8. A propos d'arbres binaires

Cet exercice est une suite de réflexions sur la réalisation en Java de la structure de données d'arbre binaire, une notion supposée connue. Pour fixer les idées nous nous donnons l'objectif suivant : écrire une application effectuant la représentation graphique d'un arbre binaire, comme montré sur la figure 1.

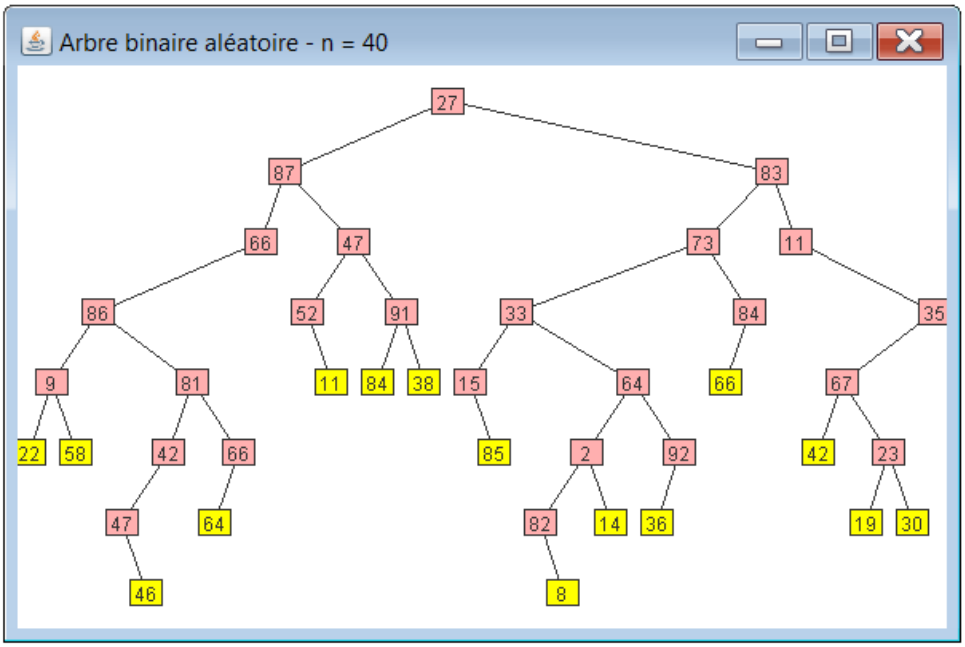

FIGURE 1 – Représentation graphique d'un arbre binaire

Exercice 8.1. Écrivez l'interface `Arbin` spécifiant les méthodes qu'il est nécessaire de posséder pour pouvoir prétendre être un arbre binaire, à savoir :

- `Object contenu()` renvoie l'information contenue dans la racine.
- `boolean existeFilsGauche()` et `boolean existeFilsDroit()` renseignent sur l'existence des fils de la racine.
- `Arbin filsGauche()` et `Arbin filsDroit()` donnent accès aux fils de la racine.
- `Arbin nouveau()` crée un nouvel arbre binaire, réduit à un nœud, *exactement de même type* que la racine.
- `void fixerContenu(Object c)` initialise ou modifie le contenu de la racine.
- `void creerFilsGauche()` et `void creerFilsDroit()` créent de nouveaux descendants directs de la racine.
- `boolean externe()` est vrai si et seulement si la racine est un nœud externe.
- `int hauteur()` renvoie la hauteur de l'arbre.
- `void preOrdre(Visiteur traitement, Stack chemin)` et `void inOrdre(Visiteur traitement, Stack chemin)` effectuent le parcours de l'arbre. Lors de l'appel d'une de ces méthodes, le premier argument doit être un objet implémentant l'interface `Visiteur`, décrit plus loin, tandis que le deuxième argument doit être une pile, en principe vide.

Les quatre méthodes restantes sont en rapport avec le placement dans un plan, rapporté à des axes de coordonnées, des nœuds d'un arbre. Tout objet implémentant l'interface `Arbin` doit fournir ces méthodes, mais il n'est pas obligatoire qu'elles aient un effet :

- `void setPosition(int x, int y)` définit la position de la racine.
  L'unité sur l'axe des abscisses [resp. des ordonnées] est la largeur [resp. la hauteur] d'un nœud (autrement dit, dans les deux directions, on compte en *nombre de nœuds*).

- `int getX()` et `int getY()` obtiennent la position de la racine,
- `int calculerPositions(int margeGauche, int margeHaute)` implémente un algorithme de placement des nœuds.

  L'arbre tout entier doit être placé dans le plus petit rectangle dont le bord gauche a `margeGauche` pour abscisse et le bord supérieur `margeHaute` pour ordonnée [1]. La méthode renvoie pour résultat la marge droite de ce rectangle.

Nous avons mentionné l'interface `Visiteur`. Elle se réduit à une unique méthode, représentant le traitement que l'on fait subir à chaque nœud lors d'un parcours de l'arbre :

```
public interface Visiteur {
    void visiter(Stack chemin);
}
```

Lors d'un parcours de l'arbre, la méthode `visiter` est appelée pour chaque nœud. Son argument est une pile de nœuds (donc d'objets `Arbin`) qui représente le chemin joignant la racine (i.e. le nœud d'où le parcours a été initié) au noeud en question.

Exercice 8.2. Écrivez la *classe abstraite* `ArbinAbstrait`, implémentant l'interface `Arbin`. Elle définit celles des méthodes de l'interface `Arbin` qui peuvent être écrites sans qu'il soit nécessaire d'en savoir plus à propos de l'implémentation effective de l'arbre : `externe`, `toString`, `preOrdre`, `inOrdre`, `hauteur` et `calculerPositions`.

De plus, on définira dans cette classe des versions inopérantes de `setPosition`, `getX` et `getY`.

Exercice 8.3. Écrivez la classe `Pepiniere`, une classe « fabrique » servant à construire des arbres particuliers. On y définira notamment les méthodes :

`public static void arbinTrivial(Arbin a, int min, int max)` construit un arbre binaire aussi équilibré que possible, avec `a` pour racine et `max`−`min`+1 nœuds, de même type que `a`, ayant pour contenu *tous* les nombres entiers compris entre `min` et `max`.

`public static void arbinAleatoire(Arbin a, int n, int min, int max)` construit un arbre binaire avec `a` pour racine et `n` nœuds, de même type que `a`, ayant comme contenu des nombres aléatoires compris entre `min` et `max`. La structure de l'arbre est elle-même déterminée par des tirages aléatoires.

Notez que ces méthodes n'ont pas besoin de connaître des implémentations concrètes de l'intarface `Arbin` pour construire les arbres qu'on leur demande.

Exercice 8.4. Écrivez la classe `DessinArbin`, sous-classe de `Panel`, qui prend en charge le dessin d'un `Arbin`. On s'inspirera de la figure au debut de ce sujet.
Bien entendu, le « cœur » de l'algorithme est la méthode `calculerPositions`, déjà écrite.

Exercice 8.5. Écrivez la classe `ArbinPointé`, sous-classe de `ArbinAbstrait`, qui représente une implémentation concrète de l'interface `Arbin` : chaque nœud est un objet contenant une information et deux références vers les deux fils.

Exercice 8.6. Écrivez la classe `ArbinPointéPlacé`, sous-classe de `ArbinPointé`, dont les instances sont des objets `ArbinPointé` capables de mémoriser leur position (autrement dit, cette classe fournit des méthodes `setPosition`, `getX` et `getY` non vides).

Exercice 8.7. Pour essayer tout cela, écrivez une application qui construit un `ArbinPlacéPointé` aléatoire, puis crée un cadre et y place un `DessinArbin` chargé de représenter cet arbre.

• • •

1. Puisqu'on fait du Java, on suppose que l'origine est en haut et à gauche, que les abscisses grandissent vers la droite et les ordonnées vers le bas.