The SG kd-Tree Data Structure

Motivation: For this part of the programming assignment, we will implement a novel data structure for dynamically maintaining a kd-tree. Geometric structures like kd-trees are not as easy to rebalance as are binary search trees because the the ubiquitous rotation operation cannot be applied to multidimensional partition trees, like the kd-tree. The alternative is to rebuild subtrees.

In geometric data structures, it is natural to employ an extended tree structure, where points are stored in the external nodes, and internal nodes store splitters, which partition space.

Thus, in order to design a dynamic data structure for geometric point sets, we will combine *subtree rebuilding* with an *extended tree structure*. This is exactly what the SG kd-tree is.

Overview: The SG kd-tree has features in common to both the extended version of the scapegoat tree (our SG tree) and kd-trees, Here are the elements that it shares in common with the SG tree:

- It is an *extended tree*, in which contents are stored only in the external nodes and splitters stored in internal nodes.
- It is rebalanced through the process of *rebuilding subtrees*. The same rules as the SG tree apply for when to trigger rebuilding, determining scapegoat nodes, and deciding how to partition objects among the left and right subtrees.
- If a point is equal to the splitter, it is stored in the *left subtree*. (Note that this differs from the kd-tree convention from class.)

The principal difference is how splitting is done. Our approach will be different than a classical kd-tree. We will *not* use a simple horizontal or vertical line to split the points, and we will *not* alternate the splitting dimension between levels of the tree.

Suppose that we wish to build a balanced kd-tree from a list of k points. As with the SG tree, we will employ a recursive approach. If $k = 1$, then we simply generate a single external node containing this point and return a reference to it. Otherwise, we will partition the array based into two roughly equal halves based on the splitting dimension. We will build subtrees for each side of the partition, and then join them under a common internal node. But how is the splitting dimension chosen and how are points partitioned?

How to Choose the Splitting Dimension? This is based on the *distribution* of the point set. Consider a minimum enclosing axis-parallel rectangle `r` for the point set. (This is not to be confused with a *cell* associated with a node of the kd-tree nor the *bounding box* of the entire kd-tree).

Let `r.low` and `r.high` denote its lower left and upper right corner points, respectively (see Fig. 1). The splitting dimension is parallel to the longer side of `r`. More formally, define $\Delta_x =$ `r.high.x` $-$ `r.lo.x` and $\Delta_y =$ `r.high.y` $-$ `r.lo.y`. If $\Delta_x \geq \Delta_y$, we split along the x-coordinate (vertically), and otherwise we split along the y-coordinate (horizontally).

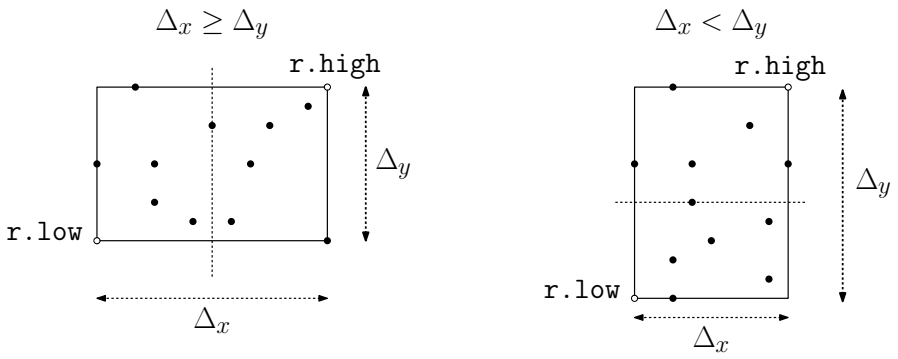

Figure 1: The splitting dimension is based on the shape of the minimum enclosing rectangle.

After the points are partitioned, this process is repeated recursively. Thus, the splitting dimension need not alternate between levels of the tree. It depends on the distribution of the points within each subtree as to how they will be split.

How to Partition the Points? In the SG tree, because there are no duplicate keys, we had perfect control of how keys were distributed into the left and right subtrees. This is a problem for kd-trees, however, since multiple points may share the same x- or y-coordinate as the splitting line. Recall that in the classical kd-tree (as given in class), all of the points lying on the splitting line are placed in the right subtree.

Unfortunately, we need finer control on the splitting process for our SG kd-trees. Our approach is based on two ideas:

- The splitter will be *point*, not a line.
- Points are ordered *lexicographically* with respect to this point. If the splitting dimension is x, the points are order lexicographically in the order (x, y). If the splitting dimension is y, the points are ordered lexicographically by (y, x). (For example, in Fig. 2(a) the splitting dimension is x. The points `c`, `d`, `e`, and `f` all share the same x-coordinate. But by making `d` the splitter, we can distinguish `c` and `d` as lying in the left subtree and `e` and `f` as lying in the right subtree.)

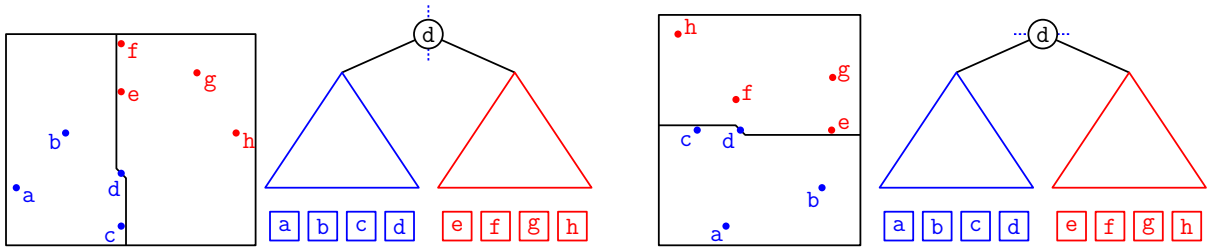

Figure 2: The splitter is a point (`d` in each case) and points are partitioned lexicographically depending on the splitting dimension. As with the SG tree, points that are less than or equal to the splitter are placed in the left subtree.

In summary, in addition to size and height information, each internal node of the SG kd-tree stores two pieces of information:

- A *splitting dimension*, which is either 0 for x or 1 for y.
- A *splitter point* $s = (s_x, s_y)$ in 2-dimensional space. If the splitting dimension is x, a point $p = (p_x, p_y)$ is placed in the left subtree if and only if $(p_x, p_y) \leq (s_x, s_y)$ lexicographically. If the splitting dimension is y, it is placed in the left subtree if and only if $(p_y, p_x) \leq (s_y, s_x)$ lexicographically.

Building a Balanced SG kd-Tree: We can now describe the algorithm for building a balanced tree formally. We'll express this in terms of Java's lists and sublists. Given a list (e.g., an `ArrayList`) `A` containing k points, if $k = 1$, we create a single external node containing this point.

Otherwise, we compute the minimum enclosing rectangle r for the points (by computing the minimum and maximum coordinates in each dimension) and select the splitting dimension to be parallel to the longer of the two sides (with ties broken in favor of x or vertical).

Next, we sort the list appropriate to the splitting dimension. In Java, this can be done by defining two `Comparator` objects: `CompareXY` sorts lexicographically on (x, y) and `CompareYX` sorts lexicographically on (y, x). Then you can invoke the `Collections.sort(A, compareXY)` or `Collections.sort(A, compareYX)` depending on the splitting dimension. (There are slightly more efficient ways to extract the sorted order from the existing kd-tree tree, but we will leave this as an exercise.)

After the list is sorted, let $m = \lceil k/2 \rceil$. Save the splitter as `A.get(m-1)`. Then apply the algorithm recursively to the sublists, `A.subList(0, m)` (that is, $A[0...m-1]$) and `A.subList(m, k)` (that is, $A[m...k-1]$) to obtain the left and right subtrees. Finally, combine these two subtrees under a common internal node that contains the splitting dimension and splitter.

(Hint: Above, I suggested that you save the splitter *before* you invoke the function recursively on the children. Can you see why?)

Dictionary Operations: Now that you know how to rebuild an SG kd-tree, you know almost everything that you need in order to rebuild and rebalance the tree. However, we still need to know how to insert, delete, and find individual points.

If you designed your SG tree code well, the answer is remarkably simple and elegant. In the SG tree, whenever we visited an internal node, we compared the point lexicographically to determine whether to continue in the left subtree or right subtree. The only difference now is to invoke the appropriate comparator (`CompareXY` or `CompareYX`) depending on the splitting dimension for this node. Otherwise, the code remains essentially unchanged.

For completeness, let us review each of the operations individually.

Find: To find a point $p = (p_x, p_y)$ in the SG kd-tree, we descend the tree as in any standard binary tree search. The difference is that when we arrive at an internal node that has an x-splitter s, we use the `CompareXY` comparator to determine whether $(p_x, p_y) \leq (s_x, s_y)$, and descend to the appropriate child. If this is a y-splitter, we use the `CompareYX` comparator to determine whether $(p_y, p_x) \leq (s_y, s_x)$. When we arrive at an external

node, we check whether we are equal to this point. If so, we have found it, and otherwise we have not.

Insert: To insert a new point p into the tree, we employ the above searching process until arriving at an external node. Let p' be the point in this node. (Let's assume that we have already checked for duplicates.) Recall that the insertion process involves creating a new external node for p, and then linking these two external nodes under a common internal node.

At first, the process seems to be quite complicated compared to the 1-dimensional case. (We now need to consider whether this internal node should be an x- or y-splitter, and this depends on the shape of the minimum bounding rectangle containing p and p'.) The good news, is that there is a very elegant solution. We simply create a 2-element list containing p and p' and then invoke the aforementioned function for building a balanced tree from this 2-element list. This will return the desired 3-node structure, which we can just link into the tree by returning a reference to the root (internal node).

Delete: The deletion code for kd-trees was amazingly complicated, but that was because of the need to find replacement nodes. The good news here is that all deletions occur at external nodes. No replacement points need be computed!

To delete a city of a given name, we first search the binary search tree to find its coordinates. We then invoke the above find function to find the external node containing the point. As in the SG tree, we simply unlink this node and its parent from the tree.

You might notice that the insert and delete functions will generally modify the structure of the minimum enclosing rectangle for the points of a subtree. However (as with the SG tree) we do not modify the contents of the internal nodes unless a rebuilding operation has been triggered.

Nearest Neighbor Searching: The nearest-neighbor searching algorithm as presented in class can be adapted very easily to this setting. In fact, the code is actually a bit shorter here. In the classical kd-tree, each node contains a point. Splitters, however, should not be considered as elements of the tree. (The reason is that when we delete points, we do not alter the splitters.)

What this means is that the two lines of the function `nearNeighbor` function of Lecture 14 that compute the distance of the query point `q` to `p.point` can be omitted for internal nodes. (These lines are invoked for external nodes, however.)

The other difference is structural. The static function given in the lecture notes now needs to become a member function of internal and external node classes. For example, rather than making the recursive call as `nearNeighbor(q, p.left, leftCell, best)` as we did in Lecture 14, we would now instead call `left.nearNeighbor(q, leftCell, best)`. Otherwise, the code is pretty much the same as in the lecture notes.

Example 1: Let us begin with a simple example from test file `mytest-input-1.xml`.

```
<createCity "Edinburgh" x="500" y="500" .../>
<createCity "Lisbon" x="750" y="600" .../>
<createCity "Varna" x="400" y="800" .../>
<createCity "London" x="300" y="400" .../>
```

When `Edinburgh` is inserted, the tree consists of a single external node. When `Lisbon` is added, we determine that the bounding rectangle for these two cities is wider and tall, so we declare the node to be an x-splitter. The splitter is the smaller point (`Edinburgh`), which goes into the left subtree, and `Lisbon` goes into the right subtree. (See Fig. 3 left.)

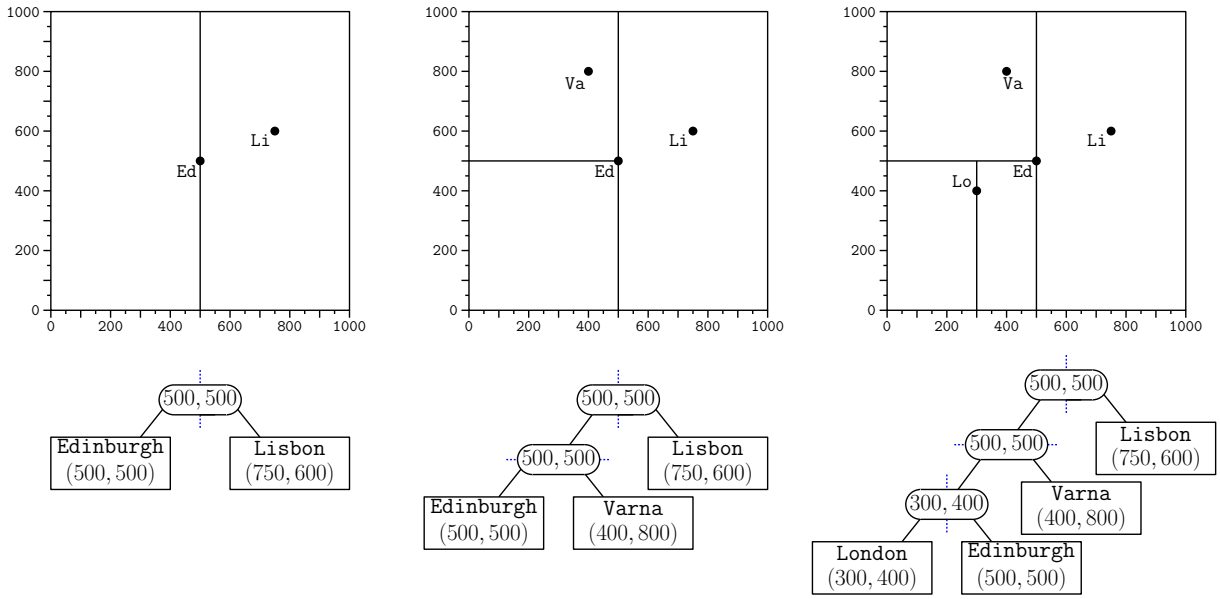

Figure 3: The SG kd-tree resulting from inserting four cities, and the associated tree structures. Note that a point that coincides with a splitter is placed in *left subtree*. The last figure shows the result of a nearest-neighbor query.

Next, `Varna` is inserted. The search ends at the external node with `Edinburgh`. The bounding rectangle for these two points is taller than wide, so the new node is a y-splitter. The splitter is the smaller point (again `Edinburgh`), which goes into the left subtree, and `Varna` goes into the right subtree. (See Fig. 3 middle.)

Finally, `London` is inserted. The search terminates at the external node containing `Edinburgh`. The bounding rectangle for these two points is wider and tall, so the new node is an x-splitter. The splitter is the smaller point (`London`), which goes into the left subtree, and `Edinburgh` goes into the right subtree. (See Fig. 3 right.)

You will notice in the above figure that we distinguish the *splitters*, which store only the (x, y) coordinates, from *external nodes*, which store all the city information. In our implementation, we distinguished `City` objects from `Point2D` objects for this reason. This is not a requirement of the programming assignment, but as a general principle, it is better to use the smallest representation needed to do the job.

Example 2: Next, let us consider a more involved example. (The successive tree structures are shown in the file `test/mytest-output-2-full-debug.txt` in the Part-2 skeleton code.)

We begin by inserting the four cities from the above example, and then one more city, `Prague` (see Fig 4 middle). This last city results in a tree of size 5 and height 4. Since $4 > \log_{3/2} 5 \approx 3.97$, a rebuild event is triggered. In this case, the root node is the scapegoat. We apply the

above rebuilding algorithm. The bounding rectangle for the points is taller than wide, so the roots splitting dimension is y. We partition the 5 points. Recalling our SG tree convention, the left subtree gets the extra point, and so `Edinburgh` is the root's splitter. The rebuilt tree is shown on the right side of Fig 4.

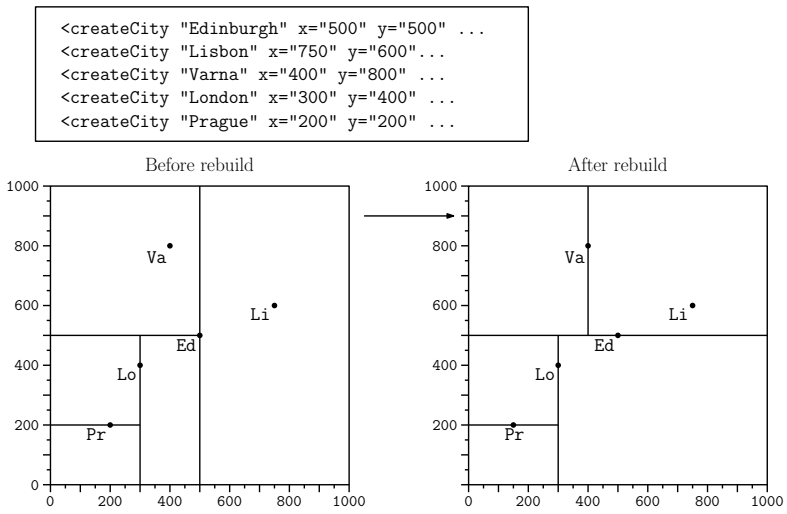

Figure 4: The SG kd-tree after inserting 5 cities both before and after rebuilding the root.

Next, we insert four more cities (see Fig 5 middle). These result in a tree of size 9 and height 6. Since $6 > \log_{3/2} 9 \approx 5.42$, another rebuilding event is signaled. Again, the root is the scapegoat, and we again apply the rebuilding algorithm. The rebuilt tree is shown on the right side of Fig 5.

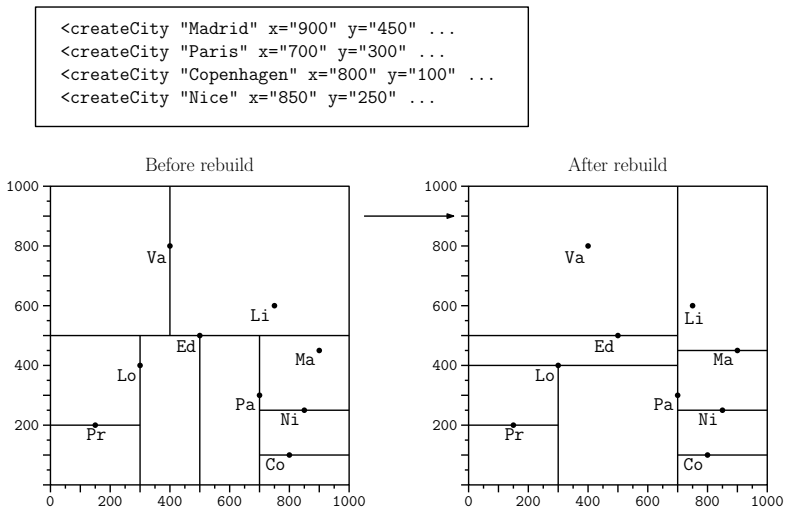

Figure 5: The SG kd-tree after inserting 4 more cities both before and after rebuilding the root.

Next, we insert an additional 7 more cities. In this case, all the cities can be inserted without triggering a rebuild event. The resulting tree is shown in Fig 6.

```
<createCity "Vienna" x="200" y="600" ...
<createCity "Amsterdam" x="550" y="150" ...
<createCity "Munich" x="100" y="900"...
<createCity "Milan" x="950" y="850"...
<createCity "Rome" x="800" y="750" ...
<createCity "Naples" x="600" y="950" ...
<createCity "Geneva" x="650" y="700" ...
```

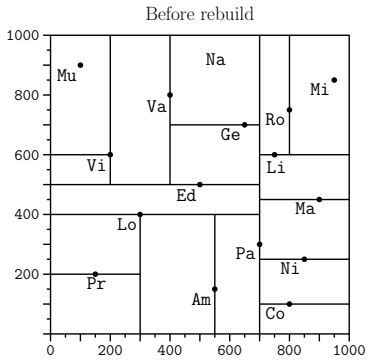

Figure 6: The SG kd-tree after inserting 7 more cities. (No rebuild needed.)

At this point, the tree contains $5+4+7=16$ cities. We delete 9 of the cities. By the rules of SG trees, the current size of the tree is $n=7$ and the upper bound on the tree size is $m=16$. Since $2n < m$, we rebuild the entire tree. The result is shown in Fig 7 on the right.

```
<deleteCity "London"/>
<deleteCity "Nice"/>
<deleteCity "Amsterdam"/>
<deleteCity "Rome"/>
<deleteCity "Munich"/>
<deleteCity "Edinburgh"/>
<deleteCity "Paris"/>
<deleteCity "Lisbon"/>
<deleteCity "Milan"/>
```

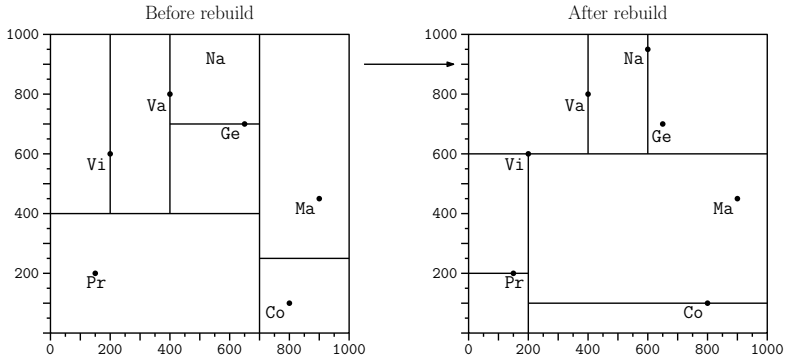

Figure 7: The SG kd-tree after deleting 9 cities both before and after rebuilding the root.