico/dico dico/abr\_atome.txt

```
1
2 # Affichage de la notation parenthesee :
3 (E)
4
5
6 # Affichage des mots du dictionnaire :
7 E
8
9 # Liberation de l'arbre :
10 free(E)
```

- Lecture -> Cas général : /dico/dico dico/abr\_verbe.txt

```
1
2 # Affichage de la notation parenthesee :
3 (c(t(a(n(c(E)s,t(E(E,S)o(n(S)))R(m(E))T)))
4
5
6 # Affichage des mots du dictionnaire :
7 chance
8 chante
9 chanteE
10 chantes
11 chantons
12 char
13 charme
14 chat
15
16 # Liberation de l'arbre :
17 free(E)
18 free(S)
19 free(E)
20 free(S)
21 free(n)
22 free(o)
23 free(E)
24 free(t)
25 free(s)
26 free(c)
27 free(E)
```



```

2  # Affichage de la notation parenthésée :
3  (E)
4
5
6  # Affichage des mots du dictionnaire :
7  E
8  EH
9
10 # Libération de l'arbre :
11 free(H)
12 free(E)

```

1. Insertion -> Avant les freres : / dico/ dico dico/ abr\_ verbe.txt chanta

```

1  # Affichage de la notation parenthésée :
2  (c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))
3
4
5  # Affichage des mots du dictionnaire :
6  chance
7  chantE
8  chantA
9  chantE
10 chantE
11 chantE
12 chantons
13 char
14 charme
15 chat
16
17 # Libération de l'arbre :
18 free(E)
19 free(S)
20 free(E)
21 free(S)
22 free(h)
23 free(o)
24 free(E)
25 free(A)
26 free(t)
27 free(s)
28 free(c)
29 free(E)
30 free(m)
31 free(T)
32 free(R)
33 free(n)
34 free(a)
35 free(h)
36 free(c)

```

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

1 ||

```

2  # Affichage de la notation parenthésée :
3  (c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))
4
5
6  # Affichage des mots du dictionnaire :
7  chance
8  chantE
9  chantEA
10 chantEE
11 chantE
12 chantons
13 char
14 charme
15 chat
16
17 # Libération de l'arbre :
18 free(E)
19 free(S)
20 free(E)
21 free(A)
22 free(S)
23 free(n)
24 free(o)
25 free(E)
26 free(t)
27 free(s)
28 free(c)
29 free(E)
30 free(m)
31 free(T)
32 free(R)
33 free(n)
34 free(a)
35 free(h)
36 free(c)

```

3. Insertion -> Entre les freres : / dico/ dico dico/ abr\_ verbe.txt chanter

```

1  # Affichage de la notation parenthésée :
2  (c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E))T))))
3
4
5  # Affichage des mots du dictionnaire :
6  chance
7  chantE
8  chantEE
9  chantE
10 chantE
11 chantish
12 chantons
13 char
14 charme
15 chat
16
17 # Libération de l'arbre :
18 free(E)
19 free(S)

```

```

20 || free(E)
21 || free(R)
22 || free(a)
23 || free(S)
24 || free(n)
25 || free(o)
26 || free(i)
27 || free(E)
28 || free(c)
29 || free(s)
30 || free(c)
31 || free(E)
32 || free(m)
33 || free(T)
34 || free(R)
35 || free(n)
36 || free(a)
37 || free(h)
38 || free(c)

```

4. Insertion -> Entre les freres (sans fils) : /dico/dico dico/abr\_ verbe.txt chanter

```

1 ||
2 || # Affichage de la notation parenthesee :
3 || (c(h(a(n(c(E)s,t(E,E,S)o(n(S))))R(m(E))T))))
4 ||
5 ||
6 || # Affichage des mots du dictionnaire :
7 || chance
8 || chante
9 || chanteE
10 || chanter
11 || chanteS
12 || chantons
13 || char
14 || charME
15 || chat
16 ||
17 || # Liberation de l'arbre :
18 || free(E)
19 || free(S)
20 || free(R)
21 || free(E)
22 || free(S)
23 || free(n)
24 || free(o)
25 || free(E)
26 || free(t)
27 || free(s)
28 || free(c)
29 || free(m)
30 || free(T)
31 || free(R)
32 || free(n)
33 || free(a)
34 || free(h)
35 || free(l)

```

```

36 || free(c)

```

5. Insertion -> Apres les freres : /dico/dico dico/abr\_ verbe.txt chans

```

1 ||
2 || # Affichage de la notation parenthesee :
3 || (c(h(a(n(c(E)s,t(E,E,S)o(n(S))))R(m(E))T))))
4 ||
5 ||
6 || # Affichage des mots du dictionnaire :
7 || chance
8 || chans
9 || chante
10 || chanteE
11 || chanteS
12 || chantons
13 || char
14 || charME
15 || chat
16 ||
17 || # Liberation de l'arbre :
18 || free(E)
19 || free(S)
20 || free(E)
21 || free(S)
22 || free(n)
23 || free(o)
24 || free(E)
25 || free(t)
26 || free(S)
27 || free(c)
28 || free(E)
29 || free(m)
30 || free(T)
31 || free(R)
32 || free(n)
33 || free(a)
34 || free(h)
35 || free(c)

```

6. Insertion -> Apres les freres (sans fils) : /dico/dico dico/abr\_ verbe.txt chant

```

1 ||
2 || # Affichage de la notation parenthesee :
3 || (c(h(a(n(c(E)s,t(E,E,S)o(n(S))))R(m(E))T))))
4 ||
5 ||
6 || # Affichage des mots du dictionnaire :
7 || chance
8 || chante
9 || chanteE
10 || chanteS
11 || chanteZ
12 || chantons
13 || char

```

```
14 charme
15 chat
16
17 # Liberation de l'arbre :
18 free(E)
19 free(Z)
20 free(S)
21 free(E)
22 free(S)
23 free(n)
24 free(o)
25 free(E)
26 free(c)
27 free(s)
28 free(c)
29 free(E)
30 free(m)
31 free(T)
32 free(R)
33 free(n)
34 free(a)
35 free(h)
36 free(c)
```

7. Insertion -> fils : /dico/dico dico/abr\_ verbe.txt chansons

```
1
2 # Affichage de la notation parenthesee :
3 (c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E)T))))
4
5
6 # Affichage des mots du dictionnaire :
7 chance
8 chanson
9 chante
10 chanteE
11 chantes
12 chantons
13 char
14 charme
15 chat
16
17 # Liberation de l'arbre :
18 free(E)
19 free(N)
20 free(o)
21 free(S)
22 free(E)
23 free(S)
24 free(n)
25 free(o)
26 free(E)
27 free(t)
28 free(s)
29 free(c)
30 free(E)
31 free(m)
```

```
32 free(T)
33 free(R)
34 free(n)
35 free(a)
36 free(h)
37 free(c)
```

8. Insertion -> Mot inclu : /dico/dico dico/abr\_ verbe.txt chant

```
1
2 # Affichage de la notation parenthesee :
3 (c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E)T))))
4
5
6 # Affichage des mots du dictionnaire :
7 chance
8 chant
9 chante
10 chanteE
11 chantes
12 chantons
13 char
14 charme
15 chat
16
17 # Liberation de l'arbre :
18 free(E)
19 free(S)
20 free(E)
21 free(S)
22 free(n)
23 free(o)
24 free(E)
25 free(T)
26 free(s)
27 free(c)
28 free(E)
29 free(m)
30 free(T)
31 free(R)
32 free(n)
33 free(a)
34 free(h)
35 free(c)
```

9. Insertion -> Mot déjà présent : /dico/dico dico/abr\_ verbe.txt chansons

```
1
2 # Affichage de la notation parenthesee :
3 (c(h(a(n(c(E)s,t(E(E,S)o(n(S))))R(m(E)T))))
4
5
6 # Affichage des mots du dictionnaire :
7 chance
8 chante
```

9	chantEE
10	chantES
11	chantONS
12	char
13	charME
14	chat
15	
16	# Liberation de l'arbre :
17	free(E)
18	free(S)
19	free(E)
20	free(S)
21	free(n)
22	free(o)
23	free(E)
24	free(t)
25	free(s)
26	free(c)
27	free(E)
28	free(m)
29	free(T)
30	free(R)
31	free(n)
32	free(a)
33	free(h)
34	free(c)

10. Insertion -> Cellule sans frère (après) : /dico/dico dico/abr\_verbe.txt chartes

1	# Affichage de la notation parenthesee :
2	(c(h(a(n(c(E)s,t(E,E,S)o(n(S)))R(m(E)T))))
3	
4	
5	
6	# Affichage des mots du dictionnaire :
7	chance
8	chantE
9	chantEE
10	chantES
11	chantONS
12	char
13	charME
14	chartES
15	chat
16	
17	# Liberation de l'arbre :
18	free(E)
19	free(S)
20	free(E)
21	free(S)
22	free(n)
23	free(o)
24	free(E)
25	free(t)
26	free(s)
27	free(c)
28	free(E)

29	free(S)
30	free(e)
31	free(t)
32	free(m)
33	free(T)
34	free(R)
35	free(n)
36	free(a)
37	free(h)
38	free(c)

11. Insertion -> Cellule sans frère (avant) : /dico/dico dico/abr\_verbe.txt chartes

1	# Affichage de la notation parenthesee :
2	(c(h(a(n(c(E)s,t(E,E,S)o(n(S)))R(m(E)T))))
3	
4	
5	
6	# Affichage des mots du dictionnaire :
7	chance
8	chantE
9	chantEE
10	chantES
11	chantONS
12	char
13	charES
14	charME
15	chat
16	
17	# Liberation de l'arbre :
18	free(E)
19	free(S)
20	free(E)
21	free(S)
22	free(n)
23	free(o)
24	free(E)
25	free(t)
26	free(s)
27	free(c)
28	free(S)
29	free(e)
30	free(E)
31	free(m)
32	free(l)
33	free(T)
34	free(R)
35	free(n)
36	free(a)
37	free(h)
38	free(c)

12. Insertion -> Avant racine : /dico/dico dico/abr\_verbe.txt bavarde

1	
---	--



```

2 | # Affichage de la notation parenthésée :
3 | (c(h(a(n(c(E),t(E(E,S)o(n(S))))R(m(E))T))))
4 |
5 |
6 | # Affichage des mots du dictionnaire :
7 | bavardE
8 | chance
9 | chantE
10 | chantEE
11 | chantes
12 | chantons
13 | char
14 | charme
15 | chat
16 |
17 | # Libération de l'arbre :
18 | free(E)
19 | free(d)
20 | free(r)
21 | free(a)
22 | free(v)
23 | free(a)
24 | free(E)
25 | free(S)
26 | free(E)
27 | free(S)
28 | free(n)
29 | free(o)
30 | free(E)
31 | free(c)
32 | free(s)
33 | free(c)
34 | free(E)
35 | free(m)
36 | free(T)
37 | free(R)
38 | free(n)
39 | free(a)
40 | free(h)
41 | free(c)
42 | free(b)

```

13. Insertion -> Après racine : /dico/dico dico/abr\_verbe.txt discute

```

1 |
2 | # Affichage de la notation parenthésée :
3 | (c(h(a(n(c(E),t(E(E,S)o(n(S))))R(m(E))T))))
4 |
5 |
6 | # Affichage des mots du dictionnaire :
7 | chance
8 | chantE
9 | chantEE
10 | chantes
11 | chantons
12 | char
13 | charme

```

```

14 | chat
15 | discute
16 |
17 | # Libération de l'arbre :
18 | free(E)
19 | free(S)
20 | free(E)
21 | free(n)
22 | free(m)
23 | free(o)
24 | free(E)
25 | free(t)
26 | free(s)
27 | free(c)
28 | free(E)
29 | free(m)
30 | free(T)
31 | free(R)
32 | free(n)
33 | free(a)
34 | free(h)
35 | free(E)
36 | free(t)
37 | free(u)
38 | free(c)
39 | free(s)
40 | free(l)
41 | free(d)
42 | free(c)

```

14. Insertion -> Fils racine : /dico/dico dico/abr\_verbe.txt calvitie

```

1 |
2 | # Affichage de la notation parenthésée :
3 | (c(h(a(n(c(E),t(E(E,S)o(n(S))))R(m(E))T))))
4 |
5 |
6 | # Affichage des mots du dictionnaire :
7 | calvitie
8 | chance
9 | chantE
10 | chantEE
11 | chantes
12 | chantons
13 | char
14 | charme
15 | chat
16 |
17 | # Libération de l'arbre :
18 | free(E)
19 | free(i)
20 | free(t)
21 | free(i)
22 | free(v)
23 | free(l)
24 | free(E)
25 | free(S)

```

```

26 | free (E)
27 | free (S)
28 | free (n)
29 | free (o)
30 | free (E)
31 | free (t)
32 | free (s)
33 | free (c)
34 | free (E)
35 | free (m)
36 | free (T)
37 | free (R)
38 | free (n)
39 | free (a)
40 | free (h)
41 | free (a)
42 | free (c)

```

# Liberation mémoire

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

```

1 | ==30969== Memcheck, a memory error detector
2 | ==30969== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
3 | ==30969== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
4 | ==30969== Command: ./dico/dico
5 | ==30969==
6 | ==30969==
7 | ==30969== HEAP SUMMARY:
8 | ==30969==   in use at exit: 0 bytes in 0 blocks
9 | ==30969==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
10 | ==30969==
11 | ==30969== All heap blocks were freed -- no leaks are possible
12 | ==30969==
13 | ==30969== For counts of detected and suppressed errors, rerun with: -v
14 | ==30969== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)

```

– Valgrind -> DICO : valgrind /dico/dico/abr\_verbe.txt zorro

```

1 | ==31301== Memcheck, a memory error detector
2 | ==31301== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
3 | ==31301== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
4 | ==31301== Command: ./dico/dico dico/abr_verbe.txt zorro
5 | ==31301==
6 | ==31301== Invalid read of size 8
7 | ==31301== at 0x400F94: liberation_abr (arbre.c:137)
8 | ==31301== by 0x400A32: main (main.c:81)
9 | ==31301== Address 0x4c2b7d0 is 16 bytes inside a block of size 24 free'd
10 | ==31301== at 0x4A0595D: free (vg_replace_malloc.c:366)
11 | ==31301== by 0x401049: liberation_abr (arbre.c:157)
12 | ==31301==
13 | ==31301==
14 | ==31301==
15 | ==31301== HEAP SUMMARY:
16 | ==31301==   in use at exit: 0 bytes in 0 blocks
17 | ==31301==   total heap usage: 32 allocs, 32 frees, 2,476 bytes allocated
18 | ==31301==
19 | ==31301== All heap blocks were freed -- no leaks are possible
20 | ==31301==
21 | ==31301== For counts of detected and suppressed errors, rerun with: -v
22 | ==31301== ERROR SUMMARY: 23 errors from 1 contexts (suppressed: 6 from 6)

```

– Valgrind -> INVERSION : valgrind /inversion/inversion.doba

```

1 | ==31682== Memcheck, a memory error detector
2 | ==31682== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
3 | ==31682== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
4 | ==31682== Command: ./inversion/inversion.doba
5 | ==31682==
6 | ==31682==
7 | ==31682== HEAP SUMMARY:
8 | ==31682==   in use at exit: 0 bytes in 0 blocks
9 | ==31682==   total heap usage: 4 allocs, 4 frees, 240 bytes allocated

```

```
10 ==31682==
11 ==31682== All heap blocks were freed -- no leaks are possible
12 ==31682==
13 ==31682== For counts of detected and suppressed errors, rerun with: -v
14 ==31682== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```