CS 266 Homework 5

Zachary DeStefano, PhD Student, 15247592

Due Date: May 15

Problem 10.1

In Section 10.1 we solved the problem of finding all horizontal line segments in a set that intersect a vertical segment. For this we used an interval tree with priority search trees as associated structures. There is also an alternative approach. We can use a 1-dimensional range tree on the y-coordinate of the segments to determine those segments whose y-coordinate lies in the y-range of the query segment. The resulting segments cannot lie above or below the query segment, but they may lie completely to the left or to the right of it. We get those segments in O(logn) canonical subsets. For each of these subsets we use as an associated structure an interval tree on the x-coordinates to find the segments that actually intersect the query segment. q a. Give the algorithm in pseudocode. b. Prove that the data structure correctly solves the queries. c. What are the bounds for preprocessing time, storage, and query time of this structure? Prove your answers.

Part A

Here is the algorithm:
Traverse binary search tree for y-coordinate until you hit a split vertex
Traverse right subtree:
- Every time you make a right turn, call the left node n and do Recurse(n)

Traverse left subtree:
- Every time you make a left turn, call the right node n and do Recurse(n)

Recurse(n):
- Report intervals in the binary search tree for n that lie in our point.

Part B

We have a binary search tree that will correctly get us the intervals with the correct y-coordinate. We then have an interval tree on our x-coordinates that will get us the matching x's. Thus we will get segments that correctly fit our requirements.

Part C

Preprocessing:
The binary search tree can be constructed in $O(n \cdot logn)$
Each segment tree can be constructed in $O(n \cdot logn)$ time.
Total thus is $O(n^2 log^2(n))$ time.

Query:
Traversing the first one will take O(log n) operations.
Traversing the query time will be done up to log(n) times.
For each time, we have a segment tree, so the count will take $logn$ times to obtain.
We thus have a query time of $O(log^2(n))$

Storage:
A binary search tree on the y-coordinate can be stored in $O(n)$ space.
The segment trees are each $O(n \cdot log(n))$ space.
Thus the total space is $O(n^2 \cdot log(n))$

Problem 10.6c

Let I be a set of intervals on the real line. We want to be able to count the number of intervals containing a query point in O(logn) time. Thus, the query time must be independent of the number of segments containing the query point.

Describe a data structure for this problem based on a simple binary search tree. Your structure should have O(n) storage and O(logn) query time. (Hence, segment trees are actually not needed to solve this problem efficiently.)

Construct a binary search tree of the endpoints. Binary search trees take $O(n)$ storage. For each leaf in the Binary Search Tree, store the number of overlapping intervals.
Construction algorithm:
1. Make a BST of endpoints.
2. Sort the list of endpoints.
3. Initialize count to zero.
4. For each endpoint in ascending order:
If left endpoint:
- Add one to count
If right endpoint:
- Subtract one from count
Put count into endpoint node in Binary Search Tree.

Query algorithm to get point q:
1. Traverse binary search tree until you find the greatest endpoint that is less than q
2. Report its count.

Construction will be $O(n \cdot logn)$, query will be $O(logn)$ since it is still a binary search tree.

Problem 14.6

In this chapter we called a quadtree balanced if two adjacent squares of the quadtree subdivision differ by no more than a factor two in size. To save a constant factor in the number of extra nodes needed to balance a quadtree, we could weaken the balance condition by allowing adjacent squares to differ by a factor of four in size. Can you still complete such a weakly balanced quadtree subdivision to a conforming mesh such that all angles are between 45 and 90 by using only O(1) triangles per square?

Now, we cannot do this, because we might end up with obtuse angles if we use the weaker condition.
TODO: Insert example, read more page 315 of book

Problem 14.12

Suppose we have quadtrees on pixel images I1 and I2 (see the previous exercise). Both images have size 2k 2k, and contain only two intensities, 0 and 1. Give algorithms for Boolean operations on these images, that is, give algorithms to compute a quadtree for I1 I2 and I1 I2. (Here I1 I2 is the 2k 2k image where pixel (i, j) has intensity 1 if and only if (i, j) has intensity 1 in image I1 or in image I2. The image I1 I2 is defined similarly.)

For this algorithm, a vertex v in the quad tree will have this data:
- subtree(i) will be pointers to subvertices if this square is split up. i = 1,2,3,4
- value will be intensity value if this vertex is a leaf.

J_1, J_2 will denote the quad trees for the images and will have the root field as pointers to the root vertex

Here is the algorithm:
Run GetANDQuadTree(J_1,J_2) and GetORQuadTree(J_1,J_2) to get the root vertices with all their subtrees

```
GetANDQuadTree(I,J):
-      Initialize vertex v with no subtrees or values
-      if both I,J have subtrees:
-           for each subtree i from 1 to 4:
-                v.subtree(i) = GetANDQuadTree(I.subtree(i),J.subtree(i))
-           return v
-      else if both I,J are leaves:
-           v.value = I.value and J.value
-      else:
```

```
-           let K_1 be the one that is a leaf
-           let K_2 be the one that ha a subtree
-           if K_1.value is 0:
-                v.value = 0
-                return v
-           else:
-                v.subtree(i)=K_2.subtree(i) for i from 1 to 4
-                return v

GetORQuadTree(I,J):
-      Initialize vertex v with no subtrees or values
-      if both I,J have subtrees:
-           for each subtree i from 1 to 4:
-                v.subtree(i) = GetORQuadTree(I.subtree(i),J.subtree(i))
-           return v
-      else if both I,J are leaves:
-           v.value = I.value and J.value
-      else:
-           let K_1 be the one that is a leaf
-           let K_2 be the one that ha a subtree
-           if K_1.value is 1:
-                v.value = 1
-                return v
-           else:
-                v.subtree(i)=K_2.subtree(i) for i from 1 to 4
-                return v
```