DS 1

Devoir sur table

Le but de ce devoir est de détailler une implémentation de la structure de file de priorité avec des tas persistants.

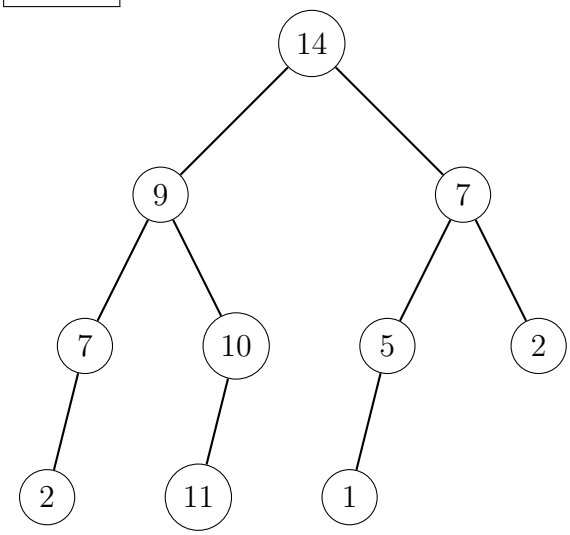
Définition d'un tas presque équilibré :

En Ocaml, nous utiliserons le type suivant :

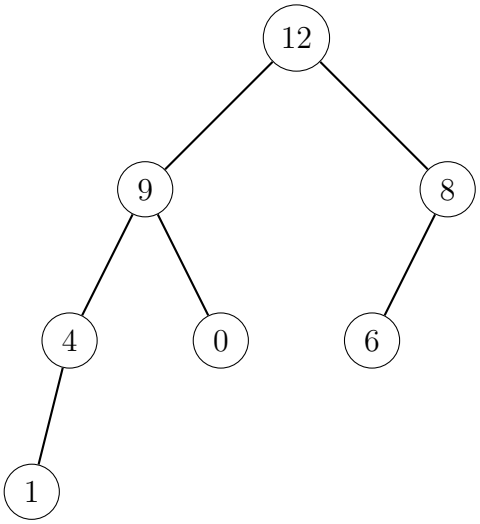
```
type 'a arbre = Vide | N of 'a arbre * 'a * 'a arbre ;;
```

Le nombre de nœuds $\mathcal{N}(a)$ d'un tel arbre est défini par induction :
$\mathcal{N}($ `Vide` $) = 0$, et $\mathcal{N}($ `(N(g,x,d))` $= 1 + \mathcal{N}($ `g` $) + \mathcal{N}($ `d` $)$.
La définition d'un arbre presque équilibré (en abrégé APE) est elle aussi inductive : un arbre est un APE s'il est vide, ou s'il est de la forme `N(g,x,d)` avec `g` et `d` deux APE vérifiant $\mathcal{N}(g) - 1 \leq \mathcal{N}(d) \leq \mathcal{N}(g)$. On parle de tas presque équilibré (en abrégé TPE) lorsque l'arbre satisfait de plus la propriété d'une file de priorité.

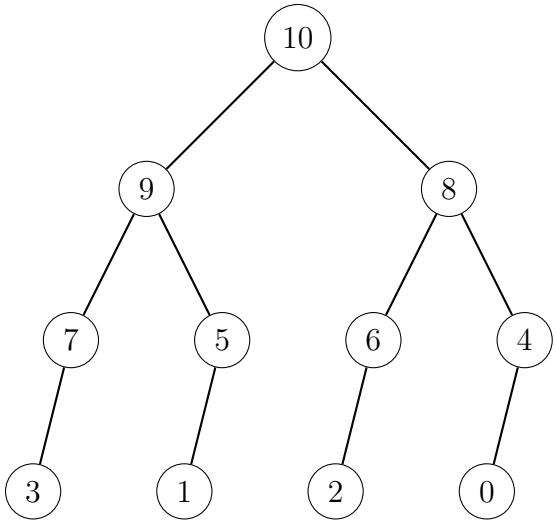
Q - 1 - Parmi les arbres suivants, quels sont les TPE ?



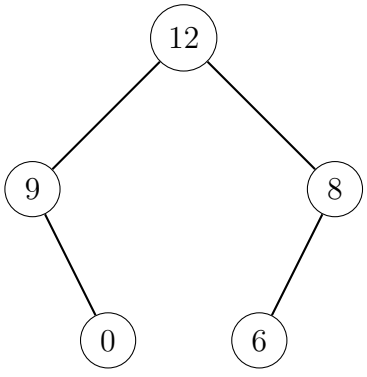
Exemple 1



Exemple 2



Exemple 3



Exemple 4

Q - 2 - Montrer qu'un APE non vide à n nœuds a une hauteur égale à $\lfloor \log_2(n) \rfloor$. (Pour rappel, la hauteur de l'arbre vide est -1).

Q - 3 - Écrire une fonction `verifie_tpe` prenant en entrée un arbre, et renvoyant le booléen associé au caractère TPE. On impose une complexité $O(n)$ avec n le nombre de nœuds. (Indication : utiliser une fonction auxiliaire `verif_taille` permettant d'obtenir le booléen et la taille. On pourra aussi écrire une fonction renvoyant la racine d'un arbre non vide).

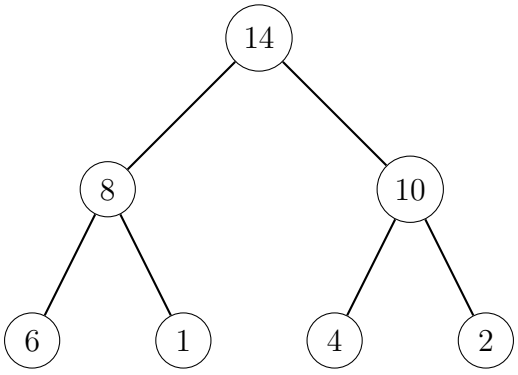
```
racine : 'a arbre -> 'a
verif_taille : 'a arbre -> bool * int
verifie_tpe : 'a arbre -> bool
```

Insertion d'un élément

On cherche a insérer un élément dans un TPE, tout en conservant le caractère initial . Pour l'insertion de x dans un arbre a, l'algorithme récursif est le suivant :

- Si l'arbre est vide, on renvoie l'arbre `N(Vide,x,Vide)`.
- Pour `a = N(g,r,d)`.
  - Si $x < r$, on insère x dans d et on échange le sous arbre gauche avec le sous arbre droit.
  - Sinon, x prend la place de la racine, puis on insère r dans d et on échange le sous arbre gauche avec le sous arbre droit.

Q - 4 - Dans l'exemple 5 suivant, procéder à l'algorithme puis dessiner le résultat de l'insertion de 15 dans l'arbre :



Exemple 5.

Q - 5 - Écrire la fonction `inserer` qui prend en argument un TPE et un élément x puis renvoie le TPE contenant en plus l'élément x.

```
inserer : 'a arbre -> 'a -> 'a arbre
```

Q - 6 - Prouver la correction de la fonction `inserer`.

Suppression d'un élément

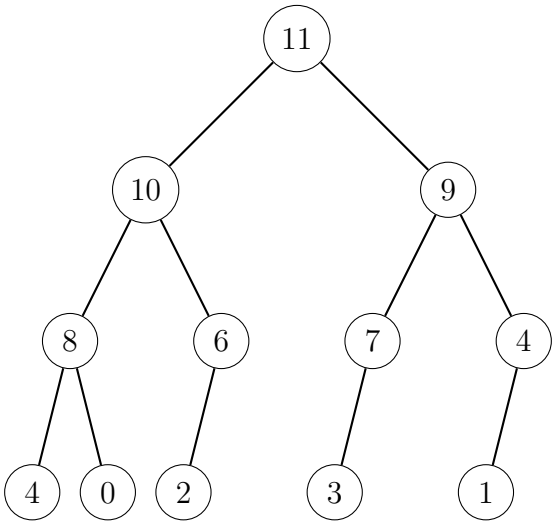
La question qui suit a pour but de faciliter la suppression de la racine.

Q - 7 - Écrire une fonction `modifier_racine a r` de type `'a arbre -> 'a -> 'a arbre`, prenant en entrée un TPE a, et remplaçant sa racine par r. On prendra garde à maintenir la structure de tas : en pratique on fait descendre la nouvelle racine r jusqu'à obtenir un tas. On impose une complexité $O(\log n)$ avec n le nombre de nœuds du TPE.

Q - 8 - Pour supprimer la racine d'un TPE a non vide, on procède de la façon suivante :

- on supprime le nœud le plus à gauche dans a et on permute les branches gauche et droite de chaque nœud ascendant du nœud supprimé. L'arbre obtenu est noté a', et l'étiquette supprimée e.
- on remplace l'étiquette r de a' par e et on reconstitue un APE a'' via la fonction de la question précédente.

On considère l'arbre a suivant :



Dessiner l'arbre a' puis l'arbre a'' décrit dans l'algorithme de suppression précédent.

Q - 9 - Écrire une fonction `supprimer_racine a` de type `'a arbre -> 'a * 'a arbre`, prenant en entrée un TPE et renvoyant sa racine et un TPE où cette racine a été supprimée. On impose une complexité $O(\log n)$ avec n le nombre de nœuds du TPE.

Tris par tas :

Le but de cette section est de construire le tri d'une liste à l'aide des TPE.

Q - 10 - Écrire la fonction `construction_tas` qui prend en argument une liste et retourne le TPE contenant les éléments de la liste.

```
construction_tas : 'a list -> 'a arbre
```

Q - 11 - Déterminer la complexité de la fonction `construction_tas`.

Q - 12 - Écrire la fonction `vider` qui prend en argument un TPE et retourne la liste triée dans l'ordre décroissant, contenant les éléments du tas.

```
vider : 'a arbre -> 'a list
```

Q - 13 - Déterminer la complexité de la fonction `vider`.

Q - 14 - En déduire la fonction `tri_tas` qui prend en argument une liste, et renvoie la liste triée dans l'ordre décroissant des valeurs de la liste initiale.

```
tri_tas : 'a list -> 'a list
```

Q - 15 - Déterminer la complexité de la fonction `tri_tas`.