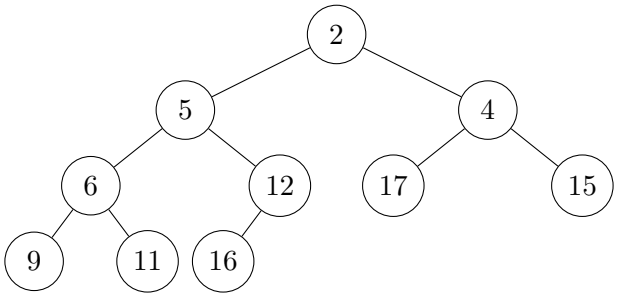
Un fichier `fenelon_tp15.zip` est à télécharger sur la page web du cours et à extraire sur votre ordinateur dans votre dossier de travail.

1 Structure de tas-min

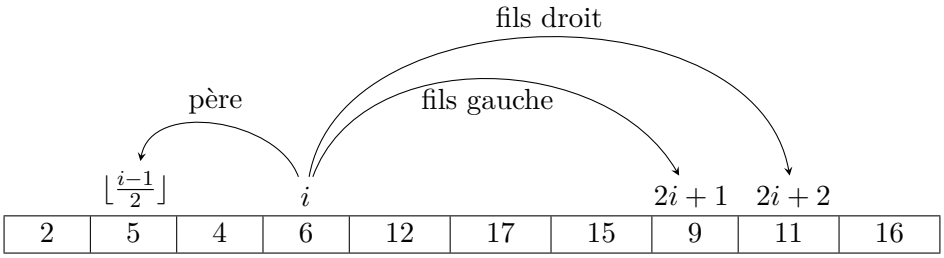
On rappelle qu'une structure de file de priorité-min peut être implémentée par un tas-min, qui est un arbre t vérifiant les propriétés suivantes :

- 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 :



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

fonctions utilitaires

```
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 {
  value_t val;
  prio_t 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).

2. 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 le tas-min est vide
value_t 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
```

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*** $\text{prio}(g) < \text{prio}(j)$ **alors**
  $j \leftarrow g$;
si *i a un fils droit d* ***et*** $\text{prio}(d) < \text{prio}(j)$ **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`).

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

monter et descendre

```
/* 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.

4. 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, value_t x, prio_t p);

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

Le fichier `graphes.c` implémente des algorithmes que vous allez voir plus tard dans l'année (Dijkstra et $A^\star$) qui utilisent une structure de file de priorité-min, elle même pouvant être implémentée par un tas-min.

5. Tester votre implémentation des tas-min en essayant de compiler et d'exécuter vos fichiers :

```
$ gcc tasmin.c graphes.c -lm -o test; ./test
Test Dijkstra
distance 0--2 : 9.000000
2 <- 1 <- 4 <- 0

Test A*
distance 0--2 : 9.000000
2 <- 1 <- 4 <- 0

Test : mauvaise heuristique
distance 0--2 : 11.000000
2 <- 1 <- 0
```

2 Tri par tas

On peut se servir de notre structure de tas pour obtenir un algorithme de tri efficace d'un tableau :

- on crée un tas dont les valeurs et les priorités ont le même type que les éléments du tableau ;
- on ajoute chaque élément du tableau dans le tas (qui sert à la fois de valeur et de priorité) ;
- on sort un par un les éléments du tas en les rangeant dans cet ordre dans le tableau de départ.

6. Écrire une fonction `void tri_par_tas(int* tab, int n);` prenant en arguments un tableau d'entiers de longueur n et le modifiant en place par effets de bords pour qu'il contienne les mêmes éléments qu'au départ mais triés dans l'ordre croissant.

Exemple

```
$ gcc tasmin.c tri.c -o test; ./test
Tableau initial : [100, 1, 5, 3, 12, 0, 42, 25, 5, 99]
Tableau trié :    [0, 1, 3, 5, 5, 12, 25, 42, 99, 100]
```

7. Quelle est la complexité temporelle de votre fonction `tri_par_tas` dans le pire des cas ?

3 Fusion de fichiers triés

Remarque. Cette partie est moins guidée, elle est destinée à occuper les plus rapides d'entre vous.

On peut également se servir de notre structure de tas pour fusionner des fichiers déjà triés en un fichier qui contient l'intégralité des données des fichiers de départ, trié.

Par exemple, on peut considérer des fichiers contenant des lignes de la forme `note;numero_candidat` où `note` est un flottant et `numero_candidat` est un entier. On suppose que chaque fichier a été créé par un examinateur différent, et contient des données triées par notes croissantes. On souhaite alors obtenir le fichier contenant les informations de tous les candidats, lui aussi trié par notes croissantes.

Indication. On peut parser une telle ligne via la commande suivante, où `fichier` est de type `FILE*` et a été ouvert via la fonction `fopen` :

```
float note;
int numero_candidat;
fscanf(fichier, "%f;%d", &note, &numero_candidat);
```

8. Créer divers fichiers qui pourront servir d'exemples pour tester votre futur code.

3.1 Première approche

La première approche consiste à insérer toutes les entrées de tous les fichiers dans un même tas, puis de les sortir les unes après les autres du tas pour les écrire dans le fichier final.

9. Implémenter cette première stratégie.

3.2 Approche moins couteuse en mémoire

La première approche avait un défaut : la taille du tas va devenir très grande s'il y a beaucoup de candidats à traiter.

On considère alors une autre approche, qui permet de n'utiliser qu'un tas de capacité k s'il y a k fichiers à fusionner :

- on remplie le tas avec la première entrée de chaque fichier ;
- tant que le tas n'est pas vide, on sort l'élément le plus prioritaire du tas (qu'on écrit dans le fichier final), et on lit la prochaine entrée du fichier d'où il provient qu'on rajoute alors dans le tas.

10. Implémenter cette nouvelle stratégie. On pourra modifier le type `value_t` au besoin.