Introduction to kd-trees

• Dimension of data is $k$ (but common to say k-d tree of dimension 3 instead of 3d-tree).
• kd-trees are binary trees
• Designed to handle spatial data in a simple way
• For $n$ points, $O(n)$ space, $O(\log n)$ height (if balanced), supports range and nearest-neighbor queries.
• Node consists of
  – Two child pointers,
  – Satellite information (such as name).
  – A key: Either a single float representing a coordinate value, or a pair of floats (representing a dimension of a rectangle)
Basic Idea Behind kd-trees

Construct a binary tree

- At each step, choose one of the coordinate as a basis of dividing the rest of the points
- For example, at the root, choose $x$ as the basis
  - Like binary search trees, all items to the left of root will have the $x$-coordinate less than that of the root
  - All items to the right of the root will have the $x$-coordinate greater than (or equal to) that of the root
- Choose $y$ as the basis for discrimination for the root’s children
- And choose $x$ again for the root’s grandchildren

Note: Equality (corresponding to right child) is significant
Example: Construct kd-tree Given Points

• Coordinates of points are (35, 90), (70, 80), (10, 75), (80, 40), (50, 90), (70, 30), (90, 60), (50, 25), (25, 10), (20, 50), and (60, 10)

• Points may be given one a time, or all at once.

• Data best visualized as shown below
Example: kdtree Insertion
Building: Dynamic Insertion

```java
KDNode insert (point p, KDNode t, int cd) {
    if (t == null) t = new KDNode (p);
    // sets up node.data.x and node.data.y
    else if (p == t.data) ... // duplicate
    else if (p.cd < t.data.cd)
        t.left = insert (p, t.left, cd+1);
    else t.right = insert (p, t.right, cd+1);
    return t;
}
```

- Initial call: `root = insert (p, root, 0);`
- Each node is associated with a rectangular region
- Tree is “balanced” if points are given in random order
- Or if all points are given in advance
Building: The Static Case

• Assume points are sorted on both $x$ and $y$ in a composite array $S$
• $S[x]$ corresponds to a list of points sorted by $x$.

```java
 KDNode buildTree(SortedArray S, int cd) {
  if (S.empty()) return null
  else if S.singleton() return new KDNode(S[x][0], cd);
  else {
    m = median (S, cd) // median (cutting dimension)
    left = leftPoints(S, cd); right = S − left;
    t = new KDNode(m);
    t.left = buildTree(left, cd+1);
    t.right = buildTree(right, cd+1);
    return t
  }
}
```

• $T(n) = kn + 2T(n/2)$, so the algorithm takes $O(n \log n)$ time.
Remove Requires Finding Minimum

- Given a node, and a cutting dimension, find the node with minimum value (with respect to that cutting dimension)

  ```
  Point findmin (KDNode t, int whichAxis, int cd) {
    if (t == null) return null;
    else if (whichAxis == cd)
      if (t.left == null) return t.data;
      else return findmin(t.left, whichAxis, cd+1)
    else return minimum(t.data, findmin(t.left, whichAxis, cd+1),
                          findmin(t.right, whichAxis, cd+1), i);
  }
  ```

- If tree is balanced, `findmin (root)` takes no more than $O(\sqrt{n})$ time in the worst case.
Example: `findmin(root, x, y)`
Basic Idea Behind Removing

- Want to remove point \( p = (a, b) \)
- First find node \( t \) which has this point
- Node \( t \) discriminates on \( x \) (say)
  - If \( t \) is a leaf node, replace it by null
  - Otherwise, find a replacement node \( r \) with coordinates \((c, d)\)
    - Replace the data at \( t \) by \((c, d)\). The kd-tree structure must not be violated
    - Recursively remove point \( p = (c, d) \)
- Finding the replacement
  - If \( t \) has a right child, use the inorder successor
  - Otherwise minimum value of the left child is appropriately used
Remove Example: Delete Point At Root
Remove Example: Delete Point
Remove Example: Delete Point
Remove Takes $O(\log n)$ Time

```c
KDNode remove (KDNode t, Point p, int cd) {
    if (t == null) return null;
    else if (p.cd < t.data) t.left = remove(t.left, p, cd+1);
    else if (p.cd > t.data) t.right = remove(t.right, p, cd+1);
    else {
        if (t.right == null && t.left == null) return null;
        if (t.right != null)
            t.data = findmin(t.right, cd, cd+1);
        else {
            t.data = findmin(t.left, cd, cd+1);
            t.left = null;
        }
        t.right = remove(t.right, t.data, cd+1);
        return t;
    }
}
```

We expect to delete nodes at leaf level. If tree is balanced, we expect `remove()` to take $O(\log n)$ time.
Remove Example: Delete Point At Root

![Diagram of a tree with points at various locations](image-url)
Remove: Solution