Laboratorio 08

Programmazione - CdS Matematica

Luca Righi

9 gennaio 2018

Operare su File

Per operare su un file occorre eseguire le seguenti operazioni:

- aprire il file in scrittura o lettura;
- scrivere o leggere su file (a seconda della modalità in cui è stato aperto)
- chiudere il file

```
# scrittura su file
outfile = open("test_file.txt","w") #w=write, a=append
outfile.write("Questo e' un testo di prova.\nProva ad
    aprire il file e guardare che cosa c'e' dentro.\n")
outfile.close()
```

```
# lettura da file
infile = open("test_file.txt","r") #r=read
testo = infile.read()
infile.close()
print testo
```

Altri metodi dell'oggetto file

- `readline()`: legge una sola riga
- `readlines()`: ritorna l'intero file come lista di righe
- `writelines(L)`: scrive in ogni nuova riga gli elementi della lista L

```
infile = open("test_file.txt","r")
l = infile.readlines()
infile.close()
print l
```

```
["Questo e' un testo di prova.\n", "Prova ad aprire il
    file e guardare che cosa c'e' dentro.\n"]
```

Scrittura su File

- Definire una funzione `genera_valori(N)` che generi N numeri interi in $[0,10^4[$ e li scriva su un file, uno per riga.

Scrittura su File

- Definire una funzione `genera_valori(N)` che generi N numeri interi in $[0,10^4[$ e li scriva su un file, uno per riga.

```
from random import randint
def genera_valori(N):
    filevalori = open("valori.txt","w")
    for v in range(N):
        val = randint(0, 10000)
        filevalori.write(str(val)+'\n')
    filevalori.close()

genera_valori(100000)
```

Lettura da File

- Leggere e stampare i valori contenuti nel file generato nel punto precedente.

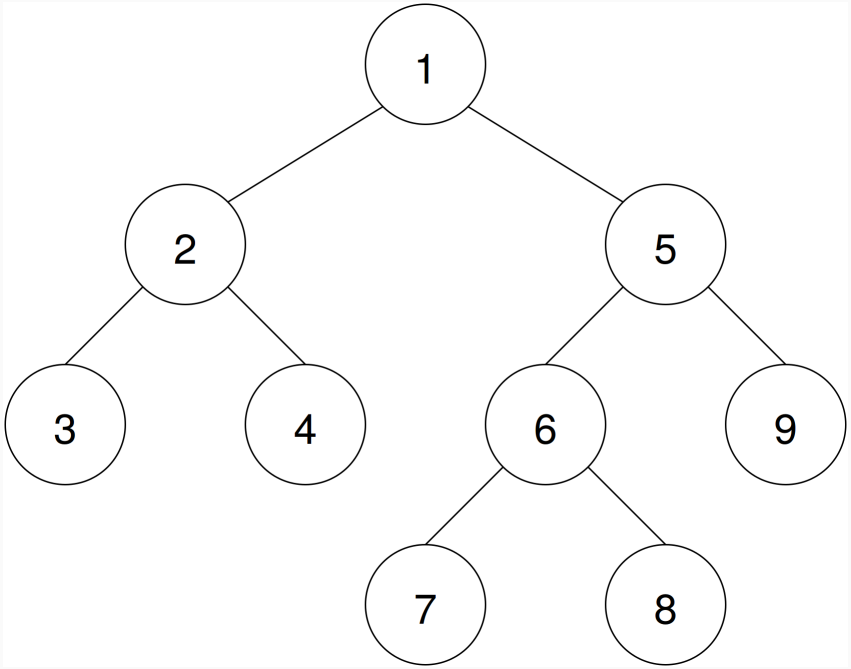
Lettura da File

- Leggere e stampare i valori contenuti nel file generato nel punto precedente.

```
filevalori = open("valori.txt","r")
for v in filevalori.readlines():
    val = int(v)
    print val
```

Alberi

Un albero è una struttura dati ricorsiva formata principalmente da nodi.
Ogni nodo, oltre a contenere informazione utile, ha un collegamento con i suoi nodi *figli*, e con al più un nodo *padre*. Ad eccezione del nodo *radice*.



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):
```

Esercizio Albero – ii

```
    # ...

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

Esercizio Albero – iii

Esempio:

```
t = albero(5)
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 – iv

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
```

Esercizio Albero – v

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

Esercizio Albero – vi

```
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 – vii

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 – viii

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 – ix

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 – x

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 – xi

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 – xii

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
```

Esercizio Albero – xiii

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

Esercizio Albero – xiv

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]
```