10 Range Trees and Segment Trees

A *range searching* problem asks us to preprocess a set X of geometric objects, so that later, given a query object q of some fixed type, we can compute some fixed function of all objects in X that share some geometric relationship with q. For example:

- X is a set of points in the plane, q is another point in the plane, and we want to know if any point in X is both above and to the right of q.
- X is a set of points in the plane, q is an axis-aligned rectangle, and we want the number of points in X that are contained in q.
- X is a set of axis-aligned rectangles in the plane, each with a priority, q is a single point, and we want the minimum-priority rectangle in X that contains q.
- X is a set of horizontal line segments in the plane, q is a vertical line segment, and we want the number of segments in X that intersect q.
- X is a set of spheres in 3-space, q is a line in 3-space, and we want the smallest sphere in X that intersects q.

In this lecture I'll describe some basic ingredients that can be combined to build data structures for *orthogonal range searching* problems, where both the stored objects and the query object are products of one-dimensional points and intervals. (All of these examples above *except the last one* are orthogonal range searching problems; the last example is also a *geometric* range searching problem, but solving it requires very different techniques than described here.)

10.1 Range Trees

We've already seen one canonical range-searching data structure, called the *range tree*. Here we assume X is a set of points on the real line (that is, real numbers), each query object q is a closed interval $[l, r]$, and we are interested in some function of the points in $X \cap q$. Let's initially assume that we want the *number* of points in $X \cap q$.

A range tree is a balanced binary search tree whose *leaves* store the points in X in sorted order from left to right; this is sometimes also called an *exogenous* binary search tree.[1] Specifically, each leaf ℓ in T stores a unique point $\ell.point \in X$, and each interior node stores three values:

- $v.pivot$ is a pivot value for v, used for searching.
- $v.size$ is the number of points/vertices in the subtree rooted at v.
- $v.min$ is the smallest point value in the subtree rooted at v.
- $v.max$ is the largest point value in the subtree rooted at v.

The interval $[v.min, v.max]$ is called the *canonical range* associated with v. In all examples, I will use the pivot value $v.pivot = v.left.max$.

The following algorithm answers any range-counting query in $O(\log n)$ time, essentially by decomposing the query range $[l, r]$ into $O(\log n)$ canonical ranges. (The leaf base case is using Iverson bracket notation; the expression is equal to 1 if the inequalities hold and 0 otherwise.)

[1] We can also build range trees from standard binary search trees, which associate a point in X with every vertex, but these lead to annoying corner cases that I'd rather avoid.

```
COUNT(v, l, r):
  if v is a leaf
      return [l ≤ v.point ≤ r]
  else if l > v.max or r < v.min      ⟨⟨canonical range outside the query range⟩⟩
      return 0
  else if l ≤ v.min and r ≥ v.max     ⟨⟨canonical range inside the query range⟩⟩
      return v.size
  else
      return COUNT(v.left, l, r) + COUNT(v.right, l, r)
```

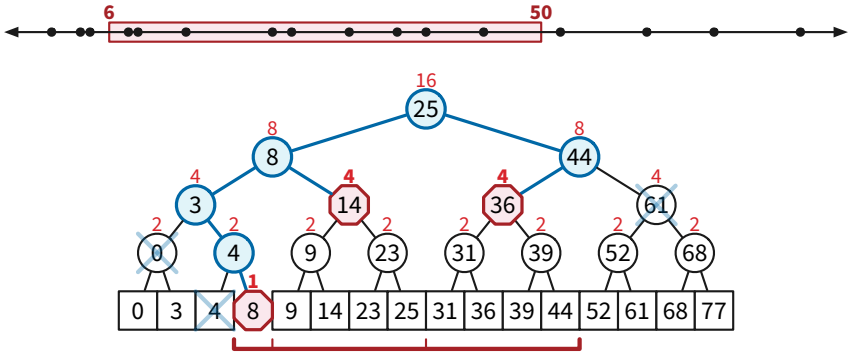

Figure 1. Answering a range-counting query using a range tree; the indicated range contains $1 + 4 + 4 = 9$ points.

We can easily adapt this structure to other types of queries. For example, suppose every point in X has a weight. If each leaf ℓ stores the weight of its point as $\ell.wt$, and we redefine $v.size$ to be the total weight of all points in v's subtree, then the following algorithm computes the total weight of all points in a query range in $O(\log n)$ time. Only the second and third line of this algorithm are different from COUNT.

```
TOTALWT(v, l, r):
  if v is a leaf
      if l ≤ v.point ≤ r
          return v.wt
  else if l > v.max or r < v.min      ⟨⟨disjoint⟩⟩
      return 0
  else if l ≤ v.min and r ≥ v.max     ⟨⟨contained⟩⟩
      return v.size
  else
      return TOTALWT(v.left, l, r) + TOTALWT(v.right, l, r)
```

We can print the complete list of all k points in the query range in $O(\log n + k)$ time as follows:

```
REPORT(v, l, r):
  if v is a leaf
      if l ≤ v.point ≤ r
          print v.point
  else if l > v.max or r < v.min      ⟨⟨disjoint⟩⟩
      do nothing
  else if l ≤ v.min and r ≥ v.max     ⟨⟨contained⟩⟩
      for every leaf ℓ below v
          print ℓ.point
  else
      REPORT(v.left, l, r)
      REPORT(v.right, l, r)
```

Finally, suppose every point in x has a priority, which we store in the corresponding leaf ℓ as $\ell.prior$, and we store the minimum priority in v's subtree in $v.best$. Then we can find the minimum-priority point in any query range in $O(\log n)$ time as follows:

```
MINIMUM(v, l, r):
  if v is a leaf
      if l ≤ v.point ≤ r
          return v.prior
      else
          return ∞
  else if l > v.max or r < v.min      ⟨⟨disjoint⟩⟩
      return ∞
  else if l ≤ v.min and r ≥ v.max     ⟨⟨contained⟩⟩
      return v.best
  else
      return min{MINIMUM(v.left, l, r), MINIMUM(v.right, l, r)}
```

10.2 Multidimensional Queries

Now suppose P is a set of points in the plane, each specified by a pair (x, y) of real numbers, and we want to query for the number of points in an axis-aligned rectangle $[l, r] \times [b, t]$. We can think of this problem as a one-dimensional range-query problem, where the function we want to compute for the points in a canonical range is another one-dimensional range-query problem!

Let $T_x(P)$ denote a range tree over the x-coordinates of points in P. For each node v in this tree, let P_v denote the *canonical subset* of points whose x-coordinates are stored in the subtree rooted at v. To support two-dimensional queries, we construct a ***secondary data structure*** $T_y(P_v)$ for each canonical subset P_v. This secondary data structure is another range tree, this time over the y-coordinates of the subset P_v. In each primary node v, we store a pointer $v.ytree$ to the root of the corresponding secondary structure $T_y(P_v)$.

Each secondary data structure $T_y(P_v)$ uses $O(|P_v|)$ space. Each point in X appears in a most one canonical subset at each level of the primary tree. Thus, assuming the primary tree T_x is balanced, each point in X appears in at most $O(\log n)$ secondary structures. It follows that the entire two-level data structure uses ***$O(n \log n)$ space***.

The top-level query algorithm is almost identical to the one-dimensional case.

```
BoxXCount(v, l, r, b, t):
  if v is a leaf
      return [l ≤ v.x ≤ r and b ≤ v.y ≤ t]
  else if l > v.maxx or r < v.minx        ⟨⟨canonical x-range outside query x-range⟩⟩
      return 0
  else if l ≤ v.minx and r ≥ v.maxx       ⟨⟨canonical x-range inside query x-range⟩⟩
      return BoxYCount(v.ytree, b, t)
  else
      return BoxXCount(v.left, l, r, b, t) + BoxXCount(v.right, l, r, b, t)
```

In the case where the query x-range $[l, r]$ contains the canonical x-range $[v.minx, v.maxx]$, we call another standard range-query algorithm on the secondary data structure at v:

```
BoxYCount(w, b, t):
  if w is a leaf
      return [b ≤ w ≤ t]
  else if b > w.maxy or t < w.miny        ⟨⟨canonical y-range outside query y-range⟩⟩
      return 0
  else if b ≤ w.miny and t ≥ w.maxy       ⟨⟨canonical y-range inside query y-range⟩⟩
      return v.size
  else
      return BoxYCount(w.down, b, t) + BoxYCount(w.up, b, t)
```

Answering a single box query requires at most two secondary queries at each level of the primary tree. The time to answer a secondary query at primary node v is at most $O(\log|P_v|)$, which is conservatively at most $O(\log n)$. So assuming the primary tree is balanced, the total worst-case query time is conservatively at most $\boldsymbol{O(\log^2 n)}$.[2]

More generally, the primary tree decomposes the x-projection of the query rectangle q into $O(\log n)$ canonical x-ranges, and each secondary tree decomposes the intersection of q with one canonical x-range into $O(\log n)$ canonical rectangles, such that a point $p \in P$ lies inside q if and only if p lies in one of these canonical rectangles. See Figure 3.

This construction generalizes to any number of dimensions. A d-dimensional range tree is a 1-dimensional range tree over one coordinate of the points, where each node v stores a $(d-1)$-dimensional range tree over the remaining $d-1$ coordinates of the canonical subset P_v. The overall data structure uses $\boldsymbol{O(n \log^{d-1} n)}$ ***space***, and answers orthogonal range queries in $\boldsymbol{O(\log^d n)}$ ***time***.

10.3 Segment Trees

Now let's turn the range-tree problem on its head. Instead of preprocessing a set of points for range queries, we now want to preprocess *intervals* for *point* queries. Specifically, we are given a set S of real intervals, each specified by its left and right boundary points $s.l$ and $s.r$. We want to build a data structure for S so that later, we can compute some fixed function of the subset of intervals $\{s \in S \mid q \in s\}$ that contain a given query point q.

[2]This conservative upper bound turns out to be tight, even if the primary tree is *perfectly* balanced. In a perfectly balanced tree, each node at height h is the root of a subtree with $2^h - 1$ nodes. So a secondary query at height h takes $\Theta(\log(2^h - 1)) = \Theta(h)$ time in the worst case. Our primary tree has depth $D = \lfloor \lg n \rfloor$, so our overall query algorithm takes $\sum_{h=0}^{D} \Theta(h) = \Theta(D^2) = \Theta(\log^2 n)$ time in total.

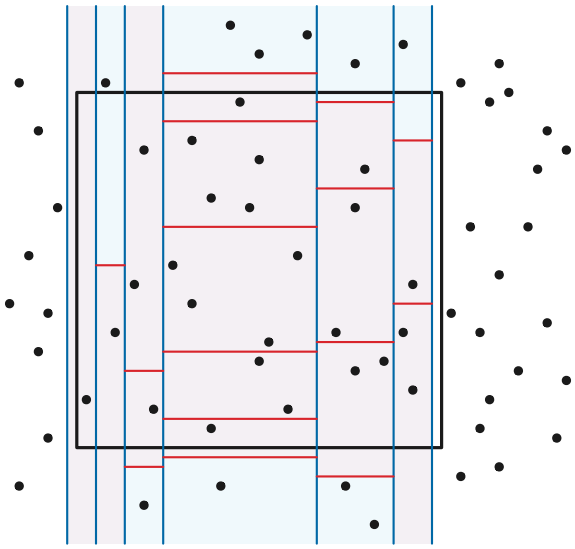

Figure 2. Subdividing the points inside a query rectangle into $O(\log^2 n)$ canonical rectangles.

A ***segment tree*** for S is a balanced binary search tree T over the endpoints of intervals in S, with some additional information stored at each vertex. (To avoid confusion, I'll refer to the elements of S as *segments*.) ⟨⟨**Interior nodes correspond to endpoints; leaves to gaps between consecutive endpoints.**⟩⟩ Recall from the previous section that T partitions every interval $[l, r]$ into $O(\log n)$ canonical intervals, with at most two at each level of T. For each node v, let S_v denote the set of segments in S whose canonical partition includes the canonical range of v.

Each node v in a segment tree stores a fixed function of the set S_v that depends on the query type. For example, suppose we want to ask for the number of segments that contain a query point q. Then every node in our segment tree stores the following information:

- $v.pivot$ is a pivot value, used for searching.
- $v.min$ is the smallest endpoint value in the subtree rooted at v.
- $v.max$ is the largest endpoint value in the subtree rooted at v.
- $v.size$ is the number of segments in S_v.

The total size of this data structure is $O(n)$.

The following algorithm returns the number of segments containing a query point q in $O(\log n)$ time; the algorithm is almost identical to a standard binary search for q:

```
StabCount(v, q):
  if v = Null
      return 0
  if q < v.pivot
      return v.size + StabCount(v.left, q)
  else
      return v.size + StabCount(v.right, q)
```

Just like range trees, segment trees can be easily adapted to other query functions, with only minimal changes to the data structure or the query algorithm. For example, if segments have weights and we want the total weight of all segments containing q, we store the total weight

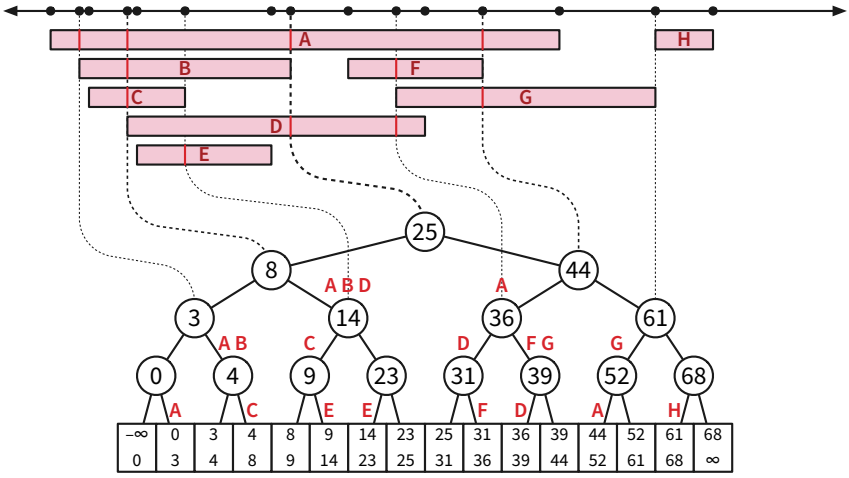

Figure 3. A segment tree for eight segments. Numbers at the leaves are the left and right endpoints of the gaps between segment endpoints. Segments in each set S_v are listed directly above the corresponding node v.

of all segments in S_v in $v.size$ and use exactly the same query algorithm. If each segment has a priority, and we want the minimum-priority segment containing q, we store the minimum priority among all segments in S_v in $v.best$, and we replace "$v.size + \ldots$" in the query algorithm with "$\min\{v.best, \ldots\}$".

10.4 Multidimensional Stabbing Queries

Also just like range trees, segment trees can be generalized to higher dimensions. For example, suppose we are given a set of R of axis-aligned rectangles in the plane, each specified by left, right, bottom, and top coordinates, and we want to support queries that ask for the number of rectangles that contain a query point. We build a two-level data structure as follows.

The primary data structure is a segment tree over the x-projections of R. For each node v in the primary tree, we build a secondary segment tree $v.ytree$ over the y-projections of the rectangles in the subset R_v. The secondary tree uses $O(|R_v|)$ space, and every rectangle appears in $O(\log n)$ subsets R_v, so the overall data structure uses ***$O(n \log n)$ space***. We can answer point-stabbing queries using the following two-phase algorithm:

```
StabX(v, q):
  if v = Null
      return 0
  if q.x < v.pivotx
      return StabY(v.ytree, q) + StabX(v.left, q)
  else
      return StabY(v.ytree, q) + +StabX(v.right, q)
```

```
StabY(w, q):
  if w = Null
      return 0
  if q.y < w.pivoty
      return w.size + StabY(v.down, q)
  else
      return w.size + StabY(v.up, q)
```

Each primary query performs $O(\log n)$ secondary queries, each of which takes $O(\log n)$ time, so the overall algorithm takes ***$O(\log^2 n)$ time***.

Again, this construction generalizes to any number of dimensions. A d-dimensional segment tree stores n d-dimensional boxes in $O(n \log^{d-1} n)$ space and answers point-stabbing queries in $O(\log^d n)$ time.