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.

```
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
```

```
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.

```
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.

```
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
```

```
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.

```
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 ----------------------
```

```
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))
```

```
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
```

```
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.

```
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)
```

```
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.

```
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
```

```
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
```

```
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
          child.
119     """
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
```

```
133
134   def _delete(self, p):
135     """Delete the node at Position p, and replace it with its child, if any.
136
137     Return the element that had been stored at Position p.
138     Raise ValueError if Position p is invalid or p has two children.
139     """
140     node = self._validate(p)
141     if self.num_children(p) == 2:
142       raise ValueError('Position has two children')
143     child = node._left if node._left else node._right  # might be None
144     if child is not None:
145       child._parent = node._parent   # child's grandparent becomes parent
146     if node is self._root:
147       self._root = child             # child becomes root
148     else:
149       parent = node._parent
150       if node is parent._left:
151         parent._left = child
152       else:
153         parent._right = child
154     self._size -= 1
155     node._parent = node              # convention for deprecated node
156     return node._element
157
158   def _attach(self, p, t1, t2):
159     """Attach trees t1 and t2, respectively, as the left and right subtrees
          of the external Position p.
160
161     As a side effect, set t1 and t2 to empty.
162     Raise TypeError if trees t1 and t2 do not match type of this tree.
163     Raise ValueError if Position p is invalid or not external.
164     """
165     node = self._validate(p)
166     if not self.is_leaf(p):
167       raise ValueError('position must be leaf')
168     if not type(self) is type(t1) is type(t2):    # all 3 trees must be
          same type
169       raise TypeError('Tree types must match')
170     self._size += len(t1) + len(t2)
171     if not t1.is_empty():          # attached t1 as left subtree of node
172       t1._root._parent = node
173       node._left = t1._root
174       t1._root = None              # set t1 instance to empty
175       t1._size = 0
176     if not t2.is_empty():          # attached t2 as right subtree of node
177       t2._root._parent = node
178       node._right = t2._root
179       t2._root = None              # set t2 instance to empty
180       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.

```
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.

```
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
```

```
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
```

```
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()
```

```
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)
```

```
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
```

```
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)
```

```
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
```

```
270
271   def _rotate(self, p):
272     """Rotate Position p above its parent.
273
274     Switches between these configurations, depending on whether p==a or
          p==b.
275
276           b                     a
277          / \                   /  \
278         a   t2               t0   b
279        / \                        / \
280       t0  t1                     t1  t2
281
282     Caller should ensure that p is not the root.
283     """
284     """Rotate Position p above its parent."""
285     x = p._node
286     y = x._parent                     # we assume this exists
287     z = y._parent                     # grandparent (possibly None)
288     if z is None:
289       self._root = x                  # x becomes root
290       x._parent = None
291     else:
292       self._relink(z, x, y == z._left)      # x becomes a direct child of z
293     # now rotate x and y, including transfer of middle subtree
294     if x == y._left:
295       self._relink(y, x._right, True)       # x._right becomes left child
              of y
296       self._relink(x, y, False)             # y becomes right child of x
297     else:
298       self._relink(y, x._left, False)       # x._left becomes right child
              of y
299       self._relink(x, y, True)              # y becomes left child of x
```

```
300
301   def _restructure(self, x):
302     """Perform a trinode restructure among Position x, its parent, and its
          grandparent.
303
304     Return the Position that becomes root of the restructured subtree.
305
306     Assumes the nodes are in one of the following configurations:
307
308         z=a                 z=c           z=a               z=c
309        /  \                 /  \          /  \              /  \
310       t0  y=b             y=b  t3       t0   y=c          y=a  t3
311          /  \             /  \               /  \         /  \
312         t1  x=c          x=a  t2            x=b  t3      t0   x=b
313            /  \         /  \                /  \             /  \
314           t2  t3       t0  t1              t1  t2           t1  t2
315
316     The subtree will be restructured so that the node with key b becomes
          its root.
317
318               b
319             /   \
320           a       c
321          / \     / \
322         t0  t1  t2  t3
323
324     Caller should ensure that x has a grandparent.
325     """
326     """Perform trinode restructure of Position x with parent/grandparent."""
327     y = self.parent(x)
328     z = self.parent(y)
329     if (x == self.right(y)) == (y == self.right(z)):  # matching alignments
330       self._rotate(y)            # single rotation (of y)
331       return y                   # y is new subtree root
332     else:                                            # opposite alignments
333       self._rotate(x)            # double rotation (of x)
334       self._rotate(x)
335       return x                   # x is new subtree root
```

Capitolo 5

AVL Tree

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

```
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
```

```
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.

```
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
```

```
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))
```

```
68
69  def _fix_deficit(self, z, y):
70    """Resolve black deficit at z, where y is the root of z's heavier
          subtree."""
71    if not self._is_red(y): # y is black; will apply Case 1 or 2
72      x = self._get_red_child(y)
73      if x is not None: # Case 1: y is black and has red child x; do
            "transfer"
74        old_color = self._is_red(z)
75        middle = self._restructure(x)
76        self._set_color(middle, old_color)   # middle gets old color of z
77        self._set_black(self.left(middle))   # children become black
78        self._set_black(self.right(middle))
79      else: # Case 2: y is black, but no red children; recolor as "fusion"
80        self._set_red(y)
81        if self._is_red(z):
82          self._set_black(z)                 # this resolves the problem
83        elif not self.is_root(z):
84          self._fix_deficit(self.parent(z), self.sibling(z)) # recur upward
85    else: # Case 3: y is red; rotate misaligned 3-node and repeat
86      self._rotate(y)
87      self._set_black(y)
88      self._set_red(z)
89      if z == self.right(y):
90        self._fix_deficit(z, self.left(z))
91      else:
92        self._fix_deficit(z, self.right(z))
```