UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA
E MATEMATICA APPLICATA

Corso:
DESIGN AND ANALYSIS OF ALGORITHMS

APPUNTI

Carmine Terracciano

Mat. IE22700109

ANNO ACCADEMICO 2025/2026

Sommario

Capitolo 1

Array Stack

Listing 1.1: Implementazione Python dell'ADT Stack utilizzando una lista per la memorizzazione degli elementi.

```python
class Empty(Exception):
    pass

class ArrayStack:
    """Implementazione di ADT Stack che utilizza un oggetto list di Python
       per la memorizzazione."""

    def __init__(self):
        """Crea uno stack vuoto."""
        self._data = []                    # instanza di list non pubblica

    def __len__(self):
        """Restituisce il numero di elementi nello stack."""
        return len(self._data)

    def is_empty(self):
        """Restituisce True se lo stack è vuoto."""
        return len(self._data) == 0

    def push(self, e):
        """Aggiunge l'elemento e al top dello stack."""
        self._data.append(e)     # il nuovo elemento è aggiunto in coda alla list
```

```python
22
23  def top(self):
24      """Restituisce (ma non rimuove) l'elemento al top dello stack.
25       Raise Empty exception se lo stack è vuoto."""
26      if self.is_empty():
27          raise Empty('lo stack è vuoto')
28          # print("lo stack è vuoto")
29      return self._data[-1]          # legge l'ultimo elemento della list
30
31  def pop(self):
32      """Rimuove e restituisce l'elemento al top dello stack.
33       Raise Empty exception se lo stack è vuoto."""
34      if self.is_empty():
35          raise Empty('lo stack è vuoto')
36          # print("lo stack è vuoto")
37      return self._data.pop()       # rimuove l'ultimo elemento della list
```

Capitolo 2

Array Queue

Listing 2.1: Classe astratta che implementa l'ADT Queue.

```python
class Queue:
  """Classe astratta che implementa l'ADT Queue."""

  def __len__(self):
    """Restituisce il numero di elementi nella coda."""
    raise NotImplementedError('deve essere implementato dalla sottoclasse.')

  def is_empty(self):
    """Restituisce True se la coda è vuota."""
    raise NotImplementedError('deve essere implementato dalla sottoclasse.')

  def first(self):
    """Restituisce (ma non rimuove) l'elemento al front della coda.
        Raise Empty exception se la coda è vuota."""
    raise NotImplementedError('deve essere implementato dalla sottoclasse.')

  def dequeue(self):
    """Rimuove e restituisce l'elemento al front della coda.
        Raise Empty exception se la coda è vuota."""
    raise NotImplementedError('deve essere implementato dalla sottoclasse.')

  def enqueue(self, e):
    """Aggiunge un elemento al back della coda."""
    raise NotImplementedError('deve essere implementato dalla sottoclasse.')
```

Listing 2.2: Implementazione Python dell'ADT Queue utilizzando una lista per la memorizzazione degli elementi.

```python
from .queue import Queue

class Empty(Exception):
  pass

class ArrayQueue(Queue):
  """Implementazione di ADT Queue basata sul tipo list di Python usato dome
      array circolare."""
  DEFAULT_CAPACITY = 10          # dimensione di default di nuove code

  def __init__(self):
    """Crea una coda vuota."""
    self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
    self._size = 0
    self._front = 0

  def __len__(self):
    """Restituisce il numero di elementi nella coda."""
    return self._size

  def is_empty(self):
    """Restituisce True se la coda è vuota."""
    return self._size == 0

  def first(self):
    """Restituisce (ma non rimuove) l'elemento al front della coda.
        Raise Empty exception se la coda è vuota.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    return self._data[self._front]

  def dequeue(self):
    """Rimuove e restituisce l'elemento al front della coda.
        Raise Empty exception se la coda è vuota.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    answer = self._data[self._front]
    self._data[self._front] = None        # favorisce garbage collection
    self._front = (self._front + 1) % len(self._data)
    self._size -= 1
    return answer
```

```python
43
44   def enqueue(self, e):
45     """Aggiunge un elemento al back della coda."""
46     if self._size == len(self._data):
47       self._resize(2 * len(self._data))      # raddoppia la dimensione
           dell'array se pieno
48     avail = (self._front + self._size) % len(self._data)
49     self._data[avail] = e
50     self._size += 1
51
52   def _resize(self, cap):        # we assume cap >= len(self)
53     """Ridimensiona l'array portandolo a lunghezza cap."""
54     old = self._data             # conserva la vecchia copia dell'array
55     self._data = [None] * cap    # alloca una nuova list di dimensione cap
56     j = self._front
57     for k in range(self._size):
58       self._data[k] = old[j]     # shifta gli indici per riallinearli
59       j = (j + 1) % len(old)     # usa la vecchia dimensione come modulo
60     self._front = 0              # front riallineato a 0
```

Capitolo 3

Alberi

Listing 3.1: Classe astratta che rappresenta una struttura ad albero.

```python
from ..queue.array_queue import ArrayQueue
# import collections

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

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

    Note that two position instaces may represent the same inherent
       location in a tree.
    Therefore, users should always rely on syntax 'p == q' rather than 'p
       is q' when testing
    equivalence of positions.
    """

    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__

  #---------------------- end nested Position class -----------------------
```

```python
29
30     # ----- abstract methods that concrete subclass must support -----
31     def root(self):
32       """Return Position representing the tree's root (or None if empty)."""
33       raise NotImplementedError('must be implemented by subclass')
34
35     def parent(self, p):
36       """Return Position representing p's parent (or None if p is root)."""
37       raise NotImplementedError('must be implemented by subclass')
38
39     def num_children(self, p):
40       """Return the number of children that Position p has."""
41       raise NotImplementedError('must be implemented by subclass')
42
43     def children(self, p):
44       """Generate an iteration of Positions representing p's children."""
45       raise NotImplementedError('must be implemented by subclass')
46
47     def __len__(self):
48       """Return the total number of elements in the tree."""
49       raise NotImplementedError('must be implemented by subclass')
50
51     # ---------- concrete methods implemented in this class ----------
52     def is_root(self, p):
53       """Return True if Position p represents the root of the tree."""
54       return self.root() == p
55
56     def is_leaf(self, p):
57       """Return True if Position p does not have any children."""
58       return self.num_children(p) == 0
59
60     def is_empty(self):
61       """Return True if the tree is empty."""
62       return len(self) == 0
63
64     def depth(self, p):
65       """Return the number of levels separating Position p from the root."""
66       if self.is_root(p):
67         return 0
68       else:
69         return 1 + self.depth(self.parent(p))
```

```python
70
71  def _height1(self):                    # works, but O(n^2) worst-case time
72    """Return the height of the tree."""
73    return max(self.depth(p) for p in self.positions() if self.is_leaf(p))
74
75  def _height2(self, p):                  # time is linear in size of subtree
76    """Return the height of the subtree rooted at Position p."""
77    if self.is_leaf(p):
78      return 0
79    else:
80      return 1 + max(self._height2(c) for c in self.children(p))
81
82  def height(self, p=None):
83    """Return the height of the subtree rooted at Position p.
84    If p is None, return the height of the entire tree.
85    """
86    if p is None:
87      p = self.root()
88    return self._height2(p)          # start _height2 recursion
89
90  def __iter__(self):
91    """Generate an iteration of the tree's elements."""
92    for p in self.positions():       # use same order as positions()
93      yield p.element()              # but yield each element
94
95  def positions(self):
96    """Generate an iteration of the tree's positions."""
97    return self.preorder()           # return entire preorder iteration
```

```python
 98
 99   def preorder(self):
100     """Generate a preorder iteration of positions in the tree."""
101     if not self.is_empty():
102       for p in self._subtree_preorder(self.root()):  # start recursion
103         yield p
104
105   def _subtree_preorder(self, p):
106     """Generate a preorder iteration of positions in subtree rooted at p."""
107     yield p              # visit p before its subtrees
108     for c in self.children(p):          # for each child c
109       for other in self._subtree_preorder(c):   # do preorder of c's subtree
110         yield other         # yielding each to our caller
111
112   def postorder(self):
113     """Generate a postorder iteration of positions in the tree."""
114     if not self.is_empty():
115       for p in self._subtree_postorder(self.root()):  # start recursion
116         yield p
117
118   def _subtree_postorder(self, p):
119     """Generate a postorder iteration of positions in subtree rooted at
          p."""
120     for c in self.children(p):       # for each child c
121       for other in self._subtree_postorder(c):  # do postorder of c's
                subtree
122         yield other        # yielding each to our caller
123     yield p           # visit p after its subtrees
124
125   def breadthfirst(self):
126     """Generate a breadth-first iteration of the positions of the tree."""
127     if not self.is_empty():
128       fringe = ArrayQueue()            # known positions not yet yielded
129       fringe.enqueue(self.root())      # starting with the root
130       while not fringe.is_empty():
131         p = fringe.dequeue()           # remove from front of the queue
132         yield p                        # report this position
133         for c in self.children(p):
134           fringe.enqueue(c)            # add children to back of queue
```

Listing 3.2: Classe astratta che rappresenta una struttura ad albero binario.

```python
from .tree import Tree

class BinaryTree(Tree):
  """Abstract base class representing a binary tree structure."""

  # -------------------- additional abstract methods --------------------
  def left(self, p):
    """Return a Position representing p's left child.
    Return None if p does not have a left child.
    """
    raise NotImplementedError('must be implemented by subclass')

  def right(self, p):
    """Return a Position representing p's right child.
    Return None if p does not have a right child.
    """
    raise NotImplementedError('must be implemented by subclass')

  # --------- concrete methods implemented in this class ----------
  def sibling(self, p):
    """Return a Position representing p's sibling (or None if no
        sibling)."""
    parent = self.parent(p)
    if parent is None:    # p must be the root
      return None         # root has no sibling
    else:
      if p == self.left(parent):
        return self.right(parent)    # possibly None
      else:
        return self.left(parent)     # possibly None

  def children(self, p):
    """Generate an iteration of Positions representing p's children."""
    if self.left(p) is not None:
      yield self.left(p)
    if self.right(p) is not None:
      yield self.right(p)
```

```python
37
38   def inorder(self):
39     """Generate an inorder iteration of positions in the tree."""
40     if not self.is_empty():
41       for p in self._subtree_inorder(self.root()):
42         yield p
43
44   def _subtree_inorder(self, p):
45     """Generate an inorder iteration of positions in subtree rooted at p."""
46     if self.left(p) is not None:          # if left child exists, traverse
                                             its subtree
47       for other in self._subtree_inorder(self.left(p)):
48         yield other
49     yield p                               # visit p between its subtrees
50     if self.right(p) is not None:         # if right child exists, traverse
                                             its subtree
51       for other in self._subtree_inorder(self.right(p)):
52         yield other
53
54   # override inherited version to make inorder the default
55   def positions(self):
56     """Generate an iteration of the tree's positions."""
57     return self.inorder()                 # make inorder the default
```

Listing 3.3: Implementazione tramite lista linkata di una struttura ad albero binario.

```python
from .binary_tree import BinaryTree

class LinkedBinaryTree(BinaryTree):
  """Linked representation of a binary tree structure."""

  #--------------------------- nested _Node class ---------------------------
  class _Node:
    """Lightweight, nonpublic class for storing a node."""
    __slots__ = '_element', '_parent', '_left', '_right' # streamline
        memory usage

    def __init__(self, element, parent=None, left=None, right=None):
      self._element = element
      self._parent = parent
      self._left = left
      self._right = right

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

    def __init__(self, container, node):
      """Constructor should not be invoked by user."""
      self._container = container
      self._node = node

    def element(self):
      """Return the element stored at this Position."""
      return self._node._element

    def __eq__(self, other):
      """Return True if other is a Position representing the same
          location."""
      return type(other) is type(self) and other._node is self._node

  #--------------------------- utility methods ---------------------------
  def _validate(self, p):
    """Return associated node, if position is valid."""
    if not isinstance(p, self.Position):
      raise TypeError('p must be proper Position type')
    if p._container is not self:
      raise ValueError('p does not belong to this container')
    if p._node._parent is p._node:      # convention for deprecated nodes
      raise ValueError('p is no longer valid')
    return p._node

  def _make_position(self, node):
    """Return Position instance for given node (or None if no node)."""
    return self.Position(self, node) if node is not None else None
```

```python
48
49    #--------------------------- binary tree constructor ---------------------------
50    def __init__(self):
51      """Create an initially empty binary tree."""
52      self._root = None
53      self._size = 0
54
55    #--------------------------- public accessors ---------------------------
56    def __len__(self):
57      """Return the total number of elements in the tree."""
58      return self._size
59
60    def root(self):
61      """Return the root Position of the tree (or None if tree is empty)."""
62      return self._make_position(self._root)
63
64    def parent(self, p):
65      """Return the Position of p's parent (or None if p is root)."""
66      node = self._validate(p)
67      return self._make_position(node._parent)
68
69    def left(self, p):
70      """Return the Position of p's left child (or None if no left child)."""
71      node = self._validate(p)
72      return self._make_position(node._left)
73
74    def right(self, p):
75      """Return the Position of p's right child (or None if no right
           child)."""
76      node = self._validate(p)
77      return self._make_position(node._right)
78
79    def num_children(self, p):
80      """Return the number of children of Position p."""
81      node = self._validate(p)
82      count = 0
83      if node._left is not None:       # left child exists
84        count += 1
85      if node._right is not None:      # right child exists
86        count += 1
87      return count
```

```python
88
89   #------------------------ nonpublic mutators ------------------------
90   def _add_root(self, e):
91     """Place element e at the root of an empty tree and return new Position.
92
93     Raise ValueError if tree nonempty.
94     """
95     if self._root is not None:
96       raise ValueError('Root exists')
97     self._size = 1
98     self._root = self._Node(e)
99     return self._make_position(self._root)
100
101   def _add_left(self, p, e):
102     """Create a new left child for Position p, storing element e.
103
104     Return the Position of new node.
105     Raise ValueError if Position p is invalid or p already has a left child.
106     """
107     node = self._validate(p)
108     if node._left is not None:
109       raise ValueError('Left child exists')
110     self._size += 1
111     node._left = self._Node(e, node)            # node is its parent
112     return self._make_position(node._left)
113
114   def _add_right(self, p, e):
115     """Create a new right child for Position p, storing element e.
116
117     Return the Position of new node.
118     Raise ValueError if Position p is invalid or p already has a right
119         child.
       """
120     node = self._validate(p)
121     if node._right is not None:
122       raise ValueError('Right child exists')
123     self._size += 1
124     node._right = self._Node(e, node)           # node is its parent
125     return self._make_position(node._right)
126
127   def _replace(self, p, e):
128     """Replace the element at position p with e, and return old element."""
129     node = self._validate(p)
130     old = node._element
131     node._element = e
132     return old
```

17

```python
      def _delete(self, p):
        """Delete the node at Position p, and replace it with its child, if any.

        Return the element that had been stored at Position p.
        Raise ValueError if Position p is invalid or p has two children.
        """
        node = self._validate(p)
        if self.num_children(p) == 2:
          raise ValueError('Position has two children')
        child = node._left if node._left else node._right  # might be None
        if child is not None:
          child._parent = node._parent   # child's grandparent becomes parent
        if node is self._root:
          self._root = child             # child becomes root
        else:
          parent = node._parent
          if node is parent._left:
            parent._left = child
          else:
            parent._right = child
        self._size -= 1
        node._parent = node              # convention for deprecated node
        return node._element

      def _attach(self, p, t1, t2):
        """Attach trees t1 and t2, respectively, as the left and right subtrees
           of the external Position p.

        As a side effect, set t1 and t2 to empty.
        Raise TypeError if trees t1 and t2 do not match type of this tree.
        Raise ValueError if Position p is invalid or not external.
        """
        node = self._validate(p)
        if not self.is_leaf(p):
          raise ValueError('position must be leaf')
        if not type(self) is type(t1) is type(t2):    # all 3 trees must be
            same type
          raise TypeError('Tree types must match')
        self._size += len(t1) + len(t2)
        if not t1.is_empty():          # attached t1 as left subtree of node
          t1._root._parent = node
          node._left = t1._root
          t1._root = None              # set t1 instance to empty
          t1._size = 0
        if not t2.is_empty():          # attached t2 as right subtree of node
          t2._root._parent = node
          node._right = t2._root
          t2._root = None              # set t2 instance to empty
          t2._size = 0
```

Capitolo 4

Binary Search Tree (BST)

Listing 4.1: Classe astratta di base che include una classe non pubblica _Item per la memorizzazione di coppie chiave-valore.

```python
from collections.abc import MutableMapping

class MapBase(MutableMapping):
  """Our own abstract base class that includes a nonpublic _Item class."""

  #------------------------- nested _Item class -------------------------
  class _Item:
    """Lightweight composite to store key-value pairs as map items."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __eq__(self, other):
      return self._key == other._key   # compare items based on their keys

    def __ne__(self, other):
      return not (self == other)       # opposite of __eq__

    def __lt__(self, other):
      return self._key < other._key    # compare items based on their keys
```

Listing 4.2: Implementazione di una sorted map utilizzando un albero di ricerca binario.

```python
from ..tree.linked_binary_tree import LinkedBinaryTree
from .map_base import MapBase

class TreeMap(LinkedBinaryTree, MapBase):
  """Sorted map implementation using a binary search tree."""

  #----------------------- override Position class -----------------------
  class Position(LinkedBinaryTree.Position):
    def key(self):
      """Return key of map's key-value pair."""
      return self.element()._key

    def value(self):
      """Return value of map's key-value pair."""
      return self.element()._value

  #----------------------- nonpublic utilities -------------------------
  def _subtree_search(self, p, k):
    """Return Position of p's subtree having key k, or last node
       searched."""
    if k == p.key():       # found match
      return p
    elif k < p.key():      # search left subtree
      if self.left(p) is not None:
        return self._subtree_search(self.left(p), k)
    else:                  # search right subtree
      if self.right(p) is not None:
        return self._subtree_search(self.right(p), k)
    return p               # unsucessful search

  def _subtree_first_position(self, p):
    """Return Position of first item in subtree rooted at p."""
    walk = p
    while self.left(walk) is not None:      # keep walking left
      walk = self.left(walk)
    return walk

  def _subtree_last_position(self, p):
    """Return Position of last item in subtree rooted at p."""
    walk = p
    while self.right(walk) is not None:     # keep walking right
      walk = self.right(walk)
    return walk
```

```python
43
44     #--------- public methods providing "positional" support ----------------
45     def first(self):
46       """Return the first Position in the tree (or None if empty)."""
47       return self._subtree_first_position(self.root()) if len(self) > 0 else
           None
48
49     def last(self):
50       """Return the last Position in the tree (or None if empty)."""
51       return self._subtree_last_position(self.root()) if len(self) > 0 else
           None
52
53     def before(self, p):
54       """Return the Position just before p in the natural order.
55
56       Return None if p is the first position.
57       """
58       self._validate(p)                # inherited from LinkedBinaryTree
59       if self.left(p):
60         return self._subtree_last_position(self.left(p))
61       else:
62         # walk upward
63         walk = p
64         above = self.parent(walk)
65         while above is not None and walk == self.left(above):
66           walk = above
67           above = self.parent(walk)
68         return above
69
70     def after(self, p):
71       """Return the Position just after p in the natural order.
72
73       Return None if p is the last position.
74       """
75       self._validate(p)                # inherited from LinkedBinaryTree
76       if self.right(p):
77         return self._subtree_first_position(self.right(p))
78       else:
79         walk = p
80         above = self.parent(walk)
81         while above is not None and walk == self.right(above):
82           walk = above
83           above = self.parent(walk)
84         return above
```

```python
85
86  def find_position(self, k):
87    """Return position with key k, or else neighbor (or None if empty)."""
88    if self.is_empty():
89      return None
90    else:
91      p = self._subtree_search(self.root(), k)
92      self._rebalance_access(p)      # hook for balanced tree subclasses
93      return p
94
95  def delete(self, p):
96    """Remove the item at given Position."""
97    self._validate(p)                  # inherited from LinkedBinaryTree
98    if self.left(p) and self.right(p):      # p has two children
99      replacement = self._subtree_last_position(self.left(p))
100      self._replace(p, replacement.element())    # from LinkedBinaryTree
101      p =  replacement
102    # now p has at most one child
103    parent = self.parent(p)
104    self._delete(p)                    # inherited from LinkedBinaryTree
105    self._rebalance_delete(parent)      # if root deleted, parent is None
106
107  #--------------- public methods for (standard) map interface -----------
108  def __getitem__(self, k):
109    """Return value associated with key k (raise KeyError if not found)."""
110    if self.is_empty():
111      raise KeyError('Key Error: ' + repr(k))
112    else:
113      p = self._subtree_search(self.root(), k)
114      self._rebalance_access(p)          # hook for balanced tree subclasses
115      if k != p.key():
116        raise KeyError('Key Error: ' + repr(k))
117      return p.value()
```

```python
118
119    def __setitem__(self, k, v):
120      """Assign value v to key k, overwriting existing value if present."""
121      if self.is_empty():
122        leaf = self._add_root(self._Item(k,v))      # from LinkedBinaryTree
123      else:
124        p = self._subtree_search(self.root(), k)
125        if p.key() == k:
126          p.element()._value = v                # replace existing item's value
127          self._rebalance_access(p)         # hook for balanced tree subclasses
128          return
129        else:
130          item = self._Item(k,v)
131          if p.key() < k:
132            leaf = self._add_right(p, item)   # inherited from
                     LinkedBinaryTree
133          else:
134            leaf = self._add_left(p, item)    # inherited from
                     LinkedBinaryTree
135      self._rebalance_insert(leaf)                # hook for balanced tree
             subclasses
136
137    def __delitem__(self, k):
138      """Remove item associated with key k (raise KeyError if not found)."""
139      if not self.is_empty():
140        p = self._subtree_search(self.root(), k)
141        if k == p.key():
142          self.delete(p)                  # rely on positional version
143          return                          # successful deletion complete
144        self._rebalance_access(p)         # hook for balanced tree subclasses
145      raise KeyError('Key Error: ' + repr(k))
146
147    def __iter__(self):
148      """Generate an iteration of all keys in the map in order."""
149      p = self.first()
150      while p is not None:
151        yield p.key()
152        p = self.after(p)
```

```python
153
154     #---------------- public methods for sorted map interface --------------
155     def __reversed__(self):
156       """Generate an iteration of all keys in the map in reverse order."""
157       p = self.last()
158       while p is not None:
159         yield p.key()
160         p = self.before(p)
161
162     def find_min(self):
163       """Return (key,value) pair with minimum key (or None if empty)."""
164       if self.is_empty():
165         return None
166       else:
167         p = self.first()
168         return (p.key(), p.value())
169
170     def find_max(self):
171       """Return (key,value) pair with maximum key (or None if empty)."""
172       if self.is_empty():
173         return None
174       else:
175         p = self.last()
176         return (p.key(), p.value())
177
178     def find_le(self, k):
179       """Return (key,value) pair with greatest key less than or equal to k.
180
181       Return None if there does not exist such a key.
182       """
183       if self.is_empty():
184         return None
185       else:
186         p = self.find_position(k)
187         if k < p.key():
188           p = self.before(p)
189         return (p.key(), p.value()) if p is not None else None
190
191     def find_lt(self, k):
192       """Return (key,value) pair with greatest key strictly less than k.
193
194       Return None if there does not exist such a key.
195       """
196       if self.is_empty():
197         return None
198       else:
199         p = self.find_position(k)
200         if not p.key() < k:
201           p = self.before(p)
202         return (p.key(), p.value()) if p is not None else None
```

```python
203
204    def find_ge(self, k):
205      """Return (key,value) pair with least key greater than or equal to k.
206
207      Return None if there does not exist such a key.
208      """
209      if self.is_empty():
210        return None
211      else:
212        p = self.find_position(k)          # may not find exact match
213        if p.key() < k:                    # p's key is too small
214          p = self.after(p)
215        return (p.key(), p.value()) if p is not None else None
216
217    def find_gt(self, k):
218      """Return (key,value) pair with least key strictly greater than k.
219
220      Return None if there does not exist such a key.
221      """
222      if self.is_empty():
223        return None
224      else:
225        p = self.find_position(k)
226        if not k < p.key():
227          p = self.after(p)
228        return (p.key(), p.value()) if p is not None else None
229
230    def find_range(self, start, stop):
231      """Iterate all (key,value) pairs such that start <= key < stop.
232
233      If start is None, iteration begins with minimum key of map.
234      If stop is None, iteration continues through the maximum key of map.
235      """
236      if not self.is_empty():
237        if start is None:
238          p = self.first()
239        else:
240          # we initialize p with logic similar to find_ge
241          p = self.find_position(start)
242          if p.key() < start:
243            p = self.after(p)
244        while p is not None and (stop is None or p.key() < stop):
245          yield (p.key(), p.value())
246          p = self.after(p)
```

```python
247
248    #-------------- hooks used by subclasses to balance a tree --------------
249    def _rebalance_insert(self, p):
250      """Call to indicate that position p is newly added."""
251      pass
252
253    def _rebalance_delete(self, p):
254      """Call to indicate that a child of p has been removed."""
255      pass
256
257    def _rebalance_access(self, p):
258      """Call to indicate that position p was recently accessed."""
259      pass
260
261    #------------ nonpublic methods to support tree balancing --------------
262    def _relink(self, parent, child, make_left_child):
263      """Relink parent node with child node (we allow child to be None)."""
264      if make_left_child:                # make it a left child
265        parent._left = child
266      else:                              # make it a right child
267        parent._right = child
268      if child is not None:              # make child point to parent
269        child._parent = parent
```

```python
def _rotate(self, p):
    """Rotate Position p above its parent.

    Switches between these configurations, depending on whether p==a or
        p==b.


          b                    a
         / \                  /  \
        a   t2              t0    b
       / \                      / \
      t0  t1                   t1  t2


    Caller should ensure that p is not the root.
    """
    """Rotate Position p above its parent."""
    x = p._node
    y = x._parent                    # we assume this exists
    z = y._parent                    # grandparent (possibly None)
    if z is None:
        self._root = x               # x becomes root
        x._parent = None
    else:
        self._relink(z, x, y == z._left)     # x becomes a direct child of z
    # now rotate x and y, including transfer of middle subtree
    if x == y._left:
        self._relink(y, x._right, True)       # x._right becomes left child
            of y
        self._relink(x, y, False)             # y becomes right child of x
    else:
        self._relink(y, x._left, False)       # x._left becomes right child
            of y
        self._relink(x, y, True)              # y becomes left child of x
```

27

```python
def _restructure(self, x):
    """Perform a trinode restructure among Position x, its parent, and its
        grandparent.

    Return the Position that becomes root of the restructured subtree.

    Assumes the nodes are in one of the following configurations:

        z=a                     z=c             z=a                     z=c
       /  \                    /  \            /  \                    /  \
      t0   y=b               y=b  t3         t0    y=c               y=a  t3
          /  \              /  \                  /  \              /  \
         t1   x=c          x=a  t2              x=b  t3         t0    x=b
             /  \         /  \                 /  \                  /  \
            t2   t3      t0   t1              t1   t2              t1   t2

    The subtree will be restructured so that the node with key b becomes
        its root.

              b
            /    \
          a        c
         / \      / \
        t0  t1   t2  t3

    Caller should ensure that x has a grandparent.
    """
    """Perform trinode restructure of Position x with parent/grandparent."""
    y = self.parent(x)
    z = self.parent(y)
    if (x == self.right(y)) == (y == self.right(z)):  # matching alignments
        self._rotate(y)                 # single rotation (of y)
        return y                        # y is new subtree root
    else:                                               # opposite alignments
        self._rotate(x)                 # double rotation (of x)
        self._rotate(x)
        return x                        # x is new subtree root
```

Capitolo 5

AVL Tree

Listing 5.1: Implementazione della classe AVLTreeMap e gestione del bilanciamento.

```python
from .binary_search_tree import TreeMap

class AVLTreeMap(TreeMap):
  """Sorted map implementation using an AVL tree."""

  #--------------------------- nested _Node class ---------------------------
  class _Node(TreeMap._Node):
    """Node class for AVL maintains height value for balancing.

    We use convention that a "None" child has height 0, thus a leaf has
        height 1.
    """
    __slots__ = '_height'         # additional data member to store height

    def __init__(self, element, parent=None, left=None, right=None):
      super().__init__(element, parent, left, right)
      self._height = 0              # will be recomputed during balancing

    def left_height(self):
      return self._left._height if self._left is not None else 0

    def right_height(self):
      return self._right._height if self._right is not None else 0

  #----------------------- 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
```

```python
30    def _tall_child(self, p, favorleft=False): # parameter controls tiebreaker
31      if p._node.left_height() + (1 if favorleft else 0) >
           p._node.right_height():
32        return self.left(p)
33      else:
34        return self.right(p)
35
36    def _tall_grandchild(self, p):
37      child = self._tall_child(p)
38      # if child is on left, favor left grandchild; else favor right
           grandchild
39      alignment = (child == self.left(p))
40      return self._tall_child(child, alignment)
41
42    def _rebalance(self, p):
43      while p is not None:
44        old_height = p._node._height                    # trivially 0
             if new node
45        if not self._isbalanced(p):                     # imbalance
             detected!
46          # perform trinode restructuring, setting p to resulting root,
47          # and recompute new local heights after the restructuring
48          p = self._restructure(self._tall_grandchild(p))
49          self._recompute_height(self.left(p))
50          self._recompute_height(self.right(p))
51        self._recompute_height(p)                       # adjust for
             recent changes
52        if p._node._height == old_height:               # has height
             changed?
53          p = None                                      # no further
               changes needed
54        else:
55          p = self.parent(p)                            # repeat with
               parent
56
57    #---------------------------- override balancing hooks
         ----------------------------
58    def _rebalance_insert(self, p):
59      self._rebalance(p)
60
61    def _rebalance_delete(self, p):
62      self._rebalance(p)
```

Capitolo 6

RB Tree

Listing 6.1: Implementazione della classe RedBlackTreeMap e gestione del colore dei nodi.

```python
from .binary_search_tree import TreeMap

class RedBlackTreeMap(TreeMap):
  """Sorted map implementation using a red-black tree."""

  #-------------------------- nested _Node class --------------------------
  class _Node(TreeMap._Node):
    """Node class for red-black tree maintains bit that denotes color."""
    __slots__ = '_red'     # add additional data member to the Node class

    def __init__(self, element, parent=None, left=None, right=None):
      super().__init__(element, parent, left, right)
      self._red = True     # new node red by default

  #----------------------- 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
```

```python
29
30      #----------------------- support for insertions
                ------------------------
31      def _rebalance_insert(self, p):
32        self._resolve_red(p)                          # new node is always red
33
34      def _resolve_red(self, p):
35        if self.is_root(p):
36          self._set_black(p)                          # make root black
37        else:
38          parent = self.parent(p)
39          if self._is_red(parent):                   # double red problem
40            uncle = self.sibling(parent)
41            if not self._is_red(uncle):              # Case 1: misshapen 4-node
42              middle = self._restructure(p)          # do trinode restructuring
43              self._set_black(middle)                # and then fix colors
44              self._set_red(self.left(middle))
45              self._set_red(self.right(middle))
46            else:                                     # Case 2: overfull 5-node
47              grand = self.parent(parent)
48              self._set_red(grand)                    # grandparent becomes red
49              self._set_black(self.left(grand))      # its children become black
50              self._set_black(self.right(grand))
51              self._resolve_red(grand)                # recur at red grandparent
52
53      #----------------------- support for deletions -------------------------
54      def _rebalance_delete(self, p):
55        if len(self) == 1:
56          self._set_black(self.root())  # special case: ensure that root is
                black
57        elif p is not None:
58          n = self.num_children(p)
59          if n == 1:                         # deficit exists unless child is a red
                leaf
60            c = next(self.children(p))
61            if not self._is_red_leaf(c):
62              self._fix_deficit(p, c)
63          elif n == 2:                        # removed black node with red child
64            if self._is_red_leaf(self.left(p)):
65              self._set_black(self.left(p))
66            else:
67              self._set_black(self.right(p))
```

```python
def _fix_deficit(self, z, y):
    """Resolve black deficit at z, where y is the root of z's heavier
        subtree."""
    if not self._is_red(y): # y is black; will apply Case 1 or 2
        x = self._get_red_child(y)
        if x is not None: # Case 1: y is black and has red child x; do
            "transfer"
            old_color = self._is_red(z)
            middle = self._restructure(x)
            self._set_color(middle, old_color)   # middle gets old color of z
            self._set_black(self.left(middle))    # children become black
            self._set_black(self.right(middle))
        else: # Case 2: y is black, but no red children; recolor as "fusion"
            self._set_red(y)
            if self._is_red(z):
                self._set_black(z)                      # this resolves the problem
            elif not self.is_root(z):
                self._fix_deficit(self.parent(z), self.sibling(z)) # recur upward
    else: # Case 3: y is red; rotate misaligned 3-node and repeat
        self._rotate(y)
        self._set_black(y)
        self._set_red(z)
        if z == self.right(y):
            self._fix_deficit(z, self.left(z))
        else:
            self._fix_deficit(z, self.right(z))
```

Capitolo 7

Hash

Listing 7.1: Classe base HashMapBase con funzione di hashing MAD.

```python
from .map_base import MapBase
from random import randrange        # used to pick MAD parameters

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

  Keys must be hashable and non-None.
  """

  def __init__(self, cap=11, p=109345121):
    """Create an empty hash-table map.

    cap      initial table size (default 11)
    p        positive prime used for MAD (default 109345121)
    """
    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
```

```python
27
28   def __getitem__(self, k):
29     j = self._hash_function(k)
30     return self._bucket_getitem(j, k)              # may raise KeyError
31
32   def __setitem__(self, k, v):
33     j = self._hash_function(k)
34     self._bucket_setitem(j, k, v)                  # subroutine maintains
             self._n
35     if self._n > len(self._table) // 2:            # keep load factor <= 0.5
36       self._resize(2 * len(self._table) - 1)       # number 2^x - 1 is often
             prime
37
38   def __delitem__(self, k):
39     j = self._hash_function(k)
40     self._bucket_delitem(j, k)                     # may raise KeyError
41     self._n -= 1
42
43   def _resize(self, c):
44     """Resize bucket array to capacity c and rehash all items."""
45     old = list(self.items())      # use iteration to record existing items
46     self._table = c * [None]      # then reset table to desired capacity
47     self._n = 0                   # n recomputed during subsequent adds
48     for (k,v) in old:
49       self[k] = v                 # reinsert old key-value pair
```

Capitolo 8

Sorted Unsorted Table

Listing 8.1: Implementazione di una mappa tramite tabella ordinata (Sorted Table).

```python
from .map_base import MapBase

class SortedTableMap(MapBase):
  """Map implementation using a sorted table."""

  #-------------------------------- nonpublic behaviors
     --------------------------------
  def _find_index(self, k, low, high):
    """Return index of the leftmost item with key greater than or equal to
      k.

    Return high + 1 if no such item qualifies.

    That is, j will be returned such that:
        all items of slice table[low:j] have key < k
        all items of slice table[j:high+1] have key >= k
    """
    if high < low:
      return high + 1                          # no element qualifies
    else:
      mid = (low + high) // 2
      if k == self._table[mid]._key:
        return mid                             # found exact match
      elif k < self._table[mid]._key:
        return self._find_index(k, low, mid - 1)   # Note: may return mid
      else:
        return self._find_index(k, mid + 1, high)  # answer is right of mid
```

37

```python
26
27     #-------------------------- public behaviors
           --------------------------
28     def __init__(self):
29       """Create an empty map."""
30       self._table = []
31
32     def __len__(self):
33       """Return number of items in the map."""
34       return len(self._table)
35
36     def __getitem__(self, k):
37       """Return value associated with key k (raise KeyError if not found)."""
38       j = self._find_index(k, 0, len(self._table) - 1)
39       if j == len(self._table) or self._table[j]._key != k:
40         raise KeyError('Key Error: ' + repr(k))
41       return self._table[j]._value
42
43     def __setitem__(self, k, v):
44       """Assign value v to key k, overwriting existing value if present."""
45       j = self._find_index(k, 0, len(self._table) - 1)
46       if j < len(self._table) and self._table[j]._key == k:
47         self._table[j]._value = v                    # reassign value
48       else:
49         self._table.insert(j, self._Item(k,v))       # adds new item
50
51     def __delitem__(self, k):
52       """Remove item associated with key k (raise KeyError if not found)."""
53       j = self._find_index(k, 0, len(self._table) - 1)
54       if j == len(self._table) or self._table[j]._key != k:
55         raise KeyError('Key Error: ' + repr(k))
56       self._table.pop(j)                             # delete item
57
58     def __iter__(self):
59       """Generate keys of the map ordered from minimum to maximum."""
60       for item in self._table:
61         yield item._key
62
63     def __reversed__(self):
64       """Generate keys of the map ordered from maximum to minimum."""
65       for item in reversed(self._table):
66         yield item._key
67
68     def find_min(self):
69       """Return (key,value) pair with minimum key (or None if empty)."""
70       if len(self._table) > 0:
71         return (self._table[0]._key, self._table[0]._value)
72       else:
73         return None
74
```

```python
    def find_max(self):
        """Return (key,value) pair with maximum key (or None if empty)."""
        if len(self._table) > 0:
            return (self._table[-1]._key, self._table[-1]._value)
        else:
            return None

    def find_le(self, k):
        """Return (key,value) pair with greatest key less than or equal to k.

        Return None if there does not exist such a key.
        """
        j = self._find_index(k, 0, len(self._table) - 1)      # j's key >= k
        if j < len(self._table) and self._table[j]._key == k:
            return (self._table[j]._key, self._table[j]._value)      # exact match
        elif j > 0:
            return (self._table[j-1]._key, self._table[j-1]._value)  # Note use
                of j-1
        else:
            return None

    def find_ge(self, k):
        """Return (key,value) pair with least key greater than or equal to k.

        Return None if there does not exist such a key.
        """
        j = self._find_index(k, 0, len(self._table) - 1)      # j's key >= k
        if j < len(self._table):
            return (self._table[j]._key, self._table[j]._value)
        else:
            return None

    def find_lt(self, k):
        """Return (key,value) pair with greatest key strictly less than k.

        Return None if there does not exist such a key.
        """
        j = self._find_index(k, 0, len(self._table) - 1)      # j's key >= k
        if j > 0:
            return (self._table[j-1]._key, self._table[j-1]._value)  # Note use
                of j-1
        else:
            return None
```

```python
116
117  def find_gt(self, k):
118    """Return (key,value) pair with least key strictly greater than k.
119
120    Return None if there does not exist such a key.
121    """
122    j = self._find_index(k, 0, len(self._table) - 1)       # j's key >= k
123    if j < len(self._table) and self._table[j]._key == k:
124      j += 1                                               # advanced past match
125    if j < len(self._table):
126      return (self._table[j]._key, self._table[j]._value)
127    else:
128      return None
129
130  def find_range(self, start, stop):
131    """Iterate all (key,value) pairs such that start <= key < stop.
132
133    If start is None, iteration begins with minimum key of map.
134    If stop is None, iteration continues through the maximum key of map.
135    """
136    if start is None:
137      j = 0
138    else:
139      j = self._find_index(start, 0, len(self._table)-1)  # find first
          result
140    while j < len(self._table) and (stop is None or self._table[j]._key <
          stop):
141      yield (self._table[j]._key, self._table[j]._value)
142      j += 1
```

Listing 8.2: Implementazione di una mappa tramite tabella non ordinata (Unsorted Table).

```python
from .map_base import MapBase

class UnsortedTableMap(MapBase):
  """Map implementation using an unordered list."""

  def __init__(self):
    """Create an empty map."""
    self._table = []                            # list of _Item's

  def __getitem__(self, k):
    """Return value associated with key k (raise KeyError if not found)."""
    for item in self._table:
      if k == item._key:
        return item._value
    raise KeyError('Key Error: ' + repr(k))

  def __setitem__(self, k, v):
    """Assign value v to key k, overwriting existing value if present."""
    for item in self._table:
      if k == item._key:                        # Found a match:
        item._value = v                         # reassign value
        return                                  # and quit
    # did not find match for key
    self._table.append(self._Item(k,v))

  def __delitem__(self, k):
    """Remove item associated with key k (raise KeyError if not found)."""
    for j in range(len(self._table)):
      if k == self._table[j]._key:              # Found a match:
        self._table.pop(j)                      # remove item
        return                                  # and quit
    raise KeyError('Key Error: ' + repr(k))

  def __len__(self):
    """Return number of items in the map."""
    return len(self._table)

  def __iter__(self):
    """Generate iteration of the map's keys."""
    for item in self._table:
      yield item._key                           # yield the KEY
```

Capitolo 9

Chain Hash

Listing 9.1: Gestione delle collisioni tramite Separate Chaining nella classe ChainHashMap.

```python
from .hash_map_base import HashMapBase
from .unsorted_table_map import UnsortedTableMap

class ChainHashMap(HashMapBase):
  """Hash map implemented with separate chaining for collision
     resolution."""

  def _bucket_getitem(self, j, k):
    bucket = self._table[j]
    if bucket is None:
      raise KeyError('Key Error: ' + repr(k))      # no match found
    return bucket[k]                               # may raise KeyError

  def _bucket_setitem(self, j, k, v):
    if self._table[j] is None:
      self._table[j] = UnsortedTableMap()      # bucket is new to the table
    oldsize = len(self._table[j])
    self._table[j][k] = v
    if len(self._table[j]) > oldsize:          # key was new to the table
      self._n += 1                             # increase overall map size

  def _bucket_delitem(self, j, k):
    bucket = self._table[j]
    if bucket is None:
      raise KeyError('Key Error: ' + repr(k))      # no match found
    del bucket[k]                                  # may raise KeyError

  def __iter__(self):
    for bucket in self._table:
      if bucket is not None:                         # a nonempty slot
        for key in bucket:
          yield key
```

Capitolo 10

Probe Hash

Listing 10.1: Implementazione di una mappa tramite tabella hash con indirizzamento aperto.

```python
from .hash_map_base import HashMapBase

class ProbeHashMap(HashMapBase):
  """Hash map implemented with linear probing for collision resolution."""
  _AVAIL = object()          # sentinal marks locations of previous deletions

  def _is_available(self, j):
    """Return True if index j is available in table."""
    return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL

  def _find_slot(self, j, k):
    """Search for key k in bucket at index j.

    Return (success, index) tuple, described as follows:
    If match was found, success is True and index denotes its location.
    If no match found, success is False and index denotes first available
        slot.
    """
    firstAvail = None
    while True:
      if self._is_available(j):
        if firstAvail is None:
          firstAvail = j                      # mark this as first avail
        if self._table[j] is None:
          return (False, firstAvail)          # search has failed
      elif k == self._table[j]._key:
        return (True, j)                       # found a match
      j = (j + 1) % len(self._table)          # keep looking (cyclically)
```

```python
28
29    def _bucket_getitem(self, j, k):
30      found, s = self._find_slot(j, k)
31      if not found:
32        raise KeyError('Key Error: ' + repr(k))        # no match found
33      return self._table[s]._value
34
35    def _bucket_setitem(self, j, k, v):
36      found, s = self._find_slot(j, k)
37      if not found:
38        self._table[s] = self._Item(k,v)               # insert new item
39        self._n += 1                                    # size has increased
40      else:
41        self._table[s]._value = v                       # overwrite existing
42
43    def _bucket_delitem(self, j, k):
44      found, s = self._find_slot(j, k)
45      if not found:
46        raise KeyError('Key Error: ' + repr(k))        # no match found
47      self._table[s] = ProbeHashMap._AVAIL             # mark as vacated
48
49    def __iter__(self):
50      for j in range(len(self._table)):                 # scan entire table
51        if not self._is_available(j):
52          yield self._table[j]._key
```

Capitolo 11

Priority Queue

Listing 11.1: Classe base PriorityQueueBase: definizione della struttura _Item e dell'interfaccia per le code di priorità.

```python
class PriorityQueueBase:
  """Abstract base class for a priority queue."""

  #------------------------------ nested _Item class
      ------------------------------
  class _Item:
    """Lightweight composite to store priority queue items."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __lt__(self, other):
      return self._key < other._key    # compare items based on their keys

    def __repr__(self):
      return '({0},{1})'.format(self._key, self._value)

  #------------------------------ public behaviors
      ------------------------------
  def is_empty(self):                    # concrete method assuming abstract
      len
    """Return True if the priority queue is empty."""
    return len(self) == 0

  def __len__(self):
    """Return the number of items in the priority queue."""
    raise NotImplementedError('deve essere implementato dalla sottoclasse.')

  def add(self, key, value):
    """Add a key-value pair."""
    raise NotImplementedError('deve essere implementato dalla sottoclasse.')
```

```python
def min(self):
    """Return but do not remove (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    raise NotImplementedError('deve essere implementato dalla sottoclasse.')

def remove_min(self):
    """Remove and return (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    raise NotImplementedError('deve essere implementato dalla sottoclasse.')
```

Capitolo 12

Sorted Unsorted Priority Queue

Listing 12.1: Implementazione di SortedPriorityQueue.

```python
from .priority_queue_base import PriorityQueueBase
from ..list.positional_list import PositionalList

class Empty(Exception):
  pass

class SortedPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with a sorted list."""

  #---------------------------- public behaviors
      ----------------------------
  def __init__(self):
    """Create a new empty Priority Queue."""
    self._data = PositionalList()

  def __len__(self):
    """Return the number of items in the priority queue."""
    return len(self._data)

  def add(self, key, value):
    """Add a key-value pair."""
    newest = self._Item(key, value)                  # make new item instance
    walk = self._data.last()        # walk backward looking for smaller key
    while walk is not None and newest < walk.element():
      walk = self._data.before(walk)
    if walk is None:
      self._data.add_first(newest)                 # new key is smallest
    else:
      self._data.add_after(walk, newest)           # newest goes after walk
```

49

```
29
30   def min(self):
31     """Return but do not remove (k,v) tuple with minimum key.
32
33     Raise Empty exception if empty.
34     """
35     if self.is_empty():
36       raise Empty('Priority queue is empty.')
37     p = self._data.first()
38     item = p.element()
39     return (item._key, item._value)
40
41   def remove_min(self):
42     """Remove and return (k,v) tuple with minimum key.
43
44     Raise Empty exception if empty.
45     """
46     if self.is_empty():
47       raise Empty('Priority queue is empty.')
48     item = self._data.delete(self._data.first())
49     return (item._key, item._value)
```

Listing 12.2: Implementazione di UnsortedPriorityQueue.

```
1  from .priority_queue_base import PriorityQueueBase
2  from ..list.positional_list import PositionalList
3
4  class Empty(Exception):
5    pass
6
7
8  class UnsortedPriorityQueue(PriorityQueueBase): # base class defines _Item
9    """A min-oriented priority queue implemented with an unsorted list."""
10
11   #---------------------------- nonpublic behavior
           ----------------------------
12   def _find_min(self):
13     """Return Position of item with minimum key."""
14     if self.is_empty():                  # is_empty inherited from base class
15       raise Empty('Priority queue is empty')
16     small = self._data.first()
17     walk = self._data.after(small)
18     while walk is not None:
19       if walk.element() < small.element():
20         small = walk
21       walk = self._data.after(walk)
22     return small
```

```python
 23
 24    #---------------------------- public behaviors
          ----------------------------
 25    def __init__(self):
 26      """Create a new empty Priority Queue."""
 27      self._data = PositionalList()
 28
 29    def __len__(self):
 30      """Return the number of items in the priority queue."""
 31      return len(self._data)
 32
 33    def add(self, key, value):
 34      """Add a key-value pair."""
 35      self._data.add_last(self._Item(key, value))
 36
 37    def min(self):
 38      """Return but do not remove (k,v) tuple with minimum key.
 39
 40      Raise Empty exception if empty.
 41      """
 42      p = self._find_min()
 43      item = p.element()
 44      return (item._key, item._value)
 45
 46    def remove_min(self):
 47      """Remove and return (k,v) tuple with minimum key.
 48
 49      Raise Empty exception if empty.
 50      """
 51      p = self._find_min()
 52      item = self._data.delete(p)
 53      return (item._key, item._value)
```

Capitolo 13

Heap Priority Queue

Listing 13.1: Implementazione di HeapPriorityQueue: gestione di un heap binario tramite array con algoritmi di upheap e downheap per il bilanciamento.

```python
from .priority_queue_base import PriorityQueueBase

class Empty(Exception):
  pass


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]
```

```python
29
30  def _upheap(self, j):
31    parent = self._parent(j)
32    if j > 0 and self._data[j] < self._data[parent]:
33      self._swap(j, parent)
34      self._upheap(parent)              # recur at position of parent
35
36  def _downheap(self, j):
37    if self._has_left(j):
38      left = self._left(j)
39      small_child = left                # although right may be smaller
40      if self._has_right(j):
41        right = self._right(j)
42        if self._data[right] < self._data[left]:
43          small_child = right
44      if self._data[small_child] < self._data[j]:
45        self._swap(j, small_child)
46        self._downheap(small_child)     # recur at position of small child
47
48  def _heapify(self):
49    """Bottom-up construction of a heap in O(n) time."""
50    # start at the parent of the last element
51    start = self._parent(len(self._data) - 1)
52    for j in range(start, -1, -1):  # go backward from start to 0
53      self._downheap(j)
54
55  #---------------------------- public behaviors
        ----------------------------
56  def __init__(self, contents=()):
57    """Create a new empty Priority Queue.
58
59    If contents is provided, it should be an iterable of (k,v) tuples.
60    """
61    self._data = [self._Item(k, v) for k, v in contents]
62    if len(self._data) > 1:
63      self._heapify()
64
65  """ without Heapify
66  def __init__(self):
67    Create a new empty Priority Queue.
68    self._data = []
69  """
70
71  def __len__(self):
72    """Return the number of items in the priority queue."""
73    return len(self._data)
```

```python
74
75  def add(self, key, value):
76      """Add a key-value pair to the priority queue."""
77      self._data.append(self._Item(key, value))
78      self._upheap(len(self._data) - 1)              # upheap newly added
                position
79
80  def min(self):
81      """Return but do not remove (k,v) tuple with minimum key.
82
83      Raise Empty exception if empty.
84      """
85      if self.is_empty():
86          raise Empty('Priority queue is empty.')
87      item = self._data[0]
88      return (item._key, item._value)
89
90  def remove_min(self):
91      """Remove and return (k,v) tuple with minimum key.
92
93      Raise Empty exception if empty.
94      """
95      if self.is_empty():
96          raise Empty('Priority queue is empty.')
97      self._swap(0, len(self._data) - 1)             # put minimum item at the
                end
98      item = self._data.pop()                        # and remove it from the
                list;
99      self._downheap(0)                              # then fix new root
100     return (item._key, item._value)
```

Capitolo 14

Adaptable Priority Queue

Listing 14.1: Implementazione di AdaptableHeapPriorityQueue: estensione dello heap con oggetti Locator per la modifica (update) e rimozione (remove) efficiente di elementi.

```python
from .heap_priority_queue import HeapPriorityQueue

class AdaptableHeapPriorityQueue(HeapPriorityQueue):
  """A locator-based priority queue implemented with a binary heap."""

  #------------------------------ nested Locator class
    ------------------------------
  class Locator(HeapPriorityQueue._Item):
    """Token for locating an entry of the priority queue."""
    __slots__ = '_index'                 # add index as additional field

    def __init__(self, k, v, j):
      super().__init__(k,v)
      self._index = j

  #------------------------------ nonpublic behaviors
    ------------------------------
  # override swap to record new indices
  def _swap(self, i, j):
    super()._swap(i,j)                   # perform the swap
    self._data[i]._index = i             # reset locator index (post-swap)
    self._data[j]._index = j             # reset locator index (post-swap)

  def _bubble(self, j):
    if j > 0 and self._data[j] < self._data[self._parent(j)]:
      self._upheap(j)
    else:
      self._downheap(j)
```

```python
27
28    #-------------------------- public behaviors
            --------------------------
29    def add(self, key, value):
30      """Add a key-value pair."""
31      token = self.Locator(key, value, len(self._data)) # initiaize locator
             index
32      self._data.append(token)
33      self._upheap(len(self._data) - 1)
34      return token
35
36    def update(self, loc, newkey, newval):
37      """Update the key and value for the entry identified by Locator loc."""
38      j = loc._index
39      if not (0 <= j < len(self) and self._data[j] is loc):
40        raise ValueError('Invalid locator')
41      loc._key = newkey
42      loc._value = newval
43      self._bubble(j)
44
45    def remove(self, loc):
46      """Remove and return the (k,v) pair identified by Locator loc."""
47      j = loc._index
48      if not (0 <= j < len(self) and self._data[j] is loc):
49        raise ValueError('Invalid locator')
50      if j == len(self) - 1:                   # item at last position
51        self._data.pop()                       # just remove it
52      else:
53        self._swap(j, len(self)-1)             # swap item to the last position
54        self._data.pop()                       # remove it from the list
55        self._bubble(j)                        # fix item displaced by the swap
56      return (loc._key, loc._value)
```

58

Capitolo 15

Graph

Listing 15.1: Implementazione della struttura dati Grafo tramite mappe di adiacenza, con classi interne Vertex/Edge e supporto a grafi orientati e non orientati.

```python
#------------------------- 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))

    def __str__(self):
      return str(self._element)

#------------------------- 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
```

```python
30
31     def endpoints(self):
32       """Return (u,v) tuple for vertices u and v."""
33       return (self._origin, self._destination)
34
35     def opposite(self, v):
36       """Return the vertex that is opposite v on this edge."""
37       if not isinstance(v, Graph.Vertex):
38         raise TypeError('v must be a Vertex')
39       if v is self._origin:
40         return self._destination
41       elif v is self._destination:
42           return self._origin
43       raise ValueError('v not incident to edge')
44
45     def element(self):
46       """Return element associated with this edge."""
47       return self._element
48
49     def __hash__(self):          # will allow edge to be a map/set key
50       return hash( (self._origin, self._destination) )
51
52     def __str__(self):
53       return
54           '({0},{1},{2})'.format(self._origin,self._destination,self._element)
54
55 class Graph:
56     """Representation of a simple graph using an adjacency map."""
57
58     def __init__(self, directed=False):
59         """Create an empty graph (undirected, by default).
60
61         Graph is directed if optional parameter is set to True.
62         """
63         self._outgoing = {}
64         # only create second map for directed graph; use alias for
                undirected
65         self._incoming = {} if directed else self._outgoing
66
67     def is_directed(self):
68         """Return True if this is a directed graph; False if undirected.
69
70         Property is based on the original declaration of the graph, not its
            contents.
71         """
72         return self._incoming is not self._outgoing # directed if maps are
            distinct
```

```python
 73
 74    def vertex_count(self):
 75        """Return the number of vertices in the graph."""
 76        return len(self._outgoing)
 77
 78    def vertices(self):
 79        """Return an iteration of all vertices of the graph."""
 80        return self._outgoing.keys()
 81
 82    def edge_count(self):
 83        """Return the number of edges in the graph."""
 84        total = sum(len(self._outgoing[v]) for v in self._outgoing)
 85        # for undirected graphs, make sure not to double-count edges
 86        return total if self.is_directed() else total // 2
 87
 88    def edges(self):
 89        """Return a set of all edges of the graph."""
 90        result = set()        # avoid double-reporting edges of undirected
                                    graph
 91        for secondary_map in self._outgoing.values():
 92            result.update(secondary_map.values()) # add edges to resulting
                    set
 93        return result
 94
 95    def _validate_vertex(self, v):
 96    """Verify that v is a Vertex of this graph."""
 97    if not isinstance(v, self.Vertex):
 98      raise TypeError('Vertex expected')
 99    if v not in self._outgoing:
100      raise ValueError('Vertex does not belong to this graph.')
101
102    def get_edge(self, u, v):
103        """Return the edge from u to v, or None if not adjacent."""
104        return self._outgoing[u].get(v) # returns None if v not adjacent
105
106
107    def degree(self, v, outgoing=True):
108        """Return number of (outgoing) edges incident to vertex v in the
            graph.
109
110        If graph is directed, optional parameter used to count incoming
            edges.
111        """
112        adj = self._outgoing if outgoing else self._incoming
113        return len(adj[v])
```

```python
    def incident_edges(self, v, outgoing=True):
        """Return all (outgoing) edges incident to vertex v in the graph.

        If graph is directed, optional parameter used to request incoming
            edges.
        """
        adj = self._outgoing if outgoing else self._incoming
        for edge in adj[v].values():
            yield edge

    def insert_vertex(self, x=None):
        """Insert and return a new Vertex with element x."""
        v = self.Vertex(x)
        self._outgoing[v] = {}
        if self.is_directed():
            self._incoming[v] = {}    # need distinct map for incoming edges
        return v

    def insert_edge(self, u, v, x=None):
        """Insert and return a new Edge from u to v with auxiliary element
            x."""
        e = self.Edge(u, v, x)
        self._outgoing[u][v] = e
        self._incoming[v][u] = e
```

```python
def remove_vertex(self, v):
    """Remove vertex v and all its incident edges from the graph."""
    # Phase 1: Remove references to v from the adjacency maps of its
        neighbors.

    # For every neighbor w connected by an outgoing edge (v -> w),
    # remove the link back to v from w's incoming map.
    for w in list(self._outgoing[v]):
        del self._incoming[w][v]

    # If the graph is directed, we also need to handle incoming edges (u ->
        v).
    # We must remove the link to v from u's outgoing map.
    # (In an undirected graph, _incoming is _outgoing, so the loop above
        covered this).
    if self.is_directed():
        for u in list(self._incoming[v]):
            del self._outgoing[u][v]

    # Phase 2: Remove v from the graph's internal dictionaries.
    del self._outgoing[v]
    if self.is_directed():
        del self._incoming[v]

def remove_edge(self, e):
    """Remove edge e from the graph."""
    u, v = e.endpoints()

    # Remove the edge from u's outgoing map
    del self._outgoing[u][v]

    # Remove the edge from v's incoming map
    del self._incoming[v][u]
```

Listing 15.2: Metodi di cancellazione per la classe Graph.

Capitolo 16

DFS

Listing 16.1: Algoritmo di ricerca in profondità (DFS): esplorazione ricorsiva del grafo, generazione della foresta di attraversamento e ricostruzione dei cammini.

```python
def DFS(g, u, discovered):
  """Perform DFS of the undiscovered portion of Graph g starting at Vertex
      u.

  discovered is a dictionary mapping each vertex to the edge that was used
      to
  discover it during the DFS. (u should be "discovered" prior to the call.)
  Newly discovered vertices will be added to the dictionary as a result.
  """
  for e in g.incident_edges(u):      # for every outgoing edge from u
    v = e.opposite(u)
    if v not in discovered:          # v is an unvisited vertex
      discovered[v] = e              # e is the tree edge that discovered v
      DFS(g, v, discovered)          # recursively explore from v

def DFS_complete(g):
  """Perform DFS for entire graph and return forest as a dictionary.

  Result maps each vertex v to the edge that was used to discover it.
  (Vertices that are roots of a DFS tree are mapped to None.)
  """
  forest = {}
  for u in g.vertices():
    if u not in forest:
      forest[u] = None               # u will be the root of a tree
      DFS(g, u, forest)
  return forest
```

```python
def construct_path(u, v, discovered):
    """
    Return a list of vertices comprising the directed path from u to v,
    or an empty list if v is not reachable from u.

    discovered is a dictionary resulting from a previous call to DFS started
        at u.
    """
    path = []                          # empty path by default
    if v in discovered:
        # we build list from v to u and then reverse it at the end
        path.append(v)
        walk = v
        while walk is not u:
            e = discovered[walk]       # find edge leading to walk
            parent = e.opposite(walk)
            path.append(parent)
            walk = parent
        path.reverse()                 # reorient path from u to v
    return path
```

Capitolo 17

BFS

Listing 17.1: Algoritmo di ricerca in ampiezza (BFS): esplorazione iterativa a livelli del grafo e generazione della foresta di attraversamento.

```python
def BFS(g, s, discovered):
  """Perform BFS of the undiscovered portion of Graph g starting at Vertex
      s.
  discovered is a dictionary mapping each vertex to the edge that was used
      to
  discover it during the BFS (s should be mapped to None prior to the call).
  Newly discovered vertices will be added to the dictionary as a result.
  """
  level = [s]                       # first level includes only s
  while len(level) > 0:
    next_level = []                 # prepare to gather newly found
        vertices
    for u in level:
      for e in g.incident_edges(u): # for every outgoing edge from u
        v = e.opposite(u)
        if v not in discovered:     # v is an unvisited vertex
          discovered[v] = e         # e is the tree edge that discovered v
          next_level.append(v)      # v will be further considered in next
              pass
    level = next_level              # relabel 'next' level to become
        current

def BFS_complete(g):
  """Perform BFS for entire graph and return forest as a dictionary.
  Result maps each vertex v to the edge that was used to discover it.
  (vertices that are roots of a BFS tree are mapped to None).
  """
  forest = {}
  for u in g.vertices():
    if u not in forest:
      forest[u] = None              # u will be a root of a tree
      BFS(g, u, forest)
  return forest
```

Capitolo 18

Transitive Closure

Listing 18.1: Calcolo della chiusura transitiva tramite l'algoritmo di Floyd-Warshall: approccio di programmazione dinamica per la raggiungibilità tra tutte le coppie di vertici.

```python
from copy import deepcopy

def floyd_warshall(g):
  """Return a new graph that is the transitive closure of g."""
  closure = deepcopy(g)                         # imported from copy module
  verts = list(closure.vertices())              # make indexable list
  n = len(verts)
  for k in range(n):
    for i in range(n):
      # verify that edge (i,k) exists in the partial closure
      if i != k and closure.get_edge(verts[i],verts[k]) is not None:
        for j in range(n):
          # verify that edge (k,j) exists in the partial closure
          if i != j != k and closure.get_edge(verts[k],verts[j]) is not
              None:
            # if (i,j) not yet included, add it to the closure
            if closure.get_edge(verts[i],verts[j]) is None:
              closure.insert_edge(verts[i],verts[j])
  return closure
```

Capitolo 19

Topological Sort

Listing 19.1: Ordinamento topologico di un grafo diretto aciclico (DAG): risoluzione sequenziale dei vincoli di precedenza basata sul grado entrante.

```python
def topological_sort(g):
  """Return a list of verticies of directed acyclic graph g in topological
     order.

  If graph g has a cycle, the result will be incomplete.
  """
  topo = []                  # a list of vertices placed in topological order
  ready = []                 # list of vertices that have no remaining
     constraints
  incount = {}               # keep track of in-degree for each vertex
  for u in g.vertices():
    incount[u] = g.degree(u, False)  # parameter requests incoming degree
    if incount[u] == 0:              # if u has no incoming edges,
      ready.append(u)                # it is free of constraints
  while len(ready) > 0:
    u = ready.pop()                  # u is free of constraints
    topo.append(u)                   # add u to the topological order
    for e in g.incident_edges(u):    # consider all outgoing neighbors of u
      v = e.opposite(u)
      incount[v] -= 1                # v has one less constraint without u
      if incount[v] == 0:
        ready.append(v)
  return topo
```

Capitolo 20

Shortest Path

Listing 20.1: Algoritmo di Dijkstra per i cammini minimi: calcolo delle distanze tramite rilassamento degli archi e ricostruzione dell'albero dei cammini minimi.

```python
from ..priority_queue.adaptable_heap_priority_queue import
    AdaptableHeapPriorityQueue

def shortest_path_lengths(g, src):
  """Compute shortest-path distances from src to reachable vertices of g.

  Graph g can be undirected or directed, but must be weighted such that
  e.element() returns a numeric weight for each edge e.

  Return dictionary mapping each reachable vertex to its distance from src.
  """
  d = {}                                        # d[v] is upper bound from
      s to v
  cloud = {}                                    # map reachable v to its
      d[v] value
  pq = AdaptableHeapPriorityQueue()             # vertex v will have key
      d[v]
  pqlocator = {}                                # map from vertex to its pq
      locator

  # for each vertex v of the graph, add an entry to the priority queue, with
  # the source having distance 0 and all others having infinite distance
  for v in g.vertices():
    if v is src:
      d[v] = 0
    else:
      d[v] = float('inf')                       # syntax for positive
          infinity
    pqlocator[v] = pq.add(d[v], v)              # save locator for future
        updates
```

```python
24
25   while not pq.is_empty():
26     key, u = pq.remove_min()
27     cloud[u] = key                          # its correct d[u] value
28     del pqlocator[u]                        # u is no longer in pq
29     for e in g.incident_edges(u):           # outgoing edges (u,v)
30       v = e.opposite(u)
31       if v not in cloud:
32         # perform relaxation step on edge (u,v)
33         wgt = e.element()
34         if d[u] + wgt < d[v]:               # better path to v?
35           d[v] = d[u] + wgt                 # update the distance
36           pq.update(pqlocator[v], d[v], v)  # update the pq entry
37
38   return cloud                              # only includes reachable
                                               vertices
39
40 def shortest_path_tree(g, s, d):
41   """Reconstruct shortest-path tree rooted at vertex s, given distance map
      d.
42
43   Return tree as a map from each reachable vertex v (other than s) to the
44   edge e=(u,v) that is used to reach v from its parent u in the tree.
45   """
46   tree = {}
47   for v in d:
48     if v is not s:
49       for e in g.incident_edges(v, False):    # consider INCOMING edges
50         u = e.opposite(v)
51         wgt = e.element()
52         if d[v] == d[u] + wgt:
53           tree[v] = e                         # edge e is used to reach v
54   return tree
```