6.851: Advanced Data Structures Spring 2010

Lecture 4 — February 11, 2010

Prof. André Schulz

1 Overview

In the previous lecture, we studied range trees and kd-trees, two structures which support efficient orthogonal range queries on a set of points. In this second of four lectures on geometric data structures, we will tie up a loose end from the last lecture — handling insertions and deletions. The main topic, however, will be two new structures addressing the *vertical line stabbing* problem: interval trees and segment trees. We will conclude with an application of vertical line stabbing to a windowing problem.

2 Dynamization

In our previous discussion of range trees, we assumed that the point set was static. On the other hand, we may be interested in supporting insertions and deletions as time progresses. Unfortunately, it is not trivial to modify our data structures to handle this dynamic scenario. We can, however, dynamize them using general techniques; the techniques and their application here are due to Overmars [1].

Idea. Overmars' dynamization is based on the idea of *decomposable searches*. A search (X, q) for the value q among the keys X is decomposable if the search result can be obtained in $O(1)$ time from the results of searching (X_1, q) and (X_2, q), where $X_1 \cup X_2 = X$. For instance, in the 2D kd-tree, the root node splits $\mathbb{R}^2$ into two halves. If we want to search for all points lying in a rectangle q, we may do this by combining the results of searching for q in each half.

Results. We will bypass the actual dynamization method, and just give the results.

- *Insertion:* $O(\frac{T_b}{n} \log n)$, where T_b is the *build time*, the time required to build the static tree
- *Query:* time increases by a $\log n$ factor

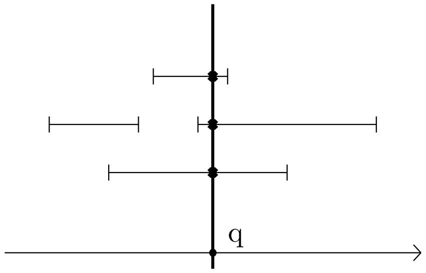
Note that this is a general result, which applies to any data structure with a decomposable search. As a concrete example, here are the times for a dynamic 2D range tree:

- *Insertion:* $O(\log^2 n + k)$, where k is the size of the output, as always
- *Query:* $O(\log^2 n)$

We have only mentioned insertion time here; deletions are more complicated.

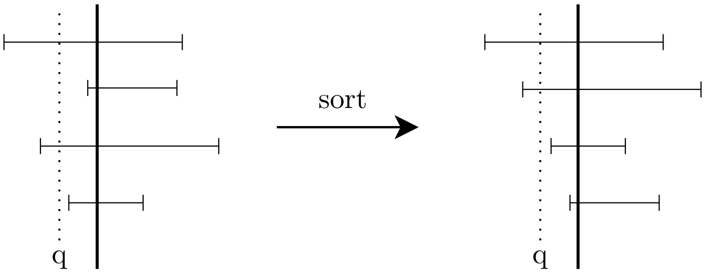
3 Vertical line stabbing

We now move to the main focus of this lecture. In the *vertical line stabbing* problem, the input is a collection $I = \{[a_1, b_1], [a_2, b_2], \dots\}$ of n closed intervals. We wish to be efficiently report, for any $q \in \mathbb{R}$, the intervals containing q. The name "vertical line stabbing" derives from one way to visualize the problem:



3.1 Interval Tree

Idea. One way to address the problem is through a divide-and-conquer approach. Consider the median m of the $2n$ endpoints in I. Some intervals may lie entirely to the left of m, some entirely to the right, and some may cross m. Those to the left and to the right can be dealt with recursively. On the other hand, the intervals crossing m seem problematic. Suppose we have a query $q < m$. We may sort the intervals crossing m by their left endpoints:

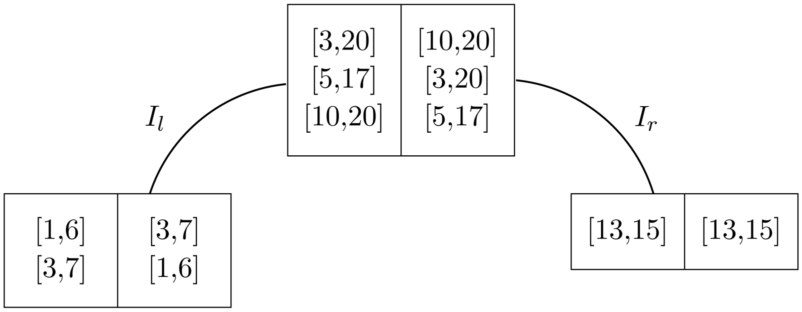


With this sorting, the intervals containing q can be listed easily, starting from the leftmost interval and stopping when we reach an interval lying to the right of q. All the intervals we examine, except for the last, contain q, so the searching process requires time linear in the output size. If q were on the right of m, instead, we would sort the intervals by their right endpoints.

Example. Take $I = \{[1, 6], [3, 20], [3, 7], [5, 17], [10, 20], [13, 15]\}$. Taking 10 as the median of the endpoints, we have three classes of intervals:

$$I_l = \{[1, 6], [3, 7]\} \qquad \text{(intervals left of median)}$$
$$I_m = \{[3, 20], [5, 17], [10, 20]\} \qquad \text{(intervals crossing median)}$$
$$I_r = \{[13, 15]\} \qquad \text{(intervals right of median)}$$

We now construct a binary tree for I, which will be its interval tree. We recursively store the interval trees for I_l and I_r in the left and right children of the root node. Then, we put two copies of I_m in the root node itself, one copy sorted by left endpoint, one by right endpoint:



To search for 18, say, we first compare it to the median 10. We can ignore I_l, since $18 > 10$. Next, we search through I_m, finding the intervals $[3, 20]$ and $[10, 20]$. Finally, we recursively search for 18 in I_r, where there are no matching intervals.

Summary. The interval tree has the following characteristics:

- *Preprocessing:* $O(n \log n)$. Sorting I_m requires $O(n \log n)$ time, and the median of $2n$ endpoints can be found in $O(n)$ time.
- *Storage:* $O(n)$. The tree structure takes up $O(n)$ space; each interval appears in only one node, and appears exactly twice there, for $O(n)$ total space.
- *Query:* $O(\log n + k)$. Searching through the intervals of each node requires time linear in the output size, contributing $O(k)$. The height of the tree is $O(\log n)$ due to the use of the median.

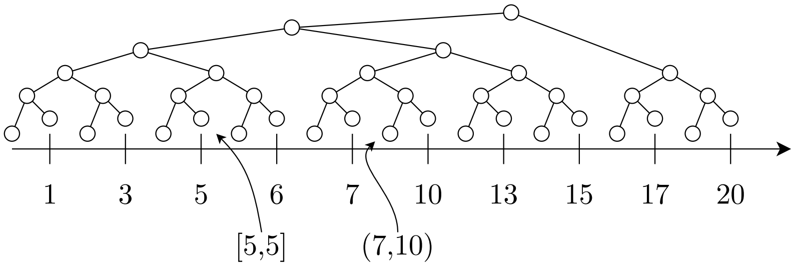
We have seen how to construct and query an interval tree:

- *Construction:*
  1. Split intervals among I_l, I_m, I_r.
  2. Handle I_m with two sorted lists.
  3. Recursively construct I_l, I_r.
- *Searching:*
  1. Perform a binary search for q.
  2. Scan the I_m lists for stabbed intervals.

3.2 Segment Tree

Construction. The second data structure we will consider for vertical line stabbing is the *segment tree.* We start by defining the construction of a segment tree, with the same I used earlier as an example. The steps are as follows:

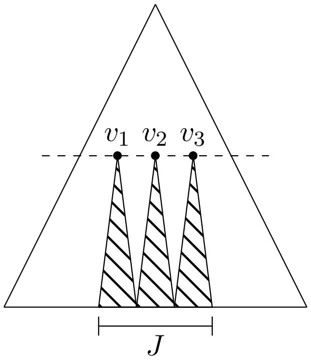
- First, we split the real line into *elementary intervals.* If $\{c_1, \ldots, c_k\}$ is the set of endpoints in I in increasing order, the elementary intervals are the single points $[c_i, c_i]$ together with the open intervals between them: (c_i, c_{i+1}).
- We build a BST over the elementary intervals. Note that each node in the BST can be associated with the union of the elementary intervals belonging to the leaf nodes below it.



- As a first try, we store in each leaf node (elementary interval) the set of intervals in I containing that node's interval. For instance, at the node for the $(3, 5)$, we store $\{[1, 6], [3, 7], [3, 20]\}$. The query time is good: $O(\log n)$ for searching and $O(k)$ for reporting. However, the storage required could be quadratic.
- To improve the storage costs, we look to see if any two siblings store the same interval $[a_i, b_i]$. If so, we remove it from them and store it in their parent instead, and repeat until no two siblings store the same intervals.
- After this process, the set of nodes storing any interval will subdivide that interval. In our example, $[3, 7]$ will be stored in the nodes [3], (3, 5], and (5, 7]. This provides a canonical subdivision for each interval in I.

Querying. To query this tree for q, we simply perform a binary search for q and report all intervals stored in the nodes we visit. This is quite different from the interval tree, where it was necessary to search through a list of potentially stabbed intervals. Here, every interval encountered is reported. The time required is $O(\log n + k)$, just as with interval trees.

Storage. We claim that every level of the tree stores any interval at most twice. The proof is by contradiction. Suppose that some interval J is stored in at least three vertices $v_1, \ldots, v_k$ in some level, as depicted below.



There cannot be any gaps between the subtrees rooted at each of these vertices, since J covers all of the real line between them. Therefore, some pair v_i,v_{i+1} must be siblings, a contradiction.

This claim shows that the segment tree requires $O(n \log n)$ storage. Although this seems worse than the interval tree, the segment tree has advantages for certain applications, as one of the homework problems shows.

Preprocessing. First, we build the BST, which takes $O(n)$ time. We can streamline the process of storing the intervals in the tree by inserting them one at a time. As an example, to insert the interval $[3, 7]$, we search for 3 and 7. At some node v_{split}, the search paths for 3 and 7 diverge. Continuing from v_{split} to $[3, 3]$, we can easily decide at each node whether $[3, 7]$ should be stored (we will not worry about the details here). Similarly for the path from v_{split} to $[7, 7]$.

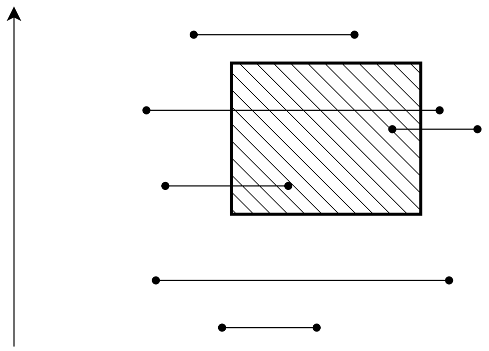
Altogether, storing the intervals requires $O(n \log n)$ time.

4 Application: windowing

Suppose we have a geometric data set containing a huge number of line segments, and want only the portion of this data lying within some window. For instance, we may have a state road map and need to a small portion of it for viewing in a GPS. We need to report all line segments that intersect W. There are two parts to this: finding everything with 1 or 2 endpoints inside W (part A), and finding everything that intersects the border of W twice (part B).

Part A. This is really a range query, so we can use a 2D orthogonal range tree on the endpoints. We need to take care if a line segment has 2 endpoints in W, to avoid reporting line segments twice.

Part B. For simplicity, we consider the case of orthogonal line segments (every segment is either horizontal or vertical). The next figure illustrates that we can check for horizontal lines crossing W using vertical line stabbing:



However, we need to get rid of the intervals lying above or below W. We will consider two approaches to doing so.

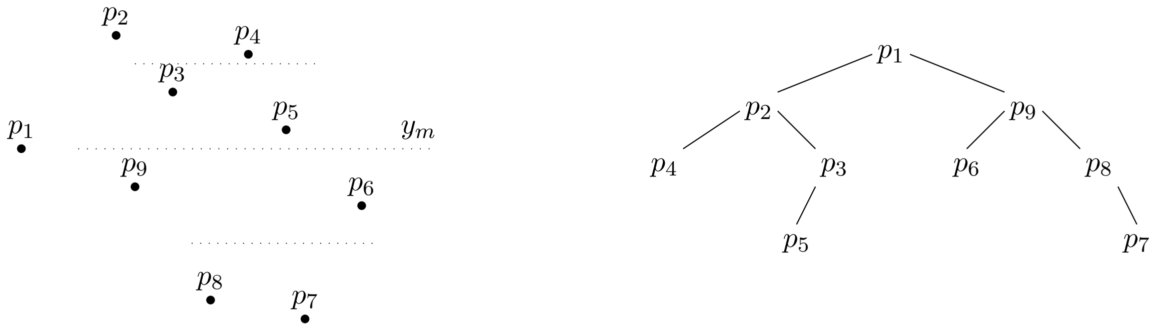
1. We form an interval tree for the horizontal lines, but store a 2D range tree with their y-values instead of the pair of sorted lists. Queries only require $O(\log^2 n + k)$ time, but the storage requirements are bad.

2. We use a priority search tree, described next.

4.1 Priority search trees

A priority search tree is a mixture of a BST and a heap. Like 2D range trees, these trees represent a 2D point set $\{p_1, p_2, \ldots\}$. Unlike 2D range trees, the queries for a priority search tree are rectangles with one open side: $(-\infty, x_{\max}] \times [y_{\min}, y_{\max}]$.

Construction. We store a point with least x-coordinate in the root of the tree, and let y_m be the median of the remaining y-coordinates. In the left subtree, we store points with $y < y_m$ in a priority search tree, and in the right subtree, the points with $y \geq y_m$. The next picture provides an example of this construction.



Queries. We search for $y_{\min}$, $y_{\max}$. Intuitively, we want to check all points along both search paths to see if they lie in the query region. More precisely, if v_{split} is the point where the two paths diverge, we go from v_{split} to $y_{\min}$, reporting all right subtrees recursively, and stopping if we reach a node with x-coordinate greater than $x_{\max}$.

A priority search tree has the following characteristics:

- *Preprocessing:* $O(n \log n)$ to sort the x and y coordinates
- *Storage:* $O(n)$
- *Query:* $O(\log n + k)$

As indicated earlier, we can modify an interval tree to store a priority search tree in each node, instead of sorted lists. This structure solves Part B for orthogonal line segments. If the line segments are not orthogonal, we can use segment trees instead (homework).

References

[1] M. H. Overmars, *The design of dynamic data structures*, Lecture Notes in Comput. Sci., 156, Springer-Verlag, Berlin, 1983.

[2] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars, *Computational Geometry: Algorithms and Applications*, third edition, Springer-Verlag, 2008. ISBN 978-3-540-77973-5. Or, second revised edition, Springer-Verlag, 2000. ISBN 3-540-65620-0.

6.851 Advanced Data Structures
Spring 2010