# TP7 – Structures de données: files et arbres binaires

Programmation en C (LC4)

Semaine du 10 mars 2014

## 1 Allocation mémoire manuelle

Dans cet exercice, nous allons implémenter une méthode d'allocation mémoire alternative à celle proposée par la librairie standard avec `malloc(3)`. Pour cela, nous allons allouer statiquement un grand tableau dont nous distribueront dynamiquement des morceaux.

Nous stockeront les informations sur chaque bloc de mémoire qui a été alloué ou qui est allouable à l'aide d'un en-tête positionné au début du bloc mémoire, composé de deux champs :

— `int free`, un booléen indiquant si le bloc est libre ou occupé,
— `int size`, indiquant la taille du bloc.

Initialement, il n'y aura aucun bloc : le tableau statique est vide. Le premier appel à la fonction d'allocation devra donc initialiser le premier bloc du tableau, marqué comme libre et de taille toute la mémoire disponible dans le tableau (moins la taille de l'en-tête). On utilisera une variable statique pour déterminer s'il s'agit du premier appel à la fonction d'allocation.

La mémoire pointée par notre grand tableau source sera à terme organisée ainsi :

| `libre` | `taille` | `zone memoire` | `libre` | `taille` | `zone memoire` | `libre` | `...` |
|---|---|---|---|---|---|---|---|

Lors d'une allocation dynamique, on cherchera un bloc (ou une suite de blocs) libre(s) ayant suffisamment de place, qu'on transformera en un bloc occupé (en ajoutant éventuellement un bloc marqué libre à la suite si la quantité demandée ne correspond pas exactement à la taille totale des blocs libres utilisés).

**Question 1.** Créez trois fichiers `alloc.c`, `alloc.h` et `main.c`. Les implémentations de fonctions liées à notre allocation de mémoire devront toutes se faire dans le fonction `alloc.c`, et les fonctions utiles à l'utilisateur de notre méthode devront être déclarées dans `alloc.h`.

**Question 2.** Définissez dans le fichier approprié une constante `N` correspondant à la taille (en octets) du tableau servant de source de mémoire, et définissez ce tableau : `static char memoire[N];`.

**Question 3.** Définissez la structure de bloc mémoire telle que décrite ci-dessus, qu'on appellera `struct memblock`.

**Question 4.** Écrivez une fonction `void* search_free_blocks(int x)` qui cherche dans les zones mémoires allouées un emplacement libre d'au moins `x` octets dans un ou plusieurs blocs consécutifs. S'il n'y en a pas, cette fonction retournera `NULL`.

**Question 5.** Écrivez une fonction `void *my_alloc(int x)` qui alloue `x` octets avec notre méthode et renvoie l'adresse de début de la zone du tableau allouée, en utilisant la fonction `search_free_blocks`. S'il n'y a plus assez d'espace disponible, la fonction retournera `NULL`.

**Question 6.** Écrivez une fonction `void my_free(void *)`, notre version de `free(3)`. Attention au fait que l'adresse connue du programmeur n'est pas l'adresse de début du bloc, mais de la zone mémoire libre (il faut donc retrancher la taille de l'entête).

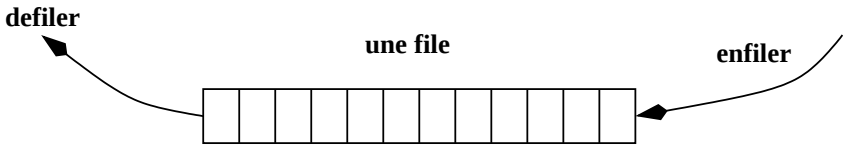
**Question 7.** Testez ces fonctions à l'aide d'appels appropriés dans `main.c` (allouez des tableaux d'entiers de plusieurs tailles différentes, désallouez-les, recommencez...). Vous pourrez même essayer de l'utiliser dans l'exercice suivant.

**Question 8.** BONUS : La fonction `search_free_blocks()` a tendance à fragmenter la mémoire. Écrivez une fonction `void *search_best_free_blocks(int x)` qui cherche la plus petite zone mémoire contenant `x` octets.

**Question 9.** BONUS : Écrivez `my_realloc()` qui se comporte comme `realloc(3)`. Si à la suite du bloc à réallouer se trouve miraculeusement un ou plusieurs blocs libres que l'on peut récupérer, c'est parfait on les concatène. Sinon il faut allouer un nouveau bloc, copier le contenu existant, et libérer l'ancien bloc.

## 2 Files

Une file est une structure de donnée dans laquelle les premiers éléments ajoutés sont les premiers à être récupérés (comme dans une file d'attente) :




Nous choisissons ici de représenter une liste par un tampon circulaire, défini par la structure `file_s` et de type `file_t` suivants :

```
struct file_s {
     size_t debut, fin, capacite;
     int *elements;
 };
 typedef struct file_s file_t;
```

Une file d'entiers sera implémentée (cf. figure) en stockant tous les éléments de la file dans le tableau pointé par le champ `elements`. Ce tableau aura pour taille la valeur indiquée par le champ `capacite`.

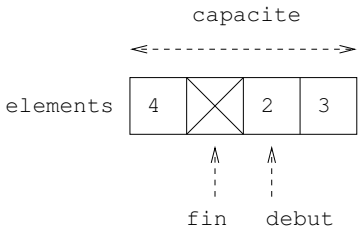
**Question 10.** Écrire les fonctions

```
file_t *alloue_file(size_t capacite);
void libere_file(file_t *file);
```

qui, respectivement,

— alloue et initialise une file pouvant contenir `capacité - 1` éléments (La fonction renverra NULL si l'allocation mémoire échoue ou si `capacité` vaut 0) : on initialisera les champs `debut` et `fin` à `0` et le tableau d'éléments devra contenir autant de cases que `capacité`,
— libère l'espace mémoire occupé par une file et ses éléments.

L'élément au début de la file se trouve dans la case d'indice `debut` et l'élément à la fin de la file se trouve dans la case précédant la case d'indice `fin`. La file est vide lorsque les indices de tableau `debut` et `fin` sont égaux. Le nombre d'éléments de la file est égal au nombre de cases utilisées du tableau, et on verra plus loin qu'après un certain nombre d'opérations sur la file, on peut avoir `fin < debut` (c'est le cas sur la figure suivante où la taille de la file est 3).



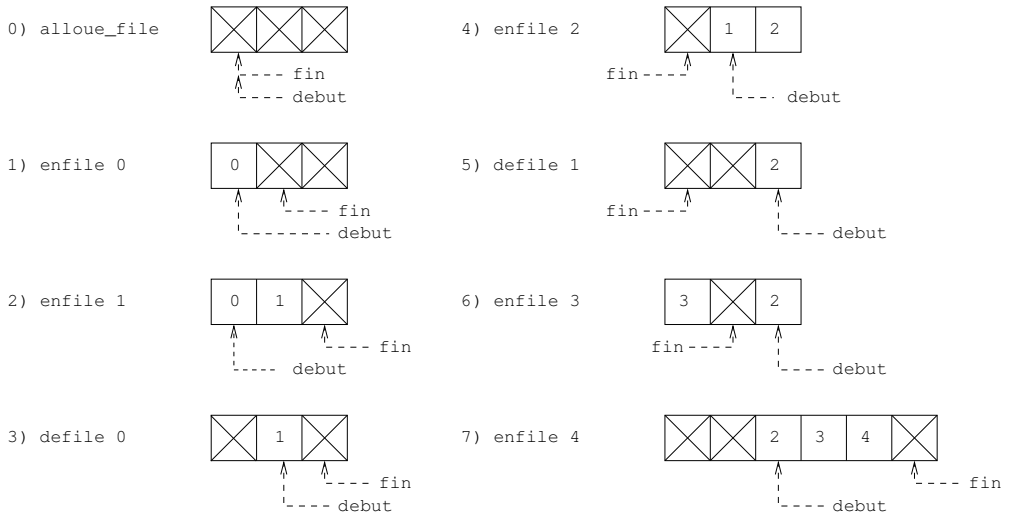

**Question 11.** Écrire les fonctions

```
int est_vide(file_t *file);
size_t taille(file_t *file);
```

qui, respectivement,

— indique si une file donnée en paramètre est vide,
— indique le nombre d'éléments dans la file.

Dans la figure suivante, on montre comment les indices `debut` et `fin` ainsi que le tableau pointé par `elements` sont modifiés par différentes opérations successives sur une file. Lorsqu'on veut ajouter (enfiler) un élément à la fin de la file, on place l'élément dans la case indicée par `fin` et on décale l'indice de fin vers la droite : s'il ne reste qu'une case disponible, on redimensionne le tableau avec `realloc()` à deux fois sa taille initiale (et on déplace éventuellement les éléments de la file). Lorsqu'on veut enlever (défiler) le début de la file, on décale l'indice de début vers la droite. Lors de ces opérations de décalage, on devra faire attention à toujours rester dans le tableau, quitte à passer de la dernière à la première case.




**Question 12.** Écrire les fonctions

```
int enfile(file_t *file, int n);
int defile(file_t *file, int *a);
```

qui, respectivement,

- enfile un entier `n` (la fonction renverra `0` si l'opération se termine avec succès et `-1` si l'éventuelle (ré)allocation mémoire a échouée),
- défile l'entier au début de la file et le place à l'adresse de l'argument `a`. La fonction retourne `0` en cas de succès, et `-1` en cas d'échec (file vide).