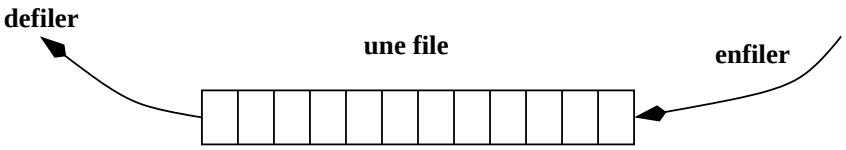
TP 7

Programmation en C (LC4)

Semaine du 19 mars 2012

1 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 allons utiliser la structure `file_s` et le 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`.

Exercice 1 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.

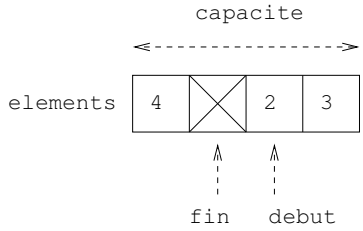
Solution:

```
file_t *alloue_file(size_t capacite) {
    file_t *file;
    if (!capacite)
        return NULL;
    if ((file = malloc(sizeof(struct file_s))) == NULL)
        return NULL;
    if ((file->elements = malloc(capacite * sizeof(int))) == NULL) {
        free(file);
```

```
        return NULL;
    }
    file->debut = file->fin = 0;
    file->capacite = capacite;
    return file;
}

void libere_file(file_t *file) {
    free(file->elements);
    free(file);
}
```

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).



Exercice 2 Écrire les fonctions

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

qui, respectivement,

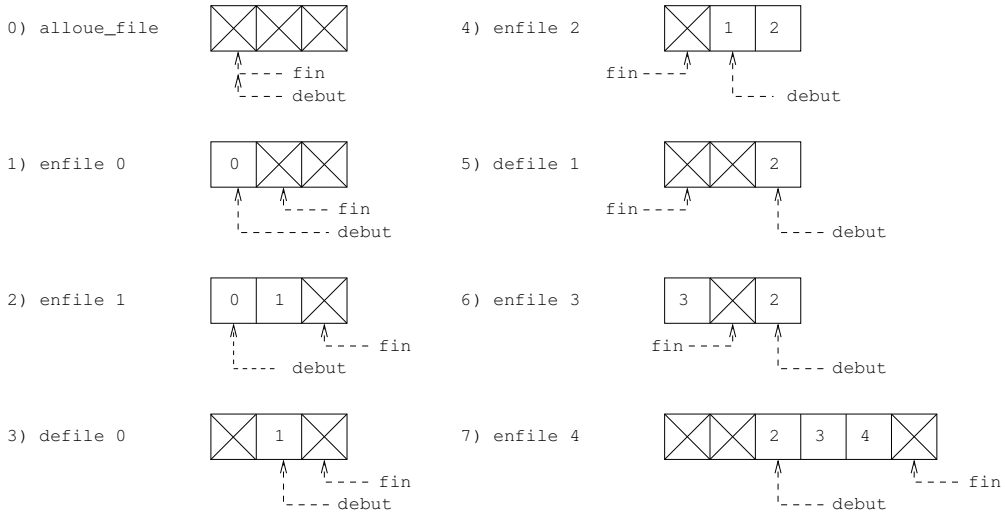
- indique si une file donnée en paramètre est vide
- indique le nombre d'éléments d'une la file

Solution:

```
int est_vide(file_t *file) {
    return (file->debut == file->fin);
}

size_t taille(file_t *file) {
    if (file->fin < file->debut)
        return ((file->capacite - file->debut) + file->fin);
    else /* file->fin >= file->debut */
        return (file->fin - file->debut);
}
```

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. Les indices `debut` et `fin` sont autorisés à dépasser la fin du tableau pour revenir au début. 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.



Exercice 3 Écrire les fonctions

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

qui, respectivement,

- enfile un entier `n` (la fonction renverra `n` si l'opération se termine avec succès et une valeur différente de `n` si l'éventuelle (ré)allocation mémoire a échouée)
- défile l'entier au début de la file et le place dans l'entier passé en argument par pointeur (la fonction renverra NULL si la file est vide).

Solution:

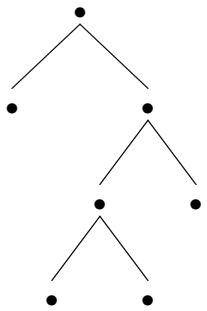
```
int enfile(file_t *file, int n) {
    size_t i;
    int *t;
    if (taille(file) == file->capacite - 1) {
        if ((t = realloc(file->elements,
                         2 * file->capacite * sizeof(int))) == NULL)
            return 0;
        file->elements = t;
        if (file->fin < file->debut) {
            for (i = 0; i < file->fin; i++)
                file->elements[file->capacite + i] = file->elements[i];
            file->fin += file->capacite;
        }
        file->capacite *= 2;
    }
    file->elements[file->fin] = n;
    file->fin++;
    if (file->fin == file->capacite)
        file->fin = 0;
    return 1;
}

int defile(file_t *file, int *n) {
    if (est_vide(file))
        return 0;
    *n = file->elements[file->debut];
    file->debut++;
    if (file->debut == file->capacite)
```

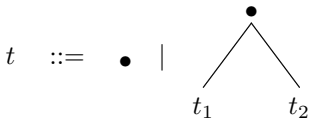
```
        file->debut = 0;
    return 1;
}
```

2 Arbres binaires

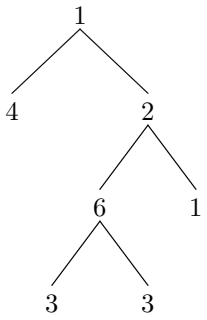
Un arbre binaire c'est un truc comme ça :



De façon générale un arbre binaire t c'est soit une feuille, soit un nœud auquel sont rattachés deux sous arbres t_1 et t_2 :



On peut aussi étiqueter les nœuds avec des entiers :



Pour représenter de tels objets en C on va utiliser la structure de donnée suivante :

```
struct noeud {
  int val; // etiquette
  struct noeud *g; // sous arbre gauche
  struct noeud *d; // sous arbre droit
};

typedef struct noeud *arbre;
```

Un nœud possède une étiquette `val` de type `int` et deux pointeurs vers ses sous arbres (gauche et droit). Les pointeurs auront tous les deux la valeur `NULL` lorsque le nœud est une feuille.

Exercice 4 Écrivez une fonction `arbre feuille(int val)` qui crée un arbre constitué d'une feuille étiquetée par la valeur `val`.

Exercice 5 Écrivez une fonction `arbre combine(int val, arbre t_g, arbre t_d)` qui crée l'arbre constitué d'un nœud étiqueté par `val` qui admet pour sous arbres `t_g` et `t_d`.

Exercice 6 Écrivez une fonction `void free_arbre(arbre t)` qui libère la mémoire allouée pour l'arbre `t`.

Exercice 7 Écrivez une fonction `int somme(arbre t)` qui renvoit la somme des étiquettes d'un arbre `t`.

Exercice 8 Écrivez une fonction `void print_arbre(arbre t, int indent)` qui affiche l'arbre `t`. L'arbre donné en exemple sera affiché de la façon suivante (pour `indent` = 0) :

```
1
 4
 2
  6
   3
   3
  1
```

Le paramètre `indent` sert lors des appels récursifs à rajouter le nombre correspondant d'espaces en tête de ligne.