ISIMA Première Année

Compte-Rendu de TP

Structures de Données

Gestion d'un dictionnaire arborescent

Benjamin BARBESANGE
Pierre-Loup PISSAVY
Groupe G21

Enseignant :
Michelle CHABROL

mars 2015

ISIMA

Table des matières

1 | Présentation

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

Les opérations suivantes sont permises avec l'arbre :

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

1.1 Structure de données employée

Les mots du dictionnaire sont rangés dans un arbre, par orbre alphabétiue. La fin d'un mot est signalée par une lettre majuscule.

1.2 Organisation du code source

Nous avons découpé le tp en 3 parties. Une partie s'occupe de la gestion de pile, une autre de la gestion de liste chaînée (utilisées dans la définition de l'arbre) et la dernière est chargée de gérer la structure d'arbre que nous avons crée.

1.2.1 Gestion de la pile

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

1.2.2 Gestion des listes chaînées

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

1.2.3 Gestion de l'arbre

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

1.2.4 Programme principal

- `src/main.c`

2 | Détails du programme

2.1 Gestion de la pile

Code C

```c
/*  stack.h
  Header

  -----| DERECURSIVATION DE FONCTION PAR PILE |-----

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#ifndef __STACK__H
#define __STACK__H

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

  typedef tree_t* datatype;

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

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

#endif
```

Code C

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

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

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

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

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

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

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

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

  Entree :
    *p : pointeur sur la tete de la pilevoid afficherArbre(tree_t *t){
  if (t != NULL) {
    printf("%c ",t->letter);
    afficherArbre(t->lv);
    afficherArbre(t->lh);
  }

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

```

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

  Entree :
    p : tete de la pile

  Sortie :
    int : booleen
      0 si la pile n'est pas vide
      1 si la pile est vide
*/
int empty(stack_t p) {
  return (p.top == -1)?1:0;
}

/*  int full(stack_t p)
Teste si la pile est pleine ou non

  Entree :
    p : tete de la pile

  Sortie :
    int : booleen
      0 si la pile n'est pas pleine
      1 si la pile est pleine
*/
int full(stack_t p) {
  return (p.top == p.max-1)?1:0;
}

/*  int pop(stack_t *p, datatype *v)
Recupere le premier element de la pile (et l'enleve) et retourne un code d'erreur

  Entree :
    *p : pointeur sur la tete de la pile
    *v : pointeur sur un element du type de la pile, variable en I/O

  Sortie :
    int : code d'erreur
      0 si rien n'est retourne dans la variable v
      1 si on a recupere l'element en tete
*/
int pop(stack_t *p, datatype *v) {
  int ok = 0;
  if (!empty(*p)) {
    *v = p->val[p->top];
    ok = 1;
    p->top--;
  }
  return ok;
}

/*  int top(stack_t *p, datatype *v)
Retourne l'element en tete de la pile (sans l'enlever) et retourne un code d'erreur

  Entree :
```

```
    *p : pointeur sur la tete de la pile
    *v : pointeur sur un element du type de la pile, variable en I/O

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

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

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

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

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

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

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

```
168 }
```

2.2 Gestion des listes chaînées

Code C

```c
/*  list.h
  Header

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

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#ifndef __LISTE_H__
#define __LISTE_H__

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

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

  typedef node_t cell_t;

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

#endif
```

Code C

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

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

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

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

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

```
19
20    Entrees :
21      cell_t **prec : pointeur sur le pointeur de l'element apres lequel ajouter
22      cell_t *elt : pointeur sur l'element a ajouter a la liste chainee
23
24    Sortie :
25      Aucune
26  */
27  void adj_cell(cell_t **prec, cell_t *elt) {
28    elt->lh = (*prec);
29    (*prec) = elt;
30  }
31
32  /*  cell_t ** rech_prec(cell_t **liste, char letter, short int *existe)
33    Recherche le precedent d'un element dans la liste chainee a partir de la
34    date de debut de message
35
36    Entrees :
37      cell_t **liste : pointeur sur le pointeur du premier element de la liste chainee
38      char lettre : caractere a chercher dans la liste
39      short int *existe : variable en entre/sortie indiquant si on a ou pas de message ayant
    ↪ la date de debut
40        0 : il n'y a pas de message avec cette date de debut
41        1 : il y a au moins un message
42
43    Sortie :
44      cell_t ** : pointeur sur le pointeur de l'element precedent
45  */
46  cell_t ** rech_prec(cell_t **liste, char letter, short int *existe) {
47    cell_t **prec = liste;
48    while ((*prec) && tolower((*prec)->letter) < tolower(letter)) {
49      prec = &((*prec)->lh);
50    }
51    /* Booleen de presence     */
52    /* 1 : present             */
53    /* 0 : absent              */
54    *existe = (*prec && tolower((*prec)->letter) == tolower(letter))?1:0;
55    return prec;
56  }
57
58  /*  void supp_cell(cell_t **prec)
59    Permet de supprimer un element dans la liste chainee a partir
60    de son precedent
61
62    Entrees :
63      cell_t **prec : pointeur sur le pointeur de l'element precedent l'element a supprimer
64
65    Sortie :
66      Aucune
67  */
68  void supp_cell(cell_t **prec) {
69    cell_t *elt = *prec;
70    *prec = elt->lh;
71    free(elt);
72  }
73
```

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

  Entrees :
    cell_t **liste : pointeur sur le pointeur du premier element de la liste chainee

  Sortie :
    Aucune
*/
void liberer_liste(cell_t **liste) {
  while (*liste) {
    supp_cell(liste);
  }
  *liste = NULL;
}

/*  void ins_cell(cell_t **liste, cell_t *elt)
  Permet d'inserer une cellule a la bonne place dans la liste chainee
  Les messages sont tries par ordre decroissant des date de debut

  Entrees :
    cell_t **liste : pointeur sur le pointeur du premier element de la liste chainee
    cell_t *elt : pointeur sur l'element a inserer dans la liste chainee

  Sortie :
    Aucune
*/
void ins_cell(cell_t **liste, cell_t *elt) {
  short int existe;
  cell_t **prec = rech_prec(liste,elt->letter,&existe);
  adj_cell(prec,elt);
}

/*  node_t * creer_cell(char letter)
  Permet de creer un element de la liste chainee a partir du
  caractere donne en parametre

  Entrees :
    char letter : lettre a mettre dans l'element

  Sortie :
    node_t* : pointeur sur l'element cree
*/
node_t * creer_cell(char letter) {
  node_t *elt = (node_t*) malloc(sizeof(node_t));
  if (elt) {
    elt->letter = letter;
  }
  return elt;
}
```

2.3 Gestion de l'arbre

Code C

```c
/*  tree.h
  Header

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

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#ifndef __TREE__H
#define __TREE__H

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

  #define DEBUG 0

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

#endif
```

Code C

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

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

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#include "tree.h"

tree_t *creerNoeud(char);

/*  int creerArbre(char *ch, tree_t **r)
Cree un arbre a partir d'un chaine caractre representant la notation parenthesee
et en prenant l'adresse du pointeur sur la tete de l'arbre
```

```
19
20    Entrees :
21      char *ch : chaine de representation de l'arbre
22      tree_t **r : pointeur double de tête de l'arbre
23
24    Sortie :
25      int : code de retour sur la creation
26            0 : probleme d'alloc d'element
27            1 : aucun probleme
28  */
29  int creerArbre(char *ch, tree_t **r) {
30    stack_t p;                    /* Pile */
31    tree_t **prec = r;            /* Pointeur de parcours de l'arbre */
32    tree_t *tmp;                  /* Pointeur temporaire */
33    char *cour = ch;              /* Caractere courant */
34    int taille = strlen(ch)/2;    /* Taille max de la pile */
35    int ret = 0;                  /* Variable de retour */
36
37    cour++;                       /* On consomme la premiere parenthese */
38    if (init(&p,taille)) {
39      ret = 1;                    /* Allocation ok */
40      while (ret && (!empty(p) || *cour != ')')) {   /* Aucun souci et chaine non-finie */
41        if (*cour == '(') {
42          push(&p,*prec);           /* Sauvegarde de l'adresse courant */
43          prec = &((*prec)->lv);    /* Deplacement sur le lien vertical */
44          cour++;                   /* Acceleration, passe au prochain caractere */
45        } else if (*cour == ',') {
46          prec = &((*prec)->lh);    /* Deplacement sur le lien horizontal */
47          cour++;
48        }
49        *prec = creerNoeud(*cour);
50        if (! (*prec)) {
51          ret = 0;                  /* Problème allocation */
52        } else {
53          cour++;                   /* Passage caractere suivant */
54        }
55        while (ret && !empty(p) && *cour == ')') {
56          pop(&p,&tmp);             /* Recuperation du lien horizontal parent */
57          prec = &(tmp->lh);
58          cour++;
59        }
60      }
61      supp(&p);                     /* Liberation pile */
62    }
63    return ret;
64  }
65
66  /*  tree_t *creerNoeud(char v)
67  Cree un noeud ayant pour valeur le caractere entre
68
69    Entrees :
70      char : valeur du nouveau noeu cree
71
72    Sortie :
73      tree_t* : pointeur sur le nouvel element cree
74  */
```

```c
tree_t *creerNoeud(char v) {
  tree_t *r = (tree_t*) malloc (sizeof(tree_t));
  if (r) {
    r->lv = NULL;
    r->lh = NULL;
    r->letter = v;
  }
  return r;
}

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

  Entrees :
    tree_t* : pointeur sur la tete de l'arbre
    char * : prefixe a ecrire avant chaque mot de l'arbre

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

  if (cour != NULL && init(&p,100)) {
    do {
      while (cour != NULL) {
        push(&p,cour);              /* Sauvegarde du point courant */
        if (isupper(cour->letter)) { /* Detection fin de mot */
          printf("%s", prefixe);    /* Affiche le prefixe */
          dump(p,afficherPoint);    /* Affichage du mot (lecture pile) */
          printf("\n");
        }
        cour = cour->lv;            /* Deplacement sur le lien vertical */
      }
      /* On a atteint une feuille */
      while (!empty(p) && cour == NULL) {  /* Recherche du premier frere des ascendants */
        pop(&p,&cour);              /* Recuperation du parent */
        cour = cour->lh;            /* Deplacement sur le lien horizontal */
      }
    } while (!empty(p) || cour != NULL);
    supp(&p);
  }
}

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

  Entrees :
    tree_t* : pointeur sur la tete de l'arbre

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

```c
}

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

  Entrees :
    tree_t* : pointeur sur le noeud

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

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

  Entrees :
    tree_t** : adresse du pointeur sur la tete de l'arbre

  Sortie :
    Aucune
*/
void libererArbre(tree_t **t) {
  stack_t p;                    /* Pile */
  tree_t *cour = *t;            /* Pointeur de parcours de l'arbre */
  tree_t *tmp;

  if (cour != NULL && init(&p,100)) {
    do {
      while (cour != NULL) {
        tmp = cour;                 /* Sauvegarde du courant */
        if (cour->lh != NULL) {
          push(&p,cour->lh);        /* Sauvegarde du frere */
        }
        cour = cour->lv;            /* Deplacement sur le lien vertical */
        free(tmp);                  /* Suppression du point courant */
      }
      if(!empty(p)) {
        pop(&p,&cour);              /* Recuperation du premier lien horizontal parmi les parents */
      }
    } while (!empty(p) || cour != NULL);
    supp(&p);
  }
  *t = NULL;
}

/*  void adj_fils(tree_t **prec, tree_t *elt)
Ajoute un element dans l'arbre, a partir de l'adresse du pointeur sur le prec

  Entrees :
    **prec : adresse du pointeur sur l'element avant lequel inserer
    *elt : pointeur sur l'element a ajouter

```

```
  Sortie :
    Aucune
*/

void adj_fils(tree_t **prec, tree_t *elt) {
  elt->lv = (*prec);
  (*prec) = elt;
}

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


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

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

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

  return arbre;
}

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

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

  Sortie :
    int : code d'erreur
          0 : probleme d'allocation ou d'insertion
```

```c
          1 : aucun soucis d'insertion
*/
int insererMot(tree_t **t, char *w) {
  int len;              /* Longueur du mot */
  int i;                /* Indice de parcours pour copie */
  int res = 1;          /* Code de retour */
  char *cour;           /* Copie du mot */
  tree_t *tmp;          /* Noeud temporaire de creation */
  tree_t **arbre = t;   /* Pointeur de parcours de l'arbre */

  if (*w != '\0') { /* Mot non vide */
    /* Traitement du mot */
    len = strlen(w);    /* Calcul longueur */
    cour = (char*) malloc ((len+1)*sizeof(char));
    if (cour) {         /* Allocation ok */
      i = 0;
      while (w[i+1] != '\0') {
        cour[i] = tolower(w[i]); /* Passage en minuscules */
        ++i;
      }
      cour[i] = toupper(w[i]); /* Derniere lettre majuscule */
      cour[++i] = '\0';

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

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

            /* Insertion des lettres restantes selon des liens verticaux */
            while (res && *cour != '\0') {
              tmp = creerNoeud(*cour);
              if (tmp) {                 /* Noeud cree */
                adj_fils(arbre,tmp);     /* Insertion lien vertical */
                arbre = &((*arbre)->lv); /* Pointeur sur noeud fils */
                cour++;                  /* Lettre suivante */
              } else {                   /* Noeud non cree */
                res = 0;
              }
            }
          } else {                   /* Noeud non cree */
            res = 0;
          }
        }
```

```
298       }
299       free(cour-len); /* Liberation a partir du pointeur sur le debut du mot */
300     } else { /* Allocation ratee */
301       res = 0;
302     }
303   }
304   return res;
305 }
306 
307 /*  void rech_motif(tree_t **t, char *w)
308 Affiche tous les mots commencant par un certain motif dans l'arbre
309 
310   Entrees :
311     **t : adresse du pointeur sur l'arbre
312     *w : chaine de caractere representant le motif des mots a afficherArbre
313 
314   Sortie :
315     Aucune
316 */
317 void rech_motif(tree_t **t, char *w) {
318         tree_t **arbre = t;
319         char *cour = w;
320 
321         arbre = rech_mot(t, &cour); /* recherche jusqu'a la fin du motif */
322     if (*cour == '\0') { /* On a trouve tout le motif */
323       afficherArbrePref(*arbre, w);
324         }
325 }
```

2.4 Programme principal

Code C

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

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

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

void test();

#include "tree.h"
int main(int argc, char *argv[]) {
  FILE *f;
  char buf[400];
  char text[400];
  tree_t *arbre = NULL;
  if (argc > 1) {
    /* Lecture de fichier de commandes */
    f = fopen(argv[1],"r");
    if (f) {
      while (!feof(f)) {
        buf[0] = '\0';
        text[0] = '\0';
        fgets(buf,400,f);
        switch (buf[0]) {
          case 'C': /* creer */
            sscanf(&buf[1],"%s",text);
            creerArbre(text,&arbre);
            break;
          case 'I': /* inserer */
            sscanf(&buf[1],"%s",text);
            insererMot(&arbre,text);
            break;
          case 'M': /* motif */
            sscanf(&buf[1],"%s",text);
            rech_motif(&arbre,text);
            break;
          case 'L': /* liberer */
            if (arbre) {
              libererArbre(&arbre);
            } else {
              printf("Liberation impossible\n");
            }
            break;
          case 'A': /* afficher */
            afficherArbre(arbre);
            break;
          case '#': /* texte */
            printf("%s",&buf[1]);
            break;
```

```
54            default:
55              puts("\n");
56          }
57        }
58        if (arbre) {
59          libererArbre(&arbre);
60        }
61        fclose(f);
62      } else {
63        fprintf(stderr, "Fichier invalide\n");
64      }
65    } else {
66      /* Fonctions de tests de base */
67      test();
68    }
69    return 0;
70  }
71
72  void test() {
73    tree_t *monArbre = NULL;
74    int i, nbMotifs = 4;
75    char *motif[] = {"a","","az","x"};
76    printf("##########\nDEBUT DU PROGRAMME DE TEST\n##########\n");
77
78    if (creerArbre("(a(l(p(h(A))))b(r(a(v(E,O))))c(h(a(r(l(i(E))))))d(e(l(t(A(p(l(a(n(E)))))))))e(c(h(O))p(i(n
    {
79      printf("**********\nAffichage avant insertion\n");
80      afficherArbre(monArbre);
81
82      insererMot(&monArbre,"ALPHABET");   /* Debut deja present */
83      insererMot(&monArbre,"foxtrot");   /* Aucune lettre deja presente */
84      insererMot(&monArbre,"echo");      /* Mot deja present */
85      insererMot(&monArbre,"epi");     /* Mot inclus dans un mot deja present */
86      insererMot(&monArbre,"");         /* Mot vide */
87
88      printf("**********\nAffichage apres insertion\n");
89      afficherArbre(monArbre);
90
91            for (i = 0; i < nbMotifs; ++i) {
92        printf("**********\nRecherche du motif \"%s\"\n", motif[i]);
93        rech_motif(&monArbre, motif[i]);
94            }
95
96      printf("**********\nInsertion dans l'arbre vide\n");
97      libererArbre(&monArbre);
98      insererMot(&monArbre,"alpha");
99
100     printf("**********\nAffichage apres insertion\n");
101     afficherArbre(monArbre);
102
103     printf("**********\nLiberation de l'arbre\n");
104     libererArbre(&monArbre);
105
106     printf("**********\nAffichage apres liberation\n");
107     afficherArbre(monArbre);
```

```
108
109     printf("##########\nFIN DU PROGRAMME DE TEST\n##########\n");
110   } else {
111     fprintf(stderr, "Probleme creation arbre\n");
112   }
113 }
```

3 | Principes et lexiques des fonctions

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

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

3.1 Gestion de la pile

La gestion de la pile s'effectue grâce aux fichiers `stack.c` et `stack.h`. Les algorithmes de principe des différentes fonctions ont été précédement détaillés dans le `tp 2`, nous ne les détaillerons donc pas.

3.2 Gestion des listes chaînées

Les fonctions de gestion des listes chaînées peuvent être trouvées dans les fichiers `list.c` et `list.h`. Les algorithmes de principe de ces fonctions ont également été fourni dans le `tp 1`. Ils ne seront donc pas inclus ici.

3.3 Gestion de l'arbre

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

3.3.1 creerArbre

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

Lexique :

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

3.3.2 creerNoeud

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

Lexique :

- *r : *nouvel élément créé*

3.3.3 afficherArbrePref

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

Lexique :

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

3.3.4 afficherArbre

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

3.3.5 afficherPoint

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

3.3.6 libererArbre

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

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

3.3.7 adj_fils

Cette fonction permet simplement d'ajouter un fils à un élément. Le pointeur sur lien vertical de l'élément prend l'adresse du précédent et le pointeur prec prend l'adresse de l'élément que l'on ajoute.

3.3.8 rech_mot

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

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

3.3.9 insererMot

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

Lexique :

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

3.3.10 rech_motif

```
Algorithme rech_motif (Principe)
Début
    Initialisation d'un pointeur arbre, sur la tête;
    Initialisation d'un pointeur cour, sur le caractère courant du mot;
    Lance rech_mot() et stocke les résultats dans arbre et cour;
    Si cour = '\0' Alors
        Affichage de l'arbre avec pour préfixe le motif entré;
    FinSi;
Fin
```

Lexique :

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

4 | Compte rendu d'exécution

4.1 Makefile

Makefile

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

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

EXEC=prog

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

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

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

4.2 Jeux de tests