Algorithmique (en C)

Julien Alexandre dit Sandretto

Department U2IS
ENSTA ParisTech
Version 2019-2020

ENSTA

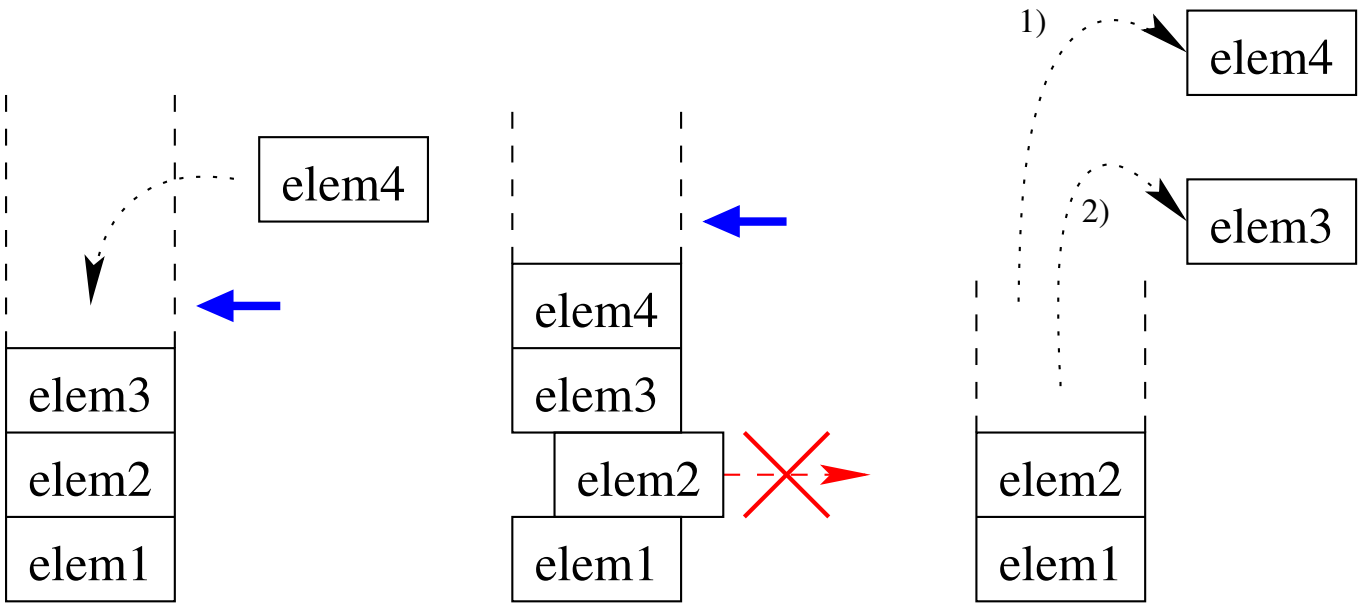
IP PARIS

ENSTA
IP PARIS

Piles

« Quand utiliser une pile ? » (« stack »)

- Comme dans la vie courante :
  stocker des choses comme une pile d'assiettes.
- ⇒ **Dernier posé** au sommet → **premier retiré**.
- En Anglais : **LIFO** (« **L***ast* **I***n* **F***irst* **O***ut* »).
- On peut consulter les éléments sous le sommet mais pas les retirer tant que ceux au-dessus sont dans la pile.



Application

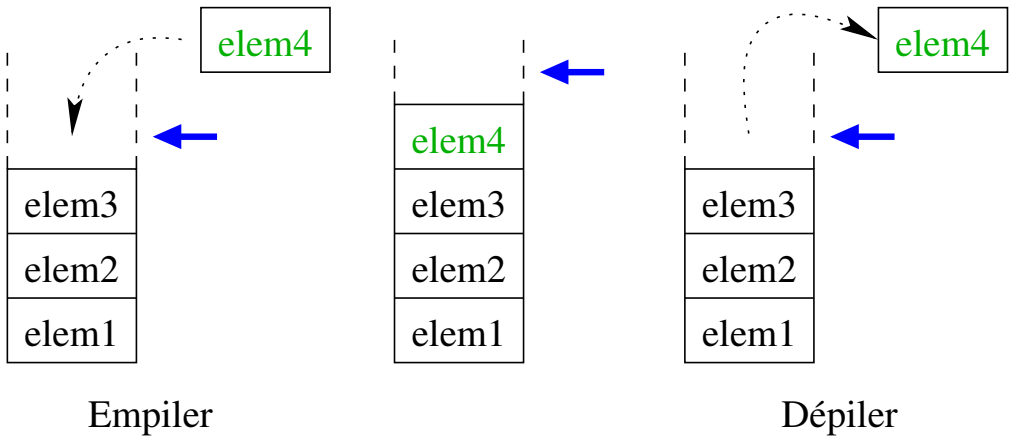
- Garder trace de traitements non complétés (« backtracking »).
- Garder trace d'un historique de traitements.

- Base de nombreuses machines virtuelles (Java par exemple).
  - Notation « *RPN* » : notation postfixe.
  - Opérateurs après les opérandes.
  - Comme les anciennes calculatrices HP.
  - Opérandes sur la pile, opérateur opérant sur les éléments de la pile.

- . . . Même structure que la **pile d'exécution** d'un programme. . .

Opérations sur une pile

- **Empiler** (« *push* ») : Ajouter un élément au sommet.
  - `push` : elem → stack → stack
- **Dépiler** (« *pop* ») : Retirer l'élément du sommet.
  - `pop` : stack → (elem * stack)
- **Consulter** (« *peek* ») : Accéder à un élément sans le retirer.
  - `peek` : stack → int → elem
- Le « **pointeur de pile** » indique la **prochaine** place **libre**.
- Le « *pointeur de pile* » varie au cours des opérations (sauf `peek`).
- On peut **tester** si la pile est **vide**.



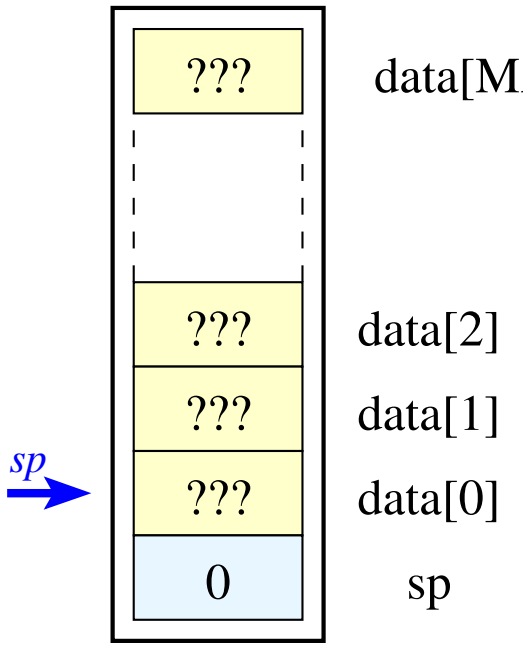
Pile : implémentation avec un tableau

- Ici, pile de `ints` avec une taille maximale `MAX_SIZE`
  - Utiliser un tableau dynamique pour supprimer cette restriction.

```
#define MAX_SIZE 64
struct stack {
  unsigned int sp ;     /* Pointeur de pile. */
  int data[MAX_SIZE] ; /* Mémoire de pile. */
};
```

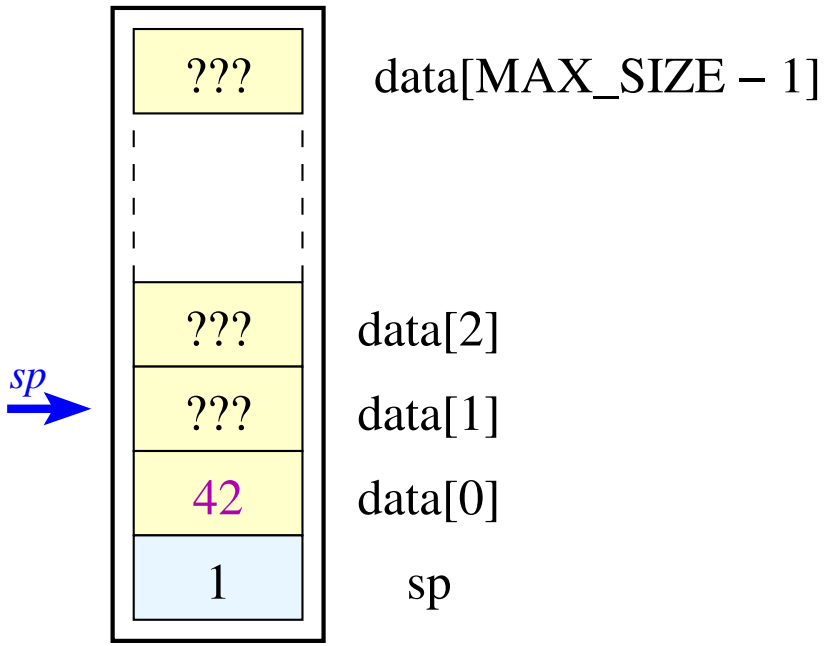
- Pile ≡ simple tableau.
- « *Pointeur* » de pile = entier **positif** :
  - Représente l'indice de la **prochaine** « *case* » **libre**.
  - ⇒ Représente le nombre d'éléments présents dans la pile.
- On ajoute et on retire toujours « *par le haut* ».

Pile : fonctionnement avec un tableau (1)



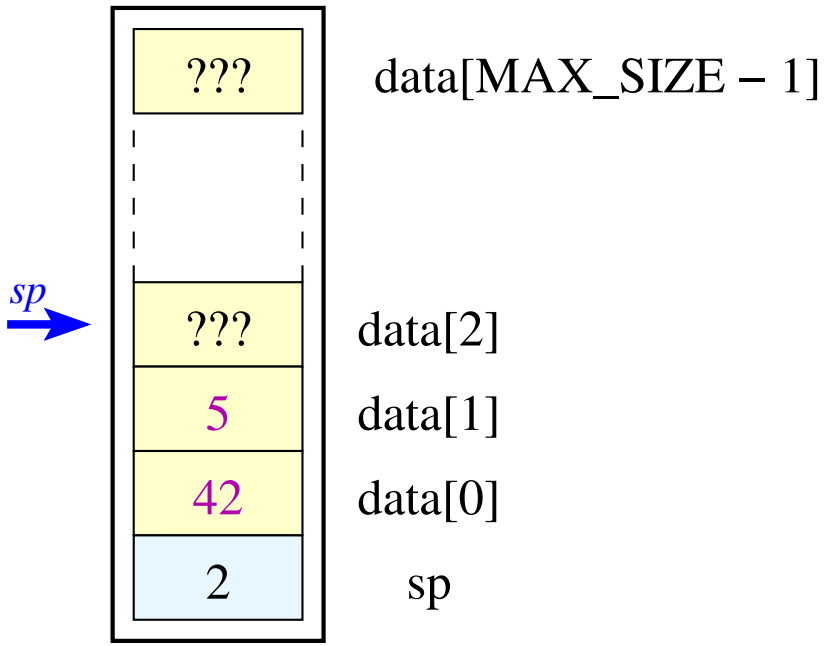
- Init : Pile vide, sp = 0.

Pile : fonctionnement avec un tableau (2)



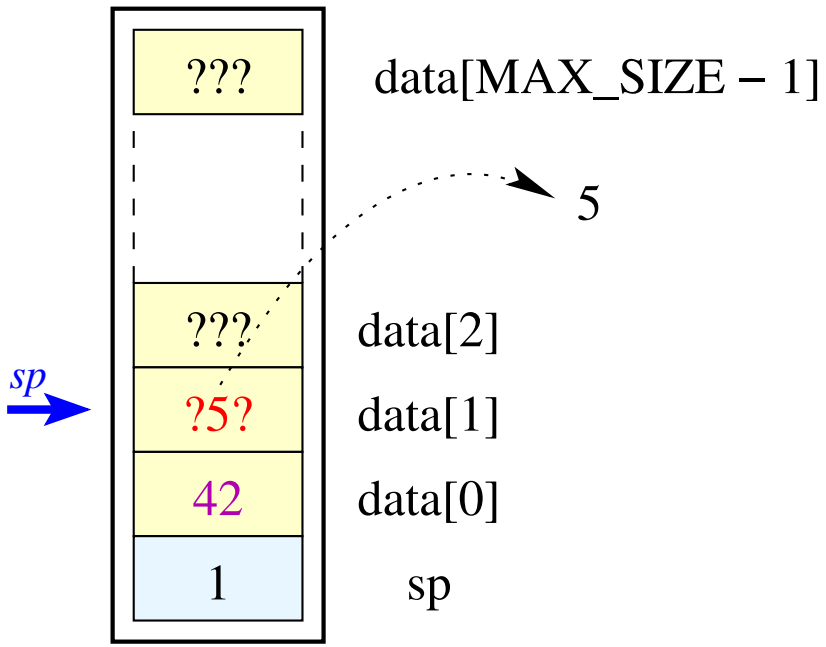
- `push 42` :
  - Vérifier que sp < MAX_SIZE
  - 42 mis à l'indice sp ⇒ 0
  - Incrémenter sp : sp ← sp + 1 ⇒ 1

Pile : fonctionnement avec un tableau (3)



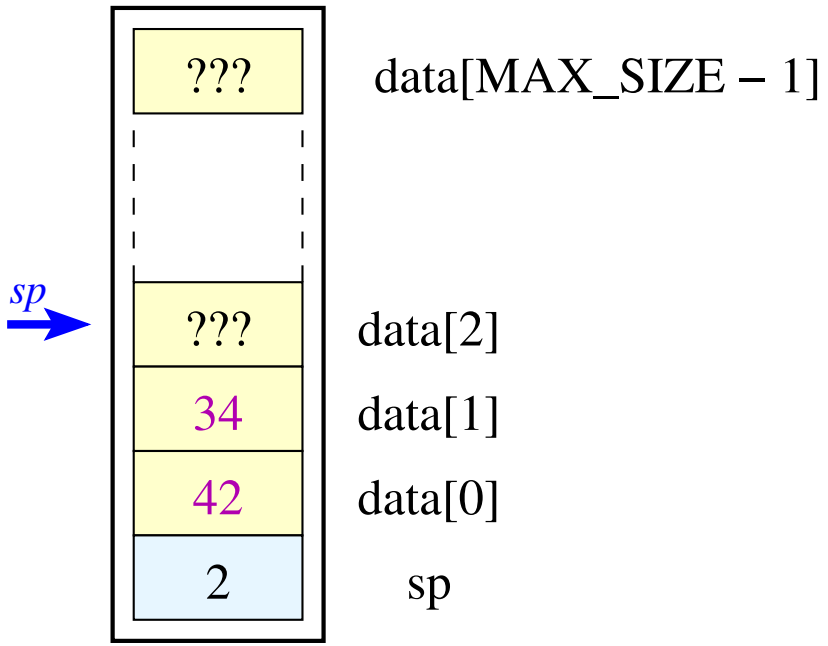
- `push 5` :
  - Vérifier que sp < MAX_SIZE
  - 5 mis à l'indice sp ⇒ 1
  - Incrémenter sp : sp ← sp + 1 ⇒ 2

Pile : fonctionnement avec un tableau (4)



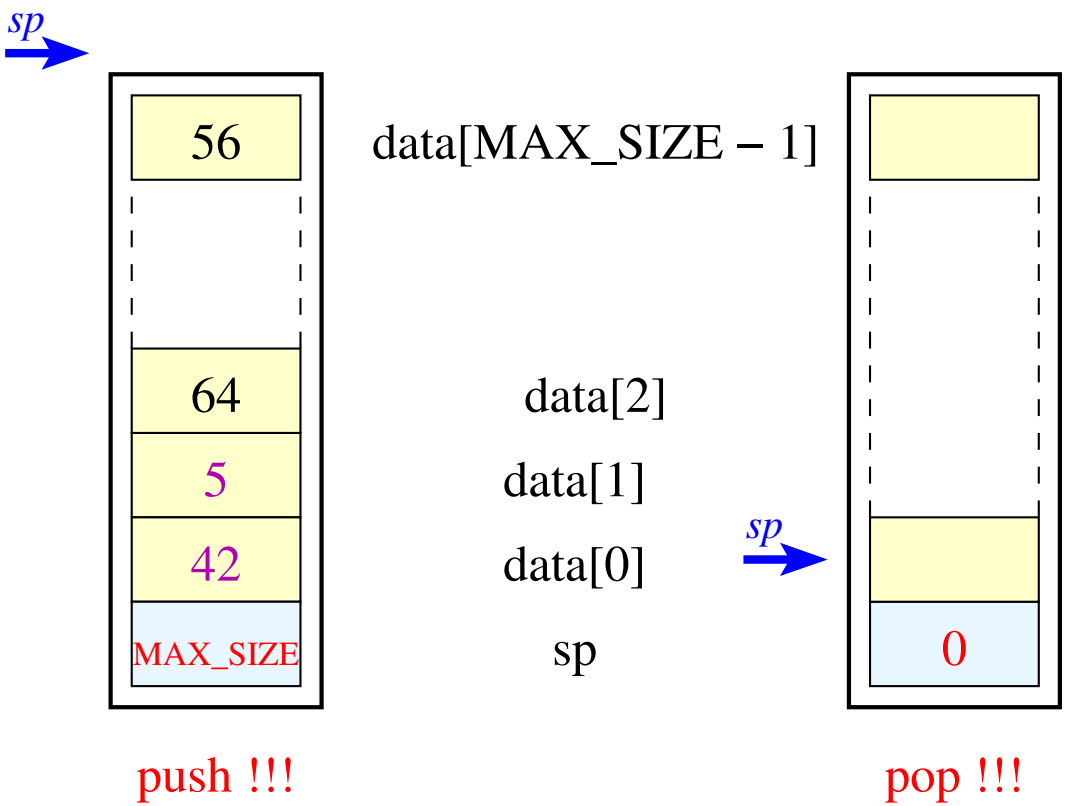
- pop :
  - Vérifier que sp > 0
  - Décrémenter sp : sp ← sp - 1 ⇒ 1
  - Retourner la valeur à l'indice sp (sommet de la pile)
- La valeur reste dans la pile mais **n'est plus à utiliser**.
- Elle **sera écrasée** au prochain push.

Pile : fonctionnement avec un tableau (5)



- push 34 :
  - Vérifier que sp < MAX_SIZE
  - 5 mis à l'indice sp ⇒ 1
  - Incrémenter sp : sp ← sp + 1 ⇒ 2
- Ancienne valeur ?5? écrasée par 34.

Pile : fonctionnement avec un tableau (6)



- push dans une pile pleine :
  - sp = MAX_SIZE.
  - ⇒ **erreur**.
- pop sur une pile vide :
  - sp = 0.
  - ⇒ **erreur**.

Pile : implémentation en C

```
int pop (struct stack *P)
{
  if (P->sp == 0) {
    /* Pile vide. */
    error () ;
  }
  P->sp-- ;
  return (P->data[P->sp]) ;
}

void push (struct stack *P, int val) {
  if (P->sp == MAX_SIZE) {
    /* Pile pleine. */
    error () ;
  }
  P->data[P->sp] = val ;
  P->sp++ ;
}
```

Pile : implémentation en C avec un tableau dynamique

- Taille maximale n'est plus une constante
  - ⇒ Appartient à la structure de pile.
  - « *Tableau dynamique* » ⇒ **pointeur**.

```
struct stack {
  unsigned int size ;           /* Taille de la pile. */
  unsigned int sp ;          /* Pointeur de pile. */
  int *data ;                /* Mémoire de pile. */
};
```

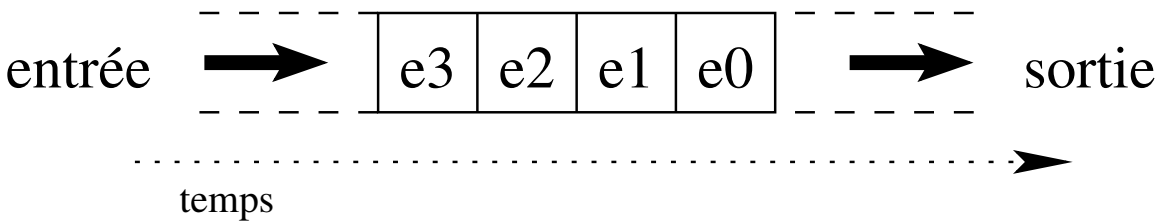
- Seul push change : en cas de pile pleine, on agrandit le tableau.

```
void push (struct stack *P, int val) {
  if (P->sp == P->size) {
    /* Pile pleine. */
    P->size *= 2 ;
    P->data = realloc (P->data, P->size) ;
  }
  P->data[P->sp] = val ;
  P->sp++ ;
}
```

Files

« *Quand utiliser une file ?* »

- Stocker des choses à traiter dans l'ordre d'arrivée (≈ file d'attente).
- ⇒ **Premier inséré** en queue → **premier retiré** en tête.
- En Anglais : **FIFO** (« **F***irst* **I***n* **F***irst* **O***ut* »).
- On peut consulter les éléments dans la file mais pas les retirer tant que ceux de devant sont dans la file.

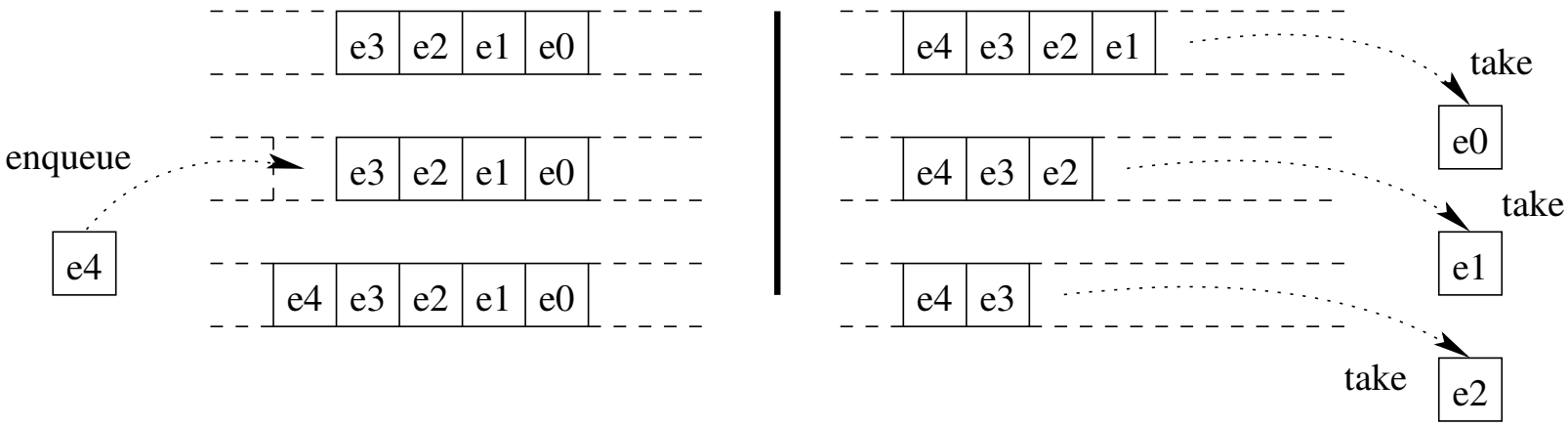


Application

- Stocker des « transactions » à effectuer.
  - Mémoire tampon (« *buffer* »).
- Ordonnancer les processus dans un OS multitâche.
- Parcours « en largeur » d'arbres ou de graphes (c.f. plus tard).
- Services client/serveur (WEB).

Opérations sur une file

- **Ajouter** (« *enqueue* ») : Ajouter un élément en queue.
  - `enqueue` : elem → file → file
- **Retirer** (« *take* ») : Retirer l'élément de tête.
  - `take` : file → (elem * file)
- **Consulter** (« *peek* ») : Accéder à un élément sans le retirer.
  - `peek` : file → int → elem
- On peut **tester** si la file est **vide**.

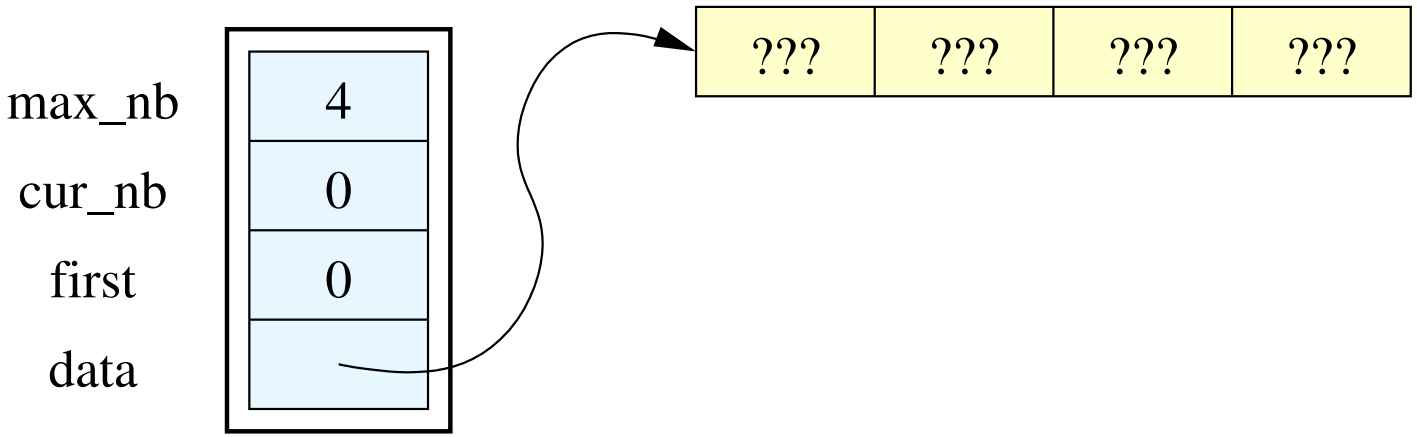


File : implémentation avec un tableau

- Utilisation d'un tableau avec taille bornée.
  - Comme les piles : on peut utiliser un tableau dynamique.
- Ajout des éléments en **fin** du tableau.
- Retrait des éléments en **début** de tableau.
- **Mais** on **ne veut pas décaler** tout le tableau (impossible en $\theta(1)$).
  - $\Rightarrow$ Utilisation du tableau de manière **cyclique**.
  - Arrivé au bout du tableau, on recommence au début.
- Besoin de garder l'indice du 1er élément (le + ancien encore vivant).

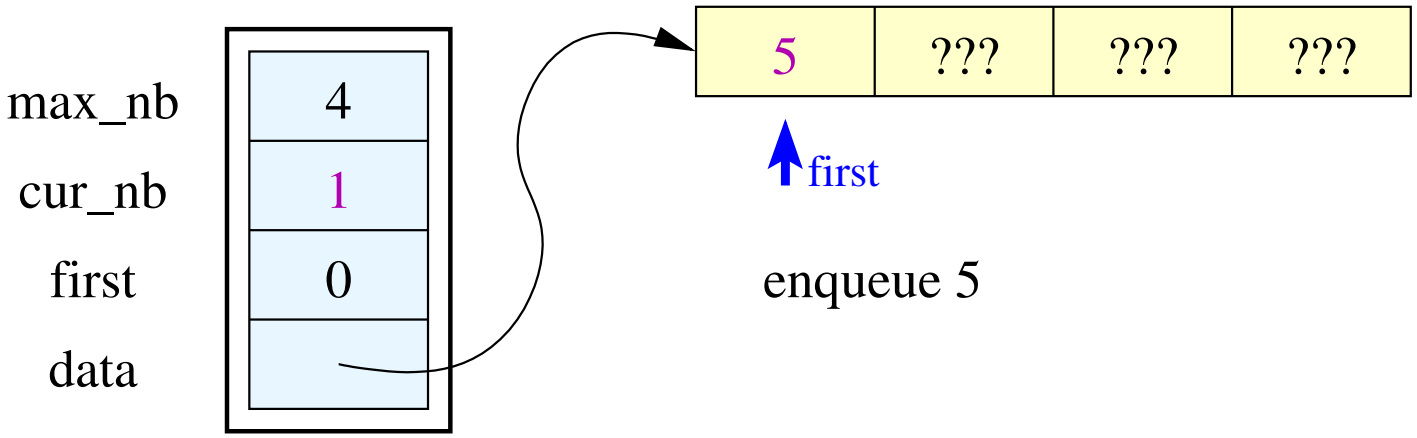
```
struct file_t {
  unsigned int max_nb ;       /* Nombre max d'éléments. */
  unsigned int cur_nb ;       /* Nombre actuel d'éléments. */
  unsigned int first ;        /* Indice du premier élément. */
  int *data ;
};
```

File : fonctionnement avec un tableau (1)



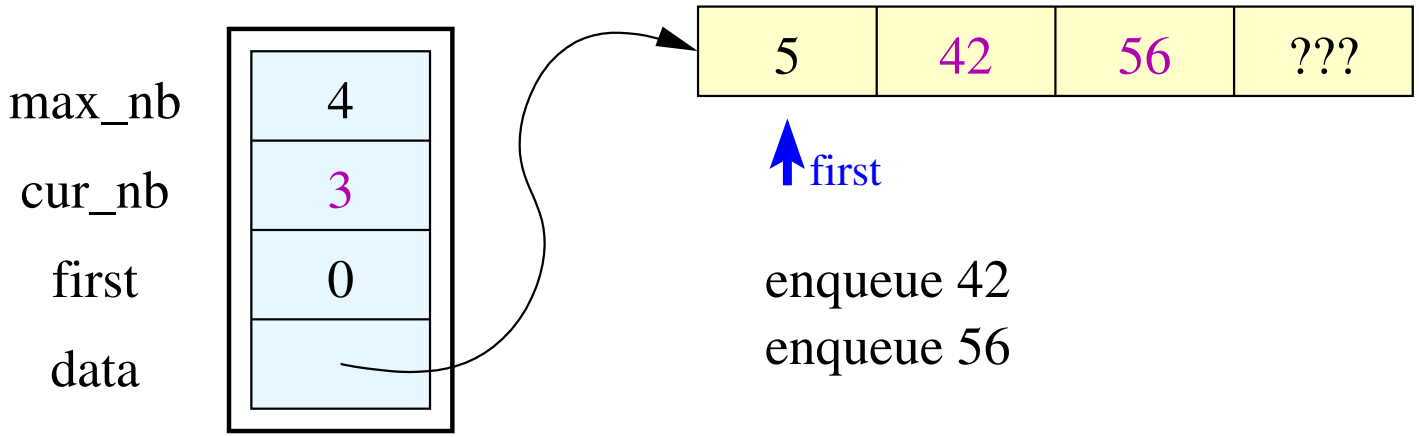
- Taille de la file : 4.
- Initialisation : file vide, i.e. aucune donnée.
- $\Rightarrow$ `cur_nb`, `first` $= 0$.

File : fonctionnement avec un tableau (2)



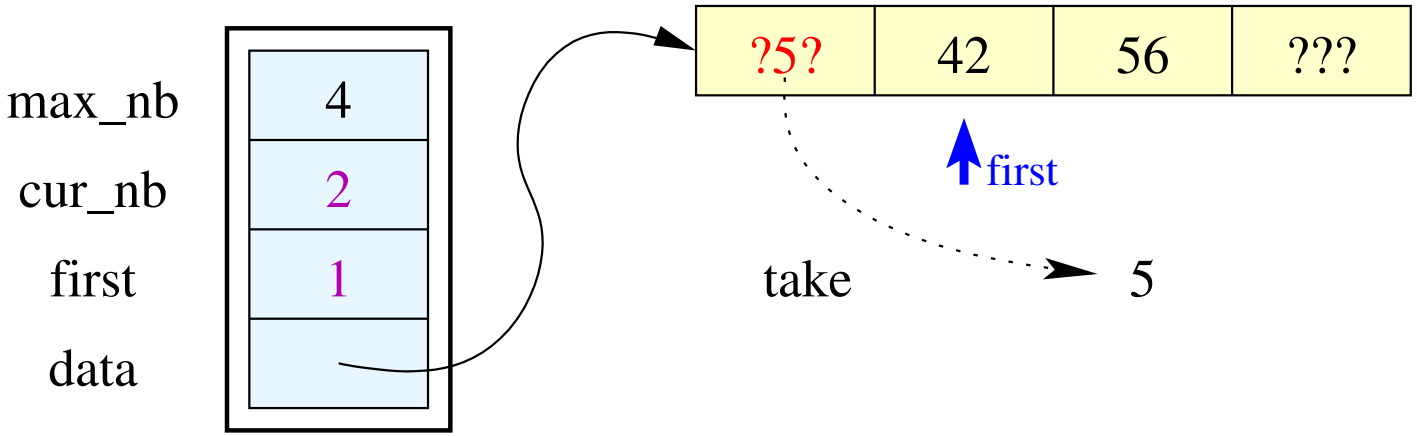
- `enqueue 5` :
  - Vérifier que la file n'est pas pleine.
  - Ajouter au *« premier indice libre »*.
  - Incrémenter `nb_cur` : `nb_cur` $\leftarrow$ `nb_cur` $+ 1 \quad \Rightarrow 1$

File : fonctionnement avec un tableau (3)



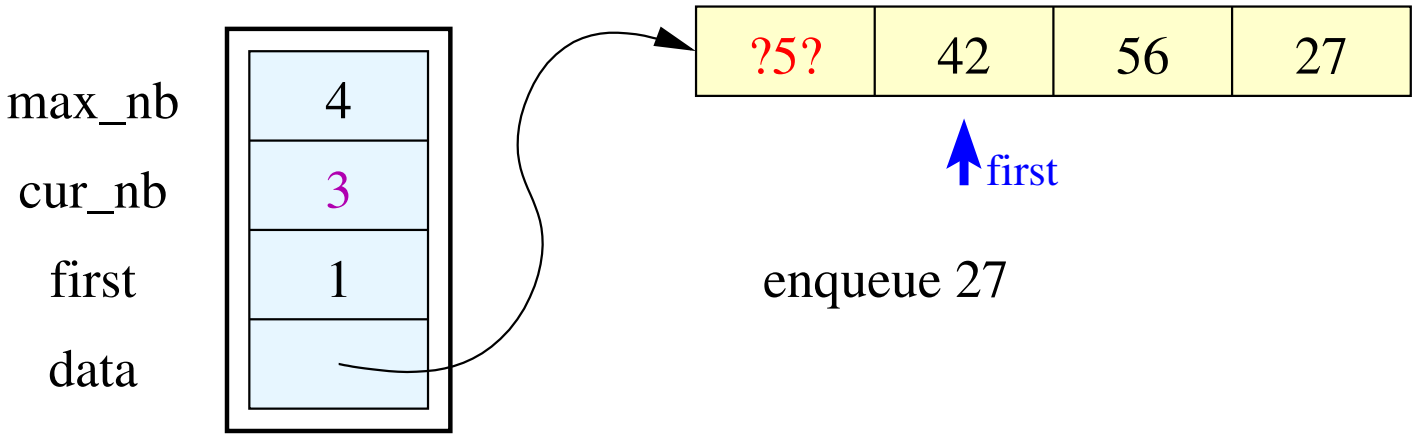
- enqueue 42, enqueue 56 :
  - Vérifier que la file n'est pas pleine.
  - Ajouter au *« premier indice libre »*.
  - Incrémenter `nb_cur` : `nb_cur` $\leftarrow$ `nb_cur` $+ 1 \quad \Rightarrow 3$
- *« Premier indice libre »* : *first* + *cur_nb* . . . tant que $\not\geq$ taille max.

File : fonctionnement avec un tableau (4)



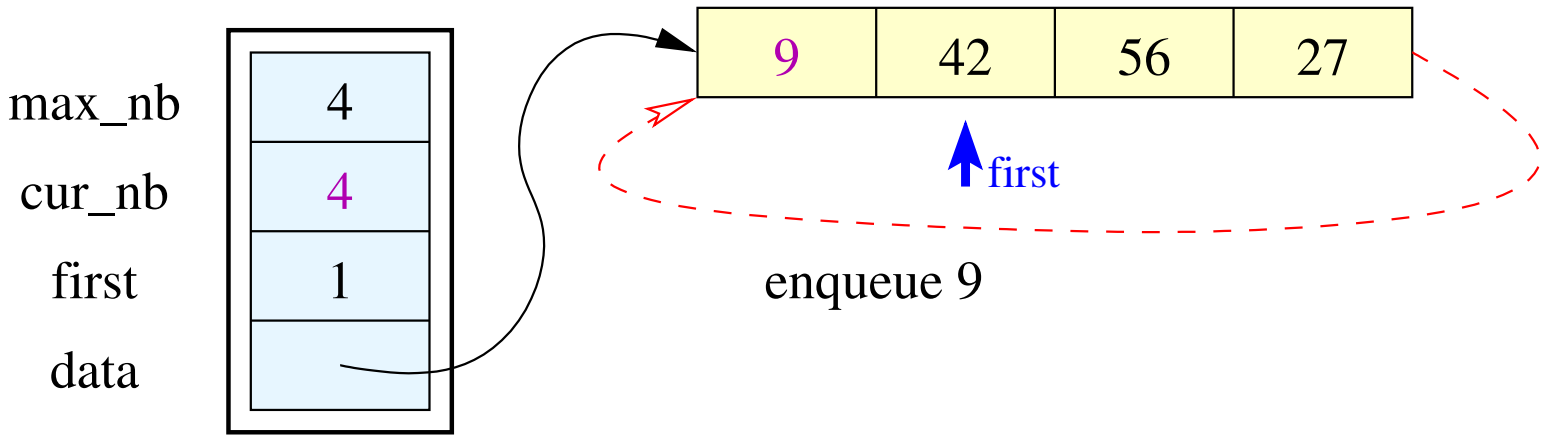
- `take` :
  - Vérifier que la file n'est pas vide.
  - Récupérer la donnée à l'indice `first`.
  - Incrémenter `first` : `first` ← `first` + 1 ⇒ 1 . . . tant que ≱ taille max.
  - Décrémenter `nb_cur` : `nb_cur` ← `nb_cur` - 1 ⇒ 2
- La « *case* » redevient libre mais n'est pas effacée.

File : fonctionnement avec un tableau (4)



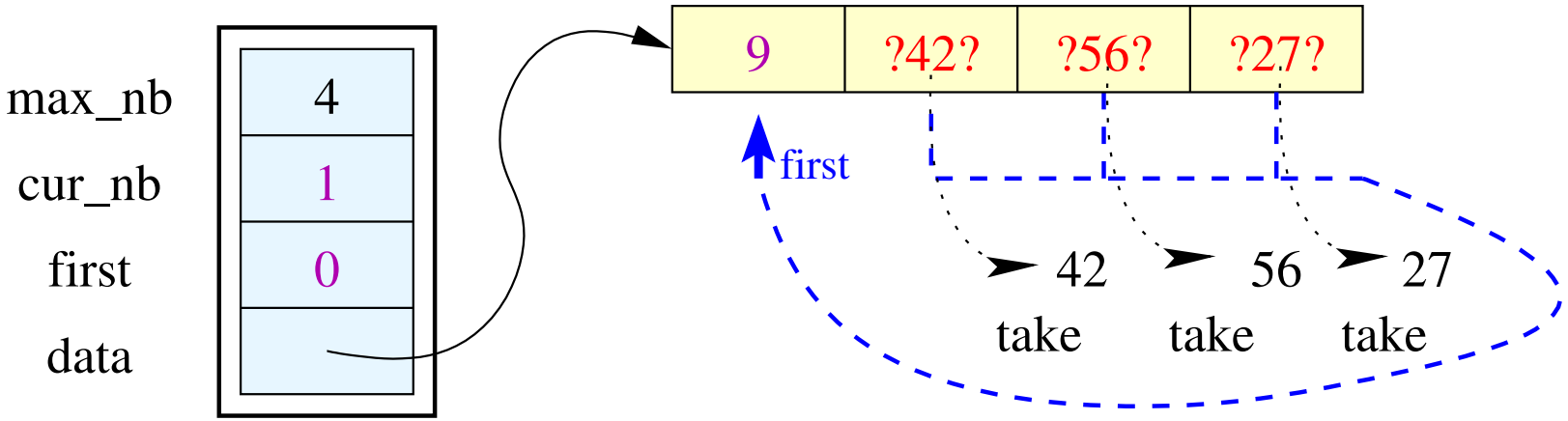
- enqueue 27

File : fonctionnement avec un tableau (5)



- `enqueue 9`
- Arrivé en bout de tableau la file n'est pas forcément pleine.
- Dépend de s'il y a de la place en début du tableau.
  - ⇒ On fait un cycle et réinsère en début.
  - « *Première case libre* » : $(first + cur\_nb)$ **mod max_nb**

File : fonctionnement avec un tableau (6)



- `take` ; `take` ; `take`
- Arrivé en bout de tableau la file n'est pas forcément vide.
- Dépend de s'il y a des données en début du tableau.
  - ⇒ On fait un cycle et extrait en début.
  - `first` ← (first + 1) **mod max_nb**
- Situation identique à celle après la 1ère insertion.

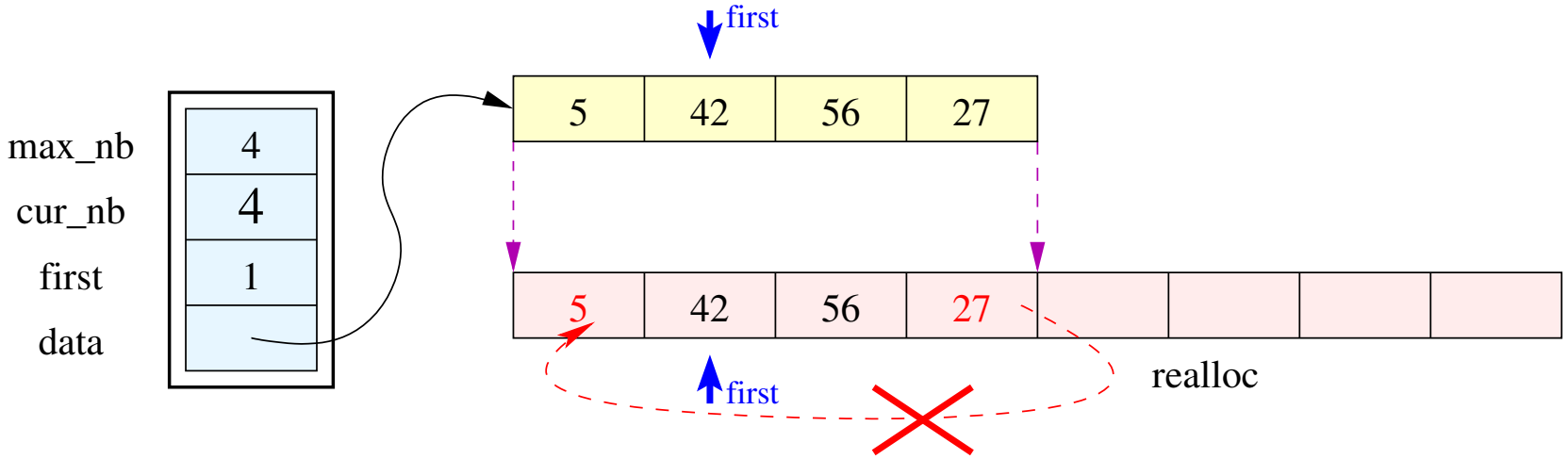
File : implémentation en C

```
int take (struct queue *F) {
  int res ;
  if (F->cur_nb == 0) {
    /* File vide. */
    error () ;
  }
  res = F->data[F->first] ;
  F->first = (F->first + 1) % F->max_nb ;
  F->cur_nb-- ;
  return (res) ;
}

void enqueue (struct queue *F, int val) {
  if (F->cur_nb == F->max_nb) {
    /* File pleine. */
    error () ;
  }
  F->data[(F->first + F->cur_nb) % F->max_nb] = val ;
  F->cur_nb++ ;
}
```

Utiliser l'aspect dynamique pour éviter la file pleine (1)

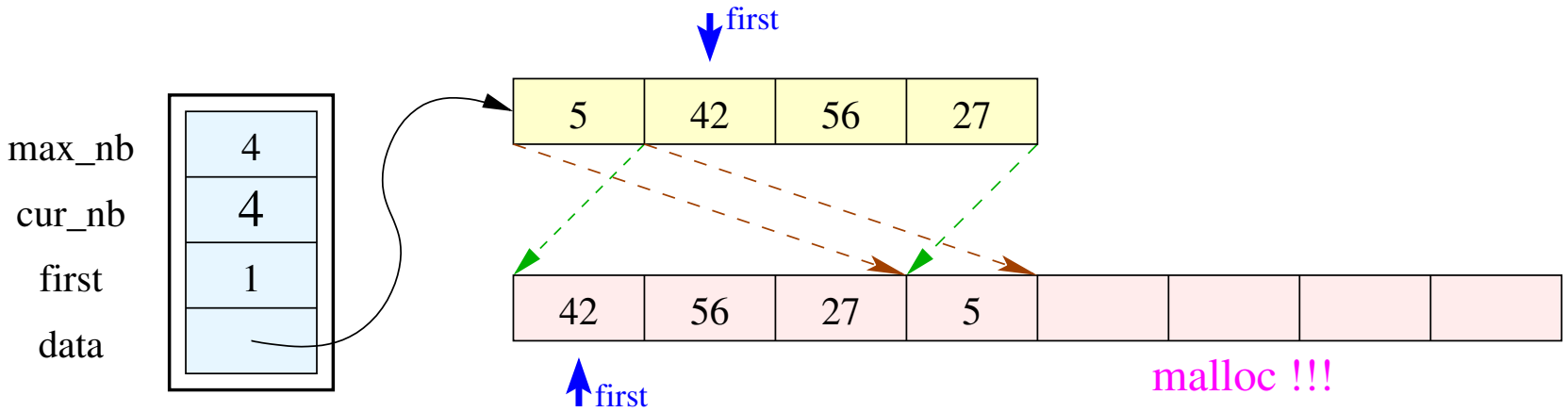
- Traitement à intercaler en cas de file pleine (`cur_nb == max_nb`).
- Plus compliqué que pour les piles : `realloc` ne convient pas.



- 5 ne suit pas 27 puisque espace libre après 27 !

Utiliser l'aspect dynamique pour éviter la file pleine (2)

- Idée :
  - Recopier depuis *first* → fin du tableau en début de nouveau tableau.
  - Recopier depuis 0 → *first* – 1 à la suite du nouveau tableau.
  - Remettre `first` au début du nouveau tableau (→ 0).



- Plus de `realloc` ⇒ gérer la libération de mémoire.

Utiliser l'aspect dynamique pour éviter la file pleine (3)

- memcpy (dest, src, size) : copie mémoire-mémoire.
- Nécessite #include <string.h>

```
void enqueue (struct queue *F, int val) {
  int* new_data ;
  if (F->cur_nb == F->max_nb) {
    /* File pleine. */
    new_data = malloc (F->max_nb * 2 * sizeof (int)) ;
    memcpy
      (new_data, &(F->data[F->first]),
       (F->max_nb - F->first) * sizeof (int)) ;
    memcpy
      (&(new_data[F->max_nb - F->first]), F->data,
       F->first * sizeof (int)) ;
    free (F->data) ;              /* Libération ancien data. */
    F->data = new_data ;
    F->first = 0 ;
    F->max_nb *= 2 ;
  }
  ...
```