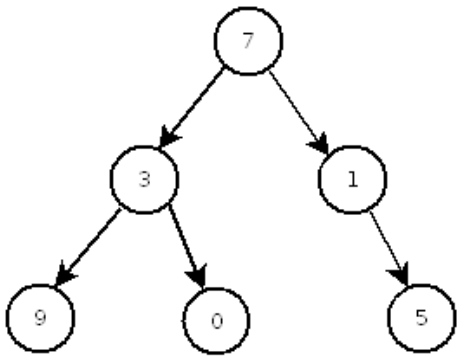
Chapitre 4 : Arbres

4.1 Définition et terminologie

Un arbre est un ensemble de nœuds, organisés de façon hiérarchique, à partir d'un nœud distingué, appelé **racine**. Une propriété intrinsèque des arbres est la récursivité, et les définitions des caractéristiques des arbres, aussi bien que les algorithmes qui manipulent des arbres s'écrivent très naturellement de manière récursive.

Exemples : système de gestion de fichiers, arbre généalogique, résultat d'un tournoi.

Nous nous intéressons plus spécifiquement aux arbres binaires. Dans un arbre binaire, chaque élément possède au plus deux éléments **fils** au niveau inférieur, habituellement appelés **gauche** et **droit**. Il s'agit de deux (sous-)arbres binaires disjoints. Du point de vue de ces éléments fils, l'élément dont ils sont issus au niveau supérieur est appelé **père**. Un nœud sans fil est également appelé **feuille**.



☺ **Exercice 4.A :** Est-ce que l'arbre dessiné ci-dessus est binaire ? Quel est le fils gauche du nœud de valeur 7 ? Quel est le fils droit du nœud de valeur 3 ? Quelles sont les feuilles de l'arbre ?

4.2. Arbres binaires

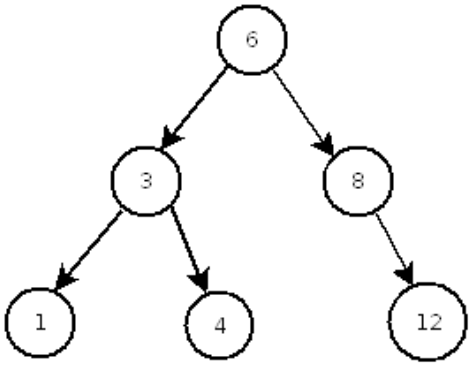
Le type BinaryTree[T] décrit les arbres binaires, dont les nœuds sont étiquetés par une valeur de type T. Les deux opérations **empty** et **makeRoot** permettent respectivement de créer un arbre vide et de créer un arbre quelconque, à partir de ses deux fils, éventuellement vides.

```
Type BinaryTree [T]

Opérations

  empty :   -> BinaryTree[T] -- créer un arbre vide
  makeRoot : BinaryTree[T] X T X BinaryTree[T] -> BinaryTree[T]
                              -- enraciner 2 ss-arbres

  isEmpty : BinaryTree[T] -> Boolean   -- arbre vide ?
  hasLeft : BinaryTree[T] -> Boolean
                              -- a un sous-arbre gauche non vide ?
  hasRight : BinaryTree[T] -> Boolean
                              -- a un sous-arbre droit non vide ?
  has : BinaryTree[T] X T -> Boolean -- existence de la valeur ?

  value : BinaryTree[T] -> T             -- valeur attachée à la racine
  left : BinaryTree[T] -> BinaryTree[T]  -- sous-arbre gauche
  right : BinaryTree[T] -> BinaryTree[T] -- sous-arbre droit

  height : BinaryTree[T] -> Integer      -- hauteur
  count : BinaryTree[T] -> Integer       -- nombre de nœuds
  prefix : BinaryTree[T] -> MyList[T]    -- noeuds dans l'ordre préfixé
  infix : BinaryTree[T] -> MyList[T]     -- noeuds dans l'ordre infixé
  postfix : BinaryTree[T] -> MyList[T]  -- noeuds dans l'ordre postfixé
```

☺ **Exercice 4.B :** compléter la spécification algébrique en indiquant les préconditions. Puis, à des fins pédagogiques (complétude suffisante non respectée), écrire les axiomes correspondant à l'application des différentes opérations sur les opérations **empty** et **makeRoot**.

4.3 Arbres binaires de recherche

Le sous-type BinarySearchTree[T] décrit les arbres binaires de recherche dont les nœuds sont étiquetés par une valeur de type T, sur lequel il existe une relation d'ordre total. C'est un cas particulier d'arbre binaire, dans lequel toutes les valeurs du sous-arbre gauche sont inférieures à la valeur de la racine, elle-même inférieure à toutes les valeurs du sous-arbre droit. Les valeurs existent en un seul exemplaire. Cette structure de données permet d'accroître l'efficacité de la fonction de recherche (has).



```
Type BinarySearchTree[T] sous-type de BinaryTree[T]

Opérations

  least : BinarySearchTree[T] -> T
                              -- plus petite valeur
  greatest : BinarySearchTree[T] -> T
                              -- plus grande valeur
  predecessor : BinarySearchTree[T] -> T
                              -- valeur précédente de la racine
  successor : BinarySearchTree[T] -> T
                              -- valeur suivante de la racine

  add : BinarySearchTree[T] X T -> BinarySearchTree[T]
                              -- ajouter une valeur

  deleteGreatest : BinarySearchTree[T] -> BinarySearchTree[T]
                              -- supprimer la plus grande valeur
  deleteLeast : BinarySearchTree[T] -> BinarySearchTree[T]
                              -- supprimer la plus petite valeur
  delete : BinarySearchTree[T] X T -> BinarySearchTree[T]
                              -- supprimer une valeur
```

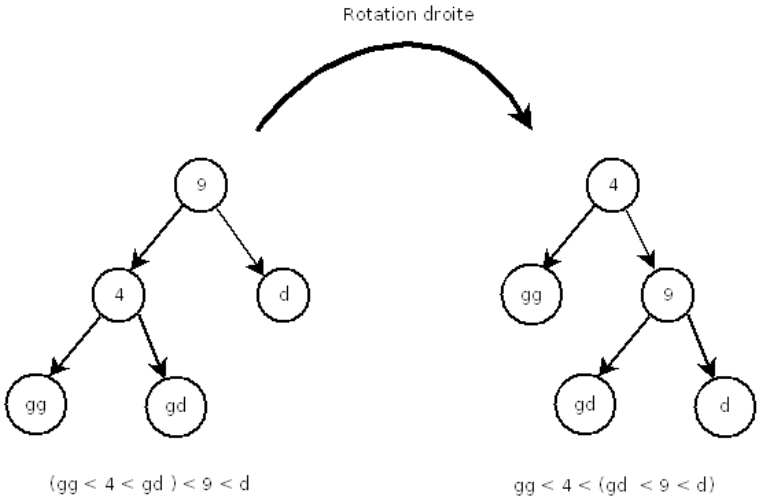
☺ **Exercice 4.C :** compléter la spécification algébrique de BinarySearchTree[T] en indiquant les préconditions. Ecrire les axiomes correspondant à l'application des opérations **least**, **greatest**, **predecessor**, **successor**, **add, deleteGreatest, deleteLeast**, et **delete** sur les opérations internes **empty** et **makeRoot**.

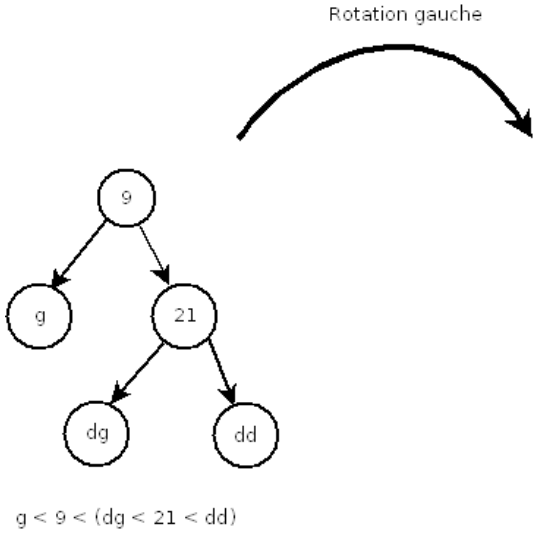
4.3.1 Insertion à la racine

Dans la définition de l'opération **add**, chaque nouvelle valeur insérée se place quelque part au bas de l'arbre. Il est possible d'envisager une autre solution qui consiste à insérer la nouvelle valeur à la racine. Dans certaines circonstances, cela peut permettre d'améliorer l'efficacité de la recherche, comme par exemple lorsque les dernières valeurs ajoutées sont aussi celles recherchées.

En pratique, on réalise l'adjonction à une feuille de l'arbre et on applique une succession de rotations, pour faire remonter la valeur à la racine. Qu'est-ce qu'une rotation ? Comme l'indiquent les schémas ci-dessous, une rotation conserve la propriété de l'arbre de recherche en permutant seulement les fils, de façon différente selon que la rotation s'effectue vers la droite ou vers la gauche.

☺ **Exercice 4.D :** compléter le schéma correspondant à une rotation gauche.





Pour ajouter une valeur à la racine, il suffit donc de l'ajouter à la racine du fils gauche ou droit et ensuite de la faire remonter par une rotation (droite ou gauche). Il reste à écrire la spécification des opérations **leftRotate**, **rightRotate** et **insertRoot**.

```
Type BinarySearchTree [T] sous-type de BinaryTree [T]

  Opérations (supplémentaires)

    leftRotate : BinarySearchTree[T] -> BinarySearchTree[T]
                                              -- rotation à gauche
    rightRotate : BinarySearchTree[T] -> BinarySearchTree[T]
                                              -- rotation à droite
    insertRoot : BinarySearchTree[T] X T -> BinarySearchTree[T]
                                              -- adjonction à la racine
```

☺ **Exercice 4.E :** à des fins pédagogiques (complétude suffisante non respectée), écrire les axiomes correspondant à l'application des opérations **leftRotate**, **rightRotate** et **insertRoot** sur les opérations internes **empty** et **makeRoot**.

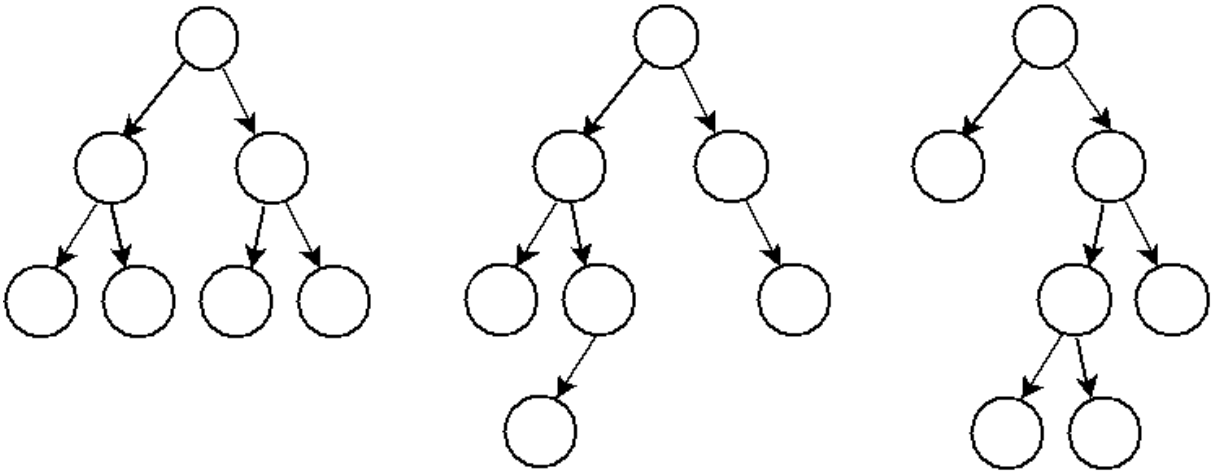
4.4 Arbres AVL

Les arbres AVL sont des arbres binaires de recherche équilibrés : les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un. La recherche, l'insertion et la suppression sont toutes en O(log(n)) dans le pire des cas. L'insertion et la suppression nécessitent d'effectuer des rotations.

La dénomination "arbre AVL" provient des noms de ses deux inventeurs, respectivement G.M. Adelson-Velsky et E.M. Landis, qui l'ont publié en 1962 sous le titre *An algorithm for the organization of information*.

Le facteur d'équilibrage d'un nœud (encore appelé balance) est la différence entre la hauteur de son sous-arbre gauche et celle de son sous-arbre droit. Un nœud dont le facteur d'équilibrage est de 1, 0, ou -1 est considéré comme équilibré. Un nœud avec tout autre facteur est considéré comme déséquilibré et requiert un rééquilibrage.

> ☺ **Exercice 4.F :** calculer les facteurs d'équilibrage, et déterminer si les trois arbres suivants sont équilibrés (c'est-à-dire si tous les nœuds de l'arbre sont équilibrés d'après la définition que nous venons de donner). Les valeurs des nœuds ne sont pas importantes ici.



> ☺ **Exercice 4.G** : construire un arbre AVL associé à la suite d'éléments 12, 3, 2, 5, 4, 7, 9, 11, 14, 10, en procédant à des adjonctions aux feuilles et en rééquilibrant l'arbre après chaque adjonction qui l'a déséquilibré.

On notera que le déséquilibre maximal après une adjonction est de +2 ou -2 et que l'adjonction d'un élément **x** ne peut entraîner de déséquilibre que sur les nœuds du chemin de la racine à **x**.
Par ailleurs, lorsqu'il y a déséquilibre après l'adjonction d'une valeur **x**, il suffit de rééquilibrer l'arbre à partir d'un seul nœud (dans le chemin de la racine à **x**, ce nœud est le dernier nœud pour lequel le déséquilibre est de -2 ou +2).

4.4.1 Principe du rééquilibrage

A partir de l'exemple précédent, nous pouvons déduire les principes de rééquilibrage pour les arbres AVL.

Soit un arbre AVL noté **α** et obtenu à partir de la racine **ρ**, le sous-arbre gauche **γ** et le sous-arbre droit **δ**.

Si l'adjonction d'une valeur **x** a lieu sur une feuille de **γ,** et qu'elle fait augmenter de 1 la hauteur de **γ**, et que **γ** reste un AVL (donc avant l'adjonction, le déséquilibre de **γ** valait 0) :

1. Si le déséquilibre de l'arbre **α** valait 0 avant l'adjonction, il vaut 1 après ; **α** reste un AVL et sa hauteur a augmenté de 1.

2. Si le déséquilibre de **α** valait -1 avant l'adjonction, il vaut 0 après ; **α** reste un AVL et sa hauteur n'est pas modifiée.

3. Si le déséquilibre de **α** valait +1 avant l'adjonction, il vaut +2 après : **α** n'est plus équilibré, il faut donc le restructurer en AVL. Dans cette situation, il n'y a que deux cas possibles selon que l'adjonction a lieu dans le sous-arbre gauche ou droit de **γ** :

    a. Si l'adjonction a lieu dans le sous-arbre gauche de **γ**, le déséquilibre de **γ** passe de 0 à 1, et on rééquilibre **γ** par une **rotation droite**.

    b. Si l'adjonction a lieu dans le sous-arbre droit de **γ**, le déséquilibre de **γ** passe de 0 à -1, et on rééquilibre par une **rotation double gauche-droite**.

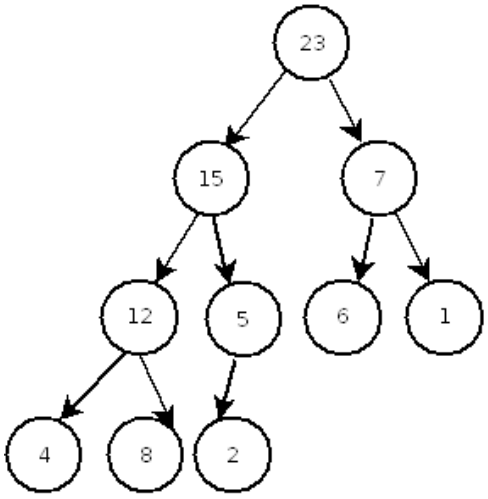
Le raisonnement est symétrique si l'adjonction de **x** a lieur sur une feuille de **δ**.

> ☺ **Exercice 4.H** : compléter la spécification algébrique de AVL[T], sous-type de BinarySearchTree[T]. Ecrire les axiomes qui correspondent à l'application des opérations **balance** et **equilibrate** sur les opérations **empty** et **makeRoot**. Vous utiliserez les opérations **rightRotate** et **leftRotate** définies précédemment pour réaliser les rotations simples et doubles.

```
Type AVL[T] sous-type de BinarySearchTree[T]

  Opérations

    balance : AVL[T] -> Integer
              -- différence de hauteurs entre les ss-arbres gauche et droit

    equilibrate : AVL[T] -> AVL[T]
              -- équilibrage de l'arbre AVL
```

4.5 Tas

On appelle **tas** un arbre binaire parfait, c'est-à-dire où tous les niveaux sont remplis sauf éventuellement le dernier, celui-ci étant rempli de la gauche vers la droite, et dans les sommets duquel figurent des éléments d'un ensemble totalement ordonné. L'élément stocké en un nœud quelconque étant plus grand que les éléments stockés dans ses deux nœuds fils s'il a deux fils, dans son fils s'il a un seul fils. Il n'y a pas d'autre relation d'ordre : un « neveu » peut être plus grand que son « oncle ».



☺ **Exercice 4.I** : Il existe une numérotation canonique des nœuds d'un tas. Si on numérote les nœuds ligne par ligne et de gauche à droite, en donnant le numéro 0 à la racine. En vous aidant d'un schéma, déterminez les numéros des fils du nœud numéro i (s'ils existent) ? Même question pour le numéro du père d'un nœud i ?

☺ **Exercice 4.J** : Si on dispose d'un tableau tab de taille m pour implanter un tas ayant au plus m nœuds, on peut stocker l'élément contenu dans le nœud i dans la case d'indice i. Ecrire une méthode **add (int v)** permettant d'insérer un nouveau nœud de valeur v (nous nous sommes restreints ici au cas d'un entier) dans la structure de tas. Ecrire également une méthode **deleteGreatest ()** qui supprime la valeur maximum du tas.

☺ **Exercice 4.K** : le tri par tas (**heapsort**) permet de trier un ensemble d'éléments quelconques E en s'appuyant sur une structure de tas. Il commence par construire un tas en ajoutant successivement les éléments au tas vide tab[0], tab[1] … Il répète ensuite les opérations suivantes : prendre le maximum, le retirer du tas, le mettre à droite du tas (dans le tableau tab). A l'aide d'un schéma, trier l'ensemble E = {9, 2, 11, 7, 4, 14, 3, 16, 8, 10, 15} en appliquant cette technique de tri, puis écrire l'algorithme correspondant.

La complexité du tri par tas est en $O(n\log(n))$: il n'y a pas de pire des cas en $O(n^2)$ comme avec **quicksort**. On démontre qu'aucun algorithme de tri par comparaison ne peut avoir de complexité asymptotiquement meilleure. Il est en place, c'est-à-dire qu'il ne nécessite pas l'allocation d'une zone mémoire supplémentaire en plus de celle contenant les données d'entrée, mais est relativement lent en moyenne comparé à quicksort.

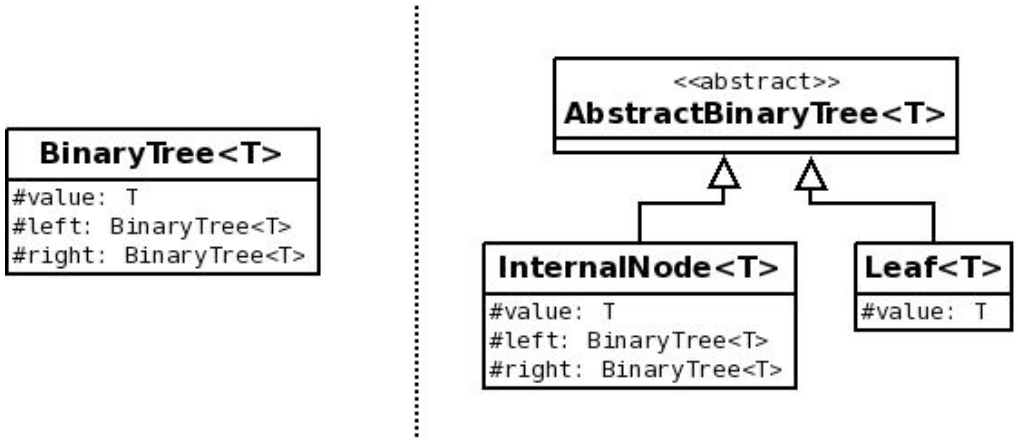
4.6 Implantations des arbres binaires

On peut envisager différentes implantations d'un arbre binaire :

- Pour implanter un arbre binaire complet (chaque nœud a exactement deux fils, sauf éventuellement le dernier en bas à droite), on peut utiliser un tableau (cf. tas). Cette solution est très restrictive, mais permet d'accélérer l'accès aux fils qui est toujours calculé ; la place mémoire requise est réduite au minimum ; les adjonctions sont compliquées à gérer si on veut pouvoir conserver la structure de tas.

- Pour représenter un arbre binaire quelconque, on peut utiliser une classe **BinaryTree** avec trois attributs : la valeur de la racine, le fils gauche et le fils droit. Il faut faire attention à la représentation de l'arbre vide : le plus simple à programmer est de le représenter par un objet avec les trois attributs null ; en contrepartie, on utilise beaucoup d'espace mémoire inutilement.

- Nous pouvons aussi chercher à tirer profit de la discrimination automatique opérée par la liaison dynamique lorsque la réalisation de certaines opérations diffère selon le type de nœud. Dans le cas général, on construit une interface **AbstractBinaryTree** et deux implantations **InternalNode** et **Leaf**. L'inconvénient majeur est que les opérations de mise à jour (adjonction et suppression) sont forcément implantées par des opérations qui retournent un nouvel arbre en résultat, puisque le type d'un arbre (vide ou non vide) peut changer.



☺ **Exercice 4.L** : traduire le type abstrait BinaryTree[T] sous forme de classes pour chacune des deux dernières approches décrites ci-dessus.

☺ **Exercice 4.M** : Même question pour le type abstrait BinarySearchTree[T].