CMSC 420: Lecture 16
Answering Queries with kd-trees

Recap: In our previous lecture we introduced kd-trees, a multi-dimensional binary partition tree that is based on axis-aligned splits. We have shown how to perform the operations of insertion and deletion from kd-trees. In this lecture, we will investigate an important geometric query using kd-trees: orthogonal range search queries.

Range Queries: Given any point set, a fundamental type of query is called a *range query* or more properly, an *orthogonal range query*. To motivate this sort of query, suppose that you querying a biomedical database with millions of records. Each medical record is encoded as a vector of health statistics, such as height, weight, blood pressure, etc. Each coordinate is the numeric value of some statistic, such as a person's height, weight, blood pressure, etc. Suppose that you want to anwwer queries of the form "how many patients whose range 70–80 kilograms, heights in the range 160–170 centimeters, etc." This is equivalent to finding the number of points in the database that lie within an axis-orthogonal rectangle, defined by the intersection of these intervals (see Fig. 1).

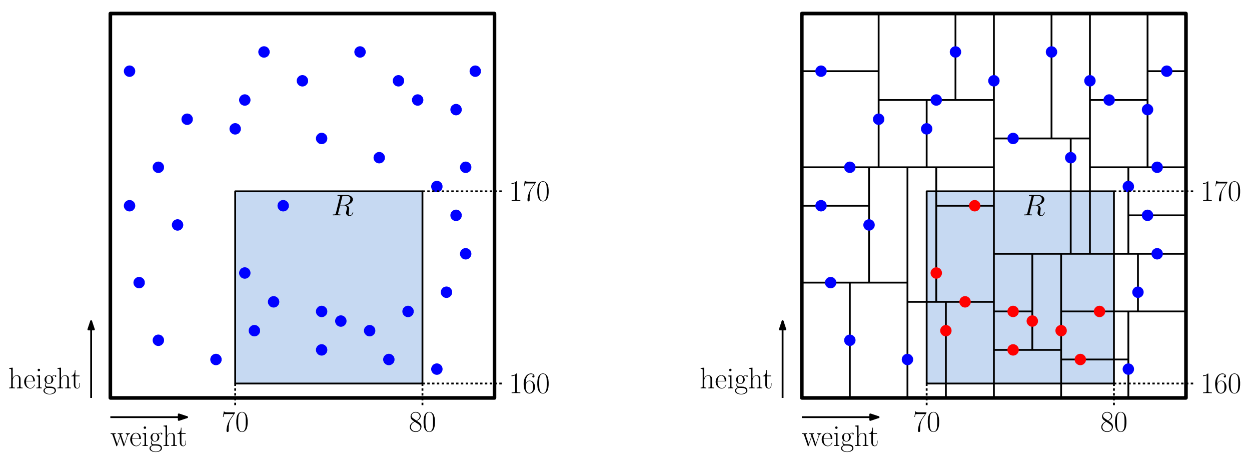

Fig. 1: Orthogonal range query.

More formally, given a set P of points in d-dimensional real space, $\mathbb{R}^d$, we wish to store these points in a kd-tree so that, given a query consisting of an axis-aligned rectangle, denoted R, we can efficiently count or report the points of P lying within R. Listing all the points lying in the range is called a *range reporting query*, and counting all the points in the range is called a *range counting query*. The solutions for the two problems are often similar, but some tricks can be employed when counting, that do not apply when reporting.

A Rectangle Class: Before we get into a description of how to answer orthogonal range queries with the kd-tree tree, let us first define a simple class for storing a multi-dimensional rectangle, or *hyper-rectangle* for short. The private data consists of two points `low` and `high`. A point q lies within the rectangle if $\text{low}[i] \leq q[i] \leq \text{high}[i]$, for $0 \leq i \leq d-1$ (assuming Java-like indexing). In addition to a constructor, the class provides a few useful geometric primitives (illustrated in Fig. 2).

`boolean contains(Point q)`: Returns `true` if and only if point q is contained within this rectangle (using the above inequalities).

`boolean contains(Rectangle c):` Returns `true` if and only if this rectangle contains rectangle c. This boils down to testing containment on all the intervals defining each of the rectangles' sides:

$$\big[c.\text{low}[i], c.\text{high}[i]\big] \;\subseteq\; \big[\text{low}[i], \text{high}[i]\big], \quad \text{for all } 0 \le i \le d-1.$$

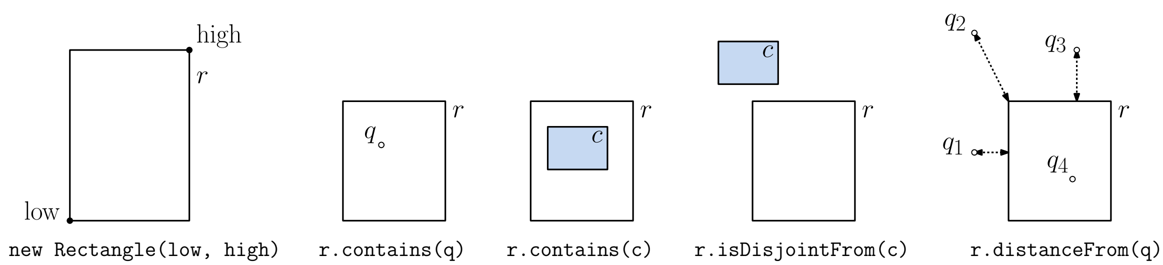

Fig. 2: An axis-parallel rectangle methods.

`boolean isDisjointFrom(Rectangle c):` Returns `true` if and only if rectangle c is disjoint from this rectangle. This boils down to testing whether any of the defining intervals are disjoint, that is

$$r.\text{high}[i] < c.\text{low}[i] \text{ or } r.\text{low}[i] > c.\text{high}[i], \quad \text{for any } 0 \le i \le d-1.$$

`float distanceTo(Point q):` Returns the minimum Euclidean distance from q to any point of this rectangle. This can be computed by computing the distance from the coordinate $q[i]$ to this rectangle's ith defining interval, taking the sums of squares of these distances, and then taking the square root of this sum:

$$\sqrt{\sum_{i=0}^{d-1} (\text{distance}(q[i], \big[\text{low}[i], \text{high}[i]\big]))^2}$$

There is one additional function worth discussing, because it is used in many algorithms that involve kd-trees. The function is given a rectangle r and a splitting point s lying within the rectangle. We want to cut the rectangle into two sub-rectangles by a line that passes through the splitting point. These are used in a context where the rectangle r represents the cell associated with a given kd-tree node, and by cutting the cell through the splitter, we generate the cells associated with the node's left and right children.

`Rectangle leftPart(int cd, Point s):` (and `rightPart(int cd, Point s)`) These are both given a cutting dimension `cd` and a point `s` that lies within the rectangle. The first returns the subrectangle lying to the left (below) of s with respect to the cutting dimension, and the other returns the subrectangle lying to the right (above) of s with respect to the cutting dimension (see Fig. 2). More formally, `leftPart(cd, s)`, returns a rectangle whose low point is the same as r.low and whose high point is the same as r.high except that the cd-th coordinate is set to $s[\text{cd}]$. Similarly, `rightPart(cd, s)`, returns a rectangle whose high point is the same as r.high and whose low point is the same as r.low except that the cd-th coordinate is set to $s[\text{cd}]$.

The following code block provides a high-level overview of the `Rectangle` class (without defining any of the functions).

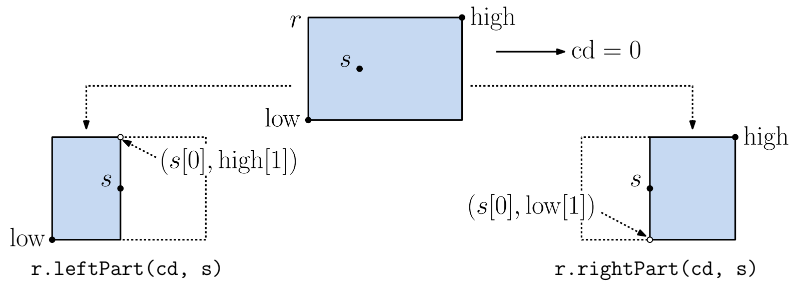

Fig. 3: The functions `leftPart` and `rightPart`.

A skelton of a simple Rectangle class

```
public class Rectangle {
    Point low;                                      // lower left corner
    Point high;                                     // upper right corner

    public Rectangle(Point low, Point high)         // constructor
    public boolean contains(Point q)                // do we contain q?
    public boolean contains(Rectangle c)            // do we contain rectangle c?
    public boolean isDisjointFrom(Rectangle c)      // disjoint from rectangle c?
    public float distanceTo(Point q)                // minimum distance to point q
    public Rectangle leftPart(int cd, Point s)      // left part from s
    public Rectangle rightPart(int cd, Point s)     // right part from s
}
```

Anwering the Range Query: In order to answer range counting queries, let us first assume that each node `p` of the tree has been augmented with a member `p.size`, indicating the number of points lying within the subtree rooted at `p`. This can easily be updated as points are inserted to and deleted from the tree. The counting function, `rangeCount(R, p, cell)` operates recursively. The first argument `R` is the query range, the second argument `p` is the node currently visited, and `cell` is its associated cell. It returns a count of the number of points within `p`'s subtree that lie within `R`. The initial call is `rangeCount(R, root, boundingBox)`, where `boundingBox` is the bounding box of the entire kd-tree.

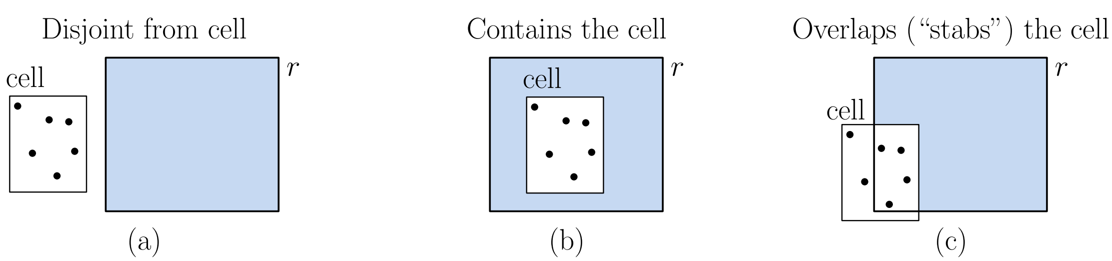

Fig. 4: Cases arising in orthogonal range searching.

The function operates recursively, working from the root down to the leaves. First, if we fall out of the tree then there is nothing to count. Second, if the current node's cell and the range are completely disjoint, we may return 0, because none of this node's points lie within the range (see Fig. 4). Next, if the query range completely contains the current cell, we can count all the points of `p` as lying within the range, and so we return `p.size`. Otherwise, the range partially overlaps the cell. We say that the range *stabs* the cell. In this case, we apply the function recursively to each of our two children. The function is presented in the code block below.

kd-tree Range Counting Query

```
int rangeCount(Rectangle R, KDNode p, Rectangle cell) {
    if (p == null) return 0              // empty subtree
    else if (R.isDisjointFrom(cell))     // no overlap with range?
        return 0
    else if (R.contains(cell))           // the range contains our entire cell?
        return p.size                    // include all points in the count
    else {                               // the range stabs this cell
        int count = 0
        if (R.contains(p.point))         // consider this point
            count += 1
                                         // apply recursively to children
        count += rangeCount(R, p.left,  cell.leftPart(p.cutDim, p.point))
        count += rangeCount(R, p.right, cell.rightPart(p.cutDim, p.point))
        return count
    }
}
```

An Example: Fig. 5 shows an example of a range search. Next to each node we store the size of the associated subtree in blue. We say that a node is *visited* if a call to `rangeCount()` is made on this node. We say that a node is *processed* if both of its children are visited. Observe that for a node to be processed, its cell must overlap the range without being contained within the range. In the example, the shaded nodes are those that are not processed. For example the subtree rooted at h is entirely contained within the range, and any points in the subtree can be safely included in the count. (In this case, this includes the two points p and h.) The subtrees rooted at k and g are entirely disjoint from the query, and the subtrees rooted at these nodes can be completely ignored. The nodes with red squares surrounding them those whose points have been added individually to the count (by the condition `R.contains(p.point)`). There are four such nodes d, f, l, and q. Combined with the two points of h's subtree, the total count returned is 6.

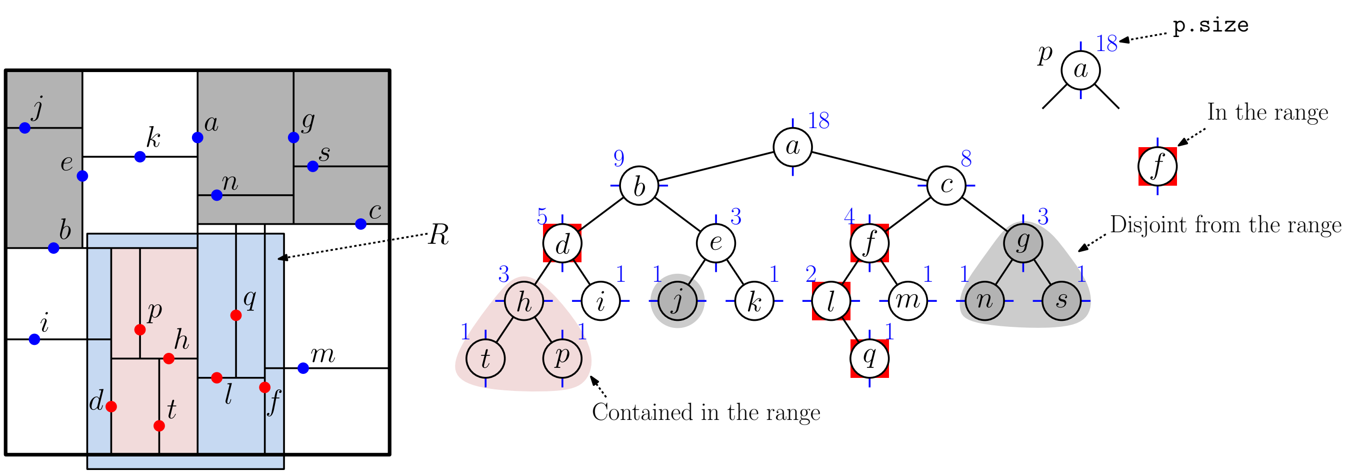

Fig. 5: Range search in kd-trees. The subtree rooted at h is counted entirely. The subtrees rooted at k and g are excluded entirely. The other points are checked individually.

Analysis of query time: How many nodes does this method visit altogether? We claim that the total number of nodes is $O(\sqrt{n})$ assuming a balanced kd-tree (which is a reasonable assumption in the average case).

Theorem: Given a balanced kd-tree with n points in $\mathbb{R}^2$ (where the cutting dimension alternates between x and y), orthogonal range counting queries can be answered in $O(\sqrt{n})$ time.

Recall from the discussion above that a node is processed (both children visited) if and only if the range partially overlaps or "stabs" the cell. To bound the total number of nodes that are processed in the search, it suffices to count the total number of nodes whose cells are stabbed by the query rectangle. Rather than prove the above theorem directly, we will prove a simpler result, which illustrates the essential ideas behind the proof. Rather than using a 4-sided rectangle, we consider an orthogonal range having a only one side, that is, an orthogonal halfplane. In this case, the query stabs a cell if the vertical or horizontal line that defines the halfplane intersects the cell.

Lemma: Given a balanced kd-tree with n points in $\mathbb{R}^2$ (where the cutting dimension alternates between x and y), any vertical or horizontal line stabs the cells of $O(\sqrt{n})$ nodes of the tree.

Proof: It will simplify the analysis to assume that the tree is "perfectly balanced", which means that if a subtree contains m points then each of its two subtrees contains at most $m/2$ points. (The proof generally works as long as the height of the tree is $O(\log n)$, but it is a bit more complicated.)

By symmetry, it suffices to consider a horizontal line. Consider a processed node which has a cutting dimension along x. A horizontal line can stab the cells of both its children. On the other hand, if the cutting dimension is along y, a horizontal line either stabs the upper cell or the lower cell, but not both. (If our horizontal line coincides with the cutting line, then we consider it to stab the upper cell and not the lower cell.)

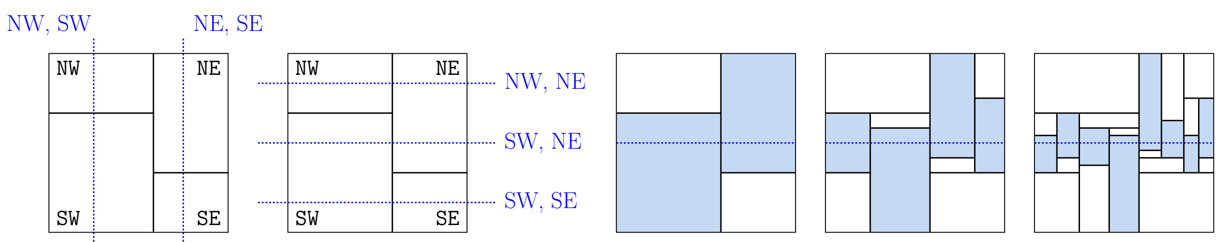

Fig. 6: An axis-parallel line in 2D can stab at most two out of four cells in two levels of the kd-tree decomposition. In general, it stabs 2^i cells at level $2i$.

Since we alternate splitting on x then y, this means that after descending two levels of the tree, we may stab the cells of at most two of the possible four grandchildren of the original node. (This is illustrated in Fig. 6.) By our assumption that the tree is balanced, if the parent node has n points, each of its two children has at most $n/2$ points, and each of the four grandchildren has at most $n/4$ points. Therefore, the total number of nodes whose cells are stabbed satisfies the following recurrence:

$$T(n) = \begin{cases} 2 + 2T(n/4) & \text{if } n \geq 2, \\ 1 & \text{if } n = 1. \end{cases}$$

We can solve this recurrence either by appealing to the *Master Theorem* (see the CLRS Algorithms book), but it is easy enough to solve directly. By expanding the recurrence and observing the trend, we obtain:

$$\begin{aligned}
T(n) &= 2 + 2T(n/4) \\
&= 2 + 2(2 + T(n/16)) \;=\; 2 + 4 + 4T(n/16) \\
&= (2 + 4) + 4(2 + 2T(n/64)) \;=\; (2 + 4 + 8) + 8T(n/64) \\
&= \ldots \\
&= \sum_{i=1}^{k} 2^i + 2^k T(n/4^k).
\end{aligned}$$

To get to the basis case of $T(1)$, we set $n/4^k = 1$, which yields $k = \log_4 n = (\lg n)/(\lg 4) = (\lg n)/2$. The summation term (which is the dominant term) in $T(n)$ is:

$$\sum_{i=1}^{k} 2^i \;\approx\; 2 \cdot 2^k \;=\; 2 \cdot 2^{(\lg n)/2} \;=\; 2 \cdot (2^{\lg n})^{1/2} \;=\; 2 \cdot n^{1/2} \;=\; 2\sqrt{n} \;=\; O(\sqrt{n}).$$

This completes the proof.

We have shown that any (infinite) vertical or horizontal line can stab only $O(\sqrt{n})$ cells of the tree. This upper bound clearly holds for any finite vertical or horizontal line segment. Thus, if we apply it to the four line segments that bound R, it follows that the the total number of cells stabbed by the query range R is $O(4\sqrt{n}) = O(\sqrt{n})$. The total query time is determined by the sum of nodes visited, which is dominated by the sum of the nodes that are stabbed by the query. Therefore, the overall running time (assuming a balanced kd-tree and alternating cutting dimensions) is $O(\sqrt{n})$. This completes the proof of the above lemma.

To see whether you understand this, you might try generalizing this analysis to arbitrary dimensions d (where d is constant). As a hint, the query time in general will be $O(n^{1-1/d})$. In the case where $d = 2$, this is $O(\sqrt{n})$. Observe that as d gets larger and larger, the query time approaches $O(n)$. Unfortunatey, $O(n)$ is the same time as brute-force search (since we can simply test every point one-by-one to see whether it lies in the range). Thus, kd-trees are efficient only for fairly small values of d.