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.

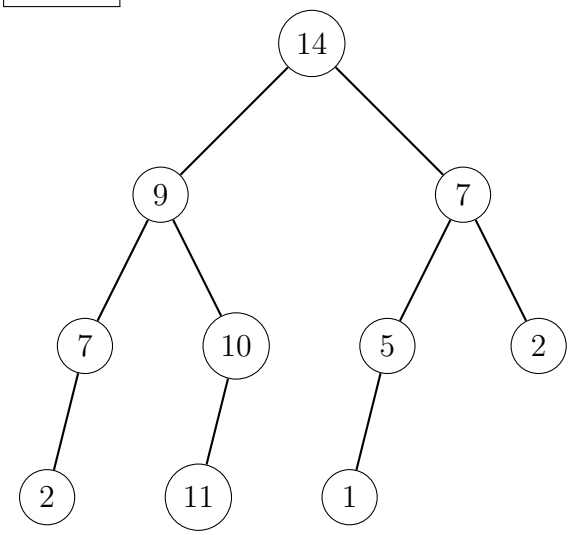
Définition d'un tas presque équilibré :

En Ocaml, nous utiliserons le type suivant :

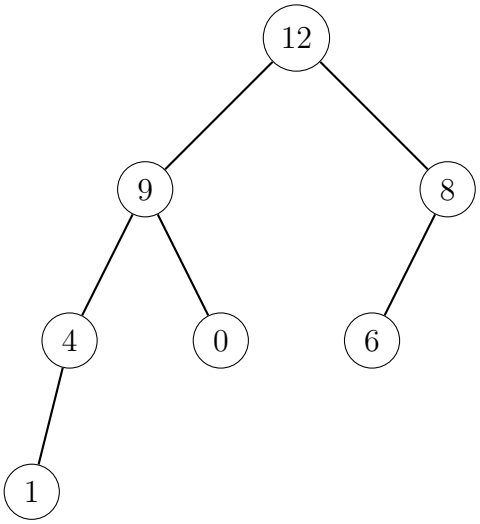
```
type 'a arbre = Vide | N of 'a arbre * 'a * 'a arbre ;;
```

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.

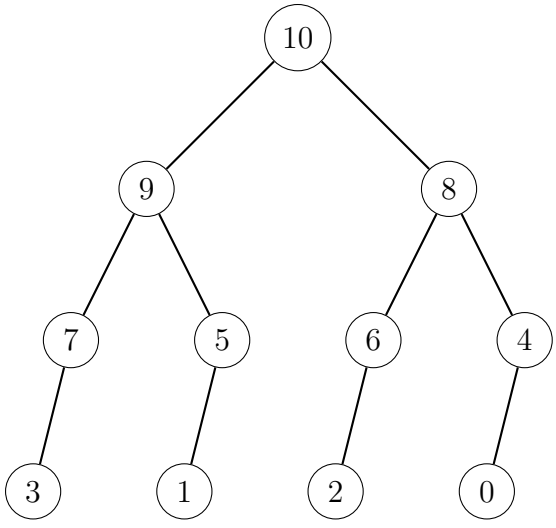
Q - 1 - Parmi les arbres suivants, quels sont les TPE ?



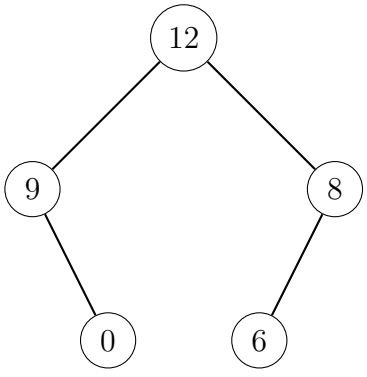
Exemple 1



Exemple 2



Exemple 3



Exemple 4

Correction

Seul l'exemple 3 est un TPE...

- L'exemple 1 ne respecte pas le caractère de file de priorité,
- dans l'exemple 2 le sous arbre gauche possède 4 nœuds, et il n'y en a que 2 dans le sous arbre droit : La relation d'ordre n'est donc pas respectée.
- de même dans l'exemple 4, le sous arbre droit du nœud 9 a plus de nœud que le sous arbre gauche.

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).

Correction

On notera ici $h(a)$ la hauteur de l'arbre a.
Démonstration par induction structurelle sur les APE.

- Pour un arbre à 1 nœud, sa hauteur est de 0, ce qui correspond bien à $\lfloor \rfloor$...
- Considérons un arbre `a = N(g,x,d)` un APE.
  - Si `g = Vide`, alors `d = Vide` pour respecter la relation d'ordre des nœuds. C'est donc le cas précédent.
  - Si `d = Vide`, pour respecter la relation d'ordre des nœuds, le sous arbre gauche est donc une feuille. On a donc bien une hauteur de $1 = \log_2(2)$.
  - Dans le cas général, on suppose que cette propriété est vraie pour deux arbres APE `d` et `g` vérifiant la relation :$\mathcal{N}(g) - 1 \leq \mathcal{N}(d) \leq \mathcal{N}(g)$. Par croissance de la fonction logarithme et parie entière, on a donc $\lfloor \log_2(\mathcal{N}(d)) \rfloor \leq \lfloor \log_2(\mathcal{N}(g)) \rfloor$, soit $h(d) \leq h(g)$, ou encore : $\max(h(d), h(g)) = h(g)$. Donc $h(a) = 1 + h(g)$.
    * Si $\mathcal{N}(d) = \mathcal{N}(g)$, on a $\mathcal{N}(a) = 2\mathcal{N}(g) + 1$. On a :
    $$\begin{array}{rrcl} & \log_2(\mathcal{N}(g)) - 1 & < h(d) \leq & \log_2(\mathcal{N}(g)) \\ \Rightarrow & 1 + \log_2(\mathcal{N}(g)) - 1 & < h(a) \leq & 1 + \log_2(\mathcal{N}(g)) \\ \Rightarrow & \log_2(2\mathcal{N}(g)) - 1 & < h(a) \leq & \log_2(2\mathcal{N}(g)) \end{array}$$
    Donc $h(a) = \lfloor \log_2(2\mathcal{N}(g)) \rfloor = \lfloor \log_2(2\mathcal{N}(g) + 1 \rfloor = \lfloor \log_2(\mathcal{N}(a)) \rfloor$.
    (remarque : on peut démontrer facilement que si p est pair (non nul), $\lfloor \log_2(p) \rfloor = \lfloor \log_2(p+1) \rfloor$...)
    * Si $\mathcal{N}(d) = \mathcal{N}(g) - 1$, alors $\mathcal{N}(a) = 2\mathcal{N}(g)$. On démontrer alors par la même méthode que $h(a) = \lfloor \log_2(2\mathcal{N}(g)) \rfloor = \lfloor \log_2(\mathcal{N}(a)) \rfloor$.

En conclusion, on a donc montrer par induction structurelle la propriété.

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
```

Correction

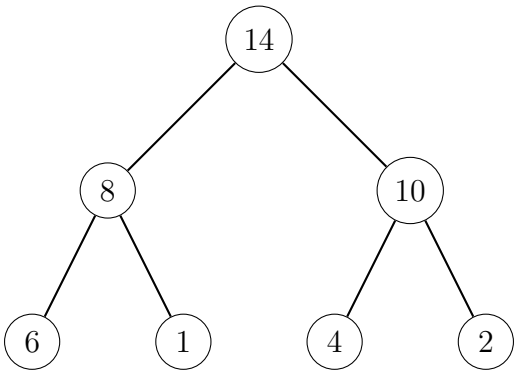
```
let racine a=match a with
  | Vide -> failwith "vide"
  | N(_,x,_) -> x ;;

let rec verif_taille a=match a with
| Vide -> true, 0
| N(g,x,d) -> let (b_g,n_g)=verif_taille g and (b_d,n_d)=verif_taille d in
b_g && b_d && (g=Vide || x>=racine g) &&
(d=Vide || x>=racine d) && n_g-1<=n_d && n_d<=n_g, n_d+n_g+1 ;;

let verifie_tpe a=
 fst (verif_taille a) ;;
```

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.

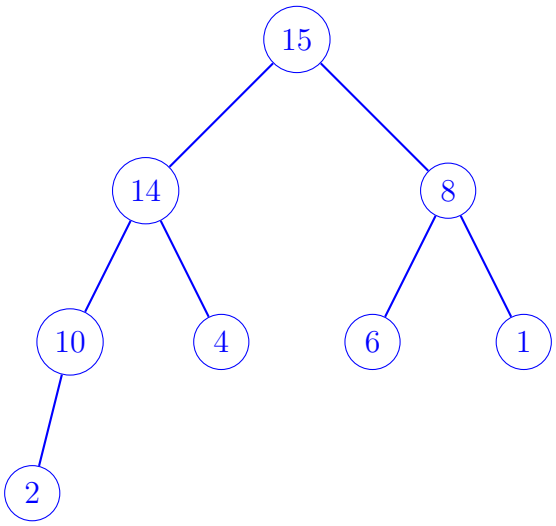
Q - 4 - Dans l'exemple 5 suivant, procéder à l'algorithme puis dessiner le résultat de l'insertion de 15 dans l'arbre :



Exemple 5.

Correction

On trouve :



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
```

Correction

```
 let rec inserer a x=match a with
| Vide -> N(Vide, x, Vide)
| N(g,r,d) when r>x -> N(inserer d x, r, g)
| N(g,r,d) -> N(inserer d r, x, g)
;;
```

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

Correction

La fonction prend en argument un TPE a et renvoie un TPE.
On utilise la preuve des terminaisons des fonctions récursives :
Notons P_a = « Si a est un TPE, alors `insere a x` est un TPE contenant x ».

- Si a est vide, la fonction renvoie `N(Vide,x,Vide)`. C'est bien un TPE.
- Soit a =`N(g,r,d)`, avec g et d des TPE :
  - on suppose que $r > x$ et que g' =`insere d x` est un TPE. Donc la racine r reste bien supérieur aux racines de ces fils, de plus g et d' sont des TPE. La relation d'ordre sur la taille des fils est bien conservée, car $\mathcal{N}(g') = \mathcal{N}(d) + 1$:
$$\begin{aligned} & \mathcal{N}(g) - 1 \leq \mathcal{N}(d) \leq \mathcal{N}(g) \\ \Rightarrow \quad & \mathcal{N}(g) \leq \mathcal{N}(d) + 1 \leq \mathcal{N}(g) + 1 \\ \Rightarrow \quad & \mathcal{N}(g) \leq \mathcal{N}(d) + 1 \wedge \mathcal{N}(d) + 1 \leq \mathcal{N}(g) + 1 \\ \Rightarrow \quad & \mathcal{N}(g) \leq \mathcal{N}(g') \wedge \mathcal{N}(g') \leq \mathcal{N}(g) + 1 \\ \Rightarrow \quad & \mathcal{N}(g) \leq \mathcal{N}(g') \wedge \mathcal{N}(g') - 1 \leq \mathcal{N}(g) \\ \Rightarrow \quad & \mathcal{N}(g') - 1 \leq \mathcal{N}(g) \leq \mathcal{N}(g') \end{aligned}$$
  - Sinon, l'insertion ce fait avec r, ce qui permet de conservé la relation d'ordre sur les racines. La relation d'ordre sur la taille est prouvé comme précédemment...

On a donc bien la preuve de la correction de la fonction récursive.

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.

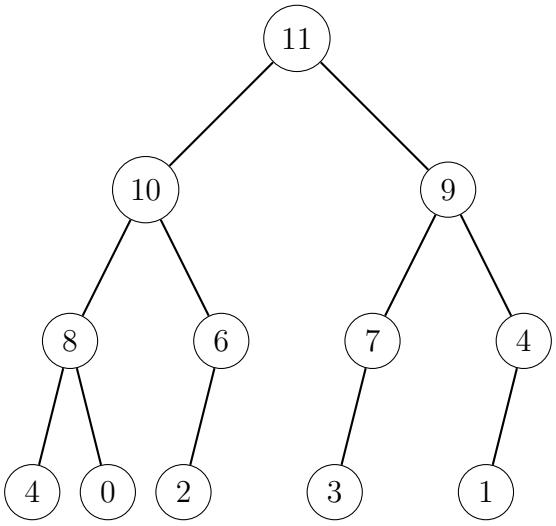
Correction

```
let rec modifier_racine a r=match a with
| Vide -> Vide
| N(g,_,d) when (g=Vide || racine g<=r) && (d=Vide || racine d<=r) -> N(g,r,d)
| N(g,x,d) when (d=Vide || racine d<=racine g) -> N(modifier_racine g r,
racine g, d)
| N(g,x,d) -> N(g,racine d, modifier_racine d r)
;;
```

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.

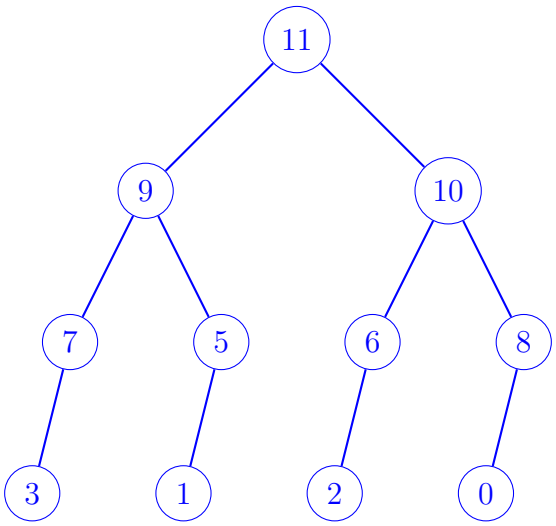
On considère l'arbre a suivant :



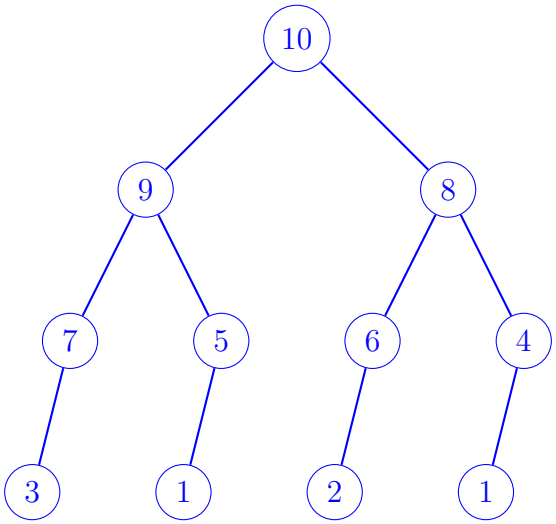
Dessiner l'arbre a' puis l'arbre a'' décrit dans l'algorithme de suppression précédent.

Correction

L'arbre a' :



L'arbre a'' :



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.

Correction

```
let supprimer_racine a=
let rec aux a=match a with
| Vide -> failwith "Vide"
| N(Vide, e, _) -> Vide, e
| N(g,x,d) -> let g2,e=aux g in N(d,x,g2), e
in let b,e=aux a in racine a, modifier_racine b e
;;
```

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
```

Correction

```
let rec construction_tas liste =
  match liste with
    | [] -> Vide
    | x::reste -> inserer (construction_tas reste ) x;;
```

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

Correction

A chaque appel de la fonction, on utilise la fonction `inserer`, qui a une complexité en $O(\log(n))$, on obtient donc une complexité en $O(n\log(n))$.

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
```

Correction

```
let rec vider tas =
  match tas with
    | Vide -> []
    | _ -> let (r, a_2 ) = supprimer_racine tas in r :: (vider a_2);;
```

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

Correction

A chaque appel de la fonction, on utilise la fonction `supprimer_racine`, qui a une complexité en $O(\log(n))$, on obtient donc une complexité en $O(n\log(n))$.

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
```

Correction

```
let tri_tas liste =
  vider ( construction_tas liste ) ;;
```

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

Correction

La fonction utilise successivement deux fonctions de complexité $O(n\log(n))$, la complexité est donc aussi en $O(n\log(n))$.