CMSC 420: Lecture 14
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 two important geometric queries using kd-trees: orthogonal range search queries and nearest-neighbor 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 point of the database is associated with the medical record of a patient. Each coordinate is the numeric value of some statistic, such as the patient's height, weight, blood pressure, HDL and LDL cholesterol numbers, etc. So, if there are 20 different numbers associated with each patient's record, each patient can be modeled as a point in a 20-dimensional space of real numbers, or $\mathbb{R}^{20}$ for short.

Suppose that as part of your study or these patients, you want to know information such as "how many patients are there with weights in the range 70–80 kilograms, heights in the range 160–170 centimeters, etc." This amounts 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). This is where the name *orthogonal range searching* originates.

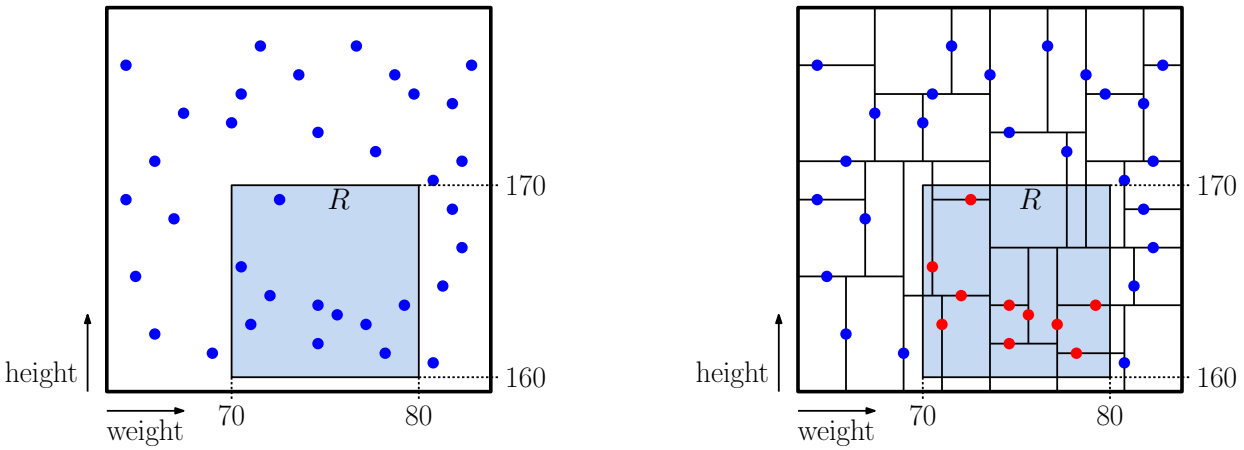

Fig. 1: Orthogonal range query.

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:

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

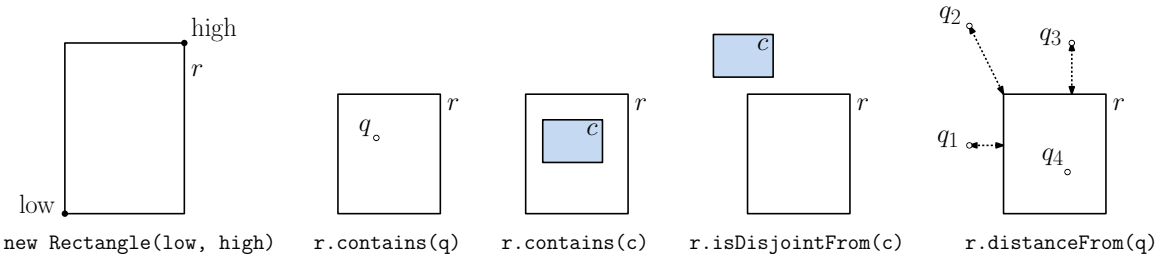

Fig. 2: An axis-parallel rectangle methods.

`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[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[cd].

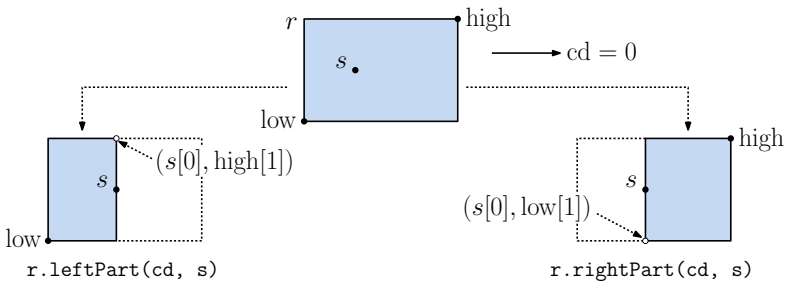

Fig. 3: The functions `leftPart` and `rightPart`.

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
}
```

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

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 range itself, 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.

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 is completely disjoint from the query range, 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`. Finally, the range must partially overlap 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.

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

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))          // range contains our entire cell?
        return p.size;                  // include all points in the count
    else {                              // range partially overlaps cell
        int count = 0;
        if (r.contains(p.point))        // consider this point
            count++;
                                        // 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;
    }
}
```

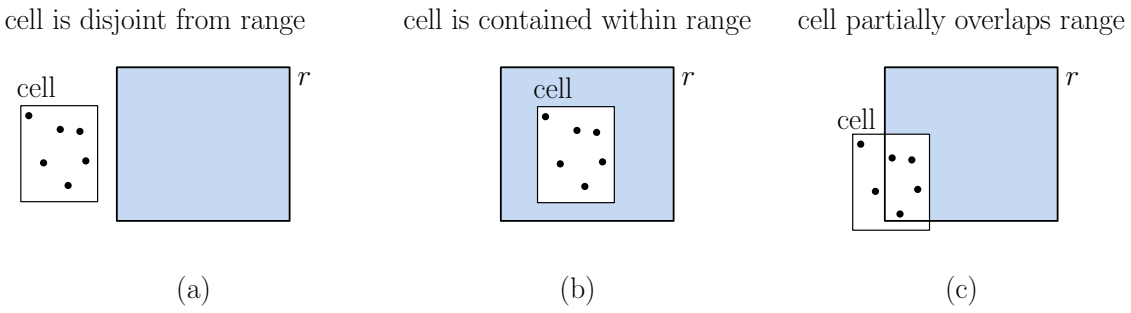

Fig. 4: Cases arising in orthogonal range searching.

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.

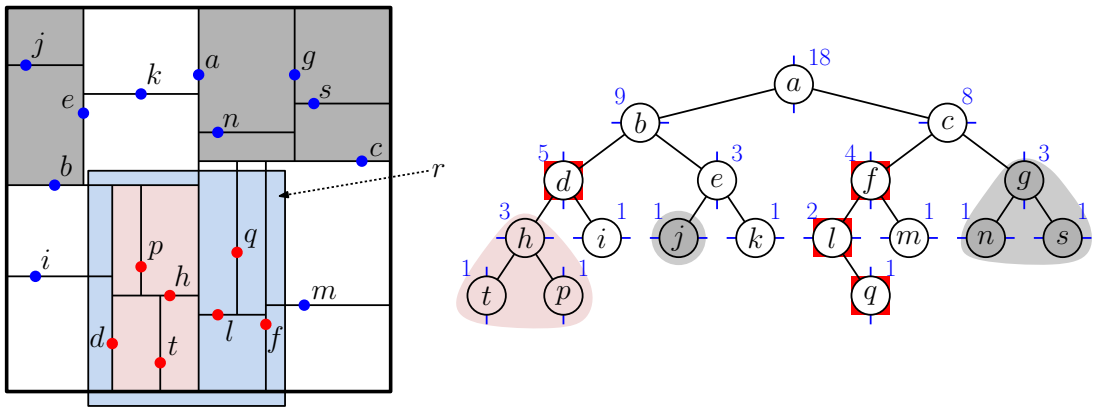

Fig. 5: Range search in kd-trees. The subtree rooted at h is counted entirely. The subtrees rooted at k and g are excluded entirely. The other points are checked individually.

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, 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 cell overlaps the range without being contained within the range. We say that such a cell is *stabbed* by the query. 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, any vertical or horizontal line stabs $O(\sqrt{n})$ cells of the tree.

Proof: Since the tree is balanced, its height is $O(\log n)$. Since the constant factor will not really matter, it will simplify matters to assume that the height is exactly $\lg n$. Let us consider the case of a vertical line $x = x_0$. The horizontal case is symmetrical.

Consider a processed node which has a cutting dimension along x. The vertical line $x = x_0$ either stabs the left child or the right child but not both. If it fails to stab one of the children, then it cannot stab any of the cells belonging to the descendents of this

child either. If the cutting dimension is along the y-axis (or generally any other axis in higher dimensions), then the line $x = x_0$ stabs both children's cells.

Since we alternate splitting on left and right, this means that after descending two levels in the tree, we may stab at most two of the possible four grandchildren of each node. (This is illustrated in Fig. 6.) In general each time we descend two more levels we double the number of nodes being stabbed. Thus, we stab the root node, at most 2 nodes at level 2 of the tree, at most 4 nodes at level 4, 8 nodes at level 6, and generally at most 2^i nodes at level $2i$.

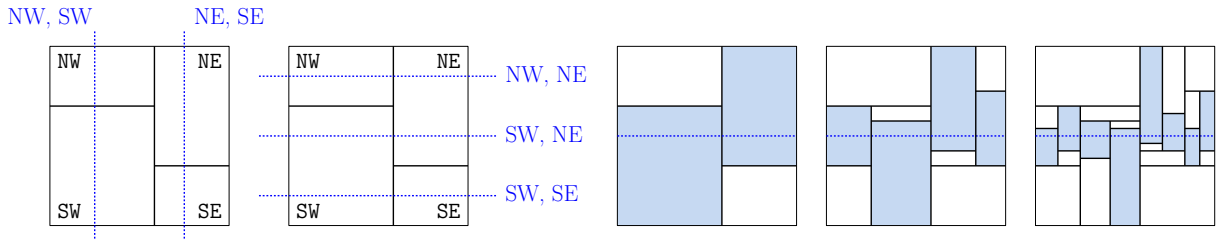

Fig. 6: An axis-parallel line in 2D can stab at most two out of four cells in two levels of the kd-tree decomposition. In general, it stabs 2^i cells at level $2i$.

Because we have an exponentially increasing number, the total sum is dominated by its last term. Thus, it suffices to count the number of nodes stabbed at the lowest level of the tree. If we assume that the kd-tree is balanced, then the tree has height of $h \approx \lg n$ (up to constant factors). The number of leaf nodes processed at the bottommost level is

$$2^{h/2} \approx 2^{(\lg n)/2} = (2^{\lg n})^{1/2} = n^{1/2} = \sqrt{n}.$$

This completes the proof.

We have shown that any vertical or horizontal line can stab only $O(\sqrt{n})$ cells of the tree. Thus, if we were to extend the four sides of Q into lines, the total number of cells stabbed by all these lines is at most $O(4\sqrt{n}) = O(\sqrt{n})$. Thus the total number of cells stabbed by the query range is $O(\sqrt{n})$, and hence the total query time is $O(\sqrt{n})$. Again, this assumes that the kd-tree is balanced (having $O(\log n)$ depth). If the points were inserted in random order, this will be true on average.

Nearest-Neighbor Queries: Next we consider how to perform an important retrieval query on a kd-tree. Nearest neighbor queries are among the most important queries. We are given a set of points P stored in a kd-tree, and a query point q, and we want to return the point of P that is closest to q. Let's assume that distances are measured using Euclidean distances. In particular, given two points $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$, their Euclidean distance is

$$\text{dist}(p, q) = \sqrt{(p_1 - q_1)^2 + \cdots + (p_d - q_d)^2}.$$

Generalizations to other sorts of distance functions (e.g., the Manhattan or taxicab distance) is also possible. An example is shown in Fig. 7. Observe that the circle centered at q and passing through its nearest neighbor p contains no other points. However, every leaf cell of the kd-tree whose cell overlaps the interior of this circle (shaded in the figure) may need to be visited in the search, since each might contribute a point that could be closer to q than p is. What makes the search efficient is that the number of such nodes is usually much smaller

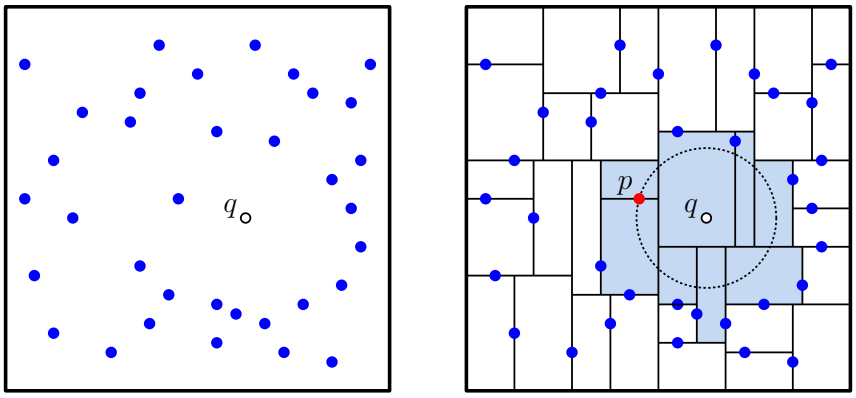

Fig. 7: Nearest-neighbor searching using a kd-tree.

than the total number of nodes in the tree. Of course, finding these nodes is the key issue in answering nearest neighbor queries.

An intuitively appealing approach to nearest neighbor queries would be to find the leaf node of the kd-tree that contains q and then search this and the neighboring cells of the kd-tree. The problem is that the nearest neighbor may actually be very far away, in the sense of the tree's structure. For example, in Fig. 8, many of the points are at nearly the same distance from the query point q. It would be necessary to visit almost all the nodes of the tree to determine which of these points is the actual nearest neighbor.

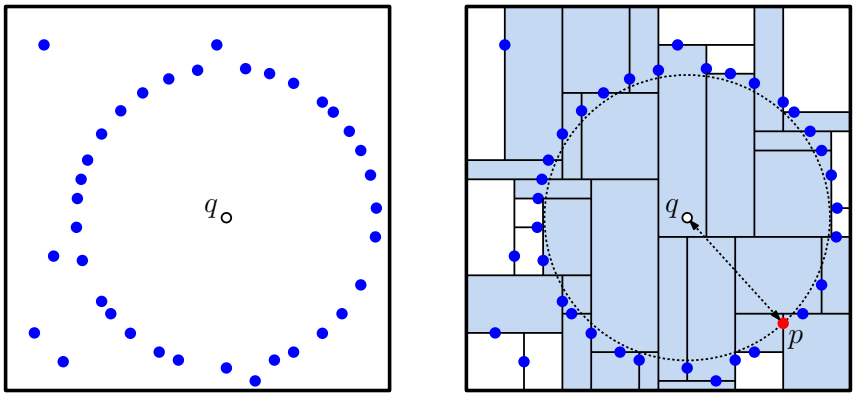

Fig. 8: A (nearly) worst-case scenario for nearest-neighbor searching. Almost all the nodes of the tree need to be visited, since any might be the nearest neighbor.

We will need a more systematic approach to finding nearest neighbors. Nearest neighbor queries illustrate three important elements of range and nearest neighbor processing.

Partial results: Store the intermediate results of the query and update these results as the query proceeds.

Traversal order: Visit the subtree first that is more likely to be relevant to the final results.

Pruning: Do not visit any subtree that be judged to be irrelevant to the final results.

Nearest-neighbor Utilities: Before presenting the code for nearest-neighbor searching, we introduce a few helpful utilities. First, recall that every cell of the kd-tree is associated with

an axis-parallel rectangle, called its *cell*. (For $d \geq 3$ the generalization of a rectangle is called a *hyperrectangle*, but we will just use the term "rectangle" for simplicity.) A convenient way to represent a rectangle in any d-dimensional space is to give two points `low` and `high`. In 2D, these represent the lower-left and upper-right corners of the rectangle, respectively. In general, the rectangle consists of all points q such that $\text{low}_i \leq q_i \leq \text{high}_i$ (see Fig. 2(a)). A possible implementation, without any details, is outlined in the code block below. (We make use of the `Point` object, which was introduced in the previous lecture.)

Nearest-neighbor Code: Our procedure for returning the nearest neighbor actually only returns the distance to the nearest neighbor, but it is an easy matter to modify the code to produce both the distance and the point achieving this distance. (Think about how you would do this.) As usual, we employ a recursive utility function that works on an individual node `p` of the tree. The function `nearNeighbor(q, p, cell, bestDist)` is given four parameters:

- the query point `q`
- the current node `p` of the tree
- the rectangular cell associated with this node, `cell`, and
- the smallest distance, `bestDist`, between `q` and any point seen so far in the search.

The procedure works as follows:

- First, if `p` is `null`, we must have fallen out of the tree, and we just return the current smallest distance, `bestDist` as the answer.
- Otherwise, we compute the distance from the point `p.point` to `q`, and update the `bestDist` value if this point is closer than the previous.
- Next, we need to search the subtrees for possibly closer points:
  - We invoke `leftPart` and `rightPart` to determine the cells of the left and right subtrees, respectively (see Fig. 9(a)).
  - Next, we check which side `p.point` the query point lies. The closer child of `p` is the one that lies on the same side of the splitter as `q` does.

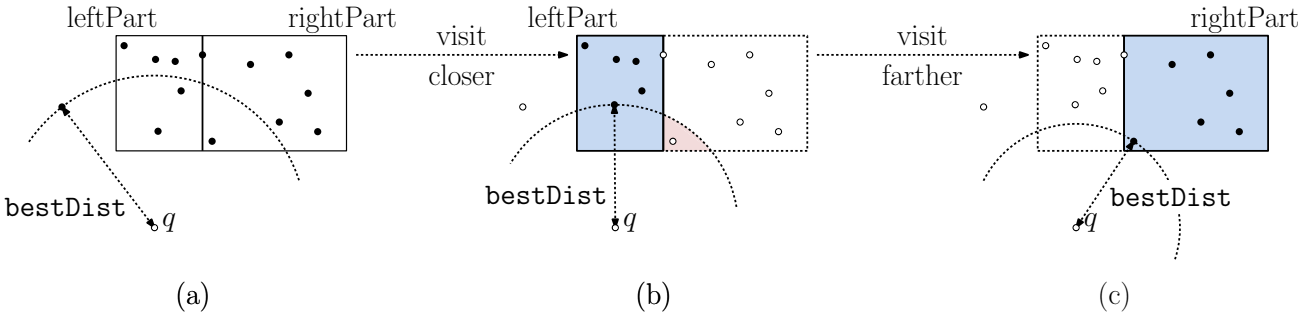

Fig. 9: Nearest-neighbor searching.

  - We visit the closer subtree first (see Fig. 9(b)), since it is more likely to yield the nearest neighbor. The value of `bestDist` will be updated to the closest point seen so far.
  - After returning from this call, we compute `q`'s distance to the right subtree cell. Observe that if this distance is greater than `bestDist`, there is no chance that

the other subtree contains the nearest neighbor, and so there is no need to visit this subtree. Otherwise, we apply the search recursively to the right subtree (see Fig. 9(c)) and update `bestDist` accordingly.

Given a query point q, the initial call is `nearNeigh(q, root, rootCell, Float.MAX_VALUE)`, where `rootCell` is the rectangle that encloses the entire tree contents, and `Float.MAX_VALUE` is the maximum possible float value. The code is presented below.

Compute distance to nearest neighbor in kd-tree

```
float nearNeighbor(Point q, KDNode p, Rectangle cell, float bestDist) {
    if (p != null) {
        float thisDist = q.distanceTo(p.point);             // distance to p's point
        bestDist = Math.min(thisDist,  bestDist);           // keep smaller distance

        int cd = p.cutDim;                                  // cutting dimension
        Rectangle leftCell = cell.leftPart(cd, p.point);    // left child's cell
        Rectangle rightCell = cell.rightPart(cd, p.point);  // right child's cell

        if (q[cd] < p.point[cd]) {                          // q is closer to left
            bestDist = nearNeighbor(q, p.left, leftCell, bestDist);
            if (rightCell.distanceTo(q) < bestDist) {       // worth visiting right?
                bestDist = nearNeighbor(q, p.right, rightCell, bestDist);
            }
        } else {                                            // q is closer to right
            bestDist = nearNeighbor(q, p.right, rightCell, bestDist);
            if (leftCell.distanceTo(q) < bestDist) {        // worth visiting left?
                bestDist = nearNeighbor(q, p.left, leftCell, bestDist);
            }
        }
    }
    return bestDist;
}
```

An example of the algorithm in action is shown in Fig. 10. The algorithm starts by descending to the leaf node (the upper child of $(70, 30)$), computing distances to all the points seen along the way. At this point $(70, 30)$ is the closest, and its distance to q defines `bestDist`. Because the lower child of $(70, 30)$ overlaps the ball of radius `bestDist`, we need to inspect this subtree. When we visit $(50, 25)$, we discover that it is even closer. We visit both its children. However, observe that when we arrive at $(60, 10)$, we visit the closer of its two children (the empty subtree lying above this point), but there is no need to visit its lower child, because it lies entirely outside of the ball of radius `bestDist`. We then return from the recursion. On returning to $(80, 40)$ and $(70, 80)$, we see that the cells of their other children lie entirely outside the ball of radius `bestDist`, and so we do not need to visit them. On returning to the root at $(35, 90)$ we see that its left subtree does overlap the `bestDist` ball, and so we recurse on that subtree as well. We continue until arriving at the closest leaf to the query point, namely the right child of $(25, 10)$. We compute distances too all the points associated with the nodes visited, and we discover along the way that $(25, 50)$ is even closer to the query point, and thus `bestDist` is again reduced. After this, all the remaining cells (shaded in white in the figure) lie outside the nearest-neighbor ball, and so we can terminate the search.

Analysis: How efficient is this procedure? It is quite difficult to analyze from the perspective of its worst-case performance, because as seen in Fig. 8, there are cases where we may need to

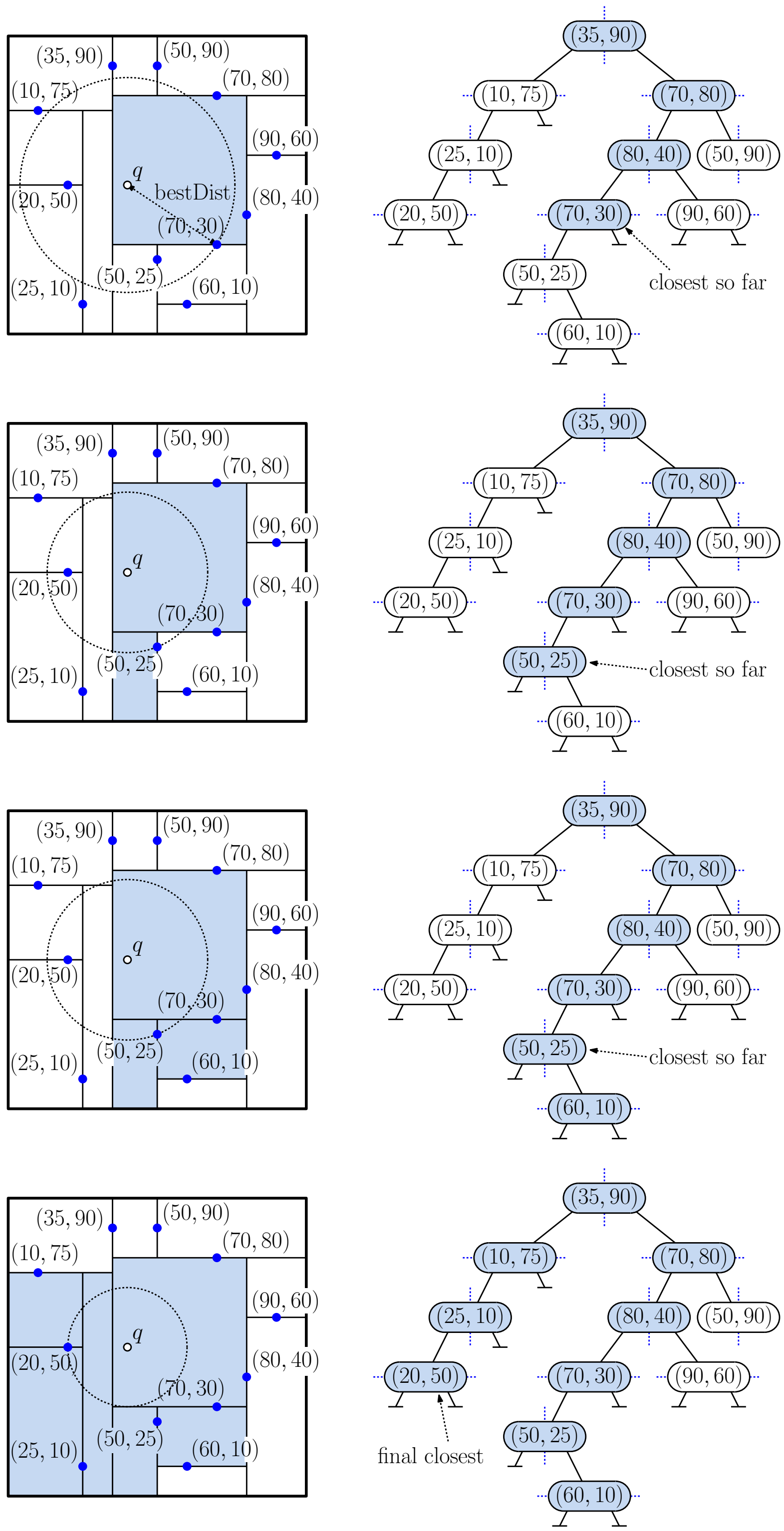

Fig. 10: Nearest-neighbor search.

visit almost every node of the tree, because almost all the points are equidistant from the query point. However, this is really a very pathological example. In most instances, the typical running time is much closer to $O(2^d + \log n)$, where d is the dimension of the space. Generally, you expect to visit some set of nodes that are in the neighborhood of the query point (giving rise to the 2^d term) and require $O(\log n)$ time to descend the tree to find these nodes.