Lecture 18: More on kd-trees

Reading: Today's material is discussed in Chapt. 2 of Samet's book on spatial data structures.

Deletion from a kd-tree: As with insertion we can model the deletion code after the deletion code for unbalanced binary search trees. However, there are some interesting twists here. Recall that in the 1-dimensional case we needed to consider a number of different cases. If the node is a leaf we just delete the node. Otherwise, its deletion would result in a "hole" in the tree. We need to find an appropriate replacement element. In the 1-dimensional case, we were able to simplify this if the node has a single child (by making this child the new child of our parent). However, this would move the child from an even level to an odd level, or vice versa. This would violate the entire structure of the kd-tree. Instead, in both the 1-child and 2-child cases we will need to find a replacement node, from whichever subtree we can.

Let us assume first that the right subtree is non-empty. Recall that in the 1-dimensional case, the replacement key was taken to be the smallest key from the right child, and after using the replacement to fill the hole, we recursively deleted the replacement. How do we generalize this to the multi-dimensional case? What does it mean to find the "smallest" element in a such a set? The right thing to do is to find the point whose coordinate along the current cutting dimension is minimum. Thus, if the cutting dimension is the x-axis, say, then the replacement key is the point with the smallest x-coordinate in the right subtree. We use the `findMin()` function (given last time) to do this for us.

What do we do if the right subtree is empty? At first, it might seem that the right thing to do is to select the maximum node from the left subtree. However, there is a subtle trap here. Recall that we maintain the invariant that points whose coordinates are equal to the cutting dimension are stored in the right subtree. If we select the replacement point to be the point with the maximum coordinate from the left subtree, and if there are other points with the same coordinate value in this subtree, then we will have violated our invariant. There is a clever trick for getting around this though. For the replacement element we will select the *minimum* (not maximum) point from the left subtree, and we move the left subtree over and becomes the new right subtree. The left child pointer is set to null. The code and an example are given below. As before recall that `cd+1` means that we increment the cutting dimension modulo the dimension of the space.

kd-tree Deletion

```
KDNode delete(Point x, KDNode t, int cd) {
    if (t == null)                          // fell out of tree
        ...error deletion of nonexistant point...
    else if (x == t.data) {                 // found it
        if (t.right != null) {              // take replacement from right
            t.data = findMin(t.right, cd, cd+1)
            t.right = delete(t.data, t.right, cd+1)
        }
        else if (t.left != null) {          // take replacement from left
            t.data = findMin(t.left, cd, cd+1)
            t.right = delete(t.data, t.left, cd+1)
            t.left = null
        }
        else t = null                       // delete this leaf
    }
    else if (x[cd] < t.data[cd])            // search left subtree
        t.left = delete(x, t.left, cd+1)
```

```
        else                                    // search right subtree
            t.right = delete(x, t.right, cd+1)
        return t
    }
```

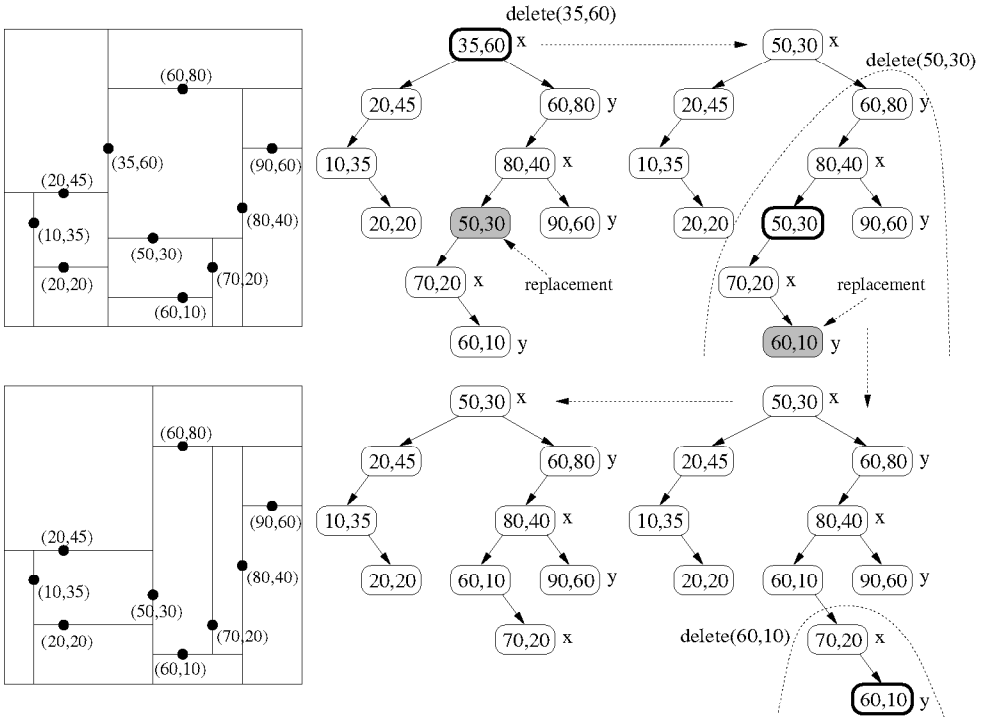

Figure 55: Deletion from a kd-tree.

Recall that in the 1-dimensional case, in the 2-child case the replacement node was guaranteed to have either zero or one child. However this is not necessarily the case here. Thus we may do many 2-child deletions. As with insertion the running time is proportional to the height of the tree.

Nearest Neighbor Queries: Next we consider how to perform retrieval queries on a kd-tree. Nearest neighbor queries are among the most important queries. We are given a set of points S stored in a kd-tree, and a query point q, and we want to return the point of S that is closest to q. We assume that distances are measured using Euclidean distances, but generalizations to other sorts of distance functions is often possible.

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 the figure shown below, the cell containing the query point belongs to node $(70, 30)$ but the nearest neighbor is far away in the tree at $(20, 50)$. 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.

Throughout the search we will maintain an object `close` which contains the partial results of the query. For nearest neighbor search this has two fields, `close.dist`, the distance to the closest point seen so far and `close.point`, the data point that is the closest so far. Initially `close.point = null` and `close.dist = INFINITY`. This structure will be passed in as an argument to the recursive search procedure, and will be returned from the procedure. (Alternatively, it could be passed in as a reference parameter or a pointer to the object and updated.)

We will also pass in two other arguments. One is the cutting dimension `cd`, which as always, cycles among the various dimensions. The other is a rectangle object r that represents the current node's cell. This argument is important, because it is used to determine whether we should visit a node's descendents. For example, if the (minimum) distance from the query point to this cell is greater than the distance from the query point to the closest point seen so far, there is no need to search the subtree rooted at this node. This is how we prune the search space.

When we arrive at a node t, we first determine whether the associated cell r is close enough to provide the nearest neighbor. If not, we return immediately. Next we test whether the associated data point `t.data` is closer to the query q than the current closest. If so, we update the current closest. Finally we search t's subtrees. Rather than just visiting the left subtree first (as we usually do), instead we will visit the subtree whose cell is closer to q. To do this, we test which side of t's splitting line q lies. We recursively search this subtree first and then visit the farther subtree second. Each call updates the nearest neighbor result. It is important to prioritize the visit order, because the sooner we find the nearest neighbor the more effective our pruning step will be. So it makes sense to search first where we think we have the greater chance of finding the nearest neighbor.

In support of this, we need to implement a rectangle class. We assume that this class supports two utilities for manipulating rectangles. Given a rectangle r representing the current cell and given the current cutting dimension `cd` and the cutting value `t.data[cd]`, one method `r.trimLeft()` returns the rectangular cell for the left child and the other `r.trimRight()` returns the rectangular cell for the right child. We leave their implementation as an exercise. We also assume there are functions `distance(q,p)` and `distance(q,r)` that compute the distance from q to a point p and from q to (the closest part of) a rectangle r. The complete code and a detailed example are given below.

kd-tree Nearest Neighbor Searching

```
NNResult nearNeigh(Point q, KDNode t, int cd, Rectangle r, NNResult close) {
    if (t == null) return close                     // fell out of tree
    if (distance(q, r) >= close.dist)               // this cell is too far away
        return close;

    Scalar dist = distance(q, t.data)               // distance to t's data
    if (dist < close.dist)                          // is this point closer?
        close = NNResult(t.data, dist)              // yes, update

    if (q[cd] < t.data[cd]) {                       // q is closer to left child
        close = nearNeigh(q, t.left,  cd+1, r.trimLeft(cd, t.data),  close)
        close = nearNeigh(q, t.right, cd+1, r.trimRight(cd, t.data), close)
```

```
        }
        else {                                          // q is closer to right child
            close = nearNeigh(q, t.right, cd+1, r.trimRight(cd, t.data), close)
            close = nearNeigh(q, t.left,  cd+1, r.trimLeft(cd, t.data),  close)
        }
        return close
    }
```

This structure is typical of complex range queries in geometric data structures. One common simplification (or cheat) that is often used to avoid having to maintain the current rectangular cell r, is to replace the line that computes the distance from q to r with a simpler computation. Just prior to visiting the farther child, we compute the distance from q to t's splitting line. If this distance is greater than the current closest distance we omit the second recursive call to the farther child. However, using the cell produces more accurate pruning, which usually leads to better run times.

The running time of nearest neighbor searching can be quite bad in the worst case. In particular, it is possible to contrive examples where the search visits every node in the tree, and hence the running time is $O(n)$, for a tree of size n. However, the running time can be shown to be 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.

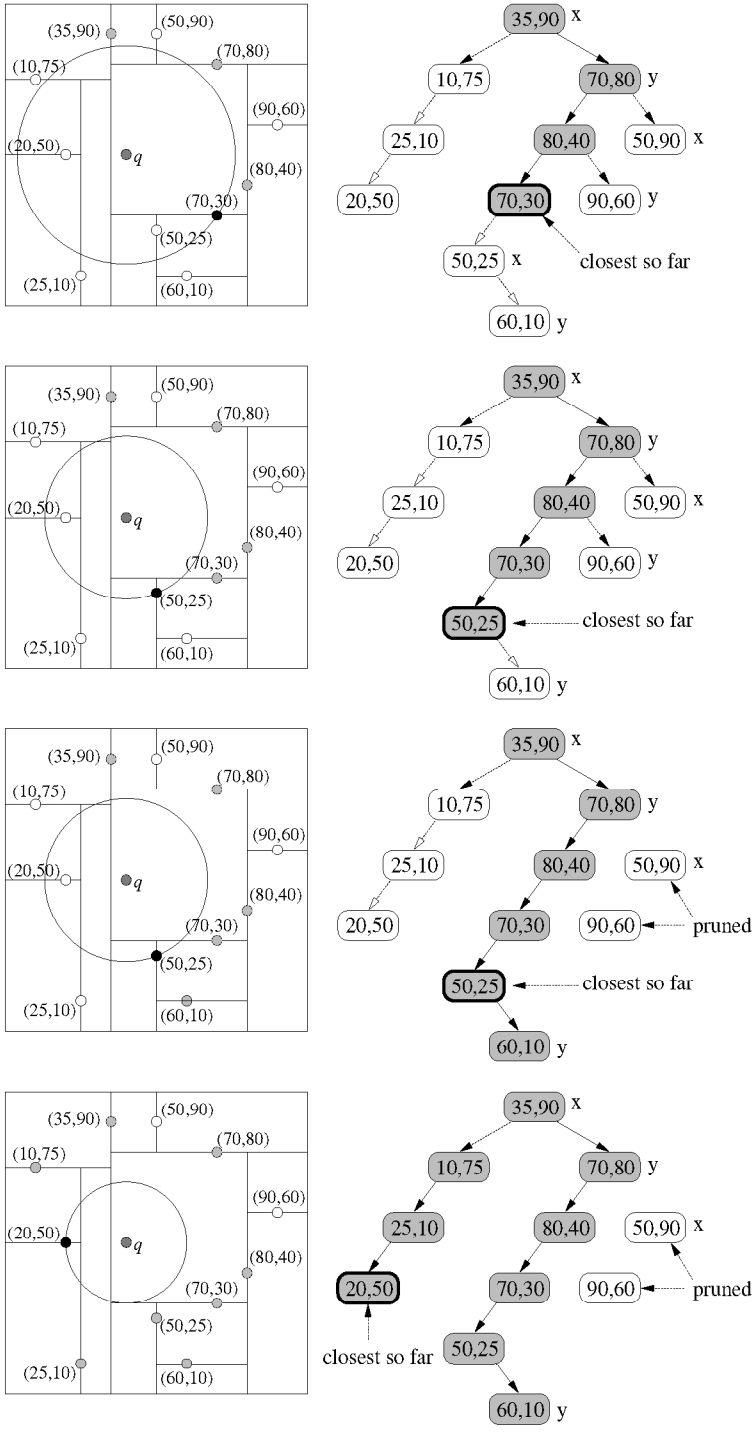

Figure 56: Nearest neighbor search.