Dans ce TP, on se propose d'implémenter nous même les fichiers `file.c` et `tasmin.c` du TP précédent. Un fichier `tp06.zip` est disponible sur la page web du cours, contenant les fichiers suivants :

- des fichiers `file.c` et `tasmin.c` dont le corps des fonctions sont à compléter ;
- les fichiers `file.h` et `tasmin.h` associés (déjà remplis) ;
- un fichier `graphes.c` contenant la correction du TP précédent.

Pour tester vos fonctions, vous pourrez tester que les tests du TP précédent fonctionnent avec votre implémentation des files (resp. des files de priorité), avec la commande suivante :

```
gcc graphes.c file.c tasmin.c -o exec -lm && ./exec
```

1 Structure de file

On rappelle qu'une **file** est une structure de données FIFO (First In First Out). Une implémentation classique des files est faite à partir de listes chaînées. Pour cela, on fournit les deux structures suivantes :

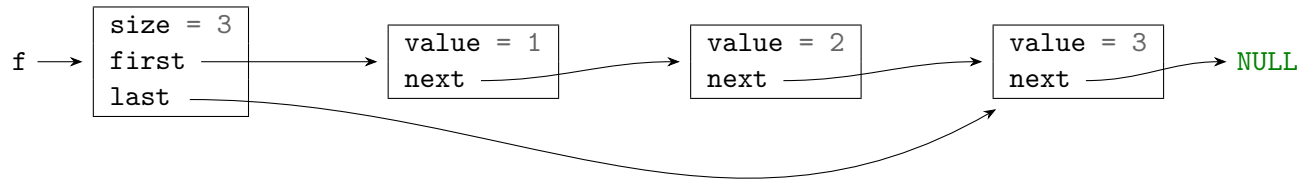
```
struct maillon{
  int value;
  maillon* next;
};

struct file{
  int size;
  maillon* first;
  maillon* last;
};
```

Nous allons principalement manipuler des pointeurs vers des files. Considérons une variable `file* f;` :

- si elle représente une file vide, alors `f->first` et `f->last` valent `NULL` ;
- sinon, `f->first` pointe vers le premier maillon de la liste chaînée (utile pour sortir le premier élément), et `f->last` pointe vers le dernier maillon (utile pour rajouter un élément à la fin).

Par exemple, voici un schéma de la mémoire utilisée par une file contenant les valeurs `{1, 2, 3}` :



1. Implémenter les fonctions suivantes en complétant le fichier `file.c` :

file.h
```
file* creer_file(void); // crée une file vide sur le tas
int file_taille(file* f); // renvoie la taille d'une file
bool file_est_vide(file* f); // teste si la file est vide
void file_push(file* f, int x); // rajoute la valeur x en queue de la file
int file_peek(file* f); // renvoie la valeur en tête de la file (sans modifier f)
int file_pop(file* f); // renvoie la valeur en tête de la file (et l'enlève de f)
void free_file(file* f); // libère la mémoire utilisée par la file
```

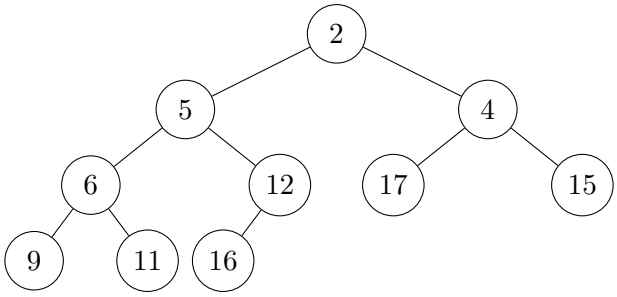
Vous pouvez désormais tester le premier algorithme de calcul des distances du fichier `graphes.c`.

2 Structure de tas-min

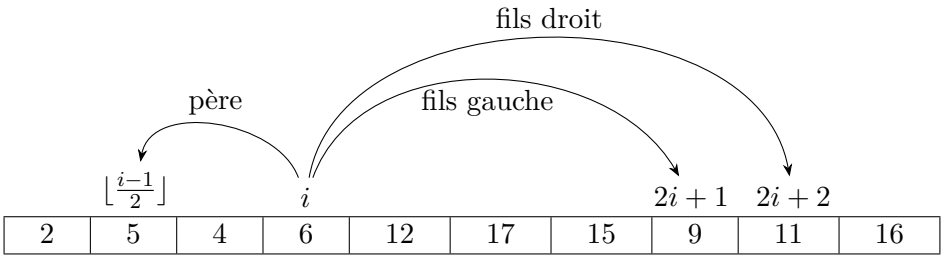
On rappelle qu'une structure de file de priorité-min peut être implémentée par un tas-min, qui est un arbre t vérifiant les propriétés suivantes :

- t est un arbre binaire complet à gauche, stockant des éléments avec leur priorité associée ;
- pour tout nœud n de t, pour tout fils f de n, $\text{prio}(n) \leq \text{prio}(f)$.

En particulier, la racine de t possède la plus petite priorité. Voici un exemple d'un tel arbre, où l'on n'a représenté que les priorités (et pas les éléments associés) :



On rappelle également que l'intérêt d'avoir un arbre binaire complet à gauche est de pouvoir le stocker dans un tableau où les éléments sont stockés dans l'ordre d'une énumération des nœuds de t par un parcours en largeur. Dans un tel tableau, on peut passer d'un nœud à son père ou à l'un de ses fils avec les relations rappelées sur le schéma ci-dessous :



2. Implémenter les fonctions utilitaires suivantes en complétant le fichier `tasmin.c`.

fonctions utilitaires

```c
int fg(int i); // renvoie l'indice du fils gauche de i
int fd(int i); // renvoie l'indice du fils droit de i
int pere(int i); // renvoie l'indice du père de i
void echanger(elem* a, int i, int j); // échange les cases d'indices i et j de a
```

On fournit une structure suivante pour représenter un tas-min (où les éléments stockés sont des entiers, et leurs priorités sont des flottants) :

tas-min

```
struct elem{
  int val;
  float prio;
};

struct tasmin{
  int capacite;
  int nb_elem;
  elem* tab;
};
```

Nous allons principalement manipuler des pointeurs vers des tas-min. Considérons une variable `tasmin* t;` :

- `t->tab` est un tableau stockant le tas selon le schéma précédent, tel que :
  - `t->nb_elem` est le nombre d'éléments stockés dans le tas ;
  - `t->capacite` est la taille de `t->tab` (utile pour savoir si le tableau est plein).

3. Implémenter les fonctions suivantes en complétant le fichier `tasmin.c` :

tasmin.h

```
tasmin* creer_tasmin(void); // crée un tas-min vide, de capacité initiale 42
int tasmin_taille(tasmin* t); // renvoie le nombre d'éléments stockés dans le tas-min
bool tasmin_est_vide(tasmin* t); // teste si tas-min est vide
int tasmin_peek(tasmin* t); // renvoie l'élément de priorité minimale, en O(1)
void free_tasmin(tasmin* t); // libère la mémoire utilisée par le tas-min
```

4. Implémenter les fonctions suivantes en complétant le fichier `tasmin.c` :

Si la priorité d'un nœud est modifiée, la propriété de tas-min n'est plus nécessairement vérifiée :

- si la nouvelle priorité du nœud est trop basse par rapport à celle de son père, il faut faire **remonter** le nœud ;
- si la nouvelle priorité du nœud est trop élevée par rapport à celle d'un de ses fils, il faut faire **descendre** le nœud.

On rappelle ici les deux algorithmes permettant d'effectuer ces opérations :

Algorithme 1 : `monter_noeud`

Données : Un (presque) tas-min t, un nœud i
si *i n'est pas la racine* ***et*** $\text{prio}(\text{pere}(i)) > \text{prio}(i)$ **alors**
  échanger les étiquettes de i et $\text{pere}(i)$;
  `monter_noeud`$(t, \text{pere}(i))$

Algorithme 2 : `descendre_noeud`

Données : Un (presque) tas-max t, un nœud i
$j \leftarrow i$;
si *i a un fils gauche g* ***et*** $\text{prio}(g) < \text{prio}(i)$ **alors**
  $j \leftarrow g$;
si *i a un fils droit d* ***et*** $\text{prio}(g) < \text{prio}(j)$ **alors**
  $j \leftarrow d$;
si $j \neq i$ **alors**
  échanger les étiquettes de i et j ;
  `descendre_noeud`(t, j)

Rappel : si n est le nombre d'éléments du tas-min, on peut tester si un nœud d'indice i possède un fils gauche (resp. fils droit) en testant si `fg(i) < n` (resp. `fd(i) < n`).

5. Implémenter les fonctions suivantes, en complétant le fichier `tasmin.c` :

monter et descendre

```c
/* remonte i dans un tableau a */
void monter_noeud(elem* a, int i);

/* descend i dans un tableau a ayant n éléments */
void descendre_noeud(elem *a, int n, int i);
```

On peut désormais implémenter les dernières opérations sur notre tas-min :

- pour **rajouter** un nœud dans le tas-min, le le rajoute dans la première case libre du tableau, et on fait **remonter** ce nouveau nœud ;
- pour **supprimer** la racine du tas-min, on place le dernier nœud du tableau à la racine, et on fait **descendre** ce nœud.

6. Implémenter les fonctions suivantes, en complétant le fichier `tasmin.c` :

tasmin.h

```c
/* rajoute un nouvel élément de valeur x et de priorité p dans le tas-min,
   et réorganise le tas-min en O(log n) */
void tasmin_push(tasmin* t, int x, float p);

/* supprime l'élément du tas-min de priorité minimale et renvoie sa valeur ;
   et réorganise le tas-min en O(log n) */
int tasmin_pop(tasmin* t);
```

Vous pouvez désormais tester les algorithmes Dijkstra et $A^\star$ du fichier `graphes.c`.