1 Introduction aux arbres

1.
```
let rec contient b a = match b,a with
      |Nil,_ -> true
      |Noeud(g,x,d), Nil -> false
      |Noeud(gb,xb,db),Noeud(ga,xa,da) -> (xa=xb) && (contient gb ga)
                                          && (contient db da);;
```

2. Dérivation

(a)
```
Exp(Somme(Produit(Id,Id),Constante(1.)))
```

(b)
```
let rec derive t = match t with
      |Exp (x) -> Produit (Exp(x), derive x)
      |Id -> Constante (1.)
      |Constante (x) -> Constante (0.)
      |Somme (x,y) -> Somme (derive x, derive y)
      |Difference (x,y) -> Difference ( derive x , derive y)
      |Produit (x,y) -> Somme ( Produit (x, derive y) , Produit (y, derive x));;
```

Sur l'exemple précédent, on obtient :

```
Produit (Exp (Somme (Produit (Id, Id), Constante 1.)),Somme (Somme (
Produit (Id, Constante 1.), Produit (Id, Constante 1.)),Constante 0.))
```

3.
```
type op = Somme | Difference | Produit | Division | Modulo;;

type arbre =
    |F of int
    |N of op * arbre * arbre;;


(* vale : arbre -> bool * int *)
let rec vale t = match t with
    |F(x) -> (true,x)
    |N(y,g,d) -> let (gab,gav) = (vale g) and (drb,drv)= (vale d) in
                 if not(gab && drb) || (y=Division || y=Modulo) && drv=0
                     then (false,0)
                 else match y with
                       |Somme -> (gab && drb,gav+drv)
                       |Difference -> (gab && drb,gav-drv)
                       |Produit -> (gab && drb,gav*drv)
                       |Division -> (gab && drb,gav/drv)
                       |Modulo -> (gab && drb,gav mod drv);;

let u = N(Division,F(2),N(Difference,F(2),F(2)));;

(* defini : arbre -> bool *)
let defini t = let (b,v) = vale t in b;;
```

La fonction `vale` effectue tous les calculs en prenant soin de vérifier à chaque étape qu'on ne va pas diviser par 0. Elle renvoie la valeur de l'expression courante et un booléen indiquant si l'expression est bien définie. Ces calculs sont nécessaires car on peut avoir une division par une expression ayant pour valeur 0.

4. Arbres non nécessairement binaires

(a)
```
type 'a noeud = Noeud of 'a * ('a noeud list);;
type 'a arbre = Vide | Racine of 'a noeud;;
```

i.
```
let rec hauteur arb = match arb with
    |Vide -> -1
    |Racine(Noeud(a,[])) -> 0
    |Racine(Noeud(a, t::q)) -> max (1+hauteur(Racine(t))) (hauteur(
                                          Racine(Noeud(a,q))));;

let rec taille arb = match arb with
    |Vide -> 0
    |Racine(Noeud(a,[])) -> 1
    |Racine(Noeud(a, t::q)) ->  taille(Racine(t)) + taille(Racine(
                                                  Noeud(a,q)));;

let rec nb_feuilles arb = match arb with
    |Vide -> 0
    |Racine(Noeud(a,[])) -> 1
    |Racine(Noeud(a, t::[]))->  nb_feuilles (Racine(t))
    |Racine(Noeud(a, t::q))->  nb_feuilles (Racine(t))
                             + nb_feuilles (Racine(Noeud(a,q)));;
```

ii. On commence par une fonction qui donne la longueur d'une liste.

```
let rec long lis = match lis with
    |[] -> 0
    |t::q -> 1 + (long q);;
```

On peut écrire la fonction.

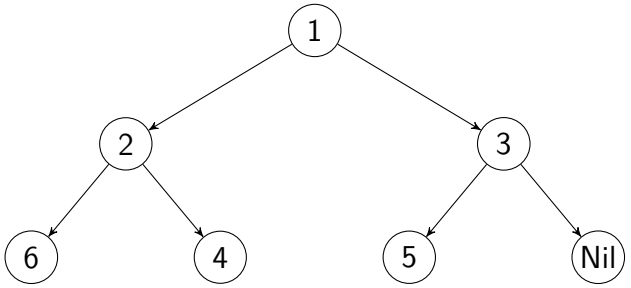
```
let rec degnoeud n = match n with
    |Noeud(x,l) -> let rec parcours l = match l with
                           |[] -> 0
                           |  t::q -> max (degnoeud t) (parcours q)
                   in max (long l) (parcours l);;

let degre a = match a with
    |Vide -> 0
    |Racine(n) -> degnoeud n;;
```

iii.
```
let rec somnoeud n = match n with
       |Noeud(x,l) -> let rec parcours l = match l with
                                |[] -> 0
                                |t::q -> (somnoeud t) + (parcours q)
                      in x + parcours l;;

   let rec somme a  = match a with
       |Vide -> 0
       |Racine(n) -> somnoeud n;;
```

(b)
```
let p = 10;;

type 'a arbp = | Vide
     | N of 'a * ('a arbp array);;



let rec hautr a = match a with
    |Vide -> -1
    |N(x,t) -> let u = ref (hautr t.(0)) in
               for i = 1 to (p-1) do
                   let v = (hautr t.(i)) in
                   if v > !u then u := v
               done;
               1+ !u;;

let rec taille a = match a with
    | Vide -> 0
    |N(x,t) -> let u = ref (taille t.(0)) in
               for i = 1 to (p-1) do
                   u := !u + (taille t.(i))
               done;
               1+ !u;;

let rec nbfeuille a =
    let tabvide = Array.make p Vide in
    match a with
         |Vide -> 0
         |N(x,t) when t = tabvide -> 1
         |N(x,t) -> let u = ref (nbfeuille t.(0)) in
                    for i = 1 to (p-1) do
                         u := !u + (nbfeuille t.(i))
                    done;
                    !u;;
```

```
let rec degre a = match a with
    |Vide -> 0
    |N(e,t) -> let d = ref 0 in
               let m = ref 0 in
               for i = 0 to p-1 do
                   if t.(i)<>Vide then (d:=!d+1; m:=max !m (degre (t.(i))))
               done;
               !d;;

let rec somme a = match a with
    |Vide -> 0
    |N(e,t) -> let s = ref e in
               for i = 0 to p-1 do
                   s := !s + (somme t.(i))
               done;
               !s;;
```

5. (a)



(b) Dans un tableau de la même taille que l'arbre, on place la racine à l'indice "taille du sous-arbre gauche" et on itère l'opération avec chacun des sous-arbres. Attention à opérer avec un décalage d'indice de "taille du sous-arbre gauche" pour placer les éléments du sous-arbre droit dans le tableau.

(c)
```
let rec taille a = match a with
    |Nil -> 0
    |Noeud(b,n,c) -> taille b + 1 + taille c;;


let arbreVersPerm a t =
    let rec arbreVersPermSousArbres i j a = match a with
        | Nil -> ()
        | Noeud (g,x,d) -> let p = taille g in
                           t.(i+p) <- x;
                           arbreVersPermSousArbres i (i+p-1) g ;
                           arbreVersPermSousArbres (i+p+1) j d in
    arbreVersPermSousArbres 0 (taille a -1) a;;
```

Remarque : La fonction ci-dessus fait de nombreux appels à la fonction `taille`. On peut éviter cela en remplissant le tableau à l'aide d'un parcours infixe de l'arbre :

```
let arbreVersPerm a t =
 let k = ref 0 in
 let rec aux a =
  match a with
   | Nil -> ()
   | N (g, x, d) -> (aux g; t.(!k) <- x; incr k; aux d)
 in
 aux a;;
```

(d)
```
let rechMin i j t =
    let m = ref i in
    for k = i+1 to j-1 do
        if t.(!m)>t.(k) then m:=k
    done;
    !m;;

let creeArbre n t  =
    let rec propagation i j =
    if i = j then Nil else
        let k = rechMin i j t in
        Noeud(propagation i k, t.(k), propagation (k+1) j)
    in propagation 0 n;;
```

6. (a)
```
let rec hauteur a = match a with
    |Vide -> -1
    |Noeud(g,x,d) -> 1 + max (hauteur g) (hauteur d);;

let rec est_equil a = match a with
    |Vide -> true
    |Noeud(g,x,d) -> (est_equil g) && (est_equil d)
                     && abs((hauteur g) - (hauteur d)) <= 1;;
```

(b)
```
let  est_equil2 arb =
    let rec est_equil_ss_arbre a = match a with
        | Vide -> (-1, true)
        |Noeud(g,x,d) -> let (hg,eg) = est_equil_ss_arbre g
                         and (hd,ed) = est_equil_ss_arbre d in
                         (1 + max hg hd, eg && ed && abs(hd-hg) <= 1) in
    snd(est_equil_ss_arbre arb);;
```

(c) Soit $h \geq 0$. On note $N(h)$ le nombre **minimal** de noeud d'un arbre équilibré de hauteur h.

Il est clair que $N(0) = 1; N(1) = 2; N(2) = 4$ et $N(3) = 7$. On remarque que pour $h \geq 1$, $N(h+1) = 1 + N(h) + N(h-1)$ (on peut faire un arbre équilibré de hauteur h avec un sous-arbre équilibré de hauteur $h-1$ et un autre de hauteur $h-2$).

Si on pose $u_h = N(h) + 1$ on obtient $u_{h+1} = u_h + u_{h-1}$ et donc $u_h \sim C_1\varphi^h + C_2\breve{\varphi}^h$ où

$\varphi = \dfrac{1+\sqrt{5}}{2}$ et $\breve{\varphi} = \dfrac{1-\sqrt{5}}{2}$.

On considère un arbre équilibré de hauteur h et de taille t. D'après ce qui précède, $t \geq N(h) = C_1\varphi^h + C_2\breve{\varphi}^h - 1$. De ce fait, il existe une constante A telle que $t \geq A\varphi^h$ (car $h \mapsto C_2\breve{\varphi}^h - 1$ est bornée). En prenant le logarithme on obtient $h\log(\varphi) + \log(A) \leq \log(t)$ et donc $h = O(\log(t))$.

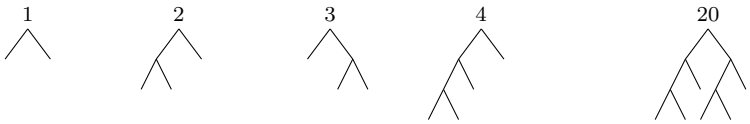
7. (a) Montrons l'existence et l'unicité de la décomposition :

— **Existence :** soit $n \in \mathbb{N}^*$, $p = \max\{k \in \mathbb{N}^* | 2^k \text{ divise } n\}$ et $q = ((n \mod 2^p) - 1)/2$. L'entier q est bien défini car par définition de p, $n \mod 2^p$ est impair. De plus, par définition de q, on a bien $n = 2^p(2q+1)$.

— bf Unicité : supposons qu'il existe $n \in \mathbb{N}^*$ ainsi que $(p, q, p', q') \in \mathbb{N}^4$ tels que $n = 2^p(2q+1) = 2^{p'}(2q'+1)$. On peut supposer sans perte de généralité que $p \geq p'$. Ainsi $n/2^{p'} = 2q' + 1 = 2^{p-p'}(2q+1)$ est un entier impair ce qui implique que $p - p' = 0$, c'est-à-dire $p = p'$. On obtient ensuite $q = ((n \mod 2^p) - 1)/2 = ((n \mod 2^{p'}) - 1)/2 = q'$. La décomposition (p, q) est donc unique.

```
let rec decomp n =
    if n mod 2 = 0 then let (p0,q0) = decomp (n/2) in (p0+1,q0)
    else (0,(n-1)/2);;
```

(b)



(c) Montrons par récurrence que tout nombre entier est un arbre.

— **Initialisation :** on a vu que 1 et 2 sont des arbres.

— **Hérédité :** soit $n \in \mathbb{N}^*$ et supposons que tout entier inférieur à n est un arbre. Soit (p, q) la décomposition de $n+1$. Comme p et q sont inférieurs à n, ils sont des arbres. Le nombre $n+1$ est donc l'arbre construit avec comme sous-arbre gauche p et sous-arbre droit q.

— **Conclusion :** tout nombre entier est un arbre.

(d) Il n'y a que 1.

(e) Montrons par récurrence que le plus petit arbre de hauteur $n \geq 2$ est $3 * 2^{n-2} - 1$.

— **Initialisation :** on a vu que $2 = 3 * 2^{2-2} - 1$ est le plus petit arbre de hauteur 2

— **Hérédité :** soit $n \in \mathbb{N}^*$ et supposons que le plus petit arbre de hauteur n est $3 * 2^{n-2} - 1$. Le plus petit arbre de hauteur $n+1$ a nécessairement un sous-arbre de hauteur n. Il a donc comme sous-arbre, le plus petit sous-arbre de hauteur n et un sous-arbre vide (sinon on obtiendrait facilement un arbre plus petit de hauteur $n+1$). Par hypothèse, le plus petit sous-arbre de hauteur n est $3*2^{n-2}-1$. Pour la hauteur $n+1$, le plus petit arbre est a donc $3*2^{n-2}-1$ comme sous-arbre gauche ou droit, c'est-à-dire qu'il a pour valeur $2^{3*2^{n-2}-1}$ ou $2*(3*2^{n-2}-1)+1 = 3*2^{n-1}-1$. Or pour $n > 2$, la deuxième valeur est toujours inférieure. Le plus petit arbre de hauteur $n+1$ a pour valeur $3 * 2^{n-1} - 1$.

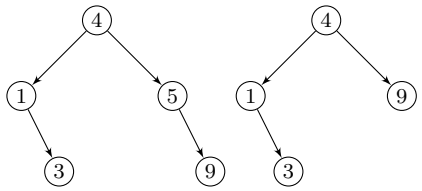
— **Conclusion :** le plus petit arbre de hauteur $n \geq 2$ est $3*2^{n-2}-1$ et il correspond au peigne droit se terminant par l'arbre 2.

(f) `let rec taille n =`

```
if n=0 then 0
else let (p,q)=decomp n in
     max (taille p) (taille q) +1;;
```

2 Arbres binaires de recherche

8.



9. (a)
```
let rec maxABR t = match t with
|Vide -> failwith "l'arbre est vide"
|Noeud(_,x,Vide) -> x
|Noeud(_,x,d) -> maxABR d;;
```

(b)
```
let estABR t =
let rec parcours t = match t with
    |Vide -> (0,true,0)
    |Noeud(Vide,x,Vide) -> (x,true,x)
    |Noeud(Vide,x,d) ->  let (xd,ed,yd) = parcours d in
                         if (ed && (x <= xd)) then (x,true,xd)
                         else (0,false,0)
    |Noeud(g,x,Vide) -> let (xg,eg,yg) = parcours g in
                        if (eg && (yg <= x)) then (yg,true,x)
                        else (0,false,0)
    |Noeud(g,x,d) -> let (xg,eg,yg) = parcours g in
                     let (xd,ed,yd) = parcours d in
                     if (eg && ed && (yg <= x) && (x <= xd))
                         then (yg,true,xd)
                     else (0,false,0) in
let (a,b,c) = parcours t in b;;
```

10. (a) Soit A un arbre fortement équilibré de hauteur $h > 0$.

Montrons par récurrence sur la hauteur des nœuds que pour tout entier naturel $k < h$, il y a 2^k nœuds de hauteur k dans A.

— **Initialisation :** il y a bien $2^0 = 1$ nœud de hauteur 0, la racine.

— **Hérédité :** Soit k un entier tel que $0 < k < h$ et tel qu'il y ait 2^{k-1} nœuds de hauteur $k-1$ dans A. Chacun de ces nœuds a deux fils d'après la définition d'un arbre fortement équilibré. Il y a donc 2^k nœuds de hauteur k dans A.

— **Conclusion :** pour tout entier $k < h$, il y a donc bien 2^k nœuds de hauteur k dans A.

Par ailleurs, A étant de hauteur h, il y a au moins un nœud de hauteur h et au plus 2^h nœuds de hauteur h.

Ainsi, on obtient

$$2^{h+1} = \sum_{k=0}^{h} 2^k + 1 > n \geq \sum_{k=0}^{h-1} 2^k + 1 = 2^h$$

La fonction logarithme étant strictement croissante :

$$h + 1 > \log_2(n) \geq h$$

En conséquence :

$$h = E(\log_2(n))$$

(b)
```
let abr_eq tab =
    let n = Array.length(tab) in
    let rec parcours i j =
        if i=j then Noeud(Vide,tab.(i),Vide)
        else if i=j-1 then Noeud(Noeud(Vide,tab.(i),Vide),tab.(j),Vide)
        else if i=j-2 then Noeud(Noeud(Vide,tab.(i),Vide),tab.(i+1)
                                                ,Noeud(Vide,tab.(j),Vide))
        else let k = (j-i)/2 in
             Noeud((parcours i (i+k-1)),tab.(i+k),(parcours (i+k+1) j))
    in parcours 0 (n-1);;
```

À chaque appel de la fonction parcours, au moins un élément du tableau est traité et ne sera plus considéré. La complexité pour un tableau de taille `n` est donc en $O(\texttt{n})$. Il est de plus aisé de vérifier que l'arbre renvoyé est bien fortement équilibré.

11. CCP 2011 :

(a) Dans l'exemple proposé on a successivement les appels suivants :

`aux1 [2; 1; 3] Vide` qui appelle

`ajouter 2 Vide` qui renvoie

`a1=Noeud(Vide,2,Vide)`

`aux1 [1; 3] a1` qui appelle

`ajouter 1 a1` qui renvoie `a2=Noeud(Noeud(Vide,1,Vide),2,Vide)`

`aux1 [3] a2` qui appelle

`ajouter 3 a2` qui renvoie

`a3=Noeud(Noeud(Vide,1,Vide),2,Noeud(Vide,3,Vide))`

puis `aux2 a3` qui

`1/` appelle `aux2 Noeud(Vide,1,Vide)` et `aux2 Noeud(Vide,3,Vide)` qui renvoient respectivement [1] et [3]

`2/` renvoie `[1]@(2::[3])=[1; 2; 3]`

(b) On démontre par récurrence sur la taille de l'arbre de recherche `p` que si `r` est le résultat de l'appel `ajouter e p`, `r` est également un arbre de recherche dont l'ensemble (avec répétition) d'étiquettes est égal à celui de `p` plus l'étiquette `e`.

— **Initialisation :** si `p` est l'arbre vide, alors `r` est un arbre réduit à sa racine d'étiquette `e`, qui est bien de recherche.

— **Hérédité :** soit `n` un entier naturel tel que la propriété soit vraie pour tout `k <` `n`. Soit `p` un arbre de recherche de taille `n`. Soit `x` la racine de l'arbre de recherche `p`.

— Si `x = e`, l'arbre `r` a pour fils gauche un arbre `r0` de racine `x`, sans fils droit et de fils gauche `G(p)`. Mais les éléments de `G(p)` sont inférieurs ou égaux à `x = e`, donc `r0` est bien de recherche et tous ses éléments sont inférieurs ou égaux à `e`. `r` a pour racine `e`, pour fils gauche `r0` dont on vient de parler et pour fils droit `D(p)` qui est bien de recherche et dont tous les éléments sont supérieurs strictement à `x = e` : on a bien prouvé que `r` est de recherche et que ses étiquettes sont `e` et les étiquettes qui figuraient déjà dans `p`.
— Si `e < x`, la racine de `r` est `e`, strictement inférieure à tous les éléments de son fils droit qui est l'arbre de recherche `D(p)`. Le fils gauche de `r` est l'arbre obtenu par l'appel `ajouter e g`, où `g = G(p)`. Par hypothèse de récurrence, cet arbre est de recherche, et ses étiquettes sont `e` et les étiquettes de `G(p)`, toutes inférieures ou égales à `x`. Bref, `r` est de recherche et ses étiquettes sont bien `e` et les étiquettes qui figuraient déjà dans `p`.
— Si `e > x`, la racine est `e`, supérieure ou égale à tous les éléments de son fils gauche qui est l'arbre de recherche `G(p)`. Le fils droit de `r` est l'arbre obtenu par l'appel `ajouter e d`, où `d = D(p)`. Par hypothèse de récurrence, cet arbre est de recherche, et ses étiquettes sont `e` et les étiquettes de `D(p)`, toutes strictement supérieures à `x`. Bref, `r` est de recherche et ses étiquettes sont bien `e` et les étiquettes qui figuraient déjà dans `p`.

— **Conclusion :** on a bien prouvé que si `p` est un arbre binaire de recherche, `r` est également de recherche.

(c) Soient $\texttt{p} = [\texttt{p}_1; \ldots; \texttt{p}_\texttt{m}]$ une liste d'entiers est $\texttt{r} = [\texttt{r}_1; \ldots; \texttt{r}_\texttt{n}]$ la liste obtenue par l'appel de `trier p;;`

i. Montrons d'abord par récurrence que pour tout $k \in \mathbb{N}$, pour tout arbre a de taille k, la fonction `ajouter`, appliquée à a renvoie un arbre de taille $n+1$.
— **Initialisation :** s'est vrai pour l'arbre `Vide` (ligne 2).
— **Hérédité :** soit $k \in \mathbb{N}$ tel que la propriété est vraie pour tout $l \leq k$ et montrons qu'elle est vraie pour $k+1$. Soit a un arbre de taille $k+1$, si on est dans le cas `v=e`, le résultat est bien de taille "taille de a+1" et sinon, par hypothèse de récurrence, on sait que le résultat est constitué d'un sous arbre auquel on a ajouté un élément et du reste de l'arbre.
— **Conclusion :** la fonction renvoie donc toujours un arbre de taille "taille de a +1".

On remarque également que la fonction `aux1` préserve la somme des tailles de ses paramètres. En effet, à chaque appel récursif, l'élément en tête de la liste est ajouté à l'arbre. Lorsque le résultat est renvoyé, la liste est vide, donc l'arbre est de la taille de s.

Pour finir, on montre par récurrence que la fonction `aux2` renvoie un résultat de la taille de son argument.
— **Initialisation :** Si l'arbre en paramètre est vide, la liste vide est renvoyée.
— **Hérédité :** soit $k \in \mathbb{N}$ tel que la propriété est vraie pour tout $l \leq k$ et montrons qu'elle est vraie pour $k+1$. Soit a un arbre de taille $k+1$. Le résultat renvoyé est alors la concaténation de deux listes dont la somme des longueurs est égale à $k+1$ (par hypothèse de récurrence).
— **Conclusion :** la fonction `aux2` préserve donc bien la taille de son argument.

En conséquence, on obtient que la liste `r` fait la même longueur que la liste `p`, c'est-à-dire que $m = n$.

ii. Même raisonnement que précédemment :
— la fonction ajouter renvoie un arbre avec les mêmes éléments que le second argument plus l'élément donné en premier argument ;
— `aux1` préserve l'union (avec les mêmes occurences) des éléments des arguments ;
— `aux2` préserve également les occurrences de chaque élément.
En conclusion, les nombres d'occurrences des p_i sont les mêmes dans `p` et dans`r`.

iii. Par récurrence directe et la question b, on obtient que l'arbre renvoyé par `aux1` est un arbre binaire de recherche.

Montrons par récurrence sur la taille de l'arbre en argument que la liste renvoyée par `aux2` est croissante.
— **Initialisation :** si l'arbre en argument est `Vide`, la liste renvoyée est vide. Elle est donc bien croissante.
— **Hérédité :** soit $k \in \mathbb{N}$ tel que la propriété est vraie pour tout $l \leq k$ et montrons qu'elle est vraie pour $k+1$. Soit a un arbre de taille $k+1$. Le résultat renvoyé est la concaténation `(aux2 g) @ (v::(aux2 d))`. Par hypothèse de récurrence, on sait que les deux appels à `aux2` renvoient des listes croissantes. De plus, on a montré que les éléments de ces listes sont exactement les éléments des arbres passés en argument. Par définition des arbres binaires de recherche, on sait que les éléments dans `aux2 g` sont tous inférieurs à `v` et les éléments dans `aux2 d` sont tous supérieurs à `v`. Le résultat est donc une suite croissante.
— **Conclusion :** le résultat d'un appel à `aux2` est donc la liste croissante des éléments de l'arbre en argument.
On obtient finalement que la fonction trier a bien le rôle attendu.Le résultat de l'appel `trier p` est bien une liste croissante.

(d) Les fonctions `ajouter`, `aux1` et `aux2` terminent car tous les appels récursifs se font sur des objets de taille strictement inférieure et le cas vide est toujours traité comme cas de base. La fonction `trier` termine également comme la composition de fonctions qui terminent.

(e) Pour `aux1` et `aux2`, il y aura toujours un nombre d'appels récursifs linéaire en la taille de l'argument. Par contre, le nombre d'appels récursifs à `ajouter` dépend de la liste initiale. Comme il est au moins linéaire en la taille de l'argument, la complexité de trier (somme des complexité des trois fonction) est du même ordre que le nombre d'appels récursifs de la fonction `ajouter`.

Dans le meilleur cas, on a toujours "`v=e`" ce qui permet de ne pas faire d'appel récursif en mettant le nœud à ajouter à la racine. La liste initiale est donc une liste constante (tous les éléments sont égaux). On remarque que l'arbre construit est alors complètement déséquilibré ce qui peut être contre-intuitif.

Dans le pire cas, on passe au maximum dans le else. Cela nécessite également un déséquilibre de l'arbre avec une liste strictement croissante ou strictement décroissante ce qui force à systématiquement parcourir tous les éléments déjà ajoutés pour ajouter le suivant. Le nombre d'appels récursifs à ajouter est alors la somme des premiers entiers, c'est donc pour une liste de longueur $n \in \mathbb{N}$ en $O(n^2)$.

12. (a) i.
```
let rec fusion a b = match a with
    |Vide -> b
    |Noeud(g,y,d) -> fusion g (fusion d (insert y b));;
```

ii. La fonction `insert` est appelée exactement une fois par nœud de `a`, c'est-à-dire `n` fois. La fonction `insert` étant de complexité $O(\log n)$ car l'arbre dans lequel on insère a entre n et $2n$ noeuds, on obtient bien la complexité $O(n \log n)$ pour la fonction `fusion`.

(b) i.
```
let rec coupure a x = match a with
    |Vide -> Vide,Vide
    |Noeud(g,y,d) when y>x -> let g1,g2 = coupure g x in
                             g1,Noeud(g2,y,d)
    |Noeud(g,y,d) -> let d1,d2 = coupure d x in
                     Noeud(g,y,d1),d2;;
```

À chaque appel récursif, la hauteur du paramètre décroît strictement. La complexité est donc linéaire en la hauteur de l'arbre.

ii.
```
let rec fusion2 a b = match a with
    |Vide -> b
    |Noeud(g,y,d) -> let bg,bd = coupure b y in
                     Noeud((fusion2 g bg),y,(fusion2 d bd));;
```

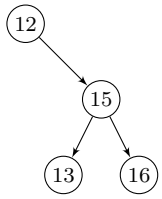
Avec cette implémentation, s'il y a le même élément dans les deux arbres, alors la fusion contiendra deux occurrences de cet élément. En revanche, dans la première version, l'existence de doublon dans le résultat dépend de la façon dont est implémenté la fonction d'insertion.

iii. Notons $C(n)$ la complexité de la fonction `fusion2` pour des arbres de taille n. La fonction $C(n)$ satisfait grossièrement (en supposant que les hauteurs des arbres sont logarithmiques en leurs tailles) la relation de récurrence suivante :

$$C(n) \leq 2C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(\log n)$$

On peut donc espérer une complexité en $O(n)$.

13.



Si on cherche le nombre 16 dans cet arbre, on voit aisément que l'étiquette 12 appartient à l'ensemble B alors que 13 appartient à l'ensemble A. On est bien face à un contre-exemple.

14. (a)
```
let rech t x =
    let u = ref t in
    let balise = ref true in
    let res = ref false in
    while !balise do
        match !u with
```

```
            |Vide -> balise := false;
            |Noeud(z) when x = z.etiquette -> (balise := false ;
                                               res := true)
            |Noeud(z) when x < z.etiquette -> (match z.fg with
                |Vide -> balise := false
                |Noeud(zg) -> u := Noeud({etiquette = zg.etiquette;
                                          fg = zg.fg;fd = zg.fd}))
            |Noeud(z) -> match z.fd with
                |Vide -> balise := false
                |Noeud(zd) -> u := Noeud({etiquette = zd.etiquette;
                                          fg = zd.fg;fd = zd.fd})
    done;
    !res;;
```

Attention, les parenthèses autour du `match` sont nécessaires. à défaut, les cas qui suivent sont associés au dernier match rencontré. De plus, l'utilisation d'une référence et la création d'un nouveau noeud à chaque étape du parcours pour `u` permet de ne pas modifier le paramètre.

(b)
```
let ins t x =
    let u = ref t in
    let balise = ref true in
    while !balise do
        match !u with
            |Noeud(z) when x = z.etiquette -> balise := false
            |Noeud(z) when x < z.etiquette -> (match z.fg with
                |Vide-> balise := false; z.fg <-Noeud({etiquette=x;
                                                       fg=Vide;fd=Vide})
                |Noeud(zg) -> u := Noeud({etiquette = zg.etiquette;
                                          fg = zg.fg;fd = zg.fd}))
            |Noeud(z) -> match z.fd with
                |Vide-> balise := false; z.fd <-Noeud({etiquette = x;
                                                      fg = Vide;fd = Vide})
                |Noeud(zd) -> u:=Noeud({etiquette = zd.etiquette;
                                        fg = zd.fg;fd = zd.fd})
    done;;
```

3 Tas

15. Conversion

(a) On utilise une fonction auxiliaire qui prend en argument l'indice à partir duquel on recopie.

```
let arbre_de_tas t =
    let rec sous_arbre i =
        if i > t.(0) then Vide
        else N(t.(i), sous_arbre (2*i), sous_arbre (2*i+1)) in
    sous_arbre 1;;
```

(b) On commence par déterminer la taille de l'arbre, puis on utilise une fonction auxiliaire qui prend en argument l'indice à partir duquel on doit recopier le tas.

```
let rec taille a = match a with
    | Vide
    | N(_, g, d) -> 1 + taille g + taille d;;

let tas_d_arbre a =
    let n = taille a in
    let t = Array.make (n+1) n in
    let rec remplissage i a = match a with
        | Vide
        | N(x ,g ,d) -> t.(i) <- x; remplissage (2*i) g;
                        remplissage (2*i+1) d in
    remplissage 1 a;
    t;;
```

16. (a) Voici les fonctions nécessaires à l'implémentation de tas à l'aide de tableau redimensionnables. Cette fois les tas seront implémentés par des références de tableaux et non des tableaux.

```
type tas = int array ref
let creeVide()= (ref [|0|] : tas);;
let estVide tab = (!tab).(0) = 0;;
```

On a encore besoin d'une fonction pour échanger deux éléments d'un tableau, d'une fonction pour descendre un élément d'un tas qui serait trop haut et inversement d'une fonction pour remonter un élément trop haut.

```
let echange tab i j =
    let t = !tab in
    let m = t.(i) in
    t.(i)<-t.(j);
    t.(j)<-m;;
let montee tab =
    let t = !tab in
    let x = t.(t.(0)) in
    let n = ref t.(0) in
    while !n<>1 && x>t.(!n/2) do
        echange tab !n (!n/2);
        n:=!n/2
    done;;
let rec descente tab k =
    let t = !tab in
    if (2*k < t.(0)) && ( t.(k) < t.(2*k)) ||
                        (2*k+1 < t.(0)) &&  (t.(k) < t.(2*k+1)) then
        if (2*k+1 < t.(0)) && t.(2*k+1) > t.(2*k) then
            (echange tab k (2*k+1);descente tab (2*k+1))
        else (echange tab k (2*k);descente tab (2*k));;
```

On a fait le choix d'une programmation itérative pour `montee` et récursive pour `descente` de manière arbitraire pour illustrer les deux possibilités.
On peut écrire la fonction pour ajouter un élément. Quand le tableau est trop petit on va créer un tableau deux fois plus grand.

```
let ajoute x tab =
    let t = !tab in
    if t.(0) <> (Array.length t -1) then
        (t.(t.(0)+1) <- x; t.(0) <- t.(0)+1)
    else (let u = Array.make (2*(t.(0)+1)) 0 in
        for k = 1 to t.(0) do
            u.(k)  <- t.(k);
        done;
        u.(0) <- t.(0)+1;
        u.(t.(0) + 1) <- x;
        tab:=u);
    montee tab;;
```

La fonction `racine` supprime la racine du tas. Si le tableau est alors occupé à moins du quart, on le recopie dans un tableau deux fois plus petit.

```
let racine tab =
    let t = !tab in
    if estVide tab then failwith "le tas est vide"
    else (echange tab (t.(0)) 1;
        t.(0) <- t.(0)-1;
        descente tab 1;
        if t.(0) < ((Array.length(t)) / 4) then
            (let u = Array.make (Array.length t/2) 0 in
            for k = 0 to t.(0) do
                u.(k) <- t.(k)
            done;
            tab:=u));;
```

(b) Pour un tas de taille n, lors de l'insertion, on risque dans le pire des cas de devoir faire autant d'échanges que la hauteur du tas pour maintenir la propriété de tas. L'opération a donc un complexité (dans le pire des cas) de $O(\log(n)$. De plus, dans le pire cas, on doit également recopier tous les éléments du tableau dans un nouveau tableau, la complexité est donc de l'ordre de $O(n)$.

(c) On effectue p insertions successives dans un tas initialement vide, et on note $\gamma(p)$ la moyenne des coûts (temporels) de ces insertions successives.
Calculons les coûts pour $p = 2^q$. Pour chaque k compris entre 1 et 2^q qui n'est pas une puissance de 2 on fait (au pire) $\log(k)$ opérations. Par contre pour les entiers qui sont une puissance de 2 on fait $\log(k) + k$ opérations. On en déduit que, le nombre $C(p)$ d'opérations nécessaires vérifie

$$\begin{aligned} C(p) &= \sum_{k=1}^{2^q} \log(k) + \sum_{k=0}^{q} 2^k \\ &\leq 2^q \log(2^q) + 2^{q+1} \end{aligned}$$

En divisant par p, $\gamma(p) = \frac{C(p)}{p} \leq q\log(2) + 2$. Il existe donc une constante C (en pratique tout nombre strictement supérieur à $\log(2)$ par exemple 1) tel que pour $p = 2^q$ assez grand, $\gamma(p) \leq C\log(p)$.

De plus, le nombre d'opérations de l'insertion de p éléments dans un tas vide est une fonction strictement croissante de p. De ce fait, pour p un entier, si on pose q tel que $2^q \leq p < 2^{q+1}$.

Dès lors

$$C(p) \leq C(2^{p+1}) \leq 2^{q+1}C\log(2^{q+1}).$$

Or $\frac{1}{p} \leq 2^{-q}$ et $q \leq \log(p)$ donc

$$\gamma(p) \leq \frac{2^{q+1}C\log(2^{q+1})}{2^q} \leq 2C(q+1) \leq 2C(\log(p)+1).$$

Finalement $\gamma(p) = O(\log(p))$.

17. (a) Il suffit d'entasser au fur et à mesure tous les éléments du tableau dans un tas initialement vide. On sait que le nombre d'opérations dans un tas de taille k est de complexité $O(\log(k))$ donc la complexité de l'insertion de n éléments dans un tas initialement vide est dominé par $n\log(n)$.

(b) Considérons l'invariant :

$I(i)$ = « tous les sous-arbre dont la racine est `t.(j)` avec $j > i$ sont des tas ».

— Au début du programme, $i = \lfloor \frac{n}{2} \rfloor$. De ce fait, pour $j > i$, les sous-arbres dont la racine est `t.(j)` sont des des arbres de taille 1 qui sont effectivement des tas.

— On suppose que $I(i)$ est vérifiée et on exécute `tamiser a i` qui n'est rien d'autre que la fonction descente du cours. Elle va faire redescendre (si nécessaire) l'étiquette du sommet i. De ce fait, après l'exécution, $I(i-1)$ est vérifiée.

— A la fin de l'exécution du programme, $I(-1)$ est vérifiée et donc on a un tas.

(c) La fonction `tamiser` est la fonction `descente` du cours. Elle a un complexité linéaire par rapport au sous arbre issu de i.

Au pire, pour un tas de hauteur h complet il y a, pour tout entier k compris entre 1 et h, 2^k nœuds de hauteur k. Chaque nœud de hauteur k nécessite au pire $h-k$ opérations. Il suffit de calculer

$$\sum_{k=0}^{h-1}(h-k)2^k = \sum_{k=1}^{h}k2^{h-k} = 2^{h-1}\sum_{k=1}^{h}k\left(\frac{1}{2}\right)^{k-1}.$$

Pour tout $x \in [\![0;1]\!]$, $\sum_{k=1}^{+\infty}kx^{k-1} = \frac{1}{(1-x)^2}$ donc pour tout $h \in \mathbb{N}^*$:

$$\sum_{k=1}^{h}kx^{k-1} < \frac{1}{(1-x)^2}.$$

Ainsi, pour $x = \frac{1}{2}$:

$$\sum_{k=1}^{h}k\left(\frac{1}{2}\right)^{k-1} < \frac{1}{(1-\frac{1}{2})^2} = 4.$$

Au final, on obtient $\sum_{k=1}^{h}(h-k)2^k < 4 \times 2^{h-1} = O(2^h) = O(n)$.

18. Arbres de priorité équilibrés

(a) On procède par récurrence forte sur la hauteur de l'arbre. Précisément on pose pour $h \geq 0$,
$\mathcal{P}(h) =$ « si a est un APE de hauteur h alors $2^h \leq N(a) \leq 2^{h+1} - 1$ »
— Soit a un APE de hauteur 0, on a $N(a) = 1$ et $\mathcal{P}(0)$ est vérifiée.
— Soit $h \geq 0$, on suppose que $\mathcal{P}(k)$ est vérifié pour tout $k \leq h$. Soit a un arbre de hauteur $h+1$ et g et d les sous-arbres de gauche et droite issus de la racine. L'un de deux est de hauteur h.
Justifions que nécessairement $H(g) = h$. En effet, g et d sont des APE de hauteur inférieur ou égale à h. On peut donc appliquer l'hypothèse de récurrence. Si $H(g) < h$ c'est-à-dire $H(g) \leq h-1$ et donc $H(d) = h$ alors $N(d) \geq 2^h$ et $N(g) \leq 2^{h-1+1} - 1 = 2^h - 1$. Cela est impossible.
On en déduit finalement que

$$N(a) = N(g) + N(d) + 1 \leq 2N(g) + 1 \leq 2(2^{h+1} - 1) + 1 = 2^{h+2} - 1$$

puis que

$$N(a) = N(g) + N(d) + 1 \geq 2N(g) \geq 2^{h+1}$$

Finalement $\mathcal{P}(h+1)$ est vérifiée.
— En conclusion, pour tout $h \geq 0$, $\mathcal{P}(h)$ est vérifiée.

(b) Afin de ne parcourir qu'une seule fois l'arbre on construit une fonction auxiliaire `aux : 'a arbre -> int * 'a * bool` qui renvoie un triplet `(n,m,b)`. Le premier argument est la taille de l'arbre, le deuxième est le maximum des étiquettes des noeuds (c'est-à-dire la valeur de l'étiquette de la racine dans le cas où l'arbre est un APE) finalement le dernier argument est un booléen qui dit si l'arbre est un APE.

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

let verifie a =
  let rec aux a = match a with
      | N (Vide, x, Vide) -> (1, x, true)
      | N (N (Vide, y, vide), x, Vide) -> (2, y, y <= x)
      | N (g, x, d) ->
       let (ng, mg, bg) = aux g and (nd, md, bd) = aux d in
          (ng + nd + 1, x, bg && bd && (ng - nd) <= 1 &&
          (ng - nd) >= 0 && (x >= (max mg md)))
  in let (n, m, b) = aux a in b;;
```

(c) Le principe est que si on veut insérer x dans un arbre de la forme $N(g, r, d)$, on compare x à r.
— Si $x \geq r$, on insère r dans g ou d (voir plus bas) et on renvoie $N(\_, x, \_)$.
— Si $x < r$, on insère x dans g ou d (voir plus bas) et on renvoie $N(\_, r, \_)$.

Une méthode pour savoir si on veut insérer dans g ou d, on peut avoir recours au calcul de taille des deux arbres et insérer dans d si $N(d) < N(g)$ et insérer dans g dans le cas contraire. L'utilisation de la fonction de calcul de taille est peu efficace. Un meilleure méthode consiste à insérer dans le sous-arbre de droite puis d'échanger l'arbre de droite et l'arbre de gauche. Précisément si on note d' l'arbre de g et g' l'arbre de droite dans lequel on a inséré l'élément, on renvoie $N(g', x, d')$.
— Si on avait $N(d) = N(g) - 1$ on a alors $N(g') = N(d')$.
— Si on avait $N(d) = N(g)$ on a alors $N(g') = N(d') + 1$.

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

(d) On procède comme dans le cas de la suppression de la racine des tas. On retire de l'arbre l'élément qui doit être enlevé par rapport à la condition squelettique de l'arbre ; ici le sommet le plus à gauche (voir plus bas), on place alors l'étiquette de ce sommet à la place de la racine de l'arbre puis, on redescend éventuellement cette étiquette (en s'inspirant de la procédure d'insertion) pour obtenir de nouveau la condition sur les étiquettes.

Quand on veut enlever le sommet le plus à gauche de l'arbre, comme dans le cas de l'insertion, on inverse les deux sous-arbres pour garantir que celui de droite à le même nombre de sommets (si le sous-arbre de gauche avait un sommet de plus) ou un de moins (si le sous-arbre de gauche avait le même nombre de sommets).

La fonction `cherche_pag : arbre -> (int * arbre)` renvoie l'étiquette du sommet le plus à gauche et l'arbre obtenu en l'enlevant (on fera attention à l'inversion entre l'arbre gauche et droite dans `N(d,x,ab)`).

La fonction `transforme : arbre -> arbre` fait descendre l'étiquette d'un arbre obtenu en remplaçant l'étiquette de la racine d'un APE par une autre valeur afin d'obtenir de nouveau un APE.

```
let extraction arbre =
 let rec cherche_pag a = match a with
   | Vide -> failwith "cela ne doit pas arriver"
   | N (Vide, x, Vide) -> (x, Vide)
   | N (g, x, d) -> let (et, ab) = cherche_pag g in (et, N (d, x, ab))
 in
 let rec transforme a = match a with
   | Vide -> Vide
   | N (Vide, _, _) -> a
   | N (N (gg, yg, dg), x, Vide) when x > yg -> a
   | N (N (gg, yg, dg), x, Vide) -> N (N (gg, x, dg), yg, Vide)
   | N (N (gg, yg, dg), x, N (gd, yd, dd)) when (x >= yg && x >= yd) -> a
   | N (N (gg, yg, dg), x, N (gd, yd, dd)) when (yg >= x && yg >= yd) ->
   N (transforme (N (gg, x, dg)), yg, N (gd, yd, dd))
   | N (N (gg, yg, dg), x, N (gd, yd, dd)) when (yd >= x && yd >= yg) ->
   N (N (gg, yg, dg), yd, transforme (N (gd, x, dd)))
 in
 match arbre with
  | Vide -> failwith "l'arbre est vide"
  | N (Vide, x, Vide) -> Vide
  | N (g, x, d) -> let (et, ab) = cherche_pag g in transforme (N (ab, et, d));;
```