Exercice 1 Vous pouvez reprendre comme base votre travail du TP précédent (copiez le dans un nouveau répertoire)

1. Compléter la méthode suivante pour qu'elle retourne le noeud racine de l'arbre donné en exemple au TP 9.

```
public static Noeud test(){
 Noeud[] n=new Noeud[10];
 n[4]=new Noeud(4);
 n[1]=new Noeud(1,n[4], null);
 ...
 return ...;
}
```

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'espace 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) ;** 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 affichePenché()** dans la classe Arbre

Exercice 2

1. On rappelle que c'est en utilisant une file qu'on implémente un affichage des noeuds d'un arbre en suivant un parcours en largeur.
   Pour simplifier vous pourrez utiliser la classe **LinkedList<Noeud>** de java et les méthodes qui permettent de la considérer comme une file. Vous construirez une instance d'une file de noeuds avec **LinkedList<Noeud> maFile=new LinkedList() ;**

```
// 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();
// test du vide
boolean isEmpty();
```

Ecrivez une méthode d'affichage en largeur de l'arbre. Dans notre exemple elle devrait produire : 3579012648

2. On veut modifier l'affichage en largeur en ajoutant des espaces, pour obtenir un résultat proche de celui ci.

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

Nous allons procéder par étapes.

(a) Ecrivez une méthode qui permette de calculer la hauteur d'un arbre

(b) Ecrivez une classe **Paire**, ayant un champs Noeud et un champs entier.

(c) Reprenez le code de l'affichage en largeur, en placant initialement dans la file la racine couplée à sa hauteur, et de sorte que chaque élément (noeud,valeur) sorti de la file y replace une paire (fils, valeur-1).

(d) Ajouter ensuite une variable entière à votre méthode pour permettre à la boucle principale qui vide la file de détecter les changements de valeurs entières sur les paires extraites. Afficher un retour à la ligne dans ce cas. 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 profondeur (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.

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

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.