

Module algorithmie et programmation

Piles

Pierre Künzli

Adapté des cours de Paul Albuquerque et Orestis Malaspinas

Les piles

Qu'est-ce donc?

- Structure de données abstraite...

Les piles

Qu'est-ce donc?

- Structure de données abstraite...
- de type LIFO (*Last in first out*).

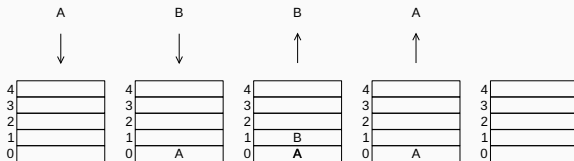


Figure 1: Une pile où on ajoute A, puis B avant de les retirer. Source: [Wikipedia](#)

Des exemples de la vraie vie

Les piles

Qu'est-ce donc?

- Structure de données abstraite...
- de type LIFO (*Last in first out*).

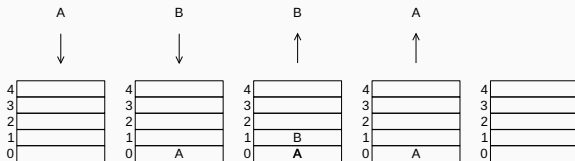


Figure 1: Une pile où on ajoute A, puis B avant de les retirer. Source: [Wikipedia](#)

Des exemples de la vraie vie

- Pile d'assiettes, de livres, ...
- Adresses visitées par un navigateur web.
- Les calculatrices du passé (en polonaise inverse).
- Les boutons *undo* de vos éditeurs de texte (aka *u* dans vim).

Fonctionnalités

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. Liste vide? (is_empty?).
4. Jeter un oeil (peek): retourner l'élément du sommet de la pile (sans le dépiler).
5. Nombre d'éléments (length).

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. Liste vide? (is_empty?).
4. Jeter un oeil (peek): retourner l'élément du sommet de la pile (sans le dépiler).
5. Nombre d'éléments (length).

Existe en deux goûts

- Pile avec ou sans limite de capacité (à concurrence de la taille de la mémoire).

Implémentation

- Jusqu'ici on n'a pas du tout parlé d'implémentation (d'où le nom de structure abstraite).
- Pas de choix unique d'implémentation.

Quelle structure de données allons nous utiliser?

Implémentation

- Jusqu'ici on n'a pas du tout parlé d'implémentation (d'où le nom de structure abstraite).
- Pas de choix unique d'implémentation.

Quelle structure de données allons nous utiliser?

Un tableau!

Implémentation

- Jusqu'ici on n'a pas du tout parlé d'implémentation (d'où le nom de structure abstraite).
- Pas de choix unique d'implémentation.

Quelle structure de données allons nous utiliser?

Un tableau!

La structure: de quoi avons-nous besoin (pile de taille fixe)?

Les piles

Implémentation

- Jusqu'ici on n'a pas du tout parlé d'implémentation (d'où le nom de structure abstraite).
- Pas de choix unique d'implémentation.

Quelle structure de données allons nous utiliser?

Un tableau!

La structure: de quoi avons-nous besoin (pile de taille fixe)?

```
#define MAX_CAPACITY 500
typedef struct _stack {
    int data[MAX_CAPACITY]; // les données
    int top;                // indice du sommet
} stack;
```

Initialisation

Les piles

Initialisation

```
void stack_init(stack *s) {  
    s->top = -1;  
}
```

Est vide?

Les piles

Initialisation

```
void stack_init(stack *s) {  
    s->top = -1;  
}
```

Est vide?

```
bool stack_is_empty(stack s) {  
    return s.top == -1;  
}
```

Empiler (ajouter un élément au sommet)

Les piles

Initialisation

```
void stack_init(stack *s) {  
    s->top = -1;  
}
```

Est vide?

```
bool stack_is_empty(stack s) {  
    return s.top == -1;  
}
```

Empiler (ajouter un élément au sommet)

```
void stack_push(stack *s, int val) {  
    s->top += 1;  
    s->data[s->top] = val;  
}
```

Les piles

Dépiler (enlever l'élément du sommet)

Les piles

Dépiler (enlever l'élément du sommet)

```
int stack_pop(stack *s) {  
    s->top -= 1;  
    return s->data[s->top+1];  
}
```

Jeter un oeil (regarder le sommet)

Les piles

Dépiler (enlever l'élément du sommet)

```
int stack_pop(stack *s) {  
    s->top -= 1;  
    return s->data[s->top+1];  
}
```

Jeter un oeil (regarder le sommet)

```
int stack_peek(stack *s) {  
    return s->data[s->top];  
}
```

Quelle est la complexité de ces opérations?

Les piles

Dépiler (enlever l'élément du sommet)

```
int stack_pop(stack *s) {  
    s->top -= 1;  
    return s->data[s->top+1];  
}
```

Jeter un oeil (regarder le sommet)

```
int stack_peek(stack *s) {  
    return s->data[s->top];  
}
```

Quelle est la complexité de ces opérations?

$\mathcal{O}(1)$

Voyez-vous des problèmes potentiels avec cette implémentation?

Les piles

Dépiler (enlever l'élément du sommet)

```
int stack_pop(stack *s) {  
    s->top -= 1;  
    return s->data[s->top+1];  
}
```

Jeter un oeil (regarder le sommet)

```
int stack_peek(stack *s) {  
    return s->data[s->top];  
}
```

Quelle est la complexité de ces opérations?

$\mathcal{O}(1)$

Voyez-vous des problèmes potentiels avec cette implémentation?

- Empiler avec une pile pleine.
- Dépiler avec une pile vide.
- Jeter un oeil au sommet d'une pile vide.

Gestion d'erreur, level 0

- Il y a plusieurs façon de traiter les erreur:
 - Ne rien faire (laisser la responsabilité à l'utilisateur).
 - Utiliser des codes d'erreurs.
 - Faire paniquer le programme (il plante plus ou moins violemment).

Les codes d'erreur

- En C, il n'y a pas de mécanisme de traitement des erreurs.
- Pour signaler à l'utilisateur d'une fonction que quelque chose s'est mal passé, on peut retourner un code d'erreur.
- Par exemple, 0 pour pas d'erreur, autre chose pour une erreur.
- Cela permet de différencier les types d'erreur.

Exemple

```
int stack_push(stack *s, int val) {  
    if(s->top == MAX_CAPACITY-1) return 1;  
    s->top += 1;  
    s->data[s->top] = val;  
    return 0;  
}
```

Assertions

```
#include <assert.h>
void assert(int expression);
```

Qu'est-ce donc?

- Macro permettant de tester une condition lors de l'exécution d'un programme:
 - Si `expression == 0` (condition fausse), `assert()` affiche un message d'erreur sur `stderr` et termine l'exécution du programme.
 - Sinon l'exécution se poursuit normalement.
 - Peuvent être désactivés à la compilation avec `-DNDEBUG` (équivalent à `#define NDEBUG`)

À quoi ça sert?

- Permet de réaliser des tests unitaires.
- Permet de tester des conditions catastrophiques d'un programme.
- **Ne permet pas** de gérer les erreurs.

Assertions

Exemple

On peut combiner assert et code d'erreur

```
#include <assert.h>
int stack_push(stack *s, int val) {
    if(s->top == MAX_CAPACITY-1) return 1;
    assert(s->top < MAX_CAPACITY-1);
    s->top += 1;
    s->data[s->top] = val;
    return 0;
}
```

Cas typiques d'utilisation

- Vérification de la validité des pointeurs (typiquement `!= NULL`).
- Vérification du domaine des indices (dépassement de tableau).

Bug vs. erreur de *runtime*

- Les assertions sont là pour détecter les bugs (erreurs d'implémentation).
- Les assertions ne sont pas là pour gérer les problèmes externes au programme (allocation mémoire qui échoue, mauvais paramètre d'entrée passé par l'utilisateur, ...). On utilise des codes d'erreur pour ça.
- Typiquement désactivées dans le code de production.

La pile dynamique

Comment modifier le code précédent pour avoir une taille dynamique?

La pile dynamique

Comment modifier le code précédent pour avoir une taille dynamique?

```
// alloue une zone mémoire de size octets
```

```
void *malloc(size_t size);
```

```
// change la taille allouée à size octets (contiguïté garantie)
```

```
void *realloc(void *ptr, size_t size);
```

```
// libère une zone mémoire précédemment allouée
```

```
void free(void *ptr);
```

Et maintenant?

La pile dynamique

Et maintenant?

// crée une pile avec une taille par défaut

```
stack_create();
```

// vérifie si la pile est pleine et réalloue si besoin

```
stack_push();
```

// vérifie si la pile est vide/trop grande

// et réalloue si besoin

```
stack_pop();
```

// libère la mémoire d'une pile

```
stack_destroy();
```

La pile dynamique

La structure (pile de taille variable)

```
#define MIN_CAPACITY 50
typedef struct _stack {
    int* data;    // les données
    int capacity; // la capacité actuelle
    int top;      // indice du sommet
} stack;
```

La pile dynamique

Créer une pile

La pile dynamique

Créer une pile

```
int stack_create(stack *s){  
    s->data = malloc(MIN_CAPACITY*sizeof(int));  
    if(s->data == NULL) return 1;  
    s->capacity = MIN_CAPACITY;  
    s->top = -1;  
    return 0;  
}
```

Empiler (ajouter un élément au sommet)

La pile dynamique

Créer une pile

```
int stack_create(stack *s){
    s->data = malloc(MIN_CAPACITY*sizeof(int));
    if(s->data == NULL) return 1;
    s->capacity = MIN_CAPACITY;
    s->top = -1;
    return 0;
}
```

Empiler (ajouter un élément au sommet)

```
int stack_push(stack *s, int val){
    if(s->top+1 == s->capacity){
        int* new_ptr = realloc(s->data, 2*s->capacity*sizeof(int));
        if(new_ptr == NULL) return 1;
        s->data = new_ptr;
        s->capacity *= 2;;
    }
    s->top += 1;
    s->data[s->top] = val;
    return 0;
}
```


Dépiler (enlever un élément du sommet)

La pile dynamique

Dépiler (enlever un élément du sommet)

```
int stack_pop(stack *s, int *res){
    if(s->top < 0) return 1;
    *res = s->data[s->top-1];
    s->top -= 1;
    if(s->top < s->capacity/2 && s->capacity>MIN_CAPACITY){
        int* new_ptr = realloc(s->data, (s->capacity/2)*sizeof(int));
        if(new_ptr == NULL) return 1;
        s->data = new_ptr;
        s->capacity /= 2;;
    }
    return 0;
}
```

Détruire une pile

La pile dynamique

Détruire une pile

```
void stack_destroy(stack *s){  
    free(s->data);  
    s->data = NULL;  
    s->capacity = 0;  
    s->top = -1;  
}
```

Le tri à deux piles

Cas pratique

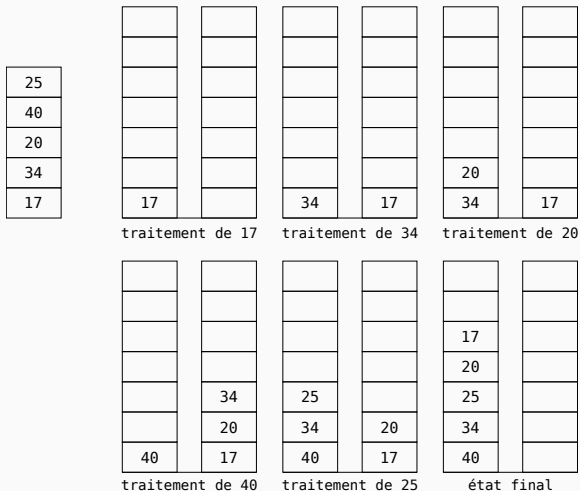


Figure 2: Un exemple de tri à deux piles

Le tri à deux piles

Exercice: formaliser l'algorithme

Le tri à deux piles

Exercice: formaliser l'algorithme

Algorithme de tri nécessitant 2 piles (G, D)

Soit tab le tableau à trier:

Pour tous les $i = 0$ 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
```

```
tab est trié dans G
```

Le tri à deux piles

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

Vocabulaire

2 + 3 = 2 3 +,

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

La calculatrice

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

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

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

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	Opération
2	2	empiler 2
3	2, 3	empiler 3
4	2, 3, 4	empiler 4
+	2, 7	dépiler 3 et 4, appliquer +, empiler 7
*	14	dépiler 2 et 7, appliquer *, empiler 14
5	14, 5	empiler 5
-	9	dépiler 14 et 5, appliquer -, empiler 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.