CMSC 420: Fall 2022

Programming Assignment 2: Tips on Bulk-Insertion

The principal differences between the data structure in Programming Assignment 2 and the standard kd-tree are (1) it is based on an extended binary tree (with points stored only in the leaves, not the internal node), (2) leaf nodes can hold multiple points (based on the bucket size), (3) points can be inserted "in-bulk" and the splitting process depends on the distribution of these points.

Bulk-insertion is the most complicate of the operations. In this handout, we will discuss the tree's node structure and how to perform bulk insertion with an example.

Node Structure: As mentioned in the assignment handout, the easiest way to implement an extended binary tree in Java is to use an inner class for the nodes, where there is a parent class `Node` and two subclasses `InternalNode` and `ExternalNode` derived from this. You should expect to have helper functions for each of your major operations (e.g., `find`, `bulkInsert`, `list`, and so on). The internal-node helpers mostly serve to direct points down to the appropriate external nodes, and the external node helpers do most of the real work.

These are abstract member functions, which means that Java will invoke the appropriate function depending on the node type. For example, given a node pointer `p`, the call `p.find(q)` will invoke the `InternalNode` find function if `p` is an internal node and the `ExternalNode` find function if `p` is an external node. (If you do this properly, you should not need to resort to checking a node's type using "`instance of`".)

Bulk Insertion: Let's consider this in the general case, from the perspective of a tree that already contains some points. Consider the bulk insertion of five points into the tree shown in Fig. 1(a), and suppose that the bucket size is two.

Helpers: You will have helper functions for both internal and external nodes. Both will take a list of points (actually, a list of type `LPoint`) as the argument.

Internal node: The helper for the internal node takes the list of points and splits it into two sublists consisting of the points to be placed in the left subtree and those for the right subtree. This is based on the cutting dimension and cutting value. There are many ways to perform this partition. I believe that the easiest (even if not the fastest) is to sort the points according to the cutting dimension, determine the index where to split the list, and then use Java's `subList` function to do the actual partition. (Remember that our convention is that points that fall on the splitting line are placed in the right subtree.)

For example, in Fig. 1(b), we start by sorting the input along the cutting dimension of the root, which is x, to obtain the list `[SFO, ORD, DFW, SEA, DCA]`. We partition this about $x = 5$ into the sublists `[SFO, ORD, DFW]`, which we recursively insert to the left subtree and `[SEA, DCA]`, which we recursively insert to the right.

This continues at all the internal nodes. For example, at the internal node "$y = 6$" the list is split based on the y-coordinate into the sublist `[SEA]`, which is sent to the left subtree and `[DCA]`, which is sent to the right.

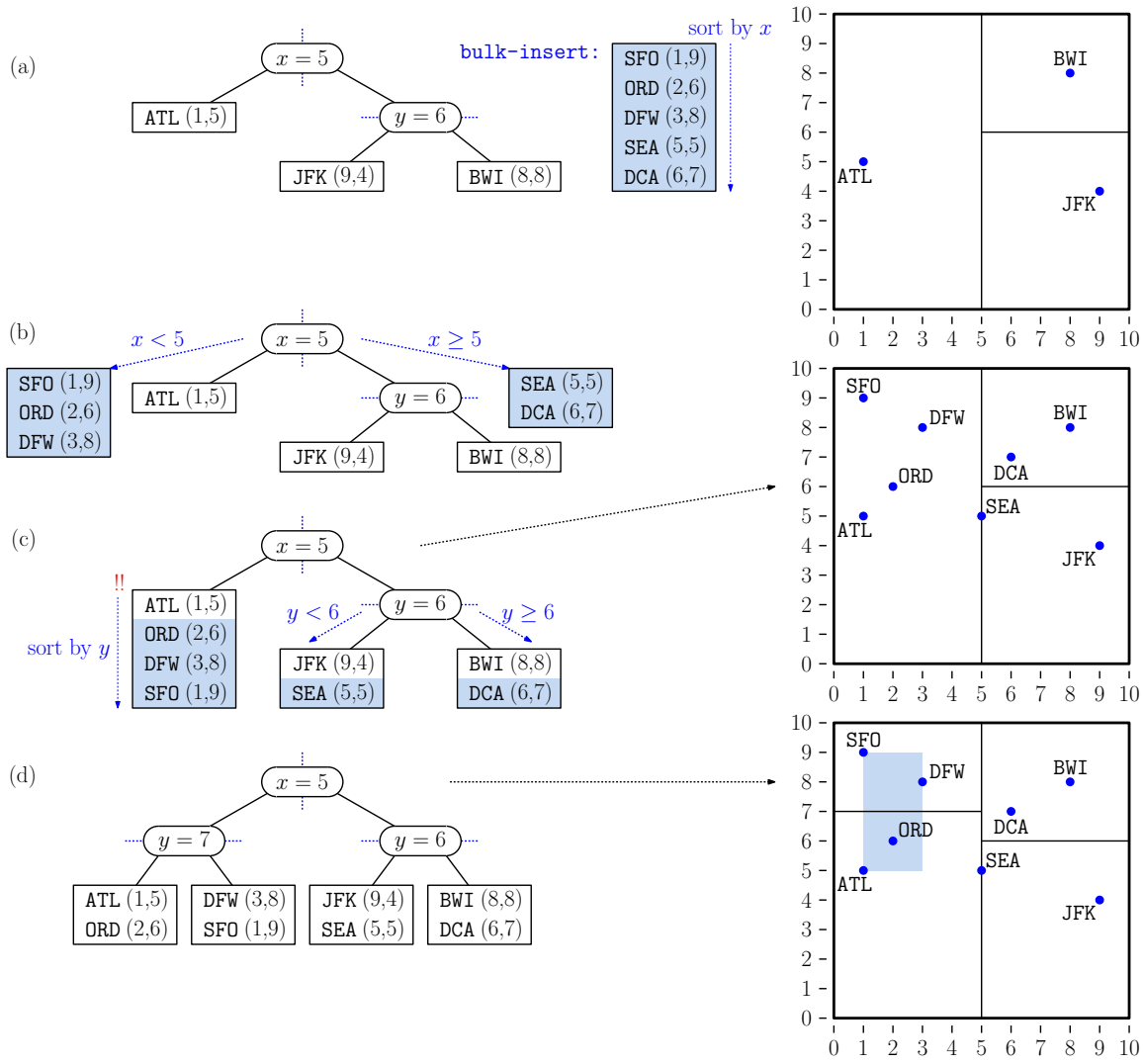

Figure 1: Bulk-insertion of five points in a tree with bucket size two.

External node: When the points arrive at an external node, they are added to the associated bucket list in this node. If the number of points does not exceed the bucket size, then we are done. (See the external nodes containing `JFK` and `BWI` in Fig. 1(c).)

Otherwise, we need to split this node. To do so, we first compute the bounding box for all the points, both new and old. (See the shaded blue rectangle in Fig. 1(d).) Depending on whether it is wider or taller, we split based on the x- or the y-axis. (In this case the rectangle is taller, so the cutting dimension is set to y (1)).

We sort the points along this dimension and select the median coordinate. (In this case it is midway between y-coordinates of `ORD` and `DFW`, which is $y = 7$.) We create an internal node having this cutting dimension and cutting value, and we partition the points to its left and right subtrees. (In this case, the sublist `[ORD, ATL]` is sent to the left subtree and `DFW, SFO]` is sent to the right. Note that in this case, both lists with within a single bucket, so we create a single external node for each and we are done.)