Les piles

Objectifs

— Connaître la structure de pile et savoir s'en servir

Introduction

Tableaux, entiers sont des structures **statiques** : leur taille et leur emplacement en mémoire sont fixés et ne changent pas au cours de l'utilisation.

Nous allons voir deux types de structures abstraites qui sont **dynamiques**.

La première est la **pile (stack)** qui fonctionne sous le principe **LIFO (Last In First Out)** : le dernier élément entré est le premier à sortir.

C'est ce genre de structure qui permet d'utiliser Ctrl+Z pour annuler une action, ou lorsque l'on utilise la touche retour du navigateur.

Structure d'une pile

Principe

Définition

Une pile `P` est une structure de données qui ne propose que les 4 opérations suivantes :
- `Lire(P)` : renvoie la valeur de l'élément au sommet de la pile
- `EstVide(P)` : renvoie `VRAI` si la pile est vide, `FAUX` sinon
- `Empiler(P, x)` : ajoute l'élément `x` au sommet de la pile
- `Dépiler(P)` : supprime l'élément au sommet de la pile et renvoie sa valeur

En théorie, chacune de ces opérations s'exécute en temps constant : $O(1)$.

Exemple d'évolution d'une pile

Considérons la séquence d'instructions suivante sur une pile initialement vide :

1. `Empiler(P, 1)` → Pile : `[1]`
2. `Empiler(P, 3)` → Pile : `[1, 3]` (3 est au sommet)
3. `Dépiler(P)` → Renvoie 3, Pile : `[1]`
4. `Dépiler(P)` → Renvoie 1, Pile : `[]` (vide)

Représentation verticale (convention usuelle) :

```
Empiler(P,1)    Empiler(P,3)    Dépiler(P)     Dépiler(P)

     []        →    [1]       →   [3]     →    [1]      →     []
                                  [1]
```

Le sommet de la pile est toujours en haut. Le principe LIFO signifie que le dernier élément empilé (3) est le premier à être dépilé.

Implémentation

On peut implémenter une pile contenant au plus n éléments à l'aide d'un tableau de taille n :

— Le tableau `P[1..n]` contient les éléments empilés
— Une variable globale `sommet` indique l'indice de l'élément situé au sommet de la pile
— La pile correspond alors aux éléments `P[1:sommet]`

Les procédures suivantes permettent de manipuler cette pile.

EstVide

```
ALGORITHME EstVide(P):
DONNEES
  P: tableau de taille n
DEBUT
  SI sommet   0 ALORS
    RENVOYER VRAI
  SINON
    RENVOYER FAUX
  FSI
FIN
```

Lire

```
ALGORITHME Lire(P):
DONNEES
  P: tableau de taille n
DEBUT
  RENVOYER P[sommet]
FIN
```

Empiler

```
ALGORITHME Empiler(P, x):
DONNEES
  P: tableau de taille n
  x: élément à empiler
DEBUT
  SI sommet   n ALORS
    AFFICHER "débordement"
    RENVOYER ERREUR
  SINON
    sommet ← sommet + 1
    P[sommet] ← x
  FSI
FIN
```

Dépiler

```
ALGORITHME Depiler(P):
DONNEES
  P: tableau de taille n
DEBUT
  SI EstVide(P) = VRAI ALORS
    RENVOYER ERREUR
  SINON
    sommet ← sommet - 1
    RENVOYER P[sommet+1]
  FSI
FIN
```

! Remarques sur l'implémentation

- La variable `sommet` est **globale** et partagée entre toutes les opérations
- On doit vérifier les conditions de débordement (pile pleine) et de sous-dépassement (pile vide)
- Chaque opération s'effectue en temps $O(1)$
- Cette implémentation utilise un tableau de taille fixe, donc la pile a une capacité maximale

Utilisations possibles

Parenthésage

i Spécification du problème

Problème	Parenthésage
Entrée	Une chaîne de caractères de longueur n
Sortie	`VRAI` si l'expression est bien parenthésée, `FAUX` sinon

i Exemples

- `"(a+5*[3+2])"` est bien parenthésée
- `"(a+5*[3+2)]"` n'est **pas** bien parenthésée (mélange de types)

Dans le cas où seules les parenthèses sont prises en compte, il est possible de s'en sortir avec seulement une variable chargée de compter le nombre de parenthèses qui n'ont pas encore été fermées. Dès que l'on prend en compte les crochets, cette approche devient plus laborieuse alors que l'utilisation d'une pile permet une résolution élégante.

Principe :

On lit l'expression caractère par caractère. Lorsqu'on rencontre une parenthèse (ou crochet) ouvrante, on l'empile. Si c'est une parenthèse (ou crochet) fermante, on dépile et on vérifie que ce qui a été dépilé corresponde bien.

À la fin, la pile doit être vide.

Pseudo-code

```
ALGORITHME Parenthesage(C):
```

```
DONNEES
  C : chaîne de longueur n
VARIABLES:
  i : entier
  P : pile
DEBUT
  i ← 1
  TQ i   n FAIRE
    SI C[i]='(' OU C[i]='[' ALORS
      Empiler(P, C[i])
    SINON SI C[i]=')' ALORS
      SI Depiler(P)   '(' ALORS
        RENVOYER FAUX
      FSI
    SINON SI C[i]=']' ALORS
      SI Depiler(P)   '[' ALORS
        RENVOYER FAUX
      FSI
    FSI
    i ← i+1
  FTQ
  RENVOYER EstVide(P)
FIN
```

Complexité

- **Meilleur cas :** $\Theta(1)$ (erreur détectée dès le début)
- **Pire cas :** $\Theta(n)$ (parcours complet de la chaîne)
- **Complexité moyenne :** $O(n)$

i Exemple de déroulement

Pour `"(a+[3*2])"` :

1. Lire `(` → Empiler `(` → Pile : `[(`
2. Lire `a` → Rien
3. Lire `+` → Rien
4. Lire `[` → Empiler `[` → Pile : `[(, [`
5. Lire `3` → Rien
6. Lire `*` → Rien
7. Lire `2` → Rien
8. Lire `]` → Dépiler `[` (correspond) → Pile : `[(`
9. Lire `)` → Dépiler `(` (correspond) → Pile : `[]`

10. Fin : pile vide → **VRAI**

Notation polonaise inversée (expression postfixe)

i Spécification du problème

Problème	Calcul en notation polonaise inversée (NPI)
Entrée	Une expression arithmétique en NPI
Sortie	Le résultat du calcul

i Exemple

L'expression infixe `"(3 + 4) * 5"` s'écrit en notation polonaise inversée `"3 4 + 5 *"`. Le résultat du calcul reste 35.

La **notation polonaise inversée (NPI)**, ou notation postfixe, place les opérateurs immédiatement après leurs opérandes, supprimant ainsi toute ambiguïté liée aux parenthèses ou à la priorité des opérations. Cette forme est particulièrement adaptée à l'évaluation des expressions via une pile.

Principe :

On lit l'expression de gauche à droite :

— Lorsqu'on rencontre un **nombre**, on l'empile
— Lorsqu'on rencontre un **opérateur**, on dépile les deux nombres, on fait le calcul et on empile le résultat

À la fin, la pile ne contient que le résultat final.

On supposera pour l'algorithme que l'on fournit l'expression en postfixe sous forme de tableau ne comportant que des nombres ou des opérateurs.

On dispose de plus des fonctions `EstOperateur(e)` qui renvoie `VRAI` si l'argument est un opérateur (+, *, -, /) et `Operation(a, b, op)` qui renvoie le résultat de l'opération `a op b`.

Pseudo-code

```
ALGORITHME EvaluerNPI(E):
DONNEES
  E : tableau de longueur n
```

```
VARIABLES
  P : pile pour les opérandes
  i : entier
  a, b, res : nombre
DEBUT
  i ← 1
  TQ i   n FAIRE
    SI EstOperateur(E[i]) = FAUX ALORS
      Empiler(P, E[i])
    SINON
      b ← Depiler(P)
      a ← Depiler(P)
      res ← Operation(a, b, E[i])
      Empiler(P, res)
    FSI
    i ← i + 1
  FTQ
  RENVOYER Depiler(P)
FIN
```

Complexité

— **Complexité :** $\Theta(n)$ (parcours linéaire du tableau)

i Exemple de déroulement

Pour l'expression `"3 4 + 5 *"` représentée par le tableau `[3, 4, +, 5, *]` :

1. Lire `3` → Empiler 3 → Pile : `[3]`
2. Lire `4` → Empiler 4 → Pile : `[3, 4]`
3. Lire `+` → Dépiler 4 et 3, calculer 3+4=7, empiler 7 → Pile : `[7]`
4. Lire `5` → Empiler 5 → Pile : `[7, 5]`
5. Lire `*` → Dépiler 5 et 7, calculer 7×5=35, empiler 35 → Pile : `[35]`
6. Fin : dépiler et renvoyer **35**

Applications et avantages des piles

Les piles sont utilisées dans de nombreux domaines :

— **Gestion des appels de fonctions** : la pile d'appels (call stack) stocke les contextes d'exécution
— **Parcours de graphes** : algorithmes de parcours en profondeur (DFS)
— **Évaluation d'expressions** : calculs arithmétiques, compilation

— **Navigation** : historique du navigateur, undo/redo
— **Algorithmes récursifs** : simulation de la récursivité de manière itérative

! Points clés

— Structure **LIFO** (Last In First Out)
— Toutes les opérations en temps **constant** $O(1)$
— Implémentation simple avec un tableau et un indice
— Utile pour de nombreux algorithmes fondamentaux
— Limitation : capacité maximale si implémentée avec un tableau statique