DM2 : Files de priorité

On utilisera le type suivant :

```
type arbre_bin =
  | Vide
  | Noeud of int*(arbre_bin)*(arbre_bin)
;;
```

On convient que la borne supérieur des étiquettes est $+1000000$ et la borne inférieur des étiquettes est -1000000. On pose :

```
let borne_sup = 1000000;;
let borne_inf = -1000000;;
```

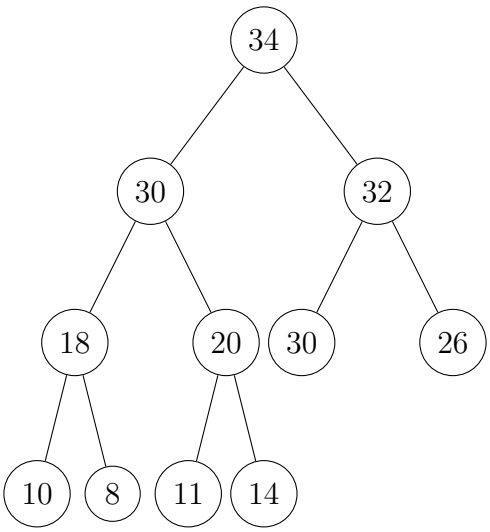
Pour se débarrasser du problème de l'étiquette de l'arbre vide, nous pouvons supposer que son étiquette est `borne_inf`.

Exercice 1 : Files de priorité représentées en arbres binaires

Définition. On considère un ensemble $\mathcal{X}$ ordonné. On appelle file de priorité dans $\mathcal{X}$ tout arbre $\mathcal{A}$ étiqueté par $\mathcal{X}$ tel que tout nœud ait une étiquette supérieure ou égale à celles de ses fils.

On représentera une file de priorité par un arbre binaire. Dans cet exercice, l'ensemble $\mathcal{X}$ utilisé est $\mathbb{N}$.

Exemple.



Partie A : premières fonctions

1. Définir l'exemple avec le type donné.
   On mémorisera le résultat dans la variable `ex_1`.
2. Écrire une fonction `etiquette` qui renvoie l'étiquette de la racine d'un arbre.

```
etiquette : arbre_bin -> int
```

3. Écrire une fonction `test_file` testant si un arbre binaire est une file de priorité.

```
test_file : arbre_bin -> bool
```

On pourra utiliser la fonction `etiquette`.

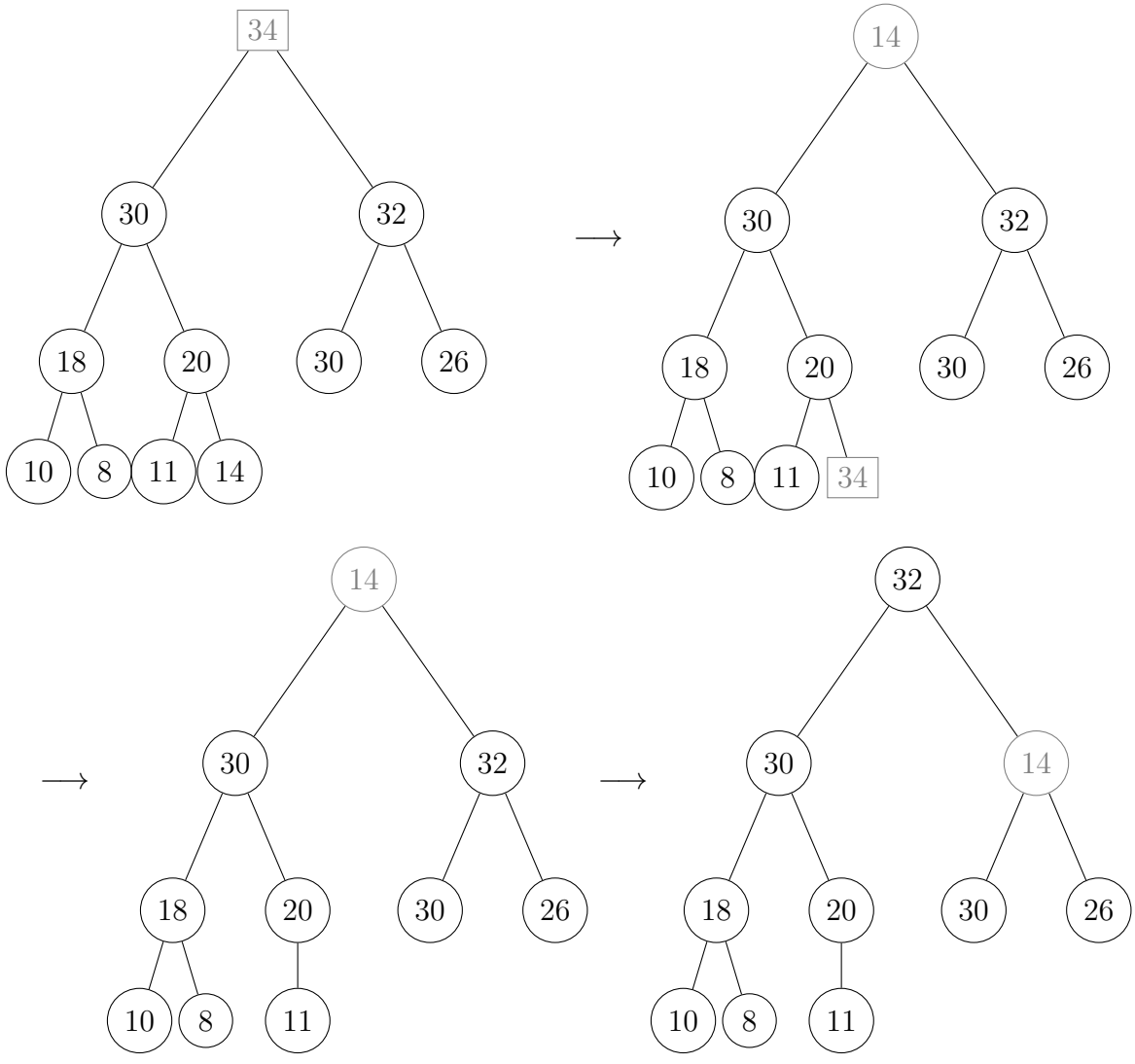
Partie B : suppression de la racine

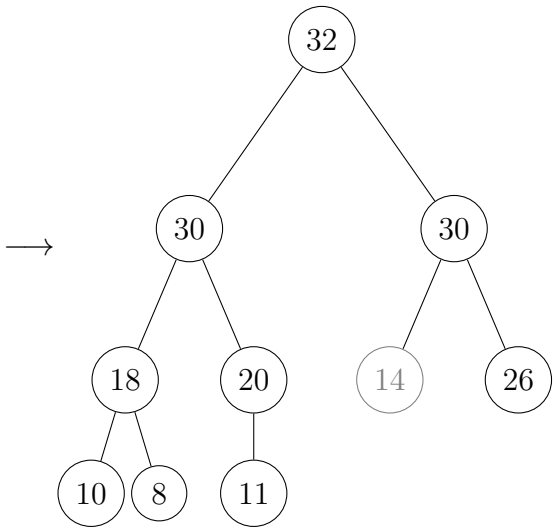
On souhaite supprimer la racine d'une file de priorité. Pour ce faire, nous allons procéder en plusieurs étapes :

— On échange la racine avec une feuille ;
— On supprime cette feuille ;
— On descend la nouvelle racine en l'échangeant avec le plus grand de ses fils tant qu'il en existe un strictement plus grand qu'elle. On a ainsi rétabli la structure de file de priorité.

Cette dernière procédure se nomme **percolation**.

Exemple : La suppression de la racine dans la file de priorité de l'exemple s'illustre ainsi :





4. Écrire la fonction `echange_etiquette` qui remplace l'étiquette de la racine d'un arbre par un élément x donné en entrée.

   ```
   echange_etiquette : arbre_bin -> int -> arbre_bin
   ```

   On utilisera la fonction `etiquette` et la fonction `echange_etiquette`.

5. Écrire la fonction `percolation` qui déplace une "mauvaise racine" dans la place qui lui convient en l'échangeant avec le plus grand de ses fils tant qu'il en existe un strictement plus grand qu'elle et qui retourne la file de priorité ainsi obtenue.

   ```
   percolation : arbre_bin -> arbre_bin
   ```

6. Écrire une fonction `supp_feuille` qui supprime un nœud terminal et renvoie le nouvel arbre, ainsi que l'étiquette de la feuille supprimée.
   - Si le nœud gauche n'est pas vide, on supprime un nœud terminal du fils gauche.
   - Sinon, si le nœud droit n'est pas vide, on supprime un nœud terminal du fils droit.

   ```
   supp_feuille : arbre_bin -> int * arbre_bin
   ```

7. Écrire une fonction `supp_racine` qui supprime la racine d'une file de priorité, en remplaçant son étiquette par l'étiquette d'un nœud terminal, supprime ce nœud terminal et effectue une percolation de la racine pour obtenir une file de priorité.

   ```
   supp_racine : arbre_bin -> arbre_bin
   ```

   On pourra utiliser les fonctions précédentes.

Partie C : insertion d'une nouvelle étiquette

On souhaite insérer une nouvelle étiquette x dans une file de priorité.

- Si l'arbre `a` est vide, on renvoie un arbre de racine étiquetée x, et avec deux fils vides.
- Sinon, si l'arbre est de la forme `Noeud(r, g, d)` :
  - Si x est strictement supérieure à la racine r, on insère récursivement la racine r dans l'un de ses fils choisi au hasard et on pose x comme nouvelle racine de l'arbre.

— Si x est inférieure à la racine, on garde r comme racine et on insère récursivement x dans l'un des fils g ou d choisi au hasard.
Le choix des arbres fils sera fait de façon aléatoire.

On rappelle qu'en OCaml, la commande `Random.int 2` renvoie un entier choisi au hasard entre 0 et 1.

8. Écrire une fonction `insere` qui insère une étiquette dans une place qui lui convient (selon l'algorithme décrit ci-avant) et qui retourne une file de priorité.

```
insere : int -> arbre_bin -> arbre_bin
```

9. En utilisant la fonction précédente, écrire une fonction `file_of_liste` qui convertit une liste en file de priorité.

```
file_of_liste : int list -> arbre_bin
```

Exercice 2 : Files de priorité représentées en tableaux (tas)

Définition. Un **tas** de hauteur h est un arbre binaire

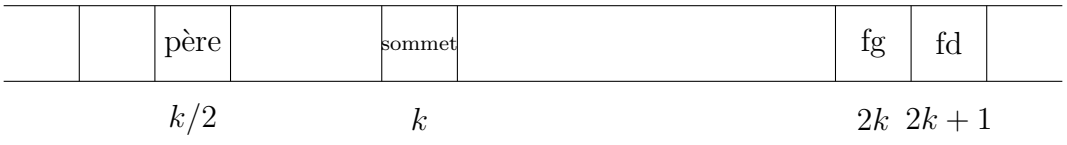
- **parfait** (condition sur le squelette) :
  — Toutes les feuilles sont de profondeur h ou $h-1$;
  — Pour tout $p < h$, il y a exactement 2^p nœuds de profondeur p ;
  — Toutes les feuilles de profondeur h sont "le plus à gauche" de l'arbre ;
- **tournoi** (condition sur les étiquettes) :
  — L'étiquette d'un nœud est supérieure ou égale à celle de ses fils.

Un tas est donc une file de priorité.

Exemple. L'arbre donné dans l'exercice 1 est un tas.

Implémentation par tableau :
Pour stocker un tas de taille n en mémoire, on utilisera un tableau de taille $N+1$ (avec N grand, pour pouvoir rajouter des éléments par la suite) avec en position 0 l'entier n qui code la longueur effective du tas et les éléments du tas entre les indices 1 et n, stockés de la façon suivante :

— Les enfants du nœud d'indice k sont d'indices $2k$ et $2k+1$.
— Le père du nœud d'indice k est d'indice $\lfloor \frac{k}{2} \rfloor$.



Exemple. L'arbre `ex_1` de l'exercice 1 peut être représenté par le tableau suivant de longueur 12 :

```
[|11; 34; 30; 32; 18; 20; 30; 26; 10; 8; 11; 14|]
```

mais aussi par le tableau suivant de longueur 16 :

```
[|11; 34; 30; 32; 18; 20; 30; 26; 10; 8; 11; 14; 0; 1; 2; 3|]
```

ou encore par le tableau suivant de longueur 19 :

`[|11; 34; 30; 32; 18; 20; 30; 26; 10; 8; 11; 14; 18; 0; 5; 3; 6; 41; 9|]`

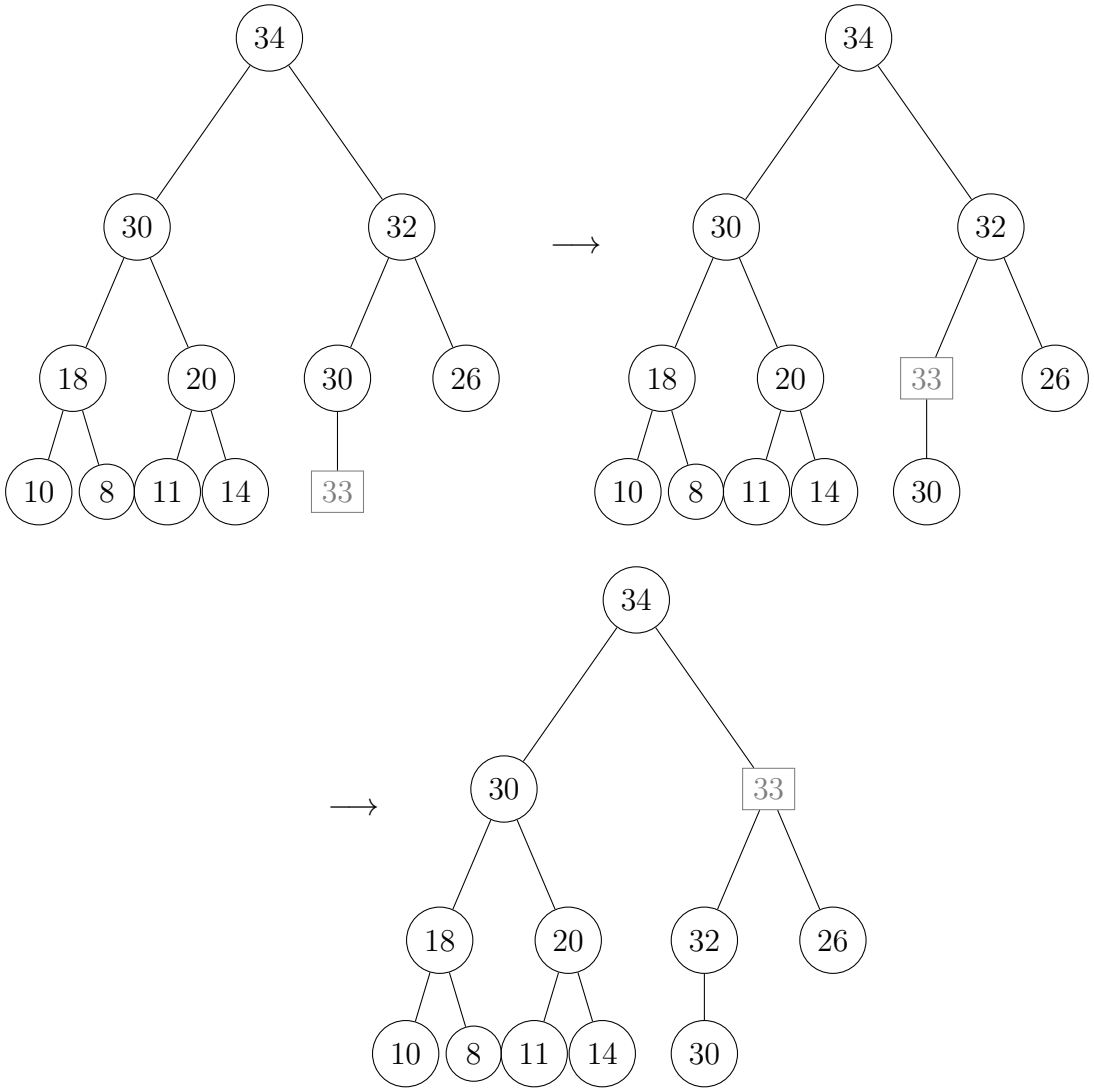
etc... Seuls les éléments d'indices 0 à `t.(0)` comptent !

1. Écrire une fonction `arbre_de_tas` qui transforme un tas codé par un tableau en une file de priorité codée par un arbre binaire de type `arbre_bin`.
2. Écrire sa fonction réciproque `tas_d_arbre`.
   On pourra commencer par définir une fonction `taille` qui renvoie la taille d'un arbre typé `arbre_bin` puis utiliser cette fonction `taille` dans la fonction `tas_d_arbre`.
3. Écrire une fonction `verification_tas` qui vérifie si un tableau correspond à un tas (c'est-à-dire qu'il représente bien une file de priorité).

Dorénavant, les tas (files de priorité) seront représentés par des tableaux.

4. **Insertion.**
   L'idée de l'insertion d'un élément dans un tas est la suivante :
   - On insère l'élément à ajouter en dernière position ;
   - On l'échange avec son père tant que son père est strictement plus petit que lui.

   Exemple. L'insertion de l'élément 33 dans le tas précédent s'illustre ainsi :



(a) Écrire une fonction `echange t i j` qui échange les contenus d'indice i et j du tas `t`.

(b) Écrire une fonction `remontee n t` qui remonte l'élément d'indice n du tas `t` à son bon emplacement dans la file de priorité.

(c) En déduire une fonction `insertion x t` qui insère le nouvel élément x dans la file de priorité `t`.

5. **Création.**

   En déduire une fonction `creation tab` qui crée un tas à partir d'un tableau de valeurs en y insérant les éléments au fur et à mesure.

6. **Suppression de la racine.**

   L'idée pour supprimer la racine d'un tas est la suivante :
   - On échange la racine avec la dernière feuille ;
   - On supprime cette dernière feuille ;
   - On descend la nouvelle racine en l'échangeant avec le plus grand de ses fils tant qu'il en existe un strictement plus grand qu'elle. On a ainsi rétabli la structure de tas.

   Cette descente est appelée "percolation".

   Exemple. On pourra reprendre l'exemple de l'exercice 1 sur la suppression de la racine.

   (a) Écrire une fonction `fils_max k t` qui renvoit l'emplacement du plus grand des deux fils de l'élément d'emplacement k.
   Si cet élément n'a pas de fils, on renverra k.

   (b) Écrire une fonction `descente t` qui effectue la percolation d'une racine "mal placée".

   (c) Écrire une fonction `suppression t` qui supprime la racine du tas `t`.

7. Justifier que les fonctions d'insertion et de suppression ont une complexité temporelle dans le pire des cas en $O(\log(n))$ où n est le nombre d'éléments présents dans le tas.

8. **Tri par tas.**

   La structure de tas permet d'implémenter un tri efficace d'un tableau : on transforme le tableau en tas, puis on effectue des extractions successives de la racine pour reformer le tableau trié.

   (a) Que renverra la fonction `tri_tas` appliquée au tableau `[|2; 8; 4; 9|]` ?

   (b) Écrire la fonction `tri_tas`.

   (c) Justifier que le tri par tas a une complexité temporelle dans le pire des cas en $O(n\log(n))$ où n est la longueur du tableau à trier.