CMSC 420: Lecture 16
Range Trees

Range Queries: The objective of *range searching* is to count or report the set of points of some point set that lie within a given shape. The most well-known instance of range searching is *orthogonal range searching*, where the shape is an axis-aligned rectangle. In our previous lecture, we discussed the use of kd-trees for answering orthogonal range queries, where we showed that if the tree is balanced, then the running time is close to $O(\sqrt{n})$ to count the points in the range. Generally, if there are k points in the range, then we can report them in total time $O(k + \sqrt{n})$. (The modification is that whenever we find a point or subtree that lies within the range, we traverse the subtree and add all it points to the output.)

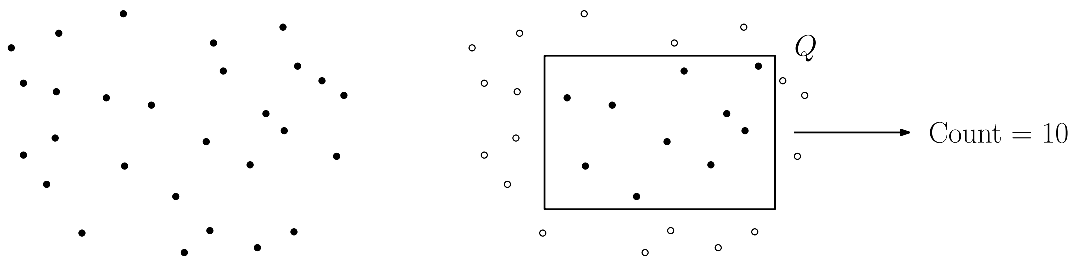

Fig. 1: 2-Dimensional orthogonal range counting query.

Range Tree Overview: In this lecture we will present a faster solution for answering these queries. We will consider the problem in 2-dimensional space, but generalizations to higher dimensions are straightforward. The data structures is called a *range tree*. Unlike kd-trees, which are general purpose and can be applied to many different types of queries, range trees are designed especially for orthogonal range queries. We will show that a range tree can answer orthogonal counting range queries in the plane in time $O(\log^2 n)$. (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.) Range trees can be generalized to higher dimensions. In any constant dimension d, it answers orthogonal range queries in $O(\log^d n)$ time.

Layering: Range trees are of additional interest because they illustrate a more general technique used in data structure design, called *layering*. Suppose that you want to design a data structure that answers queries based on multiple criteria, all of which must be satisfied. (For example, find all medical records where the patient was between ages a_{lo} and a_{hi}, whose weight is between w_{lo} and w_{hi}, and whose blood pressure is between b_{lo} and b_{hi}.) Suppose we had data structures for answering each type of query individually. How can we use these to build a data structure for answering finding the entries that satisfy all three. This can be done by "layering" the individual data structures. (Which we will describe below.) This is exactly how range trees work—they layer multiple 1-dimensional range trees to answer multi-dimensional range queries. These are called *multi-layer search trees*.

Extended Binary Search Trees: Recall that an binary tree can be *extended* by replacing each null pointer with an new type of node, called an *external node*. The original nodes are called

internal nodes. In such a tree, every internal node has exactly two children. Extended trees are often used when implementing binary search trees. (We actually saw an example when we discussed B^+-trees in an earlier lecture.)

When extended trees are used in the context of binary search trees, the two types of nodes have distinct functions.

Internal: Internal nodes each store a key (no value). They are used to direct the search towards the external nodes that actually store the data. The convention we will use is that if an internal node stores a key x, then the left subtree contains keys strictly less than x and the right subtree stores keys that are greater than or equal to x (see Fig. 2(a)).

External: External nodes contain the actual data (keys and values) that constitute the dictionary's contents (see the square nodes in Fig. 2(b)).

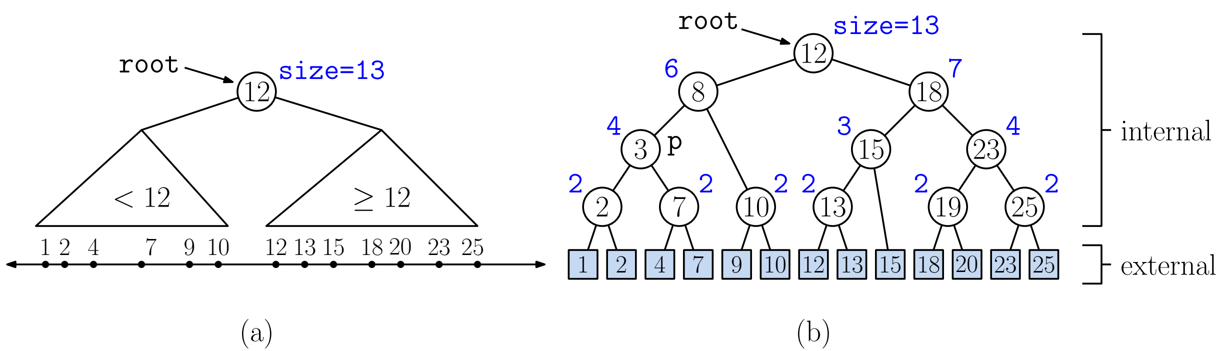

Fig. 2: Extended binary search tree (with sizes).

It is important to note that the *key values stored in the internal nodes are not part of the dictionary*. They are merely guideposts to lead us to the external nodes where the data is stored. For example, in Fig. 2(b), an internal node stores the key 8, but no leaf has this key. Therefore, 8 is *not* in the dictionary. The operation `find(8)` would return `null`.

Why are extended binary search trees useful? In practice, keys are small (e.g., a student ID number) and values are large (e.g., your major, year, college, transcript, and financial records). By storing only the keys, we keep the internal nodes small, and much of the tree can be maintained resident in memory. The values themselves take up much more space and can be stored on disk. This way, the search is fast, and we need only pay the heavy price of a disk access when retrieving the actual data item we want.

Why would we store keys like 8 in internal nodes when there is no external node with this key? This happens as a result of deletions. As we insert new data items, we need to create new internal nodes, and naturally we use the keys themselves when labeling the internal nodes. When an item is deleted from the tree, the associated external node is deleted, but the internal node with this key may remain.

When performing range counting, we want to store size information with each internal node. For each node `p`, we let `p.size` denote the number of *external nodes* descended from `p` (see Fig. 2(b)). (These nodes represent actual data items in the dictionary. We don't really care about the number of internal nodes. But, as observed in an earlier lecture, the number of internal nodes is always one less than the number of external nodes.)

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 S of scalar (say, real-valued) keys, we wish to preprocess these points so that given a 1-dimensional interval $Q = [\text{lo}, \text{hi}]$ along the x-axis, we can count (or report) all the points that lie in this interval (see Fig. 3). There are a number of simple solutions to this, but we will consider a tree-based method, because it readily generalizes to higher dimensions.

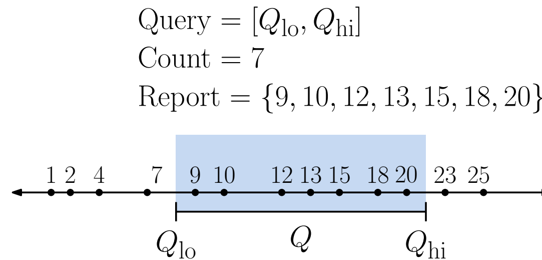

Fig. 3: 1-dimensional range query.

Let us 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. Each node p in this tree is implicitly associated with a subset $S(p) \subseteq S$ of elements of S that are in the leaves descended from p. (For example $S(\text{root}) = S$, and for the internal node `p` labeled 3 in Fig. 2(b), $S(p) = \{1, 2, 4, 7\}$.) Observe that `p.size` is equal to the size of $S(p)$.

Relevant and Canonical Nodes: Let us introduce a few definitions before continuing. Given the interval $Q = [Q_{lo}, Q_{hi}]$, we say that a node p is *relevant* to the query if $S(p) \subseteq Q$. That is, all the descendents of p lie within the interval. If p is relevant, then clearly all of the nodes descended from p are also relevant.

A relevant node p is *canonical* if p is relevant, but its parent it not. The canonical nodes are the roots of the maximal subtrees that are contained within Q (see Fig. 4(a)).

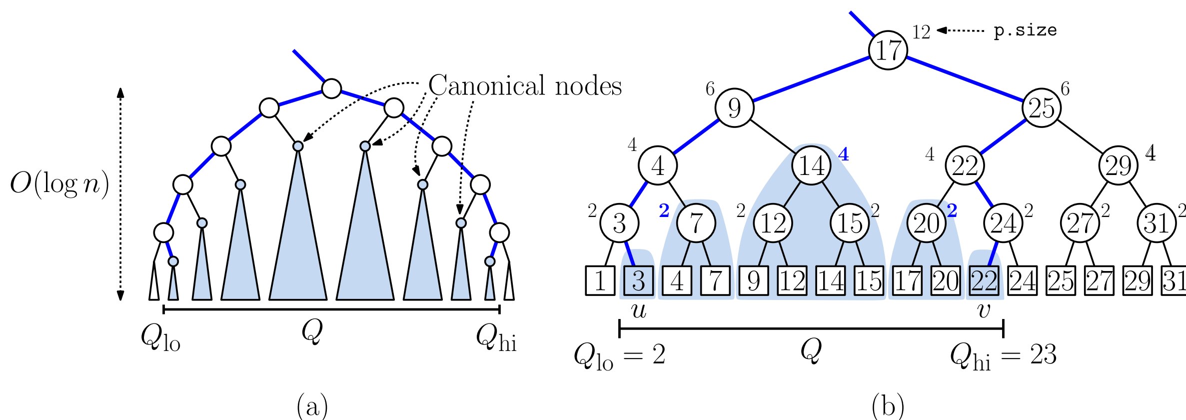

Fig. 4: The canonical subtrees for 1-dimensional range query.

For each canonical node p, the subset $S(p)$ 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.

Finding the Canonical Nodes: 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 $[Q_{\text{lo}}, Q_{\text{hi}}]$, we search the tree to find the leftmost leaf u whose key is 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. In Fig. 4(b), we illustrate this for the interval $[2, 23]$

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 $[C_{\text{lo}}, C_{\text{hi}}]$. As with kd-trees, the cell for node p 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 (or generally, any interval large enough to contain all the points). The initial call is `range1D(root, Q, C0)`.

Let $p.x$ denote the key associated with the current node p, and let $C = [x_0, x_1]$ denote the current cell for node p. 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 A contains a single point x.

The processing considers the following cases. If we arrive at an external node, we check whether the point stored in this external node is contained in Q, and if so, we we count it. Otherwise we are at an internal node. First, if Q contains the entire cell C, then all the points of this subtree lie within Q and we add `p.size` to the count (see Fig. 5(a)). Next, if Q is entirely disjoint from C (that is, $Q \cap C = \emptyset$), then none of p's descendants can contribute to the query and we return 0. Otherwise, Q and C partially overlap, and we recursively invoke the search procedure on each subtree (see Fig. 5(c)). When we invoke the procedure on the left subtree, we trim the cell to the left part $[x_0, p.x]$ and when we invoke the procedure on the right subtree, we trim the cell to the right part $[p.x, x_1]$. The recursive helper function `range1D(p, Q, C)` is shown in the code block below.

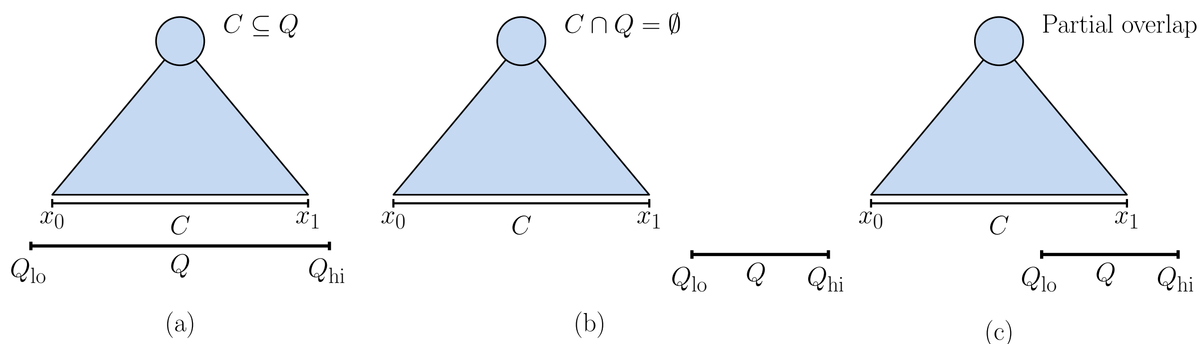

Fig. 5: Range search cases: (a) Q contains p's cell, (b) Q is disjoint from p's cell, (c) Q partially overlaps p's cell.

The external nodes counted in the second line and the internal nodes for which we return

1-Dimensional Range Counting Query

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

`p.size` are the canonical nodes. (To extend this from counting to to reporting, 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 (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 p, such that the answer to the query is the disjoint union of the associated canonical subsets $S(p)$.

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 (see the left side of Fig. 6). For each internal node p of T, recall that $S(p)$ denotes the points associated with the leaves descended from p. For each node p of this tree we build a 1-dimensional range tree for the points of $S(p)$, but sorted on y-coordinates (see the right side of Fig. 6). This called the *auxiliary tree* associated with p. Thus, there are $n - 1$ auxiliary trees, one for each internal node of T.

Notice that there is a significant amount of duplication here. Each point in a leaf of the x-range tree arises in the sets $S(p)$ for all of its ancestors p in the x-range tree. Since the tree is assumed to be balanced, it has height $O(\log n)$, and therefore, each of the n points appears in $O(\log n)$ auxilliary trees. Thus, the sum of the sizes of all the auxiliary trees is $O(n \log n)$. The original tree itself contributes space of $O(n)$. Thus, we have the following total space for the 2-dimensional range tree.

Claim: The total size of an 2-dimensional range tree storing n keys 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. (These are shown in blue in the left side of Fig. 6.) For each such node p, 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 p of the canonical subtrees, we search the associated 1-dimensional

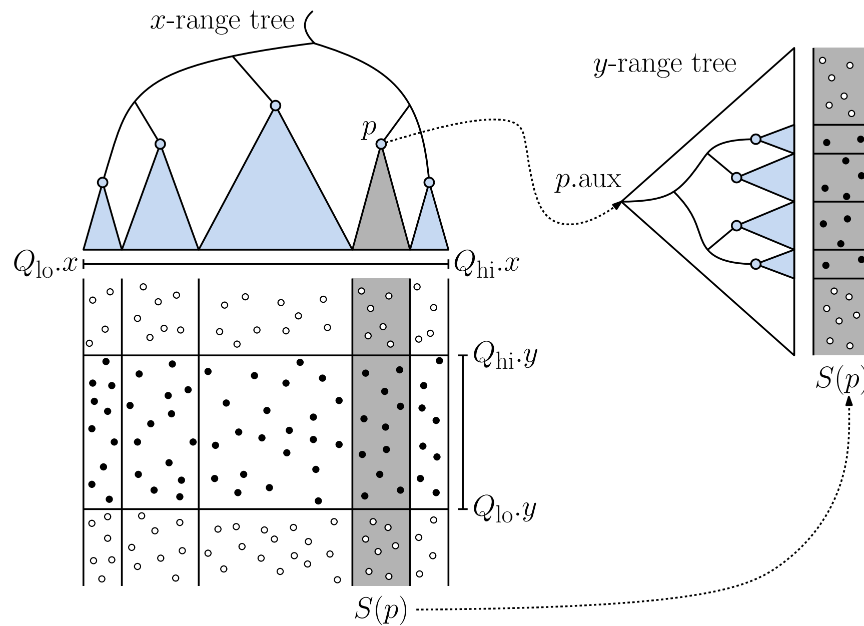

Fig. 6: 2-Dimensional Range tree.

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 `p.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 function call `Q.contains(p.point)` is applied on both coordinates, but the call `Q.x.contains(C)` only checks the x-part of Q's range. 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 p, Range2D Q, Range1D C=[x0,x1]) {
    if (p.isExternal)                              // hit the leaf level?
        return (Q.contains(p.point) ? 1 : 0)       // count if point in range
    else if (Q.x.contains(C)) {                    // Q's x-range contains C
        [y0,y1] = [-infinity, +infinity]           // initial y-cell
        return range1Dy(p.aux.root, Q, [y0, y1])   // search auxiliary tree
    }
    else if (Q.x.isDisjointFrom(C))                // no overlap
        return 0
    else
        return range2D(p.left,  Q, [x0, p.x]) +    // count left side
               range2D(p.right, Q, [p.x, x1])      // and right side
}
```

Analysis: 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 p, the running time is $O(\log |S(p)|)$. But, $|S(p)| \leq n$, 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.

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.