CS 2210a — Data Structures and Algorithms
Assignment 4
Moving Figures
Due Date: November 15, 11:59 pm
Total marks: 20

1 Overview

In this assignment you will write code for manipulating figures, where a figure is a set of points or pixels. 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 the description of a set of figures. The figures will be rendered on a window and the user will be allowed to move some them around using the computer keyboard. Figures cannot overlap, so your program needs to check for collisions: A figure will be allowed to move only when its movement would not cause it to overlap with another figure or with the borders of the window. There are 4 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 figures. 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 collisions.

2 The Figures

As stated above, each figure consists of a set of pixels. Each pixel is defined by 3 values (x, y, c): The coordinates (x, y) of the pixel and 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 would contain all the pixels; see Fig. 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 contain the same set of pixels (but not 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 pair `(position,color)`, where `position` $= (x, y)$ 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 Fig. 1 below are $(20, 10)$, so that black dot corresponds to the pixel $((20, 10)$,`black`). Since the offset of the figure is $(40, 25)$, when rendering the figure inside the window the actual position of the black dot is $(20 + 40, 10 + 25) = (60, 35)$.

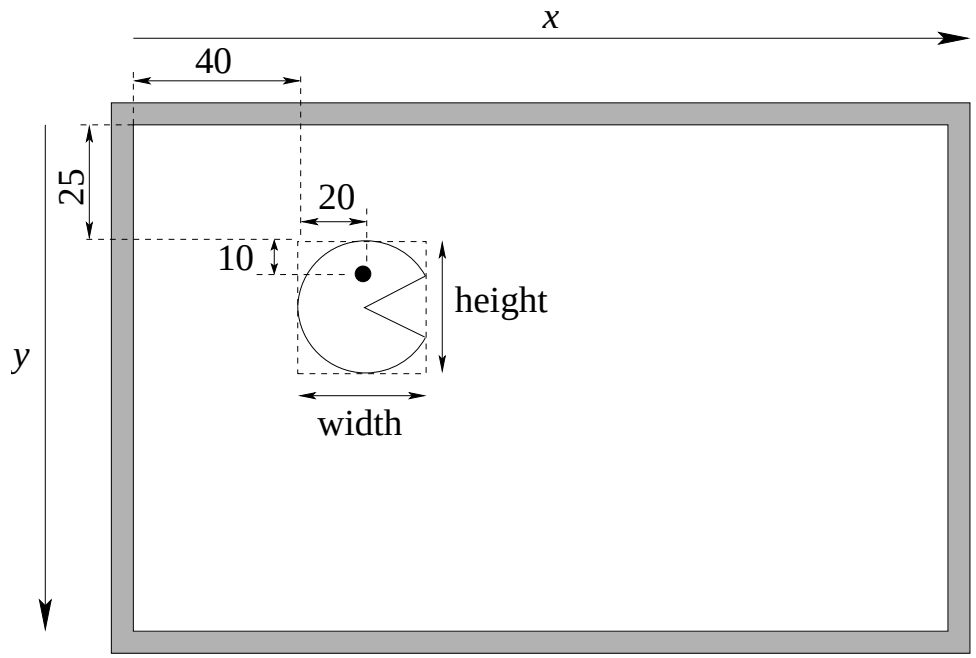

Fig. 1 Window ω.

Note that by storing in the tree pixels 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 figure's offset.

For each record `(position,color)` stored in the tree, the field `position` is used as the key. To compare two positions (x, y) and (x', y') we use *row order*: $(x, y) < (x', y')$ if either

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

So, for example, $(2, 3) < (1, 4)$ and $(3, 5) < (3, 9)$.

3 Moving Figures

As stated above, two figures cannot overlap and a figure cannot go outside the window ω. Hence, when the user requests a figure to be moved we need to verify that such a movement would not cause the figure 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 verify 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, 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; if so, then this movement should not be allowed.

  Note that if f and f' overlap, then 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'. If these conditions are satisfied 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 record $((x, y), c)$ 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 record r with key
  $(x + x_f - x_{f'}, y + y_f - y_{f'})$, then the figures overlap.
  if Condition (1) was never satisfied, then the figures do not overlap.

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

4 Classes to Implement

You are to implement at least five Java classes: `Position`, `DictEntry`, `BinarySerachTree`, `BSTException`, and `Figure`. 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 Position

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

- `public Position(int x, int y)`: A constructor which returns a new `Position` with the specified coordinates.
- `public int getX()`: Returns the x coordinate of **this** `Position`.
- `public int getY()`: Returns the y coordinate of **this** `Position`.
- `public int compareTo (Position p)`: Compares **this** `Position` with p using row 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 DictEntry

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

- `public DictEntry(Position p, int color)`: A constructor which returns a new `DictEntry` with the specified coordinates and color. `Position p` is the key for the `DictEntry`.
- `public Position getPosition()`: Returns the `Position` in the `DictEntry`.
- `public int getColor()`: Returns the color in the `DictEntry`.

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

4.3 BinarySearchTree

This class implements an ordered dictionary using a binary search tree. Each node of the tree will store a `DictEntry` object; the attribute `Position` of the `DictEntry` will be its key. In your binary search tree **only the internal nodes will store information**. The leaves will 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. Beside the constructor, the only other public methods in this class are specified in the `BinarySearchTreeADT` interface:

- `public DictEntry find (Position key)`: Returns the DictEntry storing the given key, if the key is stored in the tree. Returns null otherwise.
- `public void insert (DictEntry data) throws BSTException`: 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 BSTException.
- `public void remove (Position key) throws BSTException`: Removes the data item with the given key, if the key is stored in the tree. Throws a BSTException otherwise.
- `public DictEntry successor (Position key)`: Returns the DictEntry 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 DictEntry predecessor (Position key)`: Returns the DictEntry 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 DictEntry smallest()`: Returns the DictEntry with the smallest key. Returns null if the tree does not contain any data.
- `public DictEntry largest()`: Returns the DictEntry with the largest key. Returns null if the tree does not contain any data.

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.

4.4 BSTException

This is just the class implementing the class of exceptions thrown by the `insert` and `remove` methods of `BinarySearchTree`. See the class notes on exceptions.

4.5 Figure

The constructor for this class must be of the form

```
public Figure (int id, int width, int height, int type, Position pos);
```

where `id` is the figure's identifier, `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 figure types are

- 0: fixed figure
- 1: figure moved by the user
- 2: figure moved by the computer
- 3: target figure.
- 4: killed figure. When the figure moved by the user touches a target figure, the latter one disappears and it becomes a killed figure. Also, when a figure moved by the computer runs into a figure moved by the user, this latter disappears and it becomes a killed figure.

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

- `public void setType (int type)`: Sets type of 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 int getType ()`: Returns the type of **this** figure.
- `public int getId()`: Returns the id of **this** figure.
- `public Position getOffset()`: Returns the offset of **this** figure.
- `public void setOffset(Position value)`: Changes the offset of **this** figure to the specified value
- `public void addPixel(int x, int y, int color) throws BSTException`: Creates a DictEntry to represent the given pixel and inserts it into the binary search tree associated with **this** figure. Throws a `BSTException` if an error occurs when inserting the pixel into the tree.
- `public boolean intersects (Figure fig)`: Returns `true` if **this** figure intersects the one specified in the parameter. It returns `false` otherwise.

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

```
public class Figure implements FigureADT
```

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

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, `type` is the type of the figure (these 3 values are integer), 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. We specify the format of the input file just in case you want to create your own inputs.

From the course's website you can download the following classes: `Board.java`, `Gui.java`, `MoveFigure.java`, and `Show.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 make sure 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 object 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 `getPosition()` and `getColor()` for `DictEntry`).

8 Marking

Your mark will be computed as follows.

- Program compiles, produces meaningful output: 2 marks.

- `TestBST` tests pass: 5 marks.
- `Figure` tests pass: 3 marks
- Coding style: 2 marks.
- `BinarySearchTree` implementation: 5 marks.
- `Figure` program implementation: 3 marks.

9 Submitting Your Program

You are required to fill out and sign the Assignment Submission Form, which can be downloaded from

http://www.csd.uwo.ca/courses/CS2210a/submForm.html

Drop your submission form in the CS210 locker (locker 302 on the third floor of the Middlesex College Building, beside room MC 300) by the due date.

You must submit an electronic copy of your program. To submit your program, login to OWL and submit your java files from there.

Read the tutorials posted in the course's website to learn how to copy your program to your Gaul directory and how to test it there. Remember that the TA's will test your programs on Gaul. Please read also the tutorials on how to configure Eclipse to read command line arguments. Please do not put your code in sub-directories; this will make it easier for the markers.

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. After you submit your assignment you should receive an email message from the submissions system acknowledging the receipt of your work. **It is your responsibility** to ensure that your assignment was received by the system.