CMSC 420: Spring 2022

Programming Assignment 2: Height-Balanced kd-Trees

Handed out: Tue, Apr 19. Due: **Tue, Apr 26, 11:59pm**. (Submission via Gradescope.)

Overview: In this assignment you will implement a variant of the kd-tree data structure, called a *height-balanced kd-tree* (or `HBkdTree`) to store a set of points in 2-dimensional space. It will support insertion, deletion, and a few other queries.

This data structure borrows ideas from AVL trees, scapegoat trees, and classical point kd-trees. When constructed, a positive integer parameter `maxHeightDifference` is given. Whenever an internal node's two subtrees have heights that differ by more than this value, the subtree rooted at this node is rebuilt[1] into a perfectly balanced tree (in the manner of scapegoat trees).

The data structure is generic and is templated with the point type, which call a *labeled point*. This encapsulates the concept of a 2-dimensional point that is associated with a string, called its *label*. This may be any class that implements the Java interface (which we will provide) called `LabeledPoint2D`. Such an object is a 2-dimensional point, represented by its (x, y)-coordinates and an associated string *label*.

```
public interface LabeledPoint2D {
    public float getX(); // get point's x-coordinate
    public float getY(); // get point's y-coordinate
    public float get(int i); // get point's i-th coordinate (0=x, 1=y)
    public Point2D getPoint2D(); // get the point itself
    public String getLabel(); // get the label
}
```

The `Point2D` object is an enhanced version of the Java built-in `Point2D` object, which we will provide to you.

In our case, the labeled points represent airports, where the (x, y) coordinates are the airports location (think latitude and longitude) and the labels are the 3-letter airport codes (e.g., "`BWI`" for Baltimore-Washington Airport). The individual coordinates (which are `doubles`) can be extracted directly using the functions `getX()` and `getY()`, or `get(i)`, where $i = 0$ for x and $i = 1$ for y.

Your wrapped kd-tree will be templated with one type, which we will call `LPoint` (for "labeled point"). For example, your file `HBkdTree` will contain the following public class:

```
public class HBkdTree<LPoint extends LabeledPoint2D> { ... }
```

Height-Balanced kd-Tree: Recall that a point kd-tree is a data structure based on a hierarchical decomposition of space through the use of axis-orthogonal splits. A *height-balanced kd-tree* imposes the additional requirement that for every internal node, the heights of its two subtrees

[1] You might wonder why we don't just apply rotations as we did with AVL trees. The issue is that rotations are a one-dimensional operation and they do not make sense in the context of multidimensional structures like kd-trees.

can differ by at most a user-specified integer parameter `maxHeightDifference`, which is at least 1. The insertion and deletion processes are exactly the same as given in the lecture on kd-trees (see the latex lecture notes for Lecture 13, which has all the details spelled out), but when inserting a new point, the cutting dimension is selected based on the shape of the current cell. We select the cutting dimension so that we split the longer side of the current cell. More formally, if its width (along x) is greater than or equal to its height (along y) the cutting dimension is 0 (x or vertical) and otherwise it is 1 (y or horizontal). Note that ties are broken in favor of vertical cuts.

For example, consider the insertion of `ATL` in Fig. 1. When we fall out of the tree (along the left child link from `ORD`), the cell associated with this null pointer is the rectangle whose lower-left corner is $(0, 4)$ and whose upper-right corner is $(2, 8)$.) Since this rectangle is taller than wide, we cut horizontally, thus setting the cutting dimension of the new node to 1.

After a point has been inserted into or deleted from the tree, we walk backwards upward along the search path (exactly has we would do if this were an AVL), updating the heights as we go. Whenever we reach a node `p` where the heights of its two subtrees differ by more than `maxHeightDifference`, we completely rebuild the subtree rooted at `p`. We first traverse the subtree rooted at `p` and store all the labeled points of this subtree in a list (e.g., a Java `ArrayList`). Given this list, the subtree is rebuilt by the following recursive process (see Fig. 1):

Basis: If the list is empty, return `null`. Otherwise, continue with the following steps.

Cutting Dimension: Let `cell` denote the cell associated with the current node. As with insertion, we select the cutting dimension so that it splits the longer side of `cell`.

Sort: Sort the points according to the cutting dimension. If the cutting dimension is x, sort the points in increasing order first by x and break ties by sorting in increasing order y. If the cutting dimension is y, then sort first by y with ties broken by x.

Split and Recurse: Letting k denote the size of the list (and assuming as usual that entries are indexed from 0 to $k-1$), define the median element to be point at index $m \leftarrow \lfloor k/2 \rfloor$. Recursively build a balanced tree on the left-side sublist of entries with indices 0 through $m-1$, and recursively build a balanced tree on the right-side sublist of entries with indices $m+1$ through $k-1$. Join these two subtrees under a node whose point is the median point, and whose cutting dimension is as chosen above. Return this tree.

Unlike scapegoat trees (where each operation can trigger at most one rebuild), we continue all the way up to the root, updating the heights as we go and checking the height difference condition. This may trigger further rebuilds. (See Fig. 1 for an example.)

Requirements: Your program will implement the following functions for the `HBkdTree`. While you can implement the data structure internally however you like (subject to the style and efficiency requirements given below), the following function signatures should not be altered. As part of the skeleton code, we will provide you with the `LabeledPoint2D` interface, and two useful classes, `Point2D` and `Rectangle2D`. (If you wish to modify these objects, do not alter them. Instead, create your own copy, say `MyPoint2D`, and make modifications there.)

`HBkdTree(int maxHeightDifference, Rectangle2D bbox)`: This constructs a new `HBkdTree` with the given max height difference and the given axis aligned bounding box.

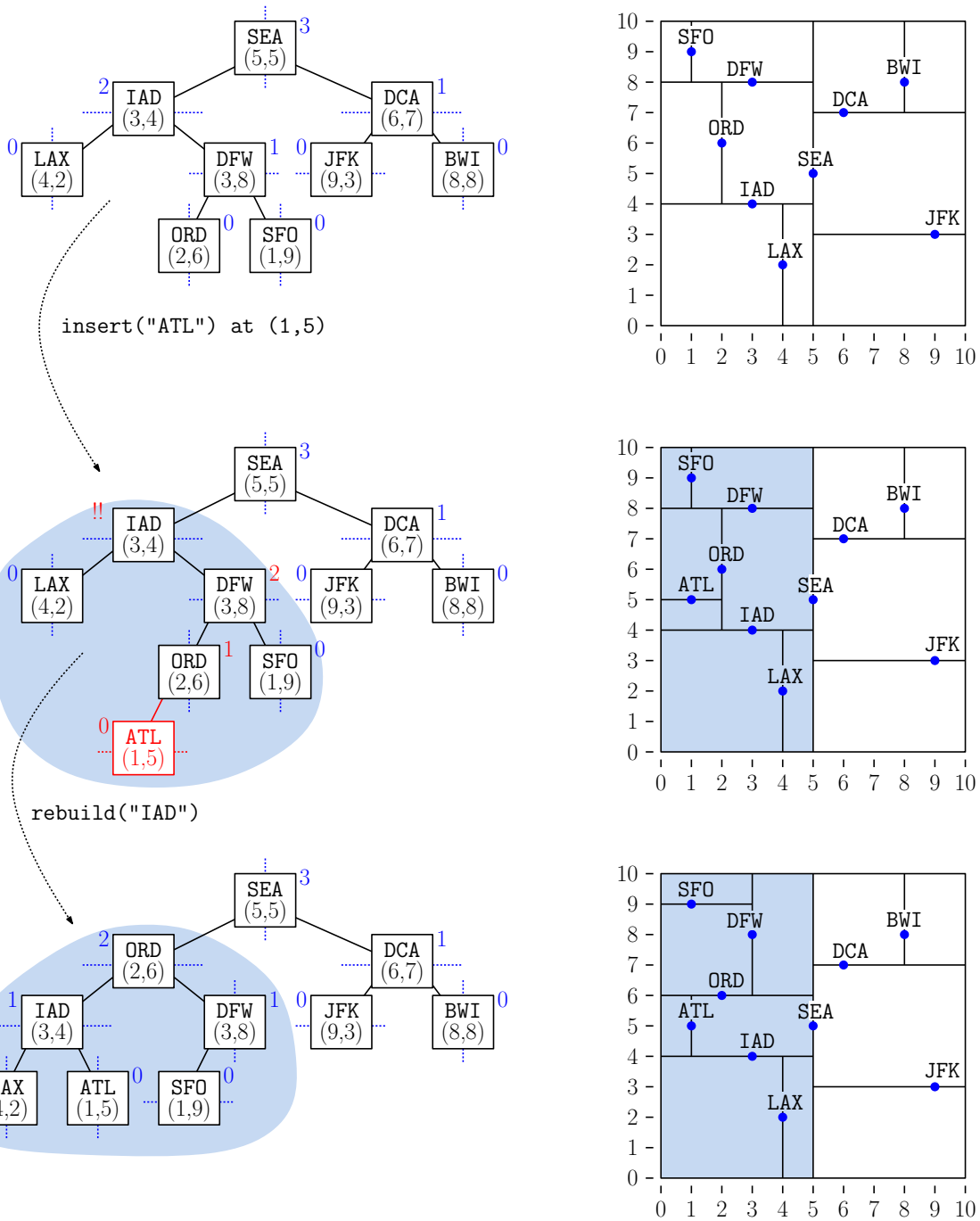

Figure 1: Let `maxHeightDifference = 1`. Suppose we insert `ATL` at $(1, 5)$. This is inserted as the left child of `ORD`. On returning from the recursive calls, we update the node heights at `ORD`, `DFW`, and `IAD`. At `IAD`,the absolute height difference in our left and right subtrees is $2 - 0 = 2$, which exceeds `maxHeightDifference`. We rebuild this entire subtree highlighted in blue. Since the cell (shaded in blue) is taller than wide, we cut horizontally. We sort along y (yielding $\langle$`LAX`, `IAD`, `ATL`, `ORD`, `DFW`, `SFO`$\rangle$) and split about the median `ORD`. We recursively build the other subtrees similarly. We continue back up the root, updating heights, but no further rebuilds are needed.

`LPoint find(Point2D pt)`: Given an (unlabeled) point, determine whether it exists within the tree, and if so return the associated labeled point. Otherwise, return `null`.

`void insert(LPoint pt)`: Inserts point given labeled point in the tree (and performs rebuilding if necessary, as described above). If the point lies outside the bounding box, throw an `Exception` with the error message `"Attempt to insert a point outside bounding box"`. If a point with the same coordinates (and possibly different label) exists in the tree, throw an `Exception` with the message `"Attempt to insert a duplicate point"`. Otherwise, apply the insertion and rebuilding process described above.

`void delete(Point2D pt) throws Exception`: Given an (unlabeled) point, this deletes the point of the tree having the same coordinates (and performs rebuilding if necessary, as described above). If there is no such point, it throws an `Exception` with the error message `"Attempt to delete a nonexistent point"`. The deletion process is the same as described in the Lecture 13 notes. (In particular, the process by which the replacement nodes are selected is the same as given in the lecture notes.)

Update (4/20): In the utility function `findMin`, which is used to find the replacement node, if there are ties for the point with the smallest ith coordinate, break the ties by taking the point with the smallest other coordinate (that is, coordinate $1 - i$).

Update (4/20): The one change is that on returning up along the search path node heights are to be updated, and whenever a node is found to fail the height difference criteria, its subtree is rebuilt. Generally, a deletion may result in multiple replacements. The balance condition testing and rebuilding is applied only *after* the standard deletion process is completely finished. Balance checking commences at the final leaf node whose deletion terminates the standard kd-tree deletion process. This is very easy to code. Simply code the deletion as given in the lecture notes, but just prior to returning from the deletion helper (that is, just prior to the line "`return p`" from the lecture notes), update the current node's height, check the height difference, and trigger rebuilding if needed. If the tree is rebuilt, return a pointer to the newly rebuilt tree.

Update (4/20): Note, by the way that due to replacement, many cells in the tree can change shape. (In particular, these are the cells that are incident on the splitting line through the deleting point.) As a result, some cells that were taller have switched to being wider and vice versa. However, you should not alter the cutting dimension for any of the nodes. Once the cutting dimension of a node has been set, it should remain unchanged until the node is deleted or it has been part of a rebuilding. (As a consequence of this, you should *not* store each node's cell as a member of the kd-tree node, since otherwise updating these cells will take too much time. Instead, you should compute cells on the fly as you traverse the tree.)

`ArrayList<String> getPreorderList()`: This operation generates a preorder enumeration of the nodes in the tree. This is represented as a Java `ArrayList` of type `String`, with one entry per node. You will probably implement this by writing a recursive helper function that starts at the root. When it visits a node `p`, it does the following. If `p == null`, then generate the string `"[]"` and return. Otherwise, generate the following string and recursively invoke the procedure on the left and right children. Depending on whether the cutting dimension is x or y, this generates either:

```
"(x=" + cutVal + " ht=" + height + ")␣" + point.toString()
```

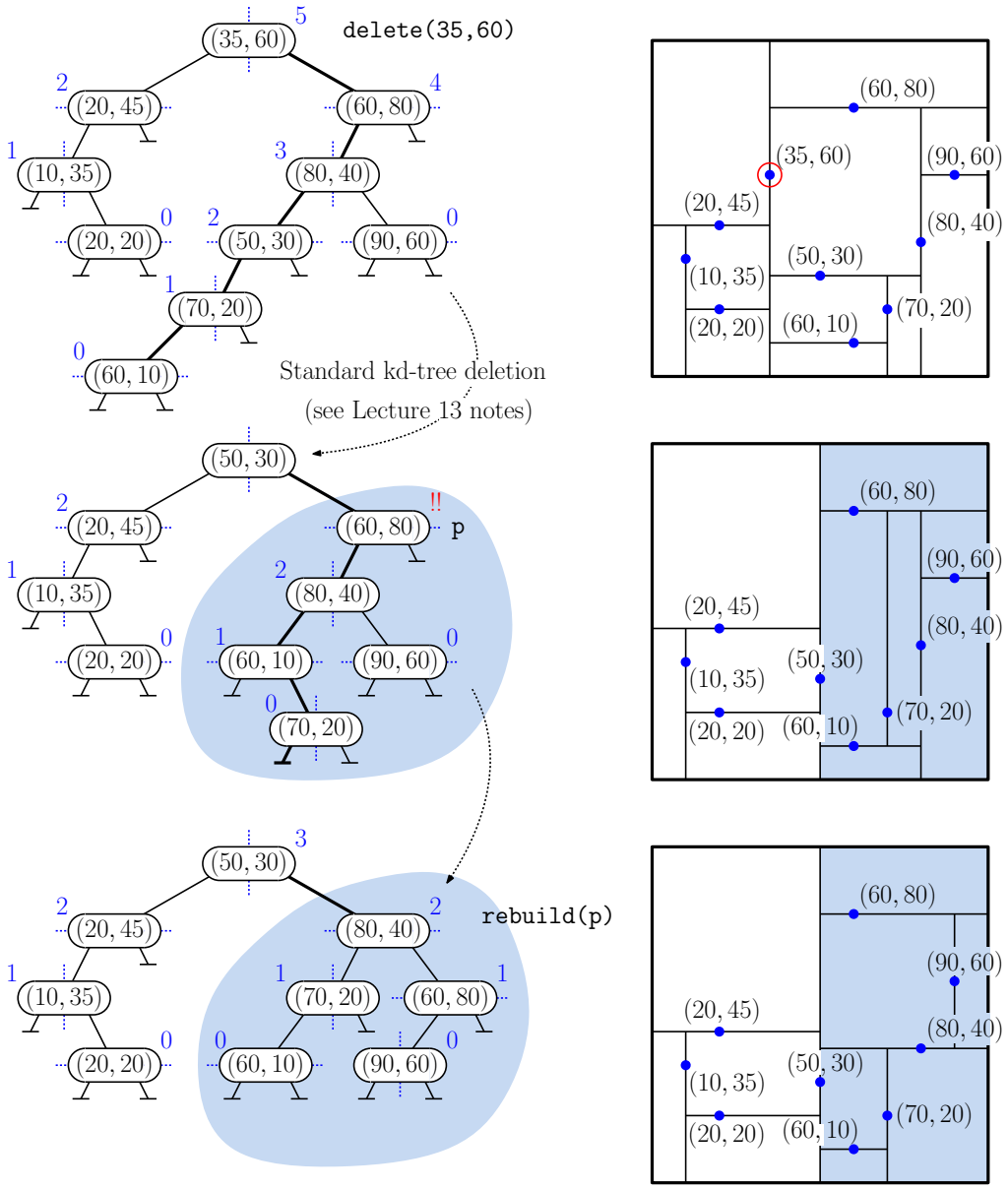

Figure 2: Let `maxHeightDifference = 1`. (Note that this tree is not valid, since it fails the height-difference condition at many nodes. We have chosen it to match the example from Lecture 13.) Suppose we delete the point `(35,60)`. We first perform the standard kd-tree deletion as described in Lecture 13. Note that when the replacement point `(50,30)` is copied to the root, the root's cutting dimension does not change. At the end of the process, on returning from the recursive calls, we update the node heights at `(70,20)` (now 0), `(60,10)` (now 1), and `(80,40)` (now 2). At `(60,80)`, the absolute height difference in the left and right subtrees is $2 - (-1) = 3$, which exceeds `maxHeightDifference`. We rebuild this entire subtree highlighted in blue. Since the cell (shaded in blue) is taller than wide, we cut horizontally. We sort along y (yielding $\langle$`(60,10)`, `(70,20)`, `(80,40)`, `(90,60)`, `(60,80)`$\rangle$) and split about the median `(80,40)`. We recursively build the other subtrees similarly. We continue back up the root, updating heights, but no further rebuilds are needed.

```
"(y=" + cutVal + " ht=" + height + ")␣" + point.toString()
```

where `cutVal` is the cutting value for this node (that is, the coordinate of the node's point associated with the cutting dimension), `height` is the height of the subtree rooted at this node, and `point.toString()` invokes the `toString()` method for the point stored in this node. (This function will be provided to you as part of our skeleton code.)

Here is example of what this would look like for the tree at the top of Fig. 1.

```
(x=5.0 ht=3) SEA: (5.0,5.0)
(y=4.0 ht=2) IAD: (3.0,4.0)
(x=4.0 ht=0) LAX: (4.0,2.0)
[]
[]
(y=8.0 ht=1) DFW: (3.0,8.0)
(x=2.0 ht=0) ORD: (2.0,6.0)
[]
[]
(x=1.0 ht=0) SFO: (1.0,9.0)
[]
[]
(y=7.0 ht=1) DCA: (6.0,7.0)
(y=3.0 ht=0) JFK: (9.0,3.0)
[]
[]
(x=8.0 ht=0) BWI: (8.0,8.0)
[]
[]
```

Note that our autograder is sensitive to both case and whitespace.

`ArrayList<LPoint> orthogRangeReport(Rectangle2D query)`: This function performs an orthogonal range reporting query. It is given an axis-aligned rectangle `query` and it returns a Java `ArrayList` containing the points lying within this rectangle. (You may find it useful to use the function from class `Rectangle2D`, such as `contains`, `leftPart`, and `rightPart`.) The order in which elements appear in the final list does not matter. We will sort the list before outputting it.

`void clear()`: This removes all the entries of the tree.

`int size()`: Returns the number of points in the tree. For example, for the tree at the top of Fig. 1, this would return 9.

`void setHeightDifference(int newDiff)`: `Update (4/19)`: You no longer need to implement this operation.

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You should replace the `HBkdTree.java` file with your own, and you should add the implementation of the above functions to `HBkdTree.java`. You should not modify any of the other files, but you can add new files of your own.

As mentioned above, you should not modify `Point2D` or `Rectangle2D` (since our testing functions use these), but you can create copies and make modifications to these copies if you like.

You must use the package "`cmsc420_s22`" for all your source files. (This is required for the autgrader to work.) As usual, we will provide a driver program (`Tester.java` and `CommandHandler.java`) that will input a set of commands. Here is a portion of the class's public interface (and of course, you will add all the private data and helper functions). You should not modify the signature of the public functions, but you are free to set up the internal structure however you like.

```
package cmsc420_s22;

import java.util.ArrayList;

public class HBkdTree<LPoint extends LabeledPoint2D> {

    public HBkdTree(int maxHeightDifference, Rectangle2D bbox) { /* ... */ }
    public LPoint find(Point2D pt) { /* ... */ return null; }
    public void insert(LPoint pt) throws Exception { /* ... */ }
    public void delete(Point2D pt) throws Exception { /* ... */ }
    // ... and so on
}
```

Efficiency requirements: `Update (4/19):` Excluding the time for rebuilding, the operations `find`, `insert`, and `delete` must run in time proportional to the tree height. (Because of rebalancing, the tree height will be $O(\log n)$.) The operation `orthogRangeReport` should be efficient in the sense that it does not waste time making recursive calls into subtrees whose cell does not overlap the query range. For this reason, the helper function for this operation will need to test the node's cell against the query range.

The operation `size` should run in constant time. (This is best handled by maintaining a separate counter that keeps track of the number of points currrently in the structure.)

Testing/Grading: `Update (4/19):` We will use the standard Gradescope-based grading process that we have used in previous assignments.