TP 2 Rapport :
Dérécursification de TRUC

Tremeau Guillaume
et
Prunier Baptiste

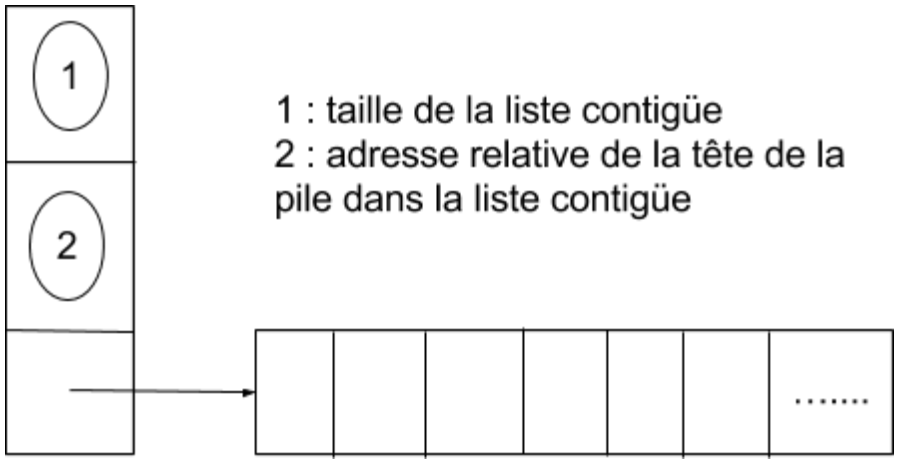
Plan :

Présentation du travail demandé

Le travail qui est demandé dans ce TP est de dérécursifier la fonction TRUC donnée. Pour ce faire, il faut d'abord faire une trace de cette fonction sur un petit cas pour prendre connaissance de son fonctionnement et implémenter la version récursive. En parallèle, il faut implémenter la structure de donnée de pile vue en cours, ainsi que les différentes fonctions et procédures de base liées à cette fonctions : empiler, dépiler, initialiser une pile vide et savoir si la pile est vide ou non. Une fois cela fait, il faut dérécursifier la fonction TRUC et implémenter le résultat.

I/ Nos structures de données

Ci-dessous le schéma de la structure de donnée de la pile utilisée lors de ce travail :

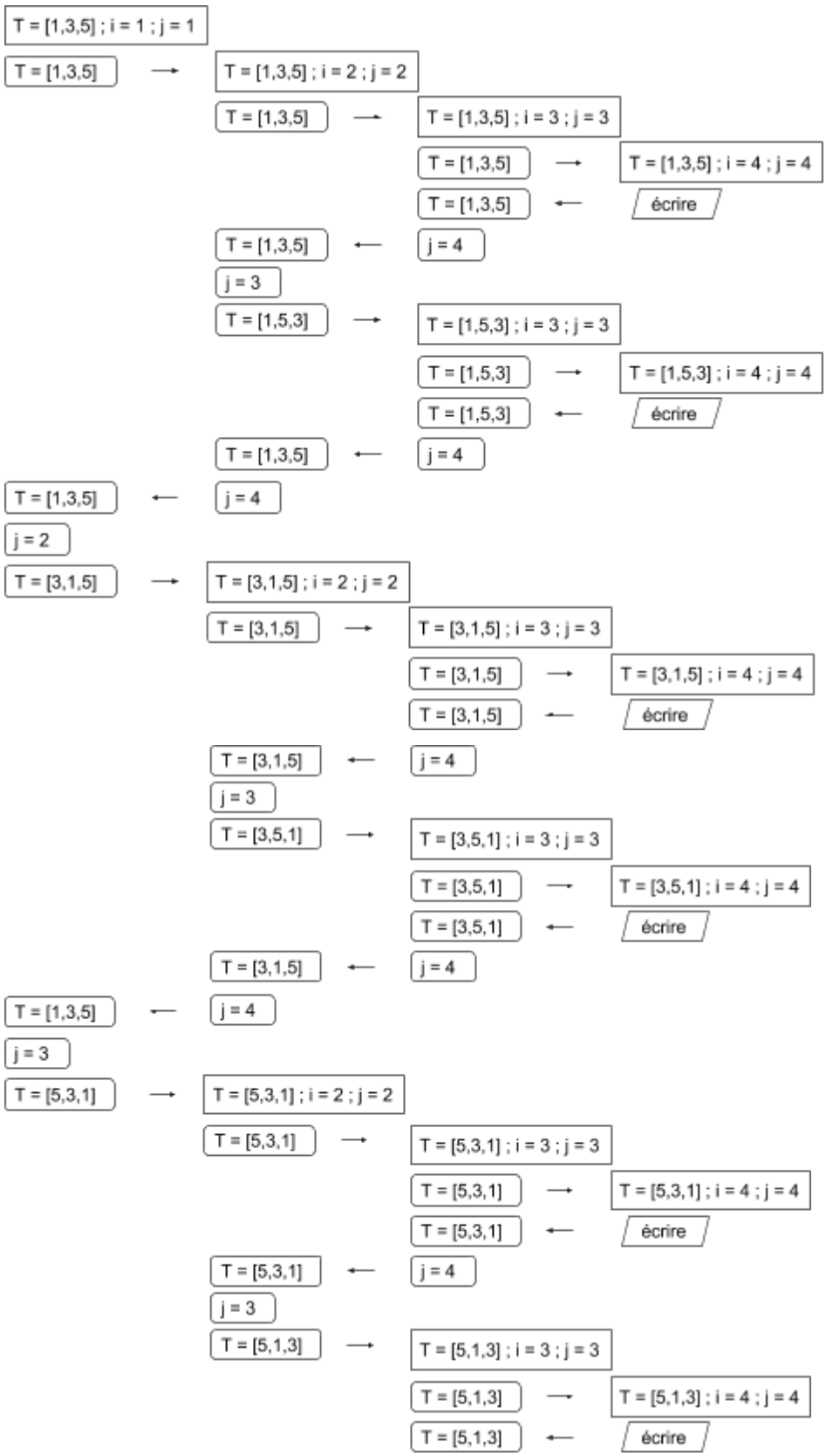


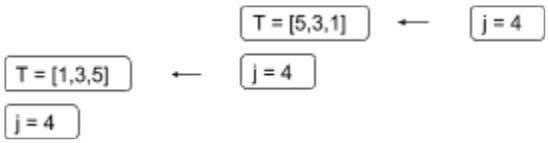
II/ Trace et dérécursification

A) Algorithme donné

```
Procédure TRUC(Entrée : i, n, T)
|       si (i = n) alors
|       |       pour j de 1 à n faire
|       |       |       Ecrire(T[j]);
|       |       fait;
|       sinon
|       |       pour j de i à n faire
|       |       |       temp := T[i];
|       |       |       T[i] := T[j];
|       |       |       T[j] := temp;
|       |       |       TRUC(i+1, n, T);
|       |       |       temp := T[i];
|       |       |       T[i] := T[j];
|       |       |       T[j] := temp;
|       |       fait;
|       fsi;
fin;
```

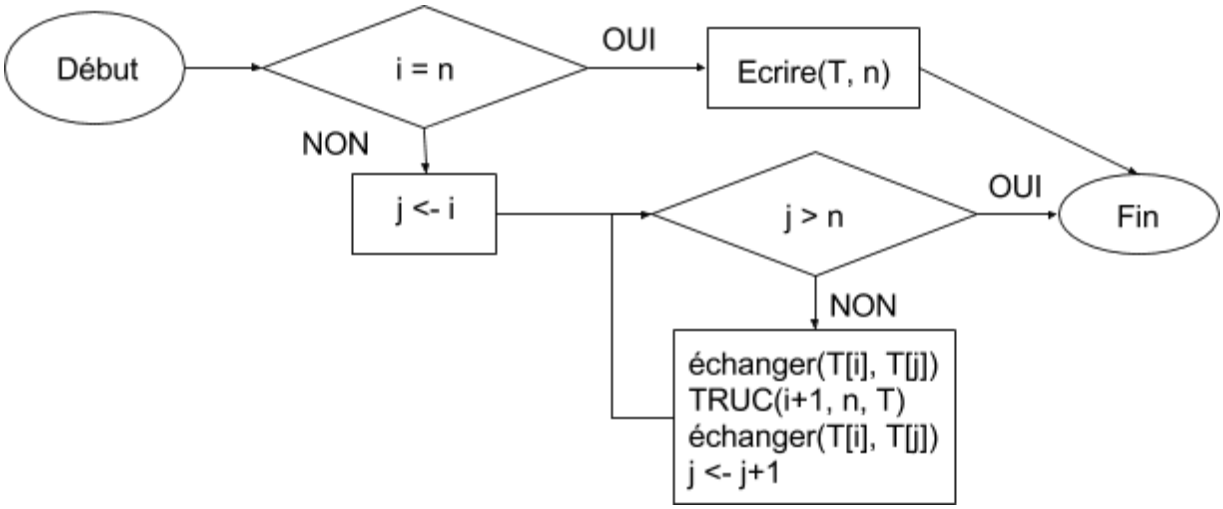
B) Trace


T = [1,3,5] ; i = 1 ; j = 1
T = [1,3,5]
T = [1,3,5] ; i = 2 ; j = 2
T = [1,3,5]
T = [1,3,5] ; i = 3 ; j = 3
T = [1,3,5]
T = [1,3,5] ; i = 4 ; j = 4
T = [1,3,5]
écrire
T = [1,3,5]
j = 4
j = 3
T = [1,5,3]
T = [1,5,3] ; i = 3 ; j = 3
T = [1,5,3]
T = [1,5,3] ; i = 4 ; j = 4
T = [1,5,3]
écrire
T = [1,3,5]
j = 4
T = [1,3,5]
j = 4
j = 2
T = [3,1,5]
T = [3,1,5] ; i = 2 ; j = 2
T = [3,1,5]
T = [3,1,5] ; i = 3 ; j = 3
T = [3,1,5]
T = [3,1,5] ; i = 4 ; j = 4
T = [3,1,5]
écrire
T = [3,1,5]
j = 4
j = 3
T = [3,5,1]
T = [3,5,1] ; i = 3 ; j = 3
T = [3,5,1]
T = [3,5,1] ; i = 4 ; j = 4
T = [3,5,1]
écrire
T = [3,1,5]
j = 4
T = [1,3,5]
j = 4
j = 3
T = [5,3,1]
T = [5,3,1] ; i = 2 ; j = 2
T = [5,3,1]
T = [5,3,1] ; i = 3 ; j = 3
T = [5,3,1]
T = [5,3,1] ; i = 4 ; j = 4
T = [5,3,1]
écrire
T = [5,3,1]
j = 4
j = 3
T = [5,1,3]
T = [5,1,3] ; i = 3 ; j = 3
T = [5,1,3]
T = [5,1,3] ; i = 4 ; j = 4
T = [5,1,3]
écrire

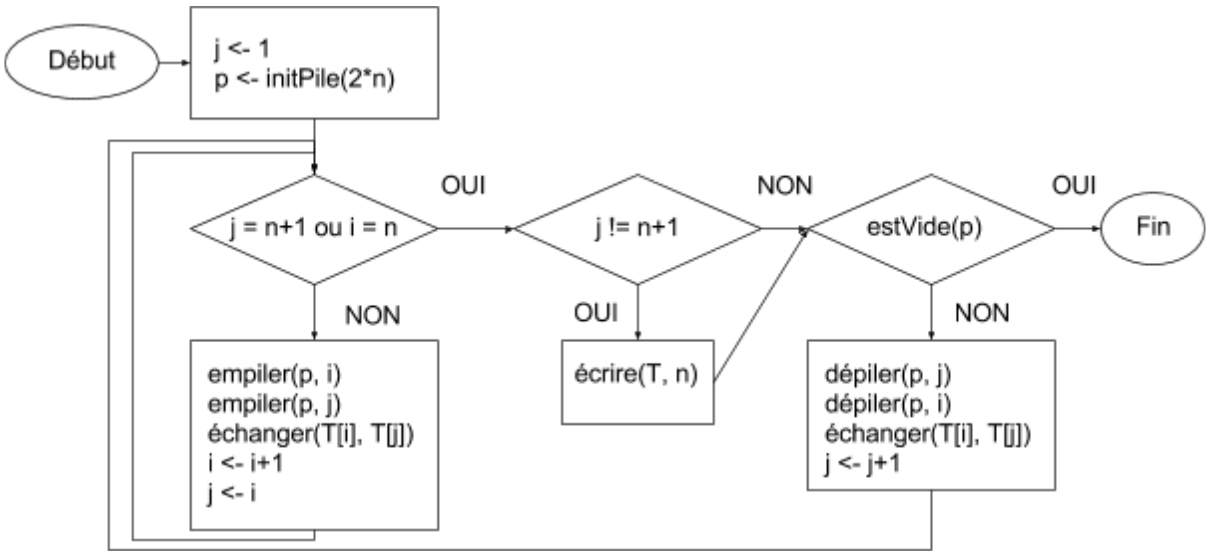


C) Dérécursification

Voici, ci-dessous, l'organigramme de l'algorithme TRUC récursif :



Et voici l'organigramme de la version dérécursifiée de l'algorithme TRUC :



L'algorithme résultant de cette organigramme est le suivant :

Procédure TRUC_Derec(Entrée : i, n, T)
| p := initPile(n*2);
| j := 1;

```
|       continue := 1;
|       tant que (continue = 1) faire
|       |       tant que (j != n+1 et i!=n) faire
|       |       |       empiler(p, i);
|       |       |       empiler(p, j);
|       |       |       échanger(T[i], T[j]);
|       |       |       i := i+1;
|       |       |       j := i;
|       |       fait;
|       |       si (j != n+1) alors
|       |       |       écrit(T, n);
|       |       fsi;
|       |       si (non(estVide(p))) alors
|       |       |       dépiler(p, j);
|       |       |       dépiler(p, i);
|       |       |       échanger(T[i], T[j]);
|       |       |       j := j+1;
|       |       sinon
|       |       |       continue := 0;
|       |       fsi;
|       fait;
fin;
```

III/ Structure de notre code

A) Schéma de cette structure

B) Fonctions contenues dans chaque fichier

Vous trouverez ci-dessous la liste des fonctions contenues dans chaque

couple de fichier source et header, ainsi qu'une brève explication de son rôle.

pile.c/.h :

- init_pile : initialise une pile vide ;
- free_pile : libère la mémoire associée à une pile ;
- est_vide : indique si la pile est vide ou non ;
- empiler : empile une valeur dans la pile si cela est possible, indique si l'empilage a pu être fait ;
- depiler : dépile une valeur de la pile si cela est possible, indique si le dépilage a pu être fait.

truc.c/.h :

- afficher_tab : affiche un tableau d'entiers dont la taille est passée en paramètre ;
- echanger : échange les valeurs pointées par deux pointeurs ;
- truc : algorithme TRUC en version itérative ;
- truc_rec : algorithme TRUC en version récursive.

IV/ Jeux d'essais

A) Ensemble des cas à traiter dans les tests

Vous trouverez ci-dessous l'ensemble des tests à réaliser par couple de fichier source et header.

pile.c/.h :

- vérifier l'initialisation de la pile ;
- vérifier est_vide sur une pile vide ;
- tester l'empilage d'une valeur dans une pile ayant suffisamment de place ;
- tester l'empilage d'une valeur dans une pile pleine ;
- vérifier est_vide sur une pile non vide ;
- tester le dépilage sur une pile non vide ;
- tester le dépilage sur une pile vide ;
- tester la libération de la mémoire.

truc.c/.h :

- vérifier que le résultat de l'implémentation de TRUC récursif ;
- comparer les résultats avec ceux de l'implémentation de TRUC itératif.

B) Traitement de ces différents cas

Pour les jeux d'essais sur le couple de fichier pile.h/.c, nous avons utilisé un main de test (voir l'annexe). Son exécution avec valgrind donne le résultat suivant :

```
baptiste@baptiste-X750LN ~/Programmation/C/ZZ1/S2/Pile_derecursification/test_pi
le $ valgrind ./truc
==2625== Memcheck, a memory error detector
==2625== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2625== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2625== Command: ./truc
==2625==
--- Tests de la Pile ---

Test d'initialisation de la pile :
Curseur : -1 == -1
Taille : 2 == 2
Tableau : (nil) != 0x5203480


Test de est_vide partie 1 :
1 == 1

Test d'empilage de valeurs :
0 == 0
0 == 0
1 == 1
Pile supposee : 3 5
Pile reelle : 3 5

Test de est_vide partie 2
0 == 0

Test de depilage :
0 == 0
Valeur depilee : 3 == 3
0 == 0
Valeur depilee : 5 == 5
1 == 1
==2625==
==2625== HEAP SUMMARY:
==2625==     in use at exit: 0 bytes in 0 blocks
==2625==   total heap usage: 3 allocs, 3 frees, 1,036 bytes allocated
==2625==
==2625== All heap blocks were freed -- no leaks are possible
==2625==
==2625== For counts of detected and suppressed errors, rerun with: -v
==2625== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Pour tester les résultats obtenus par les deux fonctions truc, nous avons testé tous les appels possibles avec un tableau de taille SIZE (macro défini dans le main général), avec comme paramètre n allant de 1 à SIZE et comme paramètre i allant de n à 1. Le résultat obtenu avec SIZE égal à trois est le suivant :

```
baptiste@baptiste-X750LN ~/Programmation/C/ZZ1/S2/Pile_derecursification $ valgrind ./truc
==3697== Memcheck, a memory error detector
==3697== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3697== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3697== Command: ./truc
==3697==
Tableau de test : 1 2 3

Taille du tableau : 1 ; indice de depart : 1.
Resultat de l'appel de truc recursif :
1
Resultat de l'appel de truc iteratif :
1

Taille du tableau : 2 ; indice de depart : 2.
Resultat de l'appel de truc recursif :
1 2
Resultat de l'appel de truc iteratif :
1 2

Taille du tableau : 2 ; indice de depart : 1.
Resultat de l'appel de truc recursif :
1 2
2 1
Resultat de l'appel de truc iteratif :
1 2
2 1

Taille du tableau : 3 ; indice de depart : 3.
Resultat de l'appel de truc recursif :
1 2 3
Resultat de l'appel de truc iteratif :
1 2 3

Taille du tableau : 3 ; indice de depart : 2.
Resultat de l'appel de truc recursif :
1 2 3
1 3 2
Resultat de l'appel de truc iteratif :
1 2 3
1 3 2

Taille du tableau : 3 ; indice de depart : 1.
Resultat de l'appel de truc recursif :
1 2 3
1 3 2
2 1 3
2 3 1
3 2 1
3 1 2
Resultat de l'appel de truc iteratif :
1 2 3
1 3 2
2 1 3
2 3 1
3 2 1
3 1 2
==3697==
==3697== HEAP SUMMARY:
==3697==     in use at exit: 0 bytes in 0 blocks
==3697==   total heap usage: 7 allocs, 7 frees, 1,136 bytes allocated
==3697==
==3697== All heap blocks were freed -- no leaks are possible
==3697==
==3697== For counts of detected and suppressed errors, rerun with: -v
==3697== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
baptiste@baptiste-X750LN ~/Programmation/C/ZZ1/S2/Pile_derecursification $
```

Conclusion

Aucun des problèmes que nous avons rencontrés n'est restés non résolus. Le projet pourrait être amélioré en changeant la structure de donnée de la pile par une liste chaînée au lieu d'une liste contiguë. En effet, la liste chaînée permet une taille mémoire dynamique en temps constant et étant donnée que la pile ne nécessite un accès qu'à sa tête, le parcours, qui est le point fort de la liste contiguë, n'est pas nécessaire.

Annexe

1. Piles :

    a. pile.h

```
#ifndef _PILE_PILEDEREC_TREMEAU_KLEVH_
#define _PILE_PILEDEREC_TREMEAU_KLEVH_

#include <stdlib.h>

typedef int valeur_t;

typedef struct
{
  valeur_t * valeurs;
  int taille;
  int curseur;
}pile_t;

/* ------------------------------------------------
 *
 *          INIT_PILE
 *
 * Retourne une pile vide de taille maximal indique
 * en parametre.
 *
 * Retour :
 *  - tete de la pile initialisee
 * ------------------------------------------------ */
pile_t init_pile(unsigned);
```

```
/* ------------------------------------------------
 *
 *          FREE_PILE
 *
 * Libere la memoire allouee a la pile passee en
 * parametre.
 * ------------------------------------------------ */
void free_pile(pile_t *);

/* ------------------------------------------------
 *
 *          EST_VIDE
 *
 * Indique si une pile passee en parametre est vide
 * ou non.
 *
 * Retour :
 *  - 1 si la pile est vide
 *  - 0 sinon
 * ------------------------------------------------ */
int est_vide(pile_t);

/* ------------------------------------------------
 *
 *          EMPILER
 *
 * Empile la valeur passee en parametre dans la pile
 * egalement passee en parametre si la pile n'est pas
 * pleine.
 *
 * Retour :
 *  - 0 si l'empilage a pu etre fait
 *  - 1 sinon
 * ------------------------------------------------ */
int empiler(pile_t *,valeur_t);

/* ------------------------------------------------
 *
 *          DEPILER
 *
 * Depile la premiere valeure de la pile passee en
 * parametre dans l'adresse egalement passee en
 * parametre.
 *
 * Retour :
 *  - 0 si le depilage a pu etre fait
 *  - 1 sinon
 * ------------------------------------------------ */
int depiler(pile_t *,valeur_t*);

#endif
```

b. pile.c

```
#include "pile.h"

/* ------------------------------------------------
 *
 *          INIT_PILE
 *
 * Retourne une pile vide de taille maximal indique
 * en parametre.
 *
 * Lexique :
 *  - p : tete d'une pile vide (sortie)
 *  - taille_max : taille maximal de la pile (entree)
 *
 * Retour :
 *  - p
 * ------------------------------------------------ */
pile_t init_pile(unsigned taille_max){
  pile_t p;

  if(taille_max){
      p.valeurs = (valeur_t *)malloc(taille_max*sizeof(valeur_t));
  }else{
      p.valeurs=NULL;
  }

  if(p.valeurs){
      p.taille=taille_max;
  }else{
      p.taille = 0;
  }

  p.curseur = -1;

  return p;
}

/* ------------------------------------------------
 *
 *          FREE_PILE
 *
 * Libere la memoire allouee a la pile passee en
 * parametre.
 *
 * Lexique :
 *  - p : adresse de la tete de la pile a liberer
 *    (entree/sortie)
 * ------------------------------------------------ */
```

```
void free_pile(pile_t * p){
  if(p && p->valeurs){
      free(p->valeurs);
      p->valeurs = NULL;
  }
}

/* -------------------------------------------------
 *
 *          EST_VIDE
 *
 * Indique si une pile passee en parametre est vide
 * ou non.
 *
 * Lexique :
 *  - p : tete de la pile (entree)
 *
 * Retour :
 *  - 1 si la pile est vide
 *  - 0 sinon
 * ------------------------------------------------- */
int est_vide(pile_t p){
  return p.curseur == -1;
}

/* -------------------------------------------------
 *
 *          EMPILER
 *
 * Empile la valeur passee en parametre dans la pile
 * egalement passee en parametre si la pile n'est pas
 * pleine.
 *
 * Lexique :
 *  - p : adresse de la tete de la pile (entree/sortie)
 *  - v : valeur a ajouter dans la pile (entree)
 *  - erreur : entier vallant 0 si l'empilage a pu
 *          etre fait, 1 sinon (sortie)
 *
 * Retour :
 *  - erreur
 * ------------------------------------------------- */
int empiler(pile_t * p,valeur_t v){
  int erreur = 1;

  if(p && p->valeurs && p->curseur < p->taille - 1){
      p->curseur++;
      p->valeurs[p->curseur] = v;
      erreur = 0;
  }
```

```
  return erreur;
}

/* ------------------------------------------------
 *
 *          DEPILER
 *
 * Depile la premiere valeure de la pile passee en
 * parametre dans l'adresse egalement passee en
 * parametre.
 *
 *
 * Lexique :
 *  - p : adresse de la tete de la pile (entree/sortie)
 *  - v : adresse de destination de la valeur depilee
 *    (entree)
 *  - erreur : entier vallant 0 si le depilage a pu
 *          etre fait, 1 sinon (sortie)
 *
 * Retour :
 *  - erreur
 * ------------------------------------------------ */
int depiler(pile_t * p,valeur_t * v){
  int erreur = 1;

  if(p && p->valeurs && !est_vide(*p)){
      *v = p->valeurs[p->curseur];
      p->curseur--;
      erreur = 0;
  }

  return erreur;
}
```

c. main de test

```
#include "../pile.h"
#include <stdio.h>

void affiche_pile(pile_t p){
  int i;
  if(p.curseur!=-1){
      if(p.valeurs){
      for(i=p.curseur;i>=0;--i){
    printf("%d ",p.valeurs[i]);
      }

      printf("\n");
      }else{
```

```
      printf("Le curseur ne vaut pas -1 mais la pile est libere\n");
      }
  }else{
      printf("Pile vide\n");
  }
}

int main()
{
  pile_t p;
  valeur_t * v;

  printf("--- Tests de la Pile ---\n\n");

  printf("Test d'initialisation de la pile :\n");
  p = init_pile(2);
  printf("Curseur : -1 == %d\n",p.curseur);
  printf("Taille : 2 == %d\n",p.taille);
  printf("Tableau : (nil) != %p\n\n",p.valeurs);

  printf("\nTest de est_vide partie 1 :\n");
  printf("1 == %d\n",est_vide(p));

  printf("\nTest d'empilage de valeurs :\n");
  printf("0 == %d\n",empiler(&p,5));
  printf("0 == %d\n",empiler(&p,3));
  printf("1 == %d\n",empiler(&p,2));
  printf("Pile supposee : 3 5\n");
  printf("Pile reelle : ");
  affiche_pile(p);

  printf("\nTest de est_vide partie 2\n");
  printf("0 == %d\n",est_vide(p));

  printf("\nTest de depilage :\n");
  v = (valeur_t *)malloc(sizeof(valeur_t));
  if(!v){
      printf("Probleme d'allocation dans le testeur (et non dans le fichier
teste)\n");
  }else{
      printf("0 == %d\n",depiler(&p,v));
      printf("Valeur depilee : 3 == %d\n",*v);
      printf("0 == %d\n",depiler(&p,v));
      printf("Valeur depilee : 5 == %d\n",*v);
      printf("1 == %d\n",depiler(&p,v));

      free(v);
  }

  free_pile(&p);
```

```
  return 0;
}
```

d. makefile de test

```
#!/bin/makefile

OFLAG = -Wall -pedantic -ansi -Wextra
CFLAG =
DEBUG = -g
PROG =truc
OFILE = main.o ../pile.o

$(PROG):$(OFILE)
    gcc $(PROG) -o $@ $(OFLAG) $(DEBUG)

.o:.c
    gcc -c $< $(CFLAG)

clean:
    rm *.o *~
clear: clean
```

2. Truc :

a. truc.h

```
#ifndef _TRUC_PILEDEREC_TREMEAU_KLEVH_
#define _TRUC_PILEDEREC_TREMEAU_KLEVH_

#include <stdio.h>
#include "pile.h"

/*-------------------------------------------
 *
 *                 AFFICHER_TAB
 *
 *    Parcours un tableau de valeur à afficher
 * ------------------------------------------*/
void afficher_tab(valeur_t *,int);


/*-------------------------------------------
 *
 *                 ECHANGER
```

```
 *
 *    Echange deux valeurs
 * ----------------------------------------*/
void echanger(valeur_t *, valeur_t *);


/*-----------------------------------------
 *
 *                 TRUC
 *
 *    Algorithme truc en version iterative
 * ----------------------------------------*/
void truc(int, int, valeur_t *);


/*-----------------------------------------
 *
 *                 TRUC_REC
 *
 *    Algorithme truc en version recursive
 * ----------------------------------------*/
void truc_rec(int, int, valeur_t *);

#endif
```

b. truc.c

```
#include "truc.h"

/*-----------------------------------------
 *
 *                 AFFICHER_TAB
 *
 *    Parcours un tableau de vakeur à afficher
 *
 *    Lexique :
 *     - tab : Tableau à afficher (entree)
 *  - n : Taille du tableau à afficher (entree)
 *
 * ----------------------------------------*/
void afficher_tab(valeur_t * tab, int n){
    int j;
    for(j=0;j<n;j++){
       printf("%d ", tab[j]);
    }
    printf("\n");
}
```

```
/*-----------------------------------------
 *
 *                 ECHANGER
 *
 *    Echange deux valeurs
 *
 *    Lexique :
 *  - a : premiere valeur d'echange
 *  - b : seconde valeur d'echange
 * ----------------------------------------*/
void echanger(valeur_t * a, valeur_t * b){
    valeur_t temp = *b;
    *b = *a;
    *a = temp;
}

/*-----------------------------------------
 *
 *                 TRUC
 *
 *    Algorithme truc en version iterative
 *
 *    Lexique :
 *  - i
 *  - n : taille max du tableau de valeur (entree)
 *  - tab : tableau de valeur (entree/sortie)
 *  - j : variable d'itération
 *  - pile : pile d'utilisation de l'algorithme
 *  - continuer : valeur (0 ou 1) indiquant s'il faut
 * continuer l'algorithme ou non
 * ----------------------------------------*/

void truc(int i, int n, valeur_t * tab){
      pile_t pile = init_pile(2*n);;
    int     j = i-1;
    int     iInit = i;
    int     continuer = 1;


    while (continuer){
       while (i != n && j!=n) {
             empiler(&pile, i);
             empiler(&pile, j);
             echanger(tab+i-1, tab+j);
             i++;
             j=i-1;
       }
       if(j!=n){
         afficher_tab(tab, n);
       }
       if(!est_vide(pile)){
```

```
            depiler(&pile, &j);
            depiler(&pile, &i);
            echanger(tab+i-1, tab+j);
            j++;
       }else{
            continuer = 0;
       }
    }

    free_pile(&pile);
}

/*-------------------------------------------
 *
 *                 TRUC_REC
 *
 *    Algorithme truc en version recursive
 *
 *    Lexique :
 *  - i
 *  - n : taille max du tableau de valeur (entree)
 *  - tab : tableau de valeur (entree/sortie)
 *  - j : variable d'iteration
 * -------------------------------------------*/
void truc_rec(int i, int n, valeur_t * tab){
    int j;
    if (i == n){
       afficher_tab(tab, n);
    }else{
       for(j=i-1; j<n; j++){
            echanger(tab+i-1, tab+j);
            truc_rec(i+1, n, tab);
            echanger(tab+i-1,tab+j);
       }
    }
}
```