Piles

Algorithmique et structures de données, 2022-2023

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

2022-12-07

En partie inspirés des supports de cours de P. Albuquerque

Les piles (1/5)

Qu'est-ce donc?

- Structure de données abstraite...

Les piles (1/5)

Qu'est-ce donc?

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

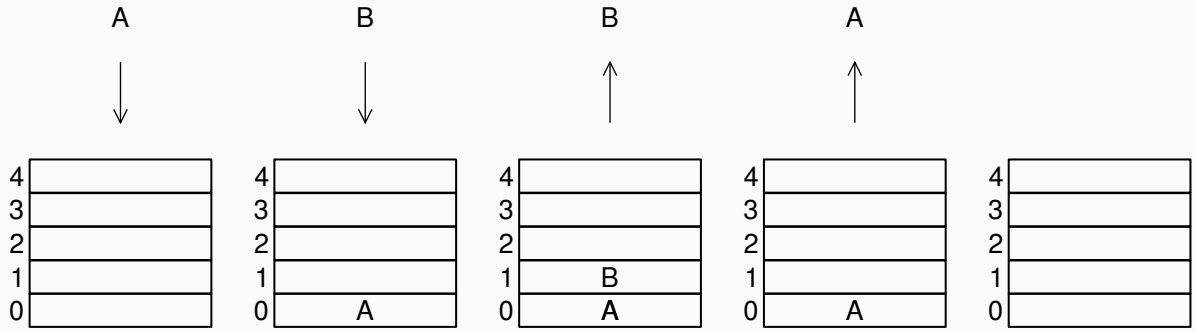

Figure 1: Une pile où on ajoute A, puis B avant de les retirer. Source: Wikipedia

Des exemples de la vraie vie

Les piles (1/5)

Qu'est-ce donc?

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

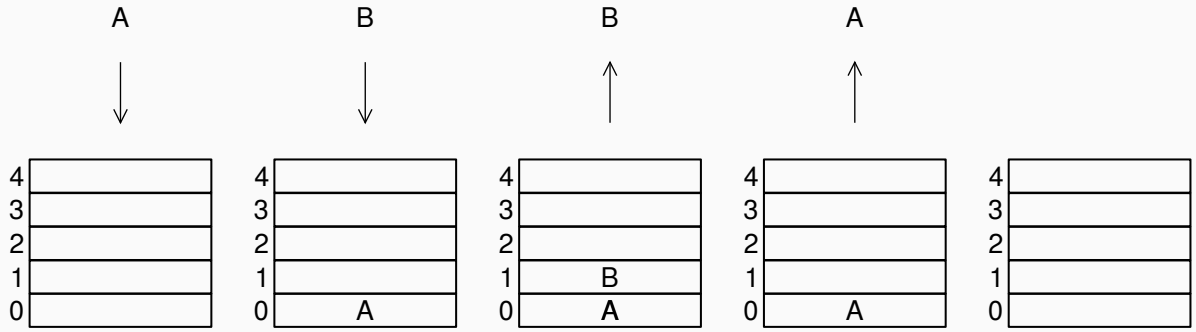

Figure 1: Une pile où on ajoute A, puis B avant de les retirer. Source: Wikipedia

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).

Les piles (2/5)

Fonctionnalités

Les piles (2/5)

Fonctionnalités

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

Les piles (2/5)

Fonctionnalités

1. Empiler (push): ajouter un élément sur la pile.
2. Dépiler (pop): retirer l'élément du sommet de la pile et le retrouner.
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).

Comment faire les 4,5 à partir de 1 à 3?

Les piles (2/5)

Fonctionnalités

1. Empiler (push): ajouter un élément sur la pile.
2. Dépiler (pop): retirer l'élément du sommet de la pile et le retrouner.
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).

Comment faire les 4,5 à partir de 1 à 3?

4. Dépiler l'élément, le copier, puis l'empiler à nouveau.
5. Dépiler jusqu'à ce que la pile soit vide, puis empiler à nouveau.

Les piles (2/5)

Fonctionnalités

1. Empiler (push): ajouter un élément sur la pile.
2. Dépiler (pop): retirer l'élément du sommet de la pile et le retrouner.
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).

Comment faire les 4,5 à partir de 1 à 3?

4. Dépiler l'élément, le copier, puis l'empiler à nouveau.
5. Dépiler jusqu'à ce que la pile soit vide, puis empiler à nouveau.

Existe en deux goûts

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

Les piles (3/5)

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?

Les piles (3/5)

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?

Et oui vous avez deviné: un tableau!

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

Les piles (3/5)

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?

Et oui vous avez deviné: un tableau!

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

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

Les piles (4/5)

Initialisation

Les piles (4/5)

Initialisation

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

Est vide?

Les piles (4/5)

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 (4/5)

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 (5/5)

Dépiler (enlever l'élément du sommet)

Les piles (5/5)

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 (5/5)

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 (5/5)

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?

Voyez-vous des problèmes potentiels avec cette implémentation?

Les piles (5/5)

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?

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).
  - Faire paniquer le programme (il plante plus ou moins violemment).
  - Utiliser des codes d'erreurs.

La panique

- En C, on a les `assert()` pour faire paniquer un programme.

Assertions (1/3)

```c
#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 (2/3)

Exemple

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

Assertions (3/3)

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, ...).

Assertions (3/3)

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, ...).
- Mais peuvent être pratiques quand même 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?

```c
// 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);
```

Et maintenant?

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);
```

Et maintenant?

```
stack_create(); // crée une pile avec une taille par défaut
// 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();
```

Exercice: ouvrir un repo/issues pour l'implémentation

- Oui-oui cela est une introduction au développement collaboratif (et hippie).

Le tri à deux piles (1/3)

Cas pratique

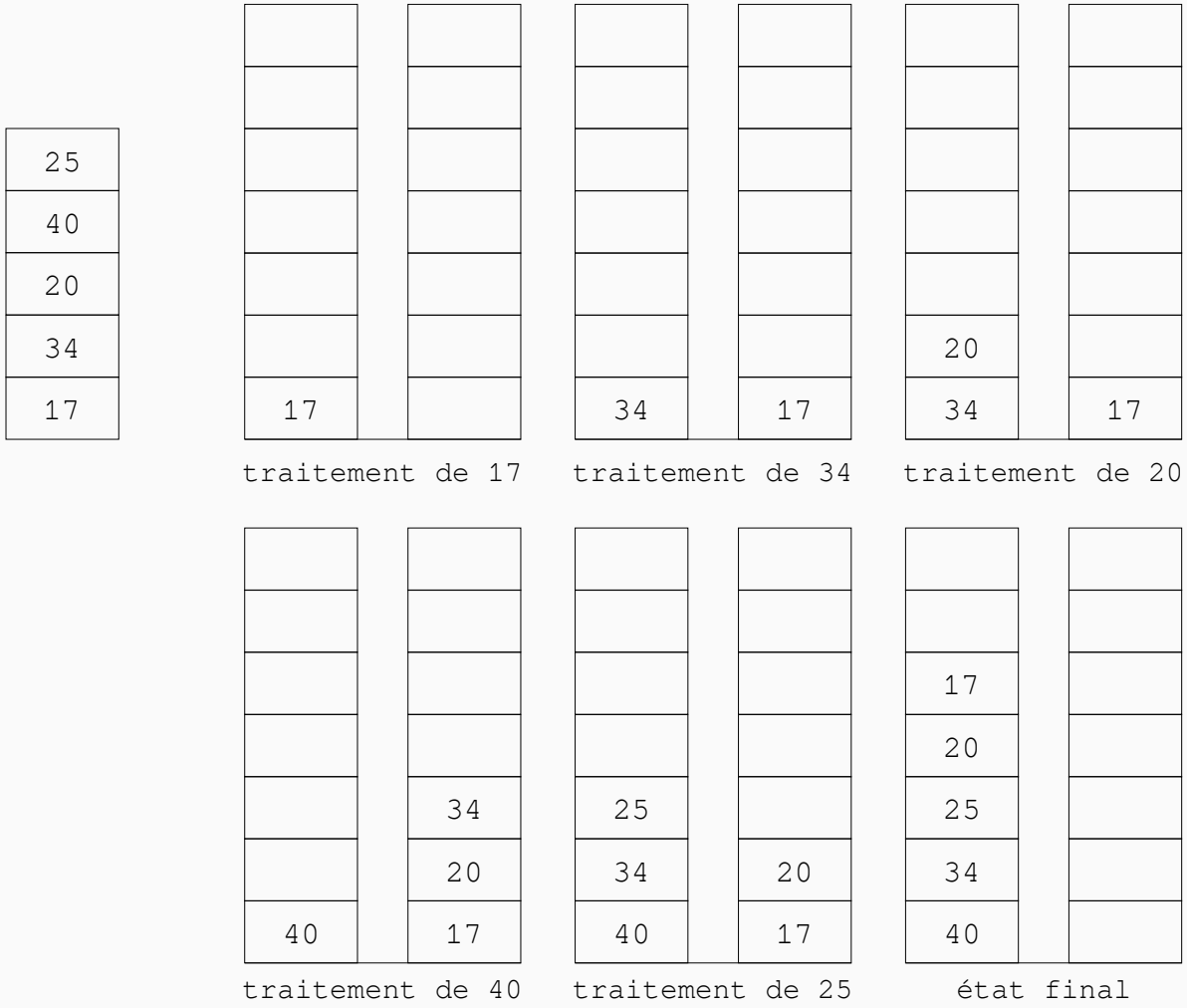

Figure 2: Un exemple de tri à deux piles

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
tab est trié dans G
```

Le tri à deux piles (3/3)

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