# Lecture 20: Range Trees

**Reading:** Samet's book, Section 2.5.

**Range Queries:** Last time we saw how to use kd-trees to solve orthogonal range queries. Recall that we are given a set of points in the plane, and we want to count or report all the points that lie within a given axis-aligned rectangle. We argued that if the tree is balanced, then the running time is close to $O(\sqrt{n})$ to count the points in the range. If there are $k$ points in the range, then we can list them in additional $O(k)$ time for a total of $O(k + \sqrt{n})$ time to answer range reporting queries.

Although $\sqrt{n}$ is much smaller than $n$, it is still larger than $\log n$. Is this the best we can do? Today we will present a data structure called a range tree which can answer orthogonal counting range queries in $O(\log^2 n)$. (Recall that $\log^2 n$ means $(\log n)^2$). If there are $k$ points in the range it can also report these points in $O(k + \log^2 n)$ time. It uses $O(n \log n)$ space, which is somewhat larger than the $O(n)$ space used by kd-trees. (There are actually two versions of range trees. We will present the simpler version. There is a significantly more complex version that can answer queries in $O(k + \log n)$ time, thus shaving off a log factor in the running time.) The data structure can be generalized to higher dimensions. In dimension $d$ it answers range queries in $O(\log^d n)$ time.

**Range Trees (Basics):** The range tree data structure works by reducing an orthogonal range query in 2-dimensions to a collection of $O(\log n)$ range queries in 1-dimension, then it solves each of these in $O(\log n)$ time, for a combined time of $O(\log^2 n)$. (In higher dimensions, it reduces a range query in dimension $d$ to $O(\log n)$ range queries in dimension $d-1$.) It works by a technique called a *multi-level tree*, which involves cascading multiple data structures together in a clever way. Throughout we assume that a range is given by a pair of points $[lo, hi]$, and we wish to report all points $p$ such that

$$lo_x \leq p_x \leq hi_x \qquad \text{and} \qquad lo_y \leq p_y \leq hi_y.$$

**1-dimensional Range Tree:** Before discussing 2-dimensional range trees, let us first consider what a 1-dimensional range tree would look like. Given a set of points $S$, we want to preprocess these points so that given a 1-dimensional interval $Q = [lo, hi]$ along the $x$-axis, we can count all the points that lie in this interval. There are a number of simple solutions to this, but we will consider a particular method that generalizes to higher dimensions.

Let us begin by storing all the points of our data set in the external nodes (leaves) of a balanced binary tree sorted by $x$-coordinates (e.g., an AVL tree). The data values in the internal nodes will just be used for searching purposes. They may or may not correspond to actual data values stored in the leaves. We assume that if an internal node contains a value $x_0$ then the leaves in the left subtree are strictly less than $x_0$, and the leaves in the right subtree are greater than or equal to $x_0$. Each node $t$ in this tree is implicitly associated with a subset $S(t) \subseteq S$ of elements of $S$ that are in the leaves descended from $t$. (For example $S(root) = S$.) We assume that for each node $t$, we store the number of leaves that are descended from $t$, denoted `t.size`. Thus `t.size` is equal to the number of elements in $S(t)$.

Let us introduce a few definitions before continuing. Given the interval $Q = [lo, hi]$, we say that a node $t$ is *relevant* to the query if $S(t) \subseteq Q$. That is, all the descendents of $t$ lie within the interval. If $t$ is relevant then clearly all of the nodes descended from $t$ are also relevant. A relevant node $t$ is *canonical* if $t$ is relevant, but its parent it not. The canonical nodes are the roots of the maximal subtrees that are contained within $Q$. For each canonical node $t$, the

subset $S(t)$ is called a *canonical subset*. Because of the hierarchical structure of the tree, it is easy to see that the canonical subsets are disjoint from each other, and they cover the interval $Q$. In other words, the subset of points of $S$ lying within the interval $Q$ is equal to the disjoint union of the canonical subsets. Thus, solving a range counting query reduces to finding the canonical nodes for the query range, and returning the sum of their sizes.

We claim that the canonical subsets corresponding to any range can be identified in $O(\log n)$ time from this structure. Intuitively, given any interval $[lo, hi]$, we search the tree to find the leftmost leaf $u$ whose key is greater than or equal to $lo$ and the rightmost leaf $v$ whose key is less than or equal to $hi$. Clearly all the leaves between $u$ and $v$ (including $u$ and $v$) constitute the points that lie within the range. Since these two paths are of length at most $O(\log n)$, there are at most $O(2 \log n)$ such trees possible, which is $O(\log n)$. To form the canonical subsets, we take the subsets of all the *maximal subtrees* lying between $u$ and $v$. This is illustrated in the following figure.

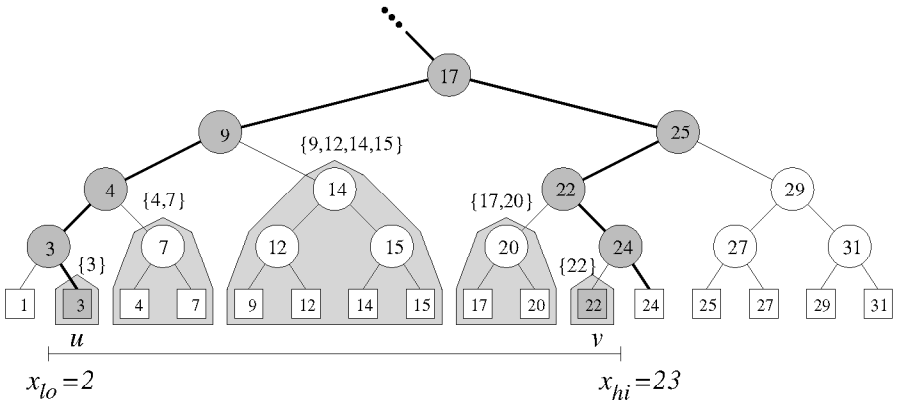


Figure 59: Canonical sets for interval queries.

There are a few different ways to map this intuition into an algorithm. Our approach will be modeled after the approach used for range searching in kd-trees. We will maintain for each node a *cell* $C$, which in this 1-dimensional case is just an interval $[C_{lo}, C_{hi}]$. As with kd-trees, the cell for node $t$ contains all the points in $S(T)$.

The arguments to the procedure are the current node, the range $Q$, and the current cell. Let $C_0 = [-\infty, +\infty]$ be the initial cell for the root node. The initial call is `range1D(root, Q, C0)`. Let $t.x$ denote the key associated with $t$. If $C = [x_0, x_1]$ denotes the current interval for node $t$, then when we recurse on the left subtree we trim this to the interval $[x_0, t.x]$ and when we recurse on the right subtree we trim the interval to $[t.x, x_1]$. We assume that given two ranges $A$ and $B$, we have utility functions `A.contains(B)` which determined whether interval $A$ contains interval $B$, and there is a similar function `A.contains(x)` that determines whether point $x$ is contained within $A$.

Since the data are only stored in the leaf nodes, when we encounter such a node we consider whether it lies in the range and count it if so. Otherwise, observe that if $t.x \leq Q.lo$ then all the points in the left subtree are less than the interval, and hence it does not need to be visited. Similarly if $t.x > Q.hi$ then the right subtree does not need to be visited. Otherwise, we need to visit these subtrees recursively.

1-Dimensional Range Counting Query

```
int range1Dx(Node t, Range Q, Interval C=[x0,x1]) {
```

```
    if (t.isExternal)                              // hit the leaf level?
        return (Q.contains(t.point) ? 1 : 0)       // count if point in range
    if (Q.contains(C))                             // Q contains entire cell?
        return t.size                              // return entire subtree size
    int count = 0
    if (t.x > Q.lo)                                // overlap left subtree?
        count += range1Dx(t.left, Q, [x0, t.x]) // count left subtree
    if (t.x <= Q.hi)                               // overlap right subtree?
        count += range1Dx(t.right, Q, [t.x, x1])// count right subtree
    return count
}
```

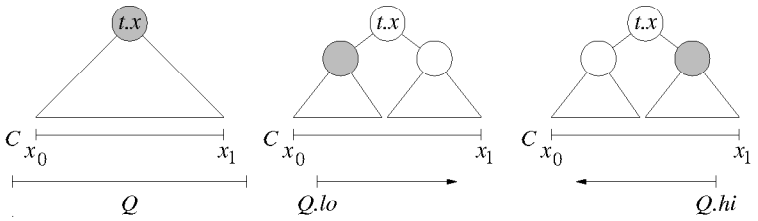


Figure 60: Range search cases.

The external nodes counted in the second line and the internal nodes for which we return `t.size` are the canonical nodes. The above procedure answers range counting queries. To extend this to range reporting queries, we simply replace the step that counts the points in the subtree with a procedure that traverses the subtree and prints the data in the leaves. Each tree can be traversed in time proportional to the number of leaves in each subtree. Combining the observations of this section we have the following results.

**Lemma:** Given a 1-dimensional range tree and any query range $Q$, in $O(\log n)$ time we can compute a set of $O(\log n)$ canonical nodes $t$, such that the answer to the query is the disjoint union of the associated canonical subsets $S(t)$.

**Theorem:** 1-dimensional range counting queries can be answered in $O(\log n)$ time and range reporting queries can be answered in $O(k + \log n)$ time, where $k$ is the number of values reported.

**Range Trees:** Now let us consider how to answer range queries in 2-dimensional space. We first create 1-dimensional tree $T$ as described in the previous section sorted by the $x$-coordinate. For each internal node $t$ of $T$, recall that $S(t)$ denotes the points associated with the leaves descended from $t$. For each node $t$ of this tree we build a 1-dimensional range tree for the points of $S(t)$, but sorted on $y$-coordinates. This called the *auxiliary tree* associated with $t$. Thus, there are $O(n)$ auxiliary trees, one for each internal node of $T$. An example of such a structure is shown in the following figure.

Notice that there is duplication here, because a given point in a leaf occurs in the point sets associated with each of its ancestors. We claim that the total sizes of all the auxiliary trees is $O(n \log n)$. To see why, observe that each point in a leaf of $T$ has $O(\log n)$ ancestors, and so each point appears in $O(\log n)$ auxiliary trees. The total number of nodes in each auxiliary tree is proportional to the number of leaves in this tree. (Recall that the number of internal nodes in an extended tree one less than the number of leaves.) Since each of the $n$ points appears as a leaf in at most $O(\log n)$ auxiliary trees, the sum of the number of leaves in all the auxiliary trees is at most $O(n \log n)$. Since $T$ has $O(n)$ nodes, the overall total is $O(n \log n)$.

Now, when a 2-dimensional range is presented we do the following. First, we invoke a variant of the 1-dimensional range search algorithm to identify the $O(\log n)$ canonical nodes. For each such node $t$, we know that all the points of the set lie within the $x$ portion of the range, but not necessarily in the $y$ part of the range. So we search the 1-dimensional auxiliary range and return a count of the resulting points. The algorithm below is almost identical the previous one, except that we make explicit reference to the $x$-coordinates in the search, and rather than adding `t.size` to the count, we invoke a 1-dimensional version of the above procedure using the $y$-coordinate instead. Let `Q.x` denote the $x$-part of $Q$'s range, consisting of the interval `[Q.lo.x,Q.hi.x]`. The call `Q.contains(t.point)` is applied on both coordinates, but the call `Q.x.contains(C)` only checks the $x$-part of $Q$'s range. The algorithm is given below. The procedure `range1Dy()` is the same procedure described above, except that it searches on $y$ rather than $x$.

2-Dimensional Range Counting Query

```
int range2D(Node t, Range2D Q, Range1D C=[x0,x1]) {
    if (t.isExternal)                           // hit the leaf level?
        return (Q.contains(t.point) ? 1 : 0)    // count if point in range
    if (Q.x.contains(C)) {                      // Q's x-range contains C
        [y0,y1] = [-infinity, +infinity]        // initial y-cell
        return range1Dy(t.aux.root, Q, [y0, y1])// search auxiliary tree
    }
    int count = 0
    if (t.x > Q.lo.x)                           // overlap left subtree?
        count += range2D(t.left, Q, [x0, t.x])  // count left subtree
    if (t.x <= Q.hi.x)                          // overlap right subtree?
        count += range2D(t.right, Q, [t.x, x1]) // count right subtree
    return count
}
```

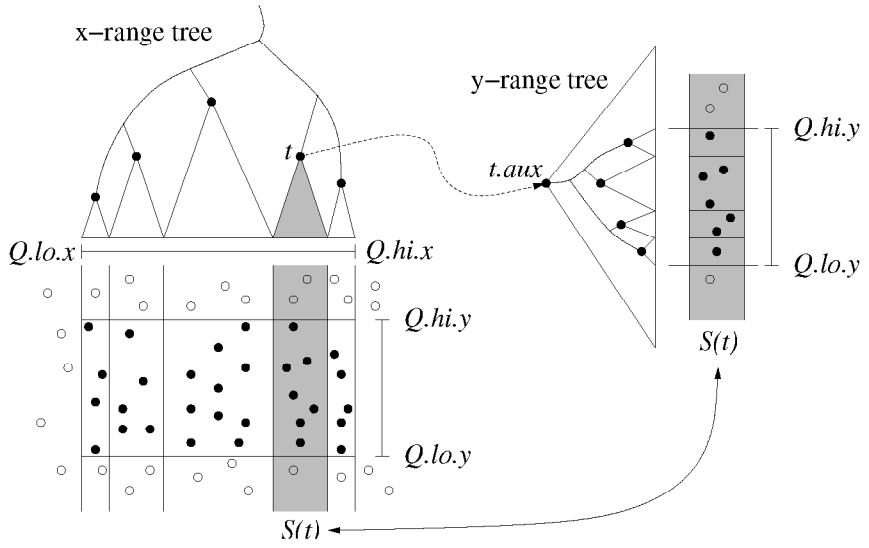


Figure 61: Range tree.

**Analysis:** It takes $O(\log n)$ time to identify the canonical nodes along the $x$-coordinates. For each of these $O(\log n)$ nodes we make a call to a 1-dimensional range tree which contains no more

than $n$ points. As we argued above, this takes $O(\log n)$ time for each. Thus the total running time is $O(\log^2 n)$. As above, we can replace the counting code with code in `range1Dy()` with code that traverses the tree and reports the points. This results in a total time of $O(k+\log^2 n)$, assuming $k$ points are reported.

Thus, each node of the 2-dimensional range tree has a pointer to a auxiliary 1-dimensional range tree. We can extend this to any number of dimensions. At the highest level the $d$-dimensional range tree consists of a 1-dimensional tree based on the first coordinate. Each of these trees has an auxiliary tree which is a $(d-1)$-dimensional range tree, based on the remaining coordinates. A straightforward generalization of the arguments presented here show that the resulting data structure requires $O(n \log^d n)$ space and can answer queries in $O(\log^d n)$ time.

**Theorem:** $d$-dimensional range counting queries can be answered in $O(\log^d n)$ time, and range reporting queries can be answered in $O(k + \log^d n)$ time, where $k$ is the number of values reported.