Tas

I. Files de priorité

Définition Une **file de priorité** est une structure de donnée permettant de manipuler un ensemble de données, chacunes affectées d'une *priorité* de telle sorte que l'on trouver et retirer efficacement l'élément de priorité maximale, et ajouter de nouveaux éléments. En version impérative, les opérations voulues ont les types suivants :

```
ajouter : 'a fp -> 'a -> unit
extraire_max : 'a fp -> 'a
```

Il est facile d'implémenter une structure de file de priorité (de tailles bornées) à l'aide de tableaux en cyclant éventuellement (assurez-vous que vous comprenez bien toutes les complexités).

Type de tableau	Minimum	Extraction	Ajout
Non trié	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Trié	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

On peut aussi réaliser une version persistante des files de priorités à l'aide d'arbres binaires de recherche, de telle sorte que toutes les opérations se font avec une complexité en $\Theta(h)$ (et donc en $\Theta(\log n)$ en utilisant des arbres équilibrés).

Exercice 1 Écrire une fonction `extraire_max : 'a abr -> 'a * 'a abr` (sans se préoccuper de l'équilibrage).

II. Une implémentation impérative : les tas

Nous allons présenter une implémentation impérative des tas (bornés) avec des opérations en $\Theta(\log n)$. Mais pour cela, nous avons besoin de faire un détour par les arbres binaires.

Définition

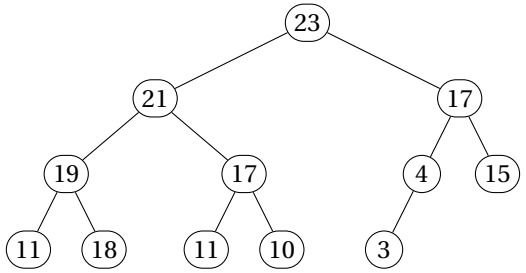
☞ Un **arbre tournoi** est un arbre dont la clé de chaque nœud est supérieure aux clés de ses enfants (s'il est nécessaire de préciser, c'est un arbre tournoi-max, par opposition à un arbre tournoi-min).

☞ Un **arbre quasi-complet à gauche** [a] est un arbre dont tous les niveaux sont remplis, sauf éventuellement le dernier niveau qui est rempli partiellement en partant de la gauche vers la droite.

☞ Un **tas** est un arbre tournoi quasi-complet à gauche.

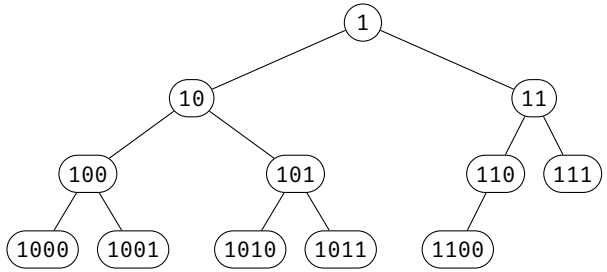
a. Un tel arbre est parfois qualifié de parfait, ou simplement d'arbre complet.

Exemple de tas On représente que les priorités.



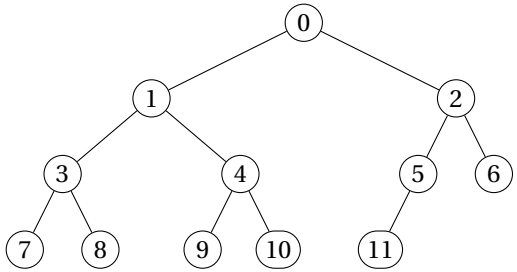
1. Apprenons à compter dans un arbre quasi-complet à gauche

Si l'on numérote les nœuds d'un arbre binaire quasi-complet à gauche en partant de 1, on obtient les numéros suivants (écrits en binaire) :



En particulier, les fils du nœud numéro n ont pour numéros $2n$ et $2n+1$ et son père a pour numéro $\lfloor \frac{n}{2} \rfloor$.

Si l'on numérote à partir de 0 (ce qui est plus naturel pour les tableaux), les fils ont pour numéros $2n+1$ et $2n+2$ et le père a pour numéro $\lfloor \frac{n+1}{2} \rfloor - 1$:

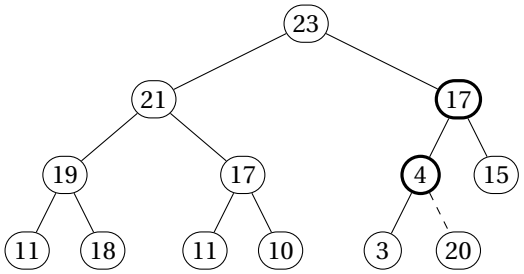


2. Percolation dans un tas

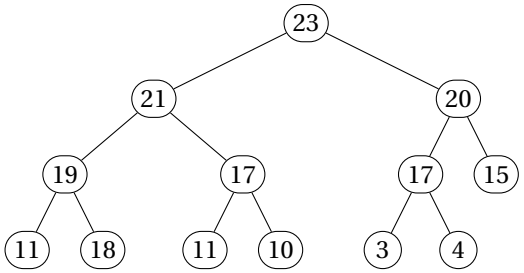
2.1. Ajout et percolation vers le haut

Si l'on ajoute une nouvelle valeur (en respectant la règle d'un arbre quasi-complet à gauche) et que sa priorité est plus grande que son père, il faut la faire « remonter » vers le haut et échangeant les éléments rencontrés sur son chemin.

Exemple On ajoute une donnée de priorité 20. Cette priorité est en conflit avec son père et son grand-père (indiqués en gras) :



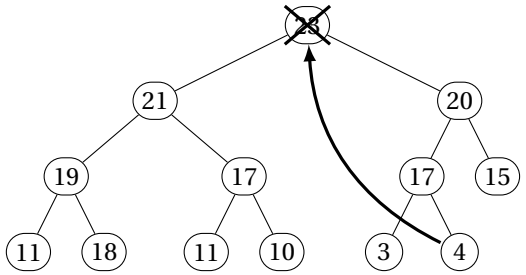
Après percolation vers le haut, on obtient le tas suivant :



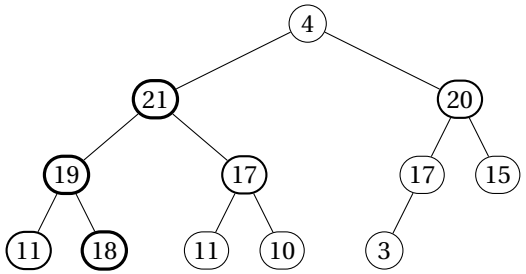
2.2. Retrait et percolation vers le bas

Lorsque l'on retire l'élément de priorité maximale (qui est à la racine), on mets à la place le dernier élément de l'arbre. Comme cela peut entraîner des conflits de priorité avec ses enfants, il faut effectuer une percolation vers le bas, en choisissant judicieusement de quel côté aller.

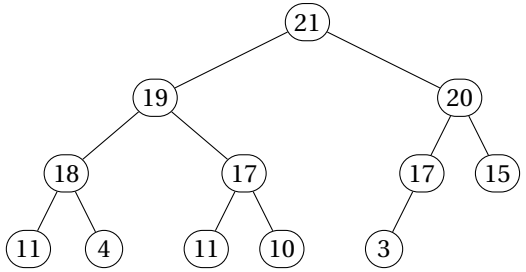
Exemple Reprenons le tas de l'exemple précédent. On commence par supprimer la donnée à la racine et la remplacer par la dernière donnée de l'arbre.



Ensuite, on percole 4 vers le bas pour résoudre les conflits de priorité.



Finalement, on obtient le tas suivant :



3. Implémentation d'un tas

Nous allons utiliser le type suivant, où le champ `taille` indique le nombre de cases utilisées dans le tableau.

```
type 'a tas = { mutable taille : int; contenu : 'a array }
```

3.1. Quelques fonctions auxilliaires

Voici le code de quelques fonctions de base (mais pas toutes). On commence par une petite fonction utilitaire pour simplifier la suite.

```
let echange t i j =
  let tmp = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- tmp
```

Pour la remontée, c'est à vous de travailler.

Exercice 2 (Remontée) Écrire la fonction

```
remontee : 'a tas -> int -> unit
```

qui implémente l'opération de remontée dans un tas à partir de la position indiquée. On supposera que :

☞ cette position est valide, i.e. que la case correspondante contient bien une donnée,

☞ que les données sont réduites à leurs priorités. On peut donc les comparer directement en écrivant par exemple `t.(i) < t.(j)`.

La fonction `descente` est un peu plus compliquée. En voici une version à bien comprendre.

```
let rec descente t i n =
  (* On doit indiquer jusqu'où est rempli le tableau *)
  let candidat = ref i in
  let j = 2 * i + 1 in
  if j < n then begin
    if t.(j) > t.(i) then candidat := j;
    if j + 1 < n && t.(j + 1) > t.(!candidat) then
      candidat := j + 1;
    if !candidat != i then begin
      echange t i !candidat;
      descente t !candidat n
    end
  end
```

3.2. Écriture des fonctions principales

```
let creer_tas taille elt =
  { taille = 0; contenu = Array.make taille elt }
```

```
let ajouter tas valeur =
  if tas.taille = Array.length tas.contenu then
    failwith "Tas plein";
  tas.contenu.(tas.taille) <- valeur;
  remontee tas.contenu tas.taille;
  tas.taille <- tas.taille + 1
```

```
let extraire tas =
  if tas.taille = 0 then failwith "Tas vide";
  let res = tas.contenu.(0) in
  tas.taille <- tas.taille - 1;
  tas.contenu.(0) <- tas.contenu.(tas.taille);
  descente tas.contenu 0 tas.taille;
  res
```

4. Une application : le tris par tas

La structure de tas permet d'obtenir un algorithme de tri en place, nommé *heapsort* (ou *tri par tas* en français). Il s'effectue en deux parties : tout d'abord, on transforme le tableau en tas avec la fonction `heapify` suivante.

```
let heapify t =
  let n = Array.length t in
  for i = 1 to n - 1 do
    remontee t i
  done
```

Ensuite, on extrait les valeurs une à une à l'aide d'extractions successives.

```
let heapsort t =
  (* Mise sous forme de tas *)
  heapify t;
  (* Extraction des éléments dans l'ordre *)
  for i = Array.length t - 1 downto 1 do
    echange t 0 i;
    descente t 0 i
  done
```

Exercice 3

1. Montrer que le tri par tas est en $O(\log n)$ dans le pire des cas (et aussi en moyenne).

Lors de la création du tas à partir d'un tableau (comme dans la première phase du tri par tas, par exemple), on a procédé en appliquant itérativement des *remontées*. C'est la méthode utilisée par John Williams, son inventeur en 1964.

2. Estimer la complexité de cette opération.

Un peu plus tard, Robert Floyd réalise que l'on peut améliorer la méthode en procédant à rebours, et en faisant « descendre » les éléments plutôt que les faire « monter ».

3. Écrire la fonction de création de tas correspondante.
4. Déterminer sa complexité obtenue.

III. Exercices supplémentaires

1. Arbres gauchistes

Nous allons étudier ici une implémentation persistente simple de file de priorité.

Définition

☞ Étant donné un nœud n d'un arbre binaire, le **rang** de ce nœud, noté $r(n)$, est la plus petite distance de n à une feuille descendant de n.

☞ Un arbre binaire est dit **gauchiste** si pour chacun de ses nœuds internes, le rang du fils gauche est supérieur ou égal au nœud du fils droit.

☞ Le rang d'un arbre gauchiste est égal au rang de sa racine.

Question 1 Montrer qu'un arbre gauchiste de rang r contient au moins $2^r - 1$ nœuds.

On appelle **chemin droit** d'un arbre le chemin partant de la racine et descendant en choisissant toujours le sous-arbre droit.

Question 2 Prouver que le chemin droit d'un arbre gauchiste de rang r est de longueur r.

Nous allons maintenant supposer que l'on a des arbres tournoi gauchistes, ce que l'on appellera des *tas gauchistes*.

Question 3 Montrer que l'on peut fusionner deux tas gauchistes de rangs r_1 et r_2 en un tas gauchiste, avec une complexité en $O(r_1 + r_2)$. Pour cela, on pourra considérer la fusion des listes ordonnées correspondant aux chemins droits de chaque tas.

Question 4 En déduire une implémentation persistente des fils de priorité, où les opérations d'ajout et d'extraction sont en $O(\log n)$.

2. Tas modifiables

Nous allons rencontrer plus tard des algorithmes utilisant une file de priorité où l'on aura besoin d'avoir la possibilité de modifier la priorité d'un élément.

Pour cela, nous allons effectuer les modifications suivantes :

☞ on a associer à chacune des données un indice entier dans $[\![0, n-1]\!]$ (où n désigne la taille du tas), le tableau `contenu` contiendra les indices des données,

☞ on a un tableau `priorité` qui contient les priorités des données suivant leur indice,

☞ et un tableau `position` qui indique, en fonction de son indice, donne la position d'une donnée dans le tas (i.e. dans le tableau `contenu`).

Les tableaux `position` et `contenu` doivent vérifier l'invariant suivant :

$$\forall\, p \in [\![0, \texttt{taille} - 1]\!], \texttt{position.(contenu.(}p\texttt{))} = p$$

Question 1 Modifier la fonction `echange` pour tenir compte de cet invariant.

Question 2 Indiquer comment modifier les fonctions `remontee`, `descente`, ainsi que `creer_tas` et `ajouter`.

Reste à programmer la modification du tas lorsque l'on augmente la priorité d'une donnée présente dans le tas.

Question 3 Écrire une fonction `ajouter_priorite` qui fait cela.