Exercice 1.

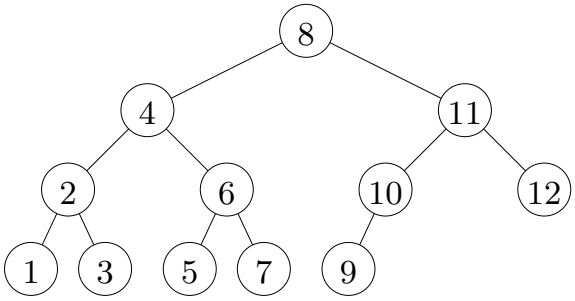
Question 1 –

→ La figure 1 n'est pas un arbre binaire donc ce n'est pas un arbre binaire de recherche.

→ La figure 2 n'est pas un arbre binaire de recherche car le noeud d'étiquette 11 est dans le sous-arbre gauche du noeud d'étiquette 10.

→ La figure 3 est un arbre binaire de recherche car la liste de ses étiquettes dans le parcours infixe est triée.

Question 2 – L'arbre suivant convient (il suffit de dessiner un arbre quelconque de hauteur 3 avec 12 noeuds, puis d'étiqueter les noeuds dans l'ordre du parcours infixe) :



Exercice 2. Dictionnaires et arbres binaires de recherche

Question 1 – On utilise un arbre binaire dont les noeuds sont étiquetés par les couples (clé, valeur) du dictionnaire. La structure d'ABR porte sur les clés : pour tout noeud de clé `c`, le sous-arbre gauche (resp. droit) ne comporte que des clés strictement inférieures (resp. supérieures) à `c`. On peut donc utiliser le type suivant :

```
type dict = (int * string) abr;;
```

Question 2 –

```
let create(): dict =
  Vide;;
```

Question 3 –

```
let rec find (d0: dict) c = match d0 with
  | Vide -> raise Not_found
  | N((c0, v0),_,_) when c = c0 -> v0
  | N((c0, v0),_,d) when c0 < c -> find d c
  | N(_,g,_) -> find g c;;
```

Question 4 –

```
let rec add (d0: dict) c v: dict = match d0 with
  | Vide -> N((c,v), Vide, Vide)
  | N((c0,v0), g, d) when c = c0 -> N((c,v), g, d)
  | N((c0,v0), g, d) when c < c0 -> N((c0,v0), add g c v, d)
  | N(e, g, d) -> N(e, g, add d c v);;
```

Question 5 –

```
let rec get_max (d : dict): (int * string) * dict = match d with
  | Vide -> failwith "Arbre vide"
  | N(e, g, Vide) -> e, g
  | N(e, g, d) -> let m, d2 = get_max d in
                  m, N(e, g, d2);;
```

Question 6 –

```
let rec remove (d0: dict) c: dict = match d0 with
  | Vide -> Vide
  | N((c0,v0), g, d) when c < c0 -> N((c0,v0), remove g c, d)
  | N((c0,v0), g, d) when c > c0 -> N((c0,v0), g, remove d c)
  | N(_, Vide, d) -> d
  | N(_, g, d) -> let m, g2 = get_max g in
                  N(m, g2, d);;
```

Question 7 – Afin de supprimer un noeud d'étiquette « `e: 'a` » dans un arbre binaire de recherche, on note `G` et `D` les sous-arbres gauche et droit de ce noeud, puis :

- Si `D` est vide, alors il suffit de remplacer le noeud par `G`.
- Sinon, on note `m` l'étiquette minimale présente dans `D`. Le noeud d'étiquette `m` n'a pas de sous-arbre gauche, on peut donc supprimer le noeud d'étiquette `m` en le remplaçant par son sous-arbre droit. Il suffit ensuite de remplacer l'étiquette `e` par `m`.

Question 8 – Pour chacune des fonctions, on fait au plus un appel récursif sur le sous-arbre droit ou le sous-arbre gauche. La complexité est donc en $\mathcal{O}(h)$ où h est la hauteur de l'arbre.

Question 9 –

```
type dict2 = dict ref;;

let create2(): dict2 =
  ref (create());;

let find2 (d: dict2) c =
  find !d c;;

let add2 (d: dict2) c v =
  let d2 = add !d c v in
  d := d2;;

let remove2 (d: dict2) c =
  let d2 = remove !d c in
  d := d2;;
```

Exercice 3. Tris et arbres binaires de recherche

Question 1 – On créé un arbre vide, puis on ajoute successivement tous les entiers à l'arbre en conservant la propriété d'arbre binaire de recherche. Il suffit ensuite de faire le parcours infixe de l'arbre pour obtenir la liste triée.

Question 2 – Soit n le nombre d'éléments dans la liste (et donc le nombre de noeuds dans l'arbre final). Le parcours infixe se calcule en temps $\Theta(n)$.

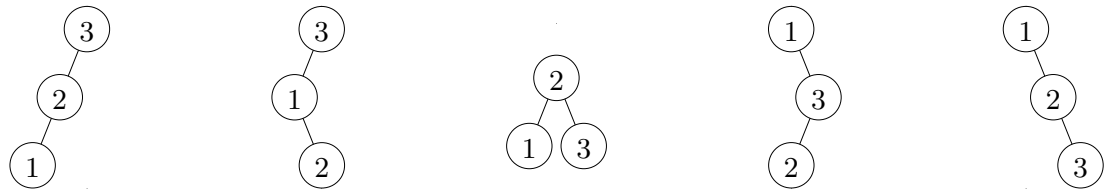
⋆ Pire cas. Dans le pire cas (la liste initiale est triée), à chaque ajout d'un noeud dans l'arbre, la hauteur de l'arbre est $m-1$ où m est le nombre de noeuds. La complexité totale pour créer l'arbre est de la forme $\sum_{m=0}^{n-1} C_m$ où $C_m = \Theta(m)$. Soit une complexité en $\Theta(n^2)$. La complexité totale pour trier la liste est en $\Theta(n^2) + \Theta(n) = \Theta(n^2)$

⋆ Meilleur cas. Soit $n \in \mathbb{N}$ et A l'arbre binaire presque-complet avec n noeuds étiquetés par $1, 2, \ldots, n$. Soit `L` la liste contenant les étiquettes de A dans l'ordre du parcours en largeur. La liste `L` correspond au meilleur cas pour l'algorithme.

À chaque ajout d'un noeud dans l'arbre, la hauteur de l'arbre est en $\Theta(\log m)$ où m est le nombre de noeuds. La complexité totale pour créer l'arbre est de la forme $\sum_{m=0}^{n-1} C_m$ où $C_m = \Theta(\log m)$. Soit une complexité en $\Theta(n \log n)$. La complexité totale pour trier la liste est en $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$

Exercice 4. Nombre d'arbres binaires de recherche

Question 1 – Voici les 5 arbres binaires de recherche ayant pour étiquettes $\{1, 2, 3\}$:



Question 2 – Pour construire un arbre binaire de recherche ayant pour étiquettes $\{1, 2, \ldots, n\}$, on peut construire un arbre binaire avec n noeuds sans étiquette puis étiqueter les noeuds avec $1, 2, \ldots, n$ dans l'ordre du parcours infixe. Le nombre d'arbres binaires de recherche ayant pour étiquettes $\{1, 2, \ldots, n\}$ est donc égal au nombre d'arbres binaires avec n noeuds sans étiquette.
Pour construire un arbre binaire avec $n+1$ noeuds sans étiquette, on construit une racine, puis on choisit $k \in [\![0, n]\!]$, on choisit le sous-arbre gauche parmi les arbres binaires avec k noeuds et le sous-arbre droit parmi les arbres binaires avec $n-k$ noeuds. On a donc :

$$\begin{cases} c_0 = 1 \\ c_{n+1} = \displaystyle\sum_{k=0}^{n} c_k c_{n-k} \qquad \text{pour tout } n \geqslant 0 \end{cases}$$

Question 3 –

```
let get_cn n0 =
  let mem = Array.make (n0+1) 0 in
  mem.(0) <- 1;
  for n = 1 to n0 do
    for k = 0 to (n-1) do
      mem.(n) <- mem.(n) + mem.(k)*mem.(n-k-1);
    done;
  done;
  mem.(n0);;
```

Question 4.a – Un arbre binaire de hauteur $h \in \{-1\} \cup \mathbb{N}$ avec n noeuds vérifie $n \geqslant h+1$, c'est à dire $h \leqslant n-1$. Ainsi, il n'existe pas d'arbre binaire vérifiant $h \geqslant n$.

$$\text{Donc } M_n = n \text{ convient.}$$

Question 4.b – Un arbre binaire de hauteur $h \in \{-1\} \cup \mathbb{N}$ avec n noeuds vérifie $n \leqslant 2^{h+1} - 1$, c'est à dire $h \geqslant \log_2(n+1) - 1$. Ainsi, il n'existe pas d'arbre binaire vérifiant $h \leqslant \lceil \log_2(n+1) \rceil - 2$.

$$\text{Donc } m_n = \lceil \log_2(n+1) \rceil - 2 \text{ convient.}$$

Question 5 – Comme dans la question 2, on utilise le fait que pour construire un arbre binaire de recherche, il suffit de construire un arbre binaire sans étiquette puis de l'étiqueter. Pour $n = 0$, on a :

$$u_{0,-1} = 1 \qquad \text{et} \qquad u_{0,h} = 0 \quad \text{pour tout } h \geqslant 0$$

Soit $n \geqslant 0$ et $h \in \{-1\} \cup \mathbb{N}$. Pour construire un arbre binaire de hauteur $h+1$ avec $n+1$ noeuds, on commence par choisir un entier $k \in [\![0, n]\!]$ qui va représenter le nombre de noeuds dans le sous-arbre gauche. On a alors trois possibilités :

- Le sous-arbre gauche et le sous-arbre droit sont de hauteur h. Il y a $u_{k,h} u_{n-k,h}$ possibilités.
- Le sous-arbre gauche est de hauteur h et le sous-arbre droit est de hauteur strictement inférieure à h. Il y a $\displaystyle\sum_{\ell=-1}^{h-1} u_{k,h} u_{n-k,\ell}$ possibilités.
- Le sous-arbre droit est de hauteur h et le sous-arbre gauche est de hauteur strictement inférieure à h. Il y a $\displaystyle\sum_{\ell=-1}^{h-1} u_{k,\ell} u_{n-k,h}$ possibilités.

Finalement, on a :

$$\begin{aligned} u_{n+1,h+1} &= \left[\sum_{k=0}^{n}\sum_{\ell=-1}^{h-1} u_{k,h}u_{n-k,\ell} + u_{k,\ell}u_{n-k,h}\right] + \left[\sum_{k=0}^{n} u_{k,h}u_{n-k,h}\right] \\ &= 2\left[\sum_{k=0}^{n}\sum_{\ell=-1}^{h-1} u_{k,h}u_{n-k,\ell}\right] + \left[\sum_{k=0}^{n} u_{k,h}u_{n-k,h}\right] \end{aligned}$$

En conclusion :

$$\begin{cases} u_{0,-1} = 1 \\ u_{0,h} = 0 \qquad \text{pour tout } h \geqslant 0 \\ u_{n+1,h+1} = 2\left[\sum_{k=0}^{n}\sum_{\ell=-1}^{h-1} u_{k,h}u_{n-k,\ell}\right] + \left[\sum_{k=0}^{n} u_{k,h}u_{n-k,h}\right] \qquad \text{pour tout } n \geqslant 0 \ h \geqslant -1 \end{cases}$$

Question 6 –

```
(* Attention: comme 'h' commence à -1, les indices de le matrice 'mem' sont
 * décalés. *)
let get_all_u n0 h0 =
  let mem = Array.make_matrix (n0+1) (h0+2) 0 in
  mem.(0).(0) <- 1;
  for h = -1 to h0-1 do
    for n = 0 to n0-1 do
      for k = 0 to n do
        mem.(n+1).(h+2) <- mem.(n+1).(h+2) + mem.(k).(h+1)*mem.(n-k).(h+1);
        for ell = -1 to h-1 do
          mem.(n+1).(h+2) <- mem.(n+1).(h+2) + 2*mem.(k).(h+1)*mem.(n-k).(ell+1);
        done;
      done;
    done;
  done;
  mem;;

let get_u n0 h0 =
  (get_all_u n0 h0).(n0).(h0+1);;
```

Exercice 5. Successeur dans un ABR

Question 1.a – On sait que dans le parcours infixe d'un arbre binaire de recherche, les étiquettes sont triées. Le successeur de k est donc l'élément qui suit k dans le parcours infixe de A.

Question 1.b – Soit n le nombre de noeuds dans l'arbre. Déterminer le parcours infixe de A se fait en temps $\Theta(n)$. Trouver le successeur de k dans le parcours infixe se fait en temps $\mathcal{O}(n)$. La complexité totale est donc en $\Theta(n)$.

Question 2 – On traite chacun des sous-cas :

- Si l'arbre A est vide, alors k n'apparaît pas dans A donc son successeur n'est pas défini.
- Sinon, l'arbre A est non vide :
  - Si la racine est d'étiquette k alors :
    - Si le sous-arbre droit est vide alors k est l'étiquette maximale de A donc son successeur n'est pas défini.
    - Sinon, le sous-arbre droit D est non vide et le successeur de k est l'élément minimal de D.
  - Sinon, si la racine est d'étiquette $e > k$:
    - Si k ne se trouve pas dans le sous-arbre gauche, alors k n'appartient pas à A donc son successeur n'est pas défini.
    - Sinon, si k est l'étiquette maximale du sous-arbre gauche, alors le successeur de k est e.
    - Sinon, le successeur de k dans le sous-arbre gauche est k', alors le successeur de k dans A est k'.
  - Sinon, la racine est d'étiquette $e < k$:
    - Si k ne se trouve pas dans le sous-arbre droit, alors k n'appartient pas à A donc son successeur n'est pas défini.
    - Sinon, si k est l'étiquette maximale du sous-arbre droit, alors k est l'étiquette maximale de A donc son successeur n'est pas défini.
    - Sinon, le successeur de k dans le sous-arbre droit est k', alors le successeur de k dans A est k'.

Question 3 –

```
let rec min_abr a = match a with
  | Vide -> failwith "Pas de min"
  | N(e, Vide, _) -> e
  | N(_, g, _) -> min_abr g;;
```

```
let rec successeur a k = match a with
  | Vide -> raise Absent
  | N(e, _, Vide) when e = k -> raise Maximum
  | N(e, _, d) when e = k -> min_abr d
  | N(e, g, _) when e > k -> begin try successeur g k with Maximum -> e end
  | N(e, _, d) -> successeur d k;;
```

Question 4 –

```
let successeur_bis a0 k =
  let rec aux succ a = match a with
    | Vide -> failwith "etiquette absente"
    | N(e, _, Vide) when e = k -> succ
    | N(e, _, d) when e = k -> min_abr d
    | N(e, g, _) when e > k -> aux e g
    | N(e, _, d) -> aux succ d
  in
  let res = aux k a0 in
  if res = k then failwith "etiquette maximum" else res;;
```