1) a)
```
let rec insertionEns v (e : ensemble) = match e with
  |[] -> ([v] : ensemble)
  |t::q -> if t = v then t::q else t::(insertionEns v q);;
```

b) Si on note n le nombre d'éléments de e. Dans le cas où l'on insère un élément qui n'est pas dans e, la fonction visite toute la liste d'où n appels récursifs. La complexité est linéaire.

2) a)
```
let rec eliminationEns v (e : ensemble) = match e with
  |[] -> ([] : ensemble)
  |t::q -> if t = v then q else t::(eliminationEns v q);;
```

b) — Si v est la tête de la liste la fonction termine immédiatement en renvoyant la queue (car on suppose qu'un élément v ne peut pas apparaitre plusieurs fois dans la liste)
— A l'inverse si v est le dernier élément de la liste (ou n'est pas dans la liste), on va devoir parcourir toute la liste.

On en déduit que la fonction s'execute en temps constant ($O(1)$) dans le premier cas et en temps linéaire par rapport à la taille de la liste ($O(n)$)dans le deuxième.

3) Soit a un arbre. S'il est vide ($a = \emptyset$) sa profondeur est 0. Sinon sa profondeur est égale à $1 + \max(|D(a)|, |G(a)|)$.

4) Soit A un arbre représentant un ensemble à n éléments. L'arbre a donc n nuds. On en déduit que

$$|A| \leq n$$

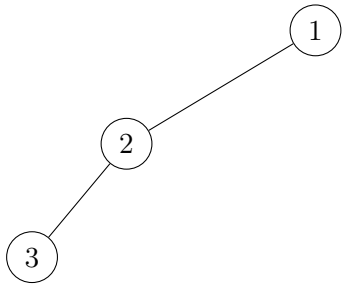
car $|A|$ est le nombre de nud sur une branche, il est donc inférieur au nombre total de nuds. De plus, en cas d'égalité l'arbre n'a qu'une seule branche.
A l'inverse pour une taille n donnée, l'arbre aura une profondeur minimale s'il a la forme d'un tas à savoir que tous les niveaux à part le dernier sont remplis.
Dans ce cas, il y a 2^{k-1} nuds de profondeur k (à part au dernier niveau). Cela se démontre par une récurrence immédiate avec le fait que la racine est de profondeur 1. On en déduit que si l'arbre est de profondeur p,

$$2^{p-1} = 1 + \sum_{k=1}^{p-1} 2^{k-1} \leq n \leq \sum_{k=1}^{p} 2^{k-1} = 2^p - 1.$$

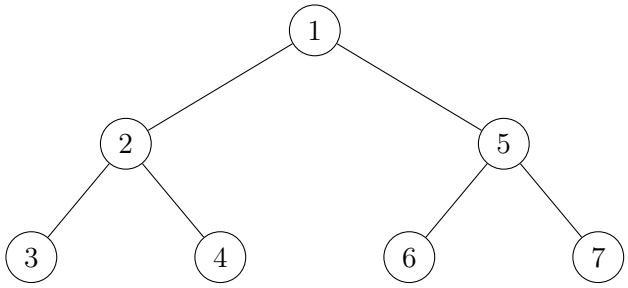
On en déduit que

$$\log_2(n+1) \leq |A| \leq 1 + \log_2(n).$$

Ce qui signifie que $|A| = 1 + E(\log_2(n))$.
En conclusion un arbre de taille n a une profondeur maximale de n et minimale de $1 + E(\log_2(n))$.



arbre de taille minimale



arbre de taille maximale

5) a) — On sait que $|a|$ est le nombre de nuds maximal sur une branche et que $\mathscr{R}(a)$ est le nombre de nuds gris sur une branche donc, $\mathscr{R}(a) \leq |a|$.
— La condition **P2** implique que sur une branche deux nuds se suivant ne peuvent pas être tous les deux blancs. De ce fait, une branche contenant p nuds gris peut contenir au plus $p + 1$ nuds blancs (un à chaque extrémité et un entre chaque couple de nuds gris). On en déduit que la longueur ℓ de la branche vérifie $\ell \leq 2p + 1$. Si on applique cela à la plus longue branche de l'arbre (vérifiant $\ell = |a|$) on obtient : $|a| \leq 2\mathscr{R}(a) + 1$.

b) Posons : $\mathscr{H}(n) =$« si a est un arbre bicolore composé de n noeuds, $2^{\mathscr{R}(a)} - 1 \leq n$ » pour $n > 0$. On va montrer par récurrence forte que pour tout entier n non nul, $\mathscr{H}(n)$ est vérifiée. Commençons par remarquer que si a est un arbre bicolore, ses sous-arbres $G(a)$ et $D(a)$ sont encore bicolores.
— Initialisation : Si l'arbre est réduit à sa racine $n = 1$, on a $\mathscr{R}(a) \in \{0, 1\}$ et $\mathscr{H}(1)$ est vérifiée.
— Hérédité : Soit n un entier au moins égal à 2, on suppose que $\mathscr{H}(p)$ est vraie pour tout $0 < p < n$. Notons n_g et n_d les tailles de $G(a)$ et de $D(a)$. Si la racine est blanche on a $\mathscr{R}(G(a)) = \mathscr{R}(D(a)) = \mathscr{R}(a)$ et $2^{\mathscr{R}(a)} - 1 \leq n_d \leq n$. Si la racine est grise on a $\mathscr{R}(a) = \mathscr{R}(G(a)) + 1 = \mathscr{R}(D(a))) + 1$. On a donc

$$2^{\mathscr{R}(a)} = 2^{\mathscr{R}(G(a))} + 2^{\mathscr{R}(D(a))} \leq (n_d + 1) + (n_g + 1) \leq n + 1.$$

C'est ce que l'on veut.
— Conclusion : Pour tout entier $n > 0$, $\mathscr{H}(n)$ est vérifiée.

c) On a déjà vu que pour tout arbre binaire de taille n, $E(\log_2(n)) \leq 1 + E(\log_2(n)) \leq |a|$.
Maintenant la question précédente nous donne, $2^{\mathscr{R}(a)} - 1 \leq n$ ce qui implique que $\mathscr{R}(a) \leq E(\log_2(n+1))$. En utilisant que de plus $|a| \leq 2\mathscr{R}(A) + 1$, on obtient $|a| \leq 2E(\log_2(n+1)) + 1$. En conclusion,

$$E(\log_2(n)) \leq |a| \leq 2E(\log_2(n+1)) + 1.$$

On en déduit que $|a| = O(\log_2(n))$.

6) On construit la fonction `rang` de manière recursive. On peut se contenter de visiter la branche de gauche car toutes les branches ont le même nombre de nuds gris.

```
let rec rang a = match a with
  |Vide -> 0
  |Noeud (c,fg,v,fd) when c = Blanc -> rang fg
  |Noeud (c,fg,v,fd) when c = Gris -> 1 + rang fg;;
```

7) On commence par écrire une fonction `racineGrise` qui permet de savoir si un arbre a une racine grise.

```
let racineGrise a = match a with
  |Vide -> true
  |Noeud(Gris,_,_,_) -> true
  |Noeud(Blanc,_,_,_) -> false;;
```

On écrit ensuite une fonction `valideEtRang` qui renvoie un couple de type `bool*int` tel que la première composante est un booléen qui vaut `true` si l'arbre est bicolore et `false` sinon, la deuxième composante est le rang de l'arbre (si c'est un arbre bicolore).

```
let rec valideEtRang a =
match a with
|Vide -> (true, 0)
|Noeud(Gris,fg,v,fd) -> let (bg,rg) = valideEtRang fg
   and (bd,rd) = valideEtRang fd in
   if bd && bg && rg = rd
   then (true, rg+1)
   else (false,0)
|Noeud(Blanc,fg,v,fd) ->  let (bg,rg) = valideEtRang fg
   and (bd,rd) = valideEtRang fd in
   if racineGrise fg && racineGrise fd && bg && bd && rg=rd
   then (true, rg)
   else (false,0);;

let validationBicolore a = let (b,r) = valideEtRang a in b;;
```

8) Dans le pire des cas qui est le cas d'un arbre bicolore, la fonction doit parcourir tout l'arbre (une fois). La complexité est donc linéaire en la taille de l'arbre.

9) On écrit une fonction `valideMinMax` qui teste si un arbre est un arbre binaire de recherche et renvoie l'étiquette minimale et l'étiquette maximale. Dans le cas générale, la fonction teste les deux sous-arbres $G(a)$ et $D(a)$. Si ce sont des arbres binaires de recherche, que le maximum de $G(a)$ est strictement inférieur à $\mathscr{E}(a)$ et que le minimum de $D(a)$ est strictement inférieur à $\mathscr{E}(a)$ alors on renvoie le booléen `true` ainsi que le minimum (qui est le minimum de $G(a)$) et le maximum (qui est le maximum de $D(a)$).

```
let rec valideMinMax a = match a with
    |Vide ->(true,0,0)
    |Noeud(c,Vide,v,Vide) -> (true,v,v)
    |Noeud(c,Vide,v,fd) -> let (bd,mind,maxd) = valideMinMax fd in
    if  bd  && (v < mind) then (true, v ,maxd) else (false,0,0)
    |Noeud(c,fg,v,Vide) -> let (bg,ming,maxg) = valideMinMax fg in
    if bg && (maxg < v)  then (true, ming,v)  else (false,0,0)
    |Noeud(c,fg,v,fd) -> let (bg,ming,maxg) = valideMinMax fg
       and (bd,mind,maxd) = valideMinMax fd in
    if bg && bd && (maxg < v) && (v < mind)
    then (true, ming,maxd)  else (false,0,0);;

let validationABR a = let (b,ma,mi) = valideMinMax a in b;;
```

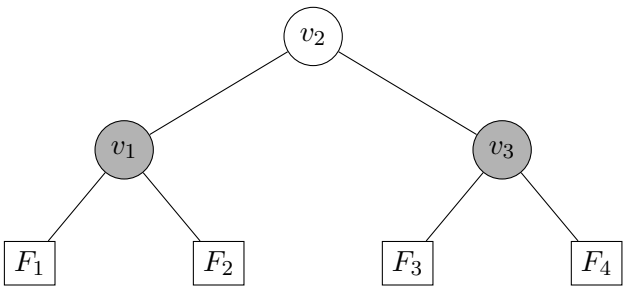
10) a) On réalise l'insertion dans un arbre binaire de recherche comme vu dans le cours en spécifiant la couleur blanche quand on insère l'élément.

```
let rec insertionABR v a = match a with
  |Vide -> Noeud(Blanc, Vide, v, Vide)
  |Noeud(c,fg,w,fd) -> if v = w then Noeud(c,fg,w,fd)
       else
 if v < w then Noeud(c,insertionABR v fg, w, fd)
 else Noeud(c,fg,w,insertionABR v fd);;
```

b) On insère un élément dans un arbre binaire de recherche a.
   — Si l'élément inséré est égal à la racine de l'arbre binaire de recherche, on ne fait rien. L'opération se fait en temps constant.
   — Si l'élément est finalement inséré en bas de la plus grande branche, l'opération nécessite $|a|$ appels recursifs.

11) Comme on insère un nud blanc, les conditions **P1** et **P3** ne peuvent pas devenir fausse. Par contre **P2** peut devenir fausse si on insère notre nouveau nud blanc comme fils d'un nud blanc.

12) On suppose que F_1, F_2, F_3 et F_4 sont des arbres bicolores de rang n.

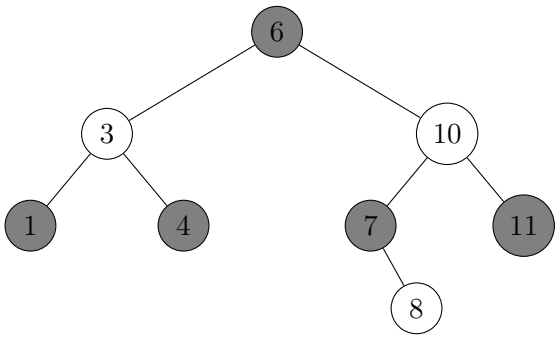


   — La condition **P1** est vérifiée
   — La condition **P2** est vérifiée car les deux fils du nud blanc v_2 sont gris. Les autres nuds blancs sont dans un des arbres F_1, F_2, F_3 ou F_4 et leurs fils sont gris car ce sont des arbres bicolores.

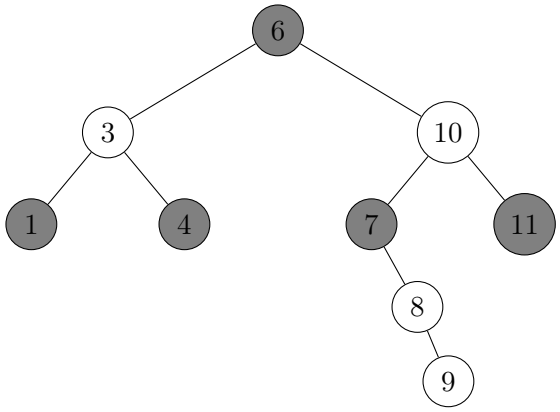
— La condition **P3** est vérifiée, chaque branche a $n+1$ nuds gris, n qui sont dans F_1, F_2, F_3 ou F_4 et le dernier est v_1 ou v_3.

13)

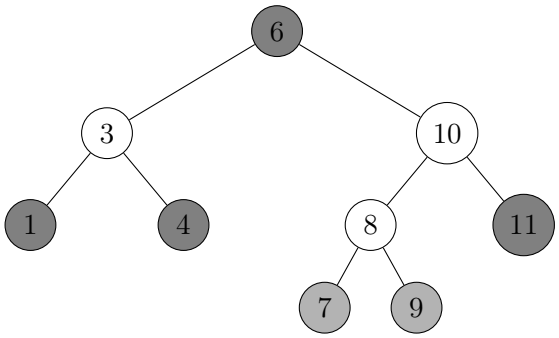
On considère l'arbre



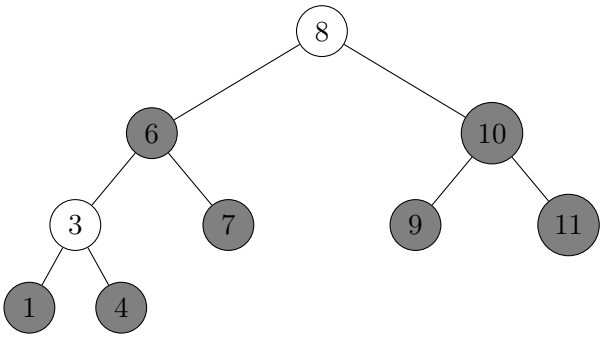
Après insertion on obtient



Après une correction blanche de type 2



On applique une correction de type 4.



14) Montrons pour commencer que si on applique une correction blanche à un arbre binaire de recherche on obtient toujours un arbre binaire de recherche. Dans les quatre cas considérés, pour tout a_1 nud de F_1, a_2 nud de F_2, a_3 nud de F_3 et a_4 nud de F_4 on a

$$a_1 < v_1 < a_2 < v_2 < a_3 < v_3 < a_4.$$

Ceci implique que l'arbre obtenu est un arbre binaire de recherche.

Maintenant, si, lors de l'insertion on viole la condition **P2** en insérant une liaison Blanc - Blanc entre un nud de profondeur p et un nud de profondeur $p+1$ de l'arbre, on voit qu'après la transformation blanche, le sous arbre issu des nuds de profondeur p sont des arbres bicolores. Par contre, on a pu créer une liaison Blanc - Blanc entre des nuds de profondeur $p-2$ et $p-1$. En réitérant le procédé on « remonte » l'enventuelle liaison Blanc-Blanc qui finira par disparaitre. Au pire on arrive au fait que la racine et qu'un de ses fils sont blancs et il suffit dans ce cas de colorer la racine en gris.

15) On vient de voir que la règle **P2** était violée en présence d'un nud blanc et qu'une transformation blanche permet de faire remonter ce problème le long de la blanche on aura besoin au plus d'autant de transformation blanche qu'il y a des nuds blancs sur la branche. Or le nombre $|A| - \mathscr{R}(A)$ est égal au nombre maximal de nuds blancs sur une branche.

16)
```
let correctionBlanche a = match a with
  |Noeud(Gris, Noeud(Blanc, Noeud(Blanc,f1,v1,f2),v2,f3), v3,f4)
     -> Noeud(Blanc,Noeud(Gris,f1,v1,f2),v2,Noeud(Gris,f3,v3,f4))
  |Noeud(Gris, f1, v1, Noeud(Blanc, f2, v2, Noeud(Blanc,f3,v3,f4)))
     -> Noeud(Blanc,Noeud(Gris,f1,v1,f2),v2,Noeud(Gris,f3,v3,f4))
  |Noeud(Gris, Noeud(Blanc, f1, v1, Noeud(Blanc,f2,v2,f3)), v3,f4)
      -> Noeud(Blanc,Noeud(Gris,f1,v1,f2),v2,Noeud(Gris,f3,v3,f4))
  |Noeud(Gris, f1, v1, Noeud(Blanc, Noeud(Blanc,f2,v2,f3),v3,f4))
   -> Noeud(Blanc,Noeud(Gris,f1,v1,f2),v2,Noeud(Gris,f3,v3,f4))
  |_ -> a;;
```

17) Il suffit de combiner les différentes fonctions.

```
let rec insertABRC v a = match a with
  |Vide -> Noeud(Blanc, Vide, v, Vide)
  |Noeud(c,fg,w,fd) -> if v = w then Noeud(c,fg,w,fd)
       else
     if v < w then correctionBlanche (Noeud(c,insertABRC v fg, w, fd))
     else correctionBlanche (Noeud(c,fg,w,insertABRC v fd));;
```

On remarquera que l'on applique la fonction correction blanche autant de fois que la profondeur de l'insertion ce qui est plus que suffisant.

Il reste à traiter le cas où la racine est blanche et un fils aussi.

```
let bonus a = match a with
  |Noeud(Blanc,Noeud(Blanc,f1,v1,f2),v2,f3) ->
     Noeud(Gris,Noeud(Blanc,f1,v1,f2),v2,f3)
  |Noeud(Blanc,f1,v1,Noeud(Blanc,f2,v2,f3)) ->
     Noeud(Gris,f1,v1,Noeud(Blanc,f2,v2,f3))
  |_ -> a;;
```

Finalement

```
let insertionABRC v a = bonus (insertABR v a);;
```