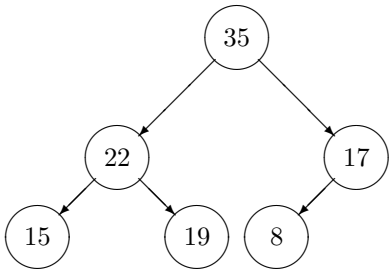
1 Position du problème

La structure de tas est une structure efficace et simple à concevoir qui implémente les files de priorité et un tri optimal : le tri par tas (en anglais *heapsort*). Une *file de priorité* est une structure de données regroupant des couples de la forme (c, v) où c est une *clé* (élément d'un ensemble totalement ordonné) et v une *information*. Les fonctions disponibles sur une file de priorité sont : l'insertion d'un nouvel élément ; l'extraction de l'élément ayant la plus grande clé (la priorité la plus importante) : structure FIFO. Comme on ne s'intéresse qu'aux performances de ces fonctions, on considèrera que chaque élément d'une file se réduit à sa clé.

2 La structure de tas

On étudie une mise en œuvre des files de priorité au moyen des *tas*, qui sont des arbres binaires particuliers (le terme *nœud* désigne aussi bien les nœuds internes que les feuilles).
Un *arbre parfait* est un arbre binaire dont tous les niveaux sont pleins, à l'exception éventuelle du dernier, dans lequel tous les nœuds doivent être situés le plus à gauche possible.
Un *arbre tournois* est un arbre binaire dont les nœuds sont étiquetés par des éléments d'un ensemble totalement ordonné, de telle façon que l'étiquette d'un nœud (autre que la racine de l'arbre) soit au plus égale à celle de son père.
Un *tas* est un arbre tournois parfait ; la figure ci-dessous présente un tas de hauteur 2.



Un (petit) tas.

Ainsi, la hauteur h d'un tas ayant n nœuds est exactement $E(\log_2 n)$. On mesurera les peformances des algorithmes qui suivent avec comme unité de complexité l'accès à un élément du tas (en lecture ou en écriture).

3 Tas versus arbre binaire

Évidemment, on peut coder en Caml un tas à l'aide d'une structure d'arbre binaire ; il existe néanmoins une implémentation plus simple et efficace à l'aide d'un simple vecteur `t` :

- la racine est logée dans la case (0) ;
- si un nœud est logé dans la case (i), son fils gauche est logé dans la case $(2i + 1)$ et son fils droit est logé dans la case $(2i + 2)$.

Remarque(s).

- si l'on considère un parcours en largeur de l'arbre, on retrouve la position des nœuds dans le vecteur (la racine étant numérotée 0),
- le père d'un nœud est stocké dans la case $E((i - 1)/2)$.

On utilise dans la suite les définitions Caml suivantes (`l` pour longueur du tas, `v` pour son vecteur contenu) :

```
(* d\'{e}finit le type tas *)
type tas = {mutable l : int; v : int vect};;

(* cr\'{e}e un tas pouvant accueillir jusqu'\`{a} n \'{e}l\'{e}ments *)
```

```
let tas_vide n = {l = 0 ; v = make_vect n -1};;

(* \'{e}change v.(i) et v.(j) *)
let echange v i j = let x = v.(i) in v.(i) <- v.(j) ; v.(j) <- x;;

(* acc\`{e}s dans le tas *)
let fils_gauche v i  = try v.(2*i+1) with _ -> -1;;
let fils_droit v i = try v.(2*i+2) with _ -> -1;;
let pere i = ((i-1) / 2);;
```

1° Pourquoi a-t-on mis le mot `mutable` devant l'identificateur `l` ? Pourquoi ne l'a-t-on pas mis devant l'identificateur `v` ?

2° Proposer deux fonctions Caml `tas_en_arbre` et `arbre_en_tas` qui passent de l'une à l'autre des structures.

4 Percolation

Les fonctions qui vont suivre nécessitent l'implémentation d'une opération élémentaire : la *percolation*. On appelle percolation l'opération qui consiste à réorganiser un arbre sous la forme d'un tas sachant que les deux sous-arbres de la racine sont déjà des tas : *i.e.* on met à sa place une racine de valeur incorrecte dans un tas.
La méthode consiste à descendre la valeur inappropriée de la racine r en cherchant sa nouvelle place dans le tas :

- si r est plus grand que ses deux fils, on a fini ;
- sinon on échange r avec le plus grand de ses deux fils.

3° Programmer la fonction `percolation`. Quelle en est la complexité ?

5 Suppression du maximum

La suppression du maximum d'un tas se fait en échangeant l'étiquette de la racine avec l'étiquette du dernier élément numéroté , en supprimant ensuite ce nœud et enfin en effectuant une percolation.

4° Écrire une fonction `supprime_racine`, dont le type est `tas -> int` qui extrait d'un tas (supposé non vide) son élément maximal.

6 Ajout d'un élément

Pour insérer un nouvel élément dans un tas, on le met aux feuilles à la première place disponible, puis on le fait remonter au besoin (s'il est plus grand que son père) pour conserver la structure de tas.

5° Écrire la fonction `promotion` qui complète la fonction d'insertion que voici :

```
let insere x tas =
  tas.v.(tas.l) <- x;
  promotion tas.v tas.l;
  tas.l <- tas.l +1 ;;
```

Quelle est la complexité d'une insertion ?

7 Application : le tri par tas

6° Expliquer comment, en utilisant un tas, on peut trier un vecteur de taille n pour un coût $\Theta(n \ln n)$. Si vous en avez le temps, programmer le tris par tas.