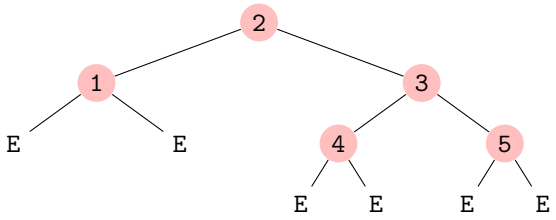
TP n°17 - Arbres binaires en OCaml.

Toutes les fonctions de ce TP seront codées dans un script OCaml `NOM_bintree.ml`.

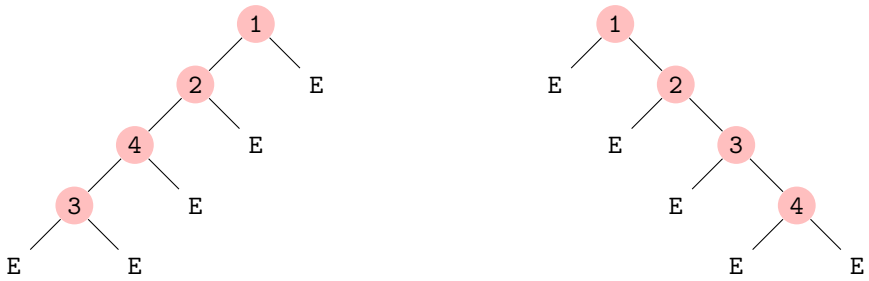
Les **arbres** sont une structure de données qui organisent les informations de manière **hiérarchique**.

Les **arbres binaires** peuvent être vus comme une généralisation de la structure de liste.
À une tête, appelée **nœud**, sont associées deux queues de listes, appelés **sous-arbres**. Ainsi, chaque sous-arbre est le descendant d'un unique **nœud parent**, excepté un nœud particulier appelé **nœud racine**, qui est tout en haut de l'arbre et n'a pas de parent.
Chaque nœud de l'arbre stocke une donnée. Le type de cette donnée est le même pour tous les nœuds de l'arbre. En pratique, il s'agit souvent d'un couple (clé, valeur) appelée **entrée**.
Un arbre particulier est ajouté à l'ensemble des arbres qu'il est possible de construire : l'**arbre vide**, que l'on créera avec un constructeur sans argument `E`.



Exercice 1 (Création d'arbres binaires en OCaml).
On s'attacher d'abord et avant tout à faire une première version naturelle des fonctions OCaml demandées, sans se préoccuper dans un premier temps de rendre ces fonctions récursives terminales. On s'attacher à bien définir le cas d'arrêt et l'expression de la récursivité pour le cas général.

1. Définir un nouveau type OCaml nommé `bintree` permettant de créer des arbres binaires dont les données stockées aux nœuds sont de type polymorphe.
2. En utilisant ce nouveau type, créer l'arbre correspondant au dessin ci-dessus.
3. Créer en OCaml les arbres représentés ci-dessous, qualifiés d'**arbres peignes**. De quelle autre structure de données de tels arbres sont-ils proches ?



4. La **taille** d'un arbre binaire t est le nombre de ses nœuds. Écrire une fonction `bintree_number_of_nodes` qui calcule le nombre total de nœuds d'un arbre binaire.
5. La **hauteur** d'un arbre binaire correspond au maximum des profondeurs de ses nœuds. Par convention, l'arbre vide (aucun nœud) a une hauteur égale à -1. Écrire une fonction `bintree_height` calculant la hauteur d'un arbre binaire. On pourra utiliser la fonction polymorphe `max : 'a -> 'a -> 'a` disponible dans la bibliothèque standard OCaml qui renvoie le plus grand de ses deux arguments.
6. Les **feuilles** d'un arbre binaire sont les nœuds dont les deux fils sont des arbres vides. Écrire une fonction `bintree_leaves` qui calcule le nombre de feuilles d'un arbre binaire.
7. Écrire une fonction `bintree_max` qui calcule le maximum des valeurs des nœuds d'un arbre binaire d'entiers.
8. Écrire une fonction `bintree_sum` qui calcule la somme des étiquettes des nœuds d'un arbre binaire d'entiers.

Le **parcours** d'un arbre binaire est un algorithme qui visite chaque nœud de l'arbre une et une seule fois et effectue un traitement sur chaque nœud. Un parcours peut avoir pour objet de synthétiser une

information à partir de toutes les étiquettes et/ou de la structure de l'arbre, de rechercher un nœud particulier ou encore de modifier l'arbre.

— Le **parcours préfixe** effectue le traitement du nœud avant le parcours des deux sous-arbres.

— Le **parcours infixe** effectue le traitement du nœud entre le parcours du sous-arbre gauche et celui du sous-arbre droit.

— Le **parcours postfixe** effectue le traitement du nœud après le parcours des deux sous-arbres.

Par exemple, si le traitement consiste à afficher l'étiquette d'un nœud, le parcours préfixe de l'arbre dessiné en début d'énoncé affiche `21345`, son parcours infixe `12435` et son parcours postfixe `14532`.

Exercice 2 (Parcours d'arbres binaires).

1. Écrire une fonction `bintree_preorder` qui affiche les étiquettes des nœuds lors d'un parcours préfixe.
2. Écrire deux fonctions `bintree_inorder` et `bintree_postorder` qui font de même lors de parcours infixe et préfixe respectivement.
3. Écrire les fonctions `bintree_preorder_list`, `bintree_inorder_list` et `bintree_postorder_list` qui effectuent toujours les parcours préfixe, infixe et postfixe mais qui renvoient la liste des nœuds rencontrés lors des différents parcours. On pourra dans un premier temps utiliser l'opérateur de concaténation `@`
4. Écrire une version 2 de ces fonctions (en rajoutant `_version2` à leur nom) qui n'utilise que l'opérateur `::` (moins coûteux que la concaténation !)
5. Les versions 2 écrites sont-elle terminales ?

Exercice 3 (Branches et poids).
Une **branche** est un chemin de la racine vers une feuille, c'est-à-dire une suite de nœuds. Le **poids** d'une branche est la somme des étiquettes qui la composent.

1. Écrire une fonction `bintree_max_sum` qui calcule le poids de la branche de poids maximal.
2. Écrire une fonction `bintree_max_sum_path` qui renvoie le chemin de la branche de poids maximal sous la forme d'une liste de nœuds.

Exercice 4 (Arbre binaire complet, arbre binaire parfait).
On appelle **arbre binaire complet** un arbre dont tous les niveaux sont remplis, sauf éventuellement le dernier.
On appelle **arbre binaire parfait** un arbre dont tous les niveaux sont remplis, y compris le dernier (donc un arbre binaire parfait est complet).

1. Dessinez un arbre binaire complet non parfait, et un arbre binaire parfait.
2. Exprimez le nombre de nœuds d'un arbre binaire parfait en fonction de sa hauteur. Vérifiez sur vos dessins précédents.
3. Écrire une fonction `bintree_create_perfect` qui crée un arbre parfait de hauteur h donnée et tel que, pour chaque nœud, sont étiquette correspond à la hauteur du sous-arbre associé à ce nœud.
4. Écrire une fonction `bintree_is_perfect` qui indique si un arbre binaire est parfait ou non.
5. Écrire une fonction `bintree_is_full` qui indique si un arbre binaire est complet ou non.