Laboratorio 08

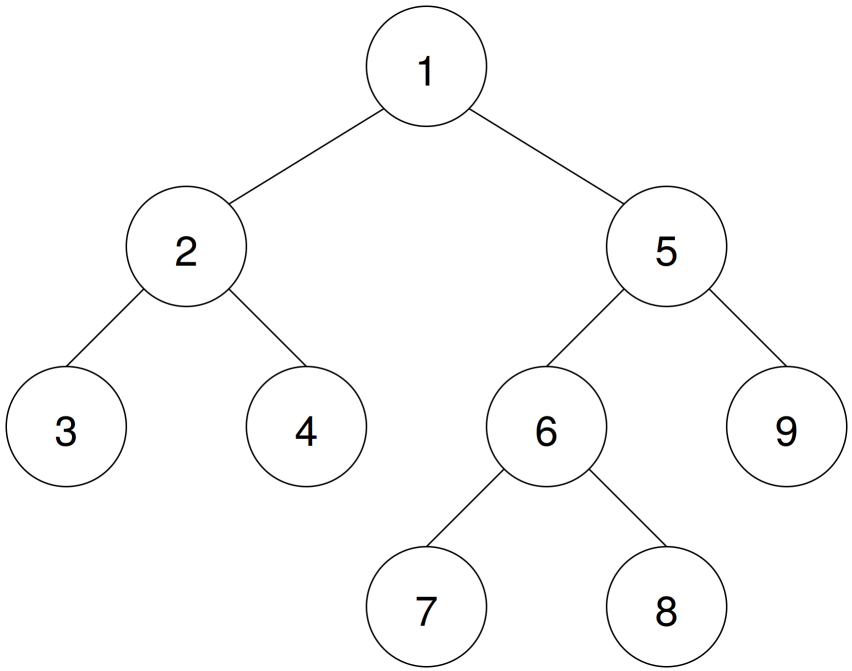
Programmazione - CdS Matematica

Lauriola Ivano

10 gennaio 2017

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):
    # ...

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

Esercizio Albero – II

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 – III

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 – IV

```
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 – 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.

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.

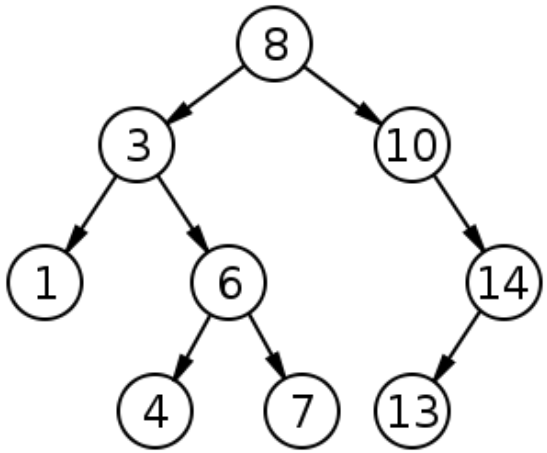
Lettura da File

- 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 binari di ricerca

Gli *alberi binari di ricerca* (binary search trees, **BST**), detti anche *alberi ordinati*, sono particolari alberi binari con la seguente proprietà:
il valore di ogni nodo è maggiore del valore di tutti i nodi del sotto-albero sinistro e minore del valore di tutti i nodi del sotto-albero destro.



Generazione BST

Definire la funzione `bst_ins(T,v)` (esterna alla classe Albero) che inserisca il valore v nell'albero T (anche vuoto) mantenendo la proprietà dei BST. La funzione deve ritornare l'albero con il nuovo valore inserito.

Generazione BST

Definire la funzione `bst_ins(T,v)` (esterna alla classe Albero) che inserisca il valore v nell'albero T (anche vuoto) mantenendo la proprietà dei BST. La funzione deve ritornare l'albero con il nuovo valore inserito.

```
def bst_ins(T,v):
    if not T:
        return Albero(v)
    if T.val>v:
        T.sx = bst_ins(T.sx,v)
    if T.val<v:
        T.dx = bst_ins(T.dx,v)
    return T
```

Generazione BST

- Leggere dal file i primi 100 valori e generare un BST dei valori letti.

Generazione BST

- Leggere dal file i primi 100 valori e generare un BST dei valori letti.

```
n_nodi = 100
filevalori = open("valori.txt","r")
A = None
lista_valori  = filevalori.readlines()
for v in range(n_nodi):
    val = int(lista_valori[v])
    A = bst_ins(A,val)
```

Funzione di stampa

- Definire una funzione `print_lista_albero(T)` che stampa a video gli elementi contenuti in un albero BST ordinati in modo crescente.

Funzione di stampa

- Definire una funzione `print_lista_albero(T)` che stampa a video gli elementi contenuti in un albero BST ordinati in modo crescente.

```
def print_lista_albero(T):
    if not T:
        return
    if T.sx:
        print_lista_albero(T.sx)
    print T.val
    if T.dx:
        print_lista_albero(T.dx)
```

Funzione di ricerca (valore esatto)

- Definire la funzione di ricerca `bst_search(T, v)` di un elemento v in un BST T: se l'elemento cercato esiste ritorna il sotto-albero che ha v come radice, altrimenti ritorna None.

Funzione di ricerca (valore esatto)

- Definire la funzione di ricerca `bst_search(T, v)` di un elemento v in un BST T: se l'elemento cercato esiste ritorna il sotto-albero che ha v come radice, altrimenti ritorna None.

```
def bst_search(T, v):
    if not T:
        return None
    if T.val==v:
        return T
    if T.val>v:
        return bst_search(T.sx,v)
    return bst_search(T.dx,v)
```

Funzione di ricerca (intervallo)

- Creare una funzione `bst_search_intervallo(T, a, b)` che trova, se esiste, un valore in T contenuto in un intervallo chiuso [a,b] (ritorna il primo elemento appartenente all'intervallo o None se non ve ne sono).

Funzione di ricerca (intervallo)

- Creare una funzione `bst_search_intervallo(T, a, b)` che trova, se esiste, un valore in T contenuto in un intervallo chiuso [a,b] (ritorna il primo elemento appartenente all'intervallo o None se non ve ne sono).

```
def bst_search_intervallo(T, a, b):
    if not T:
        return None
    if T.val>=a and T.val<=b:
        return T.val
    if T.val>b:
        return bst_search_intervallo(T.sx,a,b)
    return bst_search_intervallo(T.dx,a,b)
```

Calcolo del numero di ricorsioni

- Calcolare il numero di chiamate ricorsive necessario per la ricerca di un valore contenuto in un intervallo. Suggerimento: definire una variabile globale che faccia da contatore delle chiamate ricorsive alla funzione `bst_search_intervallo`.

```
conta_iterazioni = ...
def bst_search_intervallo(T, a, b):
    global conta_iterazioni
    ...
```

Calcolo del numero di ricorsioni

- Calcolare il numero di chiamate ricorsive necessario per la ricerca di un valore contenuto in un intervallo. Suggerimento: definire una variabile globale che faccia da contatore delle chiamate ricorsive alla funzione `bst_search_intervallo`.

```
conta_iterazioni = 0
def bst_search_intervallo(T, a, b):
    global conta_iterazioni
    conta_iterazioni = conta_iterazioni + 1
    if not T:
        return None
    if T.val>=a and T.val<=b:
        return T.val
    if T.val>b:
        return bst_search_intervallo(T.sx,a,b)
    return bst_search_intervallo(T.dx,a,b)
```

Calcolo empirico della complessità

- Calcolare il numero di chiamate ricorsive **medio** necessario per la ricerca di un valore in un intervallo di ampiezza fissa 10 contenuto in $[0,10^4[$: replicare più volte ($\approx 10^4$) il calcolo precedente variando ogni volta l'intervallo in modo casuale.
- Ripetere il calcolo modificando il numero di nodi caricati nell'albero (n_nodi = 100, 1000, 10000, ...).

Calcolo empirico della complessità

- Calcolare il numero di chiamate ricorsive **medio** necessario per la ricerca di un valore in un intervallo di ampiezza fissa 10 contenuto in $[0,10^4[$: replicare più volte ($\approx 10^4$) il calcolo precedente variando ogni volta l'intervallo in modo casuale.
- Ripetere il calcolo modificando il numero di nodi caricati nell'albero (n_nodi = 100, 1000, 10000, ...).

```
conta_iterazioni = 0
n_prove = 10000
for i in range(n_prove):
    a = randint(0, 10000 - 10)
    b = a + 10
    out = bst_search_intervallo(A,a,b)
print conta_iterazioni/float(n_prove)
```