CMSC 420: Lecture 13
Point quadtrees and kd-trees

Geometric Data Structures: In today's lecture we move in a new direction by covering a number of data structures designed for storing multi-dimensional geometric data. Geometric data structures are fundamental to the efficient processing of data sets arising from myriad applications, including spatial databases, automated cartography (maps) and navigation, computer graphics, robotics and motion planning, solid modeling and industrial engineering, particle and fluid dynamics, molecular dynamics and drug design in computational biology, machine learning, image processing and pattern recognition, computer vision.

Fundamentally, our objective is to store a large datasets consisting of geometric objects (e.g., points, lines and line segments, simple shapes (such as balls, rectangles, triangles), and complex shapes such as surface meshes) in order to answer queries on these data sets efficiently. While some of our explorations will involve delving into geometry and linear algebra, fortunately most of what we will cover assumes no deep knowledge of geometric objects or their representations. Given a collection of geometric objects, there are numerous types of queries that we may wish to answer.

Nearest-Neighbor Searching: Store a set of points so that qiven a query point q, it is possible to find the closest point of the set (or generally the closest k objects) to the query point (see Fig. 1(a)).

Range Searching: Store a set of points so that given a query region R (e.g., a rectangle or circle), it is possible to report (or count) all the points of the set that lie inside this region (see Fig. 1(b)).

Point location: Store the subdivision of space into disjoint regions (e.g., the subdivision of the globe into countries) so that given a query point q, it is possible determine the region of the subdivision containing this point efficiently (see Fig. 1(c)).

Intersection Searching: Store a collection of geometric objects (e.g., rectangles), so that given a query consisting of an object R of this same type, it is possible to report (or count) all of the objects of the set that intersect the query object (see Fig. 1(d)).

Ray Shooting: Store a collection of object so that given any query ray, it is possible to determine whether the ray hits any object of the set, and if so which object does it hit first.

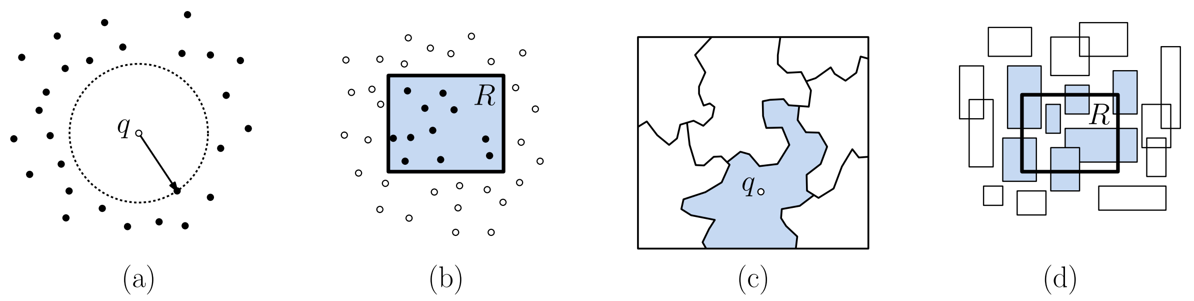


Fig. 1: Common geometric queries: (a) nearest-neighbor searching, (b) range searching, (c) point location, (d) intersection searching.

In all cases, you should imagine the size n of the set is huge, consisting for example of millions of objects, and the objective is to answer the query in time that is significantly smaller than n, ideally $O(\log n)$. We shall see that it is not always possible to achieve efficient query times with storage that grows linearly with n. In such instances, we would like the storage to slowly, for example, $O(n \log n)$. As with 1-dimensional data structures, it will also be desirable to provide dynamic updates, allowing for the insertion and deletion of objects.

No Total Ordering: While we shall see that many of the ideas that we applied in the design of 1-dimensional data structures can be adapted to the design of multi-dimensional data structure, there is one fundamental challenge that we will face. Almost all 1-dimensional data structures exploit the fact that the data are drawn from a total order. The existence of such a total ordering is critical to all the tree-based search structures we studied as well as skip lists.

The only exception to this is hashing. But hashing is applicable only when we are searching for exact matches. In typical geometric queries (as all the ones described above) exact matching does not apply. Instead we are interested in notions such as "close to" or "contained within" or "overlapping with," none of which are amenable to hashing.

Point representations: Let's first say a bit about representation and notation. We will assume that each point p_i is expressed as a d-element vector, that is $p_i = (p_{i,1}, \ldots, p_{i,d})$. To simplify our presentation, we will usually describe our data structures in a 2-dimensional context, but the generalization to higher dimensions will be straightforward. For this reason, we may sometimes refer to a point's in terms of its (x, y)-coordinates, for example, $p = (p_x, p_y)$, rather than $p = (p_1, p_2)$.

While mathematicians label indices starting with 1, programming languages like Java prefer to index starting with 0. Therefore, in Java, each point `p` is represented as a d-element vector:

```
float[][] p = new float[n][d]; // array of n points, each a d-element vector
```

In this example, the points are `p[0]`, `p[1]`, `p[n-1]`, and the coordinates of the ith point are given by `p[i][0]`, `p[i][1]`, `p[i][d-1]`.

A better approach would be to define a class that represents a point object. An example of a simple `Point` object can be found in the code block below. We will assume this in our examples. Java defines a 2-dimensional point object, called `Point2d`.

Simple Point class

```
public class Point {
    private float[] coord; // coordinate storage

    public Point(int dim) {  /* construct a zero point */ }

    public int getDim() { return coord.length; }
    public float get(int i) { return coord[i]; }

    public void set(int i, float x) { coord[i] = x; }

    public boolean equals(Point other) { /* compare with another point */ }
    public float distanceTo(Point other) { /* compute distance to another point */ }

    public String toString() { /* convert to string */  }
}
```

Now, your point set could be defined as an array of points, for example, `Point[] pointSet = new Point[n]`. Note that although we should use `pt.get(i)` to get the ith coordinate of a point `pt`, we will often be lazy in our code samples, and just write `pt[i]` instead.

Point quadtree: Let us first consider a natural way of generalizing unbalanced binary trees in the 1-dimensional case to a d-dimensional context. Suppose that we wish to store a set $P = \{p_1, \ldots, p_n\}$ of n points in d-dimensional space. In binary trees, each point naturally splits the real line in two. In two dimensions if we run a vertical and horizontal line through the point, it naturally subdivides the plane into four *quadrants* about this point. (In general d-dimensional space, we consider d axis-parallel hyperplanes passing through the point. These subdivide space into 2^d *orthants*.)

To simplify the presentation, let us assume that we are working in 2-dimensional space. The resulting data structure is called a *point quadtree*. (In dimension three, the corresponding structure is naturally called an *octtree*. As the dimension grows, it is too complicated to figure out the proper term for the number of children, and so the term *quadtree* is often used in arbitrary dimensions, even though the outdegree of each node is 2^d, not four.)

Each node has four (possibly null) children, corresponding to the four quadrants defined by the 4-way subdivision. We label these according to the compass directions, as `NW`, `NE`, `SW`, and `SE`. In terms of implementation, you can think of assigning these the values 0, 1, 2, 3, and use them as indices to a 4-element array of children pointers.

As with standard (unbalanced) binary trees, points are inserted one by one. We descend through the tree structure in a natural way. For example, we compare the newly inserted point's x and y coordinates to those of the root. If the x is larger and the y is smaller, we recurse on the `SE` child. The insertion of each point results in a subdivision of a rectangular region into four smaller rectangles. Consider the insertion of the following points (see Fig. 2):

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

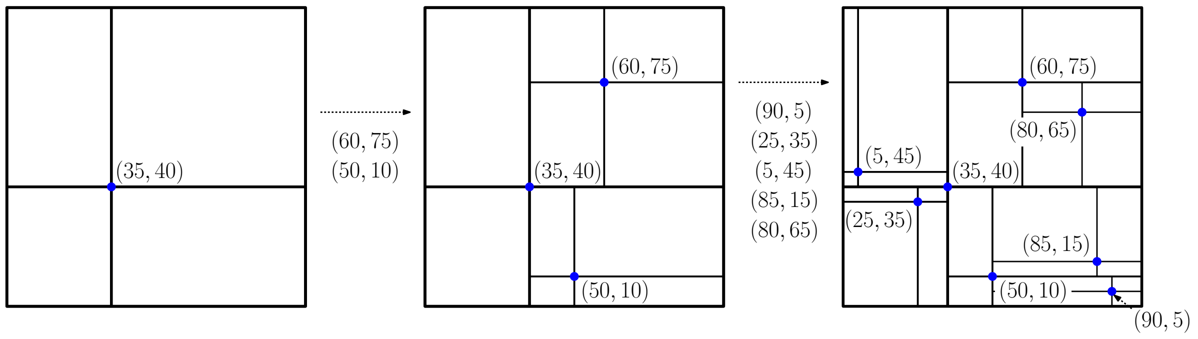
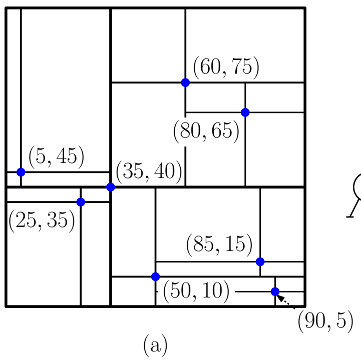


Fig. 2: Point quadtree subdivision.

The final subdivision and tree structure are shown in Fig. 3.

Each node in the tree is naturally associated with a rectangular region of space, which we call its *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.



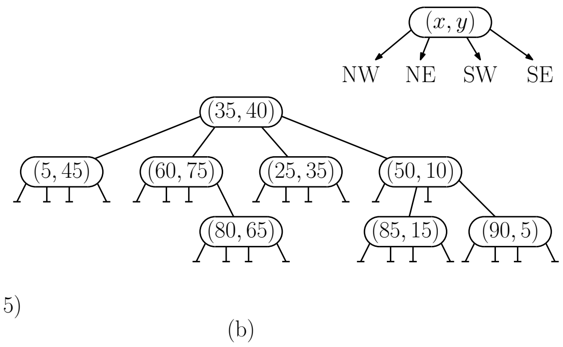


Fig. 3: Point quadtree.

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

Point kd-tree: As observed above, point quadtrees can be generalized to higher dimensions, the number of children grows exponentially in the dimension, as 2^d. For example, if you are working in 20-dimensional space, every node has 2^{20}, or roughly a million children! Clearly, the simple quadtree idea is not scalable to very high dimensions. Next, we describe an alternative data structure, that always results in a binary tree.

As in the case of a quadtree, the cell associated with each node is an axis-aligned (hyper-)rectangle. When a new point is inserted into some leaf node's cell, we split the cell by a horizontal or vertical *splitting line*, which passes through this point (or generally a $(d-1)$-dimensional axis-aligned hyperplane). The split is specified by its *cutting dimension*, `cutDim`, which can be represented as an integer from 0 to $d-1$ and its *cutting value.* As with quadtrees, the cut is made through the point, so the cutting value is `p.point[cutDim]`. By convention, points whose coordinate value is *strictly smaller* than the cutting value (`pt[cutDim] < point[cutDim]`) are stored in the left subtree and those with values *greater than or equal* are in the right subtree.

```
class KDNode {                          // node in a kd-tree
    Point point                         // splitting point
    int cutDim                          // cutting dimension (optional)
    KDNode left, right                  // children

    KDNode(Point point, int cutDim) {   // constructor
        this.point = point
        this.cutDim = cutDim
        left = right = null
    }

    boolean inLeftSubtree(Point pt) {    // is pt in left subtree?
        return pt[cutDim] < point[cutDim]
    }
}
```

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 to Cut Space? There are a number of ways to select the cutting dimension. The most common is just to alternate (or generally cycle) among the possible axes at each new level of the tree. For example, at the root node we cut orthogonal to the x-axis (or 0th coordinate), for its children we cut orthogonal to y (or 1st coordinate), for the grandchildren we cut again along x, and so on. In higher dimensions, we cycle through the various dimensions. An example is shown in Fig. 4. Note that when this is done, we do not need to explicitly store the cutting dimension in each node, since it can be computed on the fly as we traverse the tree.

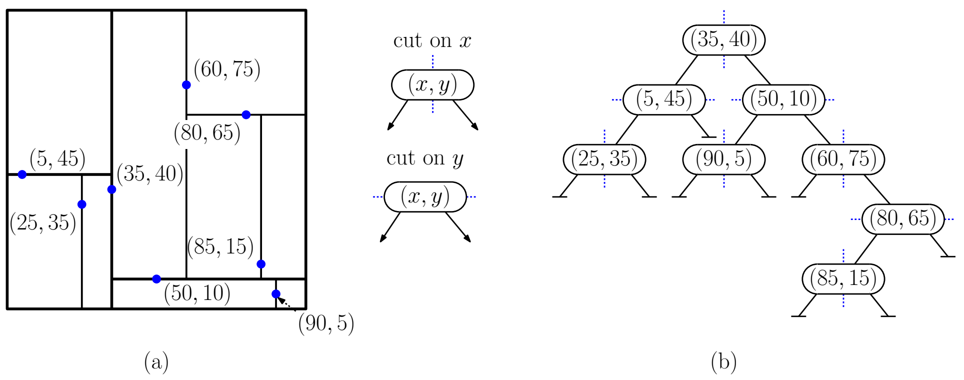


Fig. 4: Point kd-tree decomposition.

We will assume this cyclic method of choosing the cutting dimension in our examples, but there are better ways. For example, Bentley proposed that the cutting dimension should be chosen based on the point distribution. The cut should be made perpendicular to the longest side of the point set's bounding box. He also suggested that the cut should be made through the median point of the set. This tends to produce nice "fat" cells and has height $O(\log n)$.

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 search tree. We descend the tree until falling out, and then we create a node containing the point and assign its cutting dimension by whatever policy is used by the tree. The principal utility function is presented in the following code block. The function takes three arguments, the point `pt` being inserted, the current node `p`, and the cutting dimension of the newly created node. The initial call is `root = insert(pt, root, 0)`.

Observe that whenever we make a recursive call, we advance the cutting dimension by one (`cd + 1`) but we wrap around by modding this with the dimension of the space. An example is shown in Fig. 5, where we insert the point $(50, 90)$ into the kd-tree of Fig. 4. We descend the tree until we fall out on the left subtree of node $(60, 75)$. We create a new node at this point, and the cutting dimension cycles from the parent's x-cutting dimension (`cutDim = 0`) to a y-cutting dimension (`cutDim = 1`).

kd-tree Insertion

```
KDNode insert(Point pt, KDNode p, int cd) { // insert pt in subtree p with cutDim cd
    if (p == null) {                                       // fell out of tree
        p = new KDNode(pt, cd)                             // create new leaf
    } else if (p.point.equals(pt)) {
        throw Exception("Error - duplicate point")
    } else if (p.inLeftSubtree(pt)) {                      // insert into left subtree
        p.left = insert(pt, p.left, (cd + 1) % dim)
    } else {                                               // insert into right subtree
        p.right = insert(pt, p.right, (cd + 1) % dim)
    }
    return p
}
```

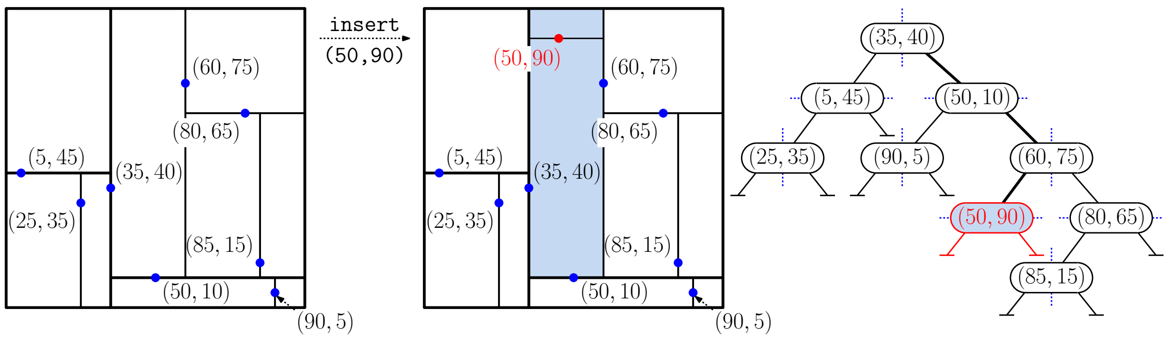


Fig. 5: Inserting (50, 90) into the point kd-tree of Fig. 4.

Analysis: The space needed by the kd-tree to store n points in d-dimensional space is $O(n)$, which is optimal. (We treat d like a constant, independent of n. In fact, it takes $O(dn)$ space to store the coordinates, but since d is a constant, we can ignore the d factor.)

The height analysis of the kd-tree is essentially the same as that of the unbalanced binary tree. If n points are inserted in random order, then the height of the tree will be $O(\log n)$ in expectation. (Below, we discuss deletion. Because we chose replacement nodes in a biased way, always from the right subtree, it is reasonable that the same systematic bias issues that leads to $O(\sqrt{n})$ tree height over a long series of random insertions and deletions.)

Probably the best way to maintain balance in a kd-tree is to simply rebuild unbalanced subtrees, similar to what we did with *scapegoat trees*. This raises the question of how to build a well-balanced tree for a fixed set of points. Suppose that we use the method of cycling the cutting dimension from level to level of the tree to build a kd-tree for a point set P. At the root level, we could choose the splitting point to be the median of P according to x-coordinates. Then, after partitioning the set about this point, into say P_L and P_R, the splitting value for each set would be the respective medians but according to the y-coordinates. By doing this, we guarantee that the number of points in each subtree is essentially half that of its parent, and this implies that the overall tree height is $O(\log n)$.

By the way, this raises an interesting computational question. We know that it is possible to build a 1-dimensional tree from a sorted point set in $O(n \log n)$ time, by repeatedly splitting on the median. Can you generalize this to construct a perfectly balanced 2-dimensional kd-tree also in $O(n \log n)$ time. The tricky issue is that sorting on x does not help you in finding the y-splits, and sorting on y does not help you with the x splits. This is an interesting computational problem to think about. (The answer is that it is possible to build such a tree in $O(n \log n)$ time, but it takes a bit of cleverness. We will leave this as an exercise.)

Finding Replacements for Deletion: We will next discuss deletion from kd-trees. As we saw with deletion in standard binary search trees, an issue that will arise when deleting a point from the middle of the tree is what to use in place of this node, that is, the *replacement point*. It is not as simple as selecting the next point in an inorder traversal of the tree, since we need a point that satisfies the necessary geometric conditions.

Suppose that we wish to delete a point in node `p` where `p.cutDim == 0` (that is, vertical cut). An appropriate choice for the replacement point is the point of `p.right` that has the smallest x-coordinate. (What do we do if the right child is `null`? We'll come to this later.) Finding such a point is a nice exercise, since it illustrates how programming is done with kd-trees.

Let us derive a procedure `findMin(p, i)` that computes the point in the subtree rooted at node p that has the smallest ith coordinate. The procedure operates recursively. When we arrive at a node p, if the cutting dimension matches i, given that the subtrees are ordered by the ith coordinate, we know that the minimum cannot lie in the right subtree. If the left child is non-null, then the desired point is there, and we recursively search this subtree (see Fig. 6(a)). If the left child is null, we return p's associated point (see Fig. 6(b)).

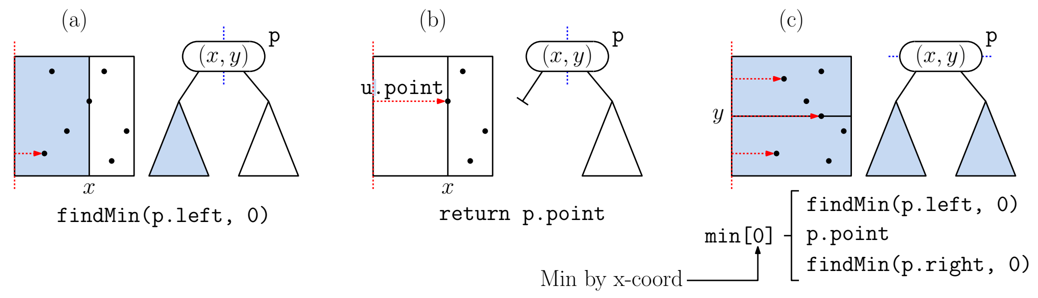


Fig. 6: Cases in `findMin`.

Finally, if the cutting dimension does not equal i, then the minimum may either in the left/right subtrees or might be `p.point` itself. We check all three cases and return the minimum along the ith coordinate (see Fig. 6(c)). The code is presented below. Let's imagine that we have utility function `min[i](q1, q2)`, which returns whichever point `q1` or `q2` that has the smaller `i`-th coordinate. (A null point is never favored over a non-null point.)

Find the minimum point in subtree along ith coordinate

```
Point findMin(KDNode p, int i) {                          // get min point along dim i
    if (p == null) return null                            // fell out of tree?
    if (p.cutDim == i) {                                  // cutting dimension matches i?
        if (p.left == null)                               // no left child?
            return p.point                                // use this point
        else
            return findMin(p.left, i)                     // get min from left subtree
    } else {                                              // it may be in either side
        return min[i](findMin(p.left), p.point, findMin(p.right))
    }
}
```

Fig. 7 presents an example of the execution of this algorithm to find the point with the minimum x-coordinate in the subtree rooted at $(55, 40)$. Since this node splits horizontally, we need to visit both of its subtrees to find their minimum x values. (These will be $(15, 10)$ for the left subtree and $(10, 65)$ for the right subtree.) These values are then compared with the point at the root to obtain the overall x-minimum point, namely $(10, 65)$. Observe that because the subtrees at $(45, 20)$ and $(35, 75)$ both split on the x-coordinate, and we are looking for the point with the minimum x-coordinate, we do not need to search their right subtrees. The nodes visited in the search are shaded in blue.

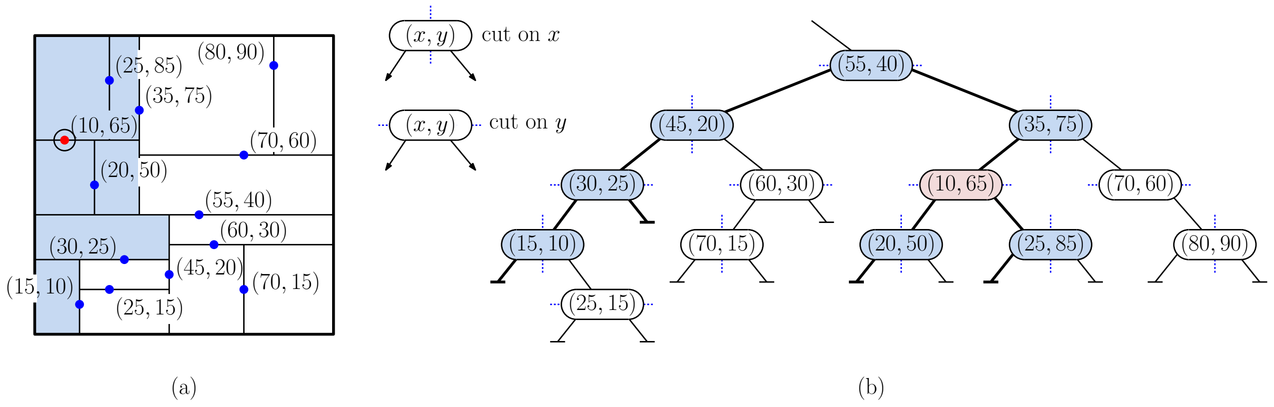


Fig. 7: Example of `findMin` when $i = 0$ (the x-coordinate) on the subtree rooted at $(55, 40)$. The function returns $(10, 65)$.

Deletion from a kd-tree: As with insertion, deletion is similar as for unbalanced binary search trees. However, there is an interesting twist 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*. 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, and this would violate our assumption that the cutting dimensions cycle among the coordinates. (Note this might not be an issue for some implementations of the kd-tree, where the cutting dimension is selected by some other policy, but to keep things as clean as possible, our deletion procedure will not alter a node's cutting dimension.)

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? The proper 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 above) to do this.

On the other hand, what 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. **Tricky! Be sure you understand why this works.** The code is given below.

kd-tree Deletion

```
KDNode delete(Point pt, KDNode p) {
    if (p == null) {                              // fell out of tree?
        throw Exception("Error - Point does not exist");
    } else if (p.point.equals(pt)) {              // found it
        if (p.right != null) {                    // can replace from right
            p.point = findMin(p.right, p.cutDim)  // find and copy replacement
            p.right = delete(p.point, p.right)    // delete from right
        } else if (p.left != null) {              // can replace from left
            p.point = findMin(p.left, p.cutDim)   // find and copy replacement
            p.right = delete(p.point, p.left)     // delete left but move to right!!
            p.left = null                         // left subtree is now empty
        } else {                                  // deleted point in leaf
            p = null                              // remove this leaf
        }
    } else if (p.inLeftSubtree(pt)) {
        p.left = delete(pt, p.left)               // delete from left subtree
    } else {                                      // delete from right subtree
        p.right = delete(pt, p.right)
    }
    return p
}
```

An example of the operation of this deletion algorithm is presented in Fig. 8. The original objective is to delete the point $(35, 60)$. This is at the root of the tree. Because the cutting dimension is vertical, we search its right subtree to find the point with the minimum x-coordinate, which is $(50, 30)$. The point is copied to the root. Note that when this happens, the associated subdivision of space immediately changes. (See the top right of Fig. 8). The picture is a bit wonky, because $(50, 30)$ appears twice in the tree. Once as a vertical splitted at the root and later as a horizontal splitter.

We then recursively delete $(50, 30)$ from the root's right subtree. This recursive call then seeks the node p containing $(50, 30)$. Note that this node has no right child, but unlike standard binary search trees, we cannot simply unlink it from the tree (for the reasons described above). Instead, we observe that its cutting dimension is horizontal, and we search for the point with the minimum y-coordinate in p's left subtree, which is $(60, 10)$. We copy $(60, 10)$ to p. Again, the subdivision changes. Node that the vertical splitting line through $(70, 20)$, which was blocked by $(50, 30)$ now extends all the way to the top of the cell.

We now recursively delete $(60, 10)$ from p's left subtree. It is a leaf, so it may simply be unlinked from the tree. (We don't see the change in the subdivision, since $(60, 10)$ already exists in the tree.) Finally, when we return to the parent of $(70, 20)$, since this was a left-side deletion, we need to move the subtree rooted at $(70, 20)$ over to the right side of its parent $(60, 10)$. Following this, we return all the way back to the root without any further changes. (Whew!)

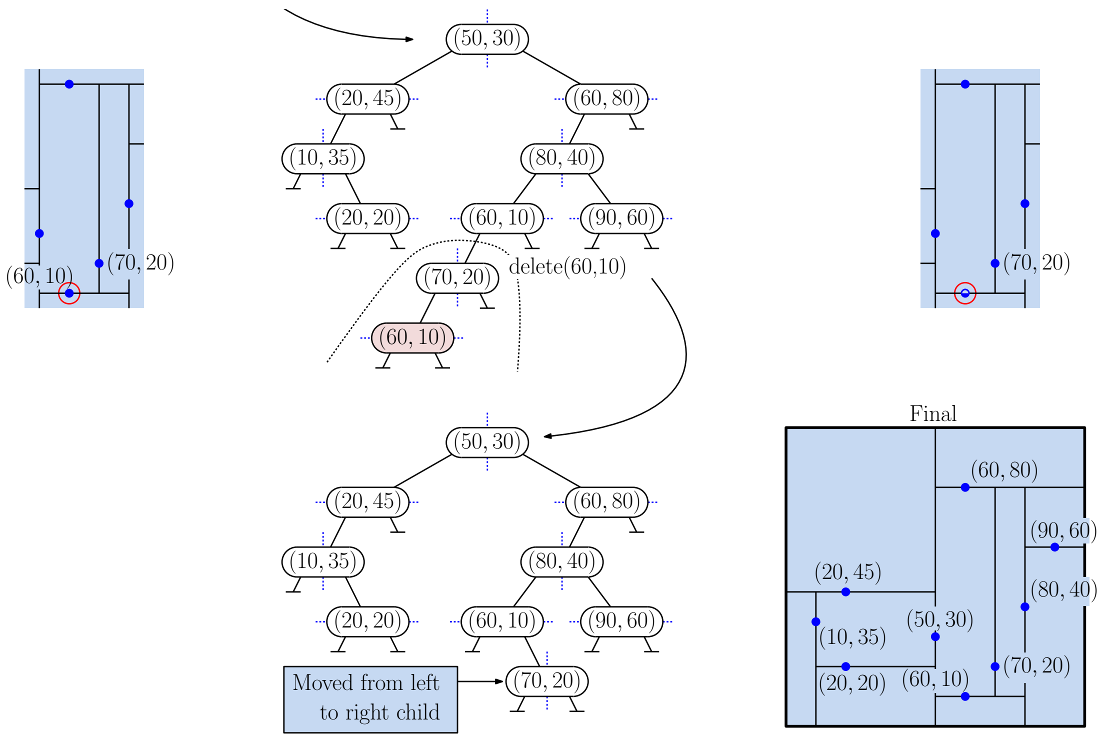


Fig. 8: Deletion from a kd-tree.