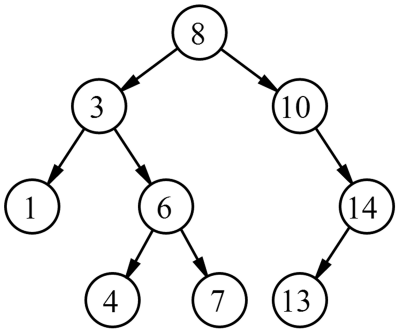
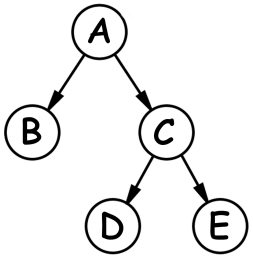
TD 8 - Arbres

1 Les incontournables

Exercice 1



1. Déclarer la structure et remplir l'arbre ci-contre dont les nœuds sont des entiers
2. Ecrire une fonction `nb_noeuds` qui retourne le nombre de nœuds (internes et externes) d'un tel arbre.
3. Ecrire une fonction `nb_internes` qui retourne le nombre de nœuds internes.
4. Ecrire une fonction `nb_feuilles` qui retourne le nombre de nœuds feuilles.



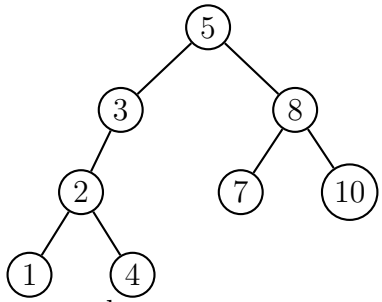
Exercice 2 On considère l'arbre binaire strict suivant.

1. Déclarer ce type d'arbre et le remplir.
2. Écrire une fonction `hauteur`.
3. Écrire une fonction `lister` qui affiche la liste de toutes les étiquettes.
4. Écrire une fonction `present` qui indique si un élément est présent dans un arbre.

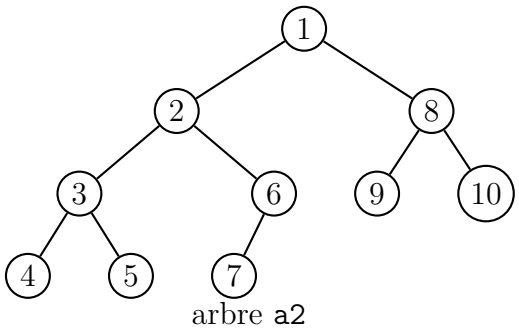
Exercice 3 Soit T un arbre binaire. On dira que T est équilibré lorsque T est vide ou que $T = (T_1, x, T_2)$, avec T_1 et T_2 équilibré, et $h(T_1) - h(T_2) \in [-1; 1]$ où $h(t)$ représente la hauteur de l'arbre t. On autorise donc un certain déséquilibre, mais celui-ci ne doit pas être trop important. Écrire une fonction qui indique si un arbre est équilibré.

Exercice 4 parcours en profondeur

1. Réaliser à la main les trois parcours présentés en cours :
   a. préfixé à gauche
   b. infixé à gauche
   c. postfixé à gauche
2. Les programmer en retournant le résultat sous forme de liste.



arbre `a1`

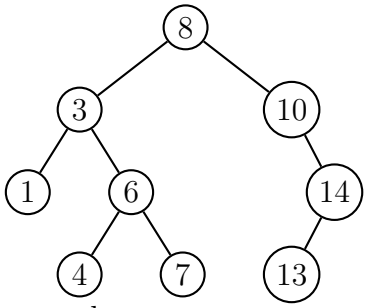


arbre `a2`

Exercice 5 Écrire une fonction de parcours **en largeur** (ou par niveau) de l'arbre ci contre. Le résultat sera affiché dans un premier temps, puis renvoyé sous forme de liste dans une seconde fonction.

Exercice 6 Arbres binaires de recherche :

1. Les arbres `a1`, `a2` et `a3` sont-ils des arbres binaires de recherche ?
2. Écrire une fonction `is_abr` qui indique si un arbre est un arbre binaire de recherche.
3. Écrire une fonction OCaml qui indique si une valeur est présente ou non dans un arbre binaire de recherche.



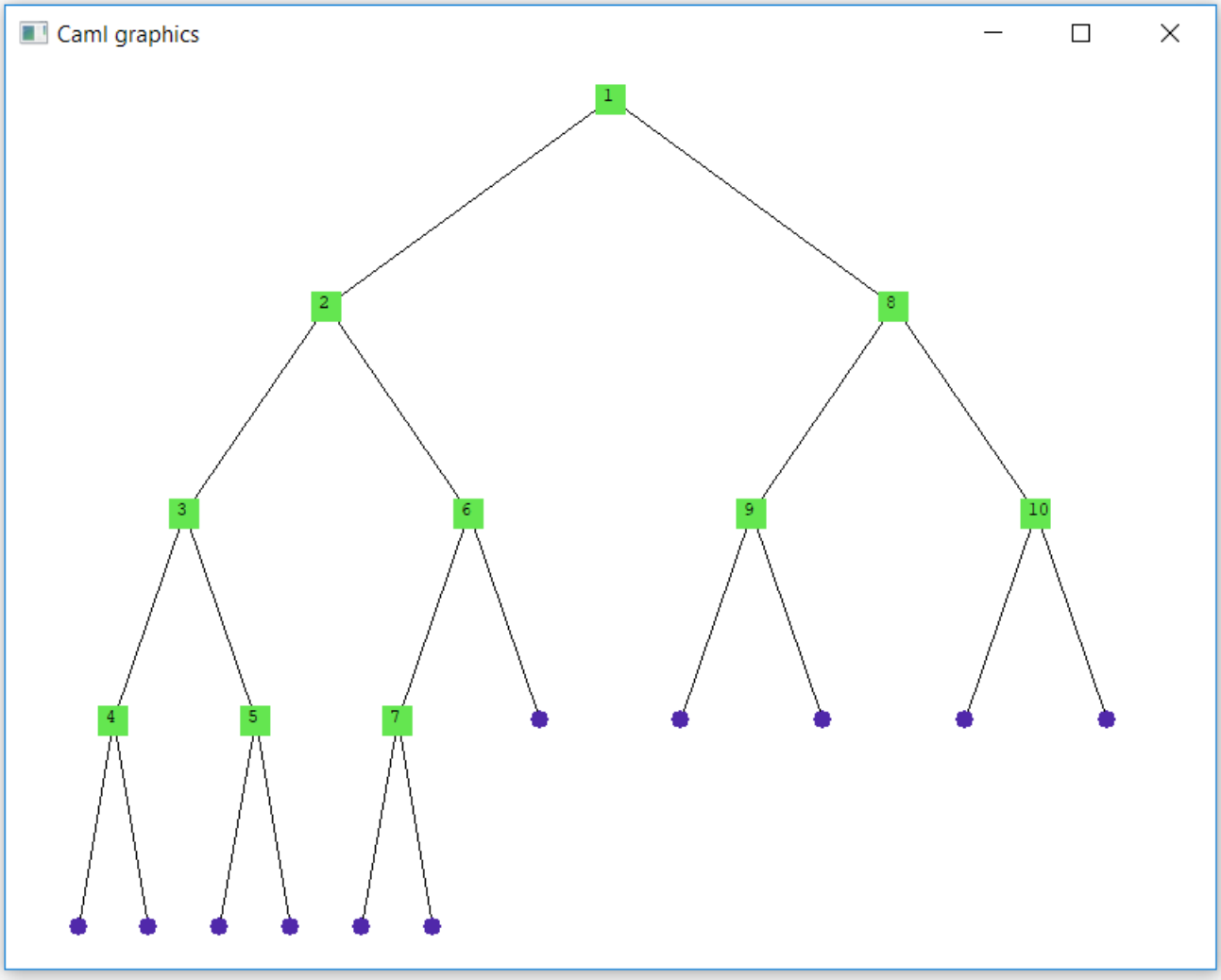
arbre `a3`

2 Pour s'entrainer

Exercice 7 Dessinons un arbre.... Écrire une fonction qui dessine un "bel" arbre avec le module graphique de OCaml. On pourra utiliser les instructions suivantes pour réaliser des graphismes :

- `#load "Graphics.cma";;` : Chargement de la bibliothèque graphique.
- `Graphics.open_graph("800x600+80+60")` : Ouvre une fenêtre de 800 pixels de large sur 600 de haut, dont le coin Nord-Ouest est placé en coordonées $(80, 60)$ sur l'écran.
- `Graphics.moveto : int -> int -> unit` : Place le point courant aux coordonnées données.
- `Graphics.lineto` : `int -> int -> unit` : trace, avec la couleur courante, un segment qui va du point courant au point donné. Le nouveau point courant est le point donné.
- `Graphics.draw_string` : `string -> unit` : Dessine une chaîne de caractères avec la police et la couleur courante : le coin inférieur gauche du premier caractère est placé à la position courante. La position courante est mise à la fin de la chaîne de caractères.

Pour embellir le rendu, voir aussi : `Graphics.set_font, Graphics.set_text_size, Graphics.text_size, Graphics.fill_rect`...



Correction 1

1. La structure et l'arbre :

```
type intArbre =
        | Nil
        | Noeud of intArbre * int * intArbre
;;

let t4 = Noeud(Nil, 4, Nil);;
let t7 = Noeud(Nil, 7, Nil);;
let t13 = Noeud(Nil, 13, Nil);;

let t1 = Noeud(Nil, 1, Nil);;
let t6 = Noeud(t4, 6, t7);;
let t14 = Noeud(t13, 14, Nil);;

let t3 = Noeud(t1, 3, t6);;
let t10 = Noeud(Nil, 10, t14);;

let t = Noeud(t3, 8, t10);;
```

2. Le nombre (total) de nœuds :

```
let rec nb_noeuds = function
   Nil -> 0
 | Noeud (ag, v, ad) ->
              1+(nb_noeuds ag)+(nb_noeuds ad)
;;
```

3. Le nombre de nœuds internes :

```
let rec nb_internes = function
   Nil -> 0
 | Noeud (Nil, v, Nil) -> 0
 | Noeud (ag, _, ad) -> 1 + (nb_internes ag)+(nb_internes ad)
;;
```

4. Le nombre de feuilles :

```
let rec nb_feuilles = function
   Nil -> 0
 | Noeud (Nil, v, Nil) -> 1
 | Noeud (ag, _, ad) -> (nb_feuilles ag)+(nb_feuilles ad)
;;
```

ou plus simplement :

```
let nb_feuilles t = (nb_noeuds t) - (nb_internes t);;
```

Correction 2 1. Déclaration du type et de l'arbre, avec le type général vu en cours :

```
type ('a,'b) arbre =
        Feuille of 'a
  | Noeud of  'b * ('a,'b) arbre * ('a,'b) arbre
;;

let a =
        let fB = Feuille "B" and fD = Feuille "D" and fE = Feuille "E" in
        let fC = Noeud ("C",fD,fE) in
        Feuille ("A",fB, fC)
;;
```

2. Pour la hauteur, on applique la définition sachant qu'ici il n'y a pas d'arbre vide, le cas d'arrêt est la feuille qui a une auteur de 0.

```
let rec hauteur = function
        | Feuille _  -> 0
        | Noeud (v, ag, ad) -> 1 + max (hauteur ag) (hauteur ad)
;;
```

3. Une solution pour lister (l'ordre n'est pas important) :

```
let rec lister = function
  | Feuille x  -> print_string (x^"-")
  | Noeud (v, ag, ad) -> print_string (v^"-") ; lister ag ; lister ad
;;
```

4. Fonction présent :

```
let rec present x = function
        | Feuille v -> v = x
        | Noeud (v, _, _) when v = x -> true
        | Noeud (v, ag, ad) -> present x ag || present x ad
;;
```

On peut aussi fusionner les deux derniers cas :

```
let rec present x = function
        | Feuille v -> v = x
        | Noeud (v, ag, ad) -> v = x  || present x ag || present x ad
;;
```

Remarque : sauriez-vous expliquer pourquoi cette fonction est de type `'a -> ('a, 'a) arbre -> bool` ?

Correction 3 On peut commencer par la fonction hauteur :

```
let rec hauteur = function
    Nil -> -1
  | Noeud (ag, _, ad) -> 1 + max (hauteur ag) (hauteur ad)
;;
```

Puis la fonction demandée :

```
let rec est_equilibre = function
      Nil -> true
    | Noeud (ag, _, ad) -> est_equilibre ag && est_equilibre ad
                  && abs( (hauteur ag)-(hauteur ad) ) < 2 ;;
```

Les arbres `a1` et `a3` ne sont pas équilibrés, l'arbre `a2` si.

Correction 4

1. Résultats des parcours :
   a. préfixe gauche : 5 - 3 - 2 - 1 - 4 - 8 - 7 - 10
   b. infixe gauche : 1 - 2 - 4 - 3 - 5 - 7 - 8 - 10
   c. postfixe gauche : 1 - 4 - 2 - 3 - 7 - 10 - 8 - 5
2. a. préfixe gauche :

```
let rec prof_prefixe_g = function
    Nil -> []
  | Noeud (ag,v,ad) -> v:: (prof_prefixe_g  ag) @ (prof_prefixe_g ad);;
```

   b. infixe gauche :

```
let rec prof_infixe_g = function
    Nil -> []
  | Noeud (ag,v,ad) -> (prof_infixe_g  ag)@ [v] @ (prof_infixe_g ad);;
```

ou encore :

```
let rec prof_infixe_g = function
    Nil -> []
  | Noeud (ag,v,ad) -> (prof_infixe_g  ag)@ (v::prof_infixe_g ad);;
```

c. postfixe gauche :

```
let rec prof_postfixe_g = function
    Nil -> []
  | Noeud (ag,v,ad) -> (prof_postfixe_g ag) @ (prof_postfixe_g ad) @ [v];;
```

Correction 5 Le parcours en largeur donne : 1 - 2 - 8 - 3 - 6 - 9 - 10 - 4 - 5 - 7. Avec un affichage :

```
let largeur_print a =
    let f = Queue.create() in
        Queue.push a f;
        while not (Queue.is_empty f) do
            match (Queue.pop f) with
                Nil -> ()
              | Noeud (ag,v,ad) ->
                        Queue.push ag f;
                        Queue.push ad f;
                        print_int v;
                        print_string "␣;"
        done;
;;
```

Avec une sortie en liste :

```
let larg_liste a =
    let rec traite f =
        if (Queue.is_empty f)
        then []
        else match (Queue.pop f) with
            Nil -> traite f
          | Noeud (ag, v, ad) ->
              Queue.push ag f;
              Queue.push ad f;
              v::(traite f)
    and file = Queue.create() in
        Queue.push a file;
        traite file
;;
```

Correction 6 1. `a1` n'est pas un ABR (le 4 est à gauche du 3). `a2` n'en et pas un non plus (par exemple 9 est à gauche de 8). Par contre `a3` en est un.

2. Première idée, on traduit la définition :

```
(* Renvoie true si a est vide ou si v > toutes les valeurs de a *)
let rec verif_gauche v = function
    Nil -> true
  | Noeud(ag , p, ad) -> v > p && verif_gauche v ag && verif_gauche v ad
;;
(* Renvoie true si a est vide ou v < toutes les valeurs de a *)
let rec verif_droite v = function
    Nil -> true
  | Noeud(ag , p, ad) -> v < p && verif_droite v ag && verif_droite v ad
;;

let rec is_abr = function
     Nil -> true
   | Noeud (ag, v, ad) -> verif_gauche v ag
                        && verif_droite v ad
                        && is_abr ag
                        && is_abr ad
;;
```

Cette solution n'est pas très satisfaisante au niveau de sa complexité, car les sous arbres sont parcourus de nombreuses fois. En remarquant que le parcours infixe d'un arbre binaire de recherche visite les nœuds dans l'ordre croissant, c'est beaucoup plus simple :

```
let is_abr a =
    let rec est_croissante = function
        | [] -> true
        | [x] -> true
        | h1::h2::t -> h1 < h2 && est_croissante (h2::t)
    in est_croissante (prof_infixe_g a)
;;
```

3. On peut créer une fonction qui détecte la présence d'un élément beaucoup plus efficace si on sait qu'il s'agit d'un ABR :

```
let rec present x = function
        | Nil -> false
        | Noeud(ag, v, ad) when x = v -> true
        | Noeud(ag, v, ad) when x < v -> present x ag
        | Noeud(ag, v, ad) -> present x ad
;;
```