Chapitre 2

Tas

Option informatique MP1, MP2 & MP3

I Files de priorité

I.1 Type abstrait

Dans ce chapitre nous allons définir et implémenter un type de données efficace pour traduire le type abstrait de file de priorité. On rappelle qu'il s'agit de maintenir un ensemble d'objets d'un type fixé accompagnés d'une priorité qu'on suppose entière.
On veut pouvoir ajouter un élément en donnant sa priorité et exhiber ou retirer l'élément de priorité maximale. On souhaite ainsi définir un type de donnée `'a fdp` de file de priorité dont les objets sont de type `'a` ainsi que les fonctions

```
fpVide : 'a -> 'a fdp
estVide : 'a fdp -> bool
ajouter : 'a  -> int -> 'a fdp -> unit
premier : 'a fdp -> 'a
enlever : 'a fdp -> unit
```

- `fpVide` demande un élément de type `'a` qui sera l'élément par défaut lorsque l'on aura besoin de tableaux.
- `ajouter` modifie une file de priorité en lui ajoutant un coupe (élément, prioruté),
- `premier` renvoie l'élément de priorité maximale dans une file de priorité sans la modifier.
- `enlever` retire l'élément de priorité maximale d'une file de priorité.

On remarque qu'on choisit une structure mutable.

On peut parfois souhaiter pouvoir modifier la priorité d'un élément :

```
changerPrio : 'a  -> int -> 'a fdp -> unit
```

On choisira ici de considérer cette fonction comme une généralisation de `ajouter` : si l'élément n'était pas dans la file de priorité, on l'ajoute. Pour cela, il peut être utile d'introduire une fonction

```
supprimer : 'a  -> 'a fdp -> unit
```

qui retire un élément de la file de priorité en fonction de sa valeur (et non de sa priorité).
La fonction de changement de priorité sera alors simple :

```
let changerPrio x p fp =
   supprimer x fp;
   ajouter x p fp;;
```

Bien entendu on souhaite une structure qui permette des fonctions dont les temps de calculs seront les plus petits possibles et dont l'occupation en mémoire est faible aussi.

I.2 Rappels : implémentations linéaires

Il y a quatre possibilités pour l'implémentation avec les structures de données classiques

- on peut choisir des listes ou des tableaux
- pour chaque structure on peut soit trier lors de l'adjonction, soit extraire le maximum lors du calcul ou du retrait de l'élément de priorité maximale.

Exercice 1 - Listes triées
Écrire les 7 fonctions définies ci-dessus au cas où le type concret est une liste maintenue triée :

```
type 'a fdp = {mutable contenu : ('a * int) list};;
```

Quelles sont les complexités des fonctions ?

Exercice 2 - Tableaux non triés
Écrire les 7 fonctions définies ci-dessus au cas où le type concret est une tableau :

```
type 'a fdp = {mutable taille : int;
               contenu :('a * int) array};;
```

On supposera que le nombre d'éléments qui seront dans la file de priorité ne peut pas dépasser une valeur maximale `nMax` qui sera définie globalement.

```
let nMax = 1000;; (* Par exemple *)
```

Quelles sont les complexités des fonctions ?

I.3 Réalisation par A.B.R.

Nous avons vu une structure qui maintient l'ordre des clés : les arbres binaires de recherche. On peut donc imaginer les utiliser pour implémenter une file de priorité.
Les arbres binaires de recherche sont des arbres binaires :

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

La clé sera ici la seconde composante (elle doit être croissante dans le parcours infixe).

```
let cle = snd;;
```

Le type est immédiat, on rappelle qu'on veut un type impératif

```
type 'a fdp =
   {mutable contenu : ('a*int) arbre};;
```

Les fonctions concernant les files vides sont simples

```
let fpVide a =
   {contenu = Vide};;

let estVide fp =
   fp.contenu = Vide;;
```

L'ajout est une traduction directe de la fonction sur les arbres binaires de recherche.

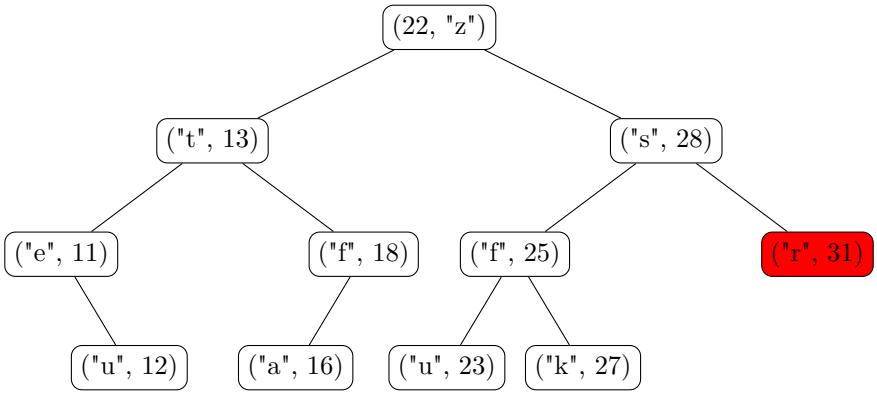


FIGURE 1 – Élément de priorité maximale

```
let ajouter x k fp  =
  fp.contenu <- ajouterRacine (x, k) fp.contenu;;
```

La recherche et le retrait de l'élément de priorité maximale ne correspondent pas au cas des arbres binaires de recherche car on n'en connaît pas la clé. On remarque cependant que la structure d'arbre binaire de recherche implique que l'élément de clé maximale est la racine du dernier fils droit non vide. On va donc adapter les fonctions qui on servi à la suppression. On peut donc écrire

Fonctions concernant l'élément de priorité maximale

```
let premier fp =
   let rec aux arbre =
      match arbre with
      |Vide -> failwith "La file est vide"
      |Noeud(_, (x, k), Vide) -> x
      |Noeud(_, _, d) -> aux d in
   aux fp.contenu;;

let enlever fp =
   let rec aux arbre =
      match arbre with
      |Vide -> failwith "La file est vide"
      |Noeud(g, _, Vide) -> g
      |Noeud(g, n, d) -> Noeud(g, n, aux d) in
   fp.contenu <- aux fp.contenu;;
```

La complexité de `ajouter`, `premier` et `enlever` est de l'ordre de la hauteur de l'arbre : en moyenne on peut compter sur une complexité en $\mathcal{O}(\log_2(n))$ mais elle peut atteindre n dans le pire des cas si on ne maintient pas un arbre équilibré. La structure d'arbre est persistante, cela implique que la complexité spatiale peut devenir importante : à chaque ajout et retrait on crée un nombre de nœuds de l'ordre de la hauteur.

Pour supprimer ou changer une priorité, on a besoin de repérer un élément par sa valeur et non sa priorité. La complexité sera alors celle d'un parcours d'arbre, linéaire en la taille et non en la hauteur.

Exercice 3 - Suppression
Écrire les fonctions `supprimer` et `changerPrio` dans le cas de cette implémentation.

II Tas

Nous allons maintenant définir une structure d'arbre moins structurée que celle d'arbre binaire de recherche mais suffisante pour gérer les priorités.

II.1 Rappel : arbres quasi-complets à gauche

Définition 1 : Complétude d'arbres binaires

Un arbre binaire de hauteur h est

complet si toutes les feuilles sont de profondeur $h+1$,

quasi-complet si toutes les feuilles sont de profondeur h ou $h+1$,

quasi-complet à gauche s'il est quasi-complet et si, de plus, les nœuds de profondeur h sont tous situés à gauche.

On a vu qu'alors le nombre de nœuds de profondeur k est 2^k pour tout $k < h$.
On en déduit que la taille n de l'arbre vérifie $2^h \leq n < 2^{h+1}$.

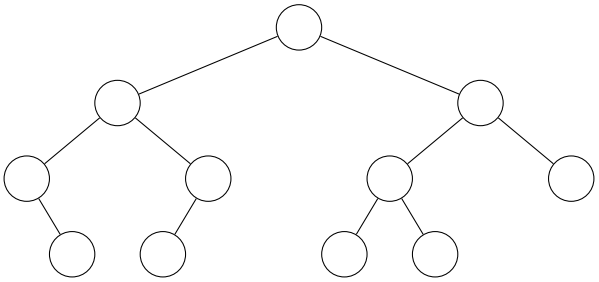


FIGURE 2 – Arbre quasi-complet

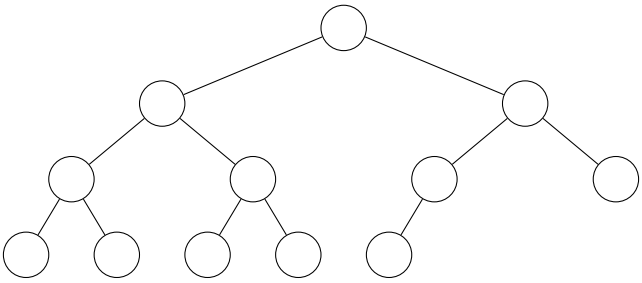


FIGURE 3 – Arbre quasi-complet à gauche

Exercice 4 - Caractérisation des arbres complets
Écrire une fonction `complet : 'a arbre -> bool` qui renvoie `true` si et seulement si l'arbre est complet.

Exercice 5 - Caractérisation des arbres quasi-complets à gauche
Donner une caractérisation (récursive) d'un arbre quasi-complet à gauche en fonction de ses fils droit et gauche (quasi-complets à gauche ou complets) ; la hauteur intervient.
En déduire une fonction `qcg : 'a arbre -> bool` qui renvoie `true` si et seulement si l'arbre est quasi-complet à gauche mais non complet.

II.2 Décroissance verticale

Définition 2 : Arbres décroissants

Un arbre décroissant est un arbre binaire dont les nœuds possèdent une clé et tel que la clé d'un nœud est toujours inférieure à celle de son père.

On a donc remplacé la croissance des clé "en largeur" des arbres binaires de recherche par une croissance du bas vers le haut moins contraignante.

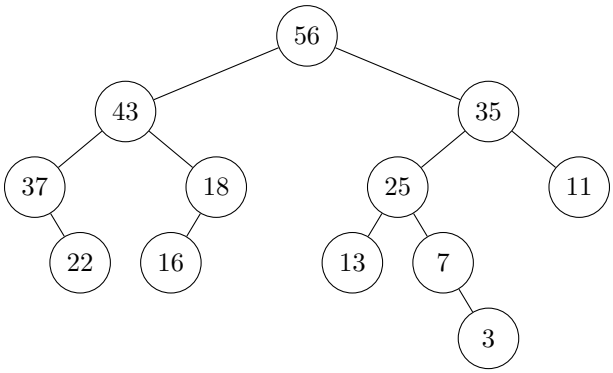


FIGURE 4 – Un arbre décroissant

Définition 3 : Tas

Un tas est un arbre décroissant quasi-complet à gauche.

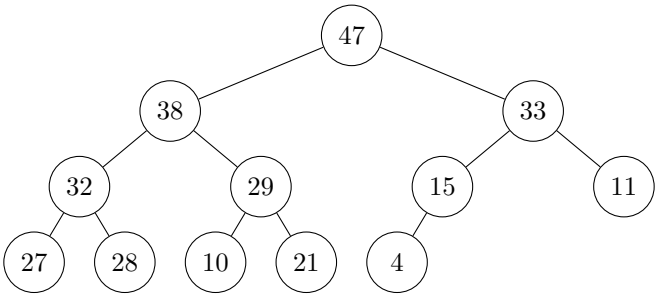


FIGURE 5 – Le tas t_0

II.3 Opérations abstraites

On suppose qu'en plus du type de données d'arbre on dispose de fonctions qui permettent

- de trouver le père d'un nœud (distinct de la racine),
- de parvenir au dernier nœud non vide (dans le parcours en largeur),
- de parvenir au premier nœud vide (à droite du précédent).

Dans t_0, le dernier nœud non vide a la valeur 4.

II.3.a Ajouter

Pour ajouter un élément commence par l'ajouter à la position suivante dans l'ordre du parcours en largeur puis on le remonte tant que sa valeur est supérieure à celle de son père.
On va ajouter 42 à t_0.

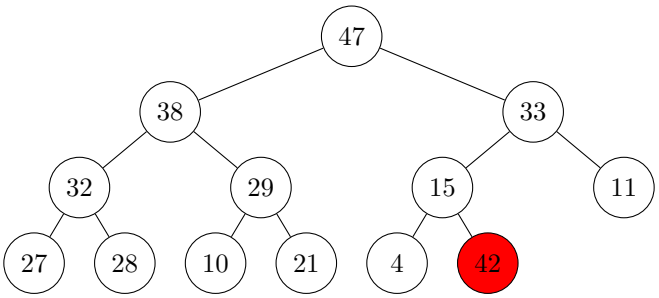


FIGURE 6 – Première étape : placement de 42 à la première feuille vide

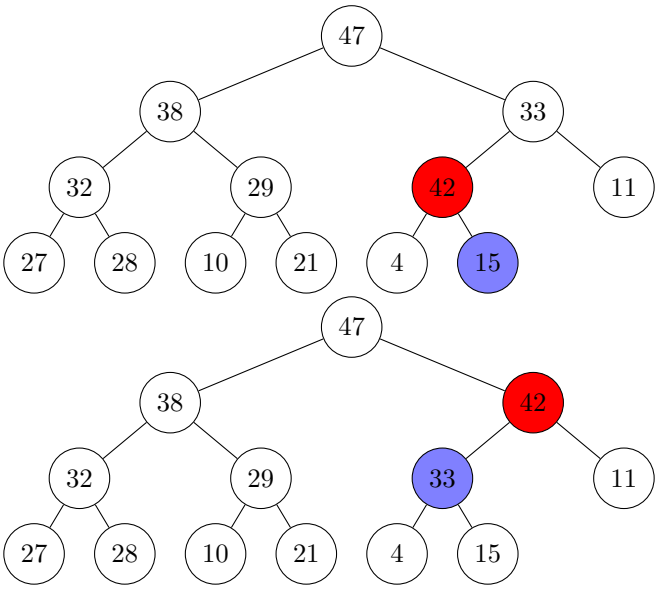


FIGURE 7 – Deuxième étape : échanges de 42 avec un père plus petit

II.3.b Enlever

Pour enlever l'élément maximum d'un tas on procède comme dans le cas de l'ajout.

- On modifie l'ensemble des élément de l'arbre en maintenant la structure d'arbre quasi-complet à gauche ; ici on va remplacer l'élément à la racine par la dernière valeur dans un nœud, en supprimant aussi le dernier nœud.
- On reconstitue la structure croissante : ici on va descendre l'élément tant qu'il est inférieur à l'un de ses fils en le remplaçant par le plus grand de ses fils.

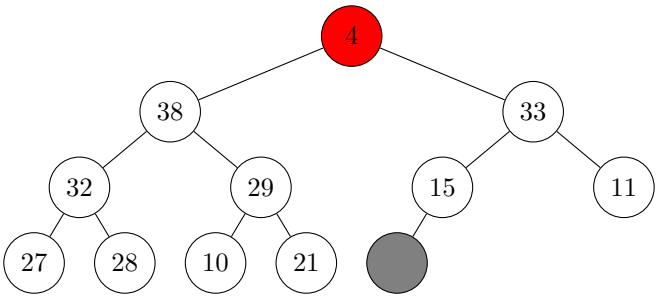


FIGURE 8 – Première étape : remplacement de 47 par 4 dans t_0

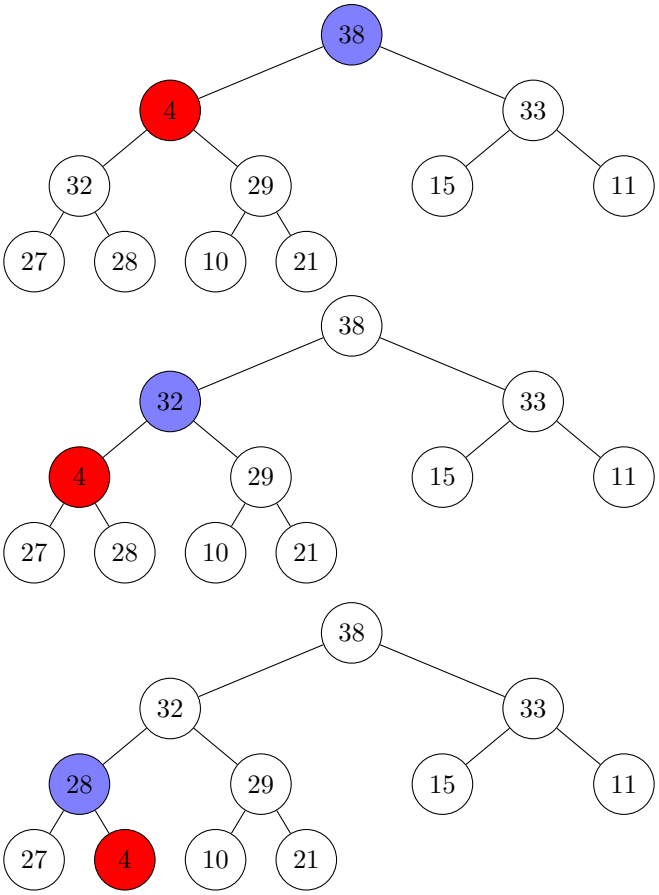


FIGURE 9 – Deuxième étape : échanges de 4 avec son plus grand fils

II.4 Réalisation par tableaux

Il est possible de réaliser les constructions ci-dessus avec des arbres : la difficulté technique est dans l'accès aux dernier nœud ou à la première feuille vide, voir l'exercice . Cependant le problème de l'occupation en mémoire demeure : chaque opération crée des nouveaux nœuds, le long d'une branche avec un ordre de grandeur de $\log_2(n)$.
Pour diminuer cette complexité spatiale, on va utiliser un type de donnée impératif pour stocker les valeurs des nœuds : un tableau. En effet le parcours en largeur d'un arbre quasi-complet à gauche fournit une bijection entre les nœuds et un tableau.

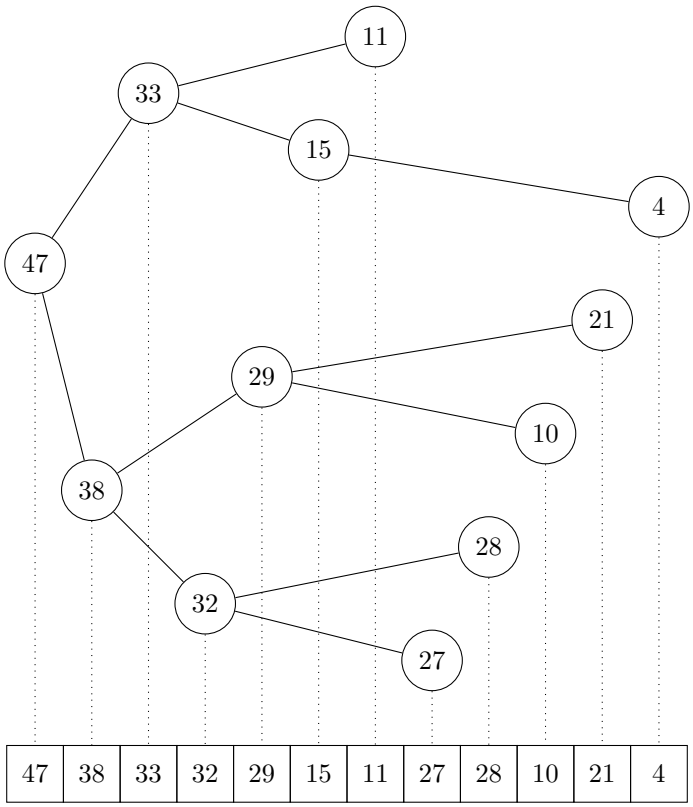


FIGURE 10 – Parcours en largeur de t_0

Comme la structure de tableau n'est pas dynamique, le nombre d'éléments est fixé, on va inclure le tableau utilisé dans un tableau assez grand pour pouvoir contenir le nombre maximal d'éléments. Bien entendu il est difficile de prévoir et il y a un risque de débordement.On doit alors maintenir un entier qui sera la taille de la file.

On a besoin d'une fonction de clé pour comparer les valeurs d'un tas. On va traiter directement le cas des files de priorité.

```
let nMax = 100;;

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

type 'a fdp = ('a * int) tas;;

let cle = snd;;
```

Les fonctions simples sont en temps constant

```
let fpVide a =
  {taille = 0; arbre = Array.make nMax (a, 0) };;

let estVide t =
  t.taille = 0;;

let premier t =
  fst t.arbre.(0);;
```

On aura besoin de la traditionnelle fonction d'échange :

```
let echange tab i j =
      let temp = tab.(i) in
      tab.(i) <- tab.(j);
      tab.(j) <- temp;;
```

Pour les fonctions `ajouter` et `enlever` on a besoin de connaître l'indice du père d'un nœud ou les indices de ses fils.

Théorème 1 : Numérotation des fils

Avec la numérotation du tableau de 0 à $n-1$, le père du nœud d'indice i a pour indice $(i-1)/2$ et ses fils guache et droit ont pour indices $2i+1$ et $2i+2$ respectivement.

L'ajout se fait comme indiqué ci-dessus avec une fonction auxiliaire pour effectuer la remontée.

Ajout d'un élément

```
let rec remonter n t =
   if n > 0
      then let m = (n-1)/2 in
         if (cle t.(m)) < (cle t.(n))
         then (echange t m n; remonter m t);;

let ajouter_tas x tas  =
  let n = tas.taille in
    tas.arbre.(n) <- x;
    tas.taille <- n + 1;
    remonter n t.arbre;;

let ajouter x p file = ajouter_tas (x, p) file;;
```

Pour la fonction `enlever` on a vu qu'il faut redescendre les éléments et effectuant un échange avec le fils de priorité maximale.
On commence par rechercher ce fils. Comme il peut ne pas exister de fils on va renvoyer un type optionnel (`Some/None`) qui permet de gérer l'absence de réponse. Le dernier indice possible est $n-1$ si n est la taille, un nœud d'indice k peut admettre des fils d'indice $2k+1$ et $2k+2$.
Le nombre de fils de k est donc 0 pour $n-1 < 2k+1$, 1 pour $n-1 = 2k+1$ et 2 pour $n-1 > 2k+1$.

Retrait du terme de priorité maximale

```
let filsGrand k t n =
   if n <= 2*k + 1
   then None
   else if n = 2*k + 2
        then Some (2*k + 1)
        else if (cle t.(2*k+1)) < (cle t.(2*k+2))
             then Some (2*k + 2) else Some (2*k + 1);;

let rec descendre k t n =
   match filsGrand k t n with
   |Some p when (cle t.(p)) > (cle t.(k))
       -> (echange t k p; descendre p t n)
   |_ -> ();;

let enlever_tas tas =
   let n  = tas.taille -1 in
   let t  = tas.arbre in
   echange t 0 n;
   tas.taille <- n;
   descendre 0 t n;;

let enlever file = fst (enlever_tas file);;
```

Les complexités des fonctions `ajouter` et `enlever` sont celles de `monter` et `descendre`. Dans ces dernières on fait un nombre fini d'opérations et on appelle récursivement avec un nœud de profondeur modifiée de 1 ou -1 donc les complexités sont en $\mathcal{O}(h)$ où h est la hauteur. Comme l'arbre est équilibré on en déduit

Théorème 2 : Complexités dans un tas

Les complexités des fonctions `ajouter` et `enlever` sont en $\mathcal{O}\big(\ln(n)\big)$ où n est la taille du tas. Les autres fonctions sont de complexité constante.

II.5 Exercices

Exercice 6 - Test de tas
Écrire une fonction `estTas t p` qui teste si les valeurs d'un tableau comprises ont des indices compris entre 0 à $p-1$ forment un tas.

Exercice 7 - Suppression dans un tas
Écrire la fonction `suprimer`. Quelle est sa complexité ?

III Heapsort

Dès qu'on a une structure de file de priorité on peut l'utiliser pour réaliser un tri.
On remplit une file à partir d'une file vide vide en ajoutant un à un les éléments d'un tableau (ou une liste). On retire ensuite les éléments un à un en formant un tableau : celui-ci sera trié.
La complexité du tri est alors un $\mathcal{O}\big(n(a+b)\big)$ où a et b sont les complexité respective de l'ajout et du retrait d'un élément.
Comme on sait que la complexité d'un tri par comparaisons est au moins un $\mathcal{O}\big(n\log(n)\big)$ on voit qu'une au moins des deux opérations ci-dessus doit être en $\mathcal{O}\big(\log(n)\big)$.
On peut considérer que le tri par insertion correspond à ce schéma avec une file de priorité implémentée par un tableau trié tandis que tri par sélection correspond à une file de priorité implémentée par un tableau non trié. On retrouve bien une complexité en $\mathcal{O}\big(n^2\big)$ car l'une des deux opérations est de complexité linéaire.
L'implémentation par un tas donne un tri de complexité asymptotique optimale.

Comme dans le cas des tris par insertions on ne va pas utiliser les fonction du type abstrait : on va travailler directement sur la structure de données.

Si `t` est un tableau, il peut être vu comme un arbre quasi-complet à gauche mais cet arbre n'est pas, en général, décroissant. Comme les éléments sont déjà présents dans le tableau on peut le transformer en tas en ajoutant les éléments depuis l'indice 0 vers l'indice $n-1$: pour chaque i on remonte `t.(i)` dans le tas formé par les éléments de 0 à $i-1$ qui forment un tas. La clé ici est simplement la valeur.

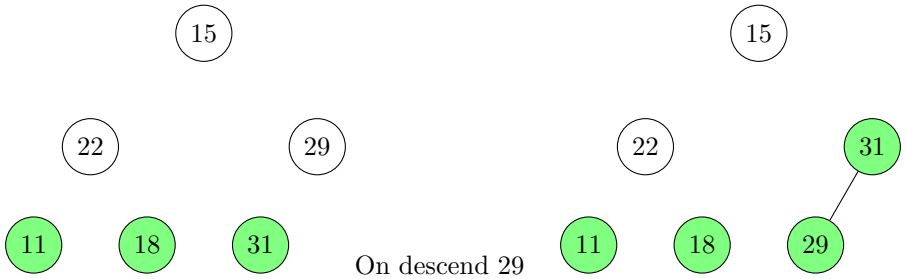
```
let cle x = x;;
```

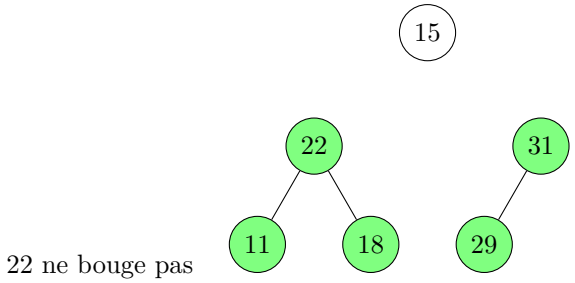
Exercice 8 - Création d'un tas
En utilisant uniquement la fonction `remonter`, écrire une fonction qui transforme `t` en un tas.
Quelle en est la complexité ?

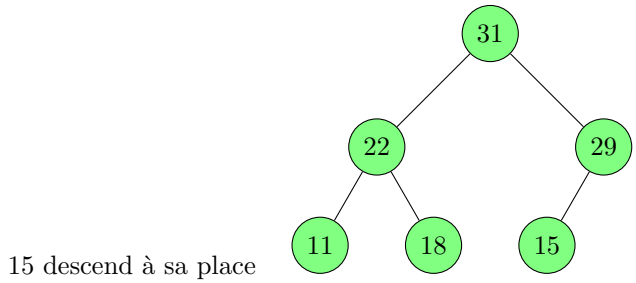
Lorsqu'on en enlève tous les éléments d'un tas un par un ceux-ci sont placés dans les emplacements depuis la position $n-1$ vers la gauche. Les n premier termes du tableau sont alors triés car le premier élément retiré est celui de priorité maximale.

Exercice 9 - Tri d'un tas
En vous inspirant de cette remarque écrire une fonction (sans résultat) qui trie un tableau en place.

La première étape peut être améliorée en construisant le tas non pas du haut vers le bas mais en construisant des tas depuis le bas. Initialement les nœuds forment des tas de taille 1 juxtaposés.
On va former des tas de plus en plus gros en descendant les pères pour former un tas avec leurs fils qui eux-mêmes des tas. Pour cela on doit travailler depuis le dernier père (dans l'ordre des indices) vers le premier.
Dans l'exemple suivant on part de `[|15; 22; 29; 11; 18; 31|]` ; les nœuds colorés forment une forêt de tas, à la fin il ne restera qu'un tas unique.







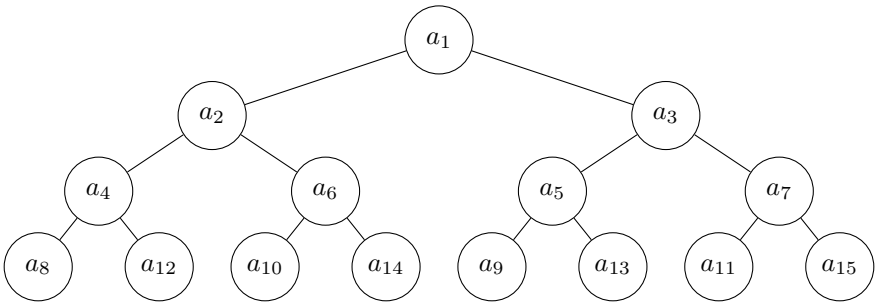
Exercice 10 - Une amélioration
Écrire une fonction qui transforme `t` en un tas avec cette méthode
Quelle est sa complexité ?

A Problème 1 : tableaux fonctionnels

On a vu qu'on pouvait réaliser un arbre (complet à gauche) par un tableau.
Nous nous intéressons ici au problème inverse : donner une implémentation d'un tableau sous la forme d'un arbre (persistant).
Un tableau peut être vu comme une fonction depuis un ensemble d'indices vers l'ensemble des valeurs. On considère la fonction suivante qui associe un nœud à un entier non nul.

```
let rec elt k arbre =
  match arbre with
  |Vide -> failwith "Element non defini"
  |Noeud(g,n,d) -> if k = 1 then n
                             else if k mod 2 = 0
                                  then elt (k/2) g else elt (k/2) d
                                     ;;
```

Dans le cas $n = 15$ on accède ainsi aux nœuds de l'arbre complet de hauteur 3.

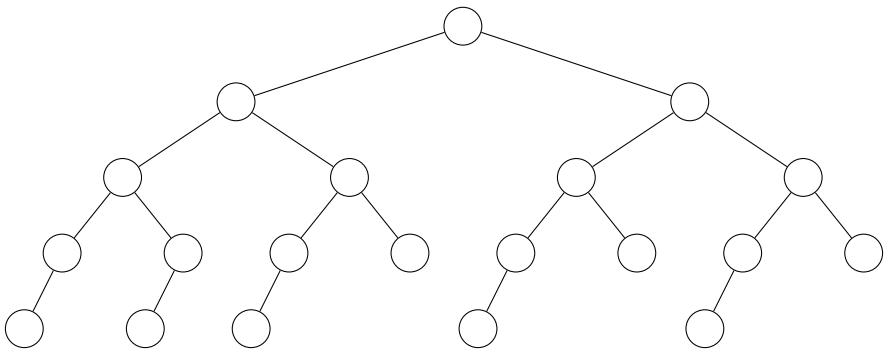


On a ainsi défini une structure de tableau à accès presque constant qui a l'avantage d'être persistante (quand on change une valeur, l'ancien tableau peut exister encore) et qui, de plus, peut-être augmentée.

Question 11 - Tableaux fonctionnels

- Écrire une fonction `change k x a` qui renvoie une nouvel arbre a' tel que $a'_k = x$ et $a'_i = a_i$ pour $i \neq k$. Si la position n'était pas occupée mais si le père existe on ajoutera l'élément à sa place.
- Écrire une fonction `creeTableau n x` qui renvoie un arbre de taille n dont tous les éléments ont la valeur x.
- Quelle est la complexité de `elt` et de `change` pour un tableau de taille n ?

On conserve la structure ci-dessus : on considère des arbres binaires de taille n tels que les nœuds occupés correspondent aux indices de 1 à n. Ces arbres correspondent donc des tableaux de taille n : on nommera **arbres de Braun** de tels arbres.



Question 12 - Structure des arbres de Braun

`t = Noeud(g,x,d)` est un arbre de Braun.

- Montrer que `g` et `d` sont des arbres de Braun.
- Montrer que `taille g` = `taille d` ou `taille g` = `taille d` +1.
- Montrer que `hauteur g` = `hauteur d` ou `hauteur g` = `hauteur d` +1.
- Montrer que la feuille la plus à gauche est à la profondeur maximale.

On définit la fonction

```
let rec ajouterBraun x t =
  match t with
  |Vide -> Noeud(Vide,x,Vide)
  |Noeud(g,y,d) -> Noeud(ajouterBraun x d,y,g);;
```

Question 13 - Modification des arbres de Braun

- Prouver que si `t` est un arbre de Braun alors `ajouter x t` est encore un arbre de Braun.
- Comment enlever un élément en conservant la structure d'arbre de Braun ?

Un **tas de Braun** est un arbre de Braun décroissant.

Question 14 - Tas de Braun

- Modifier la fonction `ajouterBraun` en préservant la structure de tas.
- Comment enlever le plus petit élément en conservant la structure de tas de Braun ?

B Problème 2 : arbres gauchistes

Nous avons implémenté les files d'attentes avec la structure de tas. Nous avons traduit la structure arborescente en tableau ce qui nous a fait perdre la persistance. Le problème ci-dessus a proposé une structure persistante, nous allons proposer une autre structure, persistante elle aussi, permettant d'implémenter une file d'attente avec des opérations de complexité garantie en $O(\log_2(n))$.

B.1 Rang vertical

Définition 4 : Rang vertical

Dans un arbre le **rang vertical** d'un nœud est la longueur de la plus courte branche (entre le nœud et une feuille) où on compte les nœuds. C'est la plus petite profondeur des fils vides.

Le rang vertical d'un tas est h, sa hauteur, sauf s'il est parfait, dans ce cas le rang est $h+1$.
On définit un nouveau type d'arbre binaire contenant le rang vertical :

```
type 'a arbreRV = VideRV
                | NoeudRV of int * 'a arbreRV * 'a * 'a arbreRV;;
```

Un arbre avec un seul élément est de rang vertical 1 : `Noeud(1 ,Vide, x, Vide)`

Question 15 - Première valeur
Écrire une fonction `rangV : arbreRV -> int` qui renvoiela première valeur donnée dans `NoeudRV(k, g, r, d)` pour un nœud et 0 sinon.

Question 16 - Test de cohérence
Écrire une fonction `testRV : arbreRV -> bool` qui teste, pour un arbre avec rang, si la valeur renvoyée par `rangV` est bien le rang vertical.

Question 17 - Conversion
Écrire une fonction `ecrireRang : arbre -> arbreRV` qui traduit un arbre en arbre avec rang vertical ; le rang vertical devra prendre la valeur exacte.

B.2 Arbres gauchistes

Définition 5 : Arbre gauchiste

Un arbre est **gauchiste** si, pour tout nœud, le rang vertical du fils gauche est supérieur au rang vertical du fils droit.

Question 18 - Test
Écrire une fonction `testG : arbreRV -> bool` qui teste si un arbre avec rang est bien un arbre gauchiste. On supposera que la première valeur donnée dans `NoeudRV(k, g, r, d)` est bien le rang vertical.

Question 19 - Structure des arbres gauchistes

- Prouver qu'un arbre gauchiste de rang r admet au moins $2^r - 1$ nœuds.
- Prouver que, dans un arbre gauchiste de rang r, le chemin qui choisit le fils droit à chaque nœud est longueur r (en nombre de nœuds).

B.3 Tas gauchistes

Définition 6 : Tas gauchiste

Un **tas gauchiste** est un arbre gauchiste décroissant.

L'opération principale avec des tas gauchistes est la fusion : on part de 2 tas gauchistes avec des éléments distincts et on veut obtenir un nouveau tas gauchiste contenant l'union des éléments des deux tas.
La racine est simple : c'est la plus grande des deux racines.
Si la racine appartient au premier tas, on lui attribue deux fils, l'un est un fils du premier tas, l'autre est la fusion de l'autre tas et du second fils du premier tas.

Question 20 - Fusion
En choisissant le fils à fusionner, montrer qu'on peut fusionner deux tas gauchistes **décroissants** avec une complexité en $O(r_1 + r_2)$ où r_1 et r_2 sont les rangs des arbres.

Question 21 - File de priorité persistante
Donner une implémentation persistante des files de priorité.

Solutions

Solution de l'exercice 1 - Listes triées

```
let fpVide x =
   {contenu = []};;

let estVide fp =
   fp.contenu = [];;
```

```
let ajouter x p fp =
   let rec inserer liste =
      match liste with
      |[] -> [(x, p)]
      |(y, k):: reste when p > k -> (x, p)::liste
      |t::q -> t::(inserer q) in
   fp.contenu <- inserer fp.contenu;;
```

```
let premier fp =
   fst (List.hd fp.contenu);;

let enlever fp =
   fp.contenu <- List.tl fp.contenu;;
```

Si un élément à supprimer n'existe pas, on laisse la file inchangée.

```
let rec supprimer x fp =
   let rec oter liste =
      match liste with
      |[] -> []
      |(y, p)::reste when y = x -> reste
      |t::q -> t::(oter q) in
    fp.contenu <- oter fp.contenu;;

let changerPrio x p fp =
   supprimer x fp;
   ajouter x p fp;;
```

`fpVide`, `estVide`, `premier` et `enlever` sont de complexité constante.
`ajouter`, `supprimer` et `changerPrio` sont de complexité linéaire.

Solution de l'exercice 2 - Tableaux non triés

```
let fpVide x =
   {taille = 0;
    contenu  = Array.make nMax (x, 0)};;

let estVide fp =
   fp.taille = 0;;
```

On ajoute en fin de tableau, ne pas oubler de modifier la taille

```
let ajouter x p fp =
   let n = fp.taille in
   let t = fp.contenu in
   if n = Array.length t
   then failwith "La file est remplie"
   else begin
        t.(n) <- (x, p);
        fp.taille <- n+1 end;;
```

On écrit une fonction qui détermine la valeur de priorité maximale (la première) ainsi que son indice.

```
let maxi t n =
   if n = 0
   then failwith "La file est remplie"
   else let prio_max = ref (snd t.(0)) in
        let val_max = ref (fst t.(0)) in
        let i_max = ref 0 in
        for i = 1 to (n-1) do
           let x, k = t.(i) in
           if k > !prio_max
           then begin prio_max := k;
                      val_max := x;
                      i_max := i end done;
        !val_max, !i_max;;
```

```
let premier fp =
   fst (maxi fp.contenu fp.taille);;

let enlever fp =
   let n = fp.taille in
   let t = fp.contenu in
   let i = snd(maxi t n) in
   t.(i) <- t.(n-1);
   fp.taille <- n-1;;
```

```
let supprimer x fp =
   let n = fp.taille in
   let t = fp.contenu in
   let i = ref 0 in
   while !i < n && fst t.(!i) <> x do incr i done;
   if !i < n
   then (t.(!i) <- t.(n-1); fp.taille <- n-1);;

let changerPrio x p fp =
   supprimer x fp;
   ajouter x p fp;;
```

`fpVide`, `estVide` et `ajouter` sont de complexité constante.
`premier`, `enlever`, `supprimer` et `changerPrio` sont de complexité linéaire.

Solution de l'exercice 3 - Suppression

```
let rec supprimer x fp =
   let rec oter arbre =
      match arbre with
      |Vide -> Vide
      |(y, p)::reste when y = x ->  suppressionRacine  arbre
      |Noeud(g, r, d) -> Noeud(oter g, r, oter d) in
    fp.contenu <- oter fp.contenu;;

let changerPrio x p fp =
   supprimer x fp;
   ajouter x p fp;;
```

Solution de l'exercice 4 - Caractérisation des arbres complets
Un arbre est complet s'il est vide ou si ses deux fils sont complets de même hauteur.
On va écrire une fonction qui renvoie le booléen et la hauteur.

```
let rec complet_h arbre =
   match arbre with
   |Vide -> true, -1
   |Noeud(g, _, d) -> let cg, hg = aux g in
                      let cd, hd = aux d in
                      hg = hd && cg && cd, (max hg hd) + 1;;

let complet arbre = fst (complet_h arbre);;
```

Solution de l'exercice 5 - Caractérisation des arbres quasi-complets à gauche
Un arbre de hauteur h est quasi-complet à gauche si et seulement si

- il est non vide, l'arbre vide est complet
- son fils gauche est quasi-complet à gauche de hauteur $h-1$ et son fils droit est complet de hauteur $h-2$
- son fils gauche est complet de hauteur $h-1$ et son fils droit est complet de hauteur $h-2$
- son fils gauche est complet de hauteur $h-1$ et son fils droit est quasi-complet à gauche de hauteur $h-1$

Pour gérer les différents cas on peut définir un nouveau type :

```
type completude = C of int | QCG of int | Non;;
```

`C h` caractérise un arbre complet de hauteur h ($h \geq -1$), `QCG h` caractérise un arbre quasi-complet à gauche de hauteur h ($h \geq 1$) et `Non` caractérise les autres arbres.

```
let rec typeC arbre =
   match arbre with
   |Vide -> C (-1)
   |Noeud(Vide, x, Vide) -> C 0
   |Noeud(g, x, d) -> begin match typeC g, typeC d with
                      |C hg, C hd when hg = hd -> C (hg + 1)
                      |C hg, C hd when hg = hd + 1 -> QCG (hg +1)
                      |C hg, QCG hd when hg = hd -> QCG (hg +1)
                      |QCG hg, C hd when hg = hd + 1 -> QCG (hg
                         +1)
                      |_ -> Non end;;
let qcg arbre =
   match typeC arbre with
   |QCG h -> true
   |_ -> false;;
```

Solution de l'exercice 6 - Test de tas

Un tableau est un tas si tous les fils ont une clé supérieure à celle de leur père. Plutôt que tester les fils d'un nœud, il faudrait s'assurer de leur existence, on va tester le père de chaque nœud.

```
let estTas t p =
   let valide = ref true in
      for i = 1 to (p-1) do
      if  t.(i) >= t.((i-1)/2) then valide := false done;
   !valide;;
```

Solution de l'exercice 7 - Suppression dans un tas

On n'a pas d'algorithme efficace pour retrouver un élément, la complexité est linéaire en la taille.

```
let supprimer x tas =
   let n  = tas.taille in
   let t = tas.arbre in
   let i = ref 0 in
   while !i < n && fst t.(i) <> x do  incr i done;
   if !i < n
   then begin t.(i) <- t.(n);
              tas.taille <- n-1 end;;
```

Solution de l'exercice 8 - Création d'un tas

```
let mettreEnTas t =
  let n = Array.length t in
  let tas = {taille = n; arbre = t} in
  for i = 0 to (n-1) do remonter i tas done;
  tas;;
```

La complexité est majorée par $\sum_{i=1}^{n-1} \lfloor \log_2(i+1) \rfloor$: c'est un $\mathcal{O}\big(n\log(n)\big)$.

Solution de l'exercice 9 - Tri d'un tas

```
let triParTas t =
  let tas = mettreEnTas t in
  let n = tas.taille in
  for i = 0 to (n-1) do enlever tas done;;
```

Le tableau `t` est alors trié.

```
let t = [|5; 12; 9; 1; 8; 13; 4|];;
triParTas t;;
t;;

# - : int array = [|1; 4; 5; 8; 9; 12; 13|]
```

La complexité est toujours un $\mathcal{O}\big(n\log(n)\big)$.

Solution de l'exercice 10 - Une amélioration
Le dernier fils admet `n-1` pour indice donc le dernier père admet `(n-2)/2` pour indice.

```
let mettreEnTas t =
  let n = Array.length t in
  let m = (n-2)/2 in
  let tas = {taille = n; arbre = t} in
  for i = 0 to m do descendre (m-i) tas done;
  tas;;
```

Il y a 2^k nœuds de profondeur k pour $k < h$, hauteur de l'arbre.
Pour chacun de ces nœuds la descente demande au plus $h-k$ étapes.
La complexité est donc majorée par $\displaystyle\sum_{k=0}^{h-1}(h-k)2^k$.

Si on pose $S_p = \displaystyle\sum_{k=0}^{p-1}(p-k)2^k$ on a

$$S_{p+1} = \sum_{k=0}^{p}(p+1-k)2^k = 2^p + \sum_{k=0}^{p-1}\big((p-k)2^k + 2^k\big) = 2^p + S_p + \sum_{k=0}^{p-1}2^k = 2.2^p + S_p - 1.$$

Comme on a $S_0 = 0$, on en déduit $S_p = \displaystyle\sum_{k=0}^{p-1}2.2^k - 1 = 2.(2^p-1) - n$.
Ainsi S_h est majoré par $2.2^h - h - 2$ avec $2^h \leq n < 2^{h+1}$ donc $S_h \leq 2n$: la complexité est un $\mathcal{O}(n)$.

Solution de la question 11 - Tableaux fonctionnels

- Pour transformer un élément il suffit de le chercher et de transformer le nœud :

```
let rec change k x arbre =
  match arbre with
  |Vide -> if k = 1
           then Noeud(Vide, x, Vide)
           else failwith "Position non accessible"
  |Noeud(g,n,d) -> if k = 1
                   then Noeud(g,x,d)
                   else if k mod 2 = 0
                        then Noeud(change (k/2) x g ,n, d)
                        else Noeud(g, n, change (k/2) x d);;
```

  Comme suggéré par l'énoncé on autorise, si on parvient à un arbre vide, l'ajout d'un élément d'indice k si l'élément d'indice $k/2$ est défini. En effet on ne peut appeler `change 1 x Vide` que depuis un arbre non vide[1].

- On peut créer un tableau à partir du vide en ajoutant un-à-un tous les éléments grâce à la possibilité de la question précédente. Si on effectue cela récursivement on conserve en fait tous les tableaux intermédiaires. On va donc utiliser une boucle et modifier une variable référencée : cela permet au **garbage collector** de libérer la place.

```
let creeTableau n x =
  let a = ref Vide in
  for i = 1 to n do a := change i x !a done;
  !a;;
```

  On verra que les tableaux fonctionnels ont une structure équilibrée et que les fils sont similaires, le fils gauche ayant un terme de plus. On peut retrouver un algorithme récursif

```
let rec creeTableau n x =
  if n = 0
  then Vide
  else let k = (n-1)/2 in
       Noeud(creeTableau (n-1-k) x,x,creeTableau k x));;
```

- Comme indiqué ci-dessus, un tableau de taille $n \leq 2^h - 1$ est représenté par un arbre de hauteur h donc les parcours de lecture ou de changement ont une complexité en $O(h) = O\big(\log(n)\big)$.

Solution de la question 12 - Structure des arbres de Braun

- L'arbre de gauche est défini par les positions k pour k pair. De plus, depuis la racine du fils gauche, on définit les positions à l'aide de $k/2$; on remplit donc un tableau fonctionnel avec les positions $2/2$, $4/2$, …, $n/2$. On construit ainsi un arbre de Braun.
  De même pour le fils droit.

- D'après la question précédente il y a $n/2 = \lfloor \frac{n}{2} \rfloor$ nœuds dans le fils gauche.
  La racine n'est dans aucun fils donc il y a $n - 1 - \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil - 1$ nœuds dans le fils droit.
  On a $\lfloor \frac{n}{2} \rfloor \leq \lceil \frac{n}{2} \rceil \leq \lfloor \frac{n}{2} \rfloor + 1$ d'où le résultat.

- Il n'y a qu'un arbre de Braun de taille p.
  De plus l'arbre de taille $p+1$ est obtenu en ajoutant un élément.
  Le fils gauche est ainsi égal au fils droit ou est obtenu en ajoutant un nœud ce qui ne peut augmenter la hauteur que de 1 au maximum.

1. ou directement en voulant changer une valeur d'un tableau vide

- La hauteur est atteinte pour le fils gauche d'après la troisième question.
  Par récurrence on en déduit que la feuille la plus à gauche est à la hauteur maximale.

Solution de la question 13 - Modification des arbres de Braun

- On va montrer, par récurrence sur n que la fonction `ajouterBraun x t` appliquée à un arbre qui représente un tableau fonctionnel d'indices de 1 à n le transforme en un tableau fonctionnel d'indices de 1 à $n+1$.
  - C'est évident pour $n = 0$.
  - On suppose que cela est vrai pour tous les entiers $k \leq n-1$ avec $n \geq 2$.
    t est un arbre de Braun de taille n, `t = Noeud(g,y,d)`, avec g et d de taille respectives t_g et t_d.
    On note `t' = ajouterBraun x d`; c'est, d'après l'hypothèse de récurrence, un arbre de Braun.
    Si d et g ont la même taille p alors t' et g sont des arbres de Braun de tailles respectives $p+1$ et p donc `Noeud(t',y,g)` a la structure d'un arbre de Braun de taille $2p+1 = n+1$.
    De même si g a un élément de plus que d.

- La réciproque de "ajouter à droite puis inverser les fils" est "inverser les fils puis retrancher à droite" c'est-à-dire "retrancher à gauche puis inverser les fils".
  La fonction renvoie la valeur enlevée en plus du nouvel arbre.

```
let rec enleverBraun t =
  match t with
  |Vide -> failwith "Il n'y a rien à enlever"
  |Noeud(Vide,x,_) -> (x,Vide) (* Si g est vide alors d l'est
     aussi*)
  |Noeud(g,x,d) -> let (y,g') = enleverBraun g in
                   (y,Noeud(d,x,g'));;
```

Solution de la question 14 - Tas de Braun

- On envoie vers le bas la plus petite clé :

```
let rec ajouterTasBraun x t =
  match t with
  |Vide -> Noeud(Vide,x,Vide)
  |Noeud(g,y,d) ->  if cle x < cle y
                    then Noeud(ajouterTasBraun x d,y,g)
                    else Noeud(ajouterTasBraun y d,x,g);;
```

- Pour enlever le plus petit élément, celui qui est à la racine, on peut
  - enlever le dernier élément, en conservant sa valeur,
  - le placer à la racine
  - le descendre en échangeant avec le plus grand de ses fils, récursivement.

  On commence par écrire une fonction qui fait descendre un nœud de clé trop petite vers le bas de l'arbre. On supposera que l'arbre vérifie toutes les propriété à l'exception de la clé de la racine qui peut être trop petite.

```
let rec descendre t =
  match t with
  |Vide  -> t
  |Noeud(Vide,_,_) -> t
  |Noeud(Noeud(g,y,d),x,Vide)
   -> if x < y then t
                else Noeud(descendre(Noeud(g,x,d)),y,Vide)
  |Noeud(Noeud(g,y,d),x,Noeud(g',y',d'))
   -> if y < y'
      then (if x > y
            then Noeud(descendre(Noeud(g,x,d)),y,Noeud(g',y',d
               '))
            else t)
      else (if x > y'
            then Noeud(Noeud(g,y,d),y',descendre(Noeud(g',x,d
               ')))
            else t);;
```

Il suffit alors de remplacer la racine par le dernier terme et de faire descendre.

```
let resteTasBraun t =
  let (x,t') = enleverBraun t in
  match t' with
  |Vide -> Vide
  |Noeud(g,y,d) -> descendre(Noeud(g,x,d));;
```

Solution de la question 15 - Première valeur

```
let rangV a =
   match a with
   |VideRV -> 0
   |NoeudRV(k, g, r, d) -> k;;
```

Solution de la question 16 - Test de cohérence

```
let rec testRV a =
   match a with
   |VideRV -> true
   |NoeudRV(k, g, r, d) -> testRV g && testRV d &&
                           k = 1 + (max (rangV g) (rangV d));;
```

Solution de la question 17 - Conversion

```
let rec ecrireRang a =
   match a with
   |Vide -> VideRV
   |Noeud(g, r, d) -> let grv = ecrireRang g in
                      let drv = ecrireRang d in
                      let rg = rangV grv in
                      let rd = rangV drv in
                      NoeudRV (1 + (max rg rd), grv, r, drv);;
```

Solution de la question 18 - Test

```
let rec testG a =
   match a with
   |VideRV -> true
   |NoeudRV(k, g, r, d) -> rangV g >= rangV d;;
```

Solution de la question 19 - Structure des arbres gauchistes

- Dans un arbre de rang r toutes les feuilles sont à une profondeur au moins $r-1$ donc tous les niveaux de 0 à $r-1$ sont remplis.
  Il y a donc au moins $1+2+\cdots+2^{r-1}=2^r-1$ nœuds.
- Le rang d'un arbre est le maximum des rangs de ses fils augmenté de 1 donc le rang d'un arbre gauchiste est le rang de son fils droit augmenté de 1. Le résultat demandé se prouve alors par récurrence sur le rang.
  On en déduit que $r=\mathcal{O}\big(\log_2(n)\big)$ où n est la taille de l'arbre.

Solution de la question 20 - Fusion
On choisit systématiquement le fils droit car il est de rang vertical minimal. On doit placer les deux fils e fonction de leurs rangs verticaux.

```
let faireArbreG racine fils1 fils2 =
  let r1 = rangV fils1 in
  let r2 = rangV fils2 in
  if r1 < r2
  then Noeud(r1+1, fils2, racine, fils1)
  else Noeud(r2+1, fils1, racine, fils2);;

let rec fusion a1 a2 =
  match a1, a2 with
  |Vide, _ -> a2
  |_, Vide -> a1
  |Noeud(_,g1,x1,d1), Noeud(_,g2,x2,d2) when x1 > x2
         -> faireArbreG x1 g1 (fusion d1 a2)
  |Noeud(_,g1,x1,d1), Noeud(_,g2,x2,d2)
         -> faireArbreG x2 g2 (fusion d2 a1);;
```

La complexité est proportionnelle au nombre d'appels récursifs donc est un $\mathcal{O}(r_1 + r_2)$.

Solution de la question 21 - File de priorité persistante

```
let fpVide () -> Vide;;

let estVide f = (f = Vide);;

let ajouter x f =
  fusion f (Noeud(1, Vide, x, Vide));;

let premier f =
  match f with
  |[] -> failwith ''La file est vide''
  |Noeud(_,_,x,_) -> x;;

let enlever f =
  match f with
  |[] -> failwith ''La vile est vide''
  |Noeud(_,g,_,d) -> fusion g d;;
```

Les opérations sont bien en $\mathcal{O}\big(\log_2(n)\big)$ d'après les résultats des exercice 19 et 20