CS 2210a — Data Structures and Algorithms
Assignment 4
Representing Figures with Binary Search Trees
Due Date: November 15, 11:59 pm
Total marks: 20

1 Overview

In this assignment you will write code for manipulating graphical figures, or simply figures, where a graphical figure is just a set of pixels forming an image. We are interested in displaying the figures and moving them around, detecting collisions when they occur.

The program that displays the figures will receive as input a file containing a list of names of image files, each corresponding to a figure. The figures will be rendered on a window and the user will move some them around using the keyboard. Figures cannot overlap, so your program will allow a figure to move only when its movement would not cause it to overlap with other figures or with the borders of the window. For this assignment, there will be four kinds of figures:

- fixed figures, which cannot move
- figures that can be moved by the user
- figures that are moved by the computer
- target figures, that disappear when the user-controlled figures run into them.

These figures will be part of a "pac-man"-like game. Figures moved by the computer will chase the user controlled ones. These latter in turn will try to get rid of the target figures. Fixed figures constrain the movement of the mobile ones.

We will provide code for reading the input file, for displaying the figures and for reading the user input. You will have to write code for storing the figures and for detecting overlaps between them.

2 The Figures

As stated above, each figure consists of a set of pixels. Each pixel is defined by 3 values x, y, and c; (x, y) are the coordinates of the pixel and c is its color. We will think that each figure f is enclosed in a rectangle r_f (so all the pixels are inside this rectangle and no smaller rectangle contains all the pixels; see Figure 1 below). The width and height of this rectangle are the width and height of the figure. To determine the position where a figure should be displayed, we need to give the coordinates (u_x, u_y) of the upper-left corner of its enclosing rectangle; (u_x, u_y) is called the *offset* of the figure.

For specifying coordinates, we assume that the upper-left corner of the window ω where the figures are displayed has coordinates $(0, 0)$. The coordinates of the lower-right corner of ω are (W, H), where W is the width and H is the height of ω.

Each figure will have an identifier; this is just an integer number that is used to distinguish a figure from another, as two figures might be identical (but they cannot be in the same position).

The pixels of a figure f will be stored in a binary search tree. Each node in the tree stores a data item of the form `(location,color)` representing one pixel, where `location` $= (x, y)$ contains the coordinates of the pixel **relative** to the upper-left corner of the rectangle r_f enclosing the figure. For example, the coordinates of the black dot in Figure 1 below are $(20, 10)$, so this black dot corresponds to the pixel $((20, 10)$,`black`). As shown in Figure 1, the offset of figure f_1 is $(40, 25)$, so when rendering f_1 inside the window ω the actual position of the black dot is $(20 + 40, 10 + 25) = (60, 35)$.

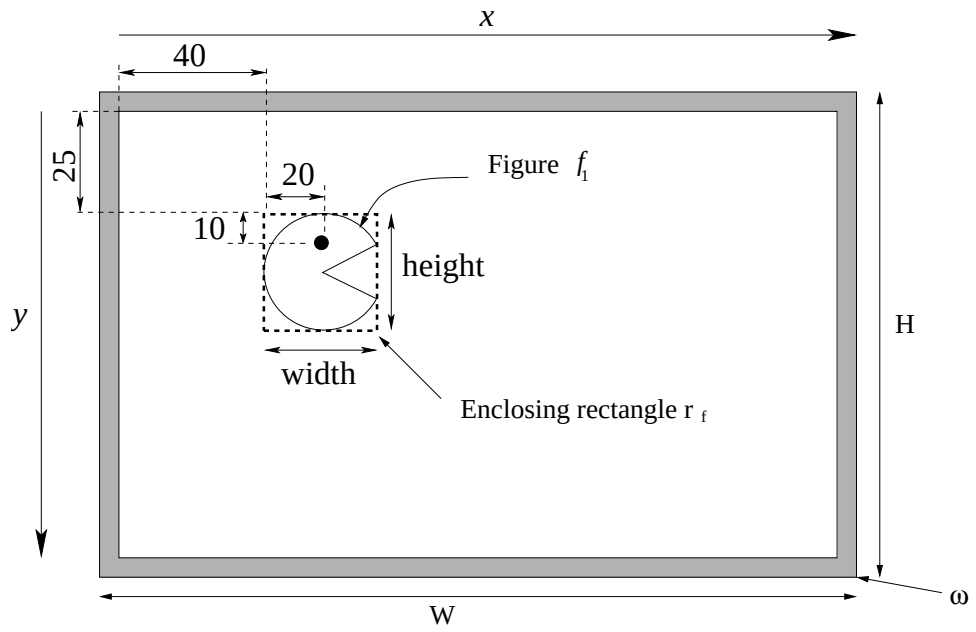

Figure 1 Window ω.

Note that by storing the pixels in the binary search tree with coordinates relative to the figure's enclosing rectangle, the data stored in the tree does not need to change when the figure moves: The only thing that needs to change is the offset of the figure.

For each data item `(location,color)` stored in the tree, the attribute `location` is used as the key. To compare two locations (x, y) and (x', y') we use *column order*. In column order, $(x, y) < (x', y')$ if either

- $x < x'$, or
- $x = x'$ and $y < y'$

So, for example, $(1, 4) < (2, 3)$ and $(5, 3) < (5, 7)$.

3 Moving Figures

As stated above, two figures cannot overlap and a figure cannot go outside the window ω. Hence, when the user tries to move a figure, we need to verify that such a movement would not cause it to cross the boundaries of the window or to overlap another figure.

A movement can be represented as a pair (d_x, d_y), where d_x is the distance to move horizontally and d_y is the distance to move vertically. To check whether a movement (d_x, d_y) on figure f with offset (x_f, y_f), width w_f and height h_f is valid, we first update the offset of f to $(x_f + d_x, y_f + d_y)$ and then check whether this new position for f would cause an overlap with another figure or with the window's borders. To do this efficiently we proceed as follows:

- Check whether the enclosing rectangle r_f of f crosses the borders of the window ω. For example, to check whether r_f crosses the right border of ω we test if $x_f + d_x + w_f \geq W$; recall that W is the width of ω.

- If r_f does not cross the borders of ω, then we check whether r_f intersects the enclosing rectangle $r_{f'}$ of another figure f'. If there is no such intersection then f does not intersect other figures or the window's borders, so the movement (d_x, d_y) is valid.

- On the other hand, if r_f intersects the enclosing rectangles of some set S of figures, then for each figure $f' \in S$ we check whether f and f' overlap and if so, then this movement should not be allowed.

  Note that for f and f' to overlap, f must have at least one pixel $((x, y), c)$ and f' must have a pixel $((x', y'), c')$ that would be displayed at precisely the same position on ω, or in other words, $x + x_f = x' + x_{f'}$ and $y + y_f = y' + y_{f'}$, where $(x_{f'}, y_{f'})$ is the offset of f'. Observe that if these pixels exist then $x + x_f - x_{f'} = x'$ and $y + y_f - y_{f'} = y'$. Therefore, to test whether f and f' overlap we can use the following algorithm:

  For each data item $((x, y), c)$ stored in the binary search tree t_f storing the pixels of f **do**
  (1) **if** in the tree $t_{f'}$ storing the pixels of f' there is a data item $((x', y'), c')$ with key $(x', y') = (x + x_f - x_{f'}, y + y_f - y_{f'})$, **then** the figures overlap.
  if Condition (1) is never satisfied, then the figures do not overlap.

  In the **for** loop above, to consider all the data items $((x, y), c)$ stored in the nodes of the tree t_f we can use the binary search tree operations `smallest()` and `successor()`.

4 Classes to Implement

You need to implement the following Java classes: `Location`, `Pixel`, `BinarySearchTree`, `BinaryNode`, `GraphicalFigure`, `DuplicatedKeyException`, `InexistentKeyException`, and `EmptyTreeException`. You can implement more classes if you need to. **You must write all the code yourself**. You **cannot** use code from the textbook, the Internet, or any other sources: however, you may implement the algorithms discussed in class.

4.1 Location

This class represents the position (x, y) of a pixel. For this class you must implement all and only the following `public methods`:

- `public Location(int x, int y)`: A constructor that initializes `this Location` object with the specified coordinates.
- `public int xCoord()`: Returns the x coordinate of **this** `Location`.
- `public int yCoord()`: Returns the y coordinate of **this** `Location`.
- `public int compareTo (Location p)`: Compares **this** `Location` with p using column order (defined above):
  - if **this** $>$ p return 1;
  - if **this** $=$ p return 0;
  - if **this** $<$ p return -1.

You can implement any other methods that you want to in this class, but they must be declared as `private` methods (i.e. not accessible to other classes).

4.2 Pixel

This class represents the data items to be stored in the binary search tree. Each data item consists of two parts: a `Location` and an `int` color. For this class you must implement all and only the following `public` methods:

- `public Pixel(Location p, int color)`: A constructor which initializes the new `Pixel` with the specified coordinates and color. `Location p` is the key for the `Pixel`.

- `public Location getLocation()`: Returns the `Location` of the `Pixel`.
- `public int getColor()`: Returns the color of the `Pixel`.

You can implement any other methods that you want to in this class, but they must be declared as `private` methods.

4.3 BinaryNode

This class represents the nodes of the binary search tree. Each node will store an object of the class `Pixel` and it will have references to its left child, its right child, and its parent. For this class you must implement all and only the following `public` methods:

- `public BinaryNode (Pixel value, BinaryNode left, BinaryNode right, BinaryNode parent)`: A constructor for the class. Stores the `Pixel` in the node and sets left child, right child, and parent to the specified values.
- `public BinaryNode ()`: A constructor for the class that initializes a leaf node. The data, children and parent are set to null.
- `public BinaryNode getParent()`: Returns the parent of **this** node.
- `public void setParent(BinaryNode parent)`: Sets the parent of **this** node to the specified value.
- `public void setLeft (BinaryNode p)`: Sets the left child of **this** node to the specified value.
- `public void setRight (BinaryNode p)`: Sets the right child of **this** node to the specified value.
- `public void setData (Pixel value)`: Stores the given `Pixel` in **this** node.
- `public boolean isLeaf()`: Returns true if **this** node is a leaf; returns false otherwise.
- `public Pixel getData ()`: Returns the `Pixel` object stored in **this** node.
- `public BinaryNode getLeft()`: Returns the left child of **this** node.
- `public BinaryNode getRight()`: Returns the right child of **this** node.

You can implement any other methods that you want to in this class, but they must be declared as `private` methods.

4.4 BinarySearchTree

This class implements an ordered dictionary using a binary search tree. Each node of the tree will store a `Pixel` object; the attribute `Location` of the `Pixel` will be its key. In your binary search tree **only the internal nodes will store information**. The leaves are nodes (leaves are **not** `null`) that do not store any data.

The constructor for the BinarySearchTree class must be of the form

```
public BinarySearchTree()
```

This will create a tree whose root is a leaf node. Besides the constructor, the only other public methods in this class are specified in the `BinarySearchTreeADT` interface and described below. In all these methods, parameter `r` is the root of the tree.

- `public Pixel get (BinaryNode r, Location key)`: Returns the Pixel storing the given key, if the key is stored in the tree; returns null otherwise.

- `public void put (BinaryNode r, Pixel data) throws DuplicatedKeyException`: Inserts the given data in the tree if no data item with the same key is already there. If a node already stores the same key, the algorithm throws a DuplicatedKeyException.
- `public void remove (BinaryNode r, Location key) throws InexistentKeyException`: Removes the data item with the given key, if the key is stored in the tree; throws an InexistentKeyException otherwise.
- `public Pixel successor (BinaryNode r, Location key)`: Returns the Pixel with the smallest key larger than the given one (note that the tree does not need to store a node with the given key). Returns null if the given key has no successor.
- `public Pixel predecessor (BinaryNode r, Location key)`: Returns the Pixel with the largest key smaller than the given one (note that the tree does not need to store a node with the given key). Returns null if the given key has no predecessor.
- `public Pixel smallest(BinaryNode r) throws EmptyTreeException`: Returns the Pixel with the smallest key. Throws an EmptyTreeException if the tree does not contain any data.
- `public Pixel largest(BinaryNode r) throws EmptyTreeException`: Returns the Pixel with the largest key. Throws an EmptyTreeException if the tree does not contain any data.
- `public BinaryNode getRoot()`: Returns the root of the binary search tree.

You can download `BinarySearchTreeADT.java` from the course's website. To implement this interface, you need to declare your `BinarySearchTree` class as follows:

```
public class BinarySearchTree implements BinarySearchTreeADT
```

You can implement any other methods that you want to in this class, but they must be declared as `private` methods.

4.5 Exception Classes

These are the classes implementing the exceptions thrown by the `insert`, `remove`, `smallest` and `largest` methods of `BinarySearchTree`. See exception classes from last asignment to see how you should implement these classes.

4.6 GraphicalFigure

The constructor for this class must be of the form

```
public GraphicalFigure (int id, int width, int height, String type, Location pos);
```

where `id` is the identifier of **this** figure, `width` and `height` are the width and height of the enclosing rectangle for **this** figure, `pos` is the offset of the figure and `type` is its type. The types of the figures are the following:

- "fixed": fixed figure
- "user": figure moved by the user
- "computer": figure moved by the computer
- "target": target figure.

Inside the constructor you will create an empty BinarySearchTree where the pixels of the figure will be stored.

Beside the constructor, the only other public methods in this class are specified in the `GraphicalFigureADT` interface:

- `public void setType (String type)`: Sets the type of **this** figure to the specified value.
- `public int getWidth ()`: Returns the width of the enclosing rectangle for **this** figure.
- `public int getHeight()`: Returns the height of the enclosing rectangle for **this** figure.
- `public String getType ()`: Returns the type of **this** figure.
- `public int getId()`: Returns the id of **this** figure.
- `public Location getOffset()`: Returns the offset of **this** figure.
- `public void setOffset(Location value)`: Changes the offset of **this** figure to the specified value
- `public void addPixel(Pixel pix) throws DuplicatedKeyException`: Inserts `pix` into the binary search tree associated with **this** figure. Throws a `DuplicatedKeyException` if an error occurs when inserting the Pixel into the tree.
- `public boolean intersects (GraphicalFigure gobj)`: Returns `true` if **this** figure intersects the one specified in the parameter. It returns `false` otherwise.

You can download `GraphicalFigureADT.java` from the course's website. To implement this interface, you need to declare your `GraphicalFigure` class as follows:

```
public class GraphicalFigure implements GraphicalFigureADT
```

You can implement any other methods that you want to in this class, but they must be declared as `private` methods.

Hint. You might find useful to implement a method, say `findPixel(Location p)`, that returns true if **this** figure has a pixel in location `p` and it returns false otherwise.

5 Classes Provided and Running the Program

The input to the program will be a file containing the descriptions of the figures to be displayed. Each line of the input file contains 4 values:

```
x y type file
```

where `(x,y)` is the offset of the figure (these two values are integer), `type` is the type of the figure (this is a String), and `file` is the name of an image file in .jpg, .bmp, or any other image format understood by java. You will be given code for reading the input file.

From the course's website you can download the following classes: `Board.java`, `Gui.java`, `MoveFigure.java`, `Show.java`, `BinarySearchTreeADT.java`, and `GraphicalFigureADT.java`. The main method is in class `Show.java`. To execute the program, on a command window you will enter the command

`java Show` *inputFile*

where *inputFile* is the name of the file containing the input for the program.

6 Testing your Program

We will run a test program called `TestBST` to check that your implementation of the `BinarySearchTree` class is as specified above. We will supply you with a copy of `TestBST` to test your implementation. We will also run other tests on your software to check whether it works properly.

7 Coding Style

Your mark will be based partly on your coding style. Among the things that we will check, are

- Variable and method names should be chosen to reflect their purpose in the program.
- Comments, indenting, and white spaces should be used to improve readability.
- No variable declarations should appear outside methods ("instance variables") unless they contain data which is to be maintained in the figure from call to call. In other words, variables which are needed only inside methods, whose values do not have to be remembered until the next method call, should be declared inside those methods.
- All variables declared outside methods ("instance variables") should be declared `private` (not `protected`), to maximize information hiding. Any access to the variables should be done with accessor methods (like `getLocation()` and `getColor()` for `Pixel`).

8 Marking

Your mark will be computed as follows.

- Program compiles, produces meaningful output: 2 marks.
- `TestBST` tests pass: 5 marks.
- `GraphicalFigure` tests pass: 3 marks
- Coding style: 2 marks.
- `BinarySearchTree` implementation: 5 marks.
- `GraphicalFigure` program implementation: 3 marks.

9 Submitting Your Program

You must submit an electronic copy of your program using OWL. Please **DO NOT** put your files in subdirectories and **DO NOT** submit a .zip, .tar or any other compressed file with your program. Make it sure you submit all your .java files not your .class files.

Read the tutorials posted in the course's website on how to configure Eclipse to read command line arguments.

When you submit your program, we will receive a copy of it with a datestamp and timestamp. If you submit your program more than once please send me an email message to let me know. We will take the latest program submitted as the final version, and will deduct marks accordingly if it is late.