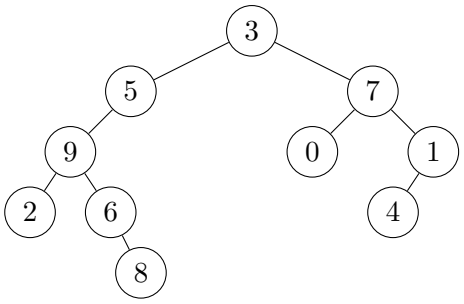
Pendant tout ce tp, nous allons prendre comme exemple l'arbre suivant.



Commençons par définir différentes façons de parcourir un arbre binaire :

1. **Parcours infixe** : on parcourt le sous-arbre de gauche dans l'ordre infixe, on visite le sommet puis on parcourt le sous-arbre de droite dans l'ordre infixe.
   Pour l'arbre précédent ce parcours est : 2 9 6 8 5 3 0 7 4 1.
2. **Parcours préfixe** : on visite le sommet, on parcourt le sous-arbre de gauche dans l'ordre préfixe puis on parcourt le sous-arbre de droite dans l'ordre préfixe.
   Pour l'arbre précédent ce parcours est : 3 5 9 2 6 8 7 0 1 4.
3. **Parcours suffixe** : on parcourt le sous-arbre de gauche dans l'ordre suffixe, on parcourt le sous-arbre de droite dans l'ordre suffixe puis on visite le sommet.
   Pour l'arbre précédent ce parcours est : 2 8 6 9 5 0 4 1 7 3.

On considère ainsi la classe `Noeud` suivante :

```
public class Noeud {
    private int etiquette;
    private Noeud gauche;
    private Noeud droit;

    public Noeud(int etiquette, Noeud g, Noeud d) {
        this.etiquette = etiquette;
        this.gauche = g;
        this.droit = d;
    }

    public Noeud(int etiquette) {
        this(etiquette,null,null);
    }
}
```

et la classe `Arbre` suivante :

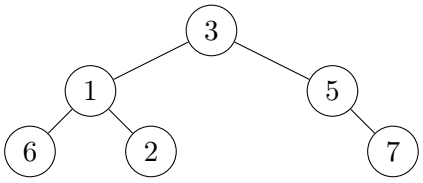
```
public class Arbre {
    private Noeud sommet;

    public Arbre(Noeud sommet) {
      this.sommet = sommet;
    }
```

```
    public Arbre() {
      this(null);
    }
}
```

1. Définir des méthodes `public void afficheInfixe()` dans les classes `Arbre` et `Noeud` qui permettent d'afficher un arbre dans l'ordre infixe.
2. Tester dans une classe `Main` la méthode `afficheInfixe()` sur l'exemple donné en introduction grâce au code suivant :

```
public class Main{
        public static void main(String[] args){
                Noeud a = new Noeud(6,null,new Noeud(8));
                Noeud b = new Noeud(9, new Noeud(2), a);
                Noeud c = new Noeud(5, b, null);
                Noeud d = new Noeud(1, new Noeud(4), null);
                Noeud e = new Noeud(7, new Noeud(0), d);
                Noeud f = new Noeud(3, c, e);
                Arbre g = new Arbre(f);
                g.afficheInfixe();
        }
}
```

3. Définir dans les classes `Arbre` et `Noeud` des méthodes `public void affichePrefixe()` et `public void afficheSuffixe()` qui permettent d'afficher un arbre respectivement dans l'ordre préfixe et suffixe.
4. Définir une méthode `int nbDeNoeuds()` qui retourne le nombre de nœuds d'un arbre.
5. Définir une méthode `int somme()` qui retourne la somme des étiquettes d'un arbre.
6. Définir une méthode `int profondeur()` qui retourne la profondeur d'un arbre. La profondeur d'un arbre est le nombre de pas dans le plus long chemin du sommet à une feuille (i.e., la plus longue *branche*) ; par exemple, la profondeur de l'arbre-exemple est 4 : sa plus longue branche est $3-5-9-6-8$, qui contient 5 nœuds, et 4 pas.
7. Définir une méthode `boolean recherche(int e)` qui renvoie `true` si un nœud de l'arbre est étiqueté par `e`.
8. Définir un constructeur `Arbre(Arbre a)` qui crée une **copie** de l'arbre donné en argument.
9. (facultatif) Définir un nouveau constructeur `Arbre(int[] tab)` qui prend en entrée un tableau non vide de taille `n`, et construit un arbre dont les étiquettes des nœuds dans l'ordre infixe sont `tab`. Par exemple, `new Arbre(`$[6,1,2,3,7,5]$`)` est :



Indications :
— Soit `r` la moitié de `n` (arrondi à l'inférieur : `int r = n/2`).
— Le sommet de `this` a pour étiquette tab$[r]$.
— Soit `tabG =` $[$`tab`$[0],...,$`tab`$[r-1]]$. Le sous-arbre de gauche est `new Arbre(tabG)`.
— Soit `tabD =` $[$`tab`$[r+1],...,$`tab`$[n-1]]$. Alors le sous-arbre de droite est `new Arbre(tabD)`.

10. (facultatif) Tester le constructeur sur l'arbre précédent via le code :

```
int[] tab = {6,1,2,3,7,5};

Arbre h = new Arbre(tab);

h.afficheInfixe();
h.affichePrefixe();
```

Ce dernier doit afficher `6 1 2 3 7 5` et `3 1 6 2 5 7 .`