Laboratorio 08

Programmazione - CdS Matematica

Andrea Burattin
17 dicembre 2013

Esercizio 1 – I

Costruire la classe albero binario

```
class albero():
    def __init___(...):
        # ...
```

Definire le principali funzioni di utilità:

```
def altezza(a):
    # ...

def n_nodi(a):
    # ...

def stampa(a):
    # ...
```

Esercizio 1 – II

Classe `albero`:

```
class albero():
   def __init__(self, val=None, sx=None, dx=None):
      self.val = val
      self.sx = sx
      self.dx = dx
```

Funzioni di utilità:

```
def vuoto(a):
   return a==None
```

```
def altezza(a):
   if vuoto(a):
      return -1
   return max(altezza(a.sx),altezza(a.dx))+1
```

Esercizio 1 – III

```
def n_nodi(a):
    if vuoto(a):
        return 0
    return n_nodi(a.sx) + n_nodi(a.dx) + 1
```

```
def stampa(a, livello=0, separator=""):
    if a == None: return
    stampa(a.dx, livello+1, ".--")
    print "    "*(livello-1) + separator + str(a.val)
    stampa(a.sx, livello+1, "`--")
```

Esercizio 1 – IV

Esempio:

```
a = albero(1, albero(2), albero(3, dx=albero(4)))
stampa(a)



   .--4
.--3
1
`--2
```

Esercizio 2 – I

Definire una funzione `build_tree` che riceve in *input* una tupla che rappresenta un albero di profondità arbitraria, e restituisce la struttura dati corrispondente

La tupla passata a `build_tree` è composta **esattamente** da tre componenti: il valore del nodo, la tupla che rappresenta il sottoalbero sinistro; la tupla che rappresenta il sottoalbero destro

Esempi di tupla:

- `(1, None, None)`
- `(1, None, (3, None, None))`
- `(1, (2,None, (4,None,None)), (3,None,None))`

Esercizio 2 – II

Verifica

```
stampa(build_tree((1, None, None)))
1

stampa(build_tree((1, None, (3, None, None))))
.--3
1

stampa(build_tree(
              (1,(2,None,(4,None,None)),(3,None,None))))
.--3
1
   .--4
`--2
```

Esercizio 2 – III

Proposta di soluzione:

```
def build_tree(t):
    v = t[0]
    tsx = t[1]
    tdx = t[2]
    if tsx:
        tsx = build_tree(tsx)
    if tdx:
        tdx = build_tree(tdx)
    return albero(v, tsx, tdx)
```

Esercizio 3 – I

Definire la funzione `nodi_maggiori_di(t, n)` che, dato un albero `t` ed un intero `n`, restituisce il numero di nodi, in `t`, che hanno valore maggiore di `n`.

Esercizio 3 – II

Verifica

```
a = build_tree((1, (2, (3,None,None), (5, (4,None,None),
    None)), (6,None,None)))
stampa(a)
print nodi_maggiori_di(a, -1)
print nodi_maggiori_di(a, 3)
print nodi_maggiori_di(a, 6)
```

```
.--6
1
   .--5
      `--4
`--2
   `--3
6
3
0
```

Esercizio 3 – III

Proposta di soluzione

```
def nodi_maggiori_di(t, n):
    if vuoto(t):
        return 0
    if t.val > n:
        return 1 + nodi_maggiori_di(t.sx, n) +
            nodi_maggiori_di(t.dx, n)
    else:
        return nodi_maggiori_di(t.sx, n) +
            nodi_maggiori_di(t.dx, n)
```

Esercizio 4 – I

Scrivere una (o più) funzione `confronta(a, l)` che attraversa l'albero e confronta la lista dei valori dei suoi nodi con la lista data.

La funzione deve ritornare vero nel seguente caso: $[1, 2, 3, 4, 5]$

Esercizio 4 – II

Proposta soluzione

```
def confronta(a, l):
    return traverse(a) == l

def traverse(a):
    l = [a.val]
    if a.sx:
        l += traverse(a.sx)
    if a.dx:
        l += traverse(a.dx)
    return l

t = build_tree((1,(2,(3, None, None),(4, None, None))
    ,(5, None, None)))
```

Esercizio 5 – I

Modificare l'esercizio precedente, in modo tale che la lista $[1, 2, 3, 4, 5, 6, 7, 8, 9]$ sia matchata con l'albero:

Esercizio 5 – II

Proposta soluzione

```
def traverse(a):
    l = []
    if a.sx:
        l += traverse(a.sx)
    if a.dx:
        l += traverse(a.dx)
    l += [a.val]
    return l

t = build_tree((9,(4, None,(3,(1, None, None),(2, None,
    None))),(8,(6, None,(5, None, None)),(7, None, None))
    ))
```