UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA
E MATEMATICA APPLICATA

Corso:
DESIGN AND ANALYSIS OF ALGORITHMS

APPUNTI

Carmine Terracciano

Mat. IE22700109

ANNO ACCADEMICO 2025/2026

Sommario

Capitolo 1

Array Stack

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

```python
class Empty(Exception):
  pass

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

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

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

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

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

```python

def top(self):
  """Restituisce (ma non rimuove) l'elemento al top dello stack.
     Raise Empty exception se lo stack è vuoto."""
  if self.is_empty():
    raise Empty('lo stack è vuoto')
    # print("lo stack è vuoto")
  return self._data[-1]           # legge l'ultimo elemento della list

def pop(self):
  """Rimuove e restituisce l'elemento al top dello stack.
     Raise Empty exception se lo stack è vuoto."""
  if self.is_empty():
    raise Empty('lo stack è vuoto')
    # print("lo stack è vuoto")
  return self._data.pop()     # rimuove l'ultimo elemento della list
```

Capitolo 2

Array Queue

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

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

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

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

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

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

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

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

```python
from .queue import Queue

class Empty(Exception):
  pass

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

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

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

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

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

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

```python

  def enqueue(self, e):
    """Aggiunge un elemento al back della coda."""
    if self._size == len(self._data):
      self._resize(2 * len(self._data))      # raddoppia la dimensione dell'array se pieno
    avail = (self._front + self._size) % len(self._data)
    self._data[avail] = e
    self._size += 1

  def _resize(self, cap):        # we assume cap >= len(self)
    """Ridimensiona l'array portandolo a lunghezza cap."""
    old = self._data              # conserva la vecchia copia dell'array
    self._data = [None] * cap     # alloca una nuova list di dimensione cap
    j = self._front
    for k in range(self._size):
      self._data[k] = old[j]      # shifta gli indici per riallinearli
      j = (j + 1) % len(old)      # usa la vecchia dimensione come modulo
    self._front = 0               # front riallineato a 0
```

Capitolo 3

Alberi

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

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

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

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

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

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

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

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

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

```python

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

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

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

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

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

    # ---------- concrete methods implemented in this class ----------
    def is_root(self, p):
        """Return True if Position p represents the root of the tree."""
        return self.root() == p

    def is_leaf(self, p):
        """Return True if Position p does not have any children."""
        return self.num_children(p) == 0

    def is_empty(self):
        """Return True if the tree is empty."""
        return len(self) == 0

    def depth(self, p):
        """Return the number of levels separating Position p from the root."""
        if self.is_root(p):
            return 0
        else:
            return 1 + self.depth(self.parent(p))
```

```
def _height1(self):              # works, but O(n^2) worst-case time
  """Return the height of the tree."""
  return max(self.depth(p) for p in self.positions() if self.is_leaf(p))

def _height2(self, p):           # time is linear in size of subtree
  """Return the height of the subtree rooted at Position p."""
  if self.is_leaf(p):
    return 0
  else:
    return 1 + max(self._height2(c) for c in self.children(p))

def height(self, p=None):
  """Return the height of the subtree rooted at Position p.
  If p is None, return the height of the entire tree.
  """
  if p is None:
    p = self.root()
  return self._height2(p)        # start _height2 recursion

def __iter__(self):
  """Generate an iteration of the tree's elements."""
  for p in self.positions():     # use same order as positions()
    yield p.element()            # but yield each element

def positions(self):
  """Generate an iteration of the tree's positions."""
  return self.preorder()         # return entire preorder iteration
```

```python
def preorder(self):
  """Generate a preorder iteration of positions in the tree."""
  if not self.is_empty():
    for p in self._subtree_preorder(self.root()):  # start recursion
      yield p

def _subtree_preorder(self, p):
  """Generate a preorder iteration of positions in subtree rooted at p."""
  yield p          # visit p before its subtrees
  for c in self.children(p):           # for each child c
    for other in self._subtree_preorder(c):   # do preorder of c's subtree
      yield other          # yielding each to our caller

def postorder(self):
  """Generate a postorder iteration of positions in the tree."""
  if not self.is_empty():
    for p in self._subtree_postorder(self.root()):  # start recursion
      yield p

def _subtree_postorder(self, p):
  """Generate a postorder iteration of positions in subtree rooted at p."""
  for c in self.children(p):      # for each child c
    for other in self._subtree_postorder(c):  # do postorder of c's subtree
      yield other          # yielding each to our caller
  yield p          # visit p after its subtrees

def breadthfirst(self):
  """Generate a breadth-first iteration of the positions of the tree."""
  if not self.is_empty():
    fringe = ArrayQueue()              # known positions not yet yielded
    fringe.enqueue(self.root())        # starting with the root
    while not fringe.is_empty():
      p = fringe.dequeue()             # remove from front of the queue
      yield p                          # report this position
      for c in self.children(p):
        fringe.enqueue(c)              # add children to back of queue
```

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

```
from .tree import Tree

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

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

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

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

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

```python

def inorder(self):
  """Generate an inorder iteration of positions in the tree."""
  if not self.is_empty():
    for p in self._subtree_inorder(self.root()):
      yield p

def _subtree_inorder(self, p):
  """Generate an inorder iteration of positions in subtree rooted at p."""
  if self.left(p) is not None:          # if left child exists, traverse its subtree
    for other in self._subtree_inorder(self.left(p)):
      yield other
  yield p                               # visit p between its subtrees
  if self.right(p) is not None:         # if right child exists, traverse its subtree
    for other in self._subtree_inorder(self.right(p)):
      yield other

# override inherited version to make inorder the default
def positions(self):
  """Generate an iteration of the tree's positions."""
  return self.inorder()                 # make inorder the default
```

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

```python
from .binary_tree import BinaryTree

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

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

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

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

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

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

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

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

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

```python

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

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

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

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

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

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

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

```
  #-------------------------- nonpublic mutators --------------------------
  def _add_root(self, e):
    """Place element e at the root of an empty tree and return new Position.

    Raise ValueError if tree nonempty.
    """
    if self._root is not None:
      raise ValueError('Root exists')
    self._size = 1
    self._root = self._Node(e)
    return self._make_position(self._root)

  def _add_left(self, p, e):
    """Create a new left child for Position p, storing element e.

    Return the Position of new node.
    Raise ValueError if Position p is invalid or p already has a left child.
    """
    node = self._validate(p)
    if node._left is not None:
      raise ValueError('Left child exists')
    self._size += 1
    node._left = self._Node(e, node)            # node is its parent
    return self._make_position(node._left)

  def _add_right(self, p, e):
    """Create a new right child for Position p, storing element e.

    Return the Position of new node.
    Raise ValueError if Position p is invalid or p already has a right child.
    """
    node = self._validate(p)
    if node._right is not None:
      raise ValueError('Right child exists')
    self._size += 1
    node._right = self._Node(e, node)           # node is its parent
    return self._make_position(node._right)

  def _replace(self, p, e):
    """Replace the element at position p with e, and return old element."""
    node = self._validate(p)
    old = node._element
    node._element = e
    return old
```

```python

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

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

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

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

Capitolo 4

Binary Search Tree (BST)

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

```python
from collections.abc import MutableMapping

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

#--------- public methods providing "positional" support ----------------
def first(self):
  """Return the first Position in the tree (or None if empty)."""
  return self._subtree_first_position(self.root()) if len(self) > 0 else None

def last(self):
  """Return the last Position in the tree (or None if empty)."""
  return self._subtree_last_position(self.root()) if len(self) > 0 else None

def before(self, p):
  """Return the Position just before p in the natural order.

  Return None if p is the first position.
  """
  self._validate(p)           # inherited from LinkedBinaryTree
  if self.left(p):
    return self._subtree_last_position(self.left(p))
  else:
    # walk upward
    walk = p
    above = self.parent(walk)
    while above is not None and walk == self.left(above):
      walk = above
      above = self.parent(walk)
    return above

def after(self, p):
  """Return the Position just after p in the natural order.

  Return None if p is the last position.
  """
  self._validate(p)           # inherited from LinkedBinaryTree
  if self.right(p):
    return self._subtree_first_position(self.right(p))
  else:
    walk = p
    above = self.parent(walk)
    while above is not None and walk == self.right(above):
      walk = above
      above = self.parent(walk)
    return above
```

```python

def find_position(self, k):
  """Return position with key k, or else neighbor (or None if empty)."""
  if self.is_empty():
    return None
  else:
    p = self._subtree_search(self.root(), k)
    self._rebalance_access(p)     # hook for balanced tree subclasses
    return p

def delete(self, p):
  """Remove the item at given Position."""
  self._validate(p)                   # inherited from LinkedBinaryTree
  if self.left(p) and self.right(p):      # p has two children
    replacement = self._subtree_last_position(self.left(p))
    self._replace(p, replacement.element())    # from LinkedBinaryTree
    p =  replacement
  # now p has at most one child
  parent = self.parent(p)
  self._delete(p)                     # inherited from LinkedBinaryTree
  self._rebalance_delete(parent)      # if root deleted, parent is None

#--------------- public methods for (standard) map interface ------------
def __getitem__(self, k):
  """Return value associated with key k (raise KeyError if not found)."""
  if self.is_empty():
    raise KeyError('Key Error: ' + repr(k))
  else:
    p = self._subtree_search(self.root(), k)
    self._rebalance_access(p)         # hook for balanced tree subclasses
    if k != p.key():
      raise KeyError('Key Error: ' + repr(k))
    return p.value()
```

```

  def __setitem__(self, k, v):
    """Assign value v to key k, overwriting existing value if present."""
    if self.is_empty():
      leaf = self._add_root(self._Item(k,v))     # from LinkedBinaryTree
    else:
      p = self._subtree_search(self.root(), k)
      if p.key() == k:
        p.element()._value = v          # replace existing item's value
        self._rebalance_access(p)       # hook for balanced tree subclasses
        return
      else:
        item = self._Item(k,v)
        if p.key() < k:
          leaf = self._add_right(p, item)   # inherited from LinkedBinaryTree
        else:
          leaf = self._add_left(p, item)    # inherited from LinkedBinaryTree
    self._rebalance_insert(leaf)            # hook for balanced tree subclasses

  def __delitem__(self, k):
    """Remove item associated with key k (raise KeyError if not found)."""
    if not self.is_empty():
      p = self._subtree_search(self.root(), k)
      if k == p.key():
        self.delete(p)                # rely on positional version
        return                        # successful deletion complete
      self._rebalance_access(p)       # hook for balanced tree subclasses
    raise KeyError('Key Error: ' + repr(k))

  def __iter__(self):
    """Generate an iteration of all keys in the map in order."""
    p = self.first()
    while p is not None:
      yield p.key()
      p = self.after(p)
```

```python

#----------------- public methods for sorted map interface --------------
def __reversed__(self):
  """Generate an iteration of all keys in the map in reverse order."""
  p = self.last()
  while p is not None:
    yield p.key()
    p = self.before(p)

def find_min(self):
  """Return (key,value) pair with minimum key (or None if empty)."""
  if self.is_empty():
    return None
  else:
    p = self.first()
    return (p.key(), p.value())

def find_max(self):
  """Return (key,value) pair with maximum key (or None if empty)."""
  if self.is_empty():
    return None
  else:
    p = self.last()
    return (p.key(), p.value())

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

  Return None if there does not exist such a key.
  """
  if self.is_empty():
    return None
  else:
    p = self.find_position(k)
    if k < p.key():
      p = self.before(p)
    return (p.key(), p.value()) if p is not None else None

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

  Return None if there does not exist such a key.
  """
  if self.is_empty():
    return None
  else:
    p = self.find_position(k)
    if not p.key() < k:
      p = self.before(p)
    return (p.key(), p.value()) if p is not None else None
```

```python

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

  Return None if there does not exist such a key.
  """
  if self.is_empty():
    return None
  else:
    p = self.find_position(k)          # may not find exact match
    if p.key() < k:                    # p's key is too small
      p = self.after(p)
    return (p.key(), p.value()) if p is not None else None

def find_gt(self, k):
  """Return (key,value) pair with least key strictly greater than k.

  Return None if there does not exist such a key.
  """
  if self.is_empty():
    return None
  else:
    p = self.find_position(k)
    if not k < p.key():
      p = self.after(p)
    return (p.key(), p.value()) if p is not None else None

def find_range(self, start, stop):
  """Iterate all (key,value) pairs such that start <= key < stop.

  If start is None, iteration begins with minimum key of map.
  If stop is None, iteration continues through the maximum key of map.
  """
  if not self.is_empty():
    if start is None:
      p = self.first()
    else:
      # we initialize p with logic similar to find_ge
      p = self.find_position(start)
      if p.key() < start:
        p = self.after(p)
    while p is not None and (stop is None or p.key() < stop):
      yield (p.key(), p.value())
      p = self.after(p)
```

```python

#-------------- hooks used by subclasses to balance a tree --------------
def _rebalance_insert(self, p):
  """Call to indicate that position p is newly added."""
  pass

def _rebalance_delete(self, p):
  """Call to indicate that a child of p has been removed."""
  pass

def _rebalance_access(self, p):
  """Call to indicate that position p was recently accessed."""
  pass

#------------ nonpublic methods to support tree balancing ---------------
def _relink(self, parent, child, make_left_child):
  """Relink parent node with child node (we allow child to be None)."""
  if make_left_child:                # make it a left child
    parent._left = child
  else:                              # make it a right child
    parent._right = child
  if child is not None:              # make child point to parent
    child._parent = parent
```

```python

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

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

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

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

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

  Return the Position that becomes root of the restructured subtree.

  Assumes the nodes are in one of the following configurations:

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

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

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

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

Capitolo 5

AVL Tree

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

```python
from .binary_search_tree import TreeMap

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

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

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

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

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

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

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

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

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

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

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

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

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

Capitolo 6

RB Tree

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

```
from .binary_search_tree import TreeMap

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

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

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

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

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

```python

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

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

#------------------------- support for deletions -------------------------
def _rebalance_delete(self, p):
  if len(self) == 1:
    self._set_black(self.root())  # special case: ensure that root is black
  elif p is not None:
    n = self.num_children(p)
    if n == 1:                    # deficit exists unless child is a red leaf
      c = next(self.children(p))
      if not self._is_red_leaf(c):
        self._fix_deficit(p, c)
    elif n == 2:                  # removed black node with red child
      if self._is_red_leaf(self.left(p)):
        self._set_black(self.left(p))
      else:
        self._set_black(self.right(p))
```

```python

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

Capitolo 7

Hash

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

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

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

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

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

    cap     initial table size (default 11)
    p       positive prime used for MAD (default 109345121)
    """
    self._table = cap * [ None ]
    self._n = 0                                 # number of entries in the map
    self._prime = p                             # prime for MAD compression
    self._scale = 1 + randrange(p-1)            # scale from 1 to p-1 for MAD
    self._shift = randrange(p)                  # shift from 0 to p-1 for MAD

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

  def __len__(self):
    return self._n
```

```

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

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

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

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

Capitolo 8

Sorted Unsorted Table

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

```python
from .map_base import MapBase

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

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

    Return high + 1 if no such item qualifies.

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

```python

  #---------------------------- public behaviors ----------------------------
  def __init__(self):
    """Create an empty map."""
    self._table = []

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

  def __getitem__(self, k):
    """Return value associated with key k (raise KeyError if not found)."""
    j = self._find_index(k, 0, len(self._table) - 1)
    if j == len(self._table) or self._table[j]._key != k:
      raise KeyError('Key Error: ' + repr(k))
    return self._table[j]._value

  def __setitem__(self, k, v):
    """Assign value v to key k, overwriting existing value if present."""
    j = self._find_index(k, 0, len(self._table) - 1)
    if j < len(self._table) and self._table[j]._key == k:
      self._table[j]._value = v                         # reassign value
    else:
      self._table.insert(j, self._Item(k,v))            # adds new item

  def __delitem__(self, k):
    """Remove item associated with key k (raise KeyError if not found)."""
    j = self._find_index(k, 0, len(self._table) - 1)
    if j == len(self._table) or self._table[j]._key != k:
      raise KeyError('Key Error: ' + repr(k))
    self._table.pop(j)                                  # delete item

  def __iter__(self):
    """Generate keys of the map ordered from minimum to maximum."""
    for item in self._table:
      yield item._key

  def __reversed__(self):
    """Generate keys of the map ordered from maximum to minimum."""
    for item in reversed(self._table):
      yield item._key

  def find_min(self):
    """Return (key,value) pair with minimum key (or None if empty)."""
    if len(self._table) > 0:
      return (self._table[0]._key, self._table[0]._value)
    else:
      return None

```

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

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

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

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

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

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

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

```python

def find_gt(self, k):
  """Return (key,value) pair with least key strictly greater than k.

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

def find_range(self, start, stop):
  """Iterate all (key,value) pairs such that start <= key < stop.

  If start is None, iteration begins with minimum key of map.
  If stop is None, iteration continues through the maximum key of map.
  """
  if start is None:
    j = 0
  else:
    j = self._find_index(start, 0, len(self._table)-1)   # find first result
  while j < len(self._table) and (stop is None or self._table[j]._key < stop):
    yield (self._table[j]._key, self._table[j]._value)
    j += 1
```

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

```python
from .map_base import MapBase

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

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

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

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

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

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

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

Capitolo 9

Chain Hash

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

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

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

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

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

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

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

Capitolo 10

Probe Hash

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

```python
from .hash_map_base import HashMapBase

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

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

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

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

```python

  def _bucket_getitem(self, j, k):
    found, s = self._find_slot(j, k)
    if not found:
      raise KeyError('Key Error: ' + repr(k))         # no match found
    return self._table[s]._value

  def _bucket_setitem(self, j, k, v):
    found, s = self._find_slot(j, k)
    if not found:
      self._table[s] = self._Item(k,v)                # insert new item
      self._n += 1                                    # size has increased
    else:
      self._table[s]._value = v                       # overwrite existing

  def _bucket_delitem(self, j, k):
    found, s = self._find_slot(j, k)
    if not found:
      raise KeyError('Key Error: ' + repr(k))         # no match found
    self._table[s] = ProbeHashMap._AVAIL              # mark as vacated

  def __iter__(self):
    for j in range(len(self._table)):                 # scan entire table
      if not self._is_available(j):
        yield self._table[j]._key
```

Capitolo 11

Priority Queue

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

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

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

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

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

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

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

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

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

```python

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

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

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

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

Capitolo 12

Sorted Unsorted Priority Queue

Listing 12.1: Implementazione di SortedPriorityQueue.

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

class Empty(Exception):
  pass

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

  #------------------------------ public behaviors ------------------------------

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

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

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

```python

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

    Raise Empty exception if empty.
    """
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    p = self._data.first()
    item = p.element()
    return (item._key, item._value)

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

    Raise Empty exception if empty.
    """
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    item = self._data.delete(self._data.first())
    return (item._key, item._value)
```

Listing 12.2: Implementazione di UnsortedPriorityQueue.

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

class Empty(Exception):
  pass


class UnsortedPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with an unsorted list."""

  #----------------------------- nonpublic behavior -----------------------------
  def _find_min(self):
    """Return Position of item with minimum key."""
    if self.is_empty():               # is_empty inherited from base class
      raise Empty('Priority queue is empty')
    small = self._data.first()
    walk = self._data.after(small)
    while walk is not None:
      if walk.element() < small.element():
        small = walk
      walk = self._data.after(walk)
    return small
```

```python

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

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

    def add(self, key, value):
      """Add a key-value pair."""
      self._data.add_last(self._Item(key, value))

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

      Raise Empty exception if empty.
      """
      p = self._find_min()
      item = p.element()
      return (item._key, item._value)

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

      Raise Empty exception if empty.
      """
      p = self._find_min()
      item = self._data.delete(p)
      return (item._key, item._value)
```

Capitolo 13

Heap Priority Queue

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

```python
from .priority_queue_base import PriorityQueueBase

class Empty(Exception):
  pass


class HeapPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with a binary heap."""

  #------------------------------ nonpublic behaviors ------------------------------
  def _parent(self, j):
    return (j-1) // 2

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

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

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

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

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

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

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

def _heapify(self):
  """Bottom-up construction of a heap in O(n) time."""
  # start at the parent of the last element
  start = self._parent(len(self._data) - 1)
  for j in range(start, -1, -1): # go backward from start to 0
    self._downheap(j)

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

  If contents is provided, it should be an iterable of (k,v) tuples.
  """
  self._data = [self._Item(k, v) for k, v in contents]
  if len(self._data) > 1:
    self._heapify()

""" without Heapify
def __init__(self):
  Create a new empty Priority Queue.
  self._data = []
"""

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

```python

def add(self, key, value):
  """Add a key-value pair to the priority queue."""
  self._data.append(self._Item(key, value))
  self._upheap(len(self._data) - 1)            # upheap newly added position

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

  Raise Empty exception if empty.
  """
  if self.is_empty():
    raise Empty('Priority queue is empty.')
  item = self._data[0]
  return (item._key, item._value)

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

  Raise Empty exception if empty.
  """
  if self.is_empty():
    raise Empty('Priority queue is empty.')
  self._swap(0, len(self._data) - 1)           # put minimum item at the end
  item = self._data.pop()                      # and remove it from the list;
  self._downheap(0)                            # then fix new root
  return (item._key, item._value)
```

Capitolo 14

Adaptable Priority Queue

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

```python
from .heap_priority_queue import HeapPriorityQueue

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

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

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

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

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

```python

#------------------------------ public behaviors ------------------------------
def add(self, key, value):
  """Add a key-value pair."""
  token = self.Locator(key, value, len(self._data)) # initiaize locator index
  self._data.append(token)
  self._upheap(len(self._data) - 1)
  return token

def update(self, loc, newkey, newval):
  """Update the key and value for the entry identified by Locator loc."""
  j = loc._index
  if not (0 <= j < len(self) and self._data[j] is loc):
    raise ValueError('Invalid locator')
  loc._key = newkey
  loc._value = newval
  self._bubble(j)

def remove(self, loc):
  """Remove and return the (k,v) pair identified by Locator loc."""
  j = loc._index
  if not (0 <= j < len(self) and self._data[j] is loc):
    raise ValueError('Invalid locator')
  if j == len(self) - 1:                # item at last position
    self._data.pop()                    # just remove it
  else:
    self._swap(j, len(self)-1)          # swap item to the last position
    self._data.pop()                    # remove it from the list
    self._bubble(j)                     # fix item displaced by the swap
  return (loc._key, loc._value)
```

Capitolo 15

Graph

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

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

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

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

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

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

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

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

```python

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

    def opposite(self, v):
      """Return the vertex that is opposite v on this edge."""
      if not isinstance(v, Graph.Vertex):
        raise TypeError('v must be a Vertex')
      if v is self._origin:
        return self._destination
      elif v is self._destination:
          return self._origin
      raise ValueError('v not incident to edge')

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

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

    def __str__(self):
      return
          '({0},{1},{2})'.format(self._origin,self._destination,self._element)

class Graph:
    """Representation of a simple graph using an adjacency map."""

    def __init__(self, directed=False):
        """Create an empty graph (undirected, by default).

        Graph is directed if optional parameter is set to True.
        """
        self._outgoing = {}
        # only create second map for directed graph; use alias for undirected
        self._incoming = {} if directed else self._outgoing

    def is_directed(self):
        """Return True if this is a directed graph; False if undirected.

        Property is based on the original declaration of the graph, not its contents.
        """
        return self._incoming is not self._outgoing # directed if maps are distinct
```

```python

    def vertex_count(self):
        """Return the number of vertices in the graph."""
        return len(self._outgoing)

    def vertices(self):
        """Return an iteration of all vertices of the graph."""
        return self._outgoing.keys()

    def edge_count(self):
        """Return the number of edges in the graph."""
        total = sum(len(self._outgoing[v]) for v in self._outgoing)
        # for undirected graphs, make sure not to double-count edges
        return total if self.is_directed() else total // 2

    def edges(self):
        """Return a set of all edges of the graph."""
        result = set()       # avoid double-reporting edges of undirected graph
        for secondary_map in self._outgoing.values():
            result.update(secondary_map.values()) # add edges to resulting set
        return result

    def _validate_vertex(self, v):
    """Verify that v is a Vertex of this graph."""
    if not isinstance(v, self.Vertex):
      raise TypeError('Vertex expected')
    if v not in self._outgoing:
      raise ValueError('Vertex does not belong to this graph.')

    def get_edge(self, u, v):
        """Return the edge from u to v, or None if not adjacent."""
        return self._outgoing[u].get(v) # returns None if v not adjacent


    def degree(self, v, outgoing=True):
        """Return number of (outgoing) edges incident to vertex v in the graph.

        If graph is directed, optional parameter used to count incoming edges.
        """
        adj = self._outgoing if outgoing else self._incoming
        return len(adj[v])
```

```python

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

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

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

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

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

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

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

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

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

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

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

Listing 15.2: Metodi di cancellazione per la classe Graph.

Capitolo 16

DFS

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

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

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

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

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

```python

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

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

Capitolo 17

BFS

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

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

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

Capitolo 18

Transitive Closure

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

```python
from copy import deepcopy

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

Capitolo 19

Topological Sort

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

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

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

Capitolo 20

Shortest Path

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

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

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

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

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

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

```python

  while not pq.is_empty():
    key, u = pq.remove_min()
    cloud[u] = key                                # its correct d[u] value
    del pqlocator[u]                              # u is no longer in pq
    for e in g.incident_edges(u):                 # outgoing edges (u,v)
      v = e.opposite(u)
      if v not in cloud:
        # perform relaxation step on edge (u,v)
        wgt = e.element()
        if d[u] + wgt < d[v]:                     # better path to v?
          d[v] = d[u] + wgt                       # update the distance
          pq.update(pqlocator[v], d[v], v)        # update the pq entry

  return cloud                                    # only includes reachable vertices

def shortest_path_tree(g, s, d):
  """Reconstruct shortest-path tree rooted at vertex s, given distance map d.

  Return tree as a map from each reachable vertex v (other than s) to the
  edge e=(u,v) that is used to reach v from its parent u in the tree.
  """
  tree = {}
  for v in d:
    if v is not s:
      for e in g.incident_edges(v, False):        # consider INCOMING edges
        u = e.opposite(v)
        wgt = e.element()
        if d[v] == d[u] + wgt:
          tree[v] = e                             # edge e is used to reach v
  return tree
```

Capitolo 21

Minimum Spanning Tree

Listing 21.1: Struttura dati Partition (Union-Find): gestione di insiemi disgiunti con compressione del cammino e unione per dimensione.

```

class Partition:
  """Union-find structure for maintaining disjoint sets."""

  #------------------------- nested Position class -------------------------
  class Position:
    __slots__ = '_container', '_element', '_size', '_parent'

    def __init__(self, container, e):
      """Create a new position that is the leader of its own group."""
      self._container = container         # reference to Partition instance
      self._element = e
      self._size = 1
      self._parent = self                 # convention for a group leader

    def element(self):
      """Return element stored at this position."""
      return self._element

  #------------------------- nonpublic utility -------------------------
  def _validate(self, p):
    if not isinstance(p, self.Position):
      raise TypeError('p must be proper Position type')
    if p._container is not self:
      raise ValueError('p does not belong to this container')
```

```python

#------------------------- public Partition methods -------------------------
def make_group(self, e):
  """Makes a new group containing element e, and returns its Position."""
  return self.Position(self, e)

def find(self, p):
  """Finds the group containging p and return the position of its leader."""
  self._validate(p)
  if p._parent != p:
    p._parent = self.find(p._parent)    # overwrite p._parent after recursion
  return p._parent

def union(self, p, q):
  """Merges the groups containg elements p and q (if distinct)."""
  a = self.find(p)
  b = self.find(q)
  if a is not b:                        # only merge if different groups
    if a._size > b._size:
      b._parent = a
      a._size += b._size
    else:
      a._parent = b
      b._size += a._size
```

Listing 21.2: Algoritmo di Prim: espansione del cluster tramite coda di priorità adattabile.

```python


from ..priority_queue.heap_priority_queue import HeapPriorityQueue
from ..priority_queue.adaptable_heap_priority_queue import
    AdaptableHeapPriorityQueue
from .partition import Partition

def MST_Prim(g):
  """Compute a minimum spanning tree of weighted graph g.

  Return a list of edges that comprise the MST (in arbitrary order).
  """
  d = {}                                   # d[v] is bound on distance to tree
  tree = []                                # list of edges in spanning tree
  pq = AdaptableHeapPriorityQueue()       # d[v] maps to value (v, e=(u,v))
  pqlocator = {}                           # map from vertex to its pq locator

  # for each vertex v of the graph, add an entry to the priority queue, with
  # the source having distance 0 and all others having infinite distance
  for v in g.vertices():
    if len(d) == 0:                                 # this is the first node
      d[v] = 0                                      # make it the root
    else:
      d[v] = float('inf')                           # positive infinity
    pqlocator[v] = pq.add(d[v], (v,None))

  while not pq.is_empty():
    key,value = pq.remove_min()
    u,edge = value                                  # unpack tuple from pq
    del pqlocator[u]                                # u is no longer in pq
    if edge is not None:
      tree.append(edge)                             # add edge to tree
    for link in g.incident_edges(u):
      v = link.opposite(u)
      if v in pqlocator:                            # thus v not yet in tree
        # see if edge (u,v) better connects v to the growing tree
        wgt = link.element()
        if wgt < d[v]:                              # better edge to v?
          d[v] = wgt                                # update the distance
          pq.update(pqlocator[v], d[v], (v, link))  # update the pq entry

  return tree
```

Listing 21.3: Algoritmo di Kruskal: approccio greedy basato sull'ordinamento degli archi e gestione dei cicli tramite Partition.

```

def MST_Kruskal(g):
  """Compute a minimum spanning tree of a graph using Kruskal's algorithm.

  Return a list of edges that comprise the MST.

  The elements of the graph's edges are assumed to be weights.
  """
  tree = []                     # list of edges in spanning tree
  pq = HeapPriorityQueue()      # entries are edges in G, with weights as key
  forest = Partition()          # keeps track of forest clusters
  position = {}                 # map each node to its Partition entry

  for v in g.vertices():
    position[v] = forest.make_group(v)

  for e in g.edges():
    pq.add(e.element(), e)      # edge's element is assumed to be its weight

  size = g.vertex_count()
  while len(tree) != size - 1 and not pq.is_empty():
    # tree not spanning and unprocessed edges remain
    weight,edge = pq.remove_min()
    u,v = edge.endpoints()
    a = forest.find(position[u])
    b = forest.find(position[v])
    if a != b:
      tree.append(edge)
      forest.union(a,b)

  return tree
```