

Of course, the PR quadtree will be implemented as a formal Java generic.

However, it may be somewhat less generic than the general BST discussed earlier.

During insertion and search, it is necessary to determine whether one point lies NW, NE, SE or SW of another point. Clearly this cannot be accomplished by using the usual `Comparable` interface design to compare points.

Two possible approaches:

- have the data type provide accessors for the  $x$ - and  $y$ -coordinates
- have the type provide a comparator that returns NW, NE, SE or SW

Either is feasible. It is possible to argue either is better, depending upon the value placed upon various design goals. It is also possible to deal with the issue in other ways.

In any case, the PR quadtree implementation will impose fairly strict requirements on any data type that is to be stored in it.

A generic PR quadtree interface might look like this::

```
public class prQuadTree< T extends Compare2D<? super T> > {  
    . . .  
}
```

The notion is that the interface `Compare2D`:

- is somewhat similar to the standard interface `Comparable`
- supplies a way to determine the directional relationship between two locations

A simple node implementation might look like this:

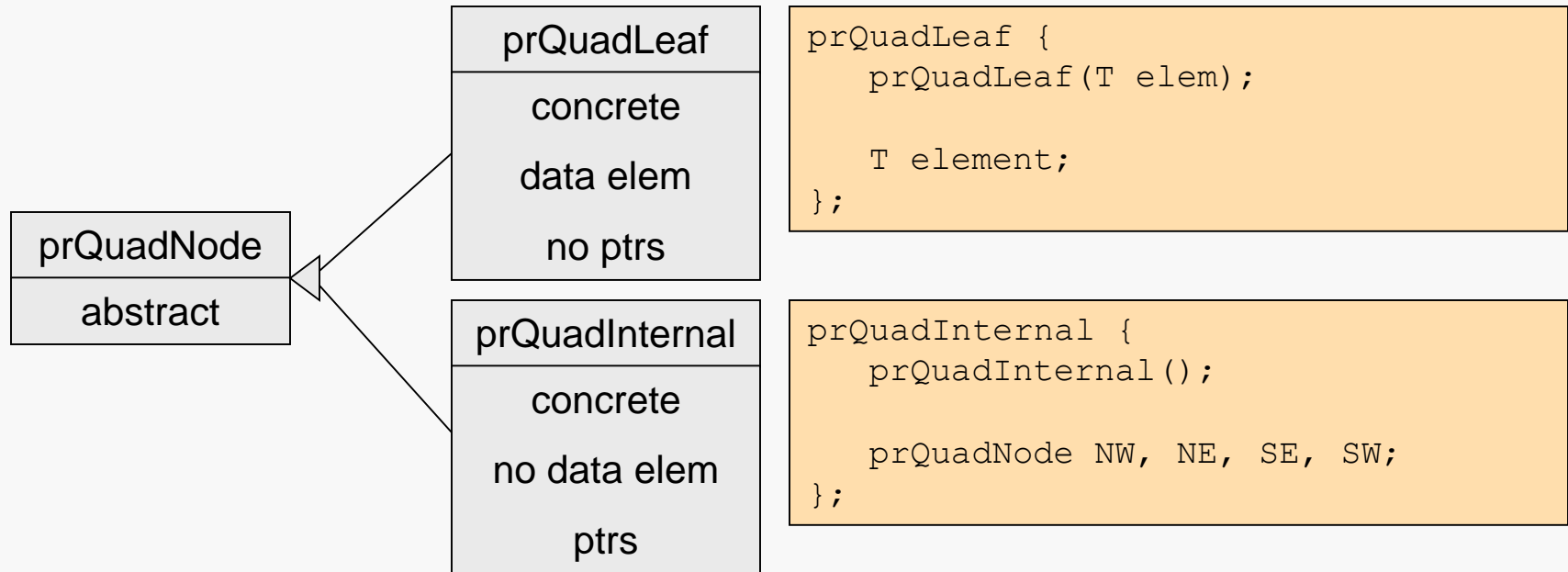
```
public class prQuadNode {  
  
    public prQuadNode() {...}  
    public prQuadNode( T data ) {...}  
  
    public T element;  
    public prQuadNode NE, NW, SW, SE;  
}
```

However, this will waste memory equivalent to one data element in each internal node, and equivalent to four pointers in each leaf node.

This suggests using a hierarchy of node types, with an abstract base type.

But, this raises some thorny implementation issues, since a child of an internal node could be either another internal node or a leaf node.

Using a single node type wastes space in every node. The unused members can be eliminated by defining a hierarchy of nodes:



The definitions of the relevant classes are straightforward.

But an internal node may point to either internal or leaf nodes, and there is no overlap in the public interfaces of the two derived types...

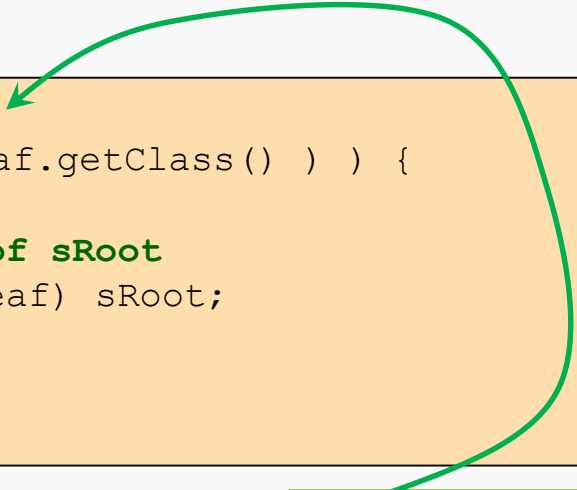
The basic problem is quite simple: given a base-type pointer how can we tell whether its target is a leaf node or an internal node?

One answer is that we may use the `getClass` method to determine the type of the target at runtime:

```
// see if we're at a leaf node
if ( sRoot.getClass().equals( Leaf.getClass() ) ) {

    // access the element member of sRoot
    prQuadLeaf current = (prQuadLeaf) sRoot;

    // use current.element...
    . . .
```



Although this is somewhat clumsy, it does allow the use of a node hierarchy to reduce the space cost of the tree.

Leaf is a `prQuadLeaf` member of the tree; `getClass()` cannot be static.

Here's a possible PR quadtree interface:

```
public class prQuadtree< T extends Compare2D<? super T> > {  
  
    private class prQuadNode {...}  
    private class prQuadLeaf extends prQuadNode {...}  
    private class prQuadInternal extends prQuadNode {...}  
  
    private prQuadNode root;  
    private int xMin, xMax, yMin, yMax;  
  
    public prQuadtree(int xMin, int xMax, int yMin, int yMax) {...}  
    public boolean insert(T elem) {...}  
    private prQuadNode insertHelper(prQuadNode sRoot, T elem,  
                                     double xLo, double xHi,  
                                     double yLo, double yHi) {...}  
  
    public boolean remove(T elem) {...}  
    public T find(T elem) {...}  
    public void clear();  
    ...  
}
```

Some comments:

- the tree must be created to organize data elements that lie within a particular, bounded region in order for the partitioning logic
- the question of how to manage different types for internal and leaf nodes raises some fascinating design and coding issues...
- the question of how to manage comparisons of the user data objects raises some fascinating design and coding issues...
- how to display the tree also raises some fascinating issues...

Implement complex code, such as a PR quadtree, feature by feature.

Write code one small chunk at a time and then test it.

The *code chunks* should implement a logical sub-part of a method/operation.

Carefully consider all of the cases of the operations, e.g. insertion, search, and deletion.

Logically order the cases that must handled.

Operation Implementation:

- Implement the first case and test it.
- Implement the second case, test it and then test any interaction with the first case.
- ...and so forth...
- After all cases are implemented, test the interaction of the operation with other operations.



Cases:

1. Item to be inserted does not exist (**null** was passed in).
2. Item to be inserted not within World.
3. Tree is empty.
4. Current node during descent is a leaf node.
5. Current node during descent is an internal node

Case #1: nothing to do.

Case #2: nothing to do.

Case #3: instantiate new leaf node with element to add make **root** point to it.

Case #4: current node is a leaf node (and hence, nonempty) and bucket is not full

- #4a     Coordinates element being inserted match those of an element in the bucket  
Either an update (different key field) or duplicate (same key field)  
So do what?
- #4b     Coordinates of element being inserted do not match any current element in bucket  
Add element to the bucket

Case #5: Current node is a leaf node (and hence, nonempty) and bucket is full

- Create a new internal node representing quadrants for the leaf
- For each element in the bucket:
  - Determine which quadrant of the new internal node that element fits in
  - Create a new leaf node for that quadrant (if none exists)
  - Add the current element to the bucket in that leaf node  
(cannot cause any splitting)
- Hang the new internal node where the old leaf node was
- For the new element being inserted to the quadtree:
  - Determine which quadrant of the new internal node that element fits in
  - Insert the new element into that leaf  
(may cause another split, recursive helper function is handy for this)



Case #6: Current node is an internal node.

- Determine in which sub-quadrant (SQ) element to insert falls
- Descend recursively, update SQ reference upon ascent



This is potentially tricky because the node you descend into may be split.  
If so, that node is replaced by a new (Internal) node.  
The parent must wind up pointing to the correct node in the end.

Draw pictures!

Use pictures to "trace" the execution of the code you are planning.

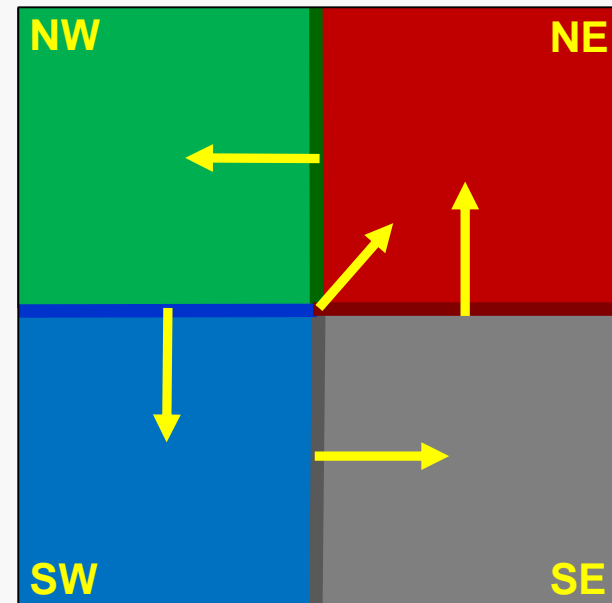
To which quadrant does a point belong when it is on the boundary separating two adjacent quadrants?

In which quadrant should the intersection point of four quadrants be placed?

Conventions:

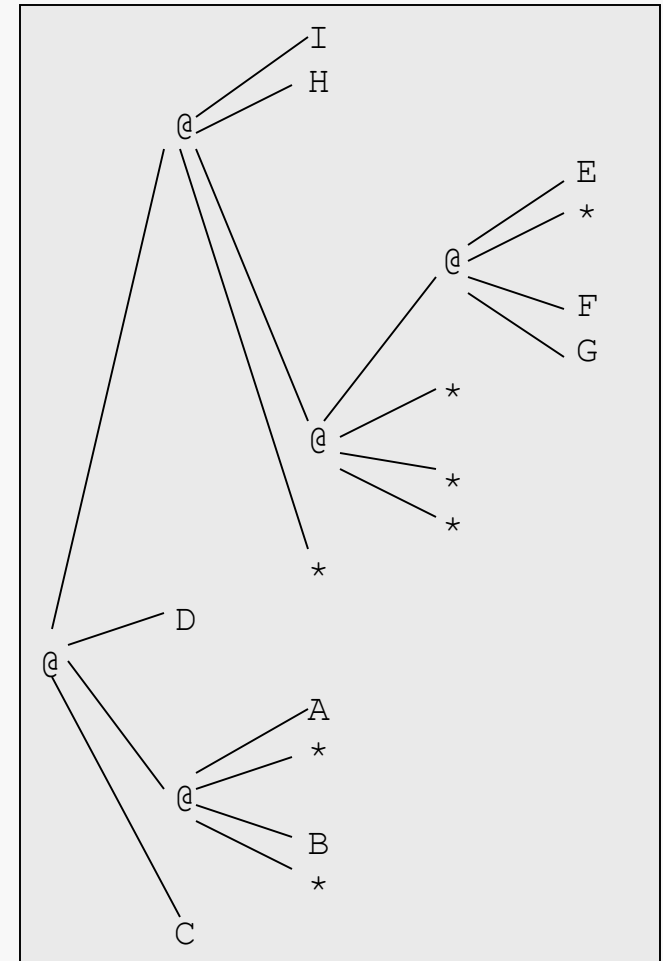
*Samet*: “...the lower and left boundaries for each block are closed, while the upper and right boundaries for each block are open.”

**3114**: points on a boundary are considered to belong to its counter-clockwise quadrant and the origin point belongs to the NE quadrant.



In order to make the structure clear:

- There are alternative ways to do this...



```
public void printTreeHelper(prQuadNode sRoot, String Padding) {

    // Check for empty leaf
    if ( sRoot == null ) {
        System.out.println(Padding + "*\n");
        return;
    }
    // Check for and process SW and SE subtrees
    if ( sRoot.getClass().equals(Internal.getClass()) ) {
        prQuadInternal p = (prQuadInternal) sRoot;
        printTreeHelper(p.SW, Padding + "  ");
        printTreeHelper(p.SE, Padding + "  ");
    }
    // Display indentation padding for current node
    System.out.println(Padding);

    // Determine if at leaf or internal and display accordingly
    if ( sRoot.getClass().equals(Leaf.getClass()) ) {
        prQuadLeaf p = (prQuadLeaf) sRoot;
        System.out.println( Padding + p.element );
    }
    else
        System.out.println( Padding + "@\n" );
    . . .
}
```

```
. . .  
// Check for and process NE and NW subtrees  
if ( sRoot.getClass().equals(Internal.getClass()) ) {  
    prQuadInternal p = (prQuadInternal) sRoot;  
    printTreeHelper(p.NE, Padding + "  ");  
    printTreeHelper(p.NW, Padding + "  ");  
}  
}
```