- Un arbre binaire (enraciné) est défini par :

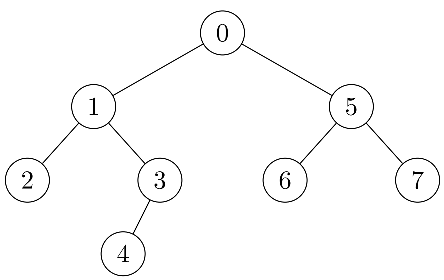
```
type 'a arbre = V | N of 'a * 'a arbre * 'a arbre
```

Si `a = N(r, g, d)`, `r` est la **racine** (ou étiquette) de `a`, `g` est son **sous-arbre gauche** et `d` est son **sous-arbre droit**.

- Un arbre binaire est **strict** si chaque noeud a exactement 0 ou 2 fils. On peut alors utiliser un type différent, si les étiquettes sont sur les feuilles seulement :

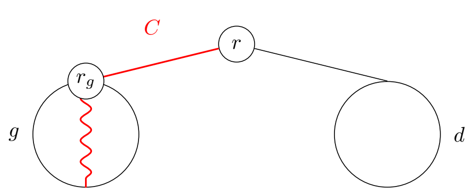
```
type 'a arb_strict =
    F of 'a | N of 'a arb_strict * 'a arb_strict
```

- Une **feuille** est un noeud sans fils.
La **profondeur** d'un noeud est la longueur du chemin (en nombre d'arêtes) depuis la racine jusqu'à ce noeud.
La **hauteur** $h(a)$ d'un arbre est la profondeur maximum d'une feuille. Par convention, on choisit souvent $h(V) = -1$.



Arbre binaire de hauteur 3, racine 0, feuilles $2, 4, 6, 7$

- Soit $a = N(r, g, d)$ un arbre binaire.
Alors $\boxed{h_a = 1 + \max(h(g), h(d))}$.
Preuve : Soit C un chemin de longueur maximum de la racine de a à une feuille.
C passe soit dans g, soit dans d (et pas dans les deux). Supposons que C passe dans g, l'autre cas étant symétrique.



Soit C_g la partie de C qui est incluse dans g.
Supposons que C_g ne soit pas un chemin de longueur maximum de la racine à une feuille dans g. Il existe alors un chemin C_g' plus long que C_g dans g. Mais alors la concaténation de l'arête de r à r_g (racine de g) et du C_g' est plus long que C, ce qui est une contradiction.
Donc C_g est un plus long chemin de r_g à une feuille de g : sa longueur est donc $h(g)$ par définition. D'où $h_a = h(g) + 1$.
Si C passe par d, on a de même : $h_a = h(d) + 1$.
Comme la hauteur est le maximum sur tous les chemins possibles : $h_a = 1 + \max(h(g), h(d))$.

```
let rec h a = match a with
    | V -> -1
    | N(_, g, d) -> 1 + max (h g) (h d)
```

Complexité : linéaire en est le nombre de noeuds $n(a)$ de a, car on `h a` effectue un appel récursif sur chaque noeud de a.

- Exercice : Le **diamètre** $diam(a)$ d'un arbre a est la longueur maximum d'un chemin entre 2 feuilles de a.
Si $a = N(r, g, d)$, montrer que $diam(a) = \max(diam(g), diam(d), h(g) + h(d) + 2)$ et en déduire une fonction pour calculer $diam(a)$.
Solution : Il y a 3 possibilités pour un chemin C dans a : soit C passe par r, soit C est entièrement dans g ou dans d.
La longueur maximum d'un chemin dans g est $diam(g)$.
La longueur maximum d'un chemin dans d est $diam(d)$.
Soit C un chemin de a, passant par r_a et de longueur $l(C)$ maximum. Montrons que $l(C) = h(g) + h(d) + 2$. Pour cela, notons C_g la partie de C dans g. Alors C_g est un chemin maximum de g depuis la racine de g (si ce n'était pas le cas, on pourrait trouver un chemin C_g' plus long et remplacer C_g par C_g' dans C pour obtenir une contradiction). Donc $l(C_g) = h(g)$, par définition de la hauteur. De même pour $l(C_d)$.
Donc $l(C) = l(C_g) + l(C_d) + 2 = h(g) + h(d) + 2$ (le $+2$ venant des 2 arêtes issues de r_a). Comme d_a correspond à la longueur du chemin maximum parmi ces 3 possibilités : $diam(a) = \max(diam(g), diam(d), h(g) + h(d) + 2)$

```
let rec h t = function (* hauteur *)
  | V -> -1
  | N(_, g, d) -> 1 + max (h g) (h d);;

let rec diam t = function
  | V -> 0
  | N(_, g, d) -> max (max (diam g) (diam d)) (h g + h d + 2)
```

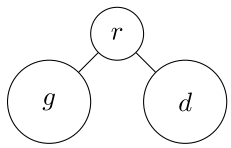
Si `t` a n noeuds, `diam t` appelle n fois `h` donc est en $O(n^2)$.
On peut obtenir une complexité linéaire en calculant diamètre et hauteur simultanément :

```
let rec diam t = function (* renvoie diamètre, hauteur *)
  | V -> -1, -1
  | N(_, g, d) ->
    let dg, hg = diam g in
    let dd, hd = diam d in
    max (max dg dd) (hg + hd + 2), 1 + max hg hd
```

- Le nombre n de noeuds et la hauteur h d'un arbre binaire a vérifie : $\boxed{h + 1 \leq n \leq 2^{h+1} - 1}$.
Preuve : Un chemin de longueur h a $h + 1$ sommets donc $n \geq h + 1$.
On montre par récurrence qu'il y a au plus 2^p sommets à profondeur p, d'où $n \leq \sum_{p=0}^{h} 2^p = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$.

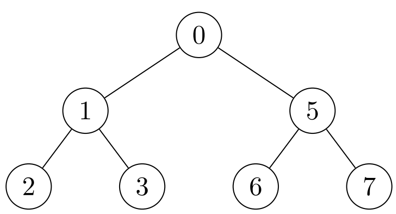
- Exercice : Soit $f(a)$ le nombre de feuilles et $n(a)$ le nombre de noeuds d'un arbre binaire strict a. Alors $n(a) = 2f(a) - 1$.

Preuve : Récurrence sur le nombre de noeuds (ou : sur la hauteur...).
Soit $P(n)$: $n = 2f(a) - 1$ pour tout arbre binaire strict a à $n \geq 1$ noeuds.
$P(1)$ est vrai car un arbre à 1 noeud possède 1 feuille.
Soit $n \geq 2$. Supposons $P(k)$ pour tout $k < n$.
Soit $a = N(r, g, d)$ un arbre à $n \geq 2$ noeuds.



Par récurrence sur g et d, on a $n(g) = 2f(g) - 1$ et $n(d) = 2f(d) - 1$.
Donc $n(a) = n(g) + n(d) + 1 = 2f(g) - 1 + 2f(d) - 1 + 1 = 2(f(g) + f(d)) - 1 = 2f(a) - 1$.
Remarque : Si on note $n_i(a)$ le nombre de noeuds **internes** de a (c'est-à-dire les noeuds qui ne sont pas des feuilles), on a $n(a) = n_i(a) + f(a)$ et donc $f(a) = n_i(a) + 1$.

- Un arbre binaire est **complet** si tous les niveaux sont remplis : il y a 2^p noeuds à profondeur p.
  Le nombre de noeuds d'un arbre binaire complet de hauteur h est donc $\displaystyle\sum_{p=0}^{h} 2^p = 2^{h+1} - 1$.



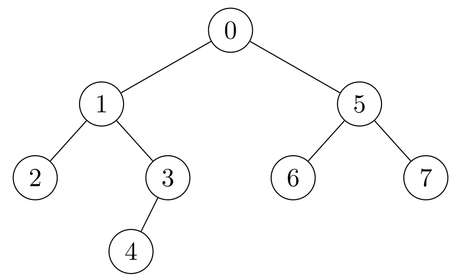
Arbre binaire complet

- a est complet $\iff$ $a = V$ où $a = N(r, g, d)$ avec g, d complets et de même hauteur.
  On en déduit une fonction récursive pour savoir si un arbre est complet :

```
let rec complet = function
    | V -> true
    | N(r, g, d) ->
        complet g && complet d && h g = h d
```

On peut obtenir une complexité linéaire avec la même astuce que pour `diam`.

- On distingue 3 types de parcours en profondeur sur un arbre binaire $N(r, g, d)$:
  - **Parcours préfixe** : visite r, puis g (appel récursif), puis d (appel récursif).
  - **Parcours infixe** : visite g (appel récursif), puis r, puis d (appel récursif).
  - **Parcours postfixe** : visite g (appel récursif), puis d (appel récursif), puis r.



Parcours préfixe $0, 1, 2, 3, 4, 5, 6, 7$
Infixe $2, 1, 4, 3, 0, 6, 5, 7$
Suffixe $2, 4, 3, 1, 6, 7, 5, 0$

Implémentation naïve en $\mathrm{O}(n^2)$ où n = nombre de noeuds :

```
let rec prefix = function
    | V -> []
    | N(r, g, d) -> r::(prefix g)@(prefix d)

let rec infixe = function
    | V -> []
    | N(r, g, d) -> (infixe g)@r::(infixe d)
```

Ou en $\mathrm{O}(n)$ avec accumulateur :

```
let rec prefix a =
    let rec aux acc = function
        | V -> acc
        | N(r, g, d) -> r::(aux (aux acc g) d)
    in aux [] a

let rec infixe a =
    let rec aux acc = function
        | V -> acc
        | N(r, g, d) -> aux (r::aux acc d) g
```