

Calculatrice, piles et listes

Algorithmique et structures de données

Pierre Künzli

Adapté des cours de Paul Albuquerque et Orestis Malaspinas

2 + 3 = 2 3 +,

2 et 3 sont les *opérandes*, + l'*opérateur*.

La notation infixe

$$2 * (3 + 2) - 4 = 6.$$

La notation postfixe

2 3 2 + * 4 - = 6.

Exercice : écrire $2 * 3 * 4 + 2$ en notation postfixe

Exercice : écrire $2 * 3 * 4 + 2$ en notation postfixe

$2\ 3\ 4\ *\ *\ 2\ +\ =\ (2\ *\ (3\ *\ 4))\ +\ 2.$

(Rappel) Évaluation d'expression postfixe : algorithme

- Chaque *opérateur* porte sur les deux opérandes qui le précèdent.
- Le *résultat d'une opération* est un nouvel *opérande* qui est remis au sommet de la pile.

Exemple

2 3 4 + * 5 - = ?

- On parcourt de gauche à droite :

Caractère lu	Pile opérandes
2	2
3	2, 3
4	2, 3, 4
+	2, (3 + 4)
*	2 * 7
5	14, 5
-	14 - 5 = 9

Évaluation d'expression postfixe : algorithme

1. La valeur d'un opérande est *toujours* empilée.
2. L'opérateur s'applique *toujours* au 2 opérandes au sommet.
3. Le résultat est remis au sommet.

Exemple d'algorithme d'évaluation postixe

```
bool evaluate(char *postfix, double *val) { // init stack
    for (size_t i = 0; i < strlen(postfix); ++i) {
        if (is_operand(postfix[i])) {
            stack_push(&s, postfix[i]);
        } else if (is_operator(postfix[i])) {
            double rhs = stack_pop(&s);
            double lhs = stack_pop(&s);
            stack_push(&s, op(postfix[i], lhs, rhs));
        }
    }
    return stack_pop(&s);
}
```

De infixe à post-fixe

- Une *pile* est utilisée pour stocker *opérateurs* et *parenthèses*.
- Les opérateurs ont des *priorités* différentes.

```
^      : priorité 3
* /    : priorité 2
+ -    : priorité 1
( )    : priorité 0 // pas un opérateur mais bon
```

De infixe à post-fixe : algorithme

- On lit l'expression infixe de gauche à droite.
- On examine le prochain caractère de l'expression infixe.
 - Si opérande, le placer dans l'expression postfixe.
 - Si parenthèse ouvrante, le mettre dans la pile (priorité 0).
 - Si opérateur, comparer sa priorité avec celui du sommet de la pile :
 - Si sa priorité est plus élevée, empiler.
 - Sinon dépiler l'opérateur de la pile dans l'expression postfixe et recommencer jusqu'à apparition d'un opérateur de priorité plus faible au sommet de la pile (ou pile vide).
 - Si parenthèse fermante, dépiler les opérateurs du sommet de la pile et les placer dans l'expression postfixe, jusqu'à ce qu'une parenthèse ouvrante apparaisse au sommet, dépiler également la parenthèse.
 - Si il n'y a plus de caractère dans l'expression infixe, dépiler tous les opérateurs de la pile dans l'expressions postfixe.

De infixe à post-fixe : exemple

Infixe	Postfixe	Pile	Priorité
$((A*B)/D-F)/(G+H)$	Vide	Vide	Néant
$(A*B)/D-F)/(G+H)$	Vide	(0
$A*B)/D-F)/(G+H)$	Vide	((0
$*B)/D-F)/(G+H)$	A	((0
$B)/D-F)/(G+H)$	A	((*	2
$)/D-F)/(G+H)$	AB	((*	2
$/D-F)/(G+H)$	AB*	(0
$D-F)/(G+H)$	AB*	(/	2
$-F)/(G+H)$	AB*D	(/	2
$F)/(G+H)$	AB*D/	(-	1
$)/(G+H)$	AB*D/F	(-	1
$/(G+H)$	AB*D/F-	Vide	Néant

De infixe à post-fixe : exemple

Infixe	Postfixe	Pile	Priorité
$((A*B)/D-F)/(G+H)$	Vide	Vide	Néant

$/ (G+H)$	$AB*D/F-$	Vide	Néant
$(G+H)$	$AB*D/F-$	$/$	2
$G+H)$	$AB*D/F-$	$/($	0
$+H)$	$AB*D/F-G$	$/($	0
$H)$	$AB*D/F-G$	$/(+$	1
$)$	$AB*D/F-GH$	$/(+$	1
Vide	$AB*D/F-GH+$	$/$	2
Vide	$AB*D/F-GH+ /$	Vide	Néant

Exemple de code de conversion infixé vers postfixé

```
char *infix_to_postfix(char* infix) {  
    // init and alloc stack and postfix  
    for (size_t i = 0; i < strlen(infix); ++i) {  
        if (is_operand(infix[i])) {  
            // we just add operands in the new postfix string  
        } else if (infix[i] == '(') {  
            // we push opening parenthesis into the stack  
            stack_push(&s, infix[i]);  
        } else if (infix[i] == ')') {  
            // we pop everything into the postfix  
        } else if (is_operator(infix[i])) {  
            // this is an operator. We add it to the postfix based  
            // on the priority of what is already in the stack and push it  
        }  
    }  
    // pop all the operators from the s at the end of postfix  
    // and end the postfix with '\0'  
    return postfix;  
}
```

La liste chaînée et pile

- Jusqu'à présent on a vu deux façons de faire une pile : en utilisant un espace mémoire défini à la compilation (pile statique) ou à l'exécution (pile dynamique).
- Il existe une troisième façon de faire : utiliser une **liste chaînée**.

Structure de données

- Chaque élément de la liste contient :
 1. une valeur,
 2. un pointeur vers le prochain élément.
- La pile est un pointeur vers le premier élément.

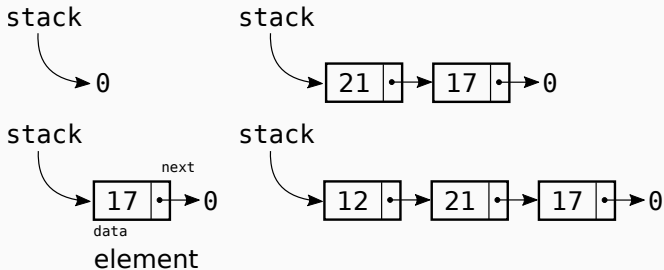


Fig. 1 : Un exemple de liste chaînée.

Une pile-liste-chaînée

```
typedef struct _element {  
    int data;  
    struct _element *next;  
} element;  
typedef element* stack;
```

Fonctionnalités ?

```
void stack_create(stack *s);  
void stack_destroy(stack *s);  
void stack_push(stack *s, int val);  
void stack_pop(stack *s, int *val);  
void stack_peek(stack s, int *val);  
bool stack_is_empty(stack s);
```

Création et test si vide

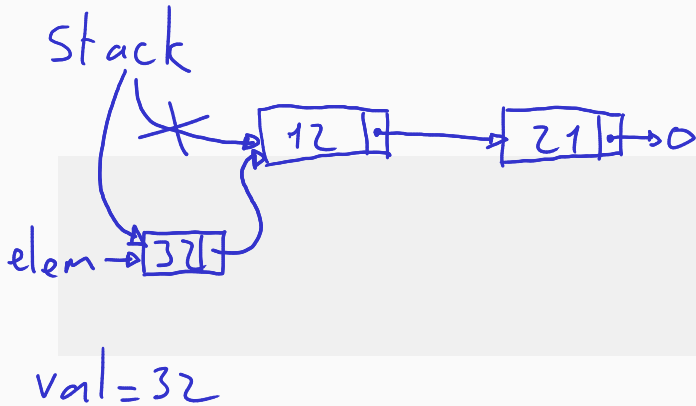
```
void stack_create(stack *s) {  
    *s = NULL;  
}  
  
bool stack_is_empty(stack s){  
    return s == NULL;  
}
```

Empiler ?

Empiler (le code)

```
void stack_push(stack *s, int val) {  
    element *elem = malloc(sizeof(*elem));  
    elem->data = val;  
    elem->next = *s;  
    *s = elem;  
}
```

Empiler (faire un dessin)



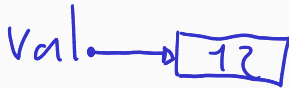
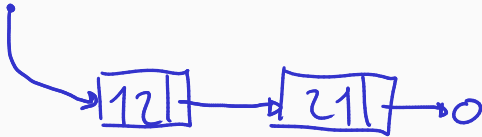
Jeter un oeil ?

Jeter un oeil (le code)

```
void stack_peek(stack s, int *val) {  
    *val = s->data;  
}
```

Jeter un oeil (faire un dessin)

Stack

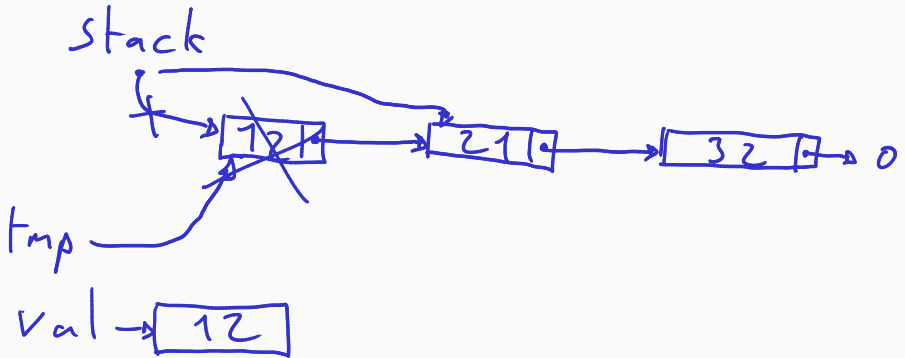


Dépiler ?

Dépiler (le code)

```
void stack_pop(stack *s, int *val) {  
    stack_peek(*s, val);  
    element *tmp = *s;  
    *s = (*s)->next;  
    free(tmp);  
}
```

Dépiler (faire un dessin)



Détruire ?

Détruire (le code)

```
void stack_destroy(stack *s) {  
    while (!stack_is_empty(*s)) {  
        int val;  
        stack_pop(s, &val)  
    }  
}
```

Détruire (faire un dessin)

