Project 5 (Kd-trees) Checklist

Prologue

Project goal: create a symbol table data type whose keys are two-dimensional points

Files:

- ⇝ `project5.pdf` (project description)
- ⇝ `project5.zip` (starter files for the exercises/problems, `report.txt` file for the project report, and test data files)

Exercises

Exercise 1. (*Array-based Symbol Table*) Develop a symbol-table implementation `ArrayST` that uses an (unordered) array as the underlying data structure to implement the basic symbol-table API.

```
>_ ~/workspace/project5

$ java edu.umb.cs210.p5.ArrayST Pluto
Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
<ctrl-d>
Mercury 1
Venus 2
Earth 3
Mars 4
Jupiter 5
Saturn 6
Uranus 7
Neptune 8
```

Exercises

ArrayST.java

```java
package edu.umb.cs210.p5;

import dsa.LinkedQueue;
import stdlib.StdIn;
import stdlib.StdOut;

public class ArrayST<Key, Value> {
    private static final int INIT_CAPACITY = 2;
    private Key[] keys;
    private Value[] values;
    private int N;

    // Create a symbol table with INIT_CAPACITY.
    public ArrayST() {
        ...
    }

    // Create a symbol table with given capacity.
    public ArrayST(int capacity) {
        ...
    }

    // Return the number of key-value pairs in the table.
    public int size() {
        ...
    }

    // Return true if the table is empty and false otherwise.
    public boolean isEmpty() {
        ...
    }

    // Return true if the table contains key and false otherwise.
    public boolean contains(Key key) {
        ...
    }
```

Exercises

ArrayST.java

```
    // Return the value associated with key, or null.
    public Value get(Key key) {
        ...
    }

    // Put the key-value pair into the table; remove key from table
    // if value is null.
    public void put(Key key, Value value) {
        ...
    }

    // Remove key (and its value) from table.
    public void delete(Key key) {
        ...
    }

    // Return all the keys in the table.
    public Iterable<Key> keys()  {
        ...
    }

    // Resize the internal arrays to capacity.
    private void resize(int capacity) {
        Key[] tempk = (Key[]) new Object[capacity];
        Value[] tempv = (Value[]) new Object[capacity];
        for (int i = 0; i < N; i++) {
            tempk[i] = keys[i];
            tempv[i] = values[i];
        }
        values = tempv;
        keys = tempk;
    }

    // Test client. [DO NOT EDIT]
    public static void main(String[] args) {
```

Exercises

ArrayST.java

```
        ArrayST<String, Integer> st = new ArrayST<String, Integer>();
        int count = 0;
        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            st.put(s, ++count);
        }
        for (String s : args) {
            st.delete(s);
        }
        for (String s : st.keys()) {
            StdOut.println(s + " " + st.get(s));
        }
    }
}
```

Exercises

Exercise 2. (*Spell Checker*) Write an `ArrayST` client called `Spell` that takes a command-line argument specifying the name of the file containing common misspellings (a line-oriented file with each comma-separated line containing a misspelled word and the correct spelling), then reads text from standard input and prints out the misspelled words in the text along with the line numbers where they occurred and their correct spellings.

```
>_ ~/workspace/project5

$ java edu.umb.cs210.p5.Spell data/misspellings.txt < data/war_and_peace.txt
wont:5370 -> won't
unconciousness:16122 -> unconsciousness
accidently:18948 -> accidentally
leaded:21907 -> led
wont:22062 -> won't
aquaintance:30601 -> acquaintance
wont:39087 -> won't
wont:50591 -> won't
planed:53591 -> planned
wont:53960 -> won't
Ukranian:58064 -> Ukrainian
wont:59650 -> won't
conciousness:59835 -> consciousness
occuring:59928 -> occurring
```

Exercises

Spell.java

```
package edu.umb.cs210.p5;

import stdlib.In;
import stdlib.StdIn;
import stdlib.StdOut;

public class Spell {
    public static void main(String[] args) {
        In in = new In(args[0]);
        String[] lines = in.readAllLines();
        in.close();

        // Create an RefArrayST<String, String> object called st.
        ...

        // For each line in lines, split it into two tokens using
        // "," as delimiter; insert into st the key-value pair
        // (token 1, token 2).
        ...

        // Read from standard input one line at a time; increment
        // a line number counter; split the line into words using
        // "\\b" as the delimiter; for each word in words, if it
        // exists in st, write the (misspelled) word, its line number, and
        // corresponding value (correct spelling) from st.
        ...
    }
}
```

Problems

The guidelines for the project problems that follow will be of help only if you have read the description of the project and have a general understanding of the problems involved. It is assumed that you have done the reading.

Problems

Java interface `PointST<Value>` specifying the API for a symbol table data type whose keys are two-dimensional points represented as `Point2D` objects

Method	Description
`boolean isEmpty()`	is the symbol table empty?
`int size()`	number of points in the symbol table
`void put(Point2D p, Value val)`	associate the value *val* with point *p*
`Value get(Point2D p)`	value associated with point *p*
`boolean contains(Point2D p)`	does the symbol table contain the point *p*?
`Iterable<Point2D> points()`	all points in the symbol table
`Iterable<Point2D> range(RectHV rect)`	all points in the symbol table that are inside the rectangle *rect*
`Point2D nearest(Point2D p)`	a nearest neighbor to point *p*; `null` if the symbol table is empty
`Iterable<Point2D> nearest(Point2D p, int k)`	*k* points that are closest to point *p*

Problems

Problem 1. (*Brute-force Implementation*)

Hints:

- ⇝ Instance variable
  - ⇝ A binary search tree to store the key/value pairs, `RedBlackBST<Point2D, Value> bst`
- ⇝ `BrutePointST()`
  - ⇝ Initialize instance variable appropriately
- ⇝ `boolean isEmpty()`
  - ⇝ Return `true` if the symbol table is empty, and `false` otherwise
- ⇝ `int size()`
  - ⇝ Return the number of key/value pairs in the symbol table
- ⇝ `void put(Point2D p, Value val)`
  - ⇝ Insert the given key/value pair into the symbol table
- ⇝ `Value get(Point2D p)`
  - ⇝ Return the value corresponding to the given key, or `null`

Problems

- `boolean contains(Point2D p)`
  - Return `true` if the given key is in the symbol table, and `false` otherwise
- `Iterable<Point2D> points()`
  - Return an iterable object containing all the keys in the symbol table
- `Iterable<Point2D> range(RectHV rect)`
  - Return an iterable object containing all the keys (ie, points) in the symbol table that are contained inside the given rectangle
- `Point2D nearest(Point2D p)`
  - Return a key (ie, point) from the symbol table that is closest to (and different from) the given key
- `Iterable<Point2D> nearest(Point2D p, int k)`
  - Return an iterable object containing upto `k` keys (ie, points) from the symbol table that are closest to (and different from) the given key

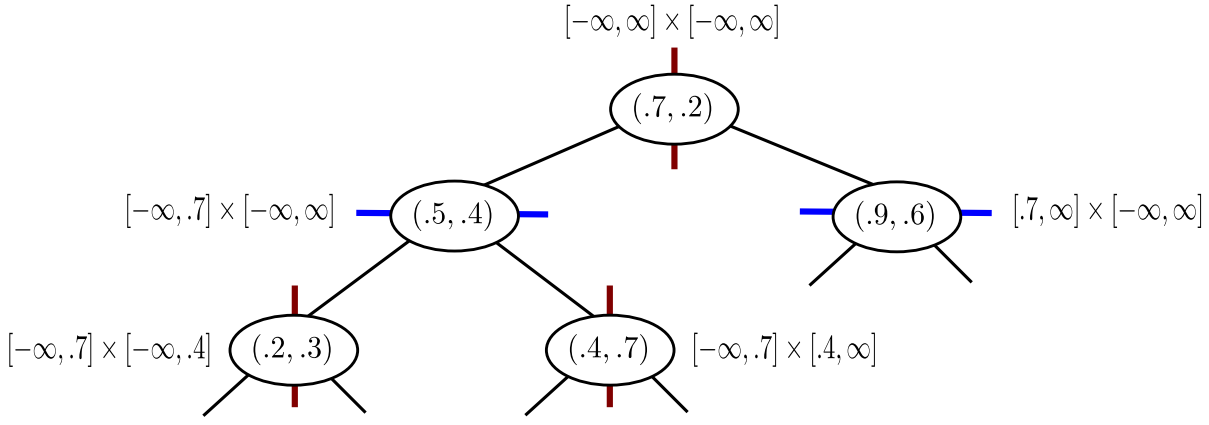
Problem 2. (*2d-tree Implementation*)

Hints

- Instance variables
  - Reference to the root of the 2d-tree, `Node root`
  - Number of nodes (ie, key/value pairs) in the tree, `int N`
- `KdTreePointST()`
  - Initialize instance variables appropriately
- `boolean isEmpty()`
  - Return `true` if the symbol table is empty, and `false` otherwise
- `int size()`
  - Return the number of key/value pairs in the symbol table

Problems

$\rightsquigarrow$ Axis-aligned rectangles for the running example in the assignment writeup



Exercise: insert the points $(.8, .4)$ and $(.8, .7)$ into the above BST and calculate the corresponding axis-aligned rectangles.

Problems

- `void put(Point2D p, Value val)`
  - Call the private helper method `put()` with appropriate arguments to insert the key/value pair into the 2d tree; the parameter `lr` in this and other helper methods represents if the currrent node is x-aligned (`lr = true`) or y-aligned (`lr = false`)
- `Node put(Node x, Point2D p, Value val, RectHV rect, boolean lr)`
  - If `x = null`, return a new `Node` object built appropriately
  - If the key in `x` is the same as the given key, update the value in `x` to the given value
  - Make a recursive call to `put()` with appropriate arguments to insert the given key/value pair into the left subtree `x.lb` or the right subtree `x.rt` depending on how the values of `x.x`/`x.y` and `p.x`/`p.y` compare (use `lr` to decide which alignment to consider)
  - Return `x`

Problems

- `Value get(Point2D p)`
  - Call the private helper method `get()` with appropriate arguments to return the value corresponding to the given key, or `null`
- `Value get(Node x, Point2D p, boolean lr)`
  - If `x = null`, return `null`
  - If the key in `x` is the same as the given key, return the value in `x`
  - Make a recursive call to `get()` with appropriate arguments to find and return the value corresponding to the given key in the left subtree `x.lb` or the right subtree `x.rt` depending on how the values of `x.x`/`x.y` and `p.x`/`p.y` compare

Problems

- `boolean contains(Point2D p)`
  - Return `true` if the given key is in the symbol table, and `false` otherwise
- `Iterable<Point2D> points()`
  - Return an iterable object containing all the keys in the symbol table, enumerated in level order using a queue
- `Iterable<Point2D> range(RectHV rect)`
  - Call the private helper method `range()` passing it an empty queue of `Point2D` objects as one of the arguments, and return the queue
- `void range(Node x, RectHV rect, Queue<Point2D> q)`
  - If `x = null`, simply return
  - If `rect` contains the key in `x`, enqueue the key into `q`
  - Make recursive calls to `range()` on the left subtree `x.lb` and on the right subtree `x.rt`
  - Incorporate the *range search* pruning rule mentioned in the assignment writeup

Problems

- `Point2D nearest(Point2D p)`
  - Return the key (ie, point) closest to (and different from) the given key by making a call to the private helper method `nearest()` with appropriate arguments
- `Point2D nearest(Node x, Point2D p, Point2D nearest, double nearestDistance, boolean lr)`
  - If `x = null`, return `nearest`
  - If the key in `x` is different from the given key and the squared distance between the two is smaller than the `nearestDistance`, update `nearest` and `nearestDistance` appropriately
  - Make a recursive call to `nearest()` on the left subtree `x.lb`
  - Make a recursive call to `nearest()` on the right subtree `x.rt`, using the value returned by the first call in an appropriate manner
  - Incorporate the *nearest neighbor* pruning rules mentioned in the assignment writeup

Problems

- `Iterable<Point2D> nearest(Point2D p, int k)`
  - Call the private helper method `nearest()` passing it an empty max-pq (consisting of `Point2D` objects and built with a suitable comparator from `Point2D`) as one of the arguments, and return the pq
- `void nearest(Node x, Point2D p, int k, MaxPQ<Point2D> pq, boolean lr)`
  - If `x = null` or if the size of pq is greater than `k`, simply return
  - If the key in `x` is different from the given key, insert it into pq
  - If the size of pq exceeds `k`, remove the maximum key from the pq
  - Make recursive calls to `nearest()` on the left subtree `x.lb` and on the right subtree `x.rt`
  - Incorporate the *nearest neighbor* pruning rules

Problems

The `data` directory contains a number of sample input files, each with N points (x, y), where $x, y \in (0, 1)$; for example

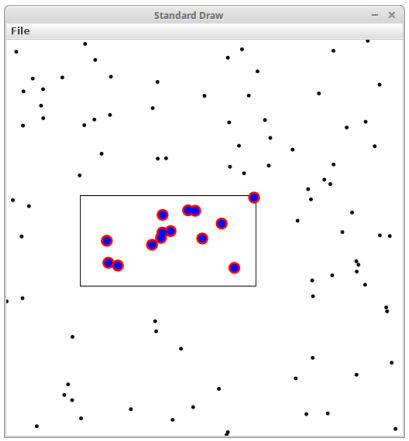
```
>_ ~/workspace/project5

$ cat data/input100.txt
0.042191 0.783317
0.390296 0.499816
0.666260 0.752352
...
0.772950 0.196867
```

The visualization client `RangeSearchVisualizer` reads a sequence of points from a file (specified as a command-line argument), inserts those points into `BrutePointST` (red) and `KdTreePointST` (blue) based symbol tables, performs range searches on the axis-aligned rectangles dragged by the user, and displays the points obtained from the symbol tables in red and blue

```
>_ ~/workspace/project5

$ java edu.umb.cs210.p5.RangeSearchVisualizer data/input100.txt
```

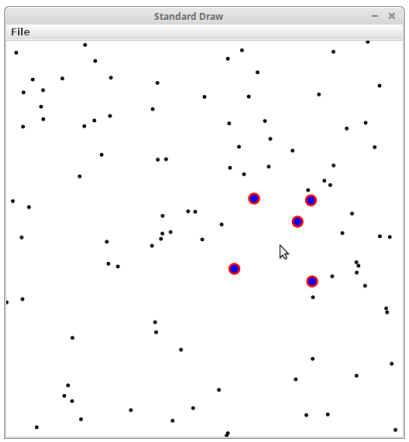


Problems

The visualization client `NearestNeighborVisualizer` reads a sequence of points from a file (specified as a command-line argument), inserts those points into `BrutePointST` (red) and `KdTreeSPointT` (blue) based symbol tables, performs k- (specified as the second command-line argument) nearest neighbor queries on the point corresponding to the location of the mouse, and displays the neighbors obtained from the symbol tables in red and blue

```
>_ ~/workspace/project5
```

```
$ java edu.umb.cs210.p5.NearestNeighborVisualizer data/input100.txt 5
```

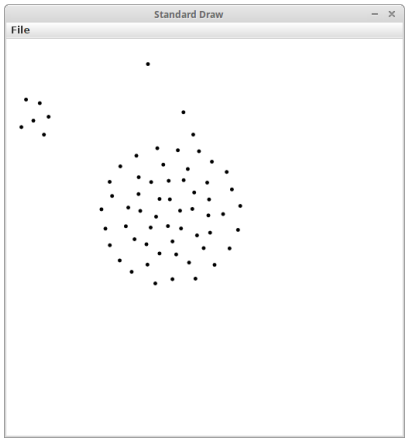


Problems

The visualization client `BoidSimulator` simulates the flocking behavior of birds, using a `BrutePointST` or `KdTreePointST` data type; the first command-line argument specifies which data type to use (`brute` or `kdtree`), the second argument specifies the number of boids, and the third argument specifies the number of friends each boid has

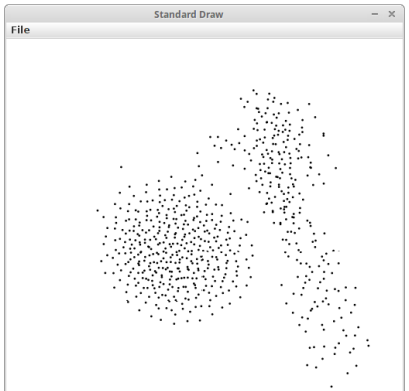
```
>_ ~/workspace/project5

$ java edu.umb.cs210.p5.BoidSimulator brute 100 10
```




```
>_ ~/workspace/project5

$ java edu.umb.cs210.p5.BoidSimulator kdtree 1000 10
```



Epilogue

Use the template file `report.txt` to write your report for the project

Your report must include:

- ⇝ Time (in hours) spent on the project
- ⇝ Difficulty level (1: very easy; 5: very difficult) of the project
- ⇝ A short description of how you approached each problem, issues you encountered, and how you resolved those issues
- ⇝ Acknowledgement of any help you received
- ⇝ Other comments (what you learned from the project, whether or not you enjoyed working on it, etc.)

Epilogue

Before you submit your files:

- ⇝ Make sure your programs meet the style requirements by running the following command on the terminal

```
>_ ~/workspace/project5
$ check_style <program>
```

  where `<program>` is the fully-qualified name of the program

- ⇝ Make sure your code is adequately commented, is not sloppy, and meets any project-specific requirements, such as corner cases and running time
- ⇝ Make sure your report uses the given template, isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling mistakes

Epilogue

Files to submit:

1. `ArrayST.java`
2. `Spell.java`
3. `BrutePointST.java`
4. `KdTreePointST.java`
5. `report.txt`