PROGRAMMATION C

TP 2 – 2 octobre 2025 – durée : 2h40

Implémentation des piles et des files

TP 2 de Programmation C

On souhaite écrire une *implémentation* des types abstraits « pile » et « file » pouvant stocker des entiers. Nous écrirons aussi des fonctions *utilisant* des piles ou des files, pour tester l'implémentation.

Préliminaires

Téléchargez et extrayez l'archive `tp2.tgz`. Elle contient les fichiers sources. Vous modifierez *uniquement* les suivants :

- `stack_by_array.c` : implémentation des piles,
- `queue_by_circular_array.c` et `queue_by_2_stacks.c` : deux implémentations différentes des files,
- `stack_use.c` et `queue_use.c` : utilisation des files et des piles, pour tester les implémentations.
- `main.c` et `main.h` : tests.

Les autres fichiers **ne doivent pas** être modifiés. Vous pouvez commencer par travailler sur `stack_by_array.c` et faire les premières fonctions de `stack_use.c`, puis activer les tests correspondants dans `main.c`, comme expliqué ci-dessous.

Les fichiers sources sont les suivants :

- `main.c` contient la fonction `main()` du programme et sert à tester vos fonctions. Il contient déjà des tests, mais vous pouvez en ajouter. Dans `main.h`, basculez à `1` les macros suivantes pour activer les tests correspondants :

```
#define TEST_STACK_BASIC 0
#define TEST_SORT_STACK  0
#define TEST_COPY_STACK  0
#define TEST_HANOI       0
```

```
#define TEST_QUEUE_BASIC 0
#define TEST_COPY_QUEUE  0
#define TEST_SORT_QUEUE  0
```

  Par ailleurs, *deux implémentations* des files sont demandées, et on choisit à la compilation laquelle utiliser.
  - Pour compiler l'implémentation d'une file par *tableau circulaire*, laissez à `0` la macro `QUEUE_BY_TWO_STACKS` du fichier `main.h`.
  - Pour compiler l'implémentation d'une file par *deux piles*, basculez à `1` la macro `QUEUE_BY_TWO_STACKS` du fichier `main.h`.
- `stack.h` et `stack_by_array.c` sont dédiés à l'*implémentation* d'une *pile* par tableau (dont la taille change dynamiquement). Les prototypes sont écrits et documentés dans `stack.h`, à ne pas modifier. Vous devez compléter le fichier `stack_by_array.c`, qui définit aussi le type utilisé.
- `stack_use.h` et `stack_use.c` sont dédiés à l'*utilisation* des *piles*. Les prototypes sont écrits et documentés dans `stack_use.h`, à ne pas modifier. Vous devez compléter le fichier `stack_use.c`.
- `queue.h` et `queue_by_circular_array.c` sont les fichiers dédiés à l'*implémentation* d'une *file* par tableau circulaire (dont la taille change dynamiquement). Les prototypes sont écrits et documentés dans le fichier `queue.h`, à ne pas modifier. Vous devez compléter le fichier `queue_by_circular_array.c`, qui définit aussi le type utilisé.

- `queue.h` et `queue_by_2_stacks.c` sont les fichiers dédiés à l'*implémentation* d'une *file* par deux piles. Les prototypes sont écrits et documentés dans le fichier `queue.h`, à ne pas modifier. Vous devez compléter le fichier `queue_by_2_stacks.c`, qui définit aussi le type utilisé.
- `queue_use.h` et `queue_use.c` sont dédiés à l'*utilisation* des *files*. Les prototypes sont écrits et documentés dans `queue_use.h`, à ne pas modifier. Vous devez compléter `queue_use.c`.

Les autres fichiers contiennent des fonctions utiles **déjà écrites**, qu'il ne faut pas modifier.

- `print.h` contient le prototype de fonctions d'affichage de piles et de files, déjà écrites dans les fichiers `stack_by_array.c`, `queue_by_circular_array.c` et `queue_by_2_stacks.c`.
- `alloc.h`, `alloc.c` et `error.h` donnent accès à des macros pour l'allocation mémoire et l'affichage de messages de débogage. Il n'est pas obligatoire de les utiliser.
- Enfin, un fichier `Makefile` fournit trois cibles :
  - `make` compile l'ensemble des fichiers et construit l'exécutable `stackqueue`. Un avertissement de `gcc` provoque une erreur. Comme les fonctions à écrire sont vides au départ, ce mode désactive les avertissements concernant les variables ou fonctions non utilisées.
  - `make no-error-on-warning` fait de même, mais avec les avertissements précédents activés. En revanche, un avertissement de `gcc` n'est pas traité comme une erreur.
  - `make clean` nettoie le répertoire. Il est *inutile de nettoyer à chaque compilation* : l'intérêt de `make` est de ne recompiler que ce qui est nécessaire. En revanche, il faut nettoyer ou compiler par `make -B` *lorsque vous changez d'implémentation des files*.

Pensez à tester votre code *régulièrement*

Vous devrez écrire des tests. Pour cela, vous disposez des fonctions d'affichage de piles et files, déjà écrites. Par exemple, une fonction permettant d'afficher une ou plusieurs piles est fournie et a pour prototype :

```
void print_stacks(int h, int nb, ...);
```

Les points de suspension (`...`) indiquent que la fonction prend un nombre variable d'arguments. La façon de l'utiliser est décrite dans `print.h`.

Fichiers à rendre

À la fin de la séance, vous devrez rendre sur Moodle une archive contenant les fichiers que vous avez écrits. Pas de panique, si vous n'avez pas terminé : vous pourrez les rendre ultérieurement.

1 Implémentation d'une pile par un tableau dynamique

Les fichiers `stack.h` et `stack_by_array.c` sont dédiés à cette implémentation. Le fichier `stack.h` **ne doit pas être modifié** : il contient la définition du type et les prototypes des primitives que vous devez écrire dans `stack_by_array.c`.

Cette implémentation est une adaptation pour le langage C de celle basée sur les tableaux vue en cours d'algorithmique. Commençons par rappeler la structure de données utilisée. Considérons une pile contenant k valeurs pour $k \in \mathbb{N}$. On représente cette pile en utilisant deux objets :

- Un entier égal au nombre de valeurs dans la pile (c'est-à-dire k dans la définition).
- Un tableau *dont la taille est une puissance de deux* supérieure ou égale à k. Les valeurs de la pile sont écrites dans les cellules 0 à $k-1$ (le bas de la pile dans la cellule 0 et le sommet dans la cellule $k-1$). Les valeurs dans les autres cellules du tableau (si il y en a) ne sont pas significatives pour la représentation.

On donne la représentation graphique d'un exemple de cette structure de données dans la Figure 1 ci-dessous.

Les éléments de la pile seront accessibles par un pointeur sur le premier élément (champ `array` de la structure ci-dessous). Notre structure de données contient également un entier enregistrant la taille du tableau, ainsi que la taille de la

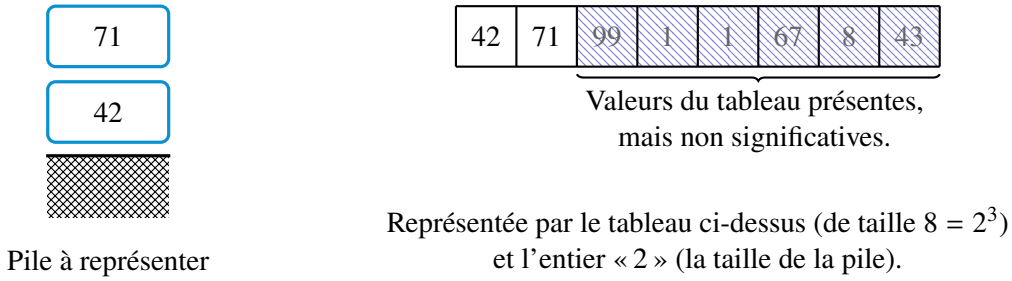


FIGURE 1 : Représentation d'une pile en utilisant un tableau

pile représentée (c'est-à-dire le nombre de valeurs significatives du tableau). On utilisera le type suivant (déjà écrit dans `stack.h`, donc à ne pas modifier) :

```c
typedef struct {
  int capacity;   // Taille totale du tableau alloué en mémoire.
  int size_stack; // Nombre d'éléments dans la pile représentée.
  int *array;     // Tableau contenant les valeurs de la pile représentée.
} stack;
```

Pour cette partie, le travail consiste à écrire les primitives du type abstrait « pile », pour cette implémentation.

Vous devrez également écrire deux fonctions auxiliaires qui servent à diviser par deux ou à multiplier par deux la taille du tableau utilisé dans la structure de données. Elles devront être appelées « au bon moment » dans les implémentations des primitives (voir commentaires dans le fichier `stack_by_array.c`) :

```c
// Double la taille du tableau utilisé dans la représentation.
static void grow_stack(p_stack);

// Divise par 2 la taille du tableau utilisé dans la représentation.
static void shrink_stack(p_stack);
```

Ces fonctions sont déclarées avec le mot-clé `static`, ce qui signifie qu'elles sont « privées » au fichier `stack_by_array.c`. Leur prototype n'apparaît donc pas dans `stack.h`. Vous pouvez ajouter autant de telles fonctions auxiliaires que vous le souhaitez dans `stack_by_array.c` (en les déclarant `static` et sans ajouter leur prototype dans `stack.h`). Nous pouvons maintenant présenter les prototypes des primitives à écrire pour le type abstrait « pile » :

```c
// Crée une pile vide.
stack *create_stack(void);

// Libère la mémoire allouée pour la pile.
void delete_stack(stack *);

// Renvoie vrai si la pile est vide, faux sinon.
bool isempty_stack(stack *);

// Renvoie la valeur au sommet de la pile et la retire. Si la pile est vide, la
// fonction affiche un message sur la sortie erreur standard et termine le
// programme. Si l'occupation du tableau tombe à 25% après le pop(), le tableau
// est redimensionné par la fonction shrink_stack().
int pop(stack *);

// Ajoute une valeur au sommet de la pile. Si le tableau est plein, il est
// redimensionné au préalable, par la fonction grow_stack().
void push(int, stack *);
```

2 Fonctions de test sur les piles

Cette partie consiste à écrire des fonctions permettant de tester votre implémentation des piles. Leurs prototypes sont déjà écrits dans le fichier `stack_use.h`. Les fonctions elles-mêmes sont à écrire dans `stack_use.c`.

Un type abstrait est indépendant de son implémentation

Votre code doit être indépendant de l'implémentation des piles écrite dans la première partie. En d'autres termes, vous pouvez utiliser les primitives décrites dans `stack.h`, mais vous ne pouvez **pas** « regarder » la structure de données. C'est la raison pour laquelle le fichier `stack.h` ne décrit pas la structure de données `stack` : il ne fait que la déclarer, par `typedef struct stack stack;`. Autrement dit, l'utilisateur de votre bibliothèque sait qu'il existe un type `stack`, mais ne peut pas manipuler directement les champs de cette structure. Il ne peut la qu'utiliser primitives déclarées dans `stack.h`.

C'est le principe d'un type abstrait : une fois l'implémentation écrite, l'utilisation du type est indépendante de celle-ci. Les tests doivent donc fonctionner même si l'implémentation des piles change.

La liste des fonctions à écrire est la suivante :

```c
// Renvoie le nombre d'éléments dans la pile.
int getsize_stack(stack *);

// Crée une pile aléatoire de size valeurs, chacune entre 0 et maxval.
// Si size est inférieur ou égal à 0, la pile construite est vide.
stack *random_stack(int size, int maxval);

// Crée la pile suivante : n, n-1, n-2, ..., 2, 1 (1 en haut de pile).
// Si n est inférieur ou égal à 0, la pile construite est vide.
stack *tower_stack(int n);

// Crée une copie de la pile prise en entrée.
// Cette dernière doit rester inchangée après appel de la fonction.
stack *copy_stack(stack *);

// Trie la pile par ordre croissant (la plus grande valeur au sommet).
// L'algorithme devra être basé sur le principe du tri à bulles.
// Une seule structure auxiliaire est autorisée: une seconde pile.
// Toute autre structure auxiliaire (comme un tableau) est interdite.
void bubble_sort_stack(stack *);

// Tours de Hanoi.
//
// Si n > 0, affiche la séquence de mouvements qui résout le problème des tours
// de Hanoi pour n disques. On utilisera la primitive qui sert à afficher une ou
// plusieurs piles.
void hanoi(int n);

// Fonction récursive auxiliaire pour les tours de Hanoi. Paramètres :
// - p1, p2, p3 sont les trois piles de départ.
// - q1, q2, q3 mémorisent les trois piles de départ et ne changent pas au cours
//   des appels. Utilisées pour l'affichage (on affiche q1, puis q2, puis q3).
// - n est le nombre de disques à déplacer.
// - max est l'espacement horizontal que prend l'affichage (utilisez max = n).
// - *count permet de compter les étapes réalisées.
void hanoi_rec(stack *p1, stack *p2, stack *p3,
               stack *q1, stack *q2, stack *q3,
               int n, int max, int *count);
```

3 Implémentation d'une file par un tableau « circulaire » dynamique

Les fichiers `queue.h` et `queue_by_circular_array.c` sont dédiés à cette implémentation. Le fichier `queue.h` **ne doit pas être modifié** : il contient les prototypes des primitives que vous devez écrire dans `queue_by_circular_array.c`.

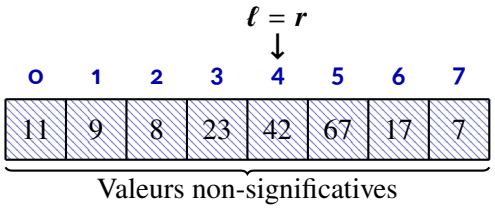
On rappelle le principe de l' implémentation d'une *file* par un tableau. Commençons par décrire la structure de données. On représente une file contenant k valeurs avec les objets suivants :

1. Un tableau dont la taille allouée est une puissance de deux supérieure ou égale à k.
2. La taille c de ce tableau (la lettre c est pour **c**apacité).
3. Deux indices du tableau appelés ℓ (pour « left ») et r (pour « right »). Il y aura trois cas possibles :
   - La file est vide et $\ell = r$.
   - La file n'est pas vide, $\ell < r$, et ses valeurs sont stockées aux indices $\ell, (\ell+1), \ldots, (r-1)$.
   - La file n'est pas vide, $r \leqslant \ell$, et ses valeurs sont stockées aux indices $\ell, (\ell+1), \ldots, (c-1), 0, \ldots, (r-1)$.
4. Un Booléen indiquant si la file est vide.

Observez que si $\ell = r$, la file peut être soit vide, soit pleine. Le Booléen indiquant si la file est vide sert à distinguer ces deux cas l'un de l'autre.

On représente cette structure de données graphiquement dans la Figure 2 ci-dessous.

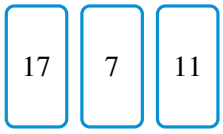
Cas vide : on a $\ell = r$

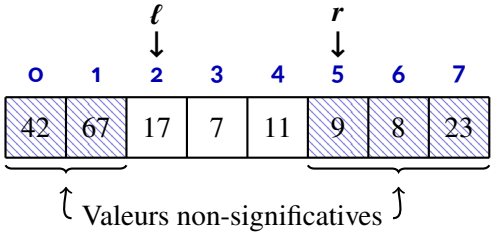


File vide à représenter ($k = 0$)

Représentation par un tableau de taille $c = 8$.

Cas non-vide avec $\ell < r$

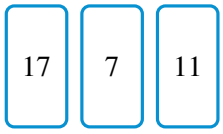


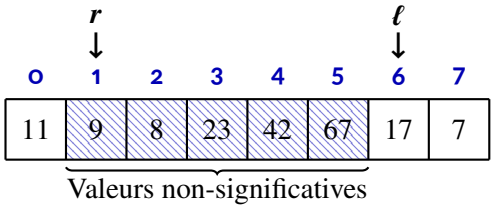


File à représenter ($k = 3$)

Représentation par un tableau de taille $c = 8$.

Cas non-vide avec $r \leqslant \ell$





File à représenter ($k = 3$)

Représentation par un tableau de taille $c = 8$.

FIGURE 2 : Représentation d'une file en utilisant un tableau

Nous utiliserons le type suivant, déjà défini dans `queue_by_circular_array.c` :

```
typedef struct {
  int *array;     // Tableau des valeurs.
  int capacity;   // Taille du tableau des valeurs.
  int left;       // Indice de la valeur à gauche de la file (si non-vide).
  int right;      // Indice qui suit celui de la valeur à droite de la file
                  // (si elle est non-vide), modulo capacity.
  bool empty;     // Booléen indiquant si la file est vide.
} queue;
```

Vous devez écrire les primitives du type « file » en utilisant cette structure de données. Comme pour les piles, on devra écrire deux fonctions auxiliaires qui serviront à doubler ou à diviser par deux la taille du tableau utilisé dans la représentation. Elles devront être appelées *au bon moment* par vos implémentations des primitives. Leur spécification est la suivante. Voir le fichier `queue_by_circular_array.c` pour plus d'information sur le moment auquel il faut les appeler.

```
// Double la taille du tableau utilisé dans la représentation.
static void grow_queue(queue *);

// Divise par deux la taille du tableau utilisé dans la représentation.
static void shrink_queue(queue *);
```

Pensez à tester votre code *régulièrement*

À nouveau, vous devez écrire des tests pour vérifier votre implémentation. Pour cela, vous disposez d'une primitive qui permet d'afficher une file sur la sortie standard :

```
void print_queue(queue *);
```

Cette fonction est déjà implémentée dans le fichier `queue_by_circular_array.c`.

4 Implémentation d'une file par deux piles

La seconde implémentation type « file » est celle vue en cours d'algorithmique, qui utilise *deux piles* :

1. La première pile contient les valeurs se trouvant « au début » de la file dans l'*ordre* (la première valeur de la file est celle se trouvant au sommet de la pile et on suit le sens de la file en descendant dans la pile).
2. La seconde pile contient les valeurs restantes de la file *en ordre inverse* (la dernière valeur de la file se trouve au sommet de la pile et on remonte le sens de la file en descendant dans la pile).

La Figure 3 qui décrit la représentation par deux piles d'une file contenant six éléments.

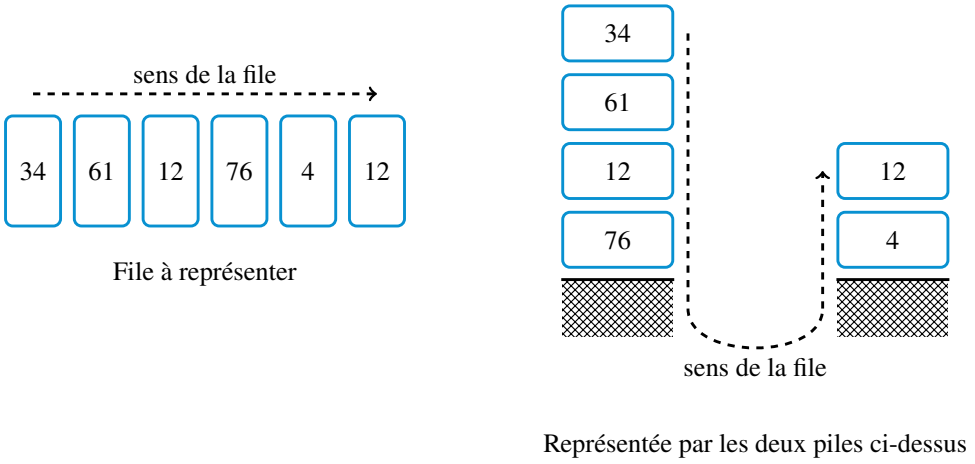


FIGURE 3 : Représentation d'une file en utilisant deux piles

Dans cette représentation, l'endroit où la file est « coupée » n'est pas significatif. Cependant, veillez à obtenir une complexité de chaque opération en $O(1)$ en coût amorti.

Comme cette seconde version de la bibliothèque implémentant les primitives des files est basée sur les piles, il faut bien entendu que votre implémentation des piles soit correcte (celle du fichier `stack_by_array.c`). L'autre conséquence est que vous n'avez pas accès aux champs de la structure `stack` : vous ne devrez utiliser **que** les fonctions dont les prototypes sont décrits dans `stack.h`. L'implémentation se fait dans le fichier `queue_by_2_stacks.c` (le fichier en-tête ne change pas : c'est `queue.h`, à ne pas modifier).

Rappel pour la compilation

- Choisir quelle implémentation est compilée se fait en basculant la macro `QUEUE_BY_TWO_STACKS` à `0` ou `1`.
- À chaque fois que vous changez d'implémentation, nettoyez par `make clean`, ou compilez par `make -B`.

5 Fonctions de test sur les files

Nous allons enfin écrire quelques fonctions pour tester notre implémentation des files. Les prototypes sont écrits dans le fichier `queue_use.h`. Vous devez écrire les fonctions elles-mêmes dans `queue_use.c`.

Un type abstrait est indépendant de son implémentation

Comme pour les piles, votre code doit être indépendant de l'implémentation des files écrite en section 3. Vous ne devez utiliser **que** les primitives décrites dans `queue.h`.

La liste des fonctions à écrire est la suivante :

```c
// Renvoie le nombre d'éléments dans la file, sans la modifier.
int getsize_queue(queue *);

// Crée une file aléatoire de valeurs entre 0 et maxval de taille size
// Si size est inférieure ou égal à 0, la file construite sera vide.
queue *random_queue(int, int);

// Crée une copie d'une file.
queue *copy_queue(queue *);

// Crée la file suivante :
//   1 2 3 4 .... n n .... 4 3 2 1
// Si n est inférieur ou égal à 0, la file construite est vide.
queue *mountain_queue(int);

// Trie la file par ordre croissant (la plus grande valeur à droite).
// L'algorithme devra être basé sur le principe du tri par sélection.
// Le principe de ce tri est le suivant :
// - On cherche le minimum de la file et on le place à gauche.
// - On cherche le minimum du reste de la file et on le place à sa droite.
// - On continue jusqu'à ce que la file soit triée.
// Voir https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html ou
// https://visualgo.net/en/sorting pour une visualisation de cet algorithme.
//
// La file doit être triée sur place, c'est-à-dire que la file passée en
// paramètre doit être modifiée. Aucune structure auxiliaire n'est autorisée,
// et la récursivité n'est pas autorisée non plus.
void select_sort_queue(queue *);
```