ISIMA Première Année

Compte-Rendu de TP

Structures de Données

# Dérécursivation à l'aide d'une pile

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 dérécursiver une fonction à l'aide d'une pile. Les fonctions de gestion d'une pile seront ainsi créées.

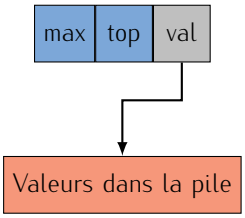
Les opérations suivantes sont permises avec la pile :

- Initialiser la pile,
- Libérer la pile,
- Tester si la pile est vide,
- Tester si la pile est pleine,
- Retourner l'élément en haut de la pile,
- Afficher l'élément en haut de la pile,
- Insérer un élément dans la pile.

## 1.1 Structure de données employée

Les données seront stockées dans une liste contiguë à accès indirect en tête. La taille et le pointeur sur cette liste ainsi que l'indice de l'élément en tête de pile sont stockés dans la structure représentée en figure 1.1.

(a) Structure utilisée




(b) Code

Code C

```c
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;
```

FIGURE 1.1 – Structure de pile et code correspondant

## 1.2 Organisation du code source

Nous avons défini deux modules, le premier contient une fonction sous forme récursive ansi que sa version sous forme itérative. Nous disposons également d'un module permettant de gérer une pile, qui est ainsi utilisé lors de la dérécursivation de la fonction.

### 1.2.1 Gestion de la pile

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

### 1.2.2 Fonction récursive

- `src/truc.h`
- `src/truc.c`

### 1.2.3 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>

  typedef int datatype; /* permet d'utiliser des types differents avec la pile */

  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);

#endif
```

Code C

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

  -----| DERECURSIVATION DE FONCTION PAR 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 = (int*) malloc(n*sizeof(int));
  if (p->val == NULL) {
    ret = 0;
  }
  return ret;
}

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

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

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

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

  Entree :
    p : tete de la pile
```

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

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

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

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

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

## 2.2 Fonction récursive

Code C

```c
/*  truc.h
  Header

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

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/
#ifndef __TRUC__H
#define __TRUC__H
        #include <stdio.h>
        #include <stdlib.h>
        #define N 10
        int TRUC(int, int);
        int truc_iter(int, int);
#endif
```

Code C

```c
/*  truc.c
  Fonction recursive et son equivalent en iteratif

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

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#include "truc.h"
#include "stack.h"

/* tableau des valeurs pour la decomposition, premiere valeur inutile */
int P[N+1] = {0,1,3,2,4,5,2,7,1,9,1};

/*  int TRUC(int S, int I)
  Fonction sous forme recursive qui affiche la decomposition de
  S a partir d'un tableau d'entiers (defini ici en statique)

  Entrees :
    int S : Nombre a decomposer
    int I : Point de depart dans le tableau pour decomposer S

  Sortie :
    int : entier sous forme de booleen
      0 si on a pas pu decomposer S exactement
      1 sinon
*/
int TRUC(int S, int I) {
  if (S == 0) {
    return 1;
```

```
  } else if (S < 0 || I > N) {
    return 0;
  } else if (TRUC(S-P[I],I+1)) {
    printf("%d\n",P[I]);
    return 1;
  } else {
    return TRUC(S,I+1);
  }
}

/*  int truc_iter(int s, int i)
  Meme fonction qu'au dessus, mais sous forme iterative

  Entrees :
    int s : Nombre a decomposer
    int i : Point de depart dans P pour decomposer S

  Sortie :
    int : entier sous forme de booleen
      0 si on a pas pu decomposer S exactement
      1 sinon
*/
int truc_iter(int s, int i) {
  int sl = s;
  int il = i;
  int r = 0;
  stack_t p;
  if (init(&p,N)) {
    do {
      while (sl > 0 && il <= N) {
        push(&p,il);
        sl -= P[il];
        ++il;
      }
      if (sl == 0) {
        r = 1;
        while (!empty(p)) {
          pop(&p,&il);
          sl += P[il];
          printf("%d\n",P[il]);
        }
      } else {
        r = 0;
        if (!empty(p)) {
          pop(&p,&il);
          sl += P[il];
          ++il;
        }
      }
    } while (!empty(p));
    supp(&p);
  }
  return r;
}
```

## 2.3 Programme principal

Code C

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

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

  BARBESANGE Benjamin,
  PISSAVY Pierre-Loup

  ISIMA 1ere Annee, 2014-2015
*/

#include "truc.h"
#include "stack.h"

extern int P[];

void pile_test(int);
void afficher_P();

int main(int argc, char *argv[]) {
  FILE *f;
  char buf[100];
  int i, s, n;
  if (argc > 1) {
    /* Lecture de fichier de commandes */
    f = fopen(argv[1],"r");
    if (f) {
      while (!feof(f)) {
        buf[0] = '\0';
        fgets(buf,100,f);
        switch (buf[0]) {
          case 'p': /* pile */
            sscanf(&buf[1],"%d",&n);
            pile_test(n);
            break;
          case 't': /* truc iteratif */
            afficher_P();
            sscanf(&buf[1],"%d %d", &s, &i);
            printf("Reponse: %s\n",(truc_iter(s,i))?"Vrai":"Faux");
            break;
          case 'T': /* truc recursif */
            afficher_P();
            sscanf(&buf[1],"%d %d", &s, &i);
            printf("Reponse: %s\n",(TRUC(s,i))?"Vrai":"Faux");
            break;
          case '#': /* texte */
            printf("%s",&buf[1]);
            break;
          default:
            puts("\n");
        }
      }
    } else {
```

```
      fprintf(stderr, "Fichier invalide\n");
    }
  } else {
    /* Fonctions de tests de base */
    printf("Test de pile:\n");
    pile_test(4);
    printf("Fonction recursive:\n");
    TRUC(9,1);
    printf("Fonction iterative:\n");
    truc_iter(9,1);
  }
  return 0;
}

void pile_test(int n) {
  stack_t p;
  int i = 0;
  if (init(&p,n)) {
    while (push(&p,i)) {
      printf("Empiler: %d\n",i);
      ++i;
    }
    while (pop(&p,&i)) {
      printf("Depiler: %d\n",i);
    }
    supp(&p);
  }
}

void afficher_P() {
  int i;
  printf("Tableau P : ");
  for (i = 1; i <= N; ++i) {
    printf("%d ",P[i]);
  }
  printf("\n");
}
```

# 3 | Principes et lexiques des fonctions

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

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

## 3.1 Gestion de la pile

La gestion de la pile s'effectue grâce aux fichiers `stack.c` et `stack.h`.

### 3.1.1 init

Algorithme init
Début
    On initialise le code de retour à 1;  *[ Il n'y a aucune erreur ]*
    On initialise la taille max de la pile;
    On initialise l'indice du haut de la pile à -1;  *[ Pour indiquer qu'elle est vide ]*
    On alloue l'espace de la pile;
    Si l'allocation n'est pas réussie Alors
        Le code de retour passe à 0;
    FinSi;
    Retourner code de retour;
Fin

Lexique :
    ret : *code de retour, 0 si il y a une erreur de création de la pile, 1 sinon*

### 3.1.2 supp

Ici, nous libérons simplement le tableau de valeurs de la pile, puisque celui-ci est alloué dynamiquement lors de la création. Nous avons choisi de modifier les champs (indice de tête de pile à −1 et taille de la pile à 0) afin d'empêcher tout traitement sur cette pile jusqu'à sa réinitialisation. En effet, les fonctions permettant d'empiler et de dépiler renverront un code d'erreur.

### 3.1.3 empty

Cette fonction teste simplement si l'indice du haut de la pile est −1, ce qui signifie qu'il n'y a aucun élément dans la pile. Ainsi la valeur 1 sera retournée. Sinon la valeur 0 est retournée.

### 3.1.4 full

Cette fonction vérifie si l'indice de l'élément en haut de la pile est égal à la taille max de la pile (moins 1, car les tableaux commencent à 0). Si c'est le cas, on renvoie 1 pour signaler que la pile est pleine, et 0 sinon.

### 3.1.5 pop

```
Algorithme pop
Début
    On initialise le code de retour à 0;   [ On n'a pas dépilé ]
    Si la pile n'est pas vide Alors
        On dépile l'élément dans une variable en Input/Output;
        Le code de retour passe à 1;   [ On a dépilé et récupéré l'élément ]
        On modifie l'indice de l'élément en haut de la pile;   [ On retranche 1 à l'indice précédent ]
    FinSi;
    Retourner code de retour;
Fin
```

Lexique :

ok : *code de retour, 0 si on n'a pas dépilé, 1 si on a dépilé la valeur en haut de la pile*

### 3.1.6 top

```
Algorithme top
Début
    On initialise le code de retour à 0;   [ Pas d'élément lu ]
    Si la pile n'est pas vide Alors
        On copie la tête de pile dans une variable en I/O;
        Le code de retour passe à 1;   [ On a récupéré la valeur ]
    FinSi;
    Retourner code de retour;
Fin
```

Lexique :

ok : *code de retour, 0 si on n'a pas récupéré l'élément en haut de la pile, 1 si on l'a récupéré*

### 3.1.7 push

Algorithme push
Début
    On initialise le code de retour à 0;  *[ L'élément n'est pas ajouté dans la pile ]*
    Si la pile n'est pas pleine Alors
        On incrémente l'indice de l'élément en haut de la pile;
        On place l'élément dans le tableau de la pile, à l'indice précédement modifié;
        Le code de retour passe à 1;  *[ L'élément est empilé ]*
    FinSi;
    Retourner code de retour;
Fin

Lexique :

ok : *code de retour, 0 si on n'a pas empilé, 1 si on a empilé la valeur*

## 3.2 Dérécursivation de la fonction

La fonction récursive ainsi que sa version itérative se trouvent dans les fichiers `truc.c` et `truc.h`.

### 3.2.1 TRUC

Cette fonction étant l'énoncé du TP, nous ne détaillerons ainsi ni le principe ni les variables utilisées dans cet algorithme.

Nous allons dérécursiver cette fonction. Il s'agit d'un traitement simplifié du problème du sac à dos.

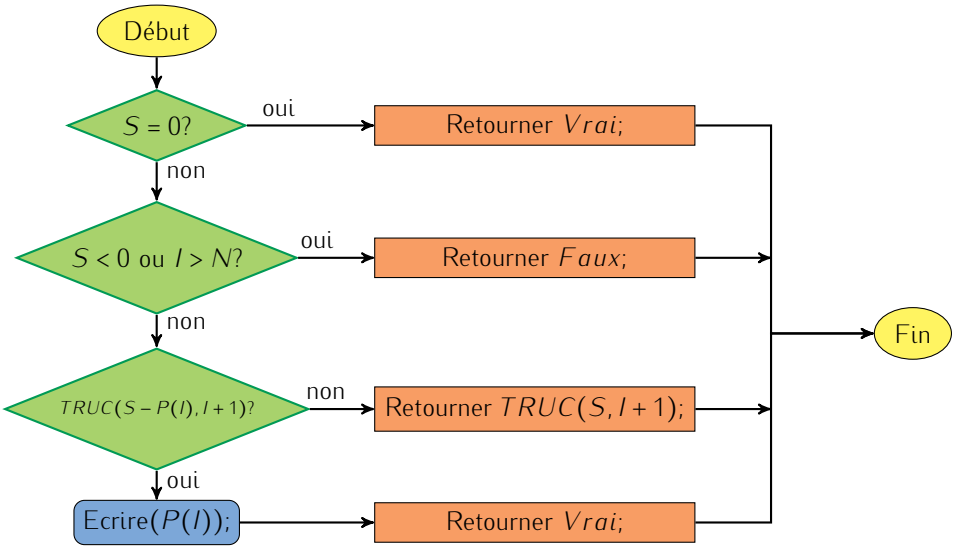



FIGURE 3.1 – Logigramme initial

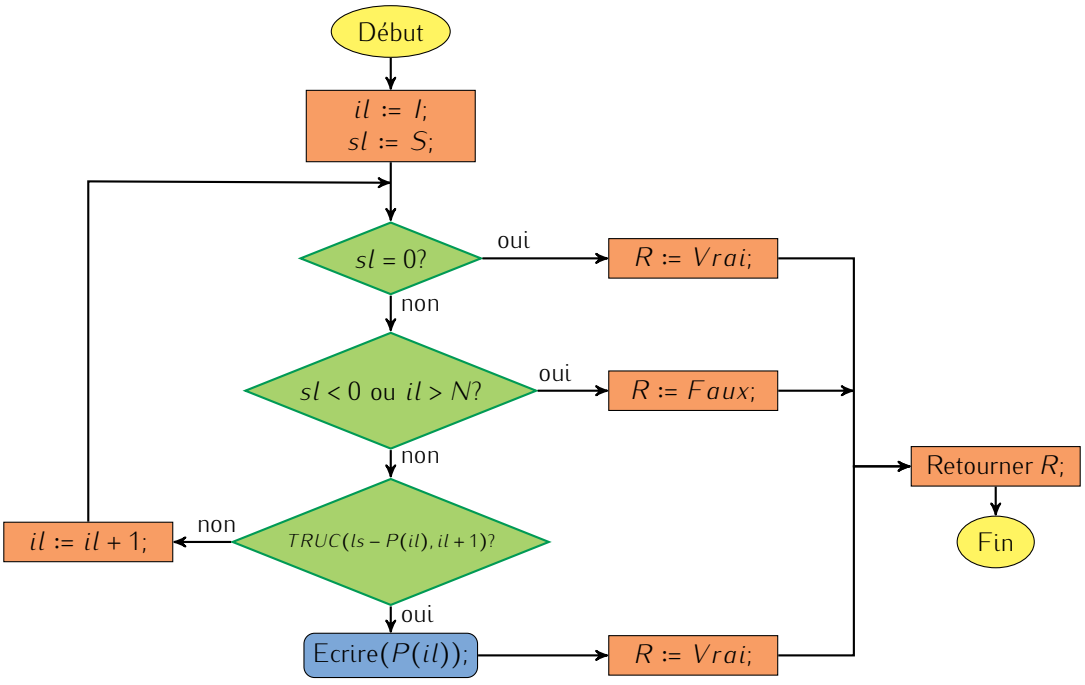



FIGURE 3.2 – Suppression des appels terminaux

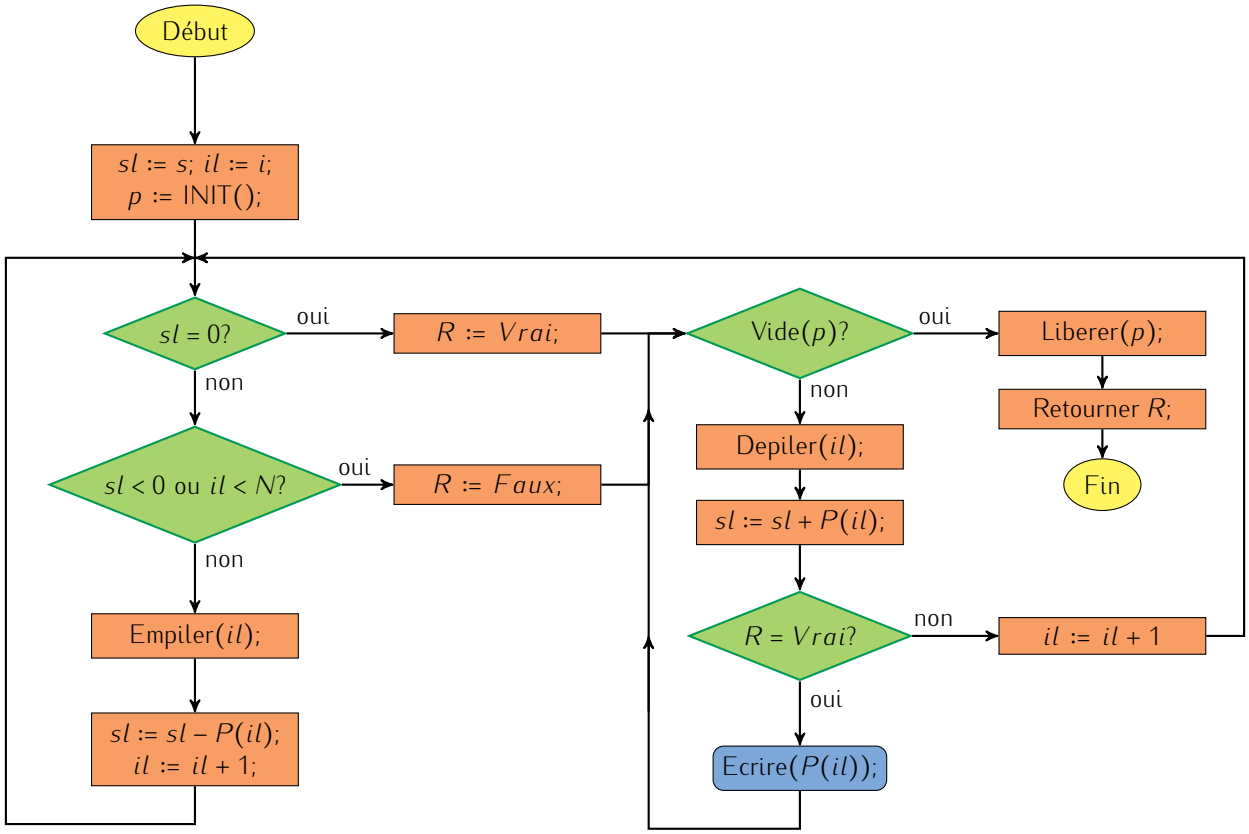



FIGURE 3.3 – Suppression des appels non-terminaux

### 3.2.2 truc_iter

```
Algorithme truc_iter (Principe)
Début
    Copie des paramètres d'entrée dans des variables locales, sl et il;
    Initialisation de la pile de la même taille que le tableau statique;
    Si l'initialisation a réussi Alors
        Répéter
            TantQue sl > 0 Et  il ≤ N Faire
                On envoie il dans la pile;
                sl := sl - P(il);
                On incrémente il;
            FinTantQue;
            Si sl = 0 Alors
                Le booléen de retour est à Vrai;
                TantQue la pile n'est pas vide Faire
                    On récupère il à partir de la pile;
                    On ajoute P(il) à sl;
                    On affiche P(il);
                FinTantQue;
            Sinon
                Le booléen de retour est à Faux;
                Si la pile n'est pas vide Alors
                    On récupère il à partir de la pile;
                    On ajoute P(il) à sl;
                    On incrémente il;
                FinSi;
            FinSi;
        TantQue la pile n'est pas vide fait;
        On supprime la pile;
    FinSi;
    Retourner Booléen de retour;
Fin
```

Lexique :

sl : *copie locale du nombre s passé en paramètre. Représente le nombre à décomposer*

il : *copie locale du nombre i passé en paramètre. Représente le point de départ dans le tableau pour décomposer s*

r : *booléen de retour, indique 1 si on a obtenu la somme s, 0 sinon*

p : *pile*

P : *tableau d'entiers, défini statiquement*

N : *taille du tableau P*

# 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 truc.c
OBJECTS=$(SOURCES:.c=.o)

EXEC=prog

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

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

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

## 4.2 Jeux de tests

### 4.2.1 Fichier de tests

La structure utilisée dans le fichier de tests est la suivante :

- Chaque ligne ne peut excéder 100 caractères,
- Le premier caractère de chaque ligne peut-être :
  - `#` : Permet d'afficher le texte présent sur la ligne, après le caractère.
  - `p` : Permet de lancer le test de la pile. Nécessite un argument : le nombre d'éléments, donné sur la même ligne.
  - `t` : Permet de lancer la fonction `truc` en version itérative. Prend deux arguments positifs, strictement pour le second. Le premier correspond à S, le second à I.
  - `T` : Idem pour la version récursive.
  - Tout autre caractère ou ligne vide affichera une ligne vide.

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

Test

```
#1) Test de pile avec 5 éléments
p 5

#Comparons les resultats donnes par les deux fonctions
#2) TRUC(15,1)
T 15 1

#3) truc(15,1)
t 15 1

#Exemples de resolutions impossibles
#4) truc(15,7)
t 15 7

#5) TRUC(256,1)
T 256 1
```

On obtient alors le résultat suivant :

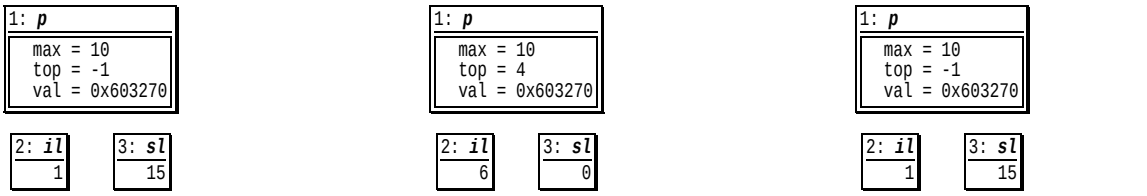
Resultat

```
1) Test de pile avec 5 éléments
Empiler: 0
Empiler: 1
Empiler: 2
Empiler: 3
Empiler: 4
Depiler: 4
Depiler: 3
Depiler: 2
Depiler: 1
Depiler: 0


Comparons les resultats donnes par les deux fonctions
2) TRUC(15,1)
Tableau P : 1 3 2 4 5 2 7 1 9 1
5
4
2
3
1
Reponse: Vrai


3) truc(15,1)
Tableau P : 1 3 2 4 5 2 7 1 9 1
5
4
2
3
1
Reponse: Vrai
```

```
Exemples de resolutions impossibles
4) truc(15,7)
Tableau P : 1 3 2 4 5 2 7 1 9 1
Reponse: Faux


5) TRUC(256,1)
Tableau P : 1 3 2 4 5 2 7 1 9 1
Reponse: Faux
```

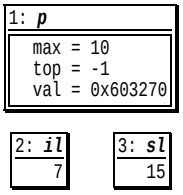
### 4.2.2 Tests de la pile

Avec l'outil ddd nous présentons ici l'utilisation de la pile dans les tests n° 3 et 4 du fichier de tests.

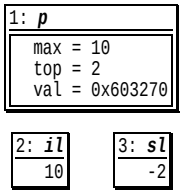



(a) Début (b) Après avoir empilé 1, 2, 3, 4 et 5 (c) Fin, après avoir dépilé 5, 4, 3, 2 et 1

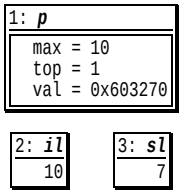
FIGURE 4.1 – Test n°3 – Aperçu des informations de la pile dans ddd



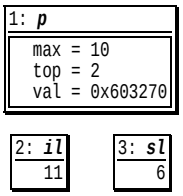

(a) Début



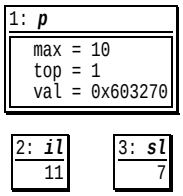

(b) Après avoir empilé 7, 8 et 9



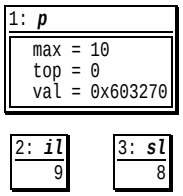

(c) Après avoir dépilé 9



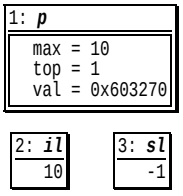

(d) Après avoir empilé 10



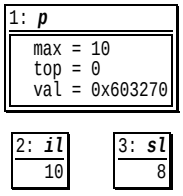

(e) Après avoir dépilé 10



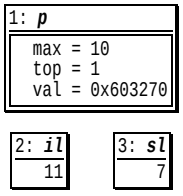

(f) Après avoir dépilé 8



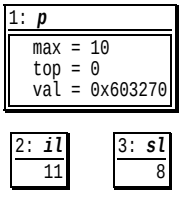

(g) Après avoir empilé 9



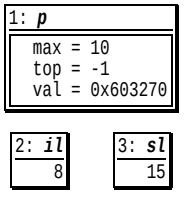

(h) Après avoir dépilé 9




(i) Après avoir empilé 10




(j) Après avoir dépilé 10




(k) Fin, après avoir dépilé 7

FIGURE 4.2 – Test n°4 – Aperçu de la pile dans ddd

### 4.2.3 Bonne utilisation de la mémoire

Pour vérifier la bonne libération de la mémoire, nous avons utilisé `valgrind` avec le programme seul. La fonction `main`, lorsqu'elle ne reçoit pas d'argument, lance un petit test sur les deux versions de la fonction `truc`, ainsi qu'un test sur la pile (initialiser, empiler jusqu'à ce que la pile soit remplie/dépiler jusqu'à ce qu'elle soit vide, libérer). Aucun bloc de mémoire n'est perdu, le retour texte de valgrind est présenté en figure 4.3.

Resultat Valgrind

```
==12709== Memcheck, a memory error detector
==12709== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==12709== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==12709== Command: ../src/prog
==12709==
Test de pile:
Empiler: 0
Empiler: 1
Empiler: 2
Empiler: 3
Depiler: 3
Depiler: 2
Depiler: 1
Depiler: 0
Fonction recursive:
1
2
2
3
1
Fonction iterative:
1
2
2
3
1
==12709==
==12709== HEAP SUMMARY:
==12709==     in use at exit: 0 bytes in 0 blocks
==12709==   total heap usage: 2 allocs, 2 frees, 56 bytes allocated
==12709==
==12709== All heap blocks were freed -- no leaks are possible
==12709==
==12709== For counts of detected and suppressed errors, rerun with: -v
==12709== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

FIGURE 4.3 – Passage dans l'outil `valgrind`