

Compléments de Programmation

Benoit Donnet
Année Académique 2024 - 2025



Agenda

- Chapitre 1: Raisonnement Mathématique
- Chapitre 2: Construction de Programme
- Chapitre 3: Introduction à la Complexité
- Chapitre 4: Récursivité
- Chapitre 5: Types Abstraits de Données
- Chapitre 6: Listes
- **Chapitre 7: Piles**
- Chapitre 8: Files
- Chapitre 9: Elimination de la Récursivité

Agenda

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

Agenda

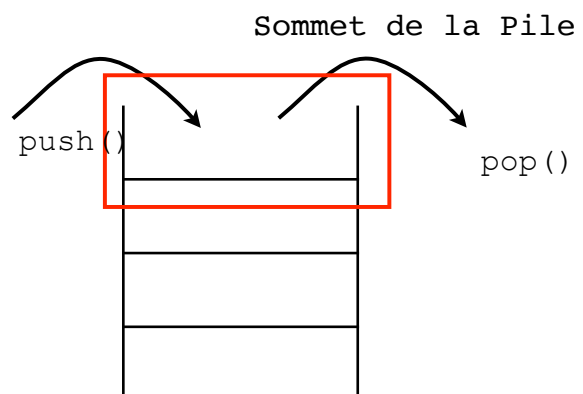
- Chapitre 7: Piles
 - Principe
 - Spécification Abstraite
 - Implémentation
 - Utilisation

Principe

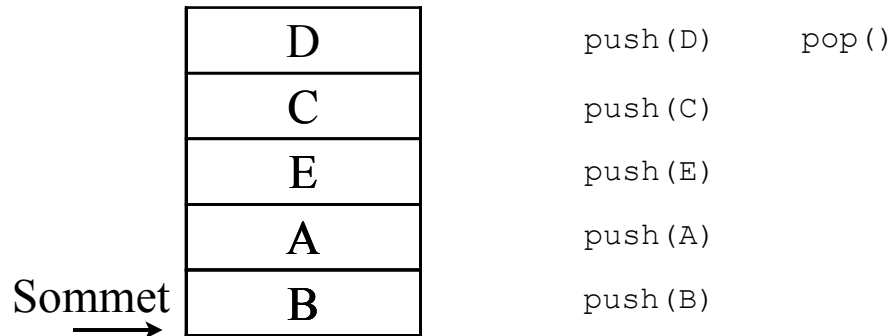
- TAD très utilisé en programmation
 - **stack**
- Notion intuitive
 - pile d'assiettes
 - pile de dossiers
 - ...
- *LIFO*
 - Last-In, First-Out

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)



Agenda

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

Spécification Abstraite

- Syntaxe

Type:

Stack

Utilise:

Boolean, Element, Natural

Opérations:

empty_stack: \rightarrow Stack

is_empty: Stack \rightarrow Boolean

size: Stack \rightarrow Natural

push: Stack \times Element \rightarrow Stack

pop: Stack \rightarrow Stack

top: Stack \rightarrow Element

Spécification Abstraite (2)

- Sémantique
 - préconditions

Préconditions:

$\forall s \in \text{Stack}$

$\forall s, \neg \text{is_empty}(s), \text{pop}(s)$

$\forall s, \neg \text{is_empty}(s), \text{top}(s)$

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:

$\forall s \in \text{Stack}, \forall e \in \text{Element}$
 $\text{is_empty}(\text{empty_stack}) = \text{True}$
 $\text{is_empty}(\text{push}(s, e)) = \text{False}$
 $\text{is_empty}(\text{pop}(s)) = \begin{cases} \text{True} & \text{if } \text{size}(s) = 1 \\ \text{False} & \text{otherwise} \end{cases}$

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

Spécification Abstraite (5)

Axiomes:

$\forall s \in \text{Stack}, \forall e \in \text{Element}$
 $\text{size}(\text{empty_stack}) = 0$
 $\text{size}(\text{push}(s, e)) = \text{size}(s) + 1$
 $\text{size}(\text{pop}(s)) = \text{size}(s) - 1$

| | | Opérations Internes | | |
|--------------|--------------------------|--------------------------|---------------------|----------------------|
| | | <code>empty_stack</code> | <code>pop(·)</code> | <code>push(·)</code> |
| Observateurs | <code>is_empty(·)</code> | ✓ | ✓ | ✓ |
| | <code>size(·)</code> | ✓ | ✓ | ✓ |
| | <code>top(·)</code> | ∅ | | |

Spécification Abstraite (6)

Axiomes:

$\forall s \in \text{Stack}, \forall e \in \text{Element}$
 $\text{top}(\text{push}(s, e)) = e$
 $\text{top}(\text{pop}(s)) = \text{top}(s)$
 $\text{pop}(\text{push}(s, e)) = s$

| | | Opérations Internes | | |
|--------------|--------------------------|--------------------------|---------------------|----------------------|
| | | <code>empty_stack</code> | <code>pop(·)</code> | <code>push(·)</code> |
| Observateurs | <code>is_empty(·)</code> | ✓ | ✓ | ✓ |
| | <code>size(·)</code> | ✓ | ✓ | ✓ |
| | <code>top(·)</code> | ∅ | ✓ | ✓ |

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);

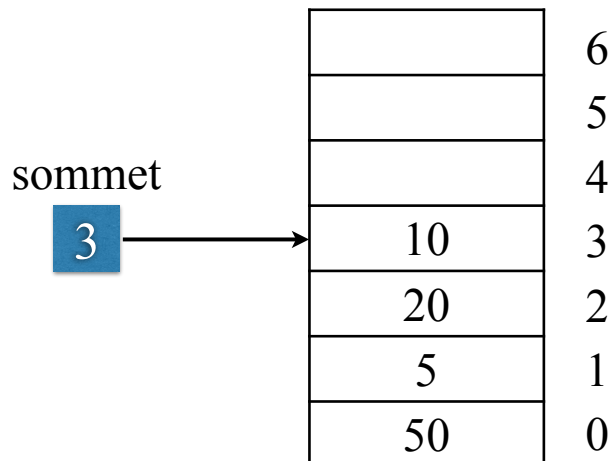
Stack *push(Stack *s, void *e);

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]); // éviter les fuites mémoires
    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()
```

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);

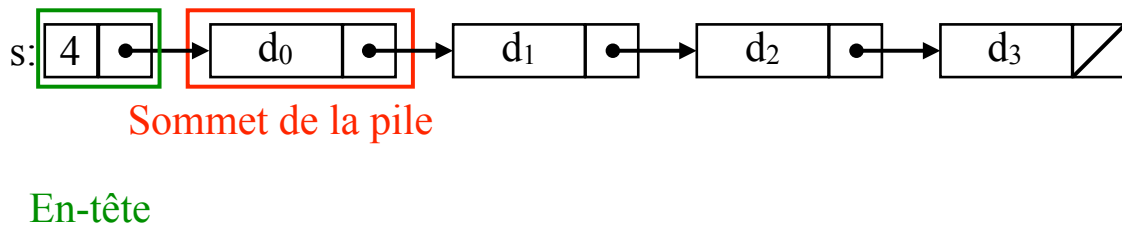
    printf("Stack size: %u\n", size(s));

    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 1^{ère} cellule
 - ✓ cfr. Chap. 6



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;
```

Liste

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

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->ptr->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

- **Expressions complètement parenthésées (ECP)?**
 1. une variable est une ECP
 2. soit x et y , des ECPs et β , opérateur binaire, $(x \beta y)$ est une ECP
 3. soit x , une ECP et α , opérateur unaire, (αx) est une ECP
- Variable?
 - lettre majuscule
- Opérateur binaire?
 - $\{+, -, \times, /\}$, $\{\leq, \geq, <, >, =, \neq\}$, et $\{\wedge, \vee\}$
- Opérateur unaire?
 - $\{\nabla, \triangle\}$, et $\{\neg\}$

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

| | |
|-----------------------------------|---|
| $\langle \text{ecp} \rangle$ | $::= (\langle \text{ecp} \rangle \langle \text{opbin} \rangle \langle \text{ecp} \rangle) \mid (\langle \text{opun} \rangle \langle \text{ecp} \rangle) \mid \langle \text{variable} \rangle$ |
| $\langle \text{opbin} \rangle$ | $::= + \mid - \mid \times \mid / \mid \leq \mid \geq \mid = \mid > \mid < \mid \neq \mid \wedge \mid \vee$ |
| $\langle \text{opun} \rangle$ | $::= \triangle \mid \nabla \mid \neg$ |
| $\langle \text{variable} \rangle$ | $::= A \mid B \mid \dots \mid Z$ |

Expression (3)

- Exemples d'ECPs conformes à la grammaire
 - $((A + B) \times C)$
 - $((A / B) = C) \wedge (E < F)$
 - (∇A)
- Exemples non conformes
 - (A)
 - $((A + B))$
 - $A \triangle B$

Expression (4)

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

$\langle \text{epref} \rangle ::= \langle \text{opbin} \rangle \langle \text{epref} \rangle \langle \text{epref} \rangle \mid \langle \text{opun} \rangle \langle \text{epref} \rangle \mid \langle \text{variable} \rangle$

- Exemples
 - $((A - B) + C) \Rightarrow + - A B C$
 - $((\neg(A < B)) \wedge C) \Rightarrow \wedge \neg < A B C$

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, $x y \beta$ est une EPOST
 3. soit x , une EPOST, et α , opérateur unaire, $x \alpha$ est une EPOST
- BNF

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

- Exemples
 - $((A - B) + C) \Rightarrow A B - C +$
 - $((\neg(A < B)) \wedge C) \Rightarrow A B < \neg C \wedge$

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
 - ✓ $A = B \wedge C > D$
 - complètement parenthésé
 - ✓ $((A = (B \wedge C)) > D)$
 - postfixé
 - ✓ $A B C \wedge = D >$
- Exemple 2
 - infixé
 - ✓ $(A = B) \wedge (C = D)$
 - complètement parenthésé
 - ✓ $((A = B) \wedge (C = D))$
 - postfixé
 - ✓ $A B = C D = \wedge$

Expression (8)

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

Evaluation Expression

- Comment évaluer une expression?
 - $S(\alpha_1, \dots, \alpha_p, a_1, \dots, a_n)$, une expression
 - ✓ α_i sont les opérateurs
 - ✓ a_i sont les variables
 - à chaque a_i , on associe une valeur v_i
 - ✓ $E=(v_1, \dots, v_n)$
 - *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)

- **Réduction successive**
 - 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[ $\beta$ ,  
    Si variable( $op_1$ )  
      Alors valeur( $op_1$ )  
      Sinon évaluation( $op_1$ ),  
    Si variable( $op_2$ )  
      Alors valeur( $op_2$ )  
      Sinon évaluation( $op_2$ )]
```

- *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 $(A \times (B + C))$ avec $A=2$, $B=3$, $C=1$

