CS 2210 — Data Structures and Algorithms
Assignment 4

Due Date: November 20, 11:55 pm

Total marks: 20

1 Overview

In this assignment you will write code needed for a program to move a set of objects around the screen. There are four types of objects:

- fixed objects, which cannot move; these objects limit the movement of other objects
- user objects that can be moved with the keyboard
- objects that are moved by the computer; if an object moved by the computer touches a user object, the user object will disappear
- target objects that the user objects will try to reach; when a user object touches a target object, the target object will disappear.

The program will receive as input a file containing a list of names of image files, each corresponding to an object. The objects will be rendered on a rectangular window and the user will move their objects around using the keyboard. Objects cannot overlap, so your program will allow an object to move only when its movement would not cause it to overlap with other objects or with the borders of the window.

We will provide code for reading the input files, for rendering the objects on the screen and for reading the keyboard input. You will have to write code for storing the objects and for detecting overlaps between them.

2 The Objects

Each object is an image consisting of a set of picture elements, pels, or *pels*. Each pel is defined by 3 values x, y, and c; (x, y) are the coordinates of the pel and c is its color. We will think that each object f is enclosed in a rectangle r_f (so all the pels are inside this rectangle and no smaller rectangle contains all the pels; see Figure 1 below). The width and height of rectangle r_f are the width and height of the object. To determine the position where an object f would be displayed, we need to give the coordinates (u_x, u_y) of the upper-left corner of its enclosing rectangle r_f; (u_x, u_y) is called the *locus* of the object.

For specifying coordinates, we assume that the upper-left corner of the window ω where the objects 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 object will have a unique integer identifier used to distinguish an object from another, as two objects might be identical (but they cannot be in the same position).

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

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

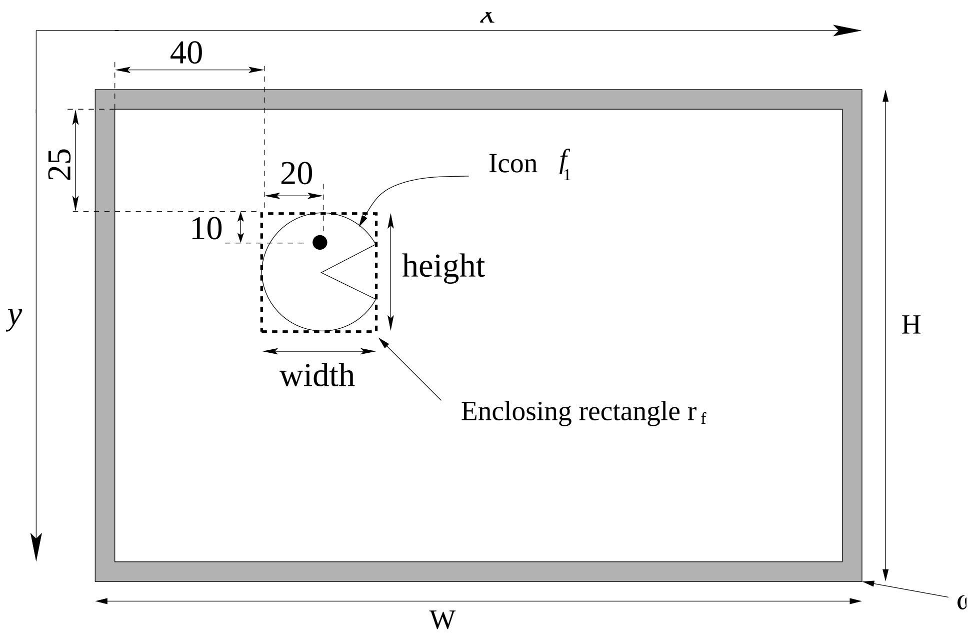

Figure 1. Window ω.

3 Classes to Implement

You need to implement the following Java classes: `Location`, `Pel`, `BinarySearchTree`, `BNode`, and `MyObject`. 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.

3.1 Location

This class represents the location (x, y) of a pel. 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 getx()`: Returns the x coordinate of **this** `Location`.
- `public int gety()`: Returns the y coordinate of **this** `Location`.
- `public int compareTo (Location p)`: returns the following values:
  - if `this.gety() > p.gety()` or if `this.gety() = p.gety()` and `this.getx() > p.getx()` return 1;
  - if `this.getx() = p.getx()` and `this.gety() = p.gety()` return 0;
  - if `this.gety() < p.gety()` or if `this.gety() = p.gety()` and `this.getx() < p.getx()` 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).

3.2 Pel

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

- `public Pel(Location p, int color)`: A constructor which initializes the new `Pel` with the specified coordinates and color. `Location p` is the key attribute for a `Pel` object.
- `public Location getLocus()`: Returns the `Location` of `this Pel`.
- `public int getColor()`: Returns the color of `this Pel` object.

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

3.3 BNode

This class represents the nodes of the binary search tree. Each node will store an object of the class `Pel` and it must 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 BNode (Pel value, BNode left, BNode right, BNode parent)`: A constructor for the class. Stores the `Pel value` in the node and sets left child, right child, and parent to the specified values.
- `public BNode ()`: A constructor for the class that initializes a leaf node. The data, children and parent attributes are set to null.
- `public BNode parent()`: Returns the parent of **this** node.
- `public void setParent(BNode newParent)`: Sets the parent of **this** node to the specified value.
- `public void setLeftChild (BNode p)`: Sets the left child of **this** node to the specified value.
- `public void setRightChild (BNode p)`: Sets the right child of **this** node to the specified value.
- `public void setContent (Pel value)`: Stores the given `Pel` object in **this** node.
- `public boolean isLeaf()`: Returns true if **this** node is a leaf; returns false otherwise.
- `public Pel getData ()`: Returns the `Pel` object stored in **this** node.
- `public BNode leftChild()`: Returns the left child of **this** node.
- `public BNode rightChild()`: 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.

3.4 BinarySearchTree

This class implements an ordered dictionary using a binary search tree. Each node of the tree will store a `Pel` object; the attribute `Location` of the `Pel` object stored in a node will be its key attribute. In a 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. Beside 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 Pel get (BNode r, Location key)`: Returns the `Pel` object storing the given key, if the key is stored in the tree; returns null otherwise.
- `public void put (BNode r, Pel newData) throws DuplicatedKeyException`: Inserts `newData` 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 (BNode 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 Pel successor (BNode r, Location key)`: Returns the `Pel` object 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 Pel predecessor (BNode r, Location key)`: Returns the `Pel` object 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 Pel smallest(BNode r) throws EmptyTreeException`: Returns the `Pel` object in the tree with the smallest key. Throws an `EmptyTreeException` if the tree does not contain any data.
- `public Pel largest(BNode r) throws EmptyTreeException`: Returns the `Pel` object in the tree with the largest key. Throws an `EmptyTreeException` if the tree does not contain any data.
- `public BNode getRoot()`: Returns the root of the binary search tree.

You can download `BinarySearchTreeADT.java` from OWL. 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.

3.5 MyObject

The constructor for this class must be of the form

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

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

- `"fixed"`: fixed object
- `"user"`: object moved by the user
- `"computer"`: object moved by the computer that chases the user objects
- `"target"`: target object.

Inside the constructor you will create an empty `BinarySearchTree` where the pels of the object will be stored.

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

- `public void setType (String type)`: Sets the type of **this** MyObject to the specified value.
- `public int getWidth ()`: Returns the width of the enclosing rectangle for **this** MyObject.

- `public int getHeight()`: Returns the height of the enclosing rectangle for **this** MyObject.
- `public String getType ()`: Returns the type of **this** Mybject.
- `public int getId()`: Returns the id of **this** MyObject.
- `public Location getLocus()`: Returns the locus of **this** MyObject.
- `public void setLocus(Location value)`: Changes the locus of **this** MyObject to the specified value
- `public void addPel(Pel pix) throws DuplicatedKeyException`: Inserts `pix` into the binary search tree associated with **this** MyObject. Throws a `DuplicatedKeyException` if an error occurs when inserting the `Pel` object `pix` into the tree.
- `public boolean intersects (MyObject otherObject)`: Returns `true` if **this** MyObject intersects the one specified in the parameter. It returns `false` otherwise. Read the next section to learn how to detect intersections between objects.

You can download `MyObjectADT.java` from OWL. To implement this interface, you need to declare your `MyObject` class as follows:

```
public class MyObject implements MyObjectADT
```

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 `findPel(Location p)`, that returns true if **this** MyObject has a `Pel` object in location `p` and it returns false otherwise.

4 Object Intersections

As stated above, objects are not allowed to overlap and an object cannot go outside the window ω. Hence, when the user tries to move one of their objects, we need to verify that such a movement would not cause it to cross the boundaries of the window or to overlap another object.

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 object f with locus (x_f, y_f), width w_f and height h_f is valid, we first temporarily update the locus 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 object or with the window's borders. You do not have to check whether an object will cross the window's borders, the java code given to you does this. However, you need to write code to check whether two objects intersect; to do this efficiently proceed as follows:

- First check whether the enclosing rectange r_f of f intersects the enclosing rectangle $r_{f'}$ of another object f'. If there is no such intersection then f does not intersect other objects.
- On the other hand, if r_f intersects the enclosing rectangles of some set S of objects, then for each object $f' \in S$ we must 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 pel $((x, y), c)$ and f' must have a pel $((x', y'), c')$ 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 locus of f'. Observe that if these pels 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 pels of $f$ do
    (1)  if in the tree $t_{f'}$ storing the pels 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 objects overlap.
if above Condition (1) is never satisfied then the objects do not overlap.
```

In this **for** loop, 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()`.

5 Classes Provided and Running the Program

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

```
x y type file
```

where `(x,y)` is the locus of the object (these two values are integer), `type` is the type of the object (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 OWL you can download the following classes: `Board.java`, `Gui.java`, `MoveObject.java`, `Run.java`, `BinarySearchTreeADT.java`, `MyObjectADT.java`, `DuplicatedKeyException`, `InexistentKeyException`, and `EmptyTreeException`. The main method is in class `Run.java`. To execute the program, on a command window you will enter the command

`java Run` *inputFile*

where *inputFile* is the name of the file containing the input for the program. If you use Eclipse you must configure it to read the input file as a command line argument. Read the comments about how to run a program from Eclipse posted in OWL and those given in the previous assignment.

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 instance variable should be used 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 instance variables should be declared `private`. Any access to the variables should be done with accessor methods (like `getLocation()` and `getColor()`.

8 Marking

Your mark will be computed as follows.

- Program compiles, produces meaningful output: 2 marks.
- `TestBST` tests pass: 5 marks.
- `MyObject` tests pass: 3 marks
- Coding style: 2 marks.
- `BinarySearchTree` implementation: 5 marks.
- `MyObject` 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 (so no `packet` statement should be used) and **DO NOT** submit a .zip, .tar or any other compressed file containing your code. Make it sure you submit all your .java files not your .class files.

If you submit your program more than once we will take the last program submitted as the final version, and will deduct marks accordingly if it is late.