Implementation Advice

Implement complex code, such as a PR quadtree, piece-meal.

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. search, insertion 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.
- Implement the third case, test it and then test any interaction with the previous cases. etc. etc.
- After all cases are implemented, test the interaction of the operation with other operations.

Search

Search logic in data structures tends to be used in multiple operations, e.g. find, delete, insert). Thus implementing and testing the search logic first will aid in the other operation development.

What are the search cases to be considered?

Tree is empty.

Select branch to search down.

Leaf node comparison.

Item to be found not in tree.

Item to be found not in World.

Item to be found does not exist (null).

Search Cases

Re-order cases for logical implementation:

1. Item to be found does not exist (null parameter).
2. Tree is empty.
3. Item to be found is not in the World.
4. Leaf node comparison: item found or not in tree.
5. Inner node: select branch to search down:
    a. Compute quadrant division midpoint.
    b. Determine quadrant in which element to be found is located.

```
Code one case, then test.
```

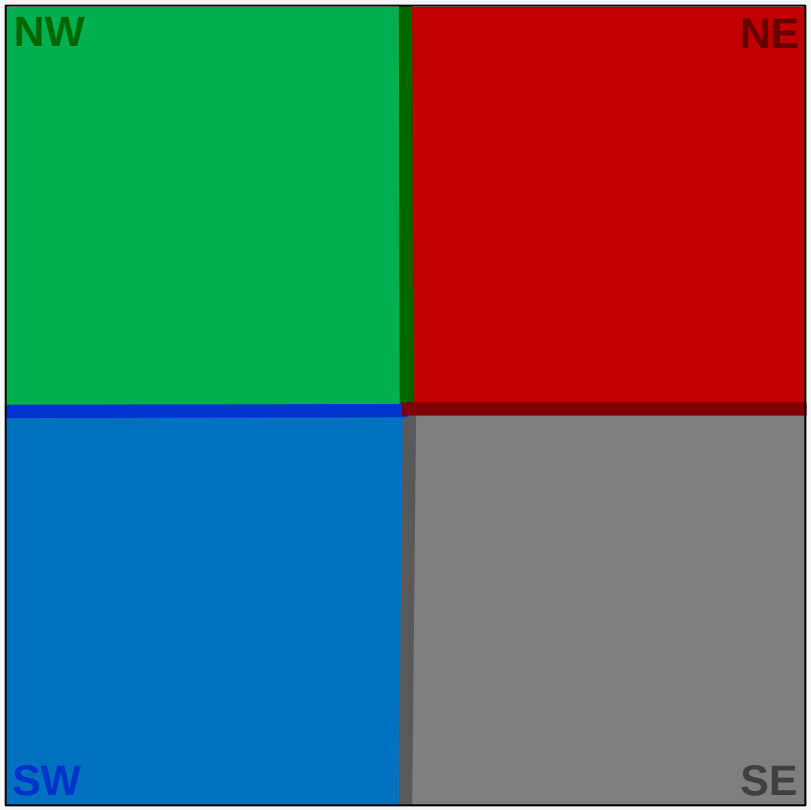
Quadrant Boundaries

In which quadrant does a point belong when it is on the boundary line of 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.



Insertion

Cases:

1. Item to be found does not exist (null).
2. Item to be found not within World.
3. Tree is empty.
4. Current node is a Leaf node.
5. Current node is an Inner node

```
This discussion
assumes one
element per leaf.
```

Case #1: nothing to do.

Case #2: nothing to do.

Case #3: instantiate new leaf node with element to add & attach.

Insertion Cases

Case #4: Current node is a Leaf node.

a. Coordinates of leaf node element and element to add are equal.
Operation is either an update or non-allowed duplicate.

b. Coordinates of leaf node element and element to add are **NOT** equal.

   i. Determine midpoint of quadrant.

   ii. Instantiate new inner node, creating sub-quadrant references.

   iii. Determine in which sub-quadrant existing leaf element falls and assign it to reference the existing leaf node.

   iv. Determine in which sub-quadrant (SQ) element to add falls

      1) SQ is empty (null): instantiate new leaf node with element to add & attach.

      2) SQ is NOT empty: descend recursively …
      (existing leaf elem & elem to add falls in the same SQ)

Case #5: Current node is an Inner node.

a. Determine in which sub-quadrant (SQ) element to add falls.

b. Descend recursively, update SQ reference upon ascent.

Deletion

Cases:

1. Item to be found does not exist (null parameter).
2. Item to be found not within World.
3. Tree is empty.
4. Current node is a Leaf node.
5. Current node is an Inner node

```
This discussion
assumes one
element per leaf.
```

Case #1: nothing to do.

Case #2: nothing to do.

Case #3: return unsuccessful deletion.

Deletion Cases

Case #4: Current node is a Leaf node.

a. Coordinates of leaf node element and element to delete are NOT equal.
Indicate element NOT found in tree, back out.

b. Coordinates of leaf node element and element to delete are equal.
Remove the element (set the reference in the node to the element to null).
Return null for recursive ascent parent reference assignment.

Case #5: Current node is an Inner node.

a. Determine in which sub-quadrant (SQ) element to delete falls.

b. Descend recursively, update SQ reference upon ascent.

c. Tree contraction?

   i. if elem was removed lower in tree & SQ now contains one child:
   Contraction: set SQ reference to child to null & return reference to child.

   ii. if elem was NOT removed lower in tree or SQ now contains > one child:
   No contraction, return inner node reference for ascent update.

Cost

Searching:

- Worst: $O(\lceil \log(\frac{\sqrt{2}s}{d}) \rceil)$

where:

s	length of side of world
d	minimum distance between any two points in tree

Assuming a PR Quadtree of size N, built using random insertion of elements.

Insertion:

- Requires search to locate element or its insertion leaf sub-quadrant plus cost to create new inner node and redistribute existing leaf + added element. For random insertion, collision of existing leaf & new element will occur infrequently. This cost of redistribution on average will be a constant.
- Worst: $O(\lceil \log(\frac{\sqrt{2}s}{d}) \rceil)$

Deletion

- Requires search to locate element plus cost of possible contraction during ascent. For randomly formed tree, contraction will occur infrequently. This cost of contraction on average will be a constant.
- Worst: $O(\lceil \log(\frac{\sqrt{2}s}{d}) \rceil)$

Node Hierarchy Delegation

The prQuadNode subclasses, (i.e., prQuadLeaf & prQuadInternal), each implement search, insert & delete methods.

- Abstract base class, (prQuadNode), will declare abstract search, insert & delete methods for over-riding.
- Subclass over-riding methods will contain appropriate node type logic.
- Example: Insertion
  - prQuadInternal insert: determines sub-quadrant (SQ) element to add falls, if SQ is null allocates new leaf, assigns element & attaches, else assigns SQ node reference to return value of (direct/indirect) recursive invocation of insert upon SQ node.
  - prQuadLeaf insert: checks for duplicate case; instantiates new inner node, assigns existing leaf element to appropriate sub-quadrant, determines sub-quadrant (SQ) for element to add, if SQ is null allocates new leaf, assigns element & attaches,
    else assigns SQ node reference to return value of (direct) recursive invocation of insert upon SQ node.