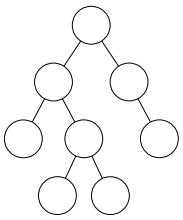
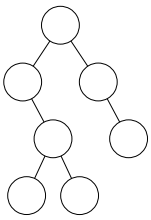
Exercice 1. Arbres binaires équilibrés

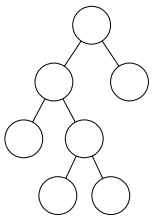
Question 1 –



bin_B1



bin_B2



bin_B3

Question 2 – L'arbre `bin_B1` est équilibré, mais pas `bin_B2` (problème de hauteur pour les sous-arbres du fils gauche de la racine), ni `bin_B3` (problème de hauteur pour les sous-arbres de la racine).

Question 3 – La fonction `aux` renvoie la hauteur de l'arbre ainsi qu'un booléen qui indique s'il est équilibré.

```
(* La fonction aux renvoie la hauteur de l'arbre ainsi qu'un booléen qui indique
   s'il est équilibré *)
let est_equilibre bin_B =
  let rec aux a = match a with
    | Vide -> -1, true
    | N_bin (_, g, d) ->
       let hg, bg = aux g and
           hd, bd = aux d in
       let est_eq = bg && bd && abs(hg - hd) <= 1 in
       1 + max hg hd, est_eq
  in
  let _, res = aux bin_B in res;;
```

Question 4.a – Puisque l'arbre est équilibré, on a trois cas :

→ Si $h_G = h_D + 1$ alors :

$$h_G + h_D = 2h_G - 1 = 2\max(h_G, h_D) - 1$$

→ Si $h_D = h_G + 1$. Ce cas se traite comme le précédent

→ Sinon, $h_D = h_G$ alors :

$$h_G + h_D = 2h_D = 2\max(h_G, h_D) \geqslant 2\max(h_G, h_D) - 1$$

Question 4.b – On montre la propriété par récurrence forte sur $n \in \mathbb{N}$:

→ Si $n = 0$ alors l'arbre est vide et on a $h = -1$ donc l'inégalité est vérifiée.

→ Sinon $n \in \mathbb{N}^*$. On suppose que tout arbre équilibré avec strictement moins de n noeuds vérifie l'inégalité. Soit A un arbre équilibré avec n noeuds, soit G le sous-arbre gauche, soit D le sous-arbre droit, soit n_G, h_G, n_D et h_D le nombre de noeuds et la hauteur de G et D. Alors :

$$n = 1 + n_D + n_G \qquad\qquad h = 1 + \max(h_D, h_G)$$

De plus, G et D sont équilibrés donc on peut leur appliquer l'hypothèse de récurrence :

$$h_G \leqslant \frac{3}{2}\log_2(n_G + 1) \qquad\qquad h_D \leqslant \frac{3}{2}\log_2(n_D + 1)$$

Avec la question précédente :

$$h_D + h_G \geqslant 2\max(h_G, h_D) - 1$$

Donc :

$$\max(h_G, h_D) \leqslant \frac{h_G + h_D + 1}{2}$$

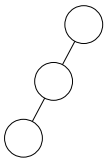
On obtient :

$$h = 1 + \max(h_D, h_G) \leqslant \frac{3}{2} + \frac{h_G + h_D}{2} \leqslant \frac{3}{2} + \frac{3}{2}\frac{\log_2(n_G + 1) + \log_2(n_D + 1)}{2}$$

Avec l'indication de l'énoncé :

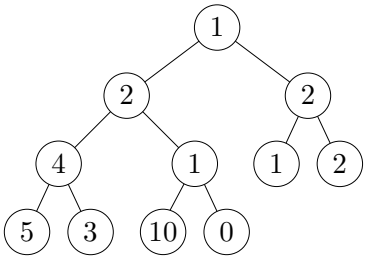
$$h \leqslant \frac{3}{2} + \frac{3}{2}\log_2\left(\frac{n_G + 1 + n_D + 1}{2}\right) = \frac{3}{2} + \frac{3}{2}\Big(\log_2(n+1) - 1\Big) = \frac{3}{2}\log_2(n+1)$$

Question 4.c – La réciproque n'est pas vraie. Par exemple l'arbre suivant vérifie l'inégalité mais n'est pas équilibré :

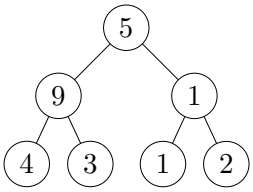


Exercice 2. Parcours d'arbres binaires stricts

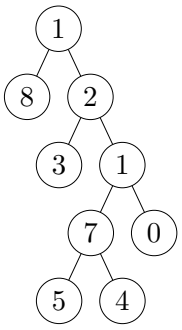
Question 1 –



`abs_A1`

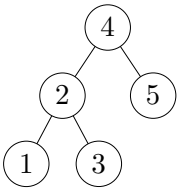


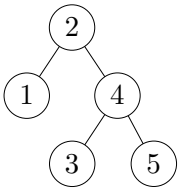
`abs_A2`



`abs_A3`

Question 2 – Les deux arbres suivants conviennent :





Question 3 – On applique la procédure vue en cours.

```
(* La fonction aux renvoie pref(a0) @ acc  *)
let get_parc_pre a0: 'a parc =
  let rec aux acc a = match a with
    | F_abs e -> (F e)::acc
    | N_abs (e, g, d) -> (N e)::(aux (aux acc d) g)
  in aux [] a0;;
```

```
(* La fonction aux renvoie inf(a0) @ acc  *)
let get_parc_inf a0: 'a parc =
  let rec aux acc a = match a with
    | F_abs e -> (F e)::acc
    | N_abs (e, g, d) -> aux (N e::aux acc d) g
  in aux [] a0;;
```

```
(* La fonction aux renvoie post(a0) @ acc  *)
let get_parc_post a0: 'a parc =
  let rec aux acc a = match a with
    | F_abs e -> (F e)::acc
    | N_abs (e, g, d) -> aux (aux (N e::acc) d) g
  in aux [] a0;;
```

Question 4 – La fonction `aux` prend en entrée une liste `li` et renvoie un couple `(A, pre4)` où `A` est un arbre et `pre4` est une liste vérifiant :

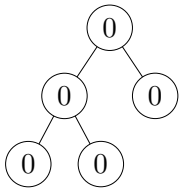
`li = Pre(A) @ pre4`

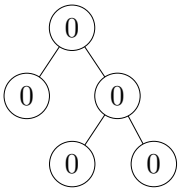
```
let abs_of_parc_pre (pre: 'a parc) =
  let rec aux li = match li with
    | [] -> failwith "abs_of_parc_pre: parcours invalide 1"
    | (F e)::pre1 -> (F_abs e), pre1
    | (N e)::pre2 -> let g, pre3 = aux pre2 in
                     let d, pre4 = aux pre3 in
                     N_abs (e, g, d), pre4
  in
  let a, vide = aux pre in
  if vide <> [] then failwith "abs_of_parc_pre: parcours invalide 2";
  a;;
```

Question 5 –

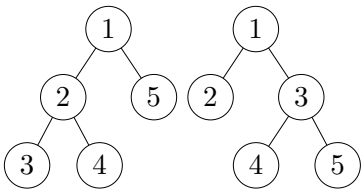
```
let suppr_FN (e : 'a elem_parc): 'a = match e with
  | N e -> e
  | F e -> e;;

let get_parc_pre2 a =
  List.map suppr_FN (get_parc_pre a);;

let get_parc_inf2 a =
  List.map suppr_FN (get_parc_inf a);;

let get_parc_post2 a =
  List.map suppr_FN (get_parc_post a);;
```

Question 6 – Les deux arbres suivants conviennent :

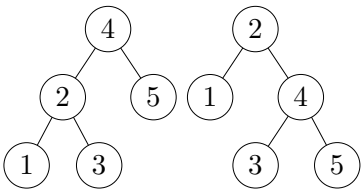




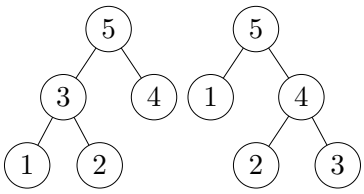
Question 7 – Les arbres suivants conviennent



Pour le parcours préfixe



Pour le parcours infixe



Pour le parcours postfixe

Question 8 – On montre la contraposée : si A_1 et A_2 sont deux arbres binaires stricts avec le même parcours préfixe et le même parcours infixe alors $A_1 = A_2$.
On sait déjà que A_1 et A_2 ont le même nombre de noeuds $n \in \mathbb{N}^*$ puisque leurs parcours font la même taille.
On montre la propriété par récurrence forte sur n :

→ Si $n = 1$ alors A_1 et A_2 sont réduits à une unique feuille. Comme leurs parcours préfixes sont égaux, leurs feuilles ont la même étiquette donc $A_1 = A_2$

→ Sinon $n > 1$. On suppose par hypothèse de récurrence que tout couple d'arbres avec strictement moins de n noeuds et ayant les mêmes parcours préfixes et infixes sont égaux.
Soit e_1 l'étiquette de la racine de A_1, soit G_1 son sous-arbre gauche et D_1 son sous-arbre droit. Soit e_2 l'étiquette de la racine de A_2, soit G_2 son sous-arbre gauche et D_2 son sous-arbre droit. Alors :

$$Pre(A_1) = [e_1]@Pre(G_1)@Pre(D_1) \qquad Pre(A_2) = [e_2]@Pre(G_2)@Pre(D_2)$$
$$Inf(A_1) = Inf(G_1)@[e_1]@Inf(D_1) \qquad Inf(A_2) = Inf(G_2)@[e_2]@Inf(D_2)$$

Comme $Pre(A_1) = Pre(A_2)$, $Inf(A_1) = Inf(A_2)$ et que les étiquettes sont distinctes deux à deux, on en déduit que :

$$e_1 = e_2 \qquad Pre(G_1)@Pre(D_1) = Pre(G_2)@Pre(D_2)$$
$$Inf(G_1) = Inf(G_2) \qquad Inf(D_1) = Inf(D_2)$$

De plus, les parcours préfixe et infixe d'un même arbre ont la même taille. Donc :

$$|Pre(G_1)| = |Inf(G_1)| = |Inf(G_2)| = |Pre(G_2)|$$
$$|Pre(D_1)| = |Inf(D_1)| = |Inf(D_2)| = |Pre(D_2)|$$

Finalement :

$$e_1 = e_2$$
$$Pre(G_1) = Pre(G_2) \qquad Pre(D_1) = Pre(D_2)$$
$$Inf(G_1) = Inf(G_2) \qquad Inf(D_1) = Inf(D_2)$$

Avec l'hypothèse de récurrence, on obtient :

$$e_1 = e_2 \qquad G_1 = G_2 \qquad D_1 = D_2$$

Donc $A_1 = A_2$.

Question 9 – La fonction `aux` prend en entrée deux listes (`pre_rgd, inf_grd`) et renvoie un triplet `(A, pre_autres, inf_autres)` tel que `A` est un arbre et `pre_autres`, `inf_autres` sont des listes vérifiant :

`pre_rgd = Pre(A) @ pre_autres` et `inf_grd = Inf(A) @ inf_autres`

Si on note `r` la racine de `A` et `(G, D)` les sous-arbres alors voici ce que contiennent les différentes variables du programme :

→ `pre_rgd = [r] @ Pre(G) @ Pre(D) @ pre_autres`

→ `pre_gd = Pre(G) @ Pre(D) @ pre_autres`

→ `pre_d = Pre(D) @ pre_autres`

→ `inf_grd = Inf(G) @ [r] @ Inf(D) @ inf_autres`

→ `inf_rd = [r] @ Inf(D) @ inf_autres`

→ `inf_d = Inf(D) @ inf_autres`

→ `r1 = r2` est l'étiquette de la racine.
Si `r1 <> r2`, c'est que les listes `pre_rgd` et `inf_grd` en entrée ne sont pas de la bonne forme.

```
let abs_of_pre_inf pre inf =
  let rec aux pre_rgd inf_grd = match pre_rgd, inf_grd with
    | r1::pre_autres, r2::inf_autres when r1 = r2 ->
       F_abs r1, pre_autres, inf_autres
    | r1::pre_gd, inf_grd ->
       let g, pre_d, inf_rd = aux pre_gd inf_grd in
       begin match inf_rd with
       | r2::inf_d when r1 = r2 ->
          let d, pre_autres, inf_autres = aux pre_d inf_d in
          N_abs (r1, g, d), pre_autres, inf_autres
       | _ -> failwith "abs_of_pre_inf: parcours invalides 1"
       end
    | _ -> failwith "abs_of_pre_inf: parcours invalides 2"
  in
  let res, vide1, vide2 = aux pre inf in
  match vide1, vide2 with
  | [], [] -> res
  | _ -> failwith "abs_of_pre_inf: parcours invalides 3";;
```