

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
8	Sorted Unsorted Table	37
9	Chain Hash	43

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__(self):
13        """Restituisce il numero di elementi nello stack."""
14        return len(self._data)
15
16    def is_empty(self):
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
10    DEFAULT_CAPACITY = 10          # dimensione di default di nuove code
11
12    def __init__(self):
13        """Crea una coda vuota."""
14        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
15        self._size = 0
16        self._front = 0
17
18    def __len__(self):
19        """Restituisce il numero di elementi nella coda."""
20        return self._size
21
22    def is_empty(self):
23        """Restituisce True se la coda è vuota."""
24        return self._size == 0
25
26    def first(self):
27        """Restituisce (ma non rimuove) l'elemento al front della coda.
28            Raise Empty exception se la coda è vuota.
29        """
30        if self.is_empty():
31            raise Empty('Queue is empty')
32        return self._data[self._front]
33
34    def dequeue(self):
35        """Rimuove e restituisce l'elemento al front della coda.
36            Raise Empty exception se la coda è vuota.
37        """
38        if self.is_empty():
39            raise Empty('Queue is empty')
40        answer = self._data[self._front]
41        self._data[self._front] = None          # favorisce garbage collection
42        self._front = (self._front + 1) % len(self._data)
43        self._size -= 1
44        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         def element(self):
20             """Return the element stored at this Position."""
21             raise NotImplementedError('must be implemented by subclass')
22
23         def __eq__(self, other):
24             """Return True if other Position represents the same location."""
25             raise NotImplementedError('must be implemented by subclass')
26
27         def __ne__(self, other):
28             """Return True if other does not represent the same location."""
29             return not (self == other)          # opposite of __eq__
30
31     #----- 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
120     p."""
121     for c in self.children(p): # for each child c
122         for other in self._subtree_postorder(c): # do postorder of c's
123             subtree
124             yield other # yielding each to our caller
125     yield p # visit p after its subtrees
126
127 def breadthfirst(self):
128     """Generate a breadth-first iteration of the positions of the tree."""
129     if not self.is_empty():
130         fringe = ArrayQueue() # known positions not yet yielded
131         fringe.enqueue(self.root()) # starting with the root
132         while not fringe.is_empty():
133             p = fringe.dequeue() # remove from front of the queue
134             yield p # report this position
135             for c in self.children(p):
136                 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
            memory usage
10
11     def __init__(self, element, parent=None, left=None, right=None):
12         self._element = element
13         self._parent = parent
14         self._left = left
15         self._right = right
16
17     #----- nested Position class -----
18     class Position(BinaryTree.Position):
19         """An abstraction representing the location of a single element."""
20
21     def __init__(self, container, node):
22         """Constructor should not be invoked by user."""
23         self._container = container
24         self._node = node
25
26     def element(self):
27         """Return the element stored at this Position."""
28         return self._node._element
29
30     def __eq__(self, other):
31         """Return True if other is a Position representing the same
            location."""
32         return type(other) is type(self) and other._node is self._node
33
34     #----- utility methods -----
35     def _validate(self, p):
36         """Return associated node, if position is valid."""
37         if not isinstance(p, self.Position):
38             raise TypeError('p must be proper Position type')
39         if p._container is not self:
40             raise ValueError('p does not belong to this container')
41         if p._node._parent is p._node: # convention for deprecated nodes
42             raise ValueError('p is no longer valid')
43         return p._node
44
45     def _make_position(self, node):
46         """Return Position instance for given node (or None if no node)."""
47         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
76         child)."""
77     node = self._validate(p)
78     return self._make_position(node._right)
79
80 def num_children(self, p):
81     """Return the number of children of Position p."""
82     node = self._validate(p)
83     count = 0
84     if node._left is not None:    # left child exists
85         count += 1
86     if node._right is not None:  # right child exists
87         count += 1
88     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     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
160     of the external Position p.
161
162     As a side effect, set t1 and t2 to empty.
163     Raise TypeError if trees t1 and t2 do not match type of this tree.
164     Raise ValueError if Position p is invalid or not external.
165     """
166     node = self._validate(p)
167     if not self.is_leaf(p):
168         raise ValueError('position must be leaf')
169     if not type(self) is type(t1) is type(t2): # all 3 trees must be
170         same type
171         raise TypeError('Tree types must match')
172     self._size += len(t1) + len(t2)
173     if not t1.is_empty(): # attached t1 as left subtree of node
174         t1._root._parent = node
175         node._left = t1._root
176         t1._root = None # set t1 instance to empty
177         t1._size = 0
178     if not t2.is_empty(): # attached t2 as right subtree of node
179         t2._root._parent = node
180         node._right = t2._root
181         t2._root = None # set t2 instance to empty
182         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
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
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 def __delitem__(self, k):
141     """Remove item associated with key k (raise KeyError if not found)."""
142     if not self.is_empty():
143         p = self._subtree_search(self.root(), k)
144         if k == p.key():
145             self.delete(p)                          # rely on positional version
146             return                                  # successful deletion complete
147         self._rebalance_access(p)                    # hook for balanced tree subclasses
148         raise KeyError('Key Error: ' + repr(k))
149
150 def __iter__(self):
151     """Generate an iteration of all keys in the map in order."""
152     p = self.first()
153     while p is not None:
154         yield p.key()
155         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
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     """Rotate Position p above its parent."""
286     x = p._node
287     y = x._parent           # we assume this exists
288     z = y._parent           # grandparent (possibly None)
289     if z is None:
290         self._root = x      # x becomes root
291         x._parent = None
292     else:
293         self._relink(z, x, y == z._left)    # x becomes a direct child of z
294         # now rotate x and y, including transfer of middle subtree
295         if x == y._left:
296             self._relink(y, x._right, True)    # x._right becomes left child
297             # of y
298             self._relink(x, y, False)           # y becomes right child of x
299         else:
300             self._relink(y, x._left, False)    # x._left becomes right child
301             # of y
302             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     """Perform trinode restructure of Position x with parent/grandparent."""
329     y = self.parent(x)
330     z = self.parent(y)
331     if (x == self.right(y)) == (y == self.right(z)): # matching alignments
332         self._rotate(y) # single rotation (of y)
333         return y # y is new subtree root
334     else: # opposite alignments
335         self._rotate(x) # double rotation (of x)
336         self._rotate(x)
337         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        __slots__ = '_height'          # additional data member to store height
14
15        def __init__(self, element, parent=None, left=None, right=None):
16            super().__init__(element, parent, left, right)
17            self._height = 0            # will be recomputed during balancing
18
19        def left_height(self):
20            return self._left._height if self._left is not None else 0
21
22        def right_height(self):
23            return self._right._height if self._right is not None else 0
24
25        #----- positional-based utility methods -----
26
27        def _recompute_height(self, p):
28            p._node._height = 1 + max(p._node.left_height(), p._node.right_height())
29
30        def _isbalanced(self, p):
31            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.

```
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 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
       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))

```


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 ]
17        self._n = 0                # number of entries in
18                                   the map
19        self._prime = p            # prime for MAD
20                                   compression
21        self._scale = 1 + randrange(p-1)    # scale from 1 to p-1 for
22                                           MAD
23        self._shift = randrange(p)         # shift from 0 to p-1 for
24                                           MAD
25
26    def _hash_function(self, k):
27        return (hash(k)*self._scale + self._shift) % self._prime %
28                len(self._table)
29
30    def __len__(self):
31        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
        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).

```
1 from .map_base import MapBase
2
3 class SortedTableMap(MapBase):
4     """Map implementation using a sorted table."""
5
6     #----- nonpublic behaviors -----
7
8     def _find_index(self, k, low, high):
9         """Return index of the leftmost item with key greater than or equal to
10            k.
11
12            Return high + 1 if no such item qualifies.
13
14            That is, j will be returned such that:
15            all items of slice table[low:j] have key < k
16            all items of slice table[j:high+1] have key >= k
17        """
18        if high < low:
19            return high + 1 # no element qualifies
20        else:
21            mid = (low + high) // 2
22            if k == self._table[mid]._key:
23                return mid # found exact match
24            elif k < self._table[mid]._key:
25                return self._find_index(k, low, mid - 1) # Note: may return mid
26            else:
27                return self._find_index(k, mid + 1, high) # answer is right of mid
```

```

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

```

```

75 def find_max(self):
76     """Return (key,value) pair with maximum key (or None if empty)."""
77     if len(self._table) > 0:
78         return (self._table[-1]._key, self._table[-1]._value)
79     else:
80         return None
81
82 def find_le(self, k):
83     """Return (key,value) pair with greatest key less than or equal to k.
84
85     Return None if there does not exist such a key.
86     """
87     j = self._find_index(k, 0, len(self._table) - 1)      # j's key >= k
88     if j < len(self._table) and self._table[j]._key == k:
89         return (self._table[j]._key, self._table[j]._value)    # exact match
90     elif j > 0:
91         return (self._table[j-1]._key, self._table[j-1]._value) # Note use
92                             of j-1
93     else:
94         return None
95
96 def find_ge(self, k):
97     """Return (key,value) pair with least key greater than or equal to k.
98
99     Return None if there does not exist such a key.
100    """
101    j = self._find_index(k, 0, len(self._table) - 1)      # j's key >= k
102    if j < len(self._table):
103        return (self._table[j]._key, self._table[j]._value)
104    else:
105        return None
106
107 def find_lt(self, k):
108     """Return (key,value) pair with greatest key strictly less than k.
109
110     Return None if there does not exist such a key.
111    """
112    j = self._find_index(k, 0, len(self._table) - 1)      # j's key >= k
113    if j > 0:
114        return (self._table[j-1]._key, self._table[j-1]._value) # Note use
115                             of j-1
116    else:
117        return None

```

```

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
140         result
141     while j < len(self._table) and (stop is None or self._table[j]._key <
142         stop):
143         yield (self._table[j]._key, self._table[j]._value)
144         j += 1

```


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

```

1 from .map_base import MapBase
2
3 class UnsortedTableMap(MapBase):
4     """Map implementation using an unordered list."""
5
6     def __init__(self):
7         """Create an empty map."""
8         self._table = []                # list of _Item's
9
10    def __getitem__(self, k):
11        """Return value associated with key k (raise KeyError if not found)."""
12        for item in self._table:
13            if k == item._key:
14                return item._value
15        raise KeyError('Key Error: ' + repr(k))
16
17    def __setitem__(self, k, v):
18        """Assign value v to key k, overwriting existing value if present."""
19        for item in self._table:
20            if k == item._key:                # Found a match:
21                item._value = v                # reassign value
22            return                            # and quit
23        # did not find match for key
24        self._table.append(self._Item(k,v))
25
26    def __delitem__(self, k):
27        """Remove item associated with key k (raise KeyError if not found)."""
28        for j in range(len(self._table)):
29            if k == self._table[j]._key:        # Found a match:
30                self._table.pop(j)            # remove item
31            return                            # and quit
32        raise KeyError('Key Error: ' + repr(k))
33
34    def __len__(self):
35        """Return number of items in the map."""
36        return len(self._table)
37
38    def __iter__(self):
39        """Generate iteration of the map's keys."""
40        for item in self._table:
41            yield item._key                    # yield the KEY

```


Capitolo 9

Chain Hash

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

```
1 from .hash_map_base import HashMapBase
2 from .unsorted_table_map import UnsortedTableMap
3
4 class ChainHashMap(HashMapBase):
5     """Hash map implemented with separate chaining for collision
6         resolution."""
7
8     def _bucket_getitem(self, j, k):
9         bucket = self._table[j]
10        if bucket is None:
11            raise KeyError('Key Error: ' + repr(k))    # no match found
12        return bucket[k]                                # may raise KeyError
13
14    def _bucket_setitem(self, j, k, v):
15        if self._table[j] is None:
16            self._table[j] = UnsortedTableMap()        # bucket is new to the table
17            oldsize = len(self._table[j])
18            self._table[j][k] = v
19            if len(self._table[j]) > oldsize:            # key was new to the table
20                self._n += 1                            # increase overall map size
21
22    def _bucket_delitem(self, j, k):
23        bucket = self._table[j]
24        if bucket is None:
25            raise KeyError('Key Error: ' + repr(k))    # no match found
26        del bucket[k]                                    # may raise KeyError
27
28    def __iter__(self):
29        for bucket in self._table:
30            if bucket is not None:                        # a nonempty slot
31                for key in bucket:
32                    yield key
```