6. Mémorisation et dessin d'arbres binaires

Cet exercice est une version *light* de celui proposé à la série 8. Il s'agit de définir une classe pour la mémorisation et la représentation graphique d'arbres binaires (une structure de données supposée connue).

Pour fixer les idées nous dirons qu'il s'agit d'arbres dont les noeuds portent comme informations associées des entiers :

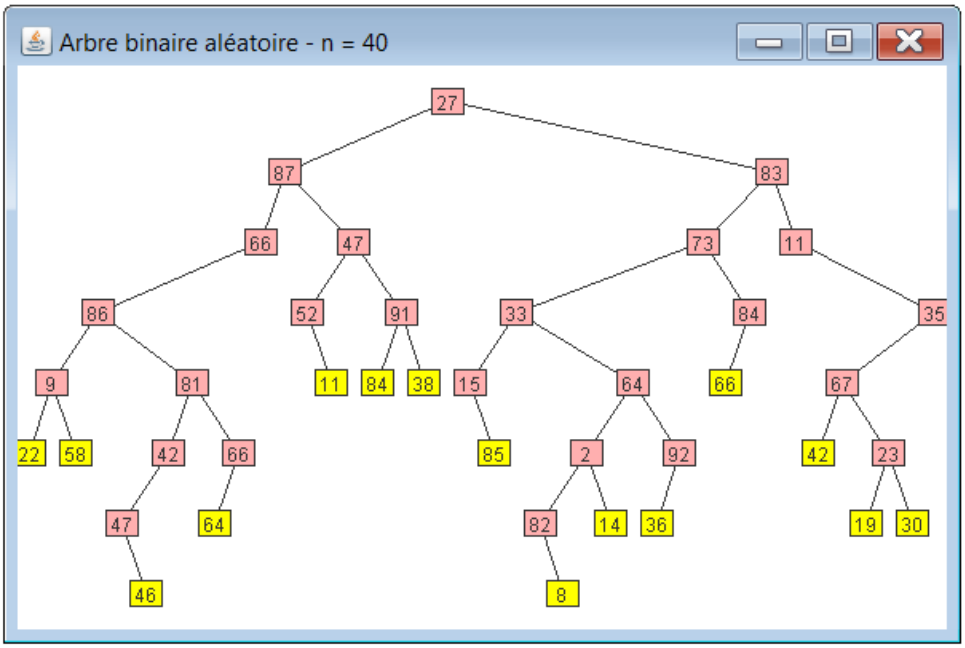

FIGURE 1 – Représentation graphique d'un arbre binaire

Exercice 6.1. Écrivez la classe `Arbin`. D'une part, il faudra des variables d'instance (privées) :
- un entier qui est l'information portée par [la racine de] l'arbre binaire,
- deux entiers qui sont les coordonnées de [la racine de] l'arbre binaire, en vue du dessin de ce dernier,
- deux arbres binaires qui sont les fils de [la racine de] l'arbre binaire.

D'autre part, on écrira au moins les méthodes publiques suivantes :

`int contenu()` renvoie l'information contenue dans [la racine de] l'arbre binaire.

`boolean existeFilsGauche()` et `boolean existeFilsDroit()` renseignent sur l'existence des fils de [la racine de] l'arbre binaire.

`Arbin filsGauche()` et `Arbin filsDroit()` donnent accès aux fils de [la racine de] l'arbre binaire.

`void fixerContenu(int c)` initialise ou modifie le contenu de [la racine de] l'arbre binaire.

`void fixerFilsGauche(Arbin fg)` et `void fixerFilsDroit(Arbin fd)` créent les descendants directs de [la racine de] l'arbre binaire.

`boolean externe()` est vrai si et seulement si [la racine de] l'arbre binaire est un nœud externe (on dit aussi une feuille).

`int hauteur()` renvoie la hauteur de l'arbre, c'est-à-dire le nombre de nœuds de sa plus longue branche.

Les quatre méthodes suivantes sont en rapport avec le placement dans un plan, rapporté à des axes de coordonnées, des nœuds d'un arbre :

`void fixerPosition(int x, int y)` définit la position de la racine de l'arbre binaire.
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 X()` et `int Y()` renvoient les coordonnées position de la racine de l'arbre binaire,

`int calculerPositions(int xCourant, int yCourant)` implémente un algorithme de placement des nœuds. Cette méthode parcourt l'arbre en donnant aux variables `x` et `y` de chaque nœud les valeurs qui déterminent sa position dans le dessin de l'arbre.

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

On écrira aussi les constructeurs :

`Arbin(int info)` qui construit un arbre binaire réduit à un unique nœud portant l'information indiquée.

`Arbin(int min, int max)` qui construit un arbre binaire aussi équilibré que possible, avec `max`−`min`+1 nœuds dont les contenus sont *tous* les nombres entiers compris entre `min` et `max`.

`Arbin(int n, int min, int max)` qui construit un arbre binaire ayant `n` nœuds dont les contenus sont des nombres aléatoires compris entre `min` et `max`. La structure de l'arbre est elle-même déterminée par des tirages aléatoires.

On pourra aussi écrire les méthodes utilitaires suivantes :

`void preOrdre(Visiteur traitement, Stack chemin)`

`void inOrdre(Visiteur traitement, Stack chemin)`

`void postOrdre(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 ci-dessous, tandis que le deuxième argument doit être une pile de sommets (initialement vide).

L'interface `Visiteur` se réduit à une unique méthode, représentant le traitement qu'on fait subir à chaque nœud lors d'un parcours de l'arbre :

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

Pendant le 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 6.2. Pour essayer tout cela, écrivez une application qui affiche un arbre binaire comme montré sur la figure 1.

On définira une classe `Panneau`, sous-classe de `javax.swing.JPanel`, dont la méthode `paint` prendra en charge le tracé des rectangles qui représentent les nœuds et les segments qui relient ces derniers. Le dessin de l'arbre devra occuper la totalité de l'espace disponible, y compris lorsque l'utilisateur modifiera la taille du cadre.

On écrira aussi une méthode principale qui crée un cadre (objet `javax.swing.JFrame`), y place un objet `Panneau` comme défini ci-dessus et affiche le tout.

• • •

1. On suppose que l'origine est en haut et à gauche, que les abscisses grandissent vers la droite et les ordonnées vers le bas.