6.851: Advanced Data Structures Spring 2010

Lecture 4 — 11 February, 2010

Dr. André Schulz *Scribe: Cai GoGwilt*

1 Overview

In the last lecture we discussed the orthogonal range query problem and solutions to it. In the orthogonal range query problem, we are given a set of points and a rectangular area, and are asked to report the points that lie in this area. We discussed three types of solutions: range trees, kd-trees (in the 2D case), and fractional cascading.

In this lecture we briefly discuss extending our orthogonal range query solutions to account for dynamic point sets. We then introduce the problem of vertical line stabbing queries, present two solutions, and review an application.

2 Orthogonal Range Queries and Dynamic Point Sets

In our discussion of the orthogonal range query problem in lecture 3, we only considered static point sets. What costs are associated with modifying the point set as we do the queries? In other words, what if we modify our data structure to handle dynamic point sets? In "The Design of Dynamic Data Structures," Mark H. Overmars discusses ideas and costs associated with turning a static data structure into a dynamic one.[1]

The orthogonal range query problem entails decomposable searches. This means we can partition queries, q, on keys, x, into queries on subsets of x: (x, q) partitions into (x_1, q) and (x_2, q) where $x_1 \cup x_2 = x$. In this case, insertions take $O(\frac{T_b}{n} \log n)$ amortized time (where T_b is the data structure build time), and queries grow by a $\log n$ factor.

Thus, for our dynamic 2D range-trees, queries will take $O(\log^2 n + k)$ time and insertions will take $O(\log^2 n)$ time.

3 Vertical Line Stabbing Queries

In the vertical line stabbing queries problem, we are given, as input, n intervals:

$$I = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\},$$

and a query $q \in \mathbb{R}$. Our task is to report all intervals that contain q.

4 Interval-Tree

The interval-tree uses the idea of divide-and-conquer. We divide the intervals into sets based on their relation to the median of the interval end-points. Some will lie completely to the left or right of the median, and others will intersect. We will recurse on the intervals that lie to the left or right. For those that intersect, we will sort them, which lets us stop after we find the first interval that we are not interested in.

4.1 Example Run

As an example of this working, we will use the interval:

$$I = \{[1, 6], [3, 20], [3, 7], [5, 17], [10, 20], [13, 15]\}.$$

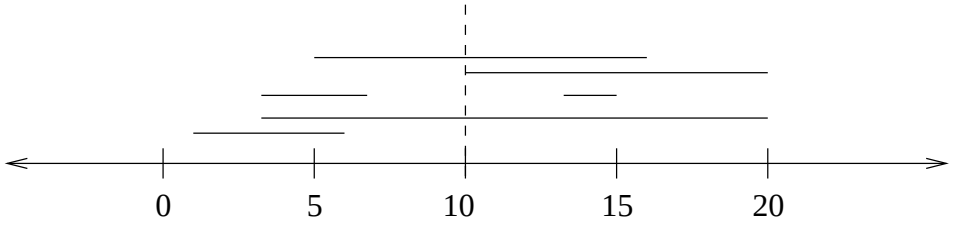

Figure 1: The intervals for the example, with the median of the end-points plotted as a dotted line.

We take the median to be 10, so we have:

$$\begin{aligned} I_l &= \{[1, 6], [3, 7]\} \\ I_r &= \{[13, 15]\} \\ I_m &= \{[3, 20], [5, 17], [10, 20]\}, \end{aligned}$$

where I_l is the set of intervals fully to the left of the median, I_r is the set fully right, and I_m is the set that intersects. We then recursively create a tree. The parent node contains the set I_m twice: one sorted by the left interval bound, the other by the right. The children are recursively created in the same manner on I_l and I_r. Thus we have the structure in figure 2.

If we query 18, for example, then we would report the right half of the parent node until we hit $[5, 17]$, at which point we would stop, and recurse on the right child node.

4.2 Analysis

Method: In general terms, the data structure splits the point set into I_l, I_r, and I_m along the median. It then stores I_m as two sorted lists in a parent node, and then creates children by recursing on I_r and I_l.

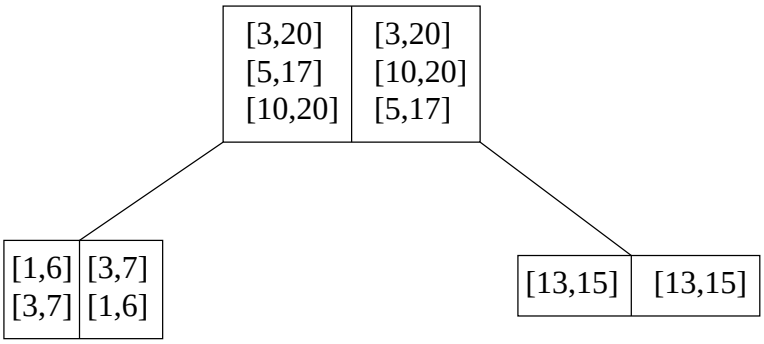

Figure 2: The interval-tree from the example.

Query: Following the path down the tree is an $O(\log n)$ operation. Once we find a node, we report $O(k)$ items, where k is the number of intervals stabbed. Thus, query runs in $O(\log n + k)$ time.

Storage: Each interval is stored exactly twice, so the structure is $O(n)$.

Preprocessing: The main cost in building the data structure is sorting the interval lists, which is $O(n \log n)$.

5 Segment-Tree

The idea behind the segment tree is to split the real line into *elementary intervals*. We can then build a binary search tree and store intervals in appropriate places in the tree.

5.1 Example Run

To split the reals into elementary intervals, we create closed intervals that create only the end-points of our input intervals, and fill in the gaps in the reals with open intervals. For example, we can take the end-points of the intervals from the example above, which leaves us with the intervals shown in figure 3.

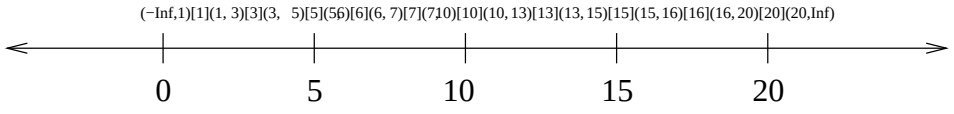

Figure 3: The elementary intervals we would get from the intervals given in the example.

We can now build a binary search tree over the elementary intervals, as shown in figure 3.

One possibility is to store intervals in all appropriate leaves, which would give us $O(\log n)$ queries very easily (binary search), but would leave us with bad storage in many cases. Instead, we store intervals in parent nodes if both children of that node are contained in that interval (and if the node's sibling is not also contained in the interval). This gives us a *canonical subdivision*. For example, the canonical subdivision of $[3, 7]$ in the example above is $[3](3, 6](6, 7]$.

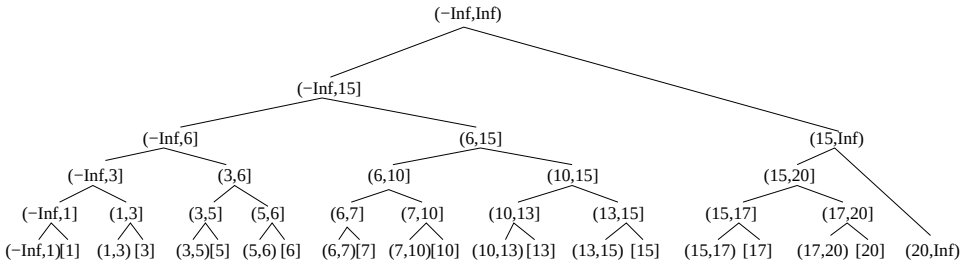

Figure 4: A binary search tree over the elementary intervals.

5.2 Analysis

Query: When we search for q, we report all intervals along the path. Thus, we again achieve $O(\log n + k)$ time. However, it is important to note that we report every interval we find.

Storage: We claim that every level of the tree stores every interval at most twice. The proof of this is by contradiction. If we had more than two instances of an interval in a single level of the tree with no shared parents, there would be a gap in the interval. However, we only store closed interval. Storage is therefore $O(n \log n)$. This is worse than for the interval-tree.

Preprocessing: Preprocessing is $O(n \log n)$.

6 Application: Windowing

An application of vertical line stabbing is windowing. We are given a large geometric structure and a window. We want to display only the part of the geometric structure that lies in the window.

To do this, we have to report all line segments that lie within or intersect the window. There are two parts to this problem:

1. Finding every line segment with one or two endpoints inside the window.
2. Finding every line segment that goes straight through the window.

For the first part, we can use our solutions to the orthogonal range searching problem. For the second part, we need something like vertical line stabbing.

To make our initial discussion simpler, we will limit ourselves to the case where the geometric figure is made up of orthogonal line segments.

6.1 Simple Approach

In the case of orthogonal line segments, part 2 of the windowing problem becomes finding the lines with y-coordinates that are in the correct region. Our solution is then to sort the line segments by their y-coordinate. As an initial approach, we use an interval tree, but store a 2D range tree in

each of the nodes. This makes the query time $O(\log^2 n + k)$, but the structure itself takes up too much space.

6.2 More complex approach

In order to solve this problem efficiently with respect to storage, we need to introduce another data structure. The main task, as we can see from figure 5, is to report the points in regions next to the window.

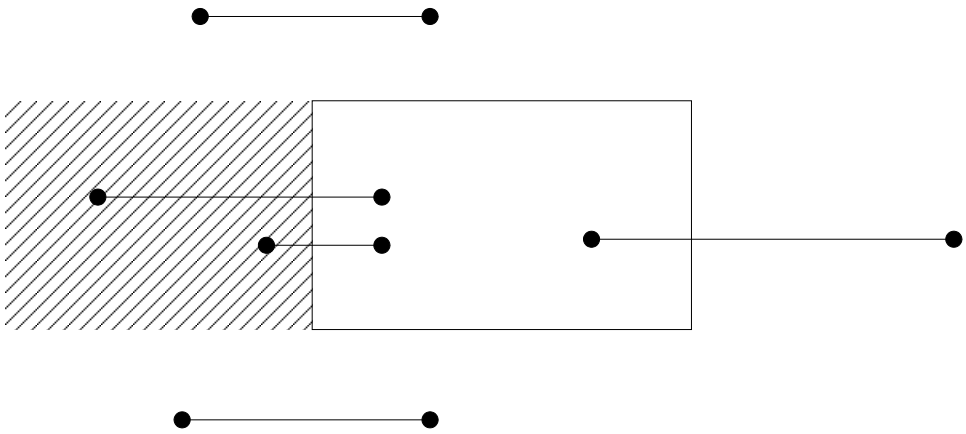

Figure 5: The region we are interested in finding points in.

Priority Search Trees In a priority search tree, we store a 2D point set like a heap. This allows us to efficiently query points in a rectangle with one open side. We construct it by selecting the minimum point (with respect to the x-coordinate) as the root. We then recursively build the left and right subtrees out of points with y-coordinates less than and greater than the median of the y-coordinates. This gives us $O(n)$ storage, and allows us to query for a rectangular region by searching the tree. When our search splits, we report all right subtrees recursively as we search for the minimum, but stop if the root is greater than our query.

6.3 Handling Lines with Slope

For lines with slopes, we can use modified segment trees. The key advantage is that whenever we touch an interval during a query, we know that interval has been stabbed. Developing this is left as an exercise.

References

[1] Mark H. Overmars, *The Design of Dynamic Data Structures*, Berlin: Springer-Verlag, 1983.