Applications des piles, listes chaînées et files d'attente

Algorithmique et structures de données, 2024-2025

P. Albuquerque (B410) et O. Malaspinas (A401), ISC, HEPIA

2024-12-09

En partie inspirés des supports de cours de P. Albuquerque

Rappel: les piles

Qu'est-ce donc?

Rappel: les piles

Qu'est-ce donc?

- Structure de données abstraite de type LIFO

Quelles fonctionnalités?

Rappel: les piles

Qu'est-ce donc?

- Structure de données abstraite de type LIFO

Quelles fonctionnalités?

1. Empiler (push): ajouter un élément sur la pile.
2. Dépiler (pop): retirer l'élément du sommet de la pile et le retourner.
3. Pile vide? (is_empty?).

Le tri à deux piles

Le tri à deux piles (1/3)

Cas pratique

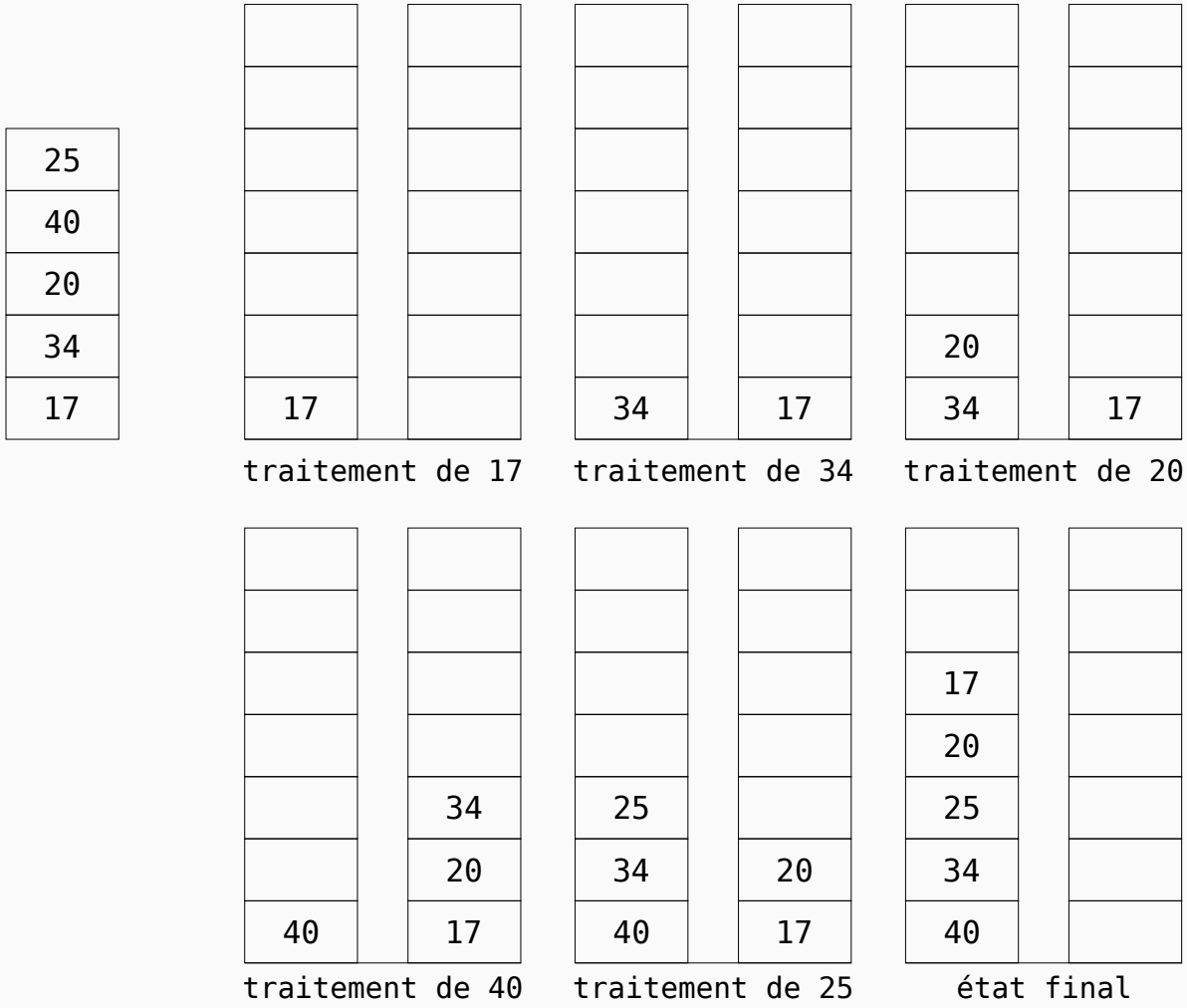

Figure 1: Un exemple de tri à deux piles

Le tri à deux piles (2/3)

Exercice: formaliser l'algorithme

Le tri à deux piles (2/3)

Exercice: formaliser l'algorithme

Algorithme de tri nécessitant 2 piles (G, D)

Soit tab le tableau à trier:

```
pour i de 0 à N-1
    tant que (tab[i] > que le sommet de G)
        dépiler G dans D
    tant que (tab[i] < que le sommet de D)
        dépiler de D dans G
    empiler tab[i] sur G
dépiler tout D dans G
dépiler tout G dans tab
```

Le tri à deux piles (3/3)

Exercice: trier le tableau `[2, 10, 5, 20, 15]`

La Calculatrice

La calculatrice (1/8)

Vocabulaire

```
2 + 3 = 2 3 +,
```

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

La calculatrice (1/8)

Vocabulaire

```
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`

La calculatrice (1/8)

Vocabulaire

```
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`

```
2 3 4 * * 2 + = (2 * (3 * 4)) + 2.
```

La calculatrice (2/8)

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
```

La calculatrice (3/8)

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 du résultat.
  - Si parenthèse 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 du résultat et recommencer jusqu'à apparition d'un opérateur de priorité plus faible au sommet de la pile (ou pile vide).
  - Si parenthèse fermée, dépiler les opérateurs du sommet de la pile et les placer dans l'expression du résultat, jusqu'à ce qu'une parenthèse ouverte apparaisse au sommet, dépiler également la parenthèse.
  - Si il n'y a pas de caractère dans l'expression dépiler tous les opérateurs dans le résultat.

La calculatrice (4/8)

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

La calculatrice (5/8)

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

La calculatrice (6/8)

Exercice: écrire le code et le poster sur matrix

- Quelle est la signature de la fonction?

La calculatrice (6/8)

Exercice: écrire le code et le poster sur matrix

- Quelle est la signature de la fonction?

- Une sorte de corrigé:

```
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
        } 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 calculatrice (7/8)

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 parcours 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
```

La calculatrice (8/8)

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.

Exercice: écrire l'algorithme en C (et poster sur matrix)

La calculatrice (8/8)

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.

Exercice: écrire l'algorithme en C (et poster sur matrix)

```c
double evaluate(char* postfix) {
    // declare and initialize stack s
    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);
```

Liste chaînée et pile

La liste chaînée et pile (1/6)

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.

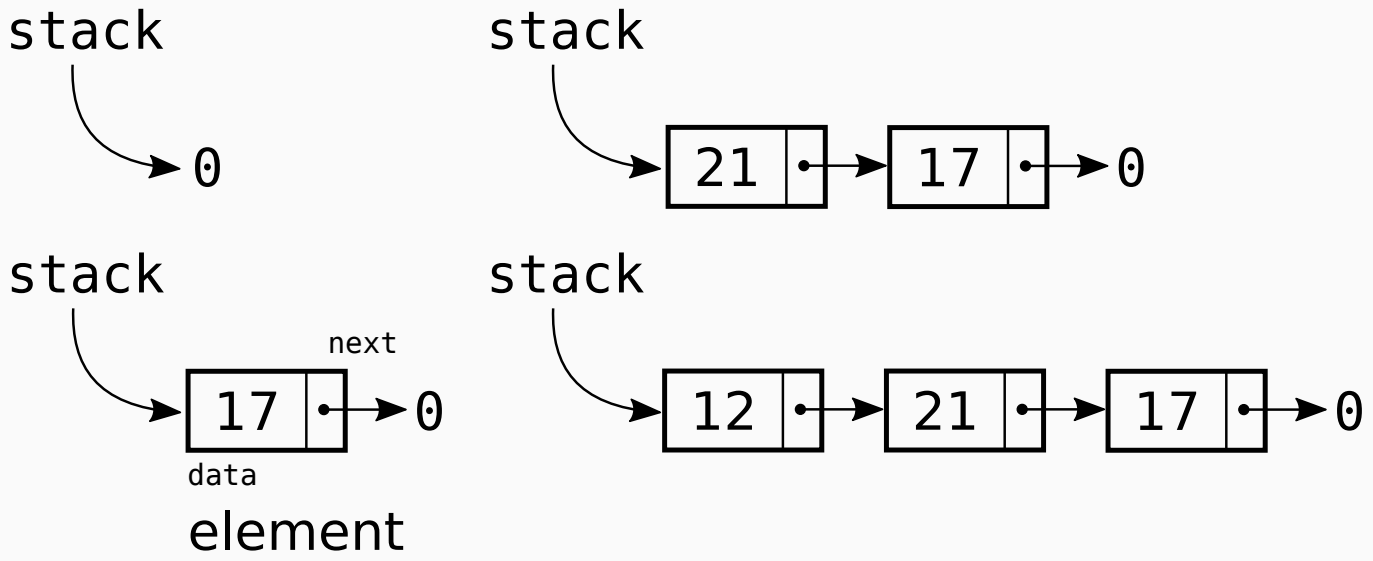

Figure 2: Un exemple de liste chaînée.

La liste chaînée et pile (2/6)

Une pile-liste-chaînée

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

Fonctionnalités?

La liste chaînée et pile (2/6)

Une pile-liste-chaînée

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

Fonctionnalités?

```
void stack_create(stack *s); // s->top = NULL;
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); // return NULL == s.top;
```

La liste chaînée et pile (3/6)

Empiler? (faire un dessin)

La liste chaînée et pile (3/6)

Empiler? (faire un dessin)

Empiler? (le code ensemble)

La liste chaînée et pile (3/6)

Empiler? (faire un dessin)

Empiler? (le code ensemble)

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

La liste chaînée et pile (4/6)

Jeter un oeil? (faire un dessin)

La liste chaînée et pile (4/6)

Jeter un oeil? (faire un dessin)

Jeter un oeil? (le code ensemble)

La liste chaînée et pile (4/6)

Jeter un oeil? (faire un dessin)

Jeter un oeil? (le code ensemble)

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

La liste chaînée et pile (5/6)

Dépiler? (faire un dessin)

La liste chaînée et pile (5/6)

Dépiler? (faire un dessin)

Dépiler? (le code ensemble)

La liste chaînée et pile (5/6)

Dépiler? (faire un dessin)

Dépiler? (le code ensemble)

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

La liste chaînée et pile (6/6)

Détruire? (faire un dessin)

La liste chaînée et pile (6/6)

Détruire? (faire un dessin)

Détruire? (le code ensemble)

La liste chaînée et pile (6/6)

Détruire? (faire un dessin)

Détruire? (le code ensemble)

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