15.3. The PR Quadtree

15.3.1. The PR Quadtree

In the **Point-Region quadtree** (hereafter referred to as the **PR quadtree**) each node either has exactly four children or is a leaf. That is, the PR quadtree is a full four-way branching (4-ary) tree in shape. The PR quadtree represents a collection of data points in two dimensions by decomposing the region containing the data points into four equal quadrants, subquadrants, and so on, until no leaf node contains more than a single point. In other words, if a region contains zero or one data points, then it is represented by a PR quadtree consisting of a single leaf node. If the region contains more than a single data point, then the region is split into four equal quadrants. The corresponding PR quadtree then contains an internal node and four subtrees, each subtree representing a single quadrant of the region, which might in turn be split into subquadrants. Each internal node of a PR quadtree represents a single split of the two-dimensional region. The four quadrants of the region (or equivalently, the corresponding subtrees) are designated (in order) NW, NE, SW, and SE. Each quadrant containing more than a single point would in turn be recursively divided into subquadrants until each leaf of the corresponding PR quadtree contains at most one point.

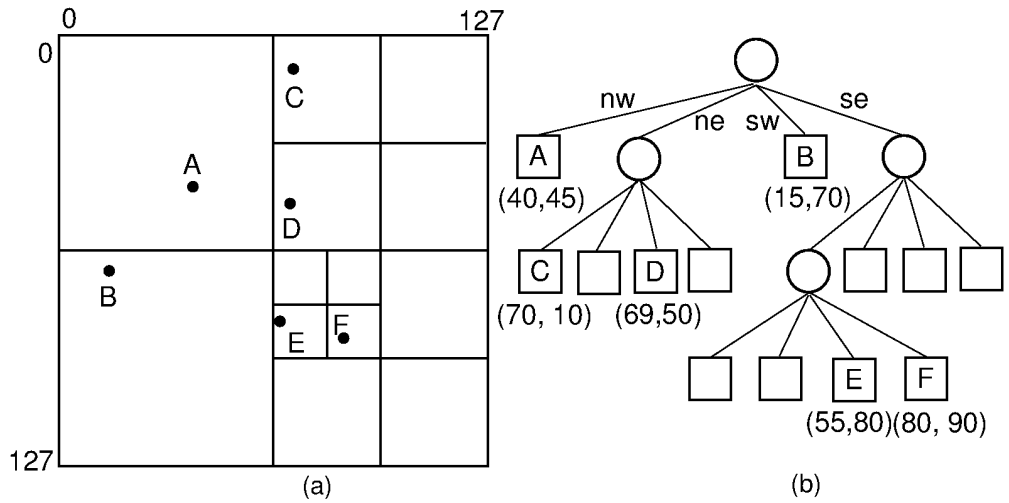


Figure 15.3.1: Example of a PR quadtree. (a) A map of data points. We define the region to be square with origin at the upper-left-hand corner and sides of length 128. (b) The PR quadtree for the points in (a). (a) also shows the block decomposition imposed by the PR quadtree for this region.

For example, consider the region of Figure **15.3.1** (a) and the corresponding PR quadtree in Figure **15.3.1** (b). The decomposition process demands a fixed key range. In this example, the region is assumed to be of size 128×128. Note that the internal nodes of the PR quadtree are used solely to indicate decomposition of the region; internal nodes do not store data records. Because the decomposition lines are predetermined (i.e, key-space decomposition is used), the PR quadtree is a trie.

Search for a record matching point Q in the PR quadtree is straightforward. Beginning at the root, we continuously branch to the quadrant that contains Q until our search reaches a leaf node. If the root is a leaf, then just check to see if the node's data record matches point Q. If the root is an internal node, proceed to the child that contains the search coordinate. For example, the NW quadrant of Figure **15.3.1** contains points whose x and y values each fall in the range 0 to 63. The NE quadrant contains points whose x value falls in the range 64 to 127, and whose y value falls in the range 0 to 63. If the root's child is a leaf node, then that child is checked to see if Q has been found. If the child is another internal node, the search process continues through the tree

until a leaf node is found. If this leaf node stores a record whose position matches Q then the query is successful; otherwise Q is not in the tree.

Here is a visualization of the PR quadtree that should help you to understand how insert a point or removing a point works.

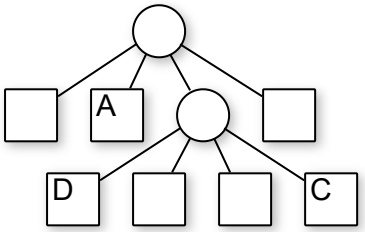
PR Quadtree Visualization

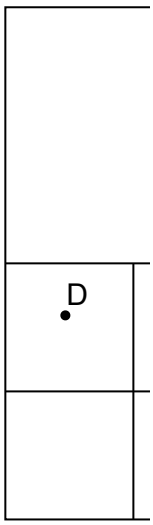
Run Reset

66 / 66

≪ < > ≫

This is the current state of the tree after merging





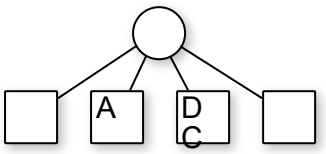
Note that there is no particular reason why the tree should split when there is more than one point in a node. This spliting criteria could be anything that the implementor wants.

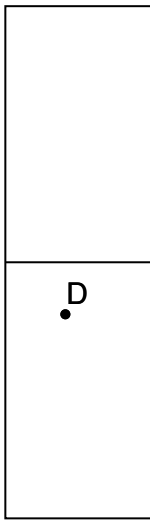
PR Quadtree 2-Point Visualization

Run Reset

52 / 52

This is the current state of the tree after merging





Here is an interactive visualization of the PR quadtree. You can build your own example by adding or removing points. See if you can create a tree with the same shape as the one in the picture at the top of this page.

The interactive visualization below will let you use a different split value if you want. How would the tree look if it had the same points as the figure in the top of the page, but a node was allowed to have two points?

PR Quadtree Interactive

submit 1 reset

insert remove

Submit number of point for splitting and start inserting

Region search is easily performed with the PR quadtree. To locate all points within radius r of query point Q, begin at the root. If the root is an empty leaf node, then no data points are found. If the root is a leaf containing a data record, then the location of the data point is examined to determine if it falls within the circle. If the root is an internal node, then the process is performed recursively, but *only* on those subtrees containing some part of the search circle.

Let us now consider how the structure of the PR quadtree affects the design of its node representation. The PR quadtree is actually a **trie**. Decomposition takes place at the mid-points for internal nodes, regardless of where the data points actually fall. The placement of the data points does determine *whether* a decomposition for a node takes place, but not *where* the decomposition for the node takes place. Internal nodes of the PR quadtree are quite different from leaf nodes, in that internal nodes have children (leaf nodes do not) and leaf nodes have data fields (internal nodes do not). Thus, it is likely to be beneficial to represent internal nodes differently from leaf nodes. Finally, there is the fact that approximately half of the leaf nodes will contain no data field.

Another issue to consider is: How does a routine traversing the PR quadtree get the coordinates for the square represented by the current PR quadtree node? One possibility is to store with each node its spatial description (such as upper-left corner and width). However, this will take a lot of space—perhaps as much as the space needed for the data records, depending on what information is being stored.

Another possibility is to pass in the coordinates when the recursive call is made. For example, consider the search process. Initially, the search visits the root node of the tree, which has origin at (0, 0), and whose width is the full size of the space being covered. When the appropriate child is visited, it is a simple matter for the search routine to determine the origin for the child, and the width of the square is simply half that of the parent. Not only does passing in the size and position information for a node save considerable space, but avoiding storing such information in the nodes enables a good design choice for empty leaf nodes, as discussed next.

How should we represent empty leaf nodes? On average, half of the leaf nodes in a PR quadtree are empty (i.e., do not store a data point). One implementation option is to use a NULL pointer in internal nodes to represent empty nodes. This will solve the problem of excessive space requirements. There is an unfortunate side effect that using a NULL pointer requires the PR quadtree processing methods to understand this convention. In other words, you are breaking encapsulation on the node representation because the tree now must know things about how the nodes are implemented. This is not too horrible for this particular application, because the node class can be considered private to the tree class, in which case the node implementation is completely invisible to the outside world. However, it is undesirable if there is another reasonable alternative.

Fortunately, there is a good alternative. It is called the **Flyweight design pattern**. In the PR quadtree, a flyweight is a single empty leaf node that is reused in all places where an empty leaf node is needed. You simply have *all* of the internal nodes with empty leaf children point to the same node object. This node object is created once at the beginning of the program, and is never removed. The node class recognizes from the pointer value that the flyweight is being accessed, and acts accordingly.

Note that when using the Flyweight design pattern, you *cannot* store coordinates for the node in the node. This is an example of the concept of intrinsic versus extrinsic state. Intrinsic state for an object is state information stored in the object. If you stored the coordinates for a node in the node object, those coordinates would be intrinsic state. Extrinsic state is state information about an object stored elsewhere in the environment, such as in global variables or passed to the method. If your recursive calls that process the tree pass in the coordinates for the current node, then the coordinates will be extrinsic state. A flyweight can have in its intrinsic state *only* information that is accurate for *all* instances of the flyweight. Clearly coordinates do not qualify, because each empty leaf node has its own location. So, if you want to use a flyweight, you must pass in coordinates.

Another design choice is: Who controls the work, the node class or the tree class? For example, on an insert operation, you could have the tree class control the flow down the tree, looking at (querying) the nodes to see their type and reacting accordingly. This is the typical approach used by the BST implementation. An alternate approach is to have the node class do the work. That is, you have an insert method for the nodes. If the node is internal, it passes the city record to the appropriate child (recursively). If the node is a flyweight, it replaces itself with a new leaf node. If the node is a full node, it replaces itself with a subtree. This is an example of the **Composite design pattern**. Use of the composite design would be difficult if NULL pointers are used to represent empty leaf nodes. It turns out that the PR quadtree insert and delete methods are easier to implement when using the composite design.