Line-Sweep Algorithms

```
1  class CrossVerifier(object):
2    def _events_from_layer(self, layer):
3      """Populates the sweep line events from the wire layer."""
4      left_edge = min([wire.x1 for wire in layer.wires.values()])
5      for wire in layer.wires.values():
6        if wire.is_horizontal():
7          self.events.append([left_edge, 0, wire.object_id, 'add',
               wire])
8        else:
9          self.events.append([wire.x1, 1, wire.object_id, 'query',
               wire])
```

```
1  class CrossVerifier(object):
2    def _compute_crossings(self, count_only):
3      """Implements count_crossings and wire_crossings."""
4      if count_only:
5        result = 0
6      else:
7        result = self.result_set
8
9      for event in self.events:
10       event_x, event_type, wire = event[0], event[3], event[4]
11
12       if event_type == 'add':
13         self.index.add(KeyWirePair(wire.y1, wire))
14       elif event_type == 'query':
15         self.trace_sweep_line(event_x)
16         cross_wires = []
17         for kwp in self.index.list(KeyWirePairL(wire.y1),
18                                    KeyWirePairH(wire.y2)):
19           if wire.intersects(kwp.wire):
20             cross_wires.append(kwp.wire)
21         if count_only:
22           result += len(cross_wires)
23         else:
24           for cross_wire in cross_wires:
25             result.add_crossing(wire, cross_wire)
26
27     return result
```

Range Index

```
1  class RangeIndex(object):
2    """Array-based range index implementation."""
3
4    def __init__(self):
5      """Initially empty range index."""
6      self.data = []
7
8    def add(self, key):
9      """Inserts a key in the range index."""
10     if key is None:
11         raise ValueError('Cannot insert nil in the index')
12     self.data.append(key)
13
14   def remove(self, key):
15     """Removes a key from the range index."""
16     self.data.remove(key)
17
18   def list(self, first_key, last_key):
19     """List of values for the keys that fall within [first_key,
           last_key]."""
20     return [key for key in self.data if first_key <= key <= last_key
           ]
21
22   def count(self, first_key, last_key):
23     """Number of keys that fall within [first_key, last_key]."""
24     result = 0
25     for key in self.data:
26       if first_key <= key <= last_key:
27         result += 1
28     return result
```

```
1  class BlitRangeIndex(object):
2    """Sorted array-based range index implementation."""
3
4    def __init__(self):
5      """Initially empty range index."""
6      self.data = []
7
8    def add(self, key):
9      """Inserts a key in the range index."""
10     if key is None:
11         raise ValueError('Cannot insert None in the index')
12     self.data.insert(self._binary_search(key), key)
13
14   def remove(self, key):
15     """Removes a key from the range index."""
16     index = self._binary_search(key)
17     if index < len(self.data) and self.data[index] == key:
18       self.data.pop(index)
19
20   def list(self, low_key, high_key):
21     """List of values for the keys that fall within [low_key,
          high_key]."""
22     low_index = self._binary_search(low_key)
23     high_index = self._binary_search(high_key)
24     return self.data[low_index:high_index]
25
26   def count(self, low_key, high_key):
27     """Number of keys that fall within [low_key, high_key]."""
28     low_index = self._binary_search(low_key)
29     high_index = self._binary_search(high_key)
30     return high_index - low_index
31
32   def _binary_search(self, key):
33     """Binary search for the given key in the sorted array."""
34     low, high = 0, len(self.data) - 1
35     while low <= high:
36       mid = (low + high) // 2
37       mid_key = self.data[mid]
38       if key < mid_key:
39         high = mid - 1
40       elif key > mid_key:
41         low = mid + 1
42       else:
43         return mid
44     return high + 1
```

Comparison Model

```
class KeyWirePair(object):
  """Wraps a wire and the key representing it in the range index.

  Once created, a key-wire pair is immutable."""

  def __init__(self, key, wire):
    """Creates a new key for insertion in the range index."""
    self.key = key
    self.wire = wire
    self.wire_id = wire.object_id
  def __lt__(self, other):
    return (self.key < other.key or
            (self.key == other.key and self.wire_id < other.wire_id)
               )
  def __le__(self, other):
    return (self.key < other.key or
            (self.key == other.key and self.wire_id <= other.wire_id
               ))
  def __gt__(self, other):
    return (self.key > other.key or
            (self.key == other.key and self.wire_id > other.wire_id)
               )
  def __ge__(self, other):
    return (self.key > other.key or
            (self.key == other.key and self.wire_id >= other.wire_id
               ))
  def __eq__(self, other):
    return self.key == other.key and self.wire_id == other.wire_id
  def __ne__(self, other):
    return self.key == other.key and self.wire_id == other.wire_id
```

```
class KeyWirePairL(KeyWirePair):
  def __init__(self, key):
    self.key = key
    self.wire = None
    self.wire_id = -1000000000

class KeyWirePairH(KeyWirePair):
  def __init__(self, key):
    self.key = key
    self.wire = None
    self.wire_id = 1000000000
```

LIST Range Queries in BSTs

```
LIST(tree, l, h)
1  lca = LCA(tree, l, h)
2  result = []
3  NODE-LIST(lca, l, h, result)
4  return result
```

```
LCA(tree, l, h)
1  node = tree.root
2  until node == NIL or (l ≤ node.key and h ≥ node.key)
3       if l < node.key
4            node = node.left
5       else
6            node = node.right
7  return node
```

```
NODE-LIST(node, l, h, result)
1  if node == NIL
2       return
3  if node.key ≥ l
4       NODE-LIST(node.left, l, h, result)
5  if l ≤ node.key and node.key ≤ h
6       ADD-KEY(result, node.key)
7  if node.key ≤ h
8       NODE-LIST(node.right, l, h, result)
```