Laboratorio 07

Programmazione - CdS Matematica

Michele Donini
10 dicembre 2015

Esercizio Lista – I

Costruire la classe lista concatenata:

```
class Lista():
    def __init___(self, val=None, succ=None):
        # ...
```

Definire le principali funzioni di utilità:

```
def vuota(lista):
    # ...

def n_nodi(lista):
    # ...

def stampa_lista(lista):
    # ...
```

Esercizio Lista – II

Classe `Lista`:

```
class Lista():
    def __init__(self, val=None, succ=None):
        self.val = val
        self.succ = succ
```

Funzioni di utilità:

```
def vuota(lista):
    return lista == None
```

```
def n_nodi(lista):
    if vuota(lista):
        return 0
    return 1 + n_nodi(lista.succ)
```

Esercizio Lista – III

```
def stampa_lista(lista):
    if not vuota(lista):
        print lista.val,
        stampa_lista(lista.succ)
    else:
        print
```

Esercizio Lista – IV

Esempio:

```
l1 = None
l2 = Lista(1, Lista(2, Lista(3, Lista(4))))
print vuota(l1)
print n_nodi(l1)
print vuota(l2)
print n_nodi(l2)
stampa_lista(l2)

True
0
False
4
1 2 3 4
```

Esercizio Lista – V

Definire le principali funzioni di gestione lista:

```
def aggiungi_testa(lista, val):
    # ...

def aggiungi_coda(lista, val):
    # ...

def lista_inversa(lista):
    # ...
```

Esercizio Lista – VI

```
def aggiungi_testa(lista, val):
    if vuota(lista):
        return Lista(val)
    return Lista(val, lista)
```

```
def aggiungi_coda(lista, val):
    if vuota(lista):
        return Lista(val)
    lista.succ = aggiungi_coda(lista.succ, val)
    return lista
```

```
def lista_inversa(lista):
    if vuota(lista):
        return None
    l = lista_inversa(lista.succ)
    return aggiungi_coda(l, lista.val)
```

Esercizio Lista – VII

Esempio:

```
l = Lista(1, Lista(2, Lista(3, Lista(4))))
l = aggiungi_testa(l, 0)
stampa_lista(l)
l = aggiungi_coda(l, 5)
stampa_lista(l)
l = lista_inversa(l)
stampa_lista(l)

0 1 2 3 4
0 1 2 3 4 5
5 4 3 2 1 0
```

Esercizio Albero – I

Costruire la classe albero binario

```
class Albero():
    def __init___(self, val=None, sx=None, dx=None):
        # ...
```

Definire le principali funzioni di utilità:

```
def vuoto(albero):
    # ...

def altezza(albero):
    # ...

def n_nodi(albero):
    # ...

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

Esercizio Albero – 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(albero):
    return albero == None
```

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

Esercizio Albero – III

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

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

Esercizio Albero – IV

Esempio:

```
a = albero(1, albero(2), albero(3, dx=albero(4)))
stampa(a)
print vuoto(a)
print altezza(a)
print n_nodi(a)

    .--4
.--3
1
'--2
False
2
4
```

Esercizio Albero – V

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 Albero – VI

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 Albero – VII

Proposta di soluzione:

```
def build_tree(t):
    v = t[0]
    sx = t[1]
    dx = t[2]
    if sx:
        sx = build_tree(sx)
    if dx:
        dx = build_tree(dx)
    return Albero(v, sx, dx)
```

Esercizio Albero – VIII

Scrivere una (o più) funzione `confronta(albero, lista)` 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:

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

stampa(t)
.--5
1
   .--4
`--2
   `--3

confronta(t, [1,2,3,4,5]) #True
```

Esercizio Albero – IX

Proposta soluzione

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

def traverse(albero):
    if vuoto(albero):
        return []
    return [albero.val] + traverse(albero.sx)
                        + traverse(albero.dx)
```

Esercizio Albero – X

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

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

confronta(t, [1,2,3,4,5,6,7,8,9]) #True
```

Esercizio Albero – XI

Proposta soluzione

```
def confronta2(t, l):
    return traverse2(t) == l

def traverse2(albero):
    if vuoto(albero):
        return []
    return traverse2(albero.sx) + traverse2(albero.dx)
                                + [albero.val]
```

Esercizio Reverse Engineering I

ATTENZIONE: esercizio facsimile all'esame

Descrivere le pre e le post condizioni della seguente funzione e dimostrarne la correttezza:

```
def A(a):
    if not a or (not a.left and not a.right):
        return 0.0
    return A(a.left) + A(a.right) + 1
```

Esercizio Reverse Engineering II

- **PRE**: il parametro `a` è un albero binario anche vuoto (== None);
- **POST**: ritorna il numero di nodi interni (non foglia) dell'albero;
- **DIM**:
  - `POS(A(None))` e `POS(A(Albero(x, None, None))`: ritorna 0 perché l'albero è vuoto oppure la radice è anche foglia;
  - `PRE(a.left)` e `PRE(a.right)`: valgono perché l'albero `a` è ben definito come lo sono i due sotto-alberi `a.left` e `a.right`;
  - `POS(A(a))`: assumendo l'albero non vuoto e le post-condizioni vere per i due sotto-alberi, allora la funzione ritorna la somma dei nodi interni del sotto-albero destro più quello sinistro più 1 (poiché il nodo corrente è interno), che corrisponde al numero di nodi interni dell'intero albero.