ISIMA Première Année

Compte-Rendu de TP

Structures de Données

Dérécurssification à l'aide d'une pile

Benjamin BARBESANGE
Pierre-Loup PISSAVY
Groupe G21

Enseignant :
Michelle CHABROL

février 2015

ISIMA

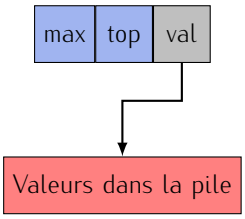
1 | Présentation

Le but de ce TP est de dérécurssifier une fonction à l'aide d'un pile. Les fonctions de gestion d'une pile seront ainsi crées. Les opérations suivantes sont permisesavec 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

(a) Structure utilisée



(b) Code

Code C

```c
typedef struct _stack_t {
  int       max;
  int       top;
  datatype *val;
} stack_t;
```

FIGURE 1.1 – Structure et code correspondant

1.2 Organisation du code source

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

1.2.1 Gestion de la pile

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

1.2.2 Fonction récurssive

- `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
#ifndef __STACK__H
#define __STACK__H

  typedef int datatype;

  typedef struct _stack_t {
    int       max;
    int       top;
    datatype *val;
  } 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
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

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) {
  free(p->val);
}

int empty(stack_t p) {
  return (p.top == -1)?1:0;
}

int full(stack_t p) {
  return (p.top == p.max-1)?1:0;
}

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) {
  int ok = 0;
  if (!empty(*p)) {
    *v = p->val[p->top];
    ok = 1;
  }
  return ok;
}

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 récurssive

Code C

```c
#ifndef __TRUC__H
#define __TRUC__H

/* Fonction sous forme recurssive */
int truc(int, int);

/* Variante iterative de la fonction */
int truc_iter(int, int);

#endif
```

Code C

```c
#include <stdio.h>
#include "stack.h"

#define N 10

int P[N+1] = {0,1,3,2,5,5,2,7,1,9,1};

int TRUC(int,int);

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) {
  int sl = s;
  int il = i;
  int r;
  stack_t p;
  init(&p,N);
  do {
    while (sl > 0 && il <= N) {
      push(&p,sl);
      push(&p,il);
      sl -= P[il];
      ++il;
    }
    if (sl == 0) {
      r = 1;
```

```
      while (!empty(p)) {
        pop(&p,&il);
        pop(&p,&sl);
        printf("%d\n",P[il]);
      }
    } else {
      r = 0;
      if (!empty(p)) {
        pop(&p,&il);
        pop(&p,&sl);
        ++il;
      }
    }
  } while (!empty(p));
  return r;
}

int main(void) {
  int S, I;
  do {
    S = -1;
    while (S <= 0) {
      printf("S? ");
      scanf("%d",&S);
    }
    printf("I? ");
    scanf("%d",&I);

    if (I >= 0) {
      printf("Recursif:\n");
      TRUC(S,I);
      printf("Iteratif:\n");
      truc_iter(S,I);
    }
  } while (I >= 0);
  return 0;
}
```

2.3 Programme principal

Code C

```c
#include <stdio.h>
#include "stack.h"

int main(void) {
  stack_t p;
  int i = 0;
  if (init(&p,10)) {
    while (push(&p,i)) {
      printf("Empiler: %d\n",i);
      ++i;
    }
    while (pop(&p,&i)) {
      printf("Depiler: %d\n",i);
    }
    supp(&p);
  }
  return 0;
}
```

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.

Le lexique des variable d'entrée, sortie et entrée/sortie sont disponibles dans le code source directement.

3.1 Gestion des news

La gestion des news s'effectue grace aux fichiers `gestion_news.c` et `gestion_news.h`.

3.1.1 charger

```
Algorithme Charger (Principe)
Début
    On ouvre en lecture seule le fichier dont le nom est passé en paramètre;
    On initialise le code de retour à 1;   [ erreur d'ouverture de fichier ]
    Si on a pu l'ouvrir Alors
        Le code de retour passe à 0;   [ pas de problème d'ouverture de fichier ]
        TantQue l'on n'arrive pas à la fin du fichier Faire
            On lit une ligne;
            On stocke les date de début et de fin de message dans des variables;
            On stocke le message dans une chaîne de caractère;
            On crée une nouvelle cellule à partir des variables ci-dessus récupérées;
            Si on a pu créer la cellule Alors
                On l'insère dans la liste chaînée;
            Sinon
                Le code de retour passe a 2;   [ erreur allocation de cellule ]
            FinSi;
        FinTantQue;
        On ferme le fichier;
    FinSi;
    Retourner le code d'erreur;
Fin
```

Lexique :

ret : *entier qui retourne un code d'erreur*
- *0 si aucune erreur*
- *1 si problème lors de l'ouverture du fichier*
- *2 si problème lors de la création de la cellule*

*fichier : *descripteur de fichier, ouvert a partir du nom passé en paramètres*

debut, fin : *entiers servant à stocker les dates de début et de fin de message lors du traitement du fichier*

*scan : *chaîne de caractère permettant de stocker le message lors du traitement du fichier*

*tmp : *pointeur de cellule qui servira de cellule temporaire à chaîner dans la liste*

3.1.2 sauver

```
Algorithme Sauver (Principe)
Début
    On crée un nouveau fichier à partir du nom donné en paramètre;
    On initialise le code de retour à 1;   [ erreur de création de fichier ]
    Si l'on a pu créer le fichier Alors
        TantQue l'on n'est pas à la fin de la liste Faire
            On écrit les élément de la cellule en cours de traitement dans le fichier;
            On passe à la cellule suivante;
        FinTantQue;
        On ferme le fichier;
        On passe le code de retour à 0;   [ pas d'erreur ]
    FinSi;
    Retourner le code d'erreur;
Fin
```

Lexique :

ret : *entier représentant le code d'erreur de la fonction*
- *0 aucune erreur*
- *1 erreur de création ou d'ouverture du fichier*

*cour : *pointeur sur la cellule en cours de traitement dans la liste chaînée*

*fichier : *descripteur de fichier crée à partir du nom passé en paramètre*

3.1.3 getDate

Cette fonction est fournie pour le TP, nous ne la détaillerons donc pas.

3.1.4 afficher_message

```
Algorithme afficher_message (Principe)
Début
    On écrit les données d'une cellule en récupérant ses attributs;
Fin
```

3.1.5 afficher_liste

Algorithme afficher_liste (Principe)
Début
    On initialise l'élément courant au début de la liste;
    TantQue l'élément courant n'est pas NULL Faire
        On affiche les élément de la cellule avec *afficher_message*;
        On passe à la cellule suivante;
    FinTantQue;
Fin

Lexique :
    *cour : *pointeur sur la cellule en cours de traitement dans la liste chaînée*

3.1.6 afficher_messages_date

Algorithme afficher_messages_date (Principe)
Début
    On initialise l'élément courant au début de la liste;
    TantQue la date de début de l'élément courant est supérieure à la date à traiter Faire
        On passe à la cellule suivante;
    FinTantQue;
    TantQue la liste n'est pas finie Faire
        TantQue la date de fin de l'élément courant est ultérieure à la date recherchée Faire
            On affiche le message de l'élément courant;
            On passe à la cellule suivante;
        FinTantQue;
        Si la liste n'est pas finie Alors
            On passe à la cellule suivante;
        FinSi;
    FinTantQue;
Fin

Lexique :
    *cour : *pointeur sur la cellule en cours de traitement dans la liste chaînée*

3.1.7 afficher_messages_jour

Algorithme afficher_messages_jour (Principe)
Début
    On appelle la fonction *afficher_messages_date* avec en paramètre la date du jour obtenue avec *getDate*;
Fin

Lexique :
    Aucune variable intermédiaire n'est utilisée dans cette fonction:

3.1.8 afficher_messages_motif

Algorithme afficher_messages_motif (Principe)
Début
    On initialise l'élément courant au début de la liste;
    TantQue l'élément courant n'est pas NULL Faire
        Si le message contenu dans la cellule courant contient le motif Alors
            On affiche ce message;
        FinSi;
        On passe à l'élément suivant;
    FinTantQue;
Fin

Lexique :
    *cour: *Pointeur sur la cellule en cours de traitement dans la liste chaînée*

3.1.9 supprimer_obsoletes

Algorithme supprimer_obsoletes (Principe)
Début
    On récupère la date du jour dans un entier;
    On initialise un pointeur double **prec au début de la liste (ceci afin d'économiser l'utilisation du'une variable supplémentaire);
    TantQue *prec n'est pas NULL Faire
        Si la date de fin de message de *prec est inférieure à la date du jour Alors
            On supprime cette cellule à partir du précédent prec;
        Sinon
            On passe à l'élément suivant;
        FinSi;
    FinTantQue;
Fin

Lexique :

date : *Date du jour récupérée avec la fonction getDate*

**prec : *Pointeur double servant à parcourir la liste en ayant toujours accès à l'élément précédent d'une cellule*

3.1.10 remplacer_date

Algorithme remplacer_date (Principe)
Début
On initialise l'élément courant que début de la liste;
TantQue l'élémént courant n'est pas NULL Et Alors La date de début de message de l'élément courant est supérieure à la date à modifier Faire
On passe à l'élément suivant;
FinTantQue;
TantQue l'élément courant n'est pas NULL Et Alors La date de début de message de l'élément courant est égale à la date à modifier Faire
Si la date de fin de l'élément courant est supérieure ou égale à la nouvelle date Alors
On modifie la date de début de message de l'élément courant à la nouvelle date passée en paramètre;
FinSi;
On passe à l'élément suivant;
FinTantQue;
On sauvegarde la liste courante dans un fichier temporaire;
On libère la liste;
On recharge la liste à partir du fichier temporaire, ce qui va recréer la liste triée;
Fin

Lexique :

*cour : *Pointeur sur la cellule en cours de traitement dans la liste chaînée*

3.2 Gestion de la liste chaînée

3.2.1 adj_cell

Algorithme adj_cell (Principe)
Début
On a **prec un pointeur sur l'adresse de l'élément suivant à l'élément après lequel insérer;
L'élément suivant de l'élément à ajouter est *prec, l'élément suivant de son précédent;
L'élément suivant du précédent devient l'élément à ajouter;
Fin

Lexique :

> Aucune variable intermédiaire n'est utilisée dans cette fonction :

3.2.2 rech_prec

Algorithme rech_prec (Principe)
Début

> On initialise **prec, un pointeur sur l'adresse du début de la liste;
> TantQue *prec n'est pas NULL Et Alors la date de début de *prec est supérieure à la date après laquelle insérer Faire
>
> > prec prend l'adresse de l'adresse de l'élément suivant dans la liste;
>
> FinTantQue;
> La variable d'entrée/sortie *existe* prend la valeur 1 si la date cherchée est présente, 0 sinon;
> Retourner prec;

Fin

Lexique :

> Aucune variable intermédiaire n'est utilisée dans cette fonction :

3.2.3 supp_cell

Algorithme supp_cell (Principe)
Début

> On sauvegarde l'adresse de l'élément à supprimer dans un pointeur temporaire;
> L'élément suivant du précédent devient l'élément suivant de l'élément à supprimer;
> On libère le texte de la cellule à supprimer (car celui-ci est alloué dynamiquement);
> On libère la cellule à supprimer;

Fin

Lexique :

> *elt : *pointeur temporaire sur la cellule à supprimer*

3.2.4 liberer_liste

Algorithme liberer_liste (Principe)
Début

> TantQue la liste n'est pas vide Faire
>
> > On supprime les cellules une à une;
>
> FinTantQue;
> On place le pointeur de tête de la liste à NULL;

Fin

Lexique :

Aucune variable intermédiaire n'est utilisée dans cette fonction :

3.2.5 ins_cell

Algorithme ins_cell (Principe)
Début

On récupère un pointeur sur l'adresse de l'élément avant lequel insérer avec la fonction *rech_prec*;
On ajoute l'élément dans la liste avec la fonction *adj_cell* en utilisant les variables prec et le pointeur sur l'élément à ajouter;

Fin

Lexique :

existe : *booléen de présence de l'élément recherché, modifié dans rech_prec*
**prec : *pointeur sur l'adresse de l'élément après lequel insérer*

3.2.6 creer_cell

Algorithme (Principe)
Début

On alloue un élément de la liste chaînée;
Si l'allocation a réussi Alors
    On place les dates de début et de fin de message dans les champs appropriés;
    On alloue l'espace nécessaire pour le message dans le champ texte;
    On recopie le texte passé en entrée dans le champ texte de la cellule;
FinSi;
Retourner le nouvel élément créé;

Fin

Lexique :

*elt : *pointeur sur le nouvel élément créé, ce sera également le retour de la fonction. Si celui-ci est à NULL, cela signifie qu'il y a eu un problème lors de son allocation*

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
SOURCES1=main.c stack.c
SOURCES2=truc.c stack.c
OBJECTS1=$(SOURCES1:.c=.o)
OBJECTS2=$(SOURCES2:.c=.o)

#Nom du programme
EXEC1=prog
EXEC2=truc

all: $(EXEC1) $(EXEC2)


$(EXEC1): $(OBJECTS1)
	$(CC) $(CFLAGS) $^ -o $(EXEC1)

$(EXEC2): $(OBJECTS2)
	$(CC) $(CFLAGS) $^ -o $(EXEC2)

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

clean:
	rm $(OBJECTS1) $(EXEC1) $(EXEC2) truc.o
```

4.2 Jeux de tests

Exécution du programme avec le fichier suivant :