# CMSC 420: Lecture X02
# Supplemental: Wrapped kd-Trees

**Overview:** In this lecture, we will discuss how to implement a variant of a kd-tree called *wrapped kd-tree*. Recall that a kd-tree is a data structure based on a hierarchical decomposition of space, using axis-orthogonal splits. A *wrapped kd-tree* involves two modifications to the standard kd-tree.

**Extension:** The tree has two types of nodes, *internal* and *external*. Each external node stores just a single point. In addition to its two children, each internal node stores the splitting information, consisting of a *cutting dimension* and a *cutting value*. The cutting dimension (or `cutDim`) indicates which axis (0 for $x$ and 1 for $y$) is to be split, and the cutting value (or `cutVal`) indicates where the cut occurs along this axis (see Fig. 1). For example, if the cutting dimension is 0 (for $x$) and the cutting value is $z$, then a point $p = (p_x, p_y)$ will be put in the left subtree if $p_x < z$ and in the right subtree if $p_x \geq z$. Note that the cutting value does *not* need to be the coordinate of any point in the tree. The flexibility to chose the cutting value independent of the points is useful because we can select splitting lines to optimize query processing.

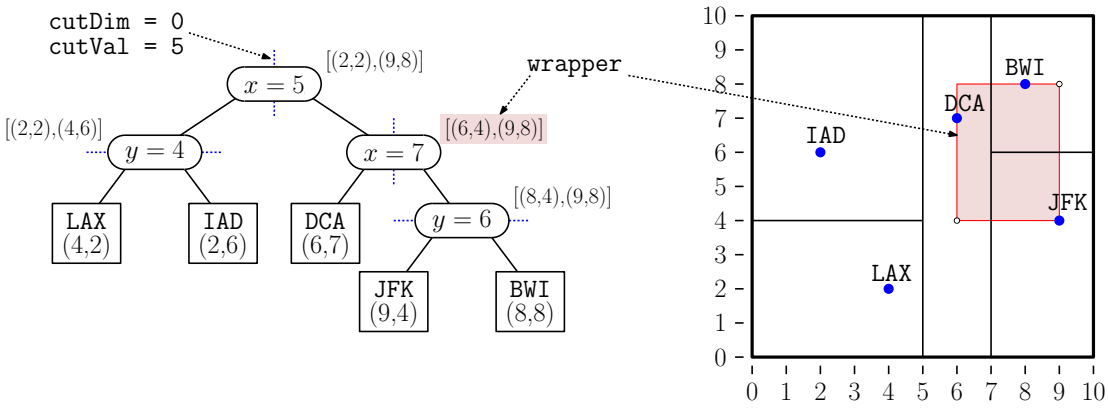


Fig. 1: A wrapped kd-tree and the associated spatial subdivision.

**Wrapping:** In normal kd-trees, each node is associated with an axis-aligned rectangular cell, which is based on the splits that have been made by this node and its ancestors. In a wrapped kd-tree each internal node explicitly stores a *wrapper*, which is defined to be a minimum axis-aligned bounding box for the points in the subtree associated with this node. (In Fig. 1, the wrapper for the internal node "$x = 7$" is highlighted. A wrapper can be represented as any axis-parallel rectangle, say its lower-left and upper-right corner points.) We will provide a class `Rectangle2D` in the skeleton code, and a node's `wrapper` is of this type.

The use of wrappers modestly increases the storage requirements of the data structure, but query processing is often much faster because a wrapper can be significantly smaller than the associated cell. This means that query processing can do a better job of filtering out subtrees that cannot contribute to the search result. This feature becomes more significant as the dimension of the space increases. As points are inserted and deleted from the tree, the wrappers associated with nodes of the tree need to be updated accordingly.

Wrappers are only computed for internal nodes. (Each external node has an "implicit" wrapper consisting of the trivial rectangle that contains the associated point.) We can define a node's wrapper recursively as the smallest axis-aligned rectangle that contains the wrappers of the node's left and right children.

**Points and Rectangles:** We assume throughout that points are in 2-dimensional space and are represented in standard form as a pair of real-valued $(x, y)$-coordinates. We will provide a simple class `Point2D` for representing points. Here are some of the functions it provides. (See the file `Point2D.java` for full information.)

```
public class Point2D {
    public Point2D(float x, float y);     // construct from coordinates
    public Point2D(float[] coord);        // construct from 2-element array
    public Point2D(Point2D p);            // copy constructor
    public float getX();                  // get the x-coordinate
    public float getY();                  // get the y-coordinate
    public float get(int i);              // get i-th coordinate (x = 0, y = 1)
    public void set(int i, float x);      // set i-th coordinate
    public boolean equals(Point2D pt);    // test for equality
    public double distance(Point2D pt);   // Euclidean distance to pt
    public double distanceSq(Point2D pt); // squared distance to pt
}
```

For the sake of testing and debugging, we will assume that each point in our data structure is also associated with a string *label*. Such a point is called a *labeled point* or `LPoint`. This includes the additional function `String getLabel()`. (See the file `LabeledPoint2D.java` for full information.)

In addition to points, the data structure will make heavy use of axis-aligned rectangles. We will provide a class `Rectangle2D` to represent such objects. This class provides a number of useful functions, including the following. (See the file `Rectangle2D.java` for full information.)

```
public class Rectangle2D {
    public Rectangle2D(Point2D low, Point2D high); // construct from corners
    public Rectangle2D(Rectangle2D r);             // copy constructor
    public Point2D getLow();                       // get lower-left corner
    public Point2D getHigh();                      // get upper-right corner
    public float getWidth(int i);                  // width along dimension i
    public boolean contains(Point2D q);            // do we contain point q?
    public boolean contains(Rectangle2D c);        // do we contain rect c?
    public boolean disjointFrom(Rectangle2D c);    // disjoint from rect c?
    public double distanceSq(Point2D pt);          // squared distance to pt
    public void add(Point2D pt);                   // enlarge to include pt
                        // compute smallest enclosing rectangle for r1 and r2
    public static Rectangle2D union(Rectangle2D r1, Rectangle2D r2);
}
```

**Java Node Structure:** Let's call our wrapped kd-tree `WKDTree`. It will be templated by the point type, which we call `LPoint` for a labeled point, which implements the `LabeledPoint2D` interface described above. The most natural way to implement the two node types is to define an inner node class (within `WKDTree`) and use inheritance. There will be an *abstract* parent class, `Node`, and two subclasses, `InternalNode` and `ExternalNode`. Since `Node` is only a placeholder, its methods are all declared to be "`abstract`." Here is a possible outline for the file `WKDTree.java`:

```
public class WKDTree<LPoint extends LabeledPoint2D> {

    private abstract class Node { // generic node (purely abstract)
        abstract LPoint find(Point2D pt);
        // ... other helper functions omitted
    }

    private class InternalNode extends Node {
        int         cutDim;      // the cutting dimension (0 = x, 1 = y)
        float       cutVal;      // the cutting value
        Rectangle2D wrapper;     // bounding box
        Node        left, right; // children

        LPoint find(Point2D pt) { /* ... */ }
        // ... other helper functions omitted
    }

    private class ExternalNode extends Node {
        LPoint thisPt;           // the associated point

        LPoint find(Point2D pt) { /* ... */ }
        // ... other helper functions omitted
    }

    // ... the rest of the class
}
```

**Tree Operations:** Let us explain how to perform the standard dictionary operations on an extended tree.

`LPoint find(Point2D pt)`: Determines whether a point coordinates `pt` occurs within the tree, and if so, it returns the associated `LPoint`. Otherwise, it returns `null`.

**Top level:** If the root is `null`, then return `null`. Otherwise, invoke the find helper function on the root node.

**Internal:** Apply the find recursively on the appropriate child and return the result. For example, if `pt[cutDim] < cutVal` (we are abusing notation a bit here), apply recurse on the left subtree and otherwise recurse on the right subtree.

**External:** If `pt` is equal to this node's point, we return a reference to the associate value, and otherwise we return `null`.

`void insert(LPoint pt)`: Inserts point `pt` in the tree, throwing an exception if a point with the same coordinates already exists.

**Top level:** If the root is `null` (meaning that the tree is empty), create a single external node containing the `pt` and set the root to point to this node. Otherwise, invoke the insert helper function on the root.

**Internal:** Apply the insert helper function recursively on the appropriate child, and update the child link. Afterwards, update this node's wrapper to include this additional point. (Why update the wrapper *after* the insertion? Does it matter? Think about it.)

**External:** If `pt` matches the point here, throw an exception. Otherwise, create a new external node containing `pt`. We will also need to create a new internal node to

complete the operation. To do this, let $R$ be the bounding rectangle for the current point and the newly added point. Let $i \in \{0, 1\}$ be the dimension along which $R$ is widest. (Because the points are distinct, the wider side must be strictly positive.) Create a new internal node whose cutting dimension is $i$, and whose cutting value bisects this rectangle (see Fig. 2), and assign the external nodes as its children. We also set the internal node's wrapper to $R$.

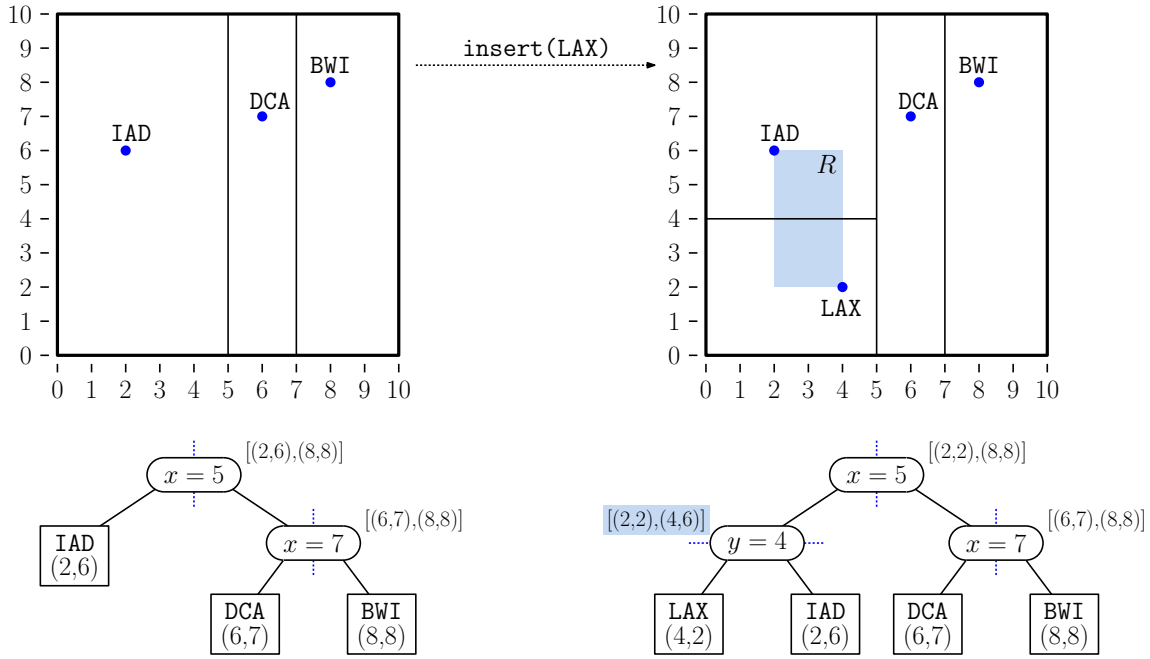


Fig. 2: Inserting into an external node.

For example, in Fig. 2, we insert the point `LAX` with coordinates $(4, 2)$. It falls into the external node with `IAD` with coordinates $(2, 6)$. The rectangle enclosing these two points is taller than wide, so we set the cutting dimension to 1 (for $y$) and split at the midpoint of the two $y$-coordinates $(2 + 6)/2 = 4$, and create the appropriate internal node with these values.

**Example:** In Fig. 3 below, we present an example of a series of insertions into a wrapped kd-tree. We highlight

`void delete(Point2D pt)`: Deletes the entry whose coordinates match those of `pt`, throwing an exception if there is no such point.

**Top level:** If the root is `null` (meaning that the tree is empty), throw a non-existent key exception. Otherwise, invoke the delete helper function on the root.

**External:** The process is the same as deletion from an extended binary search. If the point stored in this node does not match `pt`'s coordinates, we throw a non-existent key exception. Otherwise, we will remove both this node and its parent from the tree, replacing it with the parent's other child `q`. To do this, we first return `null` from the external-node delete function (see Fig. 4).

**Internal:** We apply the delete helper function recursively on the appropriate child and save the return result. If the return result is `null`, this is the signal that we just deleted an external node. In this case, we return a reference to our other child.

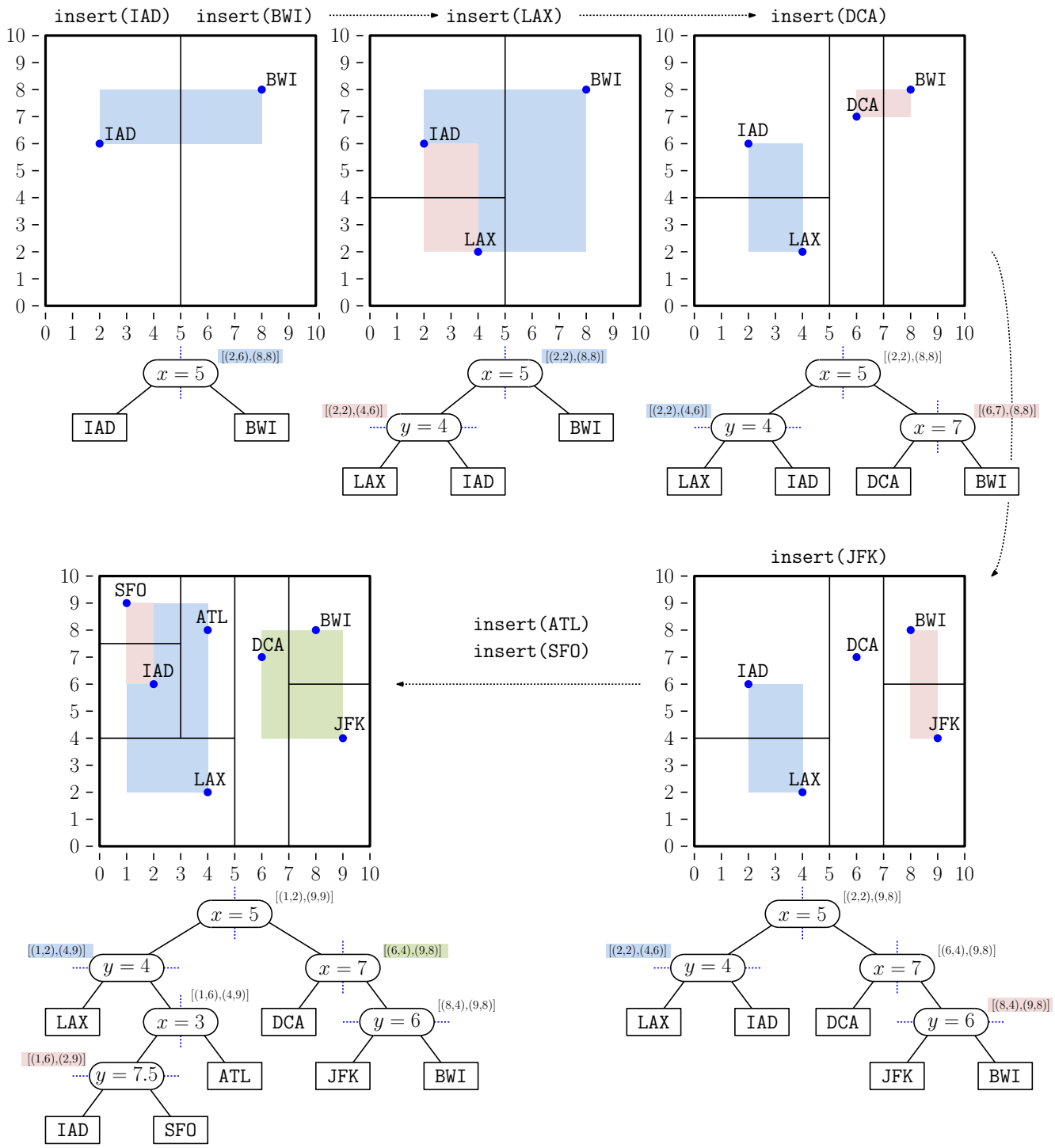


Fig. 3: Insertion of multiple points into a wrapped kd-tree with some of the wrappers highlighted.

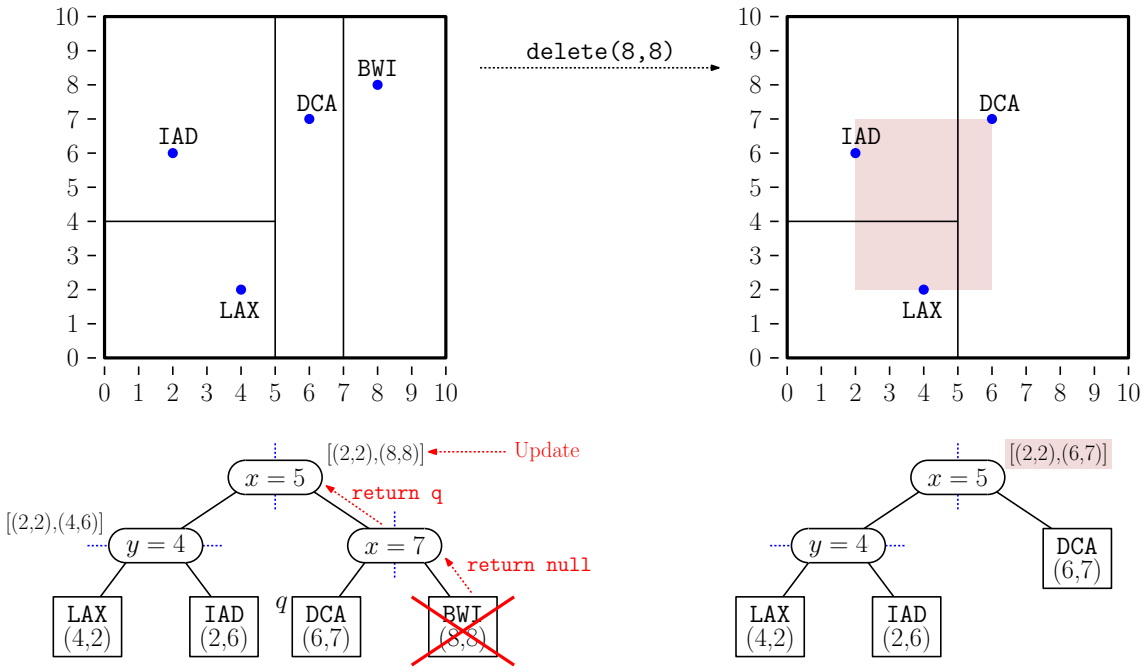


Fig. 4: Deletion at the external node level.

If the return result is not `null`, we set our child pointer to the return result. We also update our wrapper. Assuming inductively that our children's wrappers are correctly computed, we can set our wrapper to the rectangle containing the union of the wrappers of our left and right children.

`LPoint getMinX()`: (We will describe only `getMinX`, since the other three are left/right min/max symmetrical versions.) This returns a reference to the labeled point that is associated with the smallest $x$-coordinate in the tree. If two or more points have the same minimum $x$-coordinate, then the one with the smallest $y$-coordinate is returned. If the dictionary is empty, this returns `null`.

**Top level:** If the root is `null` (meaning that the tree is empty), return `null`. Otherwise, invoke the helper function on the root.

**Internal:** If the cutting dimension is $x$ (0) then we invoke the function recursively on our left child only. (The right child cannot possibly contribute to the final result.) Otherwise, invoke the function on both subtrees. Return the result that has the smaller $x$-coordinate. If the $x$-coordinates are equal, return the point from the left subtree, since it must have the smaller $y$-coordinate.

**External:** Simply return a reference to the point in this node.

By the way, you will notice that all four functions (`getMin/Max/X/Y`) are essentially identical up to minor variations. If you want to challenge your programming skills, you might try implementing a single helper function that does all four. It will have two additional arguments, one that indicates the dimension (0 for $x$ and 1 for $y$) and one that indicates the sense of comparison ($-1$ for min and $+1$ for max). Try to do this using the minimum number of "if" statements. (I needed only two.)

`LPoint findSmallerX(float x)`: (We will describe only `findSmallerX`, since the other three are left/right min/max symmetrical versions.) Among all the points whose $x$-

coordinates are *strictly smaller* than $x$, this returns a reference to the labeled point having the largest $x$-coordinate. If the tree is empty or if there is no point whose $x$-coordinate is smaller than $x$, this returns `null`. If there are ties for the largest $x$-coordinate, return the point with the largest $y$-coordinate.

We will leave the efficient implementation of this function as an exercise for you. Following the a similar approach to the kd-tree algorithm for nearest-neighbor searching, to achieve the best efficiency, you should design your search function to include as one of its arguments the "best" result it has encountered so far. This point would be passed into each recursive call. For example, your helper function might have the following signature:

```
LPoint findSmallerXHelper(float x, LPoint best)
```

Intuitively, if a node's wrapper is to the right of $x$ or strictly to the left of `best`'s $x$-coordinate, we do not need to recurse into this subtree.

For example, in Fig. 5, if we are running the helper function for `findSmallerX(6)` and the current `best` is `LAX`, we know that the answer must lie within the vertical strip between `LAX` and $x = 6$. (In this case `LAX` is the final answer, but the search algorithm doesn't yet know this.) Given this information, we can avoid recursing into the subtree rooted at the internal node "$x = 7$" (shaded in blue) because its wrapper lies entirely to the right of $x = 6$. Also, we can avoid recursing into the subtree rooted at "$x = 2.5$" (shaded in red) because its wrapper is entirely to the left of `LAX`'s $x$-coordinate. (There is no significant harm in visiting these two internal nodes, but we should not go any deeper by invoking the helper function recursively on their children.)

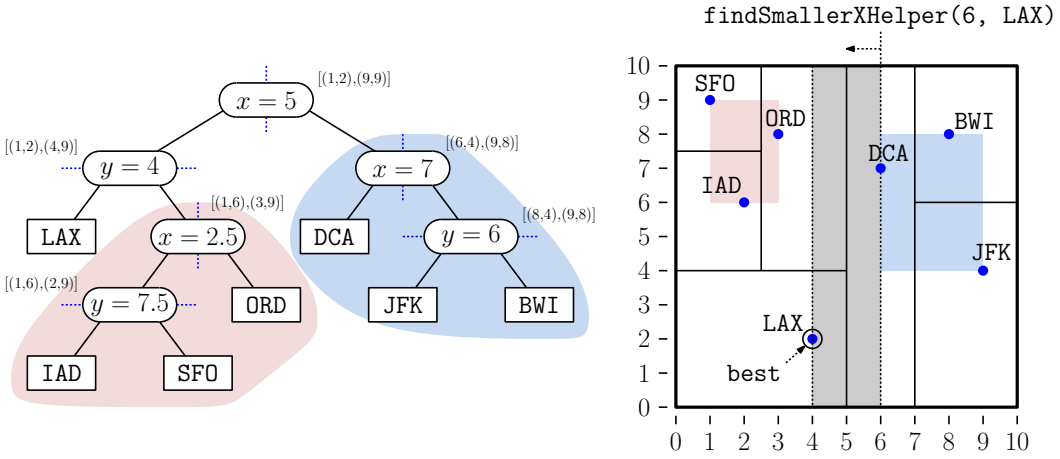


Fig. 5: Pruning the search in the helper function for `findSmallerX`.

As the algorithm visits more nodes and improves its estimate of the "best" point, our ability to prune subtrees becomes increasingly stronger.