PRA – TP 11 à 14 – Arbres binaires et Images

Il est fortement conseillé de lire la totalité du sujet avant de se lancer dans la réalisation.

L'objectif de ce TP est d'utiliser des arbres binaires pour représenter des images binaires (en noir et blanc). On utilisera pour cela une structure de données arborescente permettant de compresser les images en regroupant les zones homogènes (tout noir ou tout blanc), et on implémentera de façon efficace diverses opérations sur ces images.

1 Introduction

On considère des images binaires de taille 256×256 pixels, où chaque pixel est soit éteint (noir, valeur 0), soit allumé (blanc, valeur 1).

L'image est représentée par un arbre binaire dont chaque nœud représente une région rectangulaire de l'image. On associe à chaque région (nœud de l'arbre) un des trois états suivants :

— 0 si tous les pixels de la région sont éteints (noir),
— 1 si tous les pixels de la région sont allumés (blanc),
— 2 sinon.

La racine de l'arbre représente l'image entière (256×256 pixels).

Pour la création des régions, on procède par découpages successifs de l'image en deux "moitiés". Le découpage se fait d'abord horizontalement (région carrée coupée en deux rectangles horizontaux : haut et bas), puis verticalement (region rectangulaire coupée en deux carrés : gauche et droite), et ainsi de suite, en alternant les découpages horizontaux et verticaux. On découpe jusqu'à obtention de régions (carrées ou rectangulaires) dont la couleur est entièrement noire ou blanche (état 0 ou 1).

L'arbre binaire représentant l'image a les propriétés suivantes :

— Chaque nœud de valeur 0 ou 1 est une feuille, et représente une région homogène (tout noir ou tout blanc).
— Chaque autre nœud à la valeur 2 et a exactement deux fils, qui représentent les deux sous-régions issues du découpage de la région du nœud père.
— Les nœuds de niveau pair (dont la racine) correspondent à des régions carrées. Leur fils gauche et droit correspondent respectivement aux sous-régions du haut et du bas.
— Les nœuds de niveau impair correspondent à des régions rectangulaires deux fois plus larges que hautes. Leur fils gauche et droit correspondent respectivement aux sous-régions de gauche et de droite.

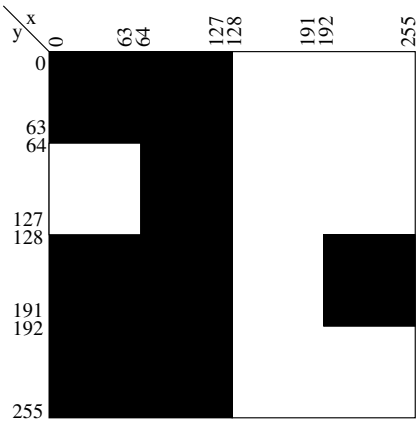


FIGURE 1 – Image binaire

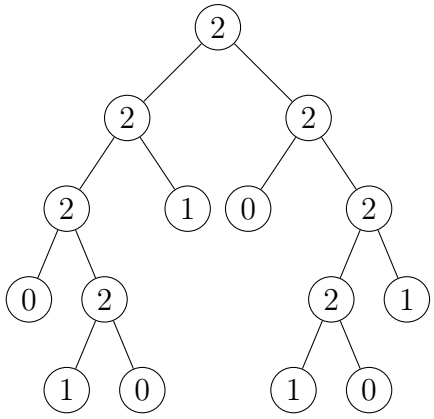


FIGURE 2 – Arbre binaire associé

Exemple de représentation d'une image par un arbre binaire

Soit l'image binaire représentée dans la figure 1. La figure 2 illustre la représentation de cette image binaire par un arbre binaire. Les feuilles de valeur 1 de l'arbre correspondent aux régions blanches de l'image de la figure 1, définies par les rectangles suivants (donnés sous la forme [x1, y1, x2, y2], où (x1, y1) sont les coordonnées du coin supérieur gauche et (x2, y2) celles du coin inférieur droit) :
$[0, 64, 63, 127]$, $[128, 0, 255, 127]$, $[128, 128, 191, 191]$ et $[128, 192, 255, 255]$.

Seules les valeurs (0, 1 ou 2) sont stockées dans les nœuds de l'arbre. Les coordonnées des régions représentées sont implicites : elles dépendent de la position du nœud dans l'arbre.

2 Manipulation d'images représentées par des arbres

On souhaite appliquer diverses manipulations sur les images binaires décrites précédemment à travers des opérations faites sur les arbres binaires associés.

2.1 Structure du projet

Le TP se présente sous la forme d'un projet *Maven*, mis à disposition sur le dépot *git* du cours à l'adresse `https://gitlab2.istic.univ-rennes1.fr/pra/tp-arbres-images`. Vous pouvez l'ouvrir avec l'IDE de votre choix (plus de détails sur comment faire dans le README du projet).

Le projet contient :

— Les squelettes des classes à implémenter `TreeImage` et `BinaryTreeImpl`
— Les classes exécutables `TpArbre` et `Benchmark`
— Une bibliothèque lib-tp-arbres.jar contenant des versions déjà compilées des interfaces et classes utilitaires.

2.2 Présentation des classes fournies

L'ensemble des opérations possibles sur les images binaires est spécifié dans l'interface `Image`. Celle-ci spécifie toutes les transformations possibles sur les images binaires, et étend également les

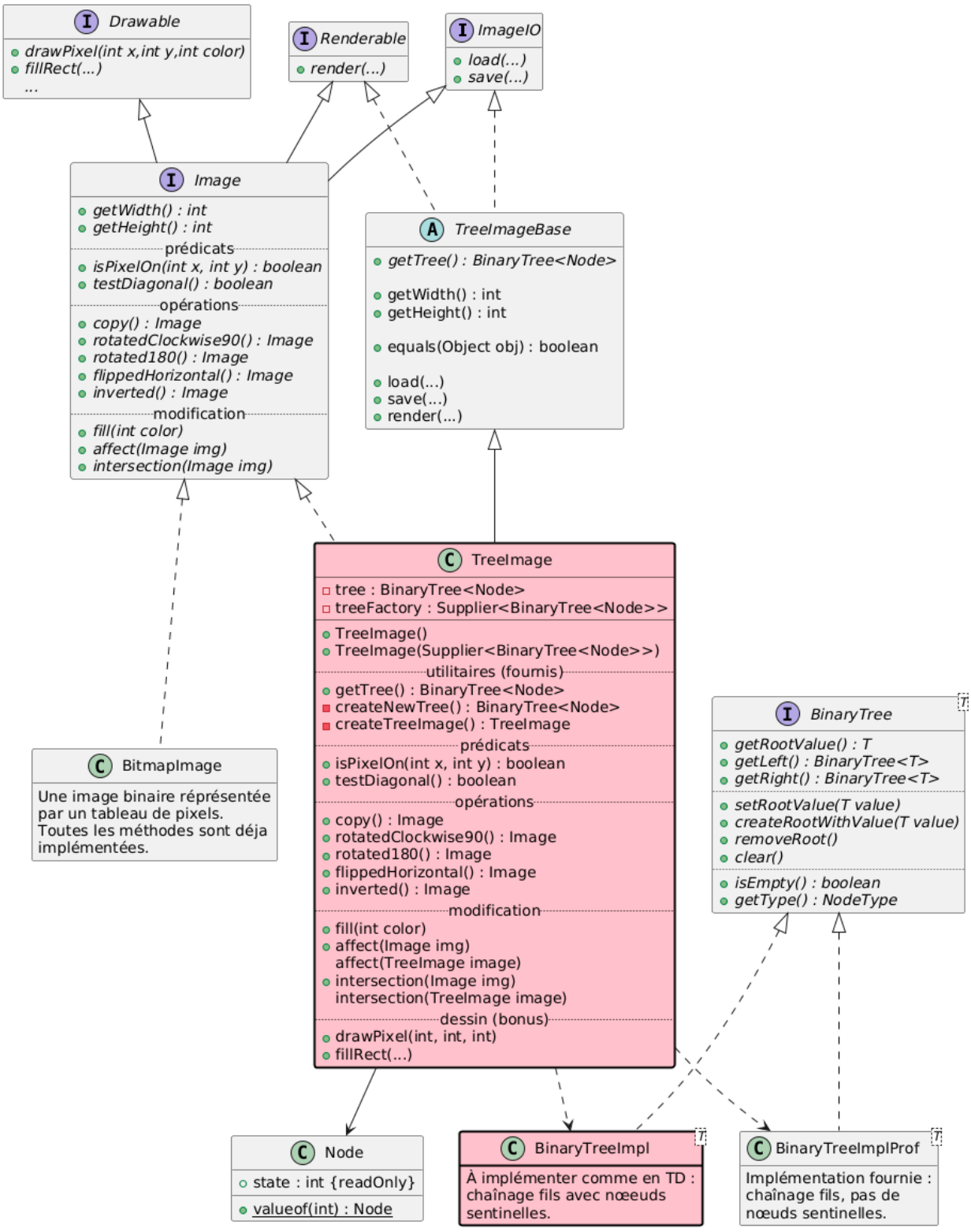


FIGURE 3 – Diagramme des classes fournies (les classes à implémenter sont mises en évidence en gras)

interfaces utilitaires `Drawable` (dessin sur une image), `Renderable` (affichage dans une fenêtre graphique) et `ImageIO` (lecture/écriture d'images depuis un fichier).

Vous implémenterez dans la classe `TreeImage` les méthodes de l'interface `Image` correspondant aux opérations sur les images, implémentées à l'aide de l'arbre binaire. Les méthodes d'affichage et de lecture/écriture sont déjà implémentées dans la classe `AbstractImage` dont hérite `TreeImage`.

Dans la classe `TreeImage`, l'arbre binaire représentant les régions de l'image est implémenté par l'attribut privé `tree` de type `BinaryTree<Node>`.

Par défaut, on utilisera la classe `BinaryTreeImplProf` fournie, qui implémente les arbres binaires par une représentation interne différente de celle vue en cours.

Le constructeur de la classe `TreeImage` peut prendre en paramètre une usine à objets `BinaryTree`, afin de permettre l'utilisation d'autres implémentations si nécessaire (en particulier votre propre implémentation quand vous l'aurez développée).

La classe `Node` représente les nœuds de l'arbre binaire, et possède un attribut `state` qui peut prendre les valeurs 0, 1 ou 2 (voir plus haut). Cet attribut est public mais non modifiable (`final`). Pour créer un nœud de valeur `i`, on utilise l'usine à objets `Node.valueOf(i)`. En pratique, seules trois instances de `Node` existent en mémoire (une pour chaque valeur) et sont partagées par tous les nœuds des arbres.

2.3 Méthodes de l'interface Image

```
public interface Image extends ImageIO, Drawable, Renderable {
  /** @return la largeur de l'image */
  int getWidth();

  /** @return la hauteur de l'image */
  int getHeight();

  /**
   * Lit la couleur d'un pixel dans l'image (allumé ou éteint).
   * @param x  abscisse du point
   * @param y  ordonnée du point
   * @return true si le point (x, y) est allumé, false sinon
   */
  boolean isPixelOn(int x, int y);

  /**
   * Remplit toute l'image avec la couleur spécifiée.
   * @param color  couleur (1=allumé, 0=éteint)
   */
  void fill(int color);

  /**
   * @return true si tous les points de la forme (x, x) sont allumés
   *         dans this, false sinon
   */
```

```
  boolean testDiagonal();

  /**
   * Crée une copie de l'image (clone).
   * @return une nouvelle image identique à this et de même type
   */
  Image copy();

  /**
   * Rotation de l'image de 90 degrés dans le sens des aiguilles d'une montre.
   * @return une nouvelle image tournée de 90 degrés vers la droite
   */
  Image rotatedClockwise90();

  /**
   * Rotation de l'image de 180 degrés.
   * @return une nouvelle image tournée de 180 degrés
   */
  Image rotated180();

  /**
   * Retournement horizontal de l'image (inversion gauche-droite)
   * @return une nouvelle image retournée horizontalement
   */
  Image flippedHorizontal();

  /**
   * Inversion des couleurs de l'image (noir devient blanc et blanc devient
    ↪  noir).
   * @return une nouvelle image, version inversée de this
   */
  Image inverted();

  /**
   * Affectation : this devient identique à image.
   * @param image  image à recopier dans this
   */
  void affect(Image image);

  /**
    * Intersection : this devient l'intersection de this et image2.
   * Un pixel reste allumé s'il était allumé dans this et dans image2
   */
  void intersection(Image image2);
}
```

```java
/** BONUS **/

/**
 * Allume ou éteint un pixel dans l'image.
 * @param x  abscisse du point
 * @param y  ordonnée du point
 * @param color  couleur (1=allumé, 0=éteint)
 */
public void drawPixel(int x, int y, int color) {
    // Implémenté dans TreeImage en dessinant un rectangle de 1x1 pixel avec
    ↪ fillRect
    fillRect(x, y, 1, 1, color);
}

/**
 * Remplit un rectangle dans l'image avec la couleur spécifiée.
 * @param x  abscisse du coin supérieur gauche du rectangle
 * @param y  ordonnée du coin supérieur gauche du rectangle
 * @param w  largeur du rectangle
 * @param h  hauteur du rectangle
 * @param color couleur de remplissage (0 ou 1)
 */
void fillRect(int x, int y, int w, int h, int color);
```

3 Test de l'implémentation de la classe `TreeImage`

3.1 Utilisation des tests unitaires JUnit de la classe `TreeImageTest`

Pour tester votre implémentation de la classe `TreeImage`, vous disposez d'un jeu de tests JUnit dans le fichier TreeImageTest.java. Ce programme de tests unitaires utilise en particulier des fichiers ".tree" décrivant des images sous forme d'arbres binaires (disponibles dans le répertoire "images").

3.2 Test interactif avec la classe `TpArbre`

La classe `TpArbre` [1] permet d'obtenir une interface écran comportant quatre fenêtres graphiques, correspondant à quatre images que le programme peut traiter.

Toutes les opérations sur les images sont accessibles par un menu déclenché par le clic droit sur l'une des fenêtres (voir figure 4).

Les arbres binaires représentant les images peuvent être affichés en cochant l'option "Afficher l'arbre" dans le menu. Attention, pour les images complexes, l'arbre peut être très grand et son affichage peut ralentir le programme.

1. `fr.istic.pra.tp_arbres.TpArbre`

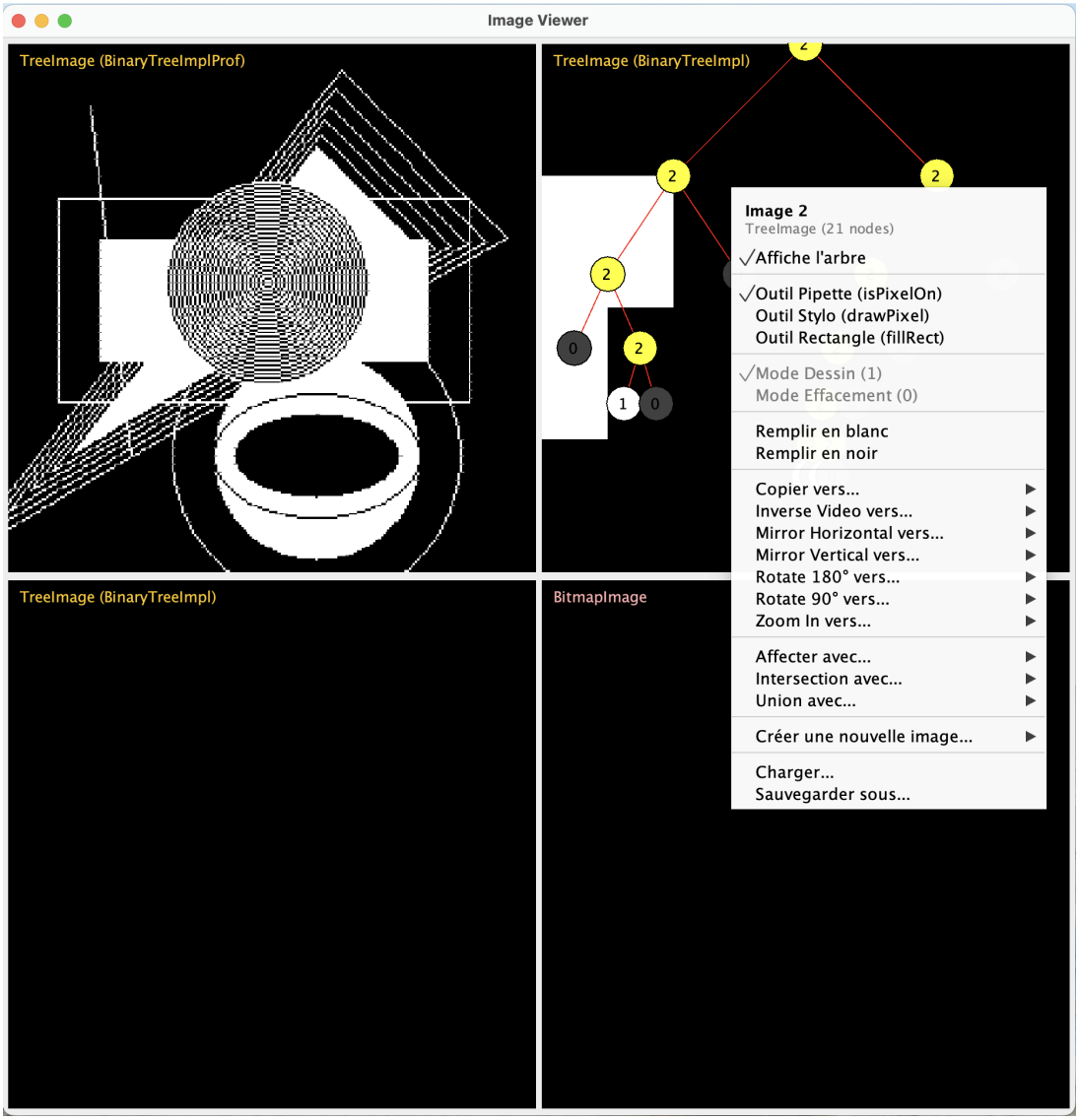


FIGURE 4 – Vue écran de la classe TpArbre

Les opérations sont appliquées à l'image affichée dans la fenêtre où le menu a été déclenché. Certaines opérations nécessitent une seconde image source (par exemple pour l'union, l'intersection, etc.) ou destination (par exemple pour inverse, rotate, etc.). Dans ce cas, la seconde image est choisie par un sous-menu.

Le bouton gauche de la souris permet soit d'appeler "isPixelOn", soit de dessiner sur l'image courante (en noir ou blanc selon le mode choisi dans le menu).

Il est possible d'instancier des images de différentes classes implémentant l'interface *Image* en utilisant le menu "Créer une nouvelle image". Vous pouvez ainsi tester les méthodes avec la classe *BitmapImage* fournie (implémentation de l'image en tableau) ou avec votre implémentation de *TreeImage*. On peut également charger une image depuis un fichier ou la sauvegarder. Les fichiers sont de type ".tree" pour les arbres (format texte dédié), et ".pbm" pour les images bitmap (format texte Portable Bitmap). Attention à ne pas écraser les fichiers images fournis, car ils sont utilisés pour les

tests.

3.3 Mesure de performances avec la classe `Benchmark`

La classe fournie `Benchmark`[2] vous permet de mesurer les temps d'exécution des méthodes de manipulation d'images que vous avez implémentées, sur différentes images de test plus ou moins complexes. Les opérations sont également réalisées sur les mêmes images en représentation matricielle (Bitmap) afin de pouvoir comparer les performances des deux types de représentation.

Étudiez la différence de performance entre les représentations en fonction de la complexité de l'image (nombre de nœuds de l'arbre). Essayez de mettre en évidence d'autres critères affectant les performances.

4 Écriture d'une mise en œuvre des arbres binaires

Dans un second temps, vous écrirez votre propre implémentation des arbres binaires.

Complétez la classe `BinaryTreeImpl.java` du package `fr.istic.pra.util` pour une mise en œuvre des arbres binaires par une représentation chaînée par références sans lien père, comme vu en cours. La classe `BinaryTreeImpl` doit implémenter les méthodes de l'interface `BinaryTree<T>`.

Pour tester votre implémentation, vous pouvez utiliser la classe `TpArbre` en choisissant votre implémentation dans le menu "Choisir l'implémentation des arbres". Des tests unitaires *JUnit* sont également fournis dans la classe `BinaryTreeImplTest`.

Vous pouvez aussi modifier le constructeur par défaut de la classe `TreeImage` pour utiliser votre implémentation des arbres binaires, et ainsi tester l'ensemble des opérations sur les images binaires.

5 Travail à rendre

Les fichiers complétés `TreeImage.java` et `BinaryTreeImpl.java` sont à rendre au plus tard le lundi 17 novembre 2025 à 20h, dans l'espace de dépôt dédié sur la page Moodle du cours[3].

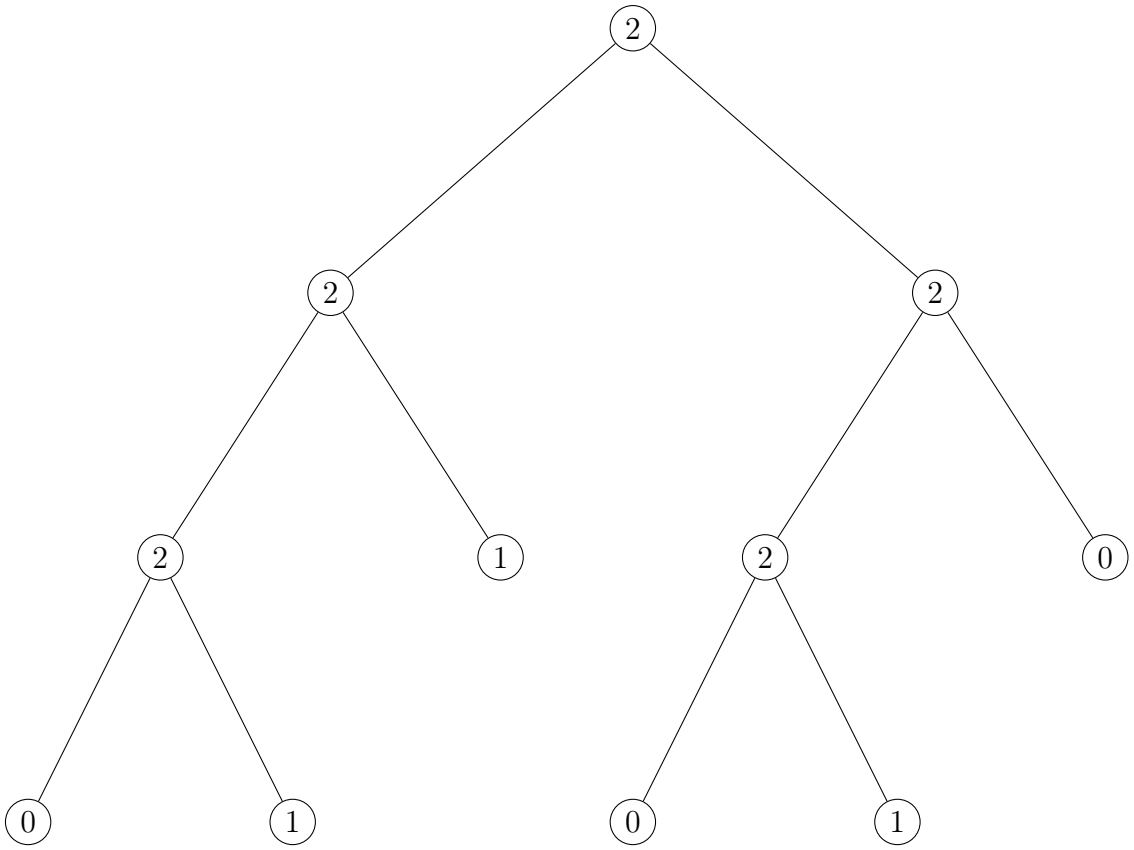
Vous veillerez à ne pas modifier les noms de classes et les packages des squelettes de code fournis. Le code source livré devra être entièrement compilable et l'en-tête de vos fichiers sera un commentaire (idéalement de type *JavaDoc*) indiquant les noms et prénoms des auteurs (les deux membres du binôme).

Attention, si vous programmez sur votre ordinateur personnel, vos classes doivent pouvoir être utilisées dans un environnement du type des salles de TP de l'Istic.

2. `fr.istic.pra.tp_arbres.TpArbre`
3. `https://foad.univ-rennes.fr/course/view.php?id=1017479`

Travail préliminaire pour la première séance de TP

Soit l'arbre binaire suivant représentant une image binaire :



Notez sur chaque nœud les coordonnées $(x_{inf}, y_{inf}, x_{sup}, y_{sup})$ de la région de l'image qu'il représente. Notez sur chaque arc quelle séparation de l'image est effectuée (par exemple $y < 128$, $y \geq 128$).

Soient les deux arbres binaires suivants :

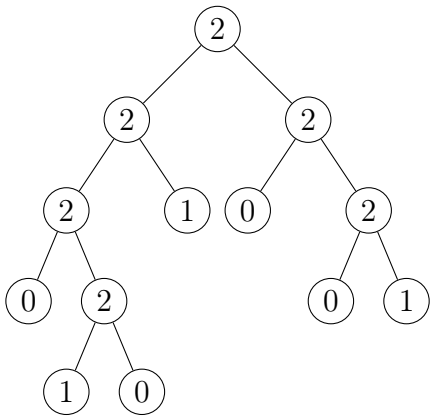


FIGURE 5 – Arbre binaire A1

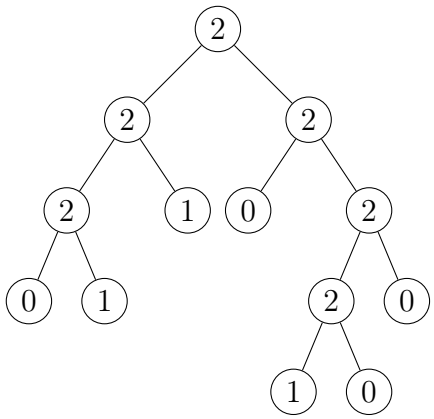


FIGURE 6 – Arbre binaire A2

Donnez l'arbre binaire résultat de l'opération d'intersection de A1 et A2. On rappelle que l'intersection de deux images binaires est définie par le fait qu'un pixel est allumé dans l'image résultante si et seulement si il est allumé dans les deux images sources.