

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

1	Array Stack	3
2	Array Queue	5
3	Alberi	9
4	Binary Search Tree (BST)	19
5	AVL Tree	29
6	RB Tree	31
7	Hash	35

Capitolo 1

Array Stack

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

```
1 class Empty(Exception):
2     pass
3
4 class ArrayStack:
5     """Implementazione di ADT Stack che utilizza un oggetto list di Python
6         per la memorizzazione."""
7
8     def __init__(self):
9         """Crea uno stack vuoto."""
10        self._data = []                      # istanza di list non pubblica
11
12    def __len__():
13        """Restituisce il numero di elementi nello stack."""
14        return len(self._data)
15
16    def is_empty():
17        """Restituisce True se lo stack è vuoto."""
18        return len(self._data) == 0
19
20    def push(self, e):
21        """Aggiunge l'elemento e al top dello stack."""
22        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.

```
1 class Queue:
2     """Classe astratta che implementa l'ADT Queue."""
3
4     def __len__(self):
5         """Restituisce il numero di elementi nella coda."""
6         raise NotImplementedError('deve essere implementato dalla sottoclasse.')
7
8     def is_empty(self):
9         """Restituisce True se la coda è vuota."""
10        raise NotImplementedError('deve essere implementato dalla sottoclasse.')
11
12    def first(self):
13        """Restituisce (ma non rimuove) l'elemento al front della coda.
14           Raise Empty exception se la coda è vuota."""
15        raise NotImplementedError('deve essere implementato dalla sottoclasse.')
16
17    def dequeue(self):
18        """Rimuove e restituisce l'elemento al front della coda.
19           Raise Empty exception se la coda è vuota."""
20        raise NotImplementedError('deve essere implementato dalla sottoclasse.')
21
22    def enqueue(self, e):
23        """Aggiunge un elemento al back della coda."""
24        raise NotImplementedError('deve essere implementato dalla sottoclasse.') 
```

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

```
1 from .queue import Queue
2
3 class Empty(Exception):
4     pass
5
6 class ArrayQueue(Queue):
7     """Implementazione di ADT Queue basata sul tipo list di Python usato come
8         array circolare."""
9     DEFAULT_CAPACITY = 10           # dimensione di default di nuove code
10
11    def __init__(self):
12        """Crea una coda vuota."""
13        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
14        self._size = 0
15        self._front = 0
16
17    def __len__(self):
18        """Restituisce il numero di elementi nella coda."""
19        return self._size
20
21    def is_empty(self):
22        """Restituisce True se la coda è vuota."""
23        return self._size == 0
24
25    def first(self):
26        """Restituisce (ma non rimuove) l'elemento al front della coda.
27            Raise Empty exception se la coda è vuota.
28        """
29        if self.is_empty():
30            raise Empty('Queue is empty')
31        return self._data[self._front]
32
33    def dequeue(self):
34        """Rimuove e restituisce l'elemento al front della coda.
35            Raise Empty exception se la coda è vuota.
36        """
37        if self.is_empty():
38            raise Empty('Queue is empty')
39        answer = self._data[self._front]
40        self._data[self._front] = None          # favorisce garbage collection
41        self._front = (self._front + 1) % len(self._data)
42        self._size -= 1
43        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
48             dell'array se pieno
49             avail = (self._front + self._size) % len(self._data)
50             self._data[avail] = e
51             self._size += 1
52
53     def _resize(self, cap):          # we assume cap >= len(self)
54         """Ridimensiona l'array portandolo a lunghezza cap."""
55         old = self._data           # conserva la vecchia copia dell'array
56         self._data = [None] * cap   # alloca una nuova list di dimensione cap
57         j = self._front
58         for k in range(self._size):
59             self._data[k] = old[j]    # shifta gli indici per riallinearli
60             j = (j + 1) % len(old)   # usa la vecchia dimensione come modulo
61             self._front = 0           # front riallineato a 0

```


Capitolo 3

Alberi

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

```
1 from ..queue.array_queue import ArrayQueue
2 # import collections
3
4 class Tree:
5     """Abstract base class representing a tree structure."""
6
7     #----- start nested Position class -----
8     class Position:
9         """An abstraction representing the location of a single element within
10            a tree.
11
12            Note that two position instances may represent the same inherent
13            location in a tree.
14            Therefore, users should always rely on syntax 'p == q' rather than 'p
15            is q' when testing
16            equivalence of positions.
17        """
18
19
20     def element(self):
21         """Return the element stored at this Position."""
22         raise NotImplementedError('must be implemented by subclass')
23
24     def __eq__(self, other):
25         """Return True if other Position represents the same location."""
26         raise NotImplementedError('must be implemented by subclass')
27
28     def __ne__(self, other):
29         """Return True if other does not represent the same location."""
30         return not (self == other)           # opposite of __eq__
31
32     #----- end nested Position class -----
```

```

29
30     # ----- abstract methods that concrete subclass must support -----
31
32     def root(self):
33         """Return Position representing the tree's root (or None if empty)."""
34         raise NotImplementedError('must be implemented by subclass')
35
36     def parent(self, p):
37         """Return Position representing p's parent (or None if p is root)."""
38         raise NotImplementedError('must be implemented by subclass')
39
40     def num_children(self, p):
41         """Return the number of children that Position p has."""
42         raise NotImplementedError('must be implemented by subclass')
43
44     def children(self, p):
45         """Generate an iteration of Positions representing p's children."""
46         raise NotImplementedError('must be implemented by subclass')
47
48     def __len__(self):
49         """Return the total number of elements in the tree."""
50         raise NotImplementedError('must be implemented by subclass')
51
52     # ----- concrete methods implemented in this class -----
53
54     def is_root(self, p):
55         """Return True if Position p represents the root of the tree."""
56         return self.root() == p
57
58     def is_leaf(self, p):
59         """Return True if Position p does not have any children."""
60         return self.num_children(p) == 0
61
62     def is_empty(self):
63         """Return True if the tree is empty."""
64         return len(self) == 0
65
66     def depth(self, p):
67         """Return the number of levels separating Position p from the root."""
68         if self.is_root(p):
69             return 0
70         else:
71             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
87         if p is None:
88             p = self.root()
89         return self._height2(p)      # start _height2 recursion
90
91     def __iter__(self):
92         """Generate an iteration of the tree's elements."""
93         for p in self.positions(): # use same order as positions()
94             yield p.element()       # but yield each element
95
96     def positions(self):
97         """Generate an iteration of the tree's positions."""
98         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.

```
1 from .tree import Tree
2
3 class BinaryTree(Tree):
4     """Abstract base class representing a binary tree structure."""
5
6     # ----- additional abstract methods -----
7     def left(self, p):
8         """Return a Position representing p's left child.
9         Return None if p does not have a left child.
10        """
11        raise NotImplementedError('must be implemented by subclass')
12
13     def right(self, p):
14         """Return a Position representing p's right child.
15         Return None if p does not have a right child.
16        """
17        raise NotImplementedError('must be implemented by subclass')
18
19     # ----- concrete methods implemented in this class -----
20     def sibling(self, p):
21         """Return a Position representing p's sibling (or None if no
22             sibling)."""
23         parent = self.parent(p)
24         if parent is None:    # p must be the root
25             return None       # root has no sibling
26         else:
27             if p == self.left(parent):
28                 return self.right(parent)    # possibly None
29             else:
30                 return self.left(parent)    # possibly None
31
32     def children(self, p):
33         """Generate an iteration of Positions representing p's children."""
34         if self.left(p) is not None:
35             yield self.left(p)
36         if self.right(p) is not None:
37             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
47             its subtree
48             for other in self._subtree_inorder(self.left(p)):
49                 yield other
50             yield p                         # visit p between its subtrees
51             if self.right(p) is not None:      # if right child exists, traverse
52                 its subtree
53                 for other in self._subtree_inorder(self.right(p)):
54                     yield other
55
56     # override inherited version to make inorder the default
57     def positions(self):
58         """Generate an iteration of the tree's positions."""
59         return self.inorder()           # make inorder the default

```

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

```
1 from .binary_tree import BinaryTree
2
3 class LinkedBinaryTree(BinaryTree):
4     """Linked representation of a binary tree structure."""
5
6     #----- nested _Node class -----
7     class _Node:
8         """Lightweight, nonpublic class for storing a node."""
9         __slots__ = '_element', '_parent', '_left', '_right' # streamline
10            memory usage
11
12     def __init__(self, element, parent=None, left=None, right=None):
13         self._element = element
14         self._parent = parent
15         self._left = left
16         self._right = right
17
18     #----- nested Position class -----
19     class Position(BinaryTree.Position):
20         """An abstraction representing the location of a single element."""
21
22         def __init__(self, container, node):
23             """Constructor should not be invoked by user."""
24             self._container = container
25             self._node = node
26
27         def element(self):
28             """Return the element stored at this Position."""
29             return self._node._element
30
31         def __eq__(self, other):
32             """Return True if other is a Position representing the same
33                 location."""
34             return type(other) is type(self) and other._node is self._node
35
36     #----- utility methods -----
37     def _validate(self, p):
38         """Return associated node, if position is valid."""
39         if not isinstance(p, self.Position):
40             raise TypeError('p must be proper Position type')
41         if p._container is not self:
42             raise ValueError('p does not belong to this container')
43         if p._node._parent is p._node:      # convention for deprecated nodes
44             raise ValueError('p is no longer valid')
45         return p._node
46
47     def _make_position(self, node):
48         """Return Position instance for given node (or None if no node)."""
49         return self.Position(self, node) if node is not None else None
```

```

48
49     #----- binary tree constructor -----
50
51     def __init__(self):
52         """Create an initially empty binary tree."""
53         self._root = None
54         self._size = 0
55
56     #----- public accessors -----
57
58     def __len__(self):
59         """Return the total number of elements in the tree."""
60         return self._size
61
62     def root(self):
63         """Return the root Position of the tree (or None if tree is empty)."""
64         return self._make_position(self._root)
65
66     def parent(self, p):
67         """Return the Position of p's parent (or None if p is root)."""
68         node = self._validate(p)
69         return self._make_position(node._parent)
70
71     def left(self, p):
72         """Return the Position of p's left child (or None if no left child)."""
73         node = self._validate(p)
74         return self._make_position(node._left)
75
76     def right(self, p):
77         """Return the Position of p's right child (or None if no right
78             child)."""
79         node = self._validate(p)
80         return self._make_position(node._right)
81
82     def num_children(self, p):
83         """Return the number of children of Position p."""
84         node = self._validate(p)
85         count = 0
86
87         if node._left is not None:      # left child exists
88             count += 1
89         if node._right is not None:    # right child exists
90             count += 1
91
92         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
119     child.
120     """
121     node = self._validate(p)
122     if node._right is not None:
123         raise ValueError('Right child exists')
124     self._size += 1
125     node._right = self._Node(e, node)          # node is its parent
126     return self._make_position(node._right)
127
128 def _replace(self, p, e):
129     """Replace the element at position p with e, and return old element."""
130     node = self._validate(p)
131     old = node._element
132     node._element = e
133     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
141         node = self._validate(p)
142         if self.num_children(p) == 2:
143             raise ValueError('Position has two children')
144         child = node._left if node._left else node._right # might be None
145         if child is not None:
146             child._parent = node._parent # child's grandparent becomes parent
147         if node is self._root:
148             self._root = child # child becomes root
149         else:
150             parent = node._parent
151             if node is parent._left:
152                 parent._left = child
153             else:
154                 parent._right = child
155             self._size -= 1
156             node._parent = node # convention for deprecated node
157             return node._element
158
159     def _attach(self, p, t1, t2):
160         """Attach trees t1 and t2, respectively, as the left and right subtrees
161             of the external Position p.
162
163             As a side effect, set t1 and t2 to empty.
164             Raise TypeError if trees t1 and t2 do not match type of this tree.
165             Raise ValueError if Position p is invalid or not external.
166             """
167
168         node = self._validate(p)
169         if not self.is_leaf(p):
170             raise ValueError('position must be leaf')
171         if not type(self) is type(t1) is type(t2): # all 3 trees must be
172             same type
173             raise TypeError('Tree types must match')
174         self._size += len(t1) + len(t2)
175         if not t1.is_empty(): # attached t1 as left subtree of node
176             t1._root._parent = node
177             node._left = t1._root
178             t1._root = None # set t1 instance to empty
179             t1._size = 0
180         if not t2.is_empty(): # attached t2 as right subtree of node
181             t2._root._parent = node
182             node._right = t2._root
183             t2._root = None # set t2 instance to empty
184             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.

```
1 from collections.abc import MutableMapping
2
3 class MapBase(MutableMapping):
4     """Our own abstract base class that includes a nonpublic _Item class."""
5
6     #----- nested _Item class -----
7     class _Item:
8         """Lightweight composite to store key-value pairs as map items."""
9         __slots__ = '_key', '_value'
10
11    def __init__(self, k, v):
12        self._key = k
13        self._value = v
14
15    def __eq__(self, other):
16        return self._key == other._key      # compare items based on their keys
17
18    def __ne__(self, other):
19        return not (self == other)          # opposite of __eq__
20
21    def __lt__(self, other):
22        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.

```
1 from ..tree.linked_binary_tree import LinkedBinaryTree
2 from .map_base import MapBase
3
4 class TreeMap(LinkedBinaryTree, MapBase):
5     """Sorted map implementation using a binary search tree."""
6
7     #----- override Position class -----
8     class Position(LinkedBinaryTree.Position):
9         def key(self):
10             """Return key of map's key-value pair."""
11             return self.element()._key
12
13         def value(self):
14             """Return value of map's key-value pair."""
15             return self.element()._value
16
17     #----- nonpublic utilities -----
18     def _subtree_search(self, p, k):
19         """Return Position of p's subtree having key k, or last node
20             searched."""
21         if k == p.key():      # found match
22             return p
23         elif k < p.key():    # search left subtree
24             if self.left(p) is not None:
25                 return self._subtree_search(self.left(p), k)
26             else:               # search right subtree
27                 if self.right(p) is not None:
28                     return self._subtree_search(self.right(p), k)
29         return p              # unsuccessful search
30
31     def _subtree_first_position(self, p):
32         """Return Position of first item in subtree rooted at p."""
33         walk = p
34         while self.left(walk) is not None:      # keep walking left
35             walk = self.left(walk)
36         return walk
37
38     def _subtree_last_position(self, p):
39         """Return Position of last item in subtree rooted at p."""
40         walk = p
41         while self.right(walk) is not None:      # keep walking right
42             walk = self.right(walk)
43         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
48         None
49
50 def last(self):
51     """Return the last Position in the tree (or None if empty)."""
52     return self._subtree_last_position(self.root()) if len(self) > 0 else
53         None
54
55 def before(self, p):
56     """Return the Position just before p in the natural order.
57
58     Return None if p is the first position.
59     """
60     self._validate(p)           # inherited from LinkedBinaryTree
61     if self.left(p):
62         return self._subtree_last_position(self.left(p))
63     else:
64         # walk upward
65         walk = p
66         above = self.parent(walk)
67         while above is not None and walk == self.left(above):
68             walk = above
69             above = self.parent(walk)
70     return above
71
72 def after(self, p):
73     """Return the Position just after p in the natural order.
74
75     Return None if p is the last position.
76     """
77     self._validate(p)           # inherited from LinkedBinaryTree
78     if self.right(p):
79         return self._subtree_first_position(self.right(p))
80     else:
81         walk = p
82         above = self.parent(walk)
83         while above is not None and walk == self.right(above):
84             walk = 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
133                         LinkedBinaryTree
134             else:
135                 leaf = self._add_left(p, item)    # inherited from
136                         LinkedBinaryTree
137             self._rebalance_insert(leaf)          # hook for balanced tree
138                         subclasses
139
140
141 def __delitem__(self, k):
142     """Remove item associated with key k (raise KeyError if not found)."""
143     if not self.is_empty():
144         p = self._subtree_search(self.root(), k)
145         if k == p.key():
146             self.delete(p)                      # rely on positional version
147             return                            # successful deletion complete
148             self._rebalance_access(p)        # hook for balanced tree subclasses
149             raise KeyError('Key Error: ' + repr(k))
150
151
152 def __iter__(self):
153     """Generate an iteration of all keys in the map in order."""
154     p = self.first()
155     while p is not None:
156         yield p.key()
157         p = self.after(p)

```

```

153
154     #----- public methods for sorted map interface -----
155
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
210         if self.is_empty():
211             return None
212         else:
213             p = self.find_position(k)           # may not find exact match
214             if p.key() < k:                  # p's key is too small
215                 p = self.after(p)
216             return (p.key(), p.value()) if p is not None else None
217
218
219     def find_gt(self, k):
220         """Return (key,value) pair with least key strictly greater than k.
221
222         Return None if there does not exist such a key.
223         """
224
225         if self.is_empty():
226             return None
227         else:
228             p = self.find_position(k)
229             if not k < p.key():
230                 p = self.after(p)
231             return (p.key(), p.value()) if p is not None else None
232
233
234     def find_range(self, start, stop):
235         """Iterate all (key,value) pairs such that start <= key < stop.
236
237         If start is None, iteration begins with minimum key of map.
238         If stop is None, iteration continues through the maximum key of map.
239         """
240
241         if not self.is_empty():
242             if start is None:
243                 p = self.first()
244             else:
245                 # we initialize p with logic similar to find_ge
246                 p = self.find_position(start)
247                 if p.key() < start:
248                     p = self.after(p)
249             while p is not None and (stop is None or p.key() < stop):
250                 yield (p.key(), p.value())
251                 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
275         p==b.
276
277             b           a
278             / \         /   \
279             a   t2       t0   b
280             / \         /   \
281             t0   t1       t1   t2
282
283         Caller should ensure that p is not the root.
284         """
285
286         """Rotate Position p above its parent."""
287
288         x = p._node
289         y = x._parent          # we assume this exists
290         z = y._parent          # grandparent (possibly None)
291
292         if z is None:
293             self._root = x          # x becomes root
294             x._parent = None
295         else:
296             self._relink(z, x, y == z._left)    # x becomes a direct child of z
297             # now rotate x and y, including transfer of middle subtree
298             if x == y._left:
299                 self._relink(y, x._right, True)      # x._right becomes left child
295                 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
295                 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
303             grandparent.
304
305             Return the Position that becomes root of the restructured subtree.
306
307             Assumes the nodes are in one of the following configurations:
308
309                 z=a                  z=c                  z=a                  z=c
310                 / \                / \                / \                / \
311                 t0   y=b          y=b   t3          t0   y=c          y=a   t3
312                 / \                / \                / \                / \
313                 t1   x=c          x=a   t2          x=b   t3          t0   x=b
314                 / \                / \                / \                / \
315                 t2   t3          t0   t1          t1   t2          t1   t2
316
317             The subtree will be restructured so that the node with key b becomes
318                 its root.
319
320                 b
321                 / \
322                 a       c
323                 / \     / \
324                 t0   t1   t2   t3
325
326             Caller should ensure that x has a grandparent.
327
328             """
329             """Perform trinode restructure of Position x with parent/grandparent."""
330
331             y = self.parent(x)
332             z = self.parent(y)
333             if (x == self.right(y)) == (y == self.right(z)): # matching alignments
334                 self._rotate(y)                         # single rotation (of y)
335                 return y                             # y is new subtree root
336             else:                                     # opposite alignments
337                 self._rotate(x)                      # double rotation (of x)
338                 self._rotate(x)
339                 return x                           # x is new subtree root

```

Capitolo 5

AVL Tree

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

```
1 from .binary_search_tree import TreeMap
2
3 class AVLTreeMap(TreeMap):
4     """Sorted map implementation using an AVL tree."""
5
6     #----- nested _Node class -----
7     class _Node(TreeMap._Node):
8         """Node class for AVL maintains height value for balancing.
9
10        We use convention that a "None" child has height 0, thus a leaf has
11        height 1.
12        """
13
14        __slots__ = '_height'           # additional data member to store height
15
16        def __init__(self, element, parent=None, left=None, right=None):
17            super().__init__(element, parent, left, right)
18            self._height = 0               # will be recomputed during balancing
19
20        def left_height(self):
21            return self._left._height if self._left is not None else 0
22
23        def right_height(self):
24            return self._right._height if self._right is not None else 0
25
26    #----- positional-based utility methods
27
28    def _recompute_height(self, p):
29        p._node._height = 1 + max(p._node.left_height(), p._node.right_height())
30
31    def _isbalanced(self, p):
32        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) >
32             p._node.right_height():
33             return self.left(p)
34         else:
35             return self.right(p)
36
37     def _tall_grandchild(self, p):
38         child = self._tall_child(p)
39         # if child is on left, favor left grandchild; else favor right
40         # grandchild
41         alignment = (child == self.left(p))
42         return self._tall_child(child, alignment)
43
44     def _rebalance(self, p):
45         while p is not None:
46             old_height = p._node._height                         # trivially 0
47             if new node
48             if not self._isbalanced(p):                      # imbalance
49                 detected!
50                 # perform trinode restructuring, setting p to resulting root,
51                 # and recompute new local heights after the restructuring
52                 p = self._restructure(self._tall_grandchild(p))
53                 self._recompute_height(self.left(p))
54                 self._recompute_height(self.right(p))
55                 self._recompute_height(p)                      # adjust for
56                 recent changes
57                 if p._node._height == old_height:            # has height
58                     changed?
59                     p = None                                # no further
60                     changes needed
61                 else:
62                     p = self.parent(p)                        # repeat with
63                     parent
64
65         #----- override balancing hooks
66         -----
67     def _rebalance_insert(self, p):
68         self._rebalance(p)
69
70     def _rebalance_delete(self, p):
71         self._rebalance(p)

```

Capitolo 6

RB Tree

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

```
1 from .binary_search_tree import TreeMap
2
3 class RedBlackTreeMap(TreeMap):
4     """Sorted map implementation using a red-black tree."""
5
6     #----- nested _Node class -----
7     class _Node(TreeMap._Node):
8         """Node class for red-black tree maintains bit that denotes color."""
9         __slots__ = '_red'      # add additional data member to the Node class
10
11     def __init__(self, element, parent=None, left=None, right=None):
12         super().__init__(element, parent, left, right)
13         self._red = True        # new node red by default
14
15     #----- positional-based utility methods
16
17     # we consider a nonexistent child to be trivially black
18     def _set_red(self, p): p._node._red = True
19     def _set_black(self, p): p._node._red = False
20     def _set_color(self, p, make_red): p._node._red = make_red
21     def _is_red(self, p): return p is not None and p._node._red
22     def _is_red_leaf(self, p): return self._is_red(p) and self.is_leaf(p)
23
24     def _get_red_child(self, p):
25         """Return a red child of p (or None if no such child)."""
26         for child in (self.left(p), self.right(p)):
27             if self._is_red(child):
28                 return child
29         return None
```

```

29
30     #----- support for insertions -----
31
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
57                                         black
58         elif p is not None:
59             n = self.num_children(p)
60             if n == 1:                      # deficit exists unless child is a red
61                                         leaf
62                 c = next(self.children(p))
63                 if not self._is_red_leaf(c):
64                     self._fix_deficit(p, c)
65             elif n == 2:                    # removed black node with red child
66                 if self._is_red_leaf(self.left(p)):
67                     self._set_black(self.left(p))
68                 else:
69                     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
71     subtree."""
72     if not self._is_red(y): # y is black; will apply Case 1 or 2
73         x = self._get_red_child(y)
74         if x is not None: # Case 1: y is black and has red child x; do
75             "transfer"
76             old_color = self._is_red(z)
77             middle = self._restructure(x)
78             self._set_color(middle, old_color) # middle gets old color of z
79             self._set_black(self.left(middle)) # children become black
80             self._set_black(self.right(middle))
81         else: # Case 2: y is black, but no red children; recolor as "fusion"
82             self._set_red(y)
83             if self._is_red(z):
84                 self._set_black(z) # this resolves the problem
85             elif not self.is_root(z):
86                 self._fix_deficit(self.parent(z), self.sibling(z)) # recur upward
87             else: # Case 3: y is red; rotate misaligned 3-node and repeat
88                 self._rotate(y)
89                 self._set_black(y)
90                 self._set_red(z)
91                 if z == self.right(y):
92                     self._fix_deficit(z, self.left(z))
93                 else:
94                     self._fix_deficit(z, self.right(z))

```


Capitolo 7

Hash

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

```
1 from .map_base import MapBase
2 from random import randrange          # used to pick MAD parameters
3
4 class HashMapBase(MapBase):
5     """Abstract base class for map using hash-table with MAD compression.
6
7     Keys must be hashable and non-None.
8     """
9
10    def __init__(self, cap=11, p=109345121):
11        """Create an empty hash-table map.
12
13        cap      initial table size (default 11)
14        p       positive prime used for MAD (default 109345121)
15        """
16        self._table = cap * [None]           # number of entries in
17        self._n = 0                         # the map
18        self._prime = p                    # prime for MAD
19        self._scale = 1 + randrange(p-1)   # scale from 1 to p-1 for
20        self._shift = randrange(p)         # shift from 0 to p-1 for
21        MAD
22
23    def _hash_function(self, k):
24        return (hash(k)*self._scale + self._shift) % self._prime %
25        len(self._table)
26
27    def __len__(self):
28        return self._n
```

```

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

```