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

```
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

  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')
    return self._data[-1]          # legge l'ultimo elemento della list
```

```
29
30  def pop(self):
31    """Rimuove e restituisce l'elemento al top dello stack.
32       Raise Empty exception se lo stack è vuoto."""
33    if self.is_empty():
34      raise Empty('lo stack è vuoto')
35      # print("lo stack è vuoto")
36    return self._data.pop()      # rimuove l'ultimo elemento della list
```

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

Capitolo 2

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

```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
```

```
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
```

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

Capitolo 3

Alberi

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