Lecture 17: Quadtrees and kd-trees

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

Geometric Data Structures: Geometric data structures are based on many of the concepts presented in typical one-dimensional (i.e. single key) data structures. There are a few reasons why generalizing 1-dimensional data structures to multi-dimensional data structures is not always straightforward. The first is that most 1-dimensional data structures are built in one way or another around the notion of *sorting* the keys. However, what does it mean to sort 2-dimensional data? (Obviously your could sort lexicographically, first by x-coordinate and then by y-coordinate, but this is a rather artificial ordering and does not convey any of the 2-dimensional structure.) The idea of sorting and binary search is typically replaced with a more general notion *hierarchical subdivision*, namely that of partitioning space into local regions using a tree structure. Let S denote a set of geometric objects (e.g. points, line segments, lines, sphere, rectangles, polygons). In addition to the standard operations of inserting and deleting elements from S, the following are example of common queries.

Find: Is a given object in the set?

Nearest Neighbor: Find the closest object (or the closest k objects) to a given point.

Range Query: Given a region (e.g. a rectangle, triangle, or circle) list (or count) all the objects that lie entirely/partially inside of this region.

Ray Shooting: Given a ray, what is the first object (if any) that is hit by this ray.

As with 1-dimensional data structures, the goal is to store our objects in some sort of intelligent way so that queries of the above form can be answered efficiently. Notice with 1-dimensional data it was important for us to consider the dynamic case (i.e. objects are being inserted and deleted). The reason was that the static case can always be solved by simply sorting the objects. Note that in higher dimensions sorting is not an option, so solving these problems is nontrivial, even for static sets.

Point quadtree: We first consider a very simple way of generalizing unbalanced binary trees in the 1-dimensional case to 4-ary trees in the two dimensional case. The resulting data structure is called a *point quadtree*. As with standard binary trees, the points are inserted one by one. The insertion of each point results in a subdivision of a rectangular region into four smaller rectangles, by passing horizontal and vertical *splitting lines* through the new point. Consider the insertion of the following points:

$$(35, 40), (50, 10), (60, 75), (80, 65), (85, 15), (5, 45), (25, 35), (90, 5).$$

The result is shown in the following figure. Each node has four children, labeled NW, NE, SW, and SE (corresponding to the compass directions). The tree is shown on the left and the associated spatial subdivision is shown on the right. Each node in the tree is associated with a rectangular *cell*. Note that some rectangles are special in that they extend to infinity. Since semi-infinite rectangles sometimes bother people, it is not uncommon to assume that everything is contained within one large bounding rectangle, which may be provided by the user when the tree is first constructed. Once you get used to the mapping between the tree and the spatial subdivision, it is common just to draw the subdivision and omit the tree.

We will not discuss algorithms for the point quad-tree in detail. Instead we will defer this discussion to kd-trees, and simply note that for each operation on a kd-tree, there is a similar algorithm for quadtrees.

[1]Copyright, David M. Mount, 2001

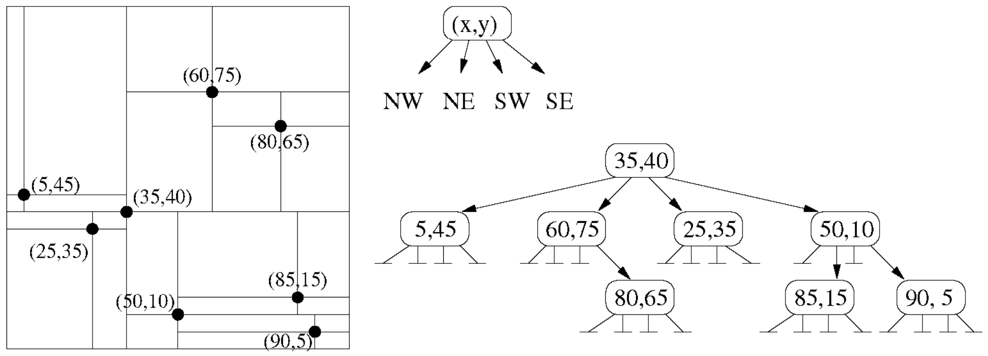

Figure 52: Point quadtree.

Point kd-tree: Point quadtrees can be generalized to higher dimensions, but observe that for each splitting point the number of children of a node increases exponentially with dimension. In dimension d, each node has 2^d children. Thus, in 3-space the resulting trees have 8 children each (called *point octrees*). The problem is that it becomes hard to generalize algorithms from one dimension to another, and if you are in a very high dimensional space (e.g. 20 dimensional space) every node has 2^{20} (or roughly a million) children. Obviously just storing all the null pointers in a single node would be a ridiculous exercise. (You can probably imagine better ways of doing this, but why bother?)

An alternative is to retain the familiar binary tree structure, by altering the subdivision method. As in the case of a quadtree the cell associated with each node is associated with a rectangle (assuming the planar case) or a hyper-rectangle in d-dimensional space. When a new point is inserted into some node (equivalently into some cell) we split the cell by a horizontal or vertical *splitting line*, which passes through this point. In higher dimensions, we split the cell by a $(d-1)$ dimensional hyperplane that is orthogonal to one of the coordinate axes. In any dimension, the split can be specified by giving the *splitting axes* (x, y or whatever), also called the *cutting dimension*. The associated splitting value is the associated coordinate of the data point.

The resulting data structure is called a *point kd-tree*. Actually, this is a bit of a misnomer. The data structure was named by its inventor Jon Bentley to be a *2-d tree* in the plane, a *3-d tree* in 3-space, and a *k-d tree* in dimension k. However, over time the name "kd-tree" became commonly used irrespective of dimension. Thus it is common to say a "kd-tree in dimension 3" rather than a "3-d tree".

How is the cutting dimension chosen? There are a number of ways, but the most common is just to alternate among the possible axes at each new level of the tree. For example, at the root node we cut orthogonal to the x-axis, for its children we cut orthogonal to y, for the grandchildren we cut along x, and so on. In general, we can just cycle through the various axes. In this way the cutting dimension need not be stored explicitly in the node, but can be determined implicitly as the tree is being traversed. An example is shown below (given the same points as in the previous example). Again we show both the tree and the spatial subdivision.

The tree representation is much the same as it is for quad-trees except now every node has only two children. The left child holds all the points that are less than the splitting point along the cutting dimension, and the right subtree holds those that are larger. What should we do if the point coordinate is the same? A good solution is just to pick one subtree (e.g. the right)

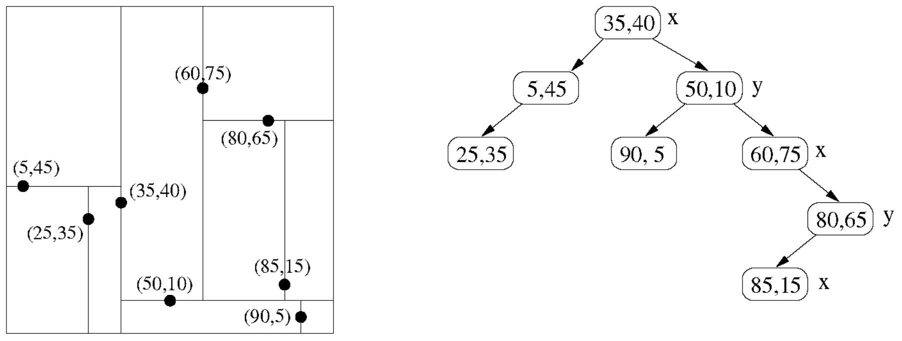

Figure 53: Point kd-tree decomposition.

and store it there.

As with unbalanced binary search trees, it is possible to prove that if keys are inserted in random order, then the expected height of the tree is $O(\log n)$, where n is the number of points in the tree.

Insertion into kd-trees: Insertion operates as it would for a regular binary tree. The code is given below. The initial call is `root = insert(x, root, 0)`. We assume that the dimensions are indexed $0, 1, \ldots, DIM - 1$. The parameter *cd* keeps track of the current cutting dimension. We advance the cutting dimension each time we descend to the next level of the tree, by adding one to its value modulo *DIM*. (For example, in the plane this value alternates between 0 and 1 as we recurse from level to level.) We assume that the + operator has been overloaded for the cutting dimension to operate modulo *DIM*. Thus, when we say `cd+1` in the pseudocode, this a shorthand for `(cd+1) % DIM`. Notice that if a point has the same coordinate value as the cutting dimension it is always inserted in the right subtree.

kd-tree Insertion

```
KDNode insert(Point x, KDNode t, int cd) {
    if (t == null)                          // fell out of tree
        t = new KDNode(x)
    else if (x == t.data)                   // duplicate data point?
        ...error duplicate point...
    else if (x[cd] < t.data[cd])            // left of splitting line
        t.left = insert(x, t.left, cd+1)
    else                                    // on or right of splitting line
        t.right = insert(x, t.right, cd+1)
    return t
}
```

FindMin and FindMax: We will discuss deletion from kd-trees next time. As a prelude to this discussion, we will discuss an interesting query problem, which is needed as part of the deletion operation. The problem is, given a node t in a kd-tree, and given the index of a coordinate axis i we wish to find the point in the subtree rooted at t whose i-th coordinate is minimum. This is a good example of a typical type of query problem in kd-trees, because it shows the fundamental feature of all such query algorithms, namely to recurse only on subtrees that might lead to a possible solution.

For the sake of concreteness, assume that the coordinate is $i = 0$ (meaning the x-axis in the plane). The procedure operates recursively. When we arrive at a node t if the cutting dimension for t is the x-axis, then we know that the minimum coordinate cannot be in t's right subtree. If the left subtree is nonnull, we recursively search the left subtree and otherwise we return t's data point. On the other hand, if t's cutting dimension is any other axis (the y-axis say) then we cannot know whether the minimum coordinate will lie in its right subtree or its left subtree, or might be t's data point itself. So, we recurse on both the left and right subtrees, and return the minimum of these and t's data point. We assume we have access to a function `minimum(p1, p2, p3, i)` which returns the point reference p_1, p_2, or p_3 which is nonnull and minimum on coordinate i. The code and a figure showing which nodes are visited are presented below.

findMin: A Utility for kd-tree Deletion

```
                                             // find minimum along axis i
Point findMin(KDNode t, int i, int cd) {
    if (t == null) return null               // fell out of tree
    if (cd == i) {                           // i == splitting axis
        if (t.left == null) return t.data    // no left child?
        else return findMin(t.left, i, cd+1)
    }
    else {                                   // else min of both children and t
        return minimum(t.data,
                    findMin(t.left,  i, cd+1),
                    findMin(t.right, i, cd+1),
                    i)
    }
}
```

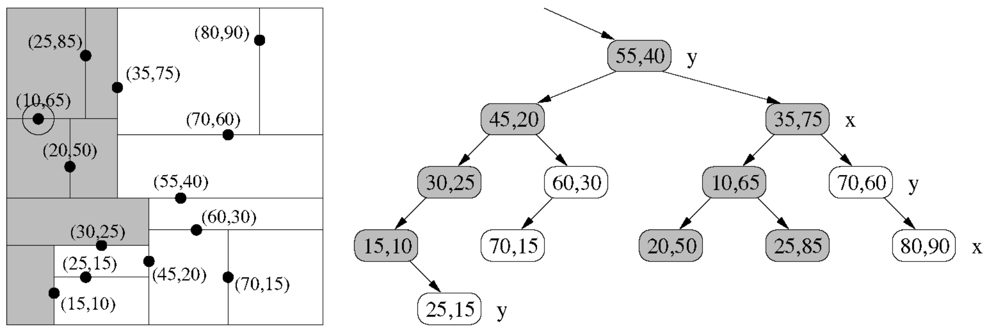

Figure 54: Example of findMin() for x-axis.