Programming Assignment 3: R-Tree

Classes:

-**NodeReference**: A simple class which stores an integer. This integer represents a node's ID, which is used to access that node from disk.

-**Rectangle**: Another simple class. Stores two points which represents two corners of a rectangle. This class also contains methods which determine whether a given point is contained in the rectangle object, or whether two rectangles intersect.

-**Entry**: This is an index entry, which is stored in an index node. It stores a NodeReference to its child node (4 bytes), and stores the minimum bounding rectangle (MBR) for its child's entries (16 bytes). Also contains functions which calculate the MBR for a given node.

-**Node**: An abstract class, which is extended by LeafNode, IndexNode, and OverFlowNode classes. All nodes contain an ID of type int, which is stored in this class. Besides being the parent class for the different types of nodes, it also acts as the gateway to the nodes stored on disk. Classes utilize static calls to this class to write nodes to disk and read nodes using their node reference.

Tuple: Stores an x and y value of type int (8 bytes), a hilbert value of type long (8 bytes), a NodeReference to any overflow pages that contain duplicates (4 bytes), and a 500 character string (500 bytes). The function compareTo() compares hilbert values for the initial sort, and equals() compares x and y values to determine whether the tuple is a duplicate.

LeafNode: Contains an array of tuples. The size of a leaf node is strictly 4096 bytes. In order to find out how many tuples can fit in a leaf node, we need to use this formula: 2d((2*4) + 8 + 500)) <= 4096. Solving for floor(2d), we have 2d = floor(7.938) = 7. 4KB leaf pages can only contain 7 tuples, so MAX_ENTRIES is set to this. When a leaf node is created, the array is initialized and has a length of MAX_ENTRIES. The function addTuple() locates the correct location to place the tuple. If the tuple is

a duplicate, then we pass off all responsibility to the OverFlowNode class.

IndexNode: Contains an array of Entry objects. Using this formula, we can find out how many entries can fit inside an index node: 2d((4 * 4) + 4) <= 4096. Solving for floor(2d), we have 2d = floor(204.8) = 204 entries per index node. MAX_ENTRIES is set to this. Functionality is very similar to the LeafNode class. When an index is added, the MBR for its child is calculated.

OverFlowNode: Contains a node reference to the next overflow page if all duplicates do not fit in the current overflow node. Resembles a linkedlist, with the head reference located in the Tuple class.

RTree: The main runner class. First checks if there are data files from the previous session and deletes them. Then, it loads all points in the dataset into the priority queue “hilbertValues”. Once the queue contains all the tuples, the bulkLoad function is called.

The algorithm to bulk load the tree is somewhat different than what you'd find elsewhere: First, create a queue (called “childrenToBe”) to store all leaf node NodeReference objects. Then, create all leaf nodes by filling each up with tuples until no more can be inserted into a node. When the node is full, write the node to disk with a newly generated ID. Also, insert the new NodeReference into the queue. Move on to the next leaf node. Do this for all leaf nodes.

When all leaf nodes have been created and written to disk, we can start creating index nodes. Create another queue where all the current level's NodeReferences will be stored (called “currentLevel”). Now, dequeue from the queue that was populated while creating the leaf nodes, and calculate the MBR for the node. Store the dequeued NodeReference and the Rectangle in the current index node. Create a new NodeReference with a newly generated ID for the current index node, and store it in the currentLevel queue. Write the index node to disk using the new NodeReference, and continue on to the next node. Do this until the queue “childrenToBe” is empty. Once it's empty, we move up to the next level. The queue “childrenToBe” is assigned to the queue that was being populated (currentLevel), and

currentLevel is assigned to a new empty queue. Throughout this process, we check if the size of the currentLevel is 1 and if the size of “childrenToBe” is 0. If that's the case, that means that the root node has been created, and the bulk loading process is complete.

The search algorithms are rudimentary in nature because of the data structure being used. Point searches are performed by checking to see if a point lies in a given bounding box. If so, then that box is searched recursively. However, it does not stop searching at that box. All boxes at the root are checked to see if the point lies within their bounding boxes. If not, then the tree isn't traversed any further for that box.

Searching with a bounding box is very similar to the process of point searching. Instead of checking whether a box is contained within another, we must check if the box intersects with the other. If so, then the tree is traversed recursively, and gathers all points that lie within the search box.

One thing I cannot understand about the assignment: Upon inspecting the size of index node files in the filesystem, I saw that they were 8KB in size, instead of 4KB. I believe the calculations are correct, so I can only assume that I did not account for any overhead for these objects. Cutting MAX_ENTRIES (in IndexNode.java) solves the issue, but doesn't really tell me anything. Other than that, this assignment was fun, frustrating, and time consuming. A nice mix, I guess.