Corrigé devoir maison TO2

Exercice 1 1-2. La stratégie du cuisinier pour trier sa pile de crêpes est expliquée dans l'énoncé : à la $p$ième étape, il replace la $p$ième plus grosse crêpe au $p$ième étage de la pile, en faisant deux retournements. On obtient le code suivant :

```
public class PileCrepes {
   private int[] diametres; // les diamètres des crêpes dans la pile

   public PileCrepes(int[] diametres) {
      // on fait une copie du tableau diametres passé en argument
      this.diametres = new int[diametres.length];
      for (int i = 0; i < diametres.length; i++) {
         this.diametres[i] = diametres[i];
      }
   }

   public void spatule(int position) {
      // on inverse l'ordre des diamètres situés après position
      for (int i = 0; i < (diametres.length-position)/2; i++) {
         int temp = diametres[position+i];
         diametres[position+i] = diametres[diametres.length-1-i];
         diametres[diametres.length-1-i] = temp;
      }
   }

   public int positionPlusGrosse(int etage) {
      // on cherche l'indice du maximum du tableau diametres après etage
      int position = etage;
      for (int i = etage; i < diametres.length; i++) {
         if (diametres[i] > diametres[position]) position = i;
      }
      return position;
   }

   public void cuisinier() {
      // on effectue autant d'étapes que la hauteur de la pile de crêpes
      for (int i = 0; i < diametres.length; i++) {
         // à chaque étape, on a deux retournements au plus
         int j = positionPlusGrosse(i);
         if (i != j) {
            spatule(j);
            spatule(i);
         }
      }
   }
}
```

3. L'invariant qui assure que la méthode du cuisinier trie bien la pile de crêpes est le suivant :

« Après la $p$ième étape, les p plus grosses crêpes sont à leur place finale (c'est-à-dire aux p premières places, et dans l'ordre décroissant) ».

Cette propriété est vrai avant de commencer l'algorithme. Elle se transmet car le cuisinier replace la $p$ième plus grosse crêpe à sa place lors de la $p$ième étape. Enfin, à la dernière étape, toutes les crêpes sont à leur bonne place, ce qui montre que l'algorithme trie bien la pile de crêpes.

4. Le nombre de retournements nécessaires pour trier une pile de n crêpes dans le pire des cas est $2n-3$. En effet, il faut au pire deux retournements pour chaque crêpe, sauf pour l'avant dernière qui ne nécessitera qu'un seul retournement, et la dernière qui ne nécessitera aucun retournement. Le pire cas est atteint par exemple par la pile de $n = 2m$ crêpes dont les diamètres sont 1, $2m-1$, $2m-3$, ..., 5, 3, 4, 6 ..., $2m-2$, $2m$, 2, qui nécessite effectivement $4m-3$ retournements.

Exercice 2 Comme expliqué dans l'énoncé, un arbre d'entiers est la donnée d'une `racine` (de type `int`) et d'un tableau `branches` d'arbres représentants les branches de l'arbre (de type `Arbre[]`). On ne considère pas d'arbre vide : on a toujours une racine. De plus, dans le tableau de branches, toutes les branches sont des arbres ayant au moins une racine (sinon, on les supprime simplement du tableau). Pour les questions 1 à 5, on obtient donc le code suivant :

```
public class Arbre {
   public int     racine;   // etiquette de la racine
   public Arbre[] branches; // branches de l'arbre

   public Arbre(int racine, Arbre[] branches) {
      this.racine = racine;
      this.branches = branches; // on ne fait pas de copie
   }

   public void parcours() {
      // on écrit d'abord la racine, ...
      System.out.print(racine + " ");
      for (int i = 0; i < branches.length; i++) {
         // ... puis on parcours une par une les branches de l'arbre
         branches[i].parcours();
      }
   }

   public int nbNoeuds() {
      // on compte d'abord la racine...
      int accumulateur = 1;
      for (int i = 0; i < branches.length; i++) {
         // puis on ajoute le nombre de noeuds dans chaque branche
         accumulateur += branches[i].nbNoeuds();
      }
      return accumulateur;
   }

   public int nbFeuilles() {
      // si on voit une feuille, on retourne 1
```

```
      if (branches.length == 0) return 1;
      int accumulateur = 0;
      for (int i = 0; i < branches.length; i++) {
         // sinon, on compte le nombre de feuilles dans chaque branche
         accumulateur += branches[i].nbFeuilles();
      }
      return accumulateur;
   }

   public boolean egale(Arbre A) {
      // arbres egaux <=> même racine, même nombre de branches et branches égales
      if (racine != A.racine || branches.length != A.branches.length) return false;
      for (int i = 0; i < branches.length; i++) {
         if (!branches[i].egale(A.branches[i])) return false;
      }
      return true;
   }

   public boolean contient(Arbre A) {
      // this contient A <=> this == A ou l'une des feuilles de this contient A
      if (this.egale(A)) return true;
      for (int i = 0; i < this.branches.length; i++) {
         if (this.branches[i].contient(A)) return true;
      }
      return false;
   }

   public Arbre copie() {
      // on copie le tableau
      Arbre[] copieBranches = new Arbre[branches.length];
      for (int i = 0; i < branches.length; i++) {
         // on copie une par une les branches pour avoir une copie en profondeur
         copieBranches[i] = branches[i].copie();
      }
      return new Arbre(racine, copieBranches);
   }
}
```

Pour la dernière partie, on a besoin de manipuler des arbres binaires incomplets. Autrement dit, des arbres dont tous les noeuds ont précisément deux fils, chacun des deux pouvant être `null`. On doit donc maintenant admettre les arbres `null` dans le tableau `branches`.

```
public class Arbre {
   ...
   public boolean estBinaire() {
      // un arbre est binaire <=> il a deux fils qui sont tous les deux binaires
      return branches.length == 2 &&
             (branches[0] == null || branches[0].estBinaire()) &&
             (branches[1] == null || branches[1].estBinaire());
   }
```

```
   public Arbre toBinaire() {
      /* pour construire l'arbre binaire associé à this, on part du fils
         le plus à droite de this, et on remonte jusqu'au fils le plus à gauche */
      Arbre result = null;
      for (int i = branches.length - 1; i >= 0; i--) {
         result = new Arbre(branches[i].racine,
                            new Arbre[] {branches[i].toBinaire(), result});
      }
      return result;
   }

   public Arbre fromBinaire(int racine) {
      /* On calcule d'abord le nombre de fils de l'arbre que l'on veut construire,
         c'est-à-dire la longueur du chemin à droite dans l'arbre binaire. */
      int longueur = 0;
      Arbre parcours = this;
      while (parcours != null) {
         longueur++;
         parcours = parcours.branches[1];
      }
      // Une fois qu'on connait la longueur, on peut créer le tableau et le remplir.
      Arbre[] branchesResultat = new Arbre[longueur];
      parcours = this;
      for (int i = 0; i < longueur; i++) {
         if (parcours.branches[0] == null) {
            branchesResultat[i] = new Arbre(parcours.racine, new Arbre[0]);
         }
         else {
            branchesResultat[i] = parcours.branches[0].fromBinaire(parcours.racine);
         }
         parcours = parcours.branches[1];
      }
      // Il ne reste plus qu'à rajouter la racine.
      return new Arbre(racine, branchesResultat);
   }
}
```