TD 6 Programmation fonctionnelle
Modules

1 Tableaux extensibles

On considère la signature ci-dessous pour un module de tableaux extensibles (les "listes" de Python).

```
module type ARRAYLIST = sig
type 'a arraylist
val create : unit -> 'a arraylist                    (* [] en Python *)
exception OutOfBound
val get : 'a arraylist -> int -> 'a                  (* L[i] en Python *)
val set : 'a arraylist -> int -> 'a -> unit          (* L[i] = a en Python *)
val len : 'a arraylist -> int                        (* len(L) en Python *)
val append : 'a arraylist -> 'a -> unit              (* L.append(x) en Python *)
val extend : 'a arraylist -> 'a arraylist -> unit    (* L.extend(L2) en Python *
end
```

On se propose d'implémenter un arraylist par un tableau OCaml (de taille fixe) dont on n'utilise pas toutes les cases. Si les cases viennent à manquer, on réalloue le tableau en prenant une taille qui est le double de celle nécessaire. Détaillez l'implémentation correspondante.

2 Files impératives

On considère la signature ci-dessous pour un module de files FIFO.

```
module type ImperativeQueue = sig
  type 'a t
  val create : unit -> 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> unit
  val pop : 'a t -> 'a
end
```

On va représenter la file par une liste chainée: le premier élément de la liste est le premier à sortir, et chaque nouvel élément est ajouté en queue de liste. Chaque cellule de la liste pointe vers un successeur

```
type 'a cell = { elt : 'a; mutable next : 'a cell }
```

et le successeur du dernier élément de la liste est le premier élément (la liste est donc en réalité cyclique). Ceci offre l'avantage d'avoir à stocker uniquement

le dernier élément. Le cas particulier de la file vide est géré en utilisant une option sur une cellule, et la file étant impérative, on utilise une référence. Au final, l'implémentation du type abstrait des cellules que l'on va choisir représenter

```
type 'a t = 'a cell option ref
```

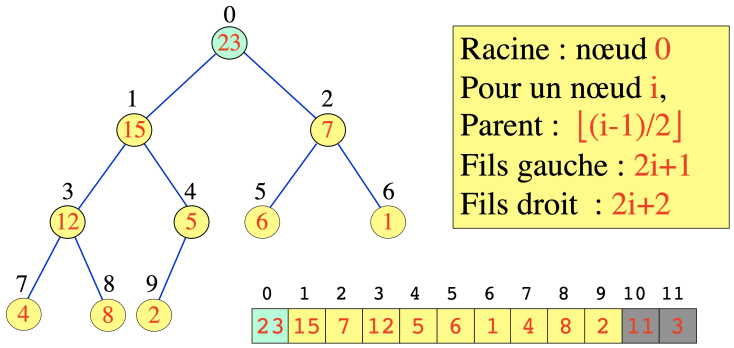
Continuez l'implémentation du module.

3 Files de priorité impératives

On considère la signature ci-dessous pour un module de file de priorité impératif.

```
module type ImperativePriorityQueue = sig
  type 'a t
  val create : unit -> 'a t
  val is_empty : 'a t -> bool
  val add : 'a -> int -> 'a t -> unit
  val get_max : 'a t -> 'a * int
  val remove_max : 'a t -> unit
end
```

Pour implémenter une file de priorité, on utilise un arbre binaire *tassé*: tous les niveaux de l'arbre sont plein sauf le dernier, dont le début est plein. On peut représenter un tel arbre à l'aide d'un tableau. Le noeud racine correspond à la case d'indice 0, le fils gauche à l'indice 1, le fils droit à l'indice 2, etc, cf dessin ci-dessous.




```
type 'a t = {mutable tab : ('a * int ) array; mutable len : int}
```

La propriété essentielle de l'arbre est que tout noeud a une valeur supérieure à celles de ses deux fils. Ainsi, l'élément de priorité maximale se trouve toujours à la racine. L'ajout d'un élément se fait en le plaçant en fin de tableau, puis en le faisant remonter dans l'arbre en permutant avec les pères successifs jusqu'à trouver un père de priorité supérieure. Le retrait de l'élément maximal se fait en plaçant le dernier élément du tableau dans la position libérée à l'indice 0, puis en le faisant descendre dans l'arbre en permutant avec les fils successifs jusqu'à atteindre une position où les deux fils sont de priorité plus faible.

Complétez l'implémentation.