UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA
E MATEMATICA APPLICATA

Corso:
DESIGN AND ANALYSIS OF ALGORITHMS

APPUNTI

Carmine Terracciano

Mat. IE22700109

ANNO ACCADEMICO 2025/2026

Sommario

Capitolo 1

Array Stack

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

```
class Empty(Exception):
  pass

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

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

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

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

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

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

Capitolo 2

Array Queue

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

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

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

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

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

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

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

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

```
from .queue import Queue

class Empty(Exception):
  pass

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

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

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

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

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

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

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

Capitolo 3

Alberi

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

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

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

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

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

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

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

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

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

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

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

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

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

```
from .tree import Tree

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

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

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

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

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

```
49   def inorder(self):
50     """Generate an inorder iteration of positions in the tree."""
51     if not self.is_empty():
52       for p in self._subtree_inorder(self.root()):
53         yield p
54
55   def _subtree_inorder(self, p):
56     """Generate an inorder iteration of positions in subtree rooted at p."""
57     if self.left(p) is not None:          # if left child exists, traverse
         its subtree
58       for other in self._subtree_inorder(self.left(p)):
59         yield other
60     yield p                               # visit p between its subtrees
61     if self.right(p) is not None:         # if right child exists, traverse
         its subtree
62       for other in self._subtree_inorder(self.right(p)):
63         yield other
64
65   # override inherited version to make inorder the default
66   def positions(self):
67     """Generate an iteration of the tree's positions."""
68     return self.inorder()                 # make inorder the default
```

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

```
from .binary_tree import BinaryTree

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

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

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

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

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

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

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

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

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

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

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

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