CPS234 Computational Geometry **September 20th, 2005**

Lecture 7: Segment, Interval, Priority-Search Trees

Lecturer: Pankaj K. Agarwal *Scribe: Mason F. Matthews*

7.1 Segment Trees

7.1.1 Problem Statement

Suppose that we are given input I, a set of n non-intersecting segments in $\mathbb{R}^2$. A query is the triple (q_x, q_y, q'_y) representing the vertical line segment from (q_x, q_y) to (q_x, q'_y). We wish to return a list of the segments in I that intersect with the query segment.

7.1.2 Data Structure

The segment tree, intially proposed in [1], is a two-level data structure used to efficiently solve the problem given above. The primary structure is a binary tree T, and each node in T is built to contains a subset of the segments in I. Each node also contains a link to a secondary data structure: a binary tree. This secondary tree will be described shortly.

Let's first gain an intuition for the primary data structure. Consider the input segments A through E given in Figure 7.1. If we project these segments down to $\mathbb{R}$, given by the horizontal number line, then the endpoints partition $\mathbb{R}$.

To build a segment tree, create one leaf node for each partition, and one leaf node for each endpoint. These nodes are represented by squares in the figure. As each leaf node is built, all intervals that project to the corresponding region of $\mathbb{R}$ are added to the node. Next, join the nodes in pairs from the bottom up to form a binary tree.

As each internal node is created, consider its two children. For all segments stored in both children, copy the segment into the new internal node and remove it from the two children. See Figure 7.1 for an example of the data structure once the internal nodes have been created and the segments have reached their final positions.

At query time, the tree is traversed from the root, and the path to a leaf node is determined by q_x. All segment names found at the nodes along this path are added to a list, and once the leaf node is reached, the list contains all segments that intersect the vertical line $x = q_x$. Unfortunately, this is a superset of the segments intersected by q; some of them may pass above or below q.

To winnow this list down, we use the secondary trees stored at each node. The right side of Figure 7.2 provides an example of one of these trees, and the left side of the figure shows the corresponding region of

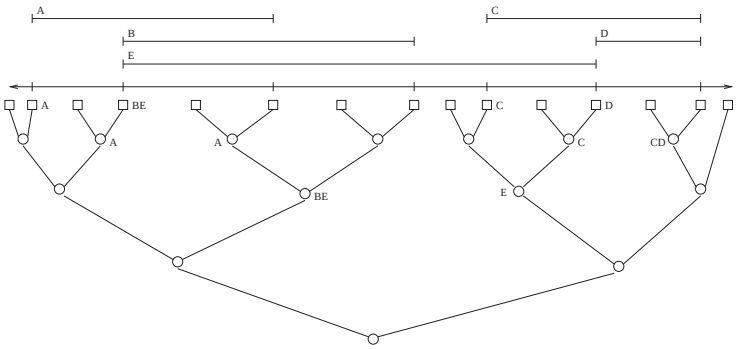

Figure 7.1: Constructing a Segment Tree

space. Here, the traversal has reached an internal node V containing the set $\{A, B, C\}$, and its descendents span the region denoted by the dotted lines. While D and E are also in this region, they do not span the entire region, and therefore do not appear in V. Since all input segments are non-intersecting, we know that the relative ordering of A, B, and C is fixed over the region. The secondary binary tree containing $\{A, B, C\}$ is indexed by this relative ordering. Two traversals in this tree can determine which of the segments are intersected by q. The segment E will be reported later as the query proceeds to a lower node in the primary tree.

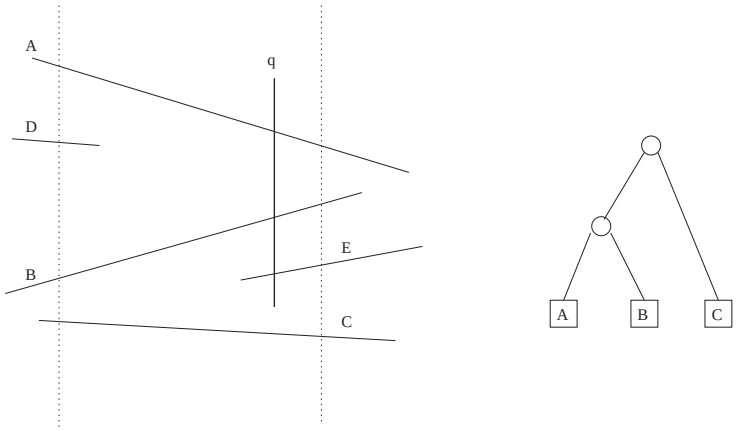

Figure 7.2: Example Query and Secondary Data Structure for an Internal Node of Segment Tree

7.1.3 Time and Space Bounds

Naive construction time for a segment tree is $O(n \lg^2 n)$. Methods exist for faster construction, and they can reduce the time to $O(n \lg n)$. Worst-case query time is $O(\lg^2 n + k)$, due to the fact that a secondary tree needs to be traversed at each primary node, and the secondary trees can be of height $O(\lg n)$.

The space required for this tree is $O(n \lg n)$. An input segment can be stored at multiple levels in the primary tree, as shown in Figure 7.1, but it can only appear at a given level twice. If it appeared more than twice, then

two of those nodes must share a parent, and the segment would have been moved to that parent. Thus, space required for storing segments in the primary tree is $O(n \lg n)$. Adding the secondary trees does not increase the space requirement by more than a constant factor.

As a side note, space and time complexity can be improved when queries are restricted to vertical lines. In this case, only the primary data structure is required. The preprocessing time for this structure is $O(n \lg n)$, and queries can be handled in $O(\lg n + k)$ time, as each step in the tree traversal requires constant time plus the number of nodes reported.

7.1.4 Klee's Measure

One useful application of segment trees has been in the computation of Klee's measure. The one-dimensional case of the problem can be phrased as follows: Given a set of intervals on the real line, how do we compute the length of their union? Klee suggested a $O(n \lg n)$ bound using simple sorting, and later this was shown to be optimal [3].

With the simple question tackled, the two-dimensional generalization then arose: given a set of axis-aligned rectangular regions, how can we compute the area of their union? Bentley [1] devised an algorithm that sweeps a vertical line from left to right. As the sweep line moves, a segment tree maintains the intersection of the sweep line with the rectangles. Whenever the sweep line passes the beginning or the end of a rectangle, the segment tree is updated. A new interval can be added, an existing interval can be removed, or merging/splitting operations can take place. At each of these points, a running total of area can be updated. Since changes to an interval tree take place in logarithmic time, each update takes time $O(\lg n)$. Since each rectangle has two vertical edges, the total number of updates is $O(n)$, and the total running time is $O(n \lg n)$. Bentley's algorithm is therefore optimal in two dimensions.

In higher dimensions, Bentley's sweepline strategy can handle a d-dimensional problem by dividing it into n $(d-1)$-dimensional measure problems; it therefore acheives a running time in three-dimensional space of $O(n^2 \lg n)$. However, this can be improved upon. In 1991, Mark Overmars and Chee Yap [6] used a data structure similar to kd-trees that contained two-dimensional boxes rather than intervals. Insertions and deletions could be performed in $O(\sqrt(n) \lg n)$ time. Therefore, the overall running time of their algorithm is $O(n^{3/2} \lg n)$. This is an open area of research, however, since the best known lower bound is still $O(n \lg n)$.

7.2 Interval Trees

7.2.1 Problem Statement

Suppose that we are given input I, a set of n intervals in $\mathbb{R}$. A query is a stabbing point q which can be expressed as a single scalar. We wish to return a list of the intervals in I that are stabbed by q.

7.2.2 Data Structure

Interval trees present an efficient solution to this problem, and were developed independently by Edelsbrunner [2] and McCreight [4] in 1980.

An interval tree is built recursively from the root down by the following procedure:

- Presort all of the interval endpoints.
- Compute the median of all endpoints, x_{mid}.
- Partition the set of segments into three disjoint sets:

$$\begin{aligned} I_{mid} &= \{[x_j, x'_j] \in I | x_j \leq x_{mid} \leq x'_j\} \\ I_{left} &= \{[x_j, x'_j] \in I | x_j, x'_j < x_{mid}\} \\ I_{right} &= \{[x_j, x'_j] \in I | x_j, x'_j > x_{mid}\} \end{aligned}$$

- Create a root node V and store two lists in it: L_{left} and L_{right}. L_{left} contains all of the items in I_{mid} sorted by left endpoint, while L_{right} contains all of the items in I_{mid} sorted by right endpoint.
- Recur on I_{left} and I_{right}. The result of the recursive call on I_{left} will become V's right child, and the result of the call on I_{right} will become V's left child.

A diagram showing the first few stages of this construction can be found in Figure 7.3.

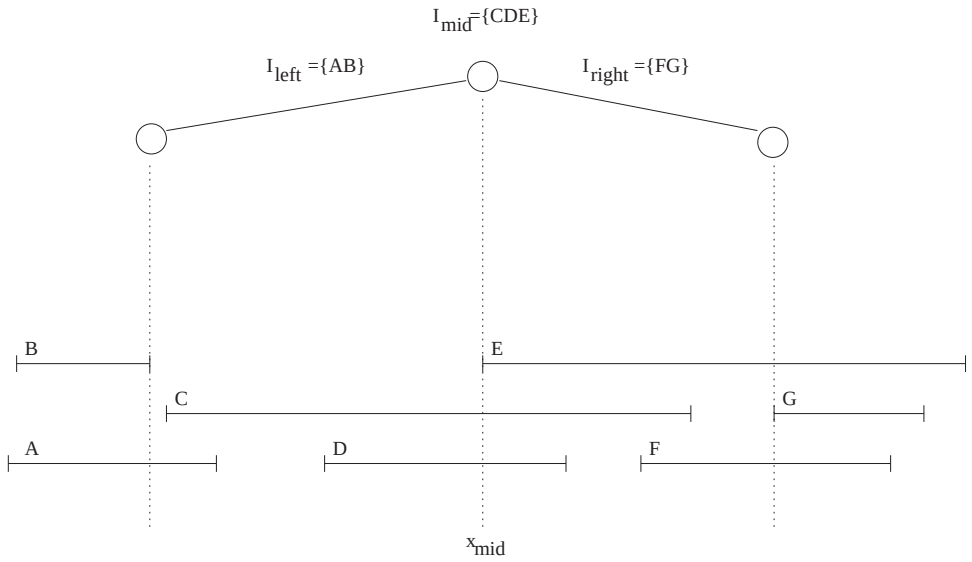

Figure 7.3: Constructing an Interval Tree

7.2.3 Time and Space Bounds

The running time of the construction of the interval tree can be broken into five parts.

1. Presort all of the interval endpoints. This requires $O(n \lg n)$ time.
2. Compute x_{mid}, which can be done in constant time once the endpoints are presorted.

3. Compute $I_{mid}, I_{left}, I_{right}$, which takes $O(n)$ time, since the endpoints of each segment must be considered before they can be assigned to a set.
4. Create L_{left} and L_{right}, which takes $O(n_{mid} \lg n_{mid})$ time, where $n_{mid} = card(I_{mid})$.
5. Recur on I_{left} and I_{right}, beginning with step 2.

Consider the time taken by the recursive steps. The second step requires constant time for each recursive call. Since the set size is divided in half at each level, the number of recursive calls is $O(n)$, and the total time for the second step is $O(n)$.

The third step is linear in the number of items that are passed down during the recursion, and no segment can be in more than one I_X set, so each level of recursion takes no more than $O(n)$ time. Over all levels, this sums to $O(n \lg n)$.

Over all recursive calls, the fourth step takes $\sum O(n_{mid} \lg n_{mid})$ time. Since $\sum n_{mid} = n$, we know that $\sum O(n_{mid} \lg n_{mid}) \leq O(n \lg n)$.

Therefore, the total preprocessing time for an interval tree is:

$$O(n \lg n) + O(n) + O(n \lg n) + O(n \lg n) = O(n \lg n)$$

At query time, compare q with the root's x_{mid}. If $q < x_{mid}$, then you know that the segments to be reported are either in I_{left} or I_{mid}. Check I_{mid} first by walking through the root's L_{left} array until you reach an interval that does not include q. All of the intervals that you examine will be stabbed by q except the last one. Therefore, the time spent at that level is $O(1 + k_v)$, where k_v is the number of intervals reported at the current node. To check I_{left}, recur on the root's left child.

If, instead, $q > x_{mid}$, walk through the root's L_{right} list and recur on the right child.

Since the tree has $O(\lg n)$ levels, and since $\sum k_v = k$, the total running time for the query is $O(\lg n + k)$.

The space required for this data structure is $O(n)$ since each interval can only be stored twice in the tree: once in L_{right} and once in the corresponding L_{left}. The sets I_{right} and I_{left} are used to build the tree, but are not explicitly stored therein.

7.3 Priority Search Tree

7.3.1 Problem Statement

Suppose that we are given input P, a set of n points in $\mathbb{R}^2$. A query is given by the triple (q_x, q_y, q'_y). The goal is to return all of the points in P that are contained in the region $(-\infty, q_x] \times [q_y, q'_y]$. An example of such a query is shown in Figure 7.4.

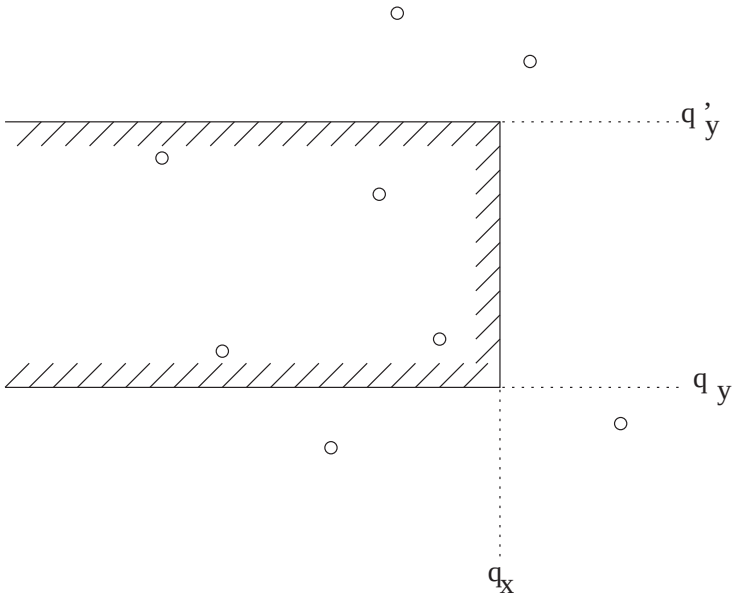

Figure 7.4: 2-D Priority Search Example

7.3.2 Data Structure

A priority search tree [5] is a data structure which can be used to solve this problem efficiently. The data strucuture at the heart of a priority search tree is the heap, an example of which is given in Figure 7.5. A heap is a binary tree structured such that the key of a parent is less than or equal to the key of its children. In many applications, it is also required that the lower level of the heap is filled from the left to the right; this is not a requirement here.

The procedure for creating a priority search tree (procedure name $createPStree(P)$) is as follows:

- Let $x_{min} \in P$ be the point with the lowest x-coordinate.
- Let $y_{mid} \in P$ be the point with the median y-coordinate.
- Create a node v with the value of x_{min} as the key, and store y_{mid} in the node as well.
- Let $P_L = \{p \in P - \{x_{min}\} | p_y \leq y_{mid}\}$.
- Let $P_R = \{p \in P - \{x_{min}\} | p_y > y_{mid}\}$.
- $v.left = createPStree(P_L)$.
- $v.right = createPStree(P_R)$.
- Return v.

The construction of such a tree is illustrated in Figure 7.6. The letter given in the node represents the x_{min} stored as the key, while the dotted line represents the median y value stored in the node.

It is very important to note that this specific construction method allows the data structure to be indexed in two different ways. First, the tree can be searched as a binary tree based on y-coordinates. However, it also operates as a heap based on x-coordinate, and therefore each subtree is also a heap on x.

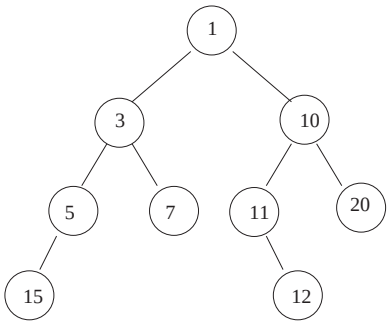

Figure 7.5: An Instance of a Heap

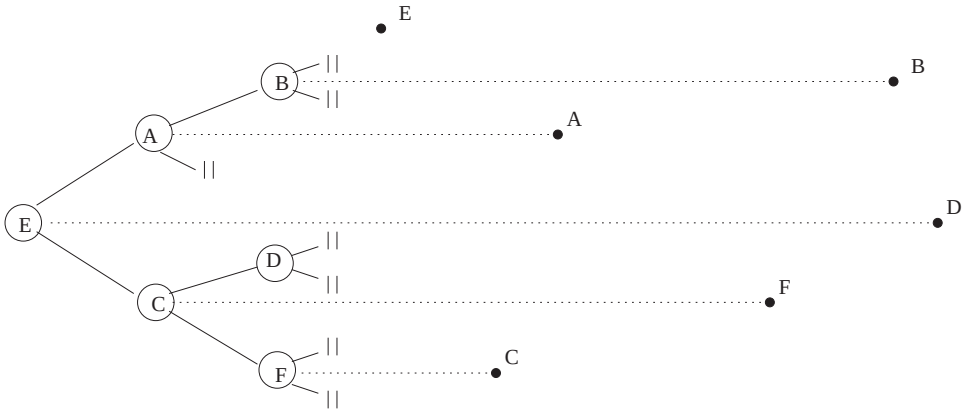

Figure 7.6: Constructing a Priority Search Tree

7.3.3 Time and Space Bounds

First let us consider the maximum depth of recursion in the construction of a priority search tree. Since we divide the input set around the median in each recursive call, we know that the size halves at every level. Therefore, the total depth of recursion is $O(\lg n)$. At each level of the recursion, it takes linear time to find the x_{min} values, linear time to find the y_{mid} values, and linear time to create P_L and P_R. Therefore, with $O(n)$ time required at each level, the total preprocessing time is $O(n \lg n)$.

At query time, we are given the triple (q_x, q_y, q'_y). A graphic describing this process is given in Figure 7.7. To respond to this query, we first perform two searches in the binary tree, one for q_y and one for q'_y. The search for q_y will follow the left-most path in the graphic, while the search for q'_y will follow the right-most. The grey subtrees, therefore, contain all the points which are in the range $[q_y, q'_y]$.

Since each of those subtrees contain a heap indexed on the values of x_{min}, the x values in the range $(-\infty, q_x]$ can be retrieved in time $O(1 + k_v)$. Here k_v is the number of items in the subtree rooted at v which are in

the range $(-\infty, q_x]$. Since there are at most $2 * \lg n$ of these subtrees to be searched, the total query time is $O(\lg n + k)$ where $k = \sum k_v$.

Since each point is stored in the priority search tree once, the number of nodes is equal to the number of points. Therefore, the total space required for this data structure is $O(n)$.

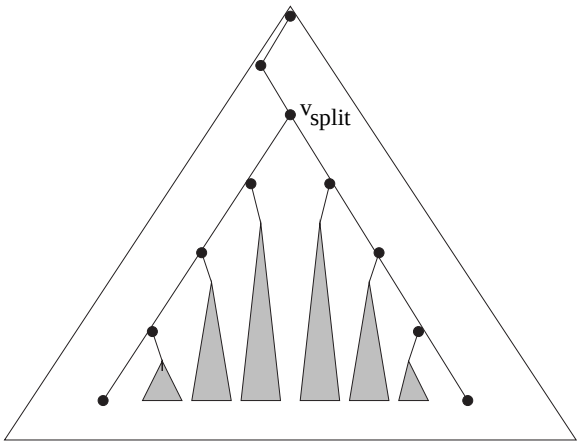

Figure 7.7: Priority Search Tree Query

References

[1] J.L. Bentley. Solutions to Klee's rectangle problems. Technical report, Carnegie-Mellon Univ., Pittsburgh, PA, 1977

[2] Dynamic data structures for orthogonal intersection queries. Report F59, Inst. Informationsverarb., Tech. Univ. Graz, Graz, Austria, 1980

[3] Michael L. Fredman and Bruce W. Weide. On the Complexity of Computing the Measure of U[ai, bi]. *Commun. ACM*, 21:540-544, 1978.

[4] E.M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Report CSL-80-9, Xerox Palo Alto Res. Center, Palo Alto, CA, 1980.

[5] E.M. McCreight. Priority Search Trees. *SIAM J. Comput.*, 14:257-276, 1985.

[6] M.H. Overmars and C.-K. Yap. New upper bounds in Klee's measure problem. *SIAM J. Comput.*, 10:460-470, 1991.