Point Quadtree and kd-Trees
Overview

- So far, we have studying data structure for 1-dimensional search problems
- Many data structure problems involve data in multi-dimensional spaces:
  - Spatial databases, automated cartography (maps), and navigation
  - Computer graphics
  - Robotics and motion planning
  - Solid modeling and industrial engineering
  - Particle and fluid dynamics
  - Molecular dynamics in computational biology
  - Machine learning
  - Image processing, pattern recognition, computer vision
Geometric Queries

Examples

- Nearest-neighbor searching - Find the closest point to a given query point $q$
- Range searching - Report/Count the points lying within a query region $R$
- Point location - Find the region of a subdivision (map) containing a query point $q$
- Intersection searching - Find all the objects that overlap a given query object $R$
- Ray shooting - Find the first (if any) object hit by shooting a ray from a point $p$
Geometric Queries

Similarities and differences

- Multi-dimensional data structures borrow many concepts from 1-dimensional search structure
  - Tree-based structures based on hierarchical partitions
  - Maintaining balance $O(\log n)$ height
  - Use key/splitters to navigate the search space

- Many differences as well
  - There is no natural total order in geometric space.
  - What does it mean to say one point is smaller than another?
Geometric Data

Point Representation

- A point in a d-dimensional space is represented by a d-vector of reals:
  \[ p = (p_1, p_2, \ldots, p_d) \]
- In Java, this could be represented by a d-element array
  ```java
  float[] p = new float[d];
  ```
- While in linear algebra, indexing is from 1 \( \ldots \) \( d \), in Java indexing is from 0 \( \ldots \) \( d - 1 \)
- A set of \( n \) points can be represented as a 2-dimensional array:
  ```java
  float[][] P = new float[n][d];
  ```
A better approach is to encapsulate points in a class structure

```java
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 String toString() { /* convert to string */ }
}
```
Point Quadtree

A Natural Generalization of Binary Search Trees

- How do we generalize a 1-dimensional tree to d-dimensional space?
- **Partition tree:**
  - Each node is associated with a region of space (e.g., a rectangle), its cell
  - Each internal node contains a splitter, which subdivides space into smaller regions
  - Data may be stored in the nodes (as the splitters) or in external nodes (as in extended binary search trees)

- **Point Quadtree:**
  - Each node stores a point (both data and splitter)
  - 2-dimensions: Horizontal and vertical lines through point subdivide cell into 4 quadrants
  - $d$-dimensions: $d$ axis-parallel hyperplanes passing through the point subdivide space into $2^d$ (generalized) orthants
  - Each node has $2^d$ (possibly null) children
Point Quadtrees
A Natural Generalization of Binary Search Trees

- In 2D, quadrants are labeled NW, NE, SW, and SE
  - Example: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5)
- To locate a point, we descend from the root, visiting the appropriate child
The point quadtree works fine in low-dimensional space, but does not scale well to high dimensional space. For example, in $d = 20$ space, each node has a fanout of $2^d \approx 1,048,576$.

Idea: Let’s just split one dimension at a time.

Point kd-tree:
- Each node stores a point (both data and splitter)
- And an index $i$, $0 \leq i \leq d - 1$, the cutting dimension
- For any point $x = (x_0, \ldots, x_{d-1})$:
  - If $x_i < p_i$, $x$ goes in the left subtree
  - If $x_i \geq p_i$, $x$ goes in the right subtree
- Cutting dimension varies by level (e.g., $p$.child.cutDim = $(p$.cutDim+1)$%dim$)
Point $\text{kd}$-Tree

A Binary Partition Tree

- Example: $(35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5)$
- Cutting dimension alternates between $x$ and $y$
class KDNode {
    // node in a kd-tree
    Point point; // splitting point
    int cutDim; // cutting dimension
    KDNode left; // children
    KDNode right;

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

    boolean inLeftSubtree(Point x) { // is x in left subtree?
        return x.get(cutDim) < point.get(cutDim);
    }
}
Point kd-tree

Point insertion

- To insert a point, descend the tree to find the leaf cell containing the point
- Create a new cell and assign its cutting dimension

```java
KDNode insert(Point x, KDNode p, int cutDim) {
    if (p == null) { // fell out of tree
        p = new KDNode(x, cutDim); // create new leaf
    } else if (p.point.equals(x)) {
        throw Exception("duplicate"); // duplicate data point!
    } else if (p.inLeftSubtree(x)) {
        p.left = insert(x, p.left, (p.cutDim + 1) % x.getDim());
    } else { // insert into right
        p.right = insert(x, p.right, (p.cutDim + 1) % x.getDim());
    }
    return p;
}
```
Point kd-Tree

Point insertion

- Insert(50,90):
Deletion is more complicated - Need a s node

How to choose the replacement?

- Can’t just take the inorder successor (inorder doesn’t make geometric sense)
- Depends on the current cutting dimension $i$
- Want the point of the right subtree with the minimum $i$ coordinate $p[i]$

Utility: Select the point with the smaller $i$th coordinate

```java
Point minAlongDim(Point p1, Point p2, int i) { // return smaller point on dim i
    if (p2 == null || p1[i] <= p2[i])
        return p1;
    else
        return p2;
}
```
Point \textit{kd-}tree

Utility for finding replacement nodes

- **Utility**: Find the point that minimizes \textit{i}th coordinate in subtree \textit{p}
  - if (\textit{p.cutDim} == \textit{i}):
    - The subtrees are ordered by the \textit{i}th coordinate
    - Look recursively in \textit{p}'s left subtree, if it exists
    - If not, take \textit{p.point}
  - if (\textit{p.cutDim} != \textit{i}):
    - The subtrees are ordered arbitrarily with respect to \textit{I}
    - Compute the minima from \textit{p}'s left and right subtrees recursively
    - Use findMin to select the overall minimum from left-min, right-min, and \textit{p.point}
Point kd-tree

Utility for finding replacement nodes

- **Utility**: Find the point that minimizes \( i \)th coordinate in subtree \( p \)

```java
Point findMin(KDNode p, int i) { // get min point along dim i
    if (p == null) { // fell out of tree?
        return null;
    }
    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
        Point q = minAlongDim(p.point, findMin(p.left, i), i);
        return minAlongDim(q, findMin(p.right, i), i);
    }
}
```
Point $kd$-tree

Utility for finding replacement nodes

- Example: Find minimum along $x$
  - If cut dim = $x$: Try left child (or $p$ itself)
  - If cut dim = $y$: Try both children
Point \textit{kd-tree}

Point deletion

- **Overview:** Delete \( x \) from subtree \( p \)
  - if (\( p == \text{null} \)):
    - Fell out of the tree - \textit{Error: attempt to delete nonexistent point!}
  - else:
    - If both of \( p \)'s children are null - Simply \textit{unlink} \( p \) (return null)
    - If \( p \)'s right child exists:
      - Invoke \textit{findMin}(\( p\.right \), \( p\.cutDim \)) to compute replacement node
      - Copy its contents to \( p \)
      - Recursively delete the replacement node from \( p\.right \)
    - Else:
      - \textit{Tricky!}
**Point kd-tree**

**Point deletion**

- **Overview**: Delete $x$ from subtree $p$, where $p$ has a left child but no right child:
  - In the 1D case, we just unlinked $p$
    - But this has the effect of promoting $p$’s child up a level
    - The cutting dimensions no longer cycle from parent to child. (Do we care? Suppose we do)
  - How about picking the maximum point in $p$’s left subtree?
    - Our tie-breaking rule assumed that points in the left subtree have coordinates strictly smaller than the splitter
    - This will cause problems if there are duplicate coordinates in $p$’s left subtree
  - **Final answer** *(very sneaky!)*
    - Compute the minimum from $p$’s left subtree as replacement (But it’s on the wrong side!)
    - Make the left subtree the new right subtree. (Amazingly, this works!)
KDNode delete(Point x, KDNode p) {
    if (p == null) {                                // fell out of tree?
        throw Exception("point does not exist");
    } else if (p.point.equals(x)) {                 // found it
        if (p.right != null) {                      // take replacement from right
            p.point = findMin(p.right, p.cutDim);
            p.right = delete(p.point, p.right);
        } else if (p.left != null) {                // take replacement from left
            p.point = findMin(p.left, p.cutDim);
            p.right = delete(p.point, p.left);
            p.left = null;                          // left subtree is now empty
        } else {                                    // deleted point in leaf
            p = null;                               // remove this leaf
        }
    } else if (p.inLeftSubtree(x)) {               // delete from left subtree
        p.left = delete(x, p.left);
    } else {                                        // delete from right subtree
        p.right = delete(x, p.right);
    }
    return p;
}
Point kd-tree

Point deletion - Example
Point kd-tree

Analysis

- Analogous to unbalanced binary search trees
  - Storage space linear in \( n \), the number of points
  - All dictionary operations (insert, delete, find) take time proportional to tree’s height
  - **Theorem**: If \( n \) points are inserted in random order, the expected height of the kd-tree is \( O(\log n) \)

- I’d conjecture that deletion suffers from the same systematic bias, which would lead to heights of \( \sqrt{n} \) after long sequences of random insertions and deletions, but I know of no results from the literature
Summary

- Geometric Search
  - Point representation
- Point Quadtree
- Point kd-Trees
  - Node representation (point and cutting dimension)
  - Insertion
  - Deletion
    - FindMin utility
    - Sneaky trick to compute replacement nodes
  - Analysis: $O(\log n)$ time assuming random insertions