The Notorious PM Quadtree

The Instrument of Your Torture

Evan Machusak

Original: June 19, 2003
Updated: November 5, 2003

The Polygonal Map

- PM Quadtree = “Polygonal Map”
- Similar to PR (Point Region) quadtree, but stores lines instead of points
- Ends up storing both (a line is defined by two points)
- Invented by Hanan Samet some time ago.

PM Data Structure

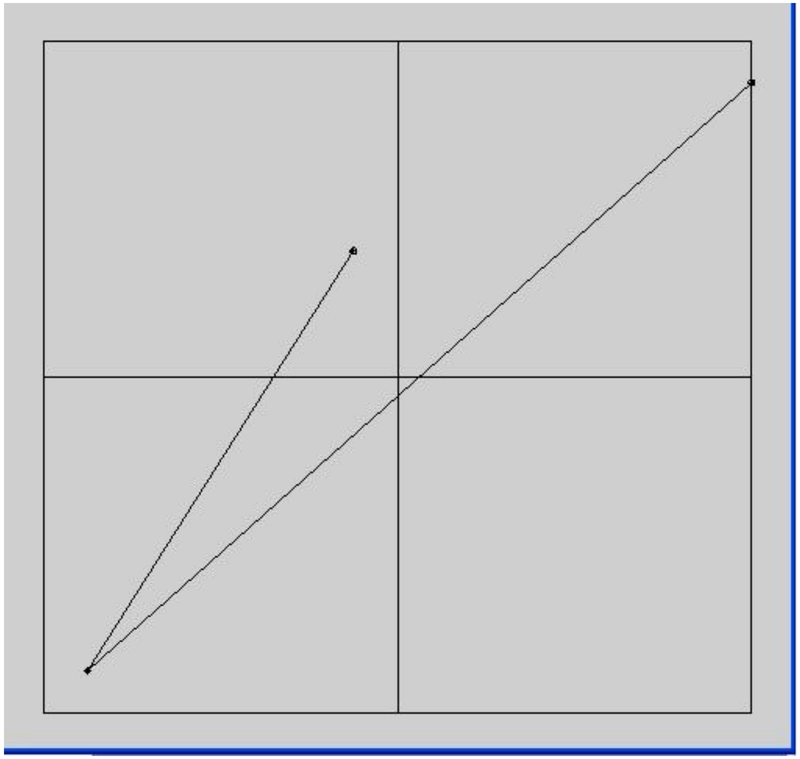
- 4-ary search trie
- Ordered (child 1 = NW, child 2 = NE, child 3 = SW, child 4 = SE)
- Key-space partition (internal nodes are guides)
- Like PR, consists of black, grey, and white nodes

PM Nodes

- Black nodes: contains a data dictionary (a set) of geometric objects contained in this region of space (more later)
- Grey nodes: define the partitions in space – for PM Quadtrees, divides space into 4 equal areas (quadrants)
- White nodes: represent empty regions in space

Tree Structure

- Partitions are same as PR Quadtree, except for handling when data lands on boundaries (later)



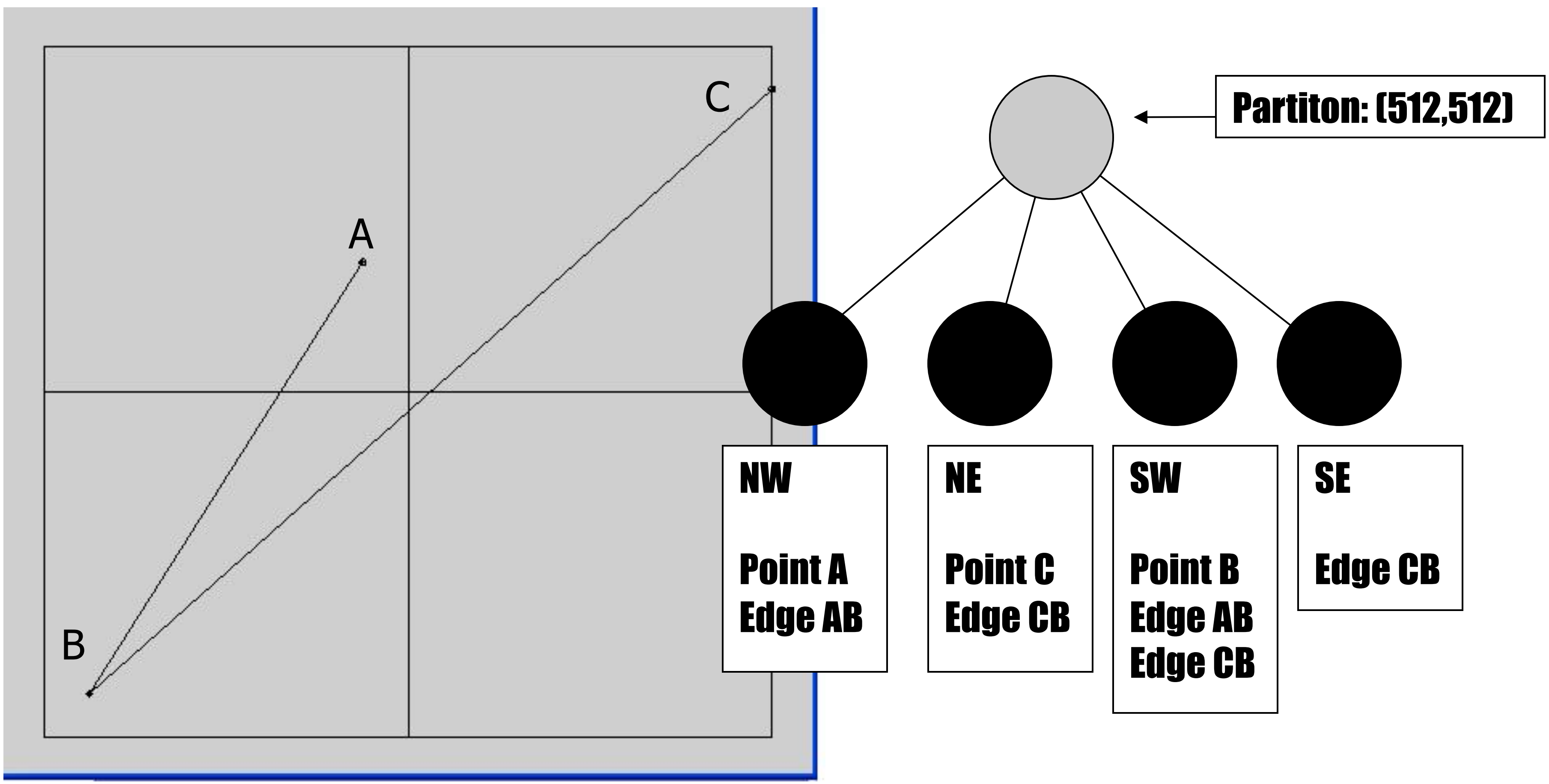
PM quadtrees are not PR's

- PM quadtrees can be partitioned by the location of line segment's endpoints, but that's not all
- PM designed to map polygons, so it stores line segments
- These line segments overlap more than one region in the tree

Q-edges

- A big difference between PR quads and PM quads is that segments must be stored in every region they overlap
- Terminology: all line data in PM trees is called a “q-edge” (q = quadrant)
  - Dr. Hugue says: “a q-edge is merely a term for a portion of a segment, restricted or clipped by a given partition, irrespective of the inclusion of its vertices”
- PM trees can be partitioned based on q-edges

Example



C
A
B
Partiton: (512,512)
NW
Point A
Edge AB
NE
Point C
Edge CB
SW
Point B
Edge AB
Edge CB
SE
Edge CB

PM# trees

- What's a PM3, a PM2, a PM1?
- The "order" of the quadtree generally refers to how strict the rules are about what a black node can contain
- The result: stricter rules mean more partitions
- PM1 is the strictest

PM3 Quadtrees

- Black nodes can contain:
  - *At most 1* dot (in other words, can only contain ONE endpoint)
  - As many q-edges as it wants
  - Q-edges are not allowed to intersect (demo linked from webpage is wrong!)
- Partitioning is the same as the PR quadtree – only challenge is adding q-edges to appropriate nodes, boundary cases different

PM2 Quadtrees

- Black nodes can contain:
  - *At most 1* vertex
  - One or more non-intersecting q-edges, all of which must share a common endpoint

PM1 Quadtrees

- Black nodes can contain:
  - EITHER:
    - At most 1 vertex, and 1 or more q-edges, as long as the q-edges <u>do not intersect</u> *and* they all share the ndoe's single vertex as one of their endpoints OR:
    - At most 1 q-edge (this is actually *exactly* 1 because a black node with 0 q-edges is a white node)

PM1 seems hard to implement

- The rules imposed by the PM1 on black nodes can make insertion and removal difficult. Here's why:
  - You have to detect line intersections, which may only occur very deeply in the tree (possibly after you've already partitioned and added the q-edge to other, shallower subtrees)
  - Partitioning becomes tricky: may have to partition many, many times for a single insert

Merging is fun

- When a line is removed, you may need to merge, because:
- The PM1 must be minimal at all times: if a grey node can be collapsed, it must be
- Merging transforms a grey node into a black node by collecting the dictionaries of all of the grey node's children into a single dictionary

But don't worry…

- Despite how challenging the PM1 may be, students like you implement this beast every semester
- Not terrible if you've already dealt with PR Quadtrees.
- Let's talk about some details of the PM1

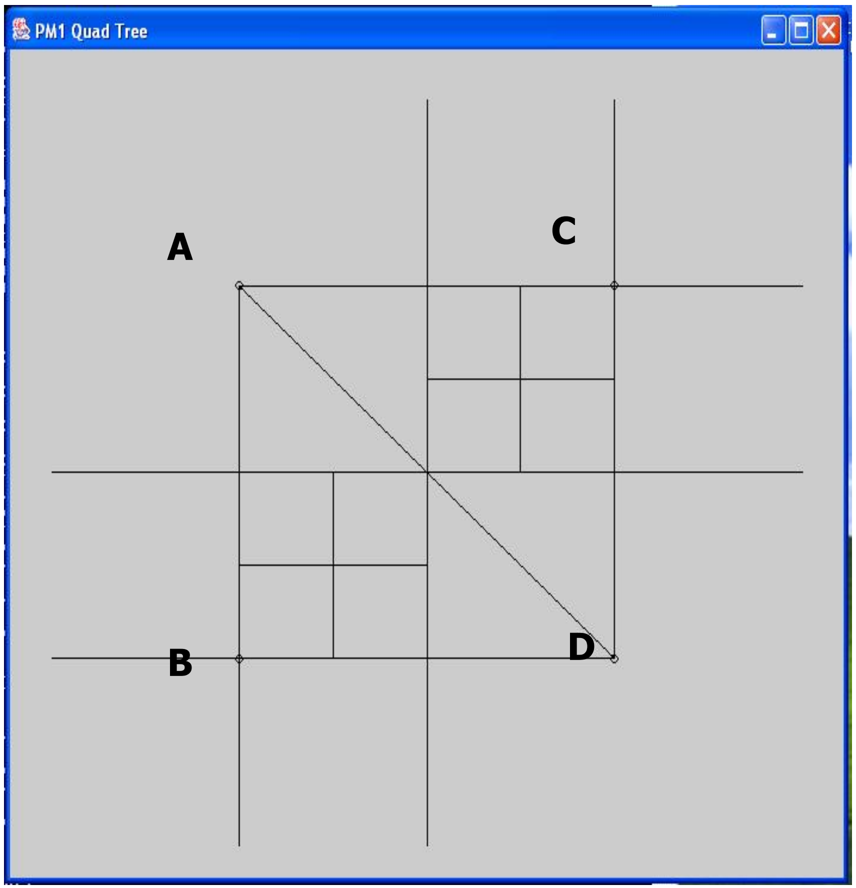
Logical questions

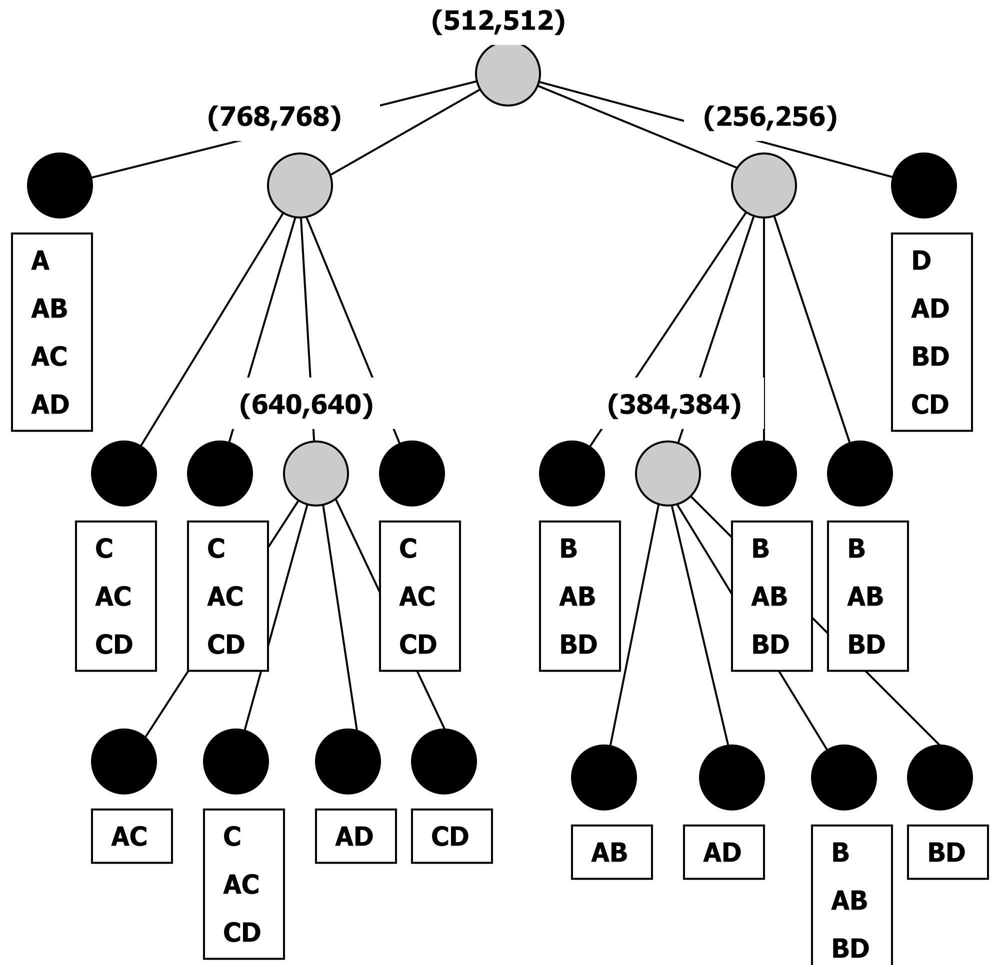
- What happens when a line’s endpoint falls on a partition boundary?
- What happens when a line is defined to fall exactly on a partition?
- What happens when an endpoint falls on a partition’s center (ex. 512,512)
- What happens when a line intersects a partition’s center?

Answer:

- Insert the point or q-edge into all of the regions it touches!
- Let’s look at a nasty example…

All Your Base Are Belong To Us





A (256,768) B (256,256) C (768,768) D (768,256)

Other issues to consider

- When do you stop partitioning?
  - Partitions can continue to occur within space constraints
  - For our purposes it’s usually when the region’s area would be less than 1x1
  - Even if endpoints are integers, lines can cross partitions at floating-point coordinates
  - Geometric computations for the PM1 are floating-point based

More issues

- How close is close enough?
- Two lines could be really, really close together (perhaps they differ by only 10^{-10}) – does that qualify as intersection or not?
- A maximum intersection distance can be defined as a way to stop partitioning (somewhat trickier than measuring partition width)

What’s it good for?

- Obvious answer: nothing!
  - But you’re wrong
- According to Samet:
  - Determination of the identity of the region in which a point lies (duh)
  - Determination of the boundaries of all regions lying within a given distance of a point (nearest segment to point)
  - Overlaying two maps

Asymptotic complexity

- Building the tree:
  - According to Samet, the build time for adding 1 edge at a time to a PM1 is:
    - $(E * 3 + L * 2^{DMAX+1}) * (DMAX + A)$
      - E = number of edges to add to the map
      - L = size of the perimeter of the largest area
      - DMAX = maximum depth of the tree (for a 1024 grid with 1x1 smallest regions, this is 10 – in general this is $\log_2(B) - \log_2(C)$
        - B is the size of one side of the grid)
        - C is the square root of the smallest allowable region's area (usually ≤ 1)
      - A = cost of inserting edge into data dictionary

Asymptotic analysis, con’t

- Very robust analysis, but if you’re thinking in big O notation, n = E, number of edges
- The build is linear according to Samet – O(n)
  - Better than a SkipList or a TreeMap, which is O(n log n)
- Remember: the logarithmic component of a PM is not based on n, it’s based on an external (constant) factor. This is because it’s a search trie with fixed maximum depth.

But…

- Look at those constants: huge!
- For a 1024 grid, let’s crunch the numbers (A is negligibly small)
  - DMAX = 10, so:
    - $(E * 3 + L * 2^{DMAX+1}) * (DMAX + A)$
  - $(30E + 4*1024*2048*(10+0) = 30E + 83886080$
  - Yes, that’s 83,886,080!
  - 30 is larger than log n when n = 1,000,000

How about a search? Remove?

- Locating a line in a PM1
  - This is a fast operation – need to search down the tree to one endpoint, if endpoint exists, search for line in this black node's data dictionary.
  - Result: O(log E)
    - Reliant on a θ(log n) structure used for data dictionaries
    - It's safe to use a linear structure like a List, but then it's O(e)
- Remove: Samet himself won't touch the analysis on this one. "I leave this to you as an exercise."
  - My guess: Ω(log E), O(E)
  - This might actually be θ(E)

Finally… how to implement

- In previous projects, you’ve dealt with SortedMaps as data dictionaries
- PM Quadtrees are neither SortedMaps nor useful data dictionaries
  - Too specific to be a general map: keys are always lines
- So what are they?
- They are built to perform certain operations quickly – remember Samet’s uses for a quadtree?

About the nodes

- Unlike SkipList, you have at least two types of nodes in your PM
- Grey nodes will turn into black nodes and black nodes will turn into grey nodes (both can turn to white)
- How to deal with that?
    - Give up now and drink yourself into oblivion
        - The CMSC department highly recommends against this

Grey to Black, vice versa

Could make one single “node” type which has a flag,

- 0 = white
- 1 = grey
- 2 = black

- Each node has a data dictionary and four children pointers

But that’s bad

- Using one node class and a flag is the naïve and inefficient way
  - May seem easier to code, but not really
- You waste lots of space. A grey node doesn’t need a data dictionary and a black node doesn’t need children pointers.
- Also run the risk of calling inappropriate methods (i.e., calling methods designed for black nodes when the node is grey)
  - Have to get around this by always checking flags at start of function – this makes bulky code. Dynamic method invocation is better.

Another option

- Make an abstract node supertype, and make a black and a grey both extend this class
  - abstract class
  - interface
- Nice because no space is wasted, and better OO design. Easier to debug, maybe easier to understand
- “Harder” to code
- Be prepared for lots of casts unless you’re a pro
  - Don’t try a method call on a Node assuming it’s black or grey and catch an exception to tell you otherwise. Throwing exceptions is *very* expensive (especially for flow control)

Data types you need

- Need a way to represent q-edges
- Need a structure for a black node’s data dictionary
- Optional structures:
  - Need a good way to represent regions (can be done many ways – more later)
  - Need a way to represent partitions
  - These two are potentially equivalent

Q-edge

- Make sure whatever you use to represent a q-edge …
  - extends java.awt.geom.Line2D.Float or java.awt.geom.Line2D.Double
- You don’t need to create your own class *per se*, unless edges can have extra information (such as names)

Java’s awesome geometry

- Java’s great for PM quadtrees because all of the geometric primitives are implemented for you to use or extend
- More importantly, all of the computational geometry algorithms necessary are already done for you
- Especially nice: the intersects() functions

Data dictionary

- For your black nodes' data dictionaries, you need a set of some kind:
  - Doesn't need to be sorted since dictionaries are often small (fewer than 10 elements)
  - Possibly only use one dictionary – in all PM trees, there can be only one point per region, so only need a list of edges and keep the point separate
  - Edges can be Comparable (or use a Comparator) if you want to use fast, sorted structures (like TreeSet), but this is usually wasted overhead

Representing partitions as grey nodes

- If you are brave, you don’t need to store anything at partitions – you can figure their center point out on the fly based on the level of the tree and the known min/max partition sizes (Krznarich does this)
- Samet precomputes his partitions; only a substantial cost reduction if maximal region’s area is not a power of 2, since bit shifting is practically free
  - Same as storing center of partitions at grey nodes
- Other options:
  - Store a point (center of the partition)
  - Store 4 java.awt.geom.Rectangle2D.Float’s

Learn to use exceptions

- Exceptions can help you tremendously on this project
- If you aren't familiar with them, learn them – they are simple to use
- Great for signaling when a partition is attempted on a region that is already the minimum size (at least, that's how I wrote my insert function …)

Stuff to keep in mind

- In this project, you're dealing with lots of binary partitioning (i.e., dividing by two)
  - For ints, x/2 == x >> 1
    - Yes, java has bitshift operators!
  - For floats, x/2 == x*0.5
    - Floating point multiplication is better than division
  - javac probably makes this optimization automatically for you, but it's good practice anyway

Java is annoying

- This is a tree, and tree algorithms often look like this:

```
public void BSTAdd(Node root, Object data) {
    if (root == null) root = new Node(data);
    else if (root.data.compareTo(data) < 0)
            BSTAdd(root.left,data);
    else BSTAdd(root.right,data);
}
```

- But this doesn't work in Java!

Wrapping your reference

```
public void add(Object data) {
    Node[] n = new Node[] { this.root };
    BSTAdd(n,data)
    this.root = n[0];
}
public void BSTAdd(Node[] r, Object data) {
    if (r[0] == null) r[0] = new Node(data);
    else if (r[0].data.compareTo(data) < 0)
          BSTAdd(r[0].left,data);
    else BSTAdd(r[0].right,data);
}
```

Alternatively…

- Rather than use an array, you could also make a wrapper class with a single public data member
- Note: Java people will hate you for this –
  - Adds unnecessary overhead (a few extra bytes and a few nanoseconds of access time)
  - Also, this is not coding "the Java way" … it's a hack
  - Purists: use return statements exclusively, return an array of objects (Object[]) in place of a series of out parameters

The right way

- The right design makes the PM Quadtree trivial to implement
  - Yes, *trivial*
- Your PM Quadtree should contain some inner classes for its node types (grey, black, and white)
  - Yes, white – you'll see why briefly
- The PM Quadtree class itself is really just an interface for accessing the root node of the tree
- The nodes themselves perform all of the work

A sample Node type

- Consider this Node interface

```
public interface Node {
    public Node add(Geometry g, Point center, Number
    width, Number height) throws PMException;
    public Node remove(Geometry g, Point center,
    Number width, Number height);
    public boolean valid();
}
```

- Notice the return types and the parameters of `add` and `remove`

Return types explained

- The reason for the return type is to circumvent the need to try something like:
  - `void add(Node n, …) { n = new Node();}`
  - `Node x = new Node(); add(x,…);`
  - Doesn’t work
- Instead:
  - `Node f(…) { return new Node(); }`
  - `Node x = new Node(); x = f(…);`

The basic idea

- When add() is invoked on a grey node, the call to add() is forwarded to one or more of that grey node's children
  - For example, a line may need to be added to all four of the grey node's children if it intersects the center point
  - If the child is grey, the process is repeated
  - If the child is black, the item you're adding gets put into the child's dictionary
    - What happens if the node is partitioned?
    - The black child becomes a grey child!
  - If the child is white, the white node becomes a black child

But the references may change…

- Say, for instance, your grey node has a Node[] to store its 4 children:
  - `Node[] children = new Node[4];`
- If you determine that the geometry you are trying to add to this grey node intersects child 0, you would write:
  - `children[0] = children[0].add(…)`
- You may be reassigning the first child (in region 1), e.g. if that child was a black node that was just partitioned
- In that case, the black node's add() method would return the resulting new grey node

Add for the black node

- Adding geometry to the black node is simple:
  - Insert the geometry into the dictionary, and then invoke `valid()`
  - `valid()` examines the dictionary
  - If `valid()` returns true, return `this`
  - Otherwise, return a new grey node that is the result of partitioning this black node

Add for the grey node

- For each child:
  - If the geometry you're trying to add intersects the child, assign the child to be the result of invoking add on that child, for example:
    - **`children[0] = children[0].add(…)`**
  - Otherwise do nothing
- Always return `this`

Add for the white node

- You may find it useful to implement a white node class
- When a grey is first created, all children are initially white
- When add() is invoked on a white node, it just returns a new black node containing the geometry to add
- If you implement a white node, make it a singleton class…

How to make a singleton class in Java

- Make the constructor protected or private, and provide a public static instance:
- Access the singleton: Singleton.instance

```
public class Singleton {
  public static final Singleton
     instance = new Singleton();
  private Singleton() {}
}
```

Remove is similiar

- For black nodes, if the last item is removed, return the singleton white node, otherwise return `this`
- For grey nodes, after calling remove() on the appropriate children, *always* check if its children can be merged or if it's still necessary
  - e.g., has more than one black or grey node
  - If the grey is still necessary, return `this`
  - Otherwise, return the new merged black node or return its only remaining black node
- For white nodes, if you ever call remove() on them your code is buggy, so throw an exception

What to do if something goes wrong…

- Consider this code:

```
public static int f() { throw new RuntimeException(); }
int x = 0;
int y = 5;
try { x = f(); y = 7; } catch(Exception e) {}
```

- What happens?
  - x stays 0, y stays 5
  - Can use similar idea when partitioning a black node
  - If there's an intersection with existing geometry or if the partition goes too deep, throw an exception

However…

- This requires some backtracking
- If child 1, 2, and 3 succeed but child 4 throws an exception, you'll need to undo the add action
- Sufficient to simply call remove() on the offending geometry if add() fails
- Obviously, this requires remove() also be implemented
- remove() is as easy as add() under this design

Alternatively…

- Some people like to do a prescan of the tree to test for problems before they insert to avoid having to call remove
  - You can tell a priori if a partition will be too deep based on the proximity of the geometry you're adding to pre-existing geometry
  - You can easily detect intersections
- However, this is costly: a prescan method might be less expensive than remove, but you're calling prescan every single time you add
- By cleaning up only when an error has occurred, you are only doing extra work when the input is bad

A detail to keep in mind

- If you choose to throw exceptions when things go wrong with the intention of catching those exceptions and then executing some code, this practice is called *exceptions for flow-control*
- This is bad practice because exceptions are expensive
- However -- Throwing an exception isn't expensive; creating one is, because a stack trace is created and during that creation the JVM needs to halt.
- Since you don't care about the stack trace when using exceptions for flow control, you can make a static member variable of type **`Throwable`** in your PM Quadtree and always throw that instead of **`throw new Exception()`**

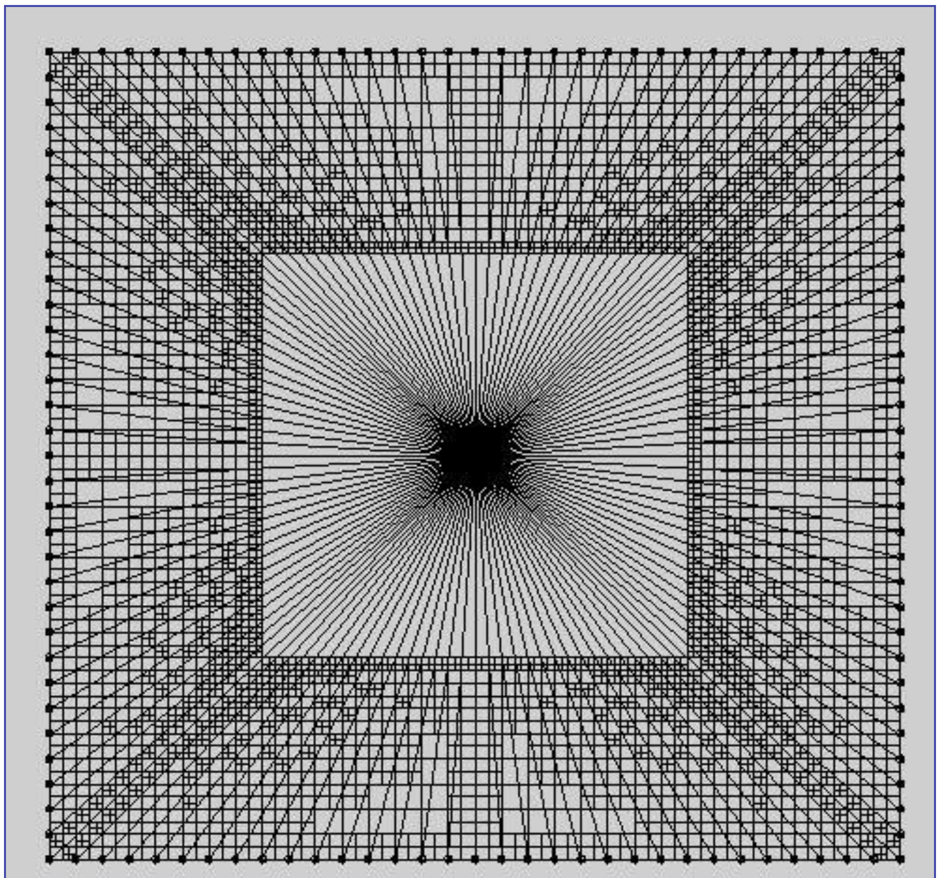
And always…

- If you feel uncomfortable with the PM1 quadtree, there are:
  - Pages and pages about them in Samet's book
  - Pascal pseudo-code by Samet
    - If you translate his Pascal, line for line, into Java:

      "Public flogging is the only answer."

      -- *Bobby Bhattacharjee*
  - Office hours
    - Keeps your friendly TAs entertained while sitting through obligatory office hours

Final thoughts



Thank you, Hanan Samet!

"I leave this to you as an exercise."

Take care of yourself, and each other.