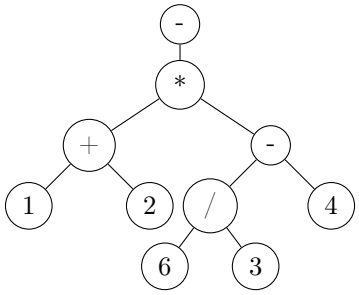
Exercice 1 : Polynôme. Dans cet exercice nous allons considérer des expressions arithmétiques.

1. Considérez l'arbre suivant :



   Son écriture infixe, parenthésée, donne $(-((1+2)*((6/3)-4)))$.
   Copiez le code du TP9 dans un nouveau dossier pour commencer cet exercice sans perdre votre dernier TP. (Vous n'utiliserez pas les questions 7 et 8 du TP9.)
2. Modifiez la variable `etiquette` pour qu'elle soit de type `String`.
3. Créez une méthode `private boolean estNombre()` qui renvoie `true` si et seulement si `etiquette` est un nombre. C'est à dire qu'elle contient au moins un chiffre et : le premier symbole est `+`, `-`, un chiffre entre `0` et `9`, ou un point `.`, et tous les autres symboles sont un chiffre entre `0` et `9` ou un point (et il y a au plus un point).
4. Créez une méthode `public boolean verif()` qui renvoie `true` si et seulement si l'arbre correspond à une expression arithmétique correcte. C'est à dire que si l'étiquette est `+`, `/`, ou `*` alors l'arbre a deux enfants qui correspondent à une expression arithmétique correcte, si c'est `-` alors il y a au moins un enfant droit correct, et sinon l'étiquette est un nombre, et les deux enfants sont `null`.
5. Testez la méthode `afficheInfixe()` sur votre arbre. Voyez-vous un souci ? Corrigez-le via une méthode `public void afficheExpression()`.
6. Créez une méthode `public double eval()` qui évalue l'expression. Renvoyez 0 si l'expression n'est pas correcte. Après avoir vérifié que `etiquette` est bien un nombre, vous pouvez utiliser par exemple `Double.parseDouble(etiquette)` pour récupérer sa valeur.

Les questions suivantes de cet exercice sont facultatives.

7. Maintenant l'étiquette d'une feuille peut aussi être la lettre `x`, nous travaillons donc sur des expressions avec une variable, donc des polynômes. Modifiez donc la méthode `public bool verif()` pour tenir compte de cet état.
8. Créez une méthode `public int evaluer(double n)` qui évalue le polynôme en `n`. On pourra s'inspirer de la méthode `eval()` écrite auparavant.
9. Créez une méthode `public Noeud simplifier()` qui va renvoyer un arbre mathématiquement équivalent, mais aussi simplifié que possible. Par exemple, vous pouvez remplacer `2+3` par `5`.
10. Créez une méthode `public int deriver()` qui rend un arbre correspondant à la fonction dérivée de `this`.

    Pour rappel :

Fonction f	Dérivée f'
constante	0
`x`	1
$-c$	$-c'$
$c+d$	$c'+d'$
$c\times d$	$c'\times d + c\times d'$
$\frac{f}{g}$	$\frac{f'\times g - f\times g'}{g^2}$

11. Créez un méthode `public void (String expression)` qui prend une `String`, et renvoie `true` si et seulement si elle correspond à une expression infixe.

12. Créez un constructeur `public Noeud (String expression)` qui prend une `String`, qui vérifie si elle correspond à une expression infixe parenthésée correcte et qui construit l'arbre qui correspond à cette expression. Sinon renvoyez `null`.
    Vous aurez probablement besoin d'utiliser une fonction auxiliaire.

Exercice 2 : Suite du TP9. Dans cet exercice, il faut commencer par terminer le TP9 puis on va le compléter. Après quelques questions sur les arbres binaires de recherche, nous allons montrer un exemple d'utilisation d'arbre.

1. Codez (si ce n'était pas déjà fait) une méthode `public int profondeur()` qui renvoie la profondeur de l'arbre.
2. Codez une méthode `public void afficheHorizontal()` qui affiche l'arbre, la racine à gauche, l'arbre droite au dessus de la racine, l'arbre gauche en dessous, avec un noeud par ligne. Vous pourrez utiliser une fonction auxiliaire (par exemple pour calculer le nombre d'espaces à insérer). Pour l'arbre du TP9, on obtient :

```
    1
      4
  7
    0
3
  5
        8
      6
    9
      2
```

3. Codez une méthode `public void afficheArbre()` qui affiche l'arbre avec la racine en haut et un noeud par colonne, comme sur la figure ci-dessous. On pourra utiliser un parcours en largeur, ainsi que la méthode `nbDeNoeuds()` du TP9 pour savoir combien d'espaces laisser à gauche du noeud qu'on souhaite afficher.

```
          3
        5     7
  9           0     1
2   6             4
      8
```

4. Codez une méthode `public Noeud retireRacine()` qui renvoie un arbre “identique" à `this`, mais sans le noeud racine. Cela se fait de la manière suivante : si le fils droit est nul, il suffit de renvoyer le fils gauche. Sinon, soit `G` le noeud le plus à gauche du fils droit [1]. Alors on renvoie un arbre dont l'étiquette de la racine vaut `G.etiquette`, dont l'arbre gauche est `this.gauche` et dont l'arbre droit est `this.droit` mais où le noeud `G` est remplacé par le sous-arbre `G.droit`.
5. Codez une méthode `public Noeud retire(int r)` qui renvoie un arbre identique à `this`, mais sans le ou les noeuds dont l'étiquette est `r` (et ne fait rien s'il n'y a pas de tel noeud).
6. Codez une méthode `public ArrayList<Integer> contenu()` qui rend une `ArrayList` qui contient le contenu de l'arbre dans l'ordre infixe. Vous pouvez utiliser une fonction intermédiaire pour être efficace. Que pouvez vous dire du résultat de cette méthode si l'arbre est un Arbre Binaire de Recherche [2] ?
7. Testez sur plusieurs exemples pour montrer que vos méthodes sont robustes.

1. En partant du fils droit de `this` et en descendant suivant les fils gauches, c'est le premier noeud pour lequel `gauche` est `null`. Dans l'arbre donné en exemple, il s'agit du sous-arbre enraciné en `0` (qui, ici, est une feuille).
2. Un ABR est un arbre où l'étiquette de n'importe quel noeud est supérieure aux étiquettes dans son sous-arbre gauche, et inférieure aux étiquettes dans son sous-arbre droit.