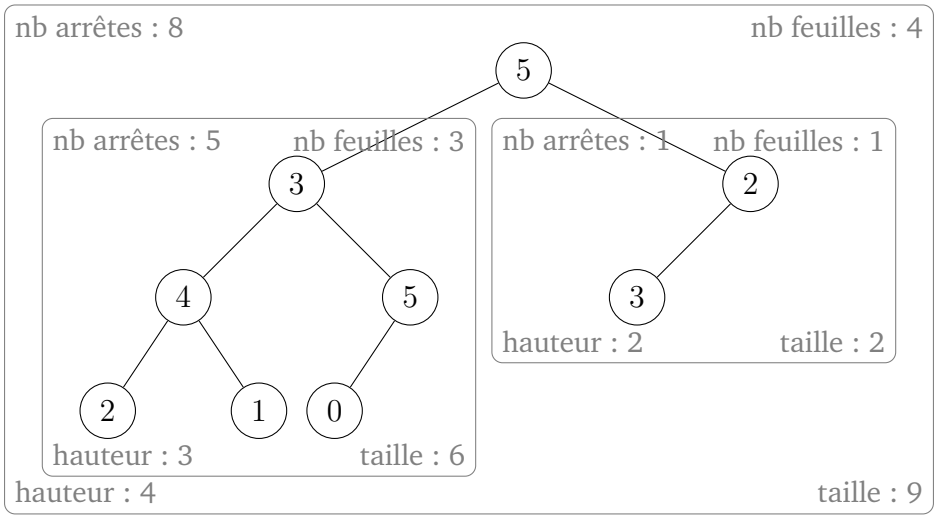
Dans tout ce TP le type `'a btree` fait référence au type OCaml ci-dessous défini.

```
type 'a btree =
| Empty
| Node of 'a * 'a btree * 'a btree
```

Exercice 1 : Arbres binaires : Taille, hauteur, nombre de feuilles, nombre d'arêtes

Le dessin ci-dessous rappelle les définitions de votre cours.



Q. 1 Définir la fonction `taille: 'a btree -> int`, s'évaluant en la taille de l'arbre passé en argument.

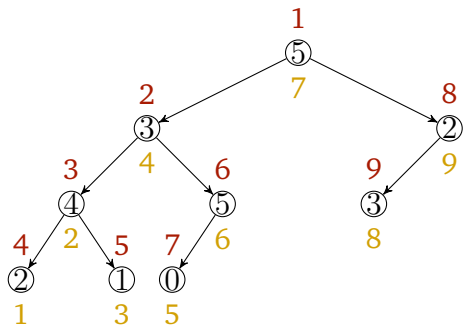
Q. 2 Définir la fonction `hauteur: 'a btree -> int`, s'évaluant en la hauteur de l'arbre passé en argument.

Q. 3 Définir la fonction `nb_feuilles: 'a btree -> int`, s'évaluant en le nombre de feuilles de l'arbre passé en argument.

Q. 4 Définir la fonction `nb_arrete: 'a btree -> int`, s'évaluant en le nombre d'arrêtes de l'arbre passé en argument.

Exercice 2 : Parcours arbres : construction de listes

L'ordre des éléments dans la liste du parcours infixe et du parcours prefixe sont indiqués dans l'arbre ci-dessous :



Q. 1 Définir une fonction `prefixe: 'a btree -> 'a list` calculant le parcours préfixe de l'arbre passé en argument.

Q. 2 Définir une fonction `infixe: 'a btree -> 'a list` calculant le parcours infixe de l'arbre passé en argument.

Q. 3 Définir une fonction `list_by_depth : 'a btree -> int -> 'a list` telle que `(list_by_depth bt n)` donne la liste de toutes les étiquettes de `bt` de profondeur `n`. Les étiquettes du sous-arbre gauche devront apparaître dans la liste avant celles du sous-arbre droit et ce récursivement.

On représente un chemin (type `path`) dans un arbre binaire par une liste de booléens (`false` pour gauche et `true` pour droit).

Q. 4 Définir une fonction `sub_tree_at : 'a btree -> path -> 'a btree` prenant en argument un arbre binaire b et un chemin p et calculant le sous-arbre de b se trouvant au bout du chemin p. Si p n'est pas un chemin admissible on lèvera une exception.

Q. 5 Définir une fonction `admissible_paths_to_nodes : 'a btree -> bool` calculant l'ensemble des chemins admissibles vers des noeuds d'un arbre binaire.

Exercice 3 : Parcours de listes : construction d'arbres

On souhaite définir des fonctions d'insertion d'un élément dans un arbre.

Q. 1 Définir une fonction `insert_left: 'a btree -> 'a -> 'a btree` d'insertion à gauche, prenant en arguments un arbre t et une étiquette e et telle que : si l'arbre passé en argument est vide alors la fonction s'évalue en une feuille contenant l'étiquette e sinon la fonction s'évalue en l'arbre obtenu en remplaçant le sous-arbre gauche g de t par l'arbre obtenu par l'insertion à gauche de e dans g.

Q. 2 Définir une fonction `from_list1: 'a list -> 'a btree` produisant l'arbre obtenue par insertion à gauche successive des éléments la liste passée en argument, à partir de l'arbre vide. Vous vous efforcerez d'utiliser la fonction `List.fold_left`.

Q. 3 Expérimenter avec la fonction `from_list1` et la critiquer.

Q. 4 Définir une fonction `insert_random: 'a btree -> 'a -> 'a btree` d'insertion aléatoire, prenant en arguments un arbre t et une étiquette e et telle que : si l'arbre passé en argument est vide alors la fonction s'évalue en une feuille contenant l'étiquette e sinon la fonction choisie à pile ou face un des deux sous-arbres et insère e dans ce sous-arbre. On pourra appeler la fonction `Random.bool` de manière suivante : `(Random.bool ())` pour obtenir un booléen tiré à pile ou face.

Q. 5 Définir une fonction `from_list2: 'a list -> 'a btree` produisant l'arbre obtenue par insertion aléatoire des éléments la liste passée en argument, à partir de l'arbre vide. Vous vous efforcerez d'utiliser la fonction `List.fold_left`.

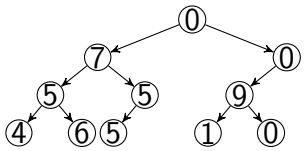
Q. 6 Expérimenter avec la fonction `from_list2` et la critiquer.

Q. 7 Définir une fonction `insert_mini: 'a btree -> 'a -> 'a btree` d'insertion minimale, prenant en arguments un arbre t et une étiquette e et telle que : si l'arbre passé en argument est vide alors la fonction s'évalue en une feuille contenant l'étiquette e sinon la fonction compare les tailles des deux sous-arbres gauche et droite et insère e dans le sous arbre de plus petite taille.

Q. 8 Définir une fonction `from_list3: 'a list -> 'a btree` produisant l'arbre obtenue par insertion aléatoire des éléments la liste passée en argument, à partir de l'arbre vide. Vous vous efforcerez d'utiliser la fonction `List.fold_left`.

Exercice 4 : Dessin d'arbres binaires

Dans cet exercice l'arbre suivant sert d'exemple :



Q. 1 Proposer une implantation d'une fonction `print_abr : int btree -> unit` prenant en argument un arbre binaire et l'affichant de la manière suivante.

```
v 0
+-v 7
| +-v 5
| | +-v 4
| | +-v 6
| +-v 5
|   +-v 5
|   +-
+-v 0
  +-v 9
  | +-v 1
  | +-v 0
  +-v 1
```

Q. 2 **Attention, question difficile** Proposer une implantation d'une fonction `print_abr : int btree -> unit` prenant en argument un arbre binaire et l'affichant de la manière suivante.

```
                +--------------0-------------------+
                |                                  |
       +---------7---------+              +---------0--+
       |                   |              |            |
  +----5----+         +----5--+      +----9----+
  |         |         |       |      |         |
+-4--+    +-6--+    +-5--+         +-1--+    +-0--+
|    |    |    |    |    |         |    |    |    |
```

Exercice 5 : Arbres binaires en C

On souhaite représenter un arbre binaire contenant des entiers en C. Afin de permettre des déplacements de bas en haut sans récursivité, les noeuds contiendront, non seulement des pointeurs vers les fils droits et gauches, mais aussi vers leur père. Lorsqu'un pointeur n'a pas lieu d'être (pointeur vers un fils lorsque celui-ci est vide, ou vers le père lorsque le noeud est la racine) on stockera le pointeur spécial `NULL` dans le champs.

Q. 1 Définir un type structuré `struct node_s` contenant 4 champs :

— une valeur entière ;
— un pointeur vers une `struct node_s` représentant le fils gauche du noeud courant ;
— un pointeur vers une `struct node_s` représentant le fils droit du noeud courant ;
— un pointeur vers une `struct node_s` représentant le père du noeud courant

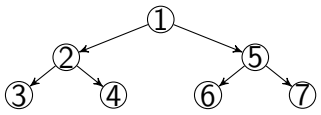
. Définir un type `node` comme étant un pointeur vers une `struct node_s`

Q. 2 Définir une fonction `node build(int v, node g, node d)` retournant un noeud correspondant à l'arbre ayant `v` comme valeur en racine, `g` comme fils gauche et `d` comme fils droit.

Q. 3 Définir une fonction `node next_prefixe(node n, int* prof)` prenant en argument un noeud que l'on supposera être à la profondeur `*prof` d'un certain arbre binaire t et retournant le noeud suivant n' dans le parcours préfixe de l'arbre t. `*prof` devra être modifié pour contenir à la fin la profondeur du noeud retourné. Si aucun noeud ne se trouve après `n` dans le parcours, on retournera le pointeur `NULL`.

Q. 4 Déduire des fonctions précédentes une fonction `print` imprimant toutes les étiquettes d'un arbre binaire de la manière suivante :

```
1(2(3)(4))(5(6)(7))
```



Q. 5 Définir une fonction `node next_infixe(node n, int* prof)` se comportant comme la fonction précédente mais pour le parcours infixe de l'arbre.