CMSC 754: Lecture 15
Orthogonal Range Trees

Reading: Chapter 5 in the 4M's.

Orthogonal Range Trees: In the previous lecture we saw that kd-trees could be used to answer orthogonal range queries in the plane in $O(\sqrt{n})$ time for counting and $O(\sqrt{n}+k)$ time for reporting. It is natural to wonder whether we can replace the $O(\sqrt{n})$ term with something closer to the ideal query time of $O(\log n)$. Today we consider a data structure, which is more highly tuned to this particular problem, called an *orthogonal range tree*. Recall that we are given a set P of n points in $\mathbb{R}^2$, and our objective is to preprocess these points so that, given any axis-parallel rectangle Q, we can count or report the points of P that lie within Q efficiently.

An orthogonal range tree is a data structure which, in the plane uses $O(n \log n)$ space and can answer range reporting queries in $O(\log n + k)$ time, where k is the number of points reported. In general in dimension $d \geq 2$, it uses $O(n \log^{(d-1)} n)$ space, and can answer orthogonal rectangular range queries in $O(\log^{(d-1)} n + k)$ time. The preprocessing time is the same as the space bound. We will present the data structure in two parts, the first is a version that can answer queries in $O(\log^2 n)$ time in the plane, and then we will show how to improve this in order to strip off a factor of $\log n$ from the query time. The generalization to higher dimensions will be straightforward.

Multi-layered Search Trees: The orthogonal range-tree data structure is a nice example of a more general concept, called a *multi-layered search tree*. In this method, a complex search is decomposed into a constant number of simpler range searches. Recall that a range space is a pair $(X, \mathcal{R})$ consisting of a set X and a collection $\mathcal{R}$ of subsets of X, called *ranges*. Given a range space $(X, \mathcal{R})$, suppose that we can decompose it into two (or generally a small number of) range subspaces $(X, \mathcal{R}_1)$ and $(X, \mathcal{R}_2)$ such that any query $Q \in \mathcal{R}$ can be expressed as $Q_1 \cap Q_2$, for $Q_i \in \mathcal{R}_i$. (For example, an orthogonal range query in the plane, $[x_{lo}, x_{hi}] \times [y_{lo}, y_{hi}]$, can be expressed as the intersection of a vertical strip and a horizontal strip, in particular, the points whose x-coordinates are in the range $Q_1 = [x_{lo}, x_{hi}] \times \mathbb{R}$ and the points whose y-coordinates are in the range $Q_2 = \mathbb{R} \times [y_{lo}, y_{hi}]$.) The idea is to then "cascade" a number of search structures, one for each range subspace, together to answer a range query for the original space.

Let's see how to build such a structure for a given point set P. We first construct an appropriate range search structure, say, a partition tree, for P for the *first* range subspace $(X, \mathcal{R}_1)$. Let's call this tree T (see Fig. 1). Recall that each node $u \in T$ is implicitly associated with a *canonical subset* of points of P, which we will denote by $P(u)$. In the case that T is a partition tree, this is just the set of points lying in the leaves of the subtree rooted at u. (For example, in Fig. 1, $P(u_6) = \{p_5, \ldots, p_8\}$.) For each node $u \in T$, we construct an *auxiliary search tree* for the points of $P(u)$, but now over the *second* range subspace $(X, \mathcal{R}_2)$. Let T_u denote the resulting tree (see Fig. 1). The final data structure consists of the primary tree T, the auxiliary search trees T_u for each $u \in T$, and a link from each node $u \in T$ to the corresponding auxiliary search tree T_u. The total space is the sum of space requirements for the primary tree and all the auxiliary trees.

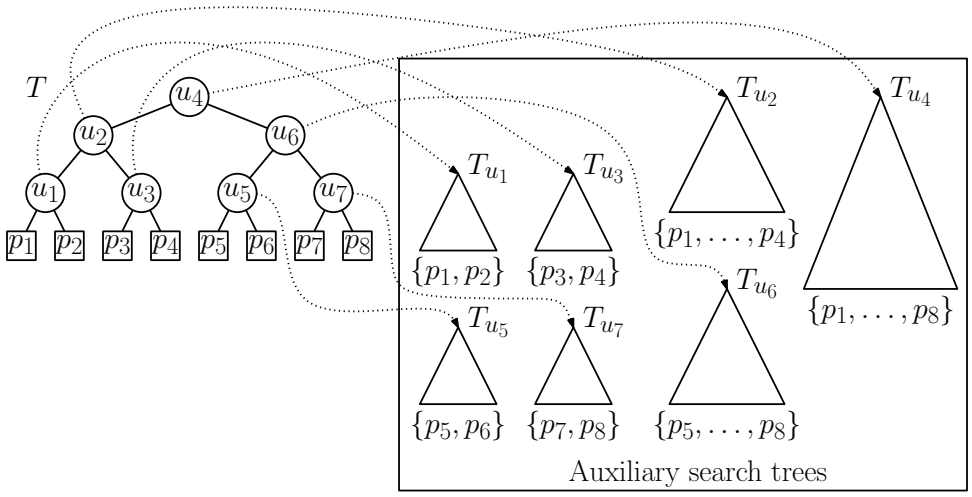

Fig. 1: Multi-layered search: A search the primary tree T identifies auxiliary trees T_{u_i} to search.

Now, given a query range $Q = Q_1 \cap Q_2$, where $Q_i \in \mathcal{R}_i$, we answer queries as follows. Recall from our earlier lecture that, the partition tree T allows us to express the answer to the query $P \cap Q_1$ as a disjoint union $\bigcup_u P(u)$ for an appropriate (and ideally small) subset of nodes $u \in T$, whose associated canonical subsets form a partition of $P \cap Q_1$. Call this subset $U(Q_1)$. In order to complete the query, for each $u \in U(Q_1)$, we access the corresponding auxiliary search tree T_u in order to determine the subset of points $P(u)$ that lie within the query range Q_2. Therefore, once we have computed the answers to all the auxiliary ranges $P(u) \cap Q_2$ for all $u \in U(Q_1)$, all that remains is to combine the results (e.g., by summing the counts or concatenating all the lists, depending on whether we are counting or reporting, respectively). The query time is equal to the sum of the query times over all the trees that were accessed.

1-dimensional Range Tree: Before discussing 2-dimensional range trees, let us first recall the 1-dimensional range treee. Given a set $P = \{p_1, \ldots, p_n\}$ of scalar keys, we wish to preprocess these points so that given a 1-dimensional interval $Q = [Q_{\text{lo}}, Q_{\text{hi}}]$ along the x-axis, we can count (or report) all the points that lie in this interval.

We begin by storing all the points of our data set in the external nodes (leaves) of any balanced binary search tree sorted by x-coordinates (e.g., an AVL tree). 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 u in this tree is implicitly associated with it *canonical subset* $P(u) \subseteq P$ of elements of P that are in the leaves descended from u. We assume that for each node u, we store the number of leaves that are descended from u, denoted u.size. Thus u.size is equal to the number of elements in $P(u)$.

Given the query interval $Q = [Q_{\text{lo}}, Q_{\text{hi}}]$, we identify all the maximal subtrees u such that the canonical set associated with u lies entirely within Q (see Fig. 2(a)). We return the sum of sizes of all these subtrees (see Fig. 2(b))

We claim that the nodes identifying the canonical subsets can be identified in $O(\log n)$ time from this structure. Intuitively, we search the tree to find the leftmost leaf u whose key is

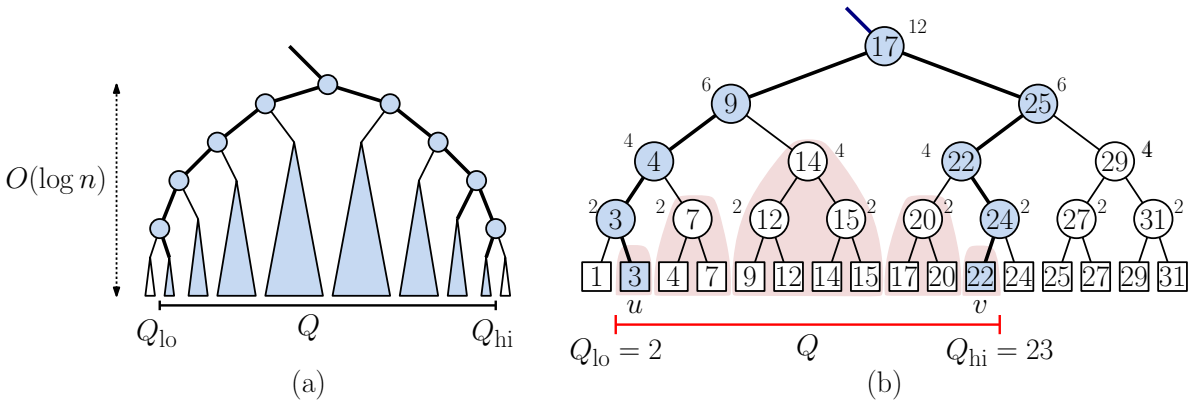

Fig. 2: The canonical subtrees (shaded in red) for the query $[2, 23]$.

greater than or equal to Q_{lo} and the rightmost leaf v whose key is less than or equal to Q_{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.

To perform this search, we maintain for each node u a *cell* $C = [x_0, x_1]$, which in this 1-dimensional case is just an interval running from the leftmost leaf key to the rightmost leaf key in this subtree.

The recursive search procedure is shown below. Its arguments to the procedure are the current node u, the range interval Q, and the current cell C. Let $C_0 = [-\infty, +\infty]$ be the initial cell for the root node (or generally, any interval large enough to contain all the points). The initial call is `range1D(root, Q, C0)`. As with the kd-tree, there are three cases. If $Q \cap C = \emptyset$, there is no overlap and we return 0. If $C \subseteq Q$, then all the points of the canononical set are included in the count. Otherwise, we recurse on the left and right subtrees.

1-Dimensional Range Counting Query

```
int range1Dx(Node u, Range Q, Interval C=[x0,x1]) {
    if (u is a leaf)                              // hit the leaf level?
        return (Q contains u.point ? 1 : 0)       // count if point in range
    else if (Q contains C )                       // Q contains entire cell?
        return u.size                             // return entire subtree size
    else if (Q is disjoint from C)                // no overlap
        return 0
    else
        return range1Dx(u.left,  Q, [x0, u.x]) + // count left side
               range1Dx(u.right, Q, [u.x, x1])   // and right side
}
```

Lemma: Given a (balanced) 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 u, such that the answer to the query is the disjoint union of the associated canonical subsets $P(u)$.

Orthogonal 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 (see the left side of Fig. 3). For each internal node u of T, recall that $P(u)$ denotes the canonical set of points associated with the leaves descended from u. For each node u of this tree we build a 1-dimensional range tree for the points of $P(u)$, but sorted on y-coordinates (see the right side of Fig. 3). This called the *auxiliary tree* associated with u. Thus, there are $n-1$ auxiliary trees, one for each internal node of T.

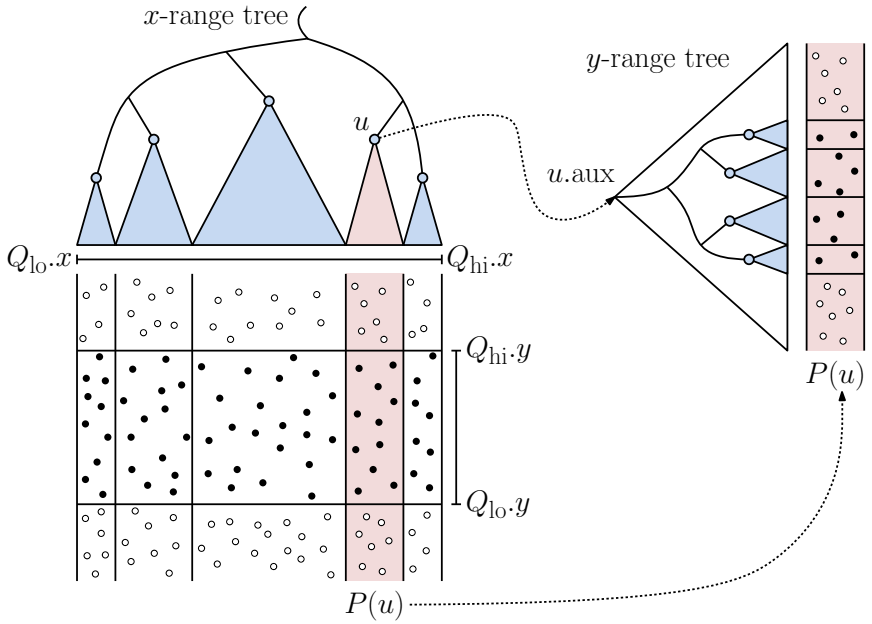

Fig. 3: 2-Dimensional Range tree.

Notice that there is a significant amount of duplication here. Each point in a leaf of the x-range tree arises in the sets $P(u)$ for all of its ancestors u in the x-range tree. We will analyze the space usage below.

Query Processing: 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. (These are shown in blue in the left side of Fig. 3.) For each such node u, 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, for each of the nodes u of the canonical subtrees, we search the associated 1-dimensional auxiliary y-range and return a count of the resulting points. These counts are summed up over all the auxiliary subtrees to obtain the final answer.

The algorithm given in the code block below is almost identical the previous one, except that we make explicit reference to the x-coordinates in the search, and rather than adding u.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-portion of Q's range, consisting of the interval $[Q_{\text{lo}}.x, Q_{\text{hi}}.x]$. The procedure `range1Dy()` is the same procedure described above, except that it searches on y-coordinates rather than x.

2-Dimensional Range Counting Query

```
int range2D(Node u, Range2D Q, Range1D C=[x0,x1]) {
    if (u is a leaf)                              // hit the leaf level?
        return (Q contains u.point ? 1 : 0)       // count if point in range
    else if (Q's x-range contains C) {            // Q's x-range contains C
        [y0, y1] = [-infinity, +infinity]         // initial y-cell
        return range1Dy(u.aux, Q, [y0, y1])       // search auxiliary tree
    }
    else if (Q.x is disjoint from C)              // no overlap
        return 0
    else
        return range2D(u.left,  Q, [x0, u.x]) +   // count left side
               range2D(u.right, Q, [u.x, x1])     // and right side
}
```

Space and Preprocessing Time: To derive a bound on the total space used, we sum the sizes of all the trees. The primary search tree T for the x-coordinates requires only $O(n)$ storage. For each node $u \in T$, the size of the auxiliary search tree T_u is clearly proportional to the number of points in this tree, which is the size of the associated canonical subset, $|P(u)|$. Thus, up to constant factors, the total space is

$$n + \sum_{u \in T} |P(u)|.$$

To bound the size of the sum, observe that each point of P appears in the set $P(u)$ for each ancestor of this leaf. Since the tree T is balanced, its depth is $O(\log n)$, and therefore, each point of P appears in $O(\log n)$ of the canonical subsets. Since each of the n points of P contributes $O(\log n)$ to the sum, it follows that the sum is $O(n \log n)$.

We claim that it is possible to construct a 2-dimensional range tree in $O(n \log n)$ time. Constructing the 1-dimensional range tree for the x-coordinates is easy to do in $O(n \log n)$ time. However, we need to be careful in constructing the auxiliary trees, because if we were to sort each list of y-coordinates separately, the running time would be $O(n \log^2 n)$. Instead, the trick is to construct the auxiliary trees in a *bottom-up manner*. The leaves, which contain a single point are trivially sorted. Then we simply merge the two sorted lists for each child to form the sorted list for the parent. Since sorted lists can be merged in linear time, the set of all auxiliary trees can be constructed in time that is linear in their total since, or $O(n \log n)$. Once the lists have been sorted, then building a tree from the sorted list can be done in linear time.

Lemma: The total size of an 2-dimensional range tree storing n keys is $O(n \log n)$, and it can be constructed in time $O(n \log n)$.

Query Time: It takes $O(\log n)$ time to identify the canonical nodes in the x-range tree. For each of these $O(\log n)$ nodes we make a call to a 1-dimensional y-range tree. When we invoke this on the subtree rooted at a node u, the running time is $O(\log |P(u)|)$. But, $|P(u)| \leq n$, so this takes $O(\log n)$ time for each auxiliary tree search. Since we are performing $O(\log n)$ searches,

each taking $O(\log n)$ time, the total search 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.

Higher Dimensions: 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: Given an n-element point set in d-dimensional space (for any constant d) orthogonal range counting queries can be answered in $O(\log^d n)$ time, and orthogonal range reporting queries can be answered in $O(k+\log^d n)$ time, where k is the number of entries reported.

Faster Queries by Cascading Search: Can we improve on the $O(\log^2 n)$ query time? We would like to reduce the query time to $O(\log n)$. (In general, this approach will shave a factor of $\log n$ from the query time, which will lead to a query time of $O(\log^{d-1} n)$ in $\mathbb{R}^d$).

What is the source of the extra log factor? As we descend the search the x-interval tree, for each node we visit, we need to search the corresponding auxiliary search tree based on the query's y-coordinates $[y_{lo}, y_{hi}]$. It is this combination that leads to the squaring of the logarithms. If we could search each auxiliary in $O(1)$ time, then we could eliminate this annoying log factor.

There is a clever trick that can be used to eliminate the additional log factor. Observe that we are repeatedly searching different lists (in particular, these are subsets of the canonical subsets $P(u)$ for $u \in U(Q_1)$) but always with the *same* search keys (in particular, y_{lo} and y_{hi}). How can we exploit the fact that the search keys are static to improve the running times of the individual searches?

The idea to rely on economies of scale. Suppose that we merge all the different lists that we need to search into a single master list. Since $\bigcup_u P(u) = P$ and $|P| = n$, we can search this master list for any key in $O(\log n)$ time. We would like to exploit the idea that, if we know where y_{lo} and y_{hi} lie within the master list, then it should be easy to determine where they are located in any canonical subset $P(u) \subseteq P$. Ideally, after making one search in the master list, we would like to be able to answer all the remaining searches in $O(1)$ time each. Turning this intuition into an algorithm is not difficult, but it is not trivial either.

In our case, the master list on which we will do the initial search is the entire set of points, sorted by y-coordinate. We will assume that each of the auxiliary search trees is a sorted array. (In dimension d, this assumption implies that we can apply this only to the last level of the multi-layered data structure.) Call these the *auxiliary lists*.

Here is how we do this. Let v be an arbitrary internal node in the range tree of x-coordinates, and let v' and v'' be its left and right children. Let A be the sorted auxiliary list for v and let A' and A'' be the sorted auxiliary lists for its respective children. Observe that A is the

disjoint union of A' and A'' (assuming no duplicate y-coordinates). For each element in A, we store two pointers, one to the item of equal or larger value in A' and the other to the item of equal or larger value in A''. (If there is no larger item, the pointer is null.) Observe that once we know the position of an item in A, then we can determine its position in either A' or A'' in $O(1)$ additional time.

Here is a quick illustration of the general idea. Let v denote a node of the x-tree, and let v' and v'' denote its left and right children. Suppose that (in increasing order of y-coordinates) the associated nodes within this range are: $\langle p_1, p_2, p_3, p_4, p_5, p_6\rangle$, and suppose that in v' we store the points $\langle p_2, p_4, p_5\rangle$ and in v'' we store $\langle p_1, p_3, p_6\rangle$ (see Fig. 4(a)). For each point in the auxiliary list for v, we store a pointer to the lists v' and v'', to the position this element would be inserted in the other list (assuming sorted by y-values). That is, we store a pointer to the largest element whose y-value is less than or equal to this point (see Fig. 4(b)).

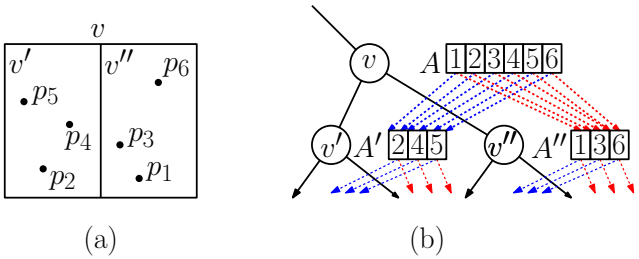

Fig. 4: Cascaded search in range trees.

At the root of the tree, we need to perform a binary search against all the y-values to determine which points lie within this interval, for all subsequent levels, once we know where the y-interval falls with respect to the order points here, we can drop down to the next level in $O(1)$ time. Thus, the running time is $O(\log n)$, rather than $O(\log^2 n)$. By applying this to the last level of the auxiliary search structures, we save one log factor, which gives us the following result.

Theorem: Given a set of n points in R^d, orthogonal rectangular range queries can be answered in $O(\log^{(d-1)} n + k)$ time, from a data structure of space $O(n \log^{(d-1)} n)$ which can be constructed in $O(n \log^{(d-1)} n)$ time.

We call this *cascading search*. This technique is special case of a more general data structures technique called *fractional cascading*. The idea is that information about the search the results "cascades" from one level of the data structure down to the next.

The result can be applied to range counting queries as well, but under the provision that we can answer the queries using a sorted array representation for the last level of the tree. For example, if the weights are drawn from a group, then the method is applicable, but if the the weights are from a general semigroup, it is not possible. (For general semigroups, we need to sum the results for individual subtrees, which implies that we need a tree structure, rather than a simple array structure.)