PR Quadtree Node Implementation

A simple node implementation might look like this:

```cpp
template <typename T> class prQuadNode {
    public:
        prQuadNode();
        prQuadNode(const T& Elem);
        T Element;
        prQuadNode<T> *NE, *SE, *SW, *NW;
    }
```

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…

PR Quadtree Implementation

Of course, the PR quadtree will be implemented as a C++ template.

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 relational operators 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.
Here's a possible PR quadtree interface:

```cpp
template <typename T> class prQuadTree {
public:
    prQuadTree(int xMinimum, int xMaximum,
                int yMinimum, int yMaximum);
    bool Insert(const T& Elem);
    bool Delete(const T& Elem);
    T* const Find(const T& Elem);
    const T* const Find(const T& Elem) const;
    void Display(std::ostream& Out) const;
    void Clear();
    ~prQuadTree();

private:
    prQuadNode<T>* Root;
    int xMin, xMax, yMin, yMax;
    // . . .
};
```

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…
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:

- **prQuadNode**: abstract
- **prQuadLeaf**: concrete, data elem, no ptrs
- **prQuadInternal**: concrete, no data elem, ptrs

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... so the usual approach to polymorphism via runtime binding of virtual functions will not work.

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 `dynamic_cast` operator to determine the type of the target at runtime:

```cpp
if ( prQuadInternal<T>* x = dynamic_cast<prQuadInternal<T>*>(p) ) {
    // we have a handle x on an internal node
} else if( prQuadLeaf<int>* y = dynamic_cast<prQuadLeaf<int>*>(p) ) {
    // we have a handle y on a leaf node
}
```

dynamic_cast takes a type `T` and a pointer `p` as parameters (note HOW).

If the target of `p` is an object of type `T`, or is an object that has a unique base class of type `T`, then dynamic_cast returns a pointer of type `T*` to the target of `p`.

Otherwise, dynamic_cast returns NULL.

So, we can determine on the fly whether we’re looking at a leaf or internal node. It’s not particularly pretty, but it’s idiomatic C++.
**dynamic_cast**

Here’s a simpler summary of the syntax and logic:

\[
\text{A* x = dynamic_cast< A* >( p )}
\]

The template parameter to \textit{dynamic_cast} must be a pointer type.

The function parameter to \textit{dynamic_cast} must be a pointer to an object.

---

**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…
```cpp
template <typename T>
void prQuadTree<T>::DisplayHelper(ostream& Out,
        prQuadNode<T>* sRoot,
        unsigned int Level) const {

    if ( sRoot == NULL ) {  // at an empty leaf
        Out << setw( 3 * Level ) << ' ' << '*' << endl;
        return;
    }

    // check for and process SW and SE subtrees
    if ( prQuadInternal<T>* p =
        dynamic_cast<prQuadInternal<T>*>(sRoot) ) {
        DisplayHelper(Out, p->SW, Level + 1);
        DisplayHelper(Out, p->SE, Level + 1);
    }

    . . .

    . . .
    // display current node
    if ( Level > 0 ) {
        Out << setw( 3 * Level ) << ' ';
    }

    // determine if leaf or internal and display accordingly
    if ( prQuadLeaf<T>* p =
        dynamic_cast<prQuadLeaf<T>*>(sRoot) )
        Out << p->Element << endl;
    else
        Out << "@" << endl;

    // check for and process NE and NW subtrees
    if ( prQuadInternal<T>* p =
        dynamic_cast<prQuadInternal<T>*>(sRoot) ) {
        DisplayHelper(Out, p->NE, Level + 1);
        DisplayHelper(Out, p->NW, Level + 1);
    }

    . . .
```