Lecture 21: Interval Trees

Reading: The presentation is not taken from any of our readings. It is derived from a description in the book *Computational Geometry: Algorithms and Applications*, by M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, Chapt 10. A copy is on reserve in the computer science library in A.V. Williams.

Segment Data: So far we have considered geometric data structures for storing points. However, there are many others types of geometric data that we may want to store in a data structure. Today we consider how to store orthogonal (horizontal and vertical) line segments in the plane. We assume that a line segment is represented by giving its pair of *endpoints*. The segments are allowed to intersect one another.

As a basic motivating query, we consider the following *window query*. Given a set of orthogonal line segments S, which have been preprocessed, and given an orthogonal query rectangle W, count or report all the line segments of S that intersect W. We will assume that W is closed and solid rectangle, so that even if a line segment lies entirely inside of W or intersects only the boundary of W, it is still reported. For example, given the window below, the query would report the segments that are shown with solid lines, and segments with broken lines would not be reported.

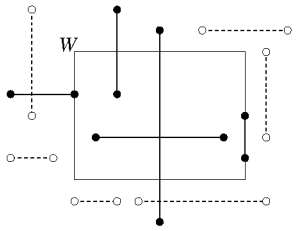

Figure 62: Window Query.

Window Queries for Orthogonal Segments: We will present a data structure, called the *interval tree*, which (combined with a range tree) can answer window counting queries for orthogonal line segments in $O(\log^2 n)$ time, where n is the number line segments. It can report these segments in $O(k + \log^2 n)$ time, where and k is the total number of segments reported. The interval tree uses $O(n \log n)$ storage and can be built in $O(n \log n)$ time.

We will consider the case of range reporting queries. (There are some subtleties in making this work for counting queries.) We will derive our solution in steps, starting with easier subproblems and working up to the final solution. To begin with, observe that the set of segments that intersect the window can be partitioned into three types: those that have no endpoint in W, those that have one endpoint in W, and those that have two endpoints in W.

We already have a way to report segments of the second and third types. In particular, we may build a range tree just for the $2n$ endpoints of the segments. We assume that each endpoint has a cross-link indicating the line segment with which it is associated. Now, by applying a range reporting query to W we can report all these endpoints, and follow the cross-links to report the associated segments. Note that segments that have both endpoints in the window will be reported twice, which is somewhat unpleasant. We could fix this either by sorting the segments in some manner and removing duplicates, or by marking each segment as it is reported and ignoring segments that have already been marked. (If we use marking, after the

query is finished we will need to go back an "unmark" all the reported segments in preparation for the next query.)

All that remains is how to report the segments that have no endpoint inside the rectangular window. We will do this by building two separate data structures, one for horizontal and one for vertical segments. A horizontal segment that intersects the window but neither of its endpoints intersects the window must pass entirely through the window. Observe that such a segment intersects any vertical line passing from the top of the window to the bottom. In particular, we could simply ask to report all horizontal segments that intersect the left side of W. This is called a *vertical segment stabbing query*. In summary, it suffices to solve the following subproblems (and remove duplicates):

Endpoint inside: Report all the segments of S that have at least one endpoint inside W. (This can be done using a range query.)

Horizontal through segments: Report all the horizontal segments of S that intersect the left side of W. (This reduces to a vertical segment stabbing query.)

Vertical through segments: Report all the vertical segments of S that intersect the bottom side of W. (This reduces to a horizontal segment stabbing query.)

We will present a solution to the problem of vertical segment stabbing queries. Before dealing with this, we will first consider a somewhat simpler problem, and then modify this simple solution to deal with the general problem.

Vertical Line Stabbing Queries: Let us consider how to answer the following query, which is interesting in its own right. Suppose that we are given a collection of horizontal line segments S in the plane and are given an (infinite) vertical query line $\ell_q : x = x_q$. We want to report all the line segments of S that intersect ℓ_q. Notice that for the purposes of this query, the y-coordinates are really irrelevant, and may be ignored. We can think of each horizontal line segment as being a closed *interval* along the x-axis. We show an example in the figure below on the left.

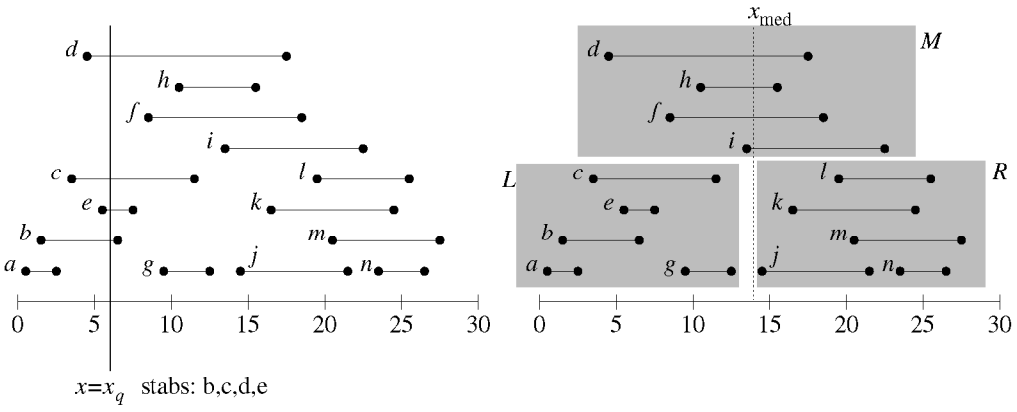

Figure 63: Line Stabbing Query.

As is true for all our data structures, we want some balanced way to decompose the set of intervals into subsets. Since it is difficult to define some notion of order on intervals, we instead will order the endpoints. Sort the interval endpoints along the x-axis. Let $\langle x_1, x_2, \ldots, x_{2n} \rangle$ be the resulting sorted sequence. Let x_{med} be the median of these $2n$ endpoints. Split the

intervals into three groups, L, those that lie strictly to the left of x_{med}, R those that lie strictly to the right of x_{med}, and M those that contain the point x_{med}. We can then define a binary tree by putting the intervals of L in the left subtree and recursing, putting the intervals of R in the right subtree and recursing. Note that if $x_q < x_{\text{med}}$ we can eliminate the right subtree and if $x_q > x_{\text{med}}$ we can eliminate the left subtree. See the figure right.

But how do we handle the intervals of M that contain x_{med}? We want to know which of these intervals intersects the vertical line ℓ_q. At first it may seem that we have made no progress, since it appears that we are back to the same problem that we started with. However, we have gained the information that all these intervals intersect the vertical line $x = x_{\text{med}}$. How can we use this to our advantage?

Let us suppose for now that $x_q \leq x_{\text{med}}$. How can we store the intervals of M to make it easier to report those that intersect ℓ_q. The simple trick is to sort these lines in increasing order of their left endpoint. Let M_L denote the resulting sorted list. Observe that if some interval in M_L does not intersect ℓ_q, then its left endpoint must be to the right of x_q, and hence none of the subsequent intervals intersects ℓ_q. Thus, to report all the segments of M_L that intersect ℓ_q, we simply traverse the sorted list and list elements until we find one that does not intersect ℓ_q, that is, whose left endpoint lies to the right of x_q. As soon as this happens we terminate. If k' denotes the total number of segments of M that intersect ℓ_q, then clearly this can be done in $O(k' + 1)$ time.

On the other hand, what do we do if $x_q > x_{\text{med}}$? This case is symmetrical. We simply sort all the segments of M in a sequence, M_R, which is sorted from right to left based on the right endpoint of each segment. Thus each element of M is stored twice, but this will not affect the size of the final data structure by more than a constant factor. The resulting data structure is called an *interval tree*.

Interval Trees: The general structure of the interval tree was derived above. Each node of the interval tree has a left child, right child, and itself contains the median x-value used to split the set, x_{med}, and the two sorted sets M_L and M_R (represented either as arrays or as linked lists) of intervals that overlap x_{med}. We assume that there is a constructor that builds a node given these three entities. The following high-level pseudocode describes the basic recursive step in the construction of the interval tree. The initial call is `root = IntTree(S)`, where S is the initial set of intervals. Unlike most of the data structures we have seen so far, this one is not built by the successive insertion of intervals (although it would be possible to do so). Rather we assume that a set of intervals S is given as part of the constructor, and the entire structure is built all at once. We assume that each interval in S is represented as a pair $(x_{\text{lo}}, x_{\text{hi}})$. An example is shown in the following figure.

Interval tree construction

```
IntTreeNode IntTree(IntervalSet S) {
    if (|S| == 0) return null                          // no more

    xMed = median endpoint of intervals in S           // median endpoint

    L = {[xlo, xhi] in S | xhi < xMed}                 // left of median
    R = {[xlo, xhi] in S | xlo > xMed}                 // right of median
    M = {[xlo, xhi] in S | xlo <= xMed <= xhi}         // contains median
    ML = sort M in increasing order of xlo             // sort M
    MR = sort M in decreasing order of xhi

    t = new IntTreeNode(xMed, ML, MR)                  // this node
    t.left  = IntTree(L)                               // left subtree
    t.right = IntTree(R)                               // right subtree
```

```
    return t
}
```

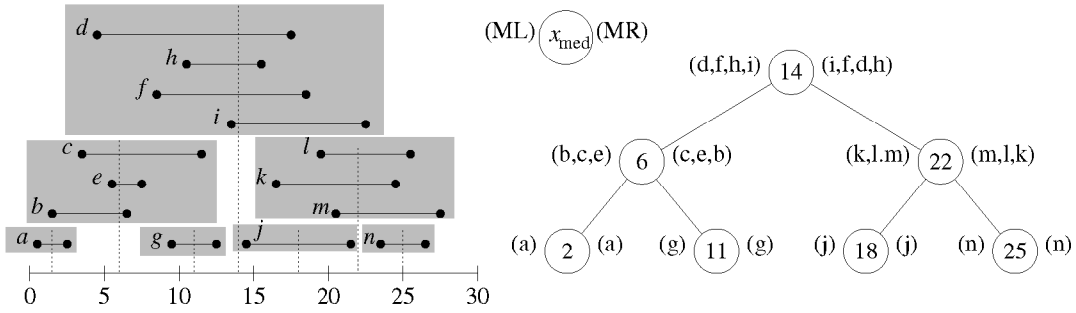

Figure 64: Interval Tree.

We assert that the height of the tree is $O(\log n)$. To see this observe that there are $2n$ endpoints. Each time through the recursion we split this into two subsets L and R of sizes at most half the original size (minus the elements of M). Thus after at most $\lg(2n)$ levels we will reduce the set sizes to 1, after which the recursion bottoms out. Thus the height of the tree is $O(\log n)$.

Implementing this constructor efficiently is a bit subtle. We need to compute the median of the set of all endpoints, and we also need to sort intervals by left endpoint and right endpoint. The fastest way to do this is to presort all these values and store them in three separate lists. Then as the sets L, R, and M are computed, we simply copy items from these sorted lists to the appropriate sorted lists, maintaining their order as we go. If we do so, it can be shown that this procedure builds the entire tree in $O(n \log n)$ time.

The algorithm for answering a stabbing query was derived above. We summarize this algorithm below. Let x_q denote the x-coordinate of the query line.

Line Stabbing Queries for an Interval Tree

```
stab(IntTreeNode t, Scalar xq) {
    if (t == null) return                       // fell out of tree
    if (xq < t.xMed) {                          // left of median?
        for (i = 0; i < t.ML.length; i++) {     // traverse ML
            if (t.ML[i].lo <= xq) print(t.ML[i])// ..report if in range
            else break                          // ..else done
        }
        stab(t.left, xq)                        // recurse on left
    }
    else {                                      // right of median
        for (i = 0; i < t.MR.length; i++) {     // traverse MR
            if (t.MR[i].hi >= xq) print(t.MR[i])// ..report if in range
            else break                          // ..else done
        }
        stab(t.right, xq)                       // recurse on right
    }
}
```

This procedure actually has one small source of inefficiency, which was intentionally included to make code look more symmetric. Can you spot it? Suppose that $x_q = t.x_{\text{med}}$? In this case we will recursively search the right subtree. However this subtree contains only intervals that

are strictly to the right of x_{med} and so is a waste of effort. However it does not affect the asymptotic running time.

As mentioned earlier, the time spent processing each node is $O(1 + k')$ where k' is the total number of points that were recorded at this node. Summing over all nodes, the total reporting time is $O(k+v)$, where k is the total number of intervals reported, and v is the total number of nodes visited. Since at each node we recurse on only one child or the other, the total number of nodes visited v is $O(\log n)$, the height of the tree. Thus the total reporting time is $O(k+\log n)$.

Vertical Segment Stabbing Queries: Now let us return to the question that brought us here. Given a set of horizontal line segments in the plane, we want to know how many of these segments intersect a vertical line segment. Our approach will be exactly the same as in the interval tree, except for how the elements of M (those that intersect the splitting line $x = x_{\text{med}}$) are handled.

Going back to our interval tree solution, let us consider the set M of horizontal line segments that intersect the splitting line $x = x_{\text{med}}$ and as before let us consider the case where the query segment q with endpoints (x_q, y_{lo}) and (x_q, y_{hi}) lies to the left of the splitting line. The simple trick of sorting the segments of M by their left endpoints is not sufficient here, because we need to consider the y-coordinates as well. Observe that a segment of M stabs the query segment q if and only if the left endpoint of a segment lies in the following semi-infinite rectangular region.

$$\{(x, y) \mid x \le x_q \text{ and } y_{\text{lo}} \le y \le y_{\text{hi}}\}.$$

This is illustrated in the figure below. Observe that this is just an orthogonal range query. (It is easy to generalize the procedure given last time to handle semi-infinite rectangles.) The case where q lies to the right of x_{med} is symmetrical.

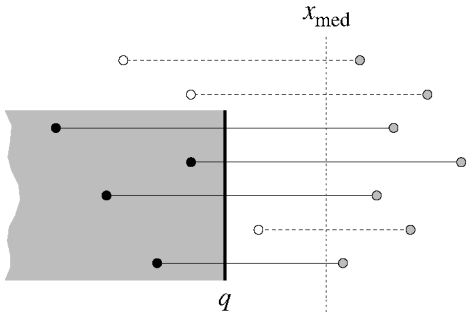

Figure 65: The segments that stab q lie within the shaded semi-infinite rectangle.

So the solution is that rather than storing M_L as a list sorted by the left endpoint, instead we store the left endpoints in a 2-dimensional range tree (with cross-links to the associated segments). Similarly, we create a range tree for the right endpoints and represent M_R using this structure.

The segment stabbing queries are answered exactly as above for line stabbing queries, except that part that searches M_L and M_R (the for-loops) are replaced by searches to the appropriate range tree, using the semi-infinite range given above.

We will not discuss construction time for the tree. (It can be done in $O(n \log n)$ time, but this involves some thought as to how to build all the range trees efficiently). The space needed is $O(n \log n)$, dominated primarily from the $O(n \log n)$ space needed for the range trees. The query time is $O(k + \log^3 n)$, since we need to answer $O(\log n)$ range queries and each takes

$O(\log^2 n)$ time plus the time for reporting. If we use the spiffy version of range trees (which we mentioned but never discussed) that can answer queries in $O(k + \log n)$ time, then we can reduce the total time to $O(k + \log^2 n)$.