Compléments de Programmation

Benoit Donnet

Année Académique 2023 - 2024

Agenda

Agenda

- Chapitre 7: Piles
  - Principe
  - Spécification Abstraite
  - Implémentation
  - Utilisation

Agenda

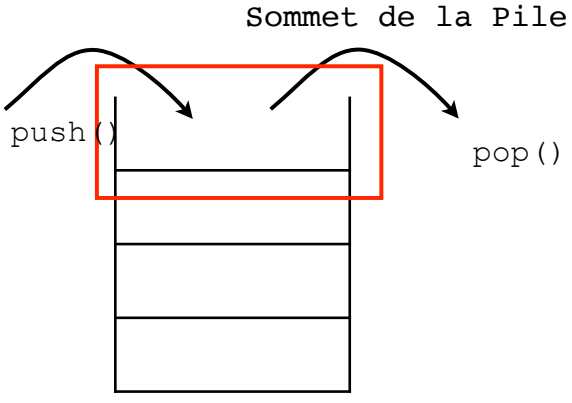
- Chapitre 7: Piles
  - Principe
  - Spécification Abstraite
  - Implémentation
  - Utilisation

Principe

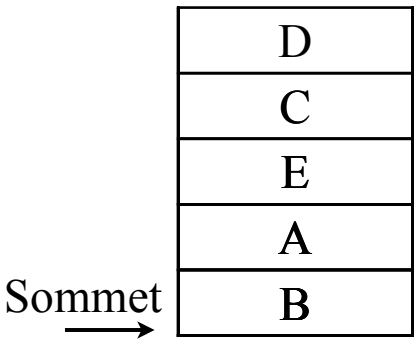
- TAD très utilisé en programmation
  - **stack**
- Notion intuitive
  - pile d'assiettes
  - pile de dossiers
  - ...
- *LIFO*
  - Last-In, First-Out

Principe (2)

- 2 opérations de base
  - insertions d'un élément
    - ✓ `push()`
  - suppression d'un élément
    - ✓ `pop()`
- Les 2 opérations sont restreintes à l'extrémité de la pile
  - **sommet de la pile**



Principe (3)




```
push(D)     pop()
push(C)
push(E)
push(A)
push(B)
```


Agenda

- Chapitre 7: Piles
  - Principe
  - Spécification Abstraite
  - Implémentation
  - Utilisation

Spécification Abstraite

- Syntaxe

```
Type:
  Stack
Utilise:
  Boolean, Element, Natural
Opérations:
  empty_stack: → Stack
  is_empty: Stack → Boolean
  size: Stack → Natural
  push: Stack × Element → Stack
  pop: Stack → Stack
  top: Stack → Element
```


Spécification Abstraite (2)

- Sémantique
  - préconditions

```
Préconditions:
  ∀ s ∈ Stack
  ∀ s, ¬is_empty(s), pop(s)
  ∀ s, ¬is_empty(s), top(s)
```

Spécification Abstraite (3)

		Opérations Internes		
		`empty_stack`	`pop(·)`	`push(·)`
Observateurs	`is_empty(·)`			
	`size(·)`			
	`top(·)`	∅		

- Sémantique
  - axiomes
- Combien d'axiomes?
  - 3 Observateurs × 3 Opérations Internes = 9 Axiomes
- Peut-on réduire?
  - préconditions
  - LIFO

Spécification Abstraite (4)

Axiomes:

∀ s ∈ Stack, ∀ e ∈ Element

`is_empty(empty_stack)` = True

`is_empty(push(s, e))` = False

$$\texttt{is\_empty(pop(s))} = \begin{cases} \textit{True} & \texttt{if size(s)} = 1 \\ \textit{False} & \texttt{otherwise} \end{cases}$$

		Opérations Internes		
		`empty_stack`	`pop(·)`	`push(·)`
Observateurs	`is_empty(·)`	✓	✓	✓
	`size(·)`			
	`top(·)`	∅		

Spécification Abstraite (5)

Axiomes:

∀ s ∈ Stack, ∀ e ∈ Element

```
size(empty_stack) = 0
size(push(s, e)) = size(s) + 1
size(pop(s)) = size(s) - 1
```

		Opérations Internes		
		empty_stack	pop(·)	push(·)
Observateurs	is_empty(·)	✓	✓	✓
	size(·)	✓	✓	✓
	top(·)	∅		

Spécification Abstraite (6)

Axiomes:

∀ s ∈ Stack, ∀ e ∈ Element

```
top(push(s, e)) = e
top(pop(s)) = top(s)

pop(push(s, e)) = s
```

		Opérations Internes		
		empty_stack	pop(·)	push(·)
Observateurs	is_empty(·)	✓	✓	✓
	size(·)	✓	✓	✓
	top(·)	∅	✓	✓

Agenda

- Chapitre 7: Piles
  - Principe
  - Spécification Abstraite
  - Implémentation
    - ✓ Interface
    - ✓ Implémentation Statique
    - ✓ Implémentation Dynamique
    - ✓ Complexité
  - Utilisation

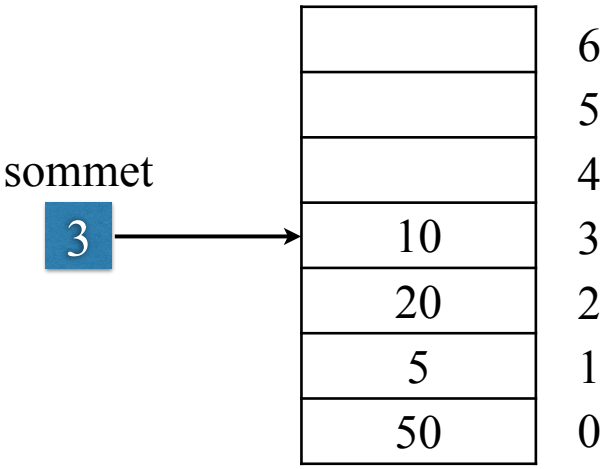
Interface

- Fichier `stack.h`

```
#ifndef __STACK__
#define __STACK__

#include "boolean.h"

typedef struct stack_t Stack;

Stack *empty_stack();

Boolean is_empty(Stack *s);

unsigned int size(Stack *s);

Stack *push(Stack *s, void *e);

Stack *pop(Stack *s);

void *top(Stack *s);
#endif
```

Implem. Statique

- Implémentation via un tableau d'entiers
  - implémentation statique
    - ✓ on peut atteindre, à un moment donné, la capacité maximale de la pile



Implem. Statique (2)

- Fichier `static_stack.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "stack.h"

#define MAX_SIZE 7

struct stack_t{
  void *stack[MAX_SIZE];
  int top;
};
```

Implem. Statique (3)

```
Stack *empty_stack(){
  Stack *s = malloc(sizeof(Stack));
  if(s==NULL)
    return NULL;
  s->top = -1;
  return s;
}//end empty_stack()

Boolean is_empty(Stack *s){
  return s->top==-1;
}//end is_empty()

unsigned int size(Stack *s){
  return s->top+1;
}//end size()

void *top(Stack *s){
  assert(!is_empty(s));

  return s->stack[s->top];
}//end top()
```


Implem. Statique (4)

```
Stack *pop(Stack *s){
  assert(!is_empty(s));

  free(s->stack[s->top]);
  s->top--;
  return s;
}//end pop()

Stack *push(Stack *s, void *e){
  if(s->top >= MAX_SIZE-1){
    printf("Error: stack full!\n");

    exit(-1);
  }
  s->top++;
  s->stack[s->top] = e;

  return s;
}//end push()
```

éviter les fuites mémoires

Implem. Statique (5)

- Utilisation

```
#include <stdio.h>
#include <stdlib.h>

#include "stack.h"
#include "element.h"

int main(){
  Stack *s = empty_stack();

  s = push(s, create_element(50));
  s = push(s, create_element(5));
  s = push(s, create_element(20));
  s = push(s, create_element(10));

  printf("Stack size: %u\n", size(s));

  //à suivre
}//end main()
```

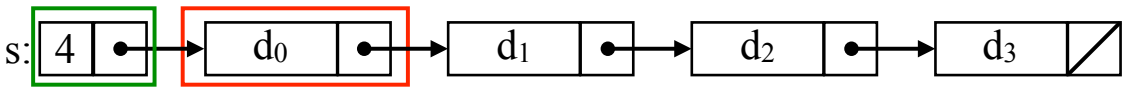

Implem. Statique (6)

```
int main(){
  //cfr. slide précédent

  Element *elem = (Element *)top(s);
  printf("%d\n", elem->e);

  s = pop(s);
  s = pop(s);

  elem = (Element *)top(s);
  printf("%d\n", elem->e);

  printf("Stack size: %u\n", size(s));

  return 1;
}//end main()
```

casting explicite non obligatoire

Implem. Dynamique

- Inconvénient de l'implémentation statique
  - taille limitée de la pile
- Quid si on veut une pile illimitée?
  - implémentation via une liste
    - ✓ l'ajout (`push()`) se fait en tête de liste
    - ✓ le retrait (`pop()`) consiste à retirer la 1ère cellule
    - ✓ cfr. Chap. 6



Sommet de la pile

En-tête

Implem. Dynamique (2)

- Fichier `dynamic_stack.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "stack.h"

typedef struct cell_t{
  void *value;
  struct cell_t *next;
}Cell;

struct stack_t{
  unsigned int size;
  Cell *ptr;
};
```

Liste

En-tête

Implem. Dynamique (3)

```
static Cell *create_cell(void *value){
 Cell *c = malloc(sizeof(Cell));
 if(c == NULL)
  return NULL;

 c->value = value;
 c->next = NULL;
 return c;
}//end create_cell()

Stack *empty_stack(){
 Stack *s = malloc(sizeof(Stack));
 if(s==NULL)
  return NULL;

 s->size = 0;
 s->ptr = NULL;
 return s;
}//end empty_stack()
```


Implem. Dynamique (4)

```
Boolean is_empty(Stack *s){
 return s->size == 0;
}//end is_empty()

unsigned int size(Stack *s){
 return s->size;
}//end size()

Stack *push(Stack *s, void *e){
 Cell *n_cell = create_cell(e);
 if(n_cell==NULL)
  return s;

 n_cell->next = s->ptr;
 s->ptr = n_cell;
 s->size++;

 return s;
}//end push()
```

Implem. Dynamique (5)

```
Stack *pop(Stack *s){
 assert(!is_empty(s));

 Cell *tmp = s->ptr;
 s->ptr = s->prt->next;

 s->size--;

 free(tmp);

 return s;
}//end pop()

void *top(Stack *s){
 assert(!is_empty(s));

 return s->ptr->value;
}//end top()
```


Implem. Dynamique (6)

- Utilisation

```
#include <stdio.h>
#include <stdlib.h>

#include "stack.h"
#include "element.h"

int main(){
  Stack *s = empty_stack();

  s = push(s, create_element(50));
  s = push(s, create_element(5));
  s = push(s, create_element(20));
  s = push(s, create_element(10));

  printf("Stack size: %u\n", size(s));

  //à suivre
}//end main()
```

Implem. Dynamique (7)

```
int main(){
  //cfr. slide précédent

  Element *elem = (Element *)top(s);
  printf("%d\n", elem->e);

  s = pop(s);
  s = pop(s);

  elem = (Element *)top(s);
  printf("%d\n", elem->e);

  printf("Stack size: %u\n", size(s));

  return 1;
}//end main()
```


Complexité

- Les opérations sur les piles sont toutes en *O(1)*
- Valable pour les 2 implémentations

Agenda

- Chapitre 7: Piles
  - Principe
  - Spécification Abstraite
  - Implémentation
  - Utilisation
    - ✓ Généralités
    - ✓ Expression
    - ✓ Evaluation d'une Expression

Généralités

- Nombreuses applications des piles
  - gestion par le compilateur des appels de fonctions
    - ✓ les paramètres, l'adresse de retour et les variables locales sont stockées dans la pile de l'application
  - mémorisation des appels de fonctions imbriqués en cours d'exécution d'un programme
    - ✓ utile pour éliminer la récursivité
    - ✓ cfr. Chapitre 9
  - parcours en profondeur des structures d'arbre
    - ✓ cfr. INFO2050
  - vérification du bon équilibrage d'une expression parenthésée
  - évaluation des expressions arithmétiques

Expression

- **<u>Expressions complètement parenthésées</u>** (ECP)?
  1. une variable est une ECP
  2. soit x et y, des ECPs et β, opérateur binaire, (x β y) est une ECP
  3. soit x, une ECP et α, opérateur unaire, (α x) est une ECP
- Variable?
  - lettre majuscule
- Opérateur binaire?
  - {+, -, ×, /}, {≤, ≥, <, >, =, ≠}, et {∧, ∨}
- Opérateur unaire?
  - {▽, △}, et {¬}

Expression (2)

- On peut exprimer une ECP via une grammaire *BNF*
  - Backus-Naur Form
    - ✓ notation permettant de décrire les règles syntaxiques des langages de programmation
      - cfr. INFO0085

```
<ecp>      ::= (<ecp><opbin><ecp>) | (<opun><ecp>) | <variable>
<opbin>    ::= + | - | × | / | ≤ | ≥ | = | > | < | ≠ | ∧ | ∨
<opun>     ::= △ | ▽ | ¬
<variable> ::= A | B | ... | Z
```

Expression (3)

- Exemples d'ECPs conformes à la grammaire
  - ((A + B) × C)
  - (((A / B) = C) ∧ (E < F))
  - (▽ A)
- Exemples non conformes
  - (A)
  - ((A + B))
  - A △ B

Expression (4)

- Une expression peut aussi être écrite sous forme **<u>préfixée</u>** (EPREF)
  1. une variable est une EPREF
  2. soit x et y, des EPREFs, et β, opérateur binaire, β x y est une EPREF
  3. soit x, une EPREF, et α, opérateur unaire, α x est une EPREF
- BNF

```
<epref> ::= <opbin><epref><epref> | <opun><epref> | <variable>
```

- Exemples
  - ((A - B) + C) ⇒ + - A B C
  - ((¬(A < B)) ∧ C) ⇒ ∧ ¬ < A B C

Expression (5)

- Une expression peut aussi être écrite sous forme **postfixée** (EPOST)
  1. une variable est une EPOST
  2. soient x et y, des EPOST, et β, opérateur binaire, x y β est une EPOST
  3. soit x, une EPOST, et α, opérateur unaire, x α est une EPOST
- BNF

```
<epost> ::= <epost><epost><opbin> | <epost><opun> | <variable>
```

- Exemples
  - ((A - B) + C) ⇒ A B - C +
  - ((¬(A < B)) ∧ C) ⇒ A B < ¬ C ∧

Expression (6)

- Les ECPs sont simples à lire mais fastidieuses à écrire
- Solution?
  - expression sous forme **infixée** (EXPINF)
    - ✓ ordre de priorité entre opérateurs
    - ✓ écriture classique dans les langages de programmation et en algèbre
- BNF

```
<expinf> ::= <expsimple> {<oprel><expsimple>}*
<expsimple> ::= <terme> {<opadd><terme>}*
<terme>   ::= <facteur> {<opmul><facteur>}*
<facteur> ::= <variable> | -<facteur> | ¬<facteur> | (<expinf>)
<oprel>   ::= ≤ | ≥ | = | > | < | ≠
<opadd>   ::= + | - | ∨
<opmul>   ::= × | / | ∧
<variable> ::= A | B | ... | Z
```

Expression (7)

- Exemple 1
  - infixé
    - A = B ∧ C > D
  - complètement parenthésé
    - ((A = (B ∧ C)) > D)
  - postfixé
    - A B C ∧ = D >
- Exemple 2
  - infixé
    - (A = B) ∧ (C = D)
  - complètement parenthésé
    - ((A = B) ∧ (C = D))
  - postfixé
    - A B = C D = ∧

Expression (8)

- ECP
  - écriture naturelle mais lourde
- EPOST et EPREF
  - algorithme d'évaluation de l'expression simple

Evaluation Expression

- Comment évaluer une expression?
  - $S(\alpha_1, ..., \alpha_p, a_1, ..., a_n)$, une expression
    - ✓ α_i sont les opérateurs
    - ✓ a_i sont les variables
  - à chaque a_i, on associe une valeur v_i
    - ✓ $E=(v_1, ..., v_n)$
      - *environnement* de l'expression
- L'évaluation consiste, pour un environnement donné, à calculer la valeur de l'expression pour les valeurs associées à chaque variable

Evaluation Expression (2)

- **<u>Réduction successive</u>**
  - on part des sous-expressions dont les opérandes sont des variables qu'on peut calculer par simple composition de valeurs
  - on est toujours amené à évaluer des expressions dont les opérandes sont
    - ✓ soit des variables
      - dont on connaît la valeur
    - ✓ soit des sous-expressions de niveau d'imbrication plus grand
      - dont l'évaluation a déjà été faite

Evaluation Expression (3)

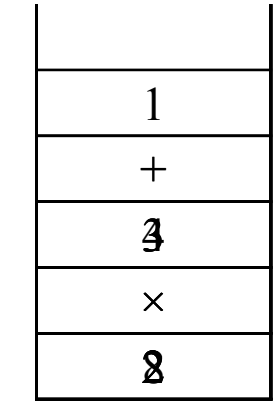
- Schéma d'évaluation générale pour un opérateur binaire β et deux opérandes op_1 et op_2

```
évaluation(S) =
  opération[β,
    Si variable(op1)
      Alors valeur(op1)
      Sinon évaluation(op1),
    Si variable(op2)
      Alors valeur(op2)
      Sinon évaluation(op2)]
```

- *valeur(a)* délivre la valeur *v* associée à la variable *a*
- *variable(a)* délivre *True* ssi *a* est une variable
- *opération(β, v_1, v_2)* évalue l'opération binaire dont *β* est l'opérateur et v_1, v_2 les opérandes

Evaluation Expression (4)

- Evaluer l'expression (A × (B + C)) avec A=2, B=3, C=1



(A × (B + C))