Code Fragment: Class

```
  template <typename E, typename C>
  class AdaptPriorityQueue {                    // adaptable priority
queue
  protected:
    typedef std::list<E> ElementList;           // list of elements
  public:
    // ...insert Position class definition here
  public:
    int size() const;                           // number of elements
    bool empty() const;                         // is the queue empty?
    const E& min() const;                       // minimum element
    Position insert(const E& e);        // insert element
    void removeMin();                           // remove minimum
    void remove(const Position& p);             // remove at position p
    Position replace(const Position& p, const E& e); // replace at
position p
  private:
    ElementList L;                              // priority queue
contents
    C isLess;                                   // less-than comparator
  };
```

Code Fragment: InsertPosition

```
  template <typename E, typename C>                     // insert
element
  typename AdaptPriorityQueue<E,C>::Position
  AdaptPriorityQueue<E,C>::insert(const E& e) {
    typename ElementList::iterator p = L.begin();
    while (p != L.end() && !isLess(e, *p)) ++p;         // find larger
element
    L.insert(p, e);                                     // insert before
p
    Position pos;  pos.q = --p;
    return pos;                                         // inserted
position
  }
```

Code Fragment: Position

```
    class Position {                              // a position in the
queue
    private:
      typename ElementList::iterator q;           // a position in the
list
    public:
      const E& operator*() { return *q; }         // the element at this
position
      friend class AdaptPriorityQueue;            // grant access
    };
```

Code Fragment: Remove

```
  template <typename E, typename C>                    // remove at
position p
  void AdaptPriorityQueue<E,C>::remove(const Position& p)
    { L.erase(p.q); }

  template <typename E, typename C>                    // replace at
position p
  typename AdaptPriorityQueue<E,C>::Position
  AdaptPriorityQueue<E,C>::replace(const Position& p, const E& e) {
    L.erase(p.q);                                      // remove the
old entry
    return insert(e);                                  // insert
replacement
  }
```

Code Fragment: BottomTop

```
  class BottomTop {                               // a bottom-top
comparator
  public:
    bool operator()(const Point2D& p, const Point2D& q) const
      { return p.getY() < q.getY(); }
  };
```

Code Fragment: LeftRight

```
  class LeftRight {                              // a left-right
comparator
  public:
    bool operator()(const Point2D& p, const Point2D& q) const
      { return p.getX() < q.getX(); }
  };
```

Code Fragment: Main

```
  Point2D p(1.3, 5.7), q(2.5, 0.6);              // two points
  LeftRight leftRight;                           // a left-right
comparator
  BottomTop bottomTop;                           // a bottom-top
comparator
  printSmaller(p, q, leftRight);          // outputs: (1.3, 5.7)
  printSmaller(p, q, bottomTop);          // outputs: (2.5, 0.6)
```

Code Fragment: Point2D

```
  priority_queue<Point2D, vector<Point2D>, LeftRight> p2;
  p2.push( Point2D(8.5, 4.6) );                  // add three points to
p2
  p2.push( Point2D(1.3, 5.7) );
  p2.push( Point2D(2.5, 0.6) );
  cout << p2.top() << endl;  p2.pop();           // output: (8.5, 4.6)
  cout << p2.top() << endl;  p2.pop();           // output: (2.5, 0.6)
  cout << p2.top() << endl;  p2.pop();           // output: (1.3, 5.7)
```

Code Fragment: PrintSmaller

```
  template <typename E, typename C>              // element type and
comparator
  void printSmaller(const E& p, const E& q, const C& isLess) {
    cout << (isLess(p, q) ? p : q) << endl;     // print the smaller of
p and q
```

```
}
```

Code Fragment: Class

```
template <typename E, typename C>
class HeapPriorityQueue {
public:
  int size() const;                              // number of elements
  bool empty() const;                            // is the queue empty?
  void insert(const E& e);                       // insert element
  const E& min();                                // minimum element
  void removeMin();                              // remove minimum
private:
  VectorCompleteTree<E> T;                       // priority queue
contents
  C isLess;                                      // less-than comparator
                                                 // shortcut for tree
position
  typedef typename VectorCompleteTree<E>::Position Position;
};
```

Code Fragment: Insert

```
template <typename E, typename C>                // insert element
void HeapPriorityQueue<E,C>::insert(const E& e) {
  T.addLast(e);                                  // add e to heap
  Position v = T.last();                         // e's position
  while (!T.isRoot(v)) {                         // up-heap bubbling
    Position u = T.parent(v);
    if (!isLess(*v, *u)) break;                  // if v in order, we're
done
    T.swap(v, u);                                // ...else swap with
parent
    v = u;
  }
}
```

Code Fragment: RemoveMin

```
  template <typename E, typename C>             // remove minimum
  void HeapPriorityQueue<E,C>::removeMin() {
    if (size() == 1)                            // only one node?
      T.removeLast();                           // ...remove it
    else {
      Position u = T.root();                    // root position
      T.swap(u, T.last());                      // swap last with root
      T.removeLast();                           // ...and remove last
      while (T.hasLeft(u)) {                    // down-heap bubbling
        Position v = T.left(u);
        if (T.hasRight(u) && isLess(*(T.right(u)), *v))
          v = T.right(u);                       // v is u's smaller
child
        if (isLess(*v, *u)) {                   // is u out of order?
          T.swap(u, v);                         // ...then swap
          u = v;
        }
        else break;                             // else we're done
      }
    }
  }
```

Code Fragment: Simple

```
  template <typename E, typename C>             // number of elements
  int HeapPriorityQueue<E,C>::size() const
    { return T.size(); }

  template <typename E, typename C>             // is the queue empty?
  bool HeapPriorityQueue<E,C>::empty() const
    { return size() == 0; }

  template <typename E, typename C>             // minimum element
  const E& HeapPriorityQueue<E,C>::min()
    { return *(T.root()); }                     // return reference to
root element
```

Code Fragment: Class

```
  template <typename E, typename C>
  class ListPriorityQueue {
```

```
  public:
    int size() const;                          // number of elements
    bool empty() const;                        // is the queue empty?
    void insert(const E& e);                   // insert element
    const E& min() const;                      // minimum element
    void removeMin();                          // remove minimum
  private:
    std::list<E> L;                            // priority queue
contents
    C isLess;                                  // less-than comparator
  };
```

Code Fragment: Insert

```
  template <typename E, typename C>                    // insert
element
  void ListPriorityQueue<E,C>::insert(const E& e) {
    typename std::list<E>::iterator p;
    p = L.begin();
    while (p != L.end() && !isLess(e, *p)) ++p;        // find larger
element
    L.insert(p, e);                                    // insert e
before p
  }
```

Code Fragment: RemoveMin

```
  template <typename E, typename C>                    // minimum
element
  const E& ListPriorityQueue<E,C>::min() const
    { return L.front(); }                              // minimum is at
the front

  template <typename E, typename C>                    // remove
minimum
  void ListPriorityQueue<E,C>::removeMin()
    { L.pop_front(); }
```

Code Fragment: Simple

```
template <typename E, typename C>            // number of elements
int ListPriorityQueue<E,C>::size() const
  { return L.size(); }

template <typename E, typename C>            // is the queue empty?
bool ListPriorityQueue<E,C>::empty() const
  { return L.empty(); }
```

Code Fragment: Class

```
template <typename E>
class VectorCompleteTree {
  //... insert private member data and protected utilities here
public:
  VectorCompleteTree() : V(1) {}             // constructor
  int size() const                           { return V.size() - 1; }
  Position left(const Position& p)           { return pos(2*idx(p)); }
  Position right(const Position& p)          { return pos(2*idx(p) + 1); }
  Position parent(const Position& p)         { return pos(idx(p)/2); }
  bool hasLeft(const Position& p) const      { return 2*idx(p) <= size(); }
  bool hasRight(const Position& p) const     { return 2*idx(p) + 1 <= size(); }
  bool isRoot(const Position& p) const       { return idx(p) == 1; }
  Position root()                            { return pos(1); }
  Position last()                            { return pos(size()); }
  void addLast(const E& e)                   { V.push_back(e); }
  void removeLast()                          { V.pop_back(); }
  void swap(const Position& p, const Position& q)
                                             { E e = *q; *q = *p; *p = e; }
};
```

Code Fragment: Utilities

```
private:                                     // member data
  std::vector<E> V;                          // tree contents
```

```
  public:                                  // publicly accessible types
    typedef typename std::vector<E>::iterator Position; // a position
in the tree
  protected:                                          // protected utility
functions
    Position pos(int i)                               // map an index to a
position
      { return V.begin() + i; }
    int idx(const Position& p) const                  // map a position to an
index
      { return p - V.begin(); }
```