PR Quadtree Implementation

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.

PR Quadtree Node Implementation

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;
}
```

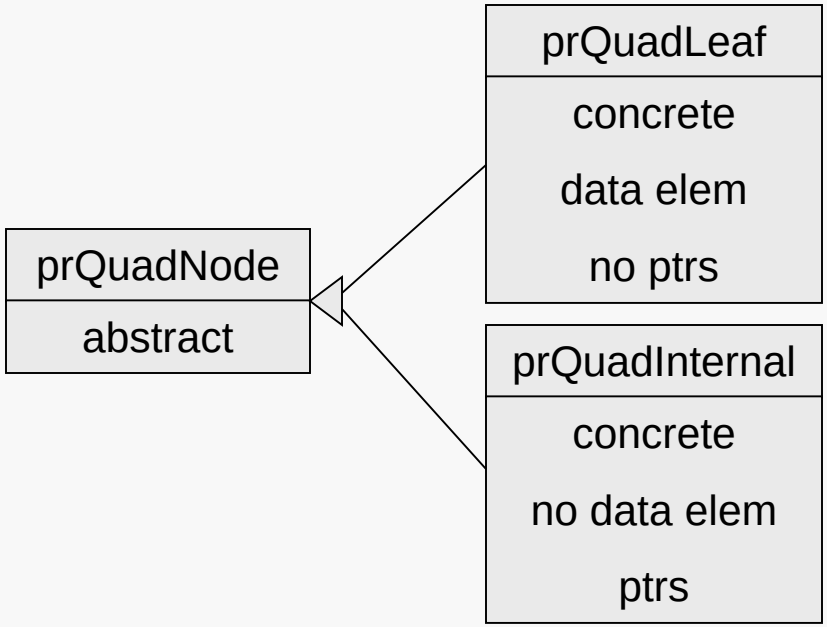
However, this will waste memory equivalent to one data element in each internal node, and equivalent to four pointers in each leaf node.

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.

Improving Storage Efficiency

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




```
prQuadLeaf {
   prQuadLeaf(T elem);

   T element;
};
```

```
prQuadInternal {
   prQuadInternal();

   prQuadNode NW, NE, SE, SW;
};
```

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…

Getting It to Work

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…
   . . .
```

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

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

PR Quadtree Implementation

Here's a possible PR quadtree interface:

```
public class prQuadtree< T extends 2DComparable<? 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();
   ...
}
```

PR Quadtree Implementation

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…

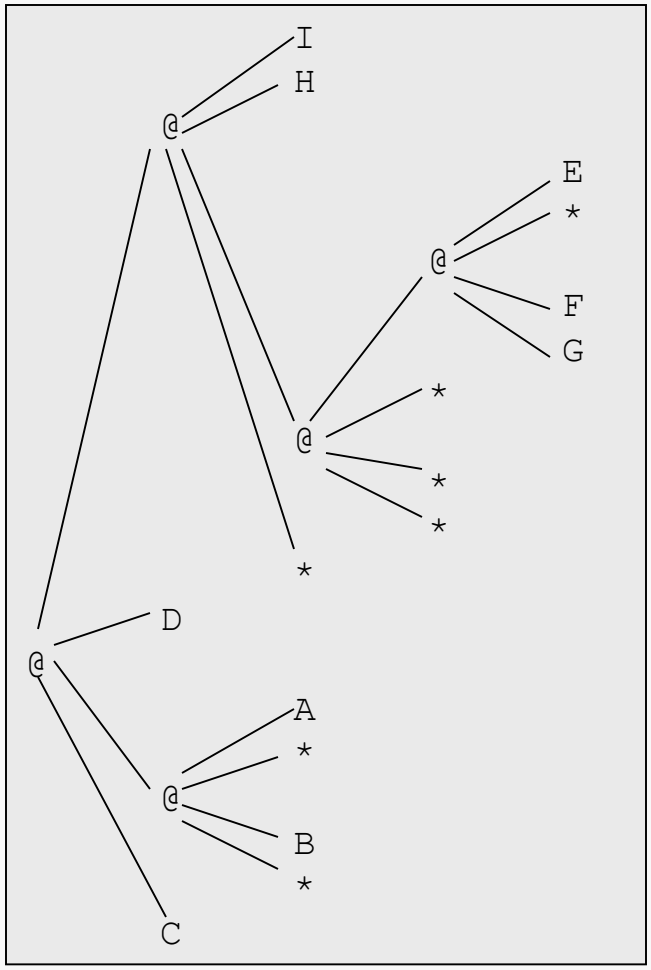
Displaying the PR Quadtree

The display here (aside from the edges) was produced by using a modification of the inorder traversal for binary trees.

In order to make the structure clear:

- internal nodes are represented by `'@'`
- empty children are represented by `'*'`

There are alternative ways to do this…



Example Code

```
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" );
   . . .
```

Example Code

```
    . . .
    // 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 + "   ");
    }
}
```