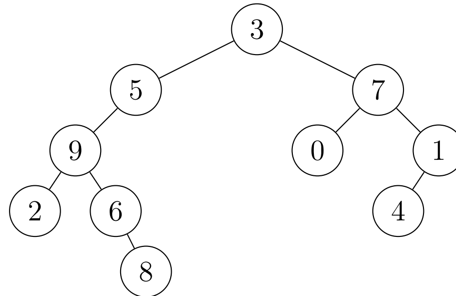
Exercice 1 Vous pouvez reprendre comme base les classes `Arbre` et `Noeud` que vous avez écrits dans le TP 9 (copiez-les dans un nouveau répertoire). Nous reprenons l'arbre donné en exemple au TP 9 :



1. Crééz une classe `Main` qui contient une méthode `main`, ainsi que la méthode `test` suivante.

```
public static Arbre test(){
        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);
        return new Arbre(f);
}
```

2. On souhaite obtenir un affichage des arbres "en penchant la tête". Dans le cas de notre exemple ce serait :

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

L'artifice qui donne un peu de relief à cet affichage consiste à ajouter des espaces avant d'afficher une étiquette. Ce nombre d'espaces est fonction de la profondeur. Ainsi la racine apparaît sur la colonne 0, et ses deux fils sur la colonne 1. On remarque que sur cette présentation le fils droit de 3 est affiché avant son fils gauche (7 est au dessus de 5).

(a) Ecrivez une méthode **espace(int n)** de la classe `Noeud` qui affiche n espaces.

(b) Ecrivez une méthode **affiche(int p)** de la classe `Noeud` qui se charge de l'affichage du sous arbre issu du noeud courant, supposé être à la profondeur p.

(c) Conclure en écrivant la méthode **public void affichePenche()** dans la classe `Arbre`.

(d) Testez, en mettant le code suivant dans la méthode **main** de la classe `Main` :

```
Arbre a = test();
a.affichePenche();
```

Exercice 2

1. Un affichage **en largeur** d'un arbre affiche les étiquettes des nœuds en traversant l'arbre "ligne par ligne", et chaque ligne de gauche à droite ; dans notre exemple : 3 5 7 9 0 1 2 6 4 8. Ecrivez une méthode `afficheLargeur()` d'affichage en largeur d'un arbre.
   Indications. Vous pourrez utiliser la classe `LinkedList<E>` du langage Java, qui implemente les listes simplement chaînées d'objets d'une classe `E`. Mettez en première ligne de la classe `Arbre` une ligne `import java.util.LinkedList;` Vous construirez une instance d'une liste de noeuds avec `LinkedList<Noeud> noeuds = new LinkedList<Noeud>();` Voici un extrait de l'API Java[1] qui décrit les trois méthodes de la classe `LinkedList` dont vous aurez besoin.

```
// Adds the specified element as the tail
// (last element) of this list.
boolean offer(E e);
// Retrieves and removes the head
// (first element) of this list.
E poll();
// Returns true if this collection contains no elements.
boolean isEmpty();
```

2. On veut écrire une méthode `afficheTopdown()`, pour obtenir un résultat proche de celui ci. Nous allons procéder par étapes.

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

   (a) Ecrivez une méthode qui permette de calculer la profondeur d'un noeud (normalement vous l'avez déjà fait la semaine dernière).

   (b) Ecrivez une classe `Paire`, ayant un champs `Noeud` et un champs de type entier.

   (c) Commencez à écrire la méthode `afficheTopdown` : reprenez le code de l'affichage en largeur, remplacant la `LinkedList<Noeud>` par une `LinkedList<Paire>`. Vous placerez initialement dans la liste le sommet couplée à sa profondeur, et de sorte que chaque élément (noeud,valeur) sorti de la file y replace une ou deux paires (fils, valeur-1), s'il n'est pas un noeud terminal.

   (d) Ajouter ensuite une variable entière `hautCour` à votre méthode qui contient la hauteur de la ligne courante. Si, dans la boucle principale qui vide la file, la hauteur du prochain noeud retiré de la file est inférieur à `hautCour`, afficher un retour à la ligne et décrementer `hautCour`. A ce stade la méthode devrait afficher :

```
3
57
901
264
8
```

   (e) On réalise que l'espace entre 6 et 4 dans notre exemple est fonction de deux paramètres : la hauteur des noeuds (qu'on connait puisqu'elle est stockée dans la paire), mais aussi du nombre de noeuds actuellement absents à cette hauteur, des noeuds qui auraient pu être des cousins. Reprenez votre code pour introduire dans la file des paires correspondant à tous les noeuds absents, ce sont ceux qu'on aurait trouvé dans l'arbre complet. Assurez-vous que votre boucle se termine toujours !

   (f) Finalement ajoutez suffisamment d'espaces. Pouvez vous justifier ce nombre ?

1. `https://docs.oracle.com/en/java/javase/11/docs/api/index.html`

Exercice 3 - Si vous avez du temps ...

1. Codez une méthode `nouveauLeader()` qui va remplacer/supprimer la racine d'un arbre d'une façon un peu originale : si son fils droit est nul, c'est son fils gauche qui devient racine.

   Sinon, de proche en proche en partant de la racine, et en progressant toujours vers la droite, on cherche le premier nœud qui n'a pas de fils droit. On remplace alors la valeur portée par la racine par celle portée par le noeud trouvé. Cette valeur identifiera un nouveau leader. Il reste à effacer la trace de sa présence dans son ancien nœud, en reliant son éventuel ancien enfant à son ancien père.

   Sur l'arbre précédent, l'élection d'un nouveau leader fera reporter la valeur 1 à la racine, à la place de 3, et le noeud portant initialement 1 sera court circuité : son père d'étiquette 7 prenant comme fils le fils gauche de 1 (c. à d. 4)

   Une nouvelle élection reportera la valeur 4 à la racine, et supprimera l'ancien noeud de 4 de la même façon, en donnant pour fils à 7 le fils gauche de 4 (ici **null**).

   Une troisième exécution de `nouveauLeader()` verra la valeur 7 remplacer celle couramment sur le noeud racine, et la racine adoptera pour fils droit le noeud étiqueté 0, etc ...

2. Codez une méthode `retire(int r)` qui localement procède à l'élection d'un nouveau leader sur tous les nœuds d'étiquette r.