```
class Vector:
  """Represent a vector in a multidimensional space."""

  def __init__(self, d):
    """Create d-dimensional vector of zeros."""
    self._coords = [0] * d

  def __len__(self):
    """Return the dimension of the vector."""
    return len(self._coords)

  def __getitem__(self, j):
    """Return jth coordinate of vector."""
    return self._coords[j]

  def __setitem__(self, j, val):
    """Set jth coordinate of vector to given value."""
    self._coords[j] = val

  def __add__(self, other):
    """Return sum of two vectors."""
    if len(self) != len(other):              # relies on __len__ method
      raise ValueError('dimensions must agree')
    result = Vector(len(self))               # start with vector of zeros
    for j in range(len(self)):
      result[j] = self[j] + other[j]
    return result

  def __eq__(self, other):
    """Return True if vector has same coordinates as other."""
    return self._coords == other._coords

  def __ne__(self, other):
    """Return True if vector differs from other."""
    return not self == other                 # rely on existing __eq__ definition

  def __str__(self):
    """Produce string representation of vector."""
    return '<' + str(self._coords)[1:-1] + '>'    # adapt list representation
```

Code Fragment 2.4: Definition of a simple Vector class.

```
class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------- nested Position class -------------------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def element(self):
      """Return the element stored at this Position."""
      raise NotImplementedError('must be implemented by subclass')

    def __eq__(self, other):
      """Return True if other Position represents the same location."""
      raise NotImplementedError('must be implemented by subclass')

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)                    # opposite of __eq__

  # ---------- abstract methods that concrete subclass must support ----------
  def root(self):
    """Return Position representing the tree's root (or None if empty)."""
    raise NotImplementedError('must be implemented by subclass')

  def parent(self, p):
    """Return Position representing p's parent (or None if p is root)."""
    raise NotImplementedError('must be implemented by subclass')

  def num_children(self, p):
    """Return the number of children that Position p has."""
    raise NotImplementedError('must be implemented by subclass')

  def children(self, p):
    """Generate an iteration of Positions representing p's children."""
    raise NotImplementedError('must be implemented by subclass')

  def __len__(self):
    """Return the total number of elements in the tree."""
    raise NotImplementedError('must be implemented by subclass')
```

Code Fragment 8.1: A portion of our Tree abstract base class (continued in Code Fragment 8.2).

```
class HeapPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with a binary heap."""
  #------------------------------ nonpublic behaviors ------------------------------
  def _parent(self, j):
    return (j−1) // 2

  def _left(self, j):
    return 2*j + 1

  def _right(self, j):
    return 2*j + 2

  def _has_left(self, j):
    return self._left(j) < len(self._data)      # index beyond end of list?

  def _has_right(self, j):
    return self._right(j) < len(self._data)     # index beyond end of list?

  def _swap(self, i, j):
    """Swap the elements at indices i and j of array."""
    self._data[i], self._data[j] = self._data[j], self._data[i]

  def _upheap(self, j):
    parent = self._parent(j)
    if j > 0 and self._data[j] < self._data[parent]:
      self._swap(j, parent)
      self._upheap(parent)                      # recur at position of parent

  def _downheap(self, j):
    if self._has_left(j):
      left = self._left(j)
      small_child = left                        # although right may be smaller
      if self._has_right(j):
        right = self._right(j)
        if self._data[right] < self._data[left]:
          small_child = right
      if self._data[small_child] < self._data[j]:
        self._swap(j, small_child)
        self._downheap(small_child)             # recur at position of small child
```

Code Fragment 9.4: An implementation of a priority queue using an array-based heap (continued in Code Fragment 9.5). The extends the PriorityQueueBase class from Code Fragment 9.1.

```
#------------------------- positional-based utility methods -------------------------
# we consider a nonexistent child to be trivially black
def _set_red(self, p): p._node._red = True
def _set_black(self, p): p._node._red = False
def _set_color(self, p, make_red): p._node._red = make_red
def _is_red(self, p): return p is not None and p._node._red
def _is_red_leaf(self, p): return self._is_red(p) and self.is_leaf(p)

def _get_red_child(self, p):
  """Return a red child of p (or None if no such child)."""
  for child in (self.left(p), self.right(p)):
    if self._is_red(child):
      return child
  return None

#------------------------- support for insertions -------------------------
def _rebalance_insert(self, p):
  self._resolve_red(p)                          # new node is always red

def _resolve_red(self, p):
  if self.is_root(p):
    self._set_black(p)                          # make root black
  else:
    parent = self.parent(p)
    if self._is_red(parent):                    # double red problem
      uncle = self.sibling(parent)
      if not self._is_red(uncle):               # Case 1: misshapen 4-node
        middle = self._restructure(p)           # do trinode restructuring
        self._set_black(middle)                 # and then fix colors
        self._set_red(self.left(middle))
        self._set_red(self.right(middle))
      else:                                     # Case 2: overfull 5-node
        grand = self.parent(parent)
        self._set_red(grand)                    # grandparent becomes red
        self._set_black(self.left(grand))       # its children become black
        self._set_black(self.right(grand))
        self._resolve_red(grand)                # recur at red grandparent
```

Code Fragment 11.16: Continuation of the RedBlackTreeMap class. (Continued from Code Fragment 11.15, and concluded in Code Fragment 11.17.)

```
#------------------------- nested Vertex class -------------------------
class Vertex:
  """Lightweight vertex structure for a graph."""
  __slots__ = '_element'

  def __init__(self, x):
    """Do not call constructor directly. Use Graph's insert_vertex(x)."""
    self._element = x

  def element(self):
    """Return element associated with this vertex."""
    return self._element

  def __hash__(self):              # will allow vertex to be a map/set key
    return hash(id(self))

#------------------------- nested Edge class -------------------------
class Edge:
  """Lightweight edge structure for a graph."""
  __slots__ = '_origin', '_destination', '_element'

  def __init__(self, u, v, x):
    """Do not call constructor directly. Use Graph's insert_edge(u,v,x)."""
    self._origin = u
    self._destination = v
    self._element = x

  def endpoints(self):
    """Return (u,v) tuple for vertices u and v."""
    return (self._origin, self._destination)

  def opposite(self, v):
    """Return the vertex that is opposite v on this edge."""
    return self._destination if v is self._origin else self._origin

  def element(self):
    """Return element associated with this edge."""
    return self._element

  def __hash__(self):              # will allow edge to be a map/set key
    return hash( (self._origin, self._destination) )
```

Code Fragment 14.1: Vertex and Edge classes (to be nested within Graph class).

```
#------------------------------ mutators ------------------------------
# override inherited version to return Position, rather than Node
def _insert_between(self, e, predecessor, successor):
  """Add element between existing nodes and return new Position."""
  node = super()._insert_between(e, predecessor, successor)
  return self._make_position(node)

def add_first(self, e):
  """Insert element e at the front of the list and return new Position."""
  return self._insert_between(e, self._header, self._header._next)

def add_last(self, e):
  """Insert element e at the back of the list and return new Position."""
  return self._insert_between(e, self._trailer._prev, self._trailer)

def add_before(self, p, e):
  """Insert element e into list before Position p and return new Position."""
  original = self._validate(p)
  return self._insert_between(e, original._prev, original)

def add_after(self, p, e):
  """Insert element e into list after Position p and return new Position."""
  original = self._validate(p)
  return self._insert_between(e, original, original._next)

def delete(self, p):
  """Remove and return the element at Position p."""
  original = self._validate(p)
  return self._delete_node(original)      # inherited method returns element

def replace(self, p, e):
  """Replace the element at Position p with e.

  Return the element formerly at Position p.
  """
  original = self._validate(p)
  old_value = original._element           # temporarily store old element
  original._element = e                   # replace with new element
  return old_value                        # return the old element value
```

Code Fragment 7.16: A PositionalList class based on a doubly linked list. (Continued from Code Fragments 7.14 and 7.15.)

```
#-------------------------- binary tree constructor --------------------------
def __init__(self):
  """Create an initially empty binary tree."""
  self._root = None
  self._size = 0

#-------------------------- public accessors --------------------------
def __len__(self):
  """Return the total number of elements in the tree."""
  return self._size

def root(self):
  """Return the root Position of the tree (or None if tree is empty)."""
  return self._make_position(self._root)

def parent(self, p):
  """Return the Position of p's parent (or None if p is root)."""
  node = self._validate(p)
  return self._make_position(node._parent)

def left(self, p):
  """Return the Position of p's left child (or None if no left child)."""
  node = self._validate(p)
  return self._make_position(node._left)

def right(self, p):
  """Return the Position of p's right child (or None if no right child)."""
  node = self._validate(p)
  return self._make_position(node._right)

def num_children(self, p):
  """Return the number of children of Position p."""
  node = self._validate(p)
  count = 0
  if node._left is not None:          # left child exists
    count += 1
  if node._right is not None:         # right child exists
    count += 1
  return count
```

Code Fragment 8.9: Public accessors for our LinkedBinaryTree class. The class begins in Code Fragment 8.8 and continues in Code Fragments 8.10 and 8.11.

```
class HashMapBase(MapBase):
  """Abstract base class for map using hash-table with MAD compression."""

  def __init__(self, cap=11, p=109345121):
    """Create an empty hash-table map."""
    self._table = cap * [ None ]
    self._n = 0                                 # number of entries in the map
    self._prime = p                             # prime for MAD compression
    self._scale = 1 + randrange(p−1)            # scale from 1 to p-1 for MAD
    self._shift = randrange(p)                  # shift from 0 to p-1 for MAD

  def _hash_function(self, k):
    return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)

  def __len__(self):
    return self._n

  def __getitem__(self, k):
    j = self._hash_function(k)
    return self._bucket_getitem(j, k)           # may raise KeyError

  def __setitem__(self, k, v):
    j = self._hash_function(k)
    self._bucket_setitem(j, k, v)               # subroutine maintains self._n
    if self._n > len(self._table) // 2:         # keep load factor <= 0.5
      self._resize(2 * len(self._table) − 1)    # number 2^x - 1 is often prime

  def __delitem__(self, k):
    j = self._hash_function(k)
    self._bucket_delitem(j, k)                  # may raise KeyError
    self._n −= 1

  def _resize(self, c):                   # resize bucket array to capacity c
    old = list(self.items( ))             # use iteration to record existing items
    self._table = c * [None]              # then reset table to desired capacity
    self._n = 0                           # n recomputed during subsequent adds
    for (k,v) in old:
      self[k] = v                         # reinsert old key-value pair
```

Code Fragment 10.4: A base class for our hash table implementations, extending our MapBase class from Code Fragment 10.2.

```
#------------------------ positional-based utility methods ------------------------
def _recompute_height(self, p):
  p._node._height = 1 + max(p._node.left_height( ), p._node.right_height( ))

def _isbalanced(self, p):
  return abs(p._node.left_height( ) − p._node.right_height( )) <= 1

def _tall_child(self, p, favorleft=False): # parameter controls tiebreaker
  if p._node.left_height( ) + (1 if favorleft else 0) > p._node.right_height( ):
    return self.left(p)
  else:
    return self.right(p)

def _tall_grandchild(self, p):
  child = self._tall_child(p)
  # if child is on left, favor left grandchild; else favor right grandchild
  alignment = (child == self.left(p))
  return self._tall_child(child, alignment)

def _rebalance(self, p):
  while p is not None:
    old_height = p._node._height          # trivially 0 if new node
    if not self._isbalanced(p):           # imbalance detected!
      # perform trinode restructuring, setting p to resulting root,
      # and recompute new local heights after the restructuring
      p = self._restructure(self._tall_grandchild(p))
      self._recompute_height(self.left(p))
      self._recompute_height(self.right(p))
    self._recompute_height(p)             # adjust for recent changes
    if p._node._height == old_height:     # has height changed?
      p = None                            # no further changes needed
    else:
      p = self.parent(p)                  # repeat with parent

#-------------------------- override balancing hooks --------------------------
def _rebalance_insert(self, p):
  self._rebalance(p)

def _rebalance_delete(self, p):
  self._rebalance(p)
```

Code Fragment 11.13: AVLTreeMap class (continued from Code Fragment 11.12).

until a yield statement indicates the next value. At that point, the procedure is temporarily interrupted, only to be resumed when another value is requested. When the flow of control naturally reaches the end of our procedure (or a zero-argument return statement), a StopIteration exception is automatically raised. Although this particular example uses a single yield statement in the source code, a generator can rely on multiple yield statements in different constructs, with the generated series determined by the natural flow of control. For example, we can greatly improve the efficiency of our generator for computing factors of a number, n, by only testing values up to the square root of that number, while reporting the factor $n//k$ that is associated with each k (unless $n//k$ equals k). We might implement such a generator as follows:

```
def factors(n):                # generator that computes factors
  k = 1
  while k * k < n:             # while k < sqrt(n)
    if n % k == 0:
      yield k
      yield n // k
    k += 1
  if k * k == n:               # special case if n is perfect square
    yield k
```

We should note that this generator differs from our first version in that the factors are not generated in strictly increasing order. For example, factors(100) generates the series $1, 100, 2, 50, 4, 25, 5, 20, 10$.

In closing, we wish to emphasize the benefits of lazy evaluation when using a generator rather than a traditional function. The results are only computed if requested, and the entire series need not reside in memory at one time. In fact, a generator can effectively produce an infinite series of values. As an example, the Fibonacci numbers form a classic mathematical sequence, starting with value 0, then value 1, and then each subsequent value being the sum of the two preceding values. Hence, the Fibonacci series begins as: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$. The following generator produces this infinite series.

```
def fibonacci():
  a = 0
  b = 1
  while True:                # keep going...
    yield a                  # report value, a, during this pass
    future = a + b
    a = b                    # this will be next value reported
    b = future               # and subsequently this
```