Laboratorio 8

Programmazione - CdS Matematica

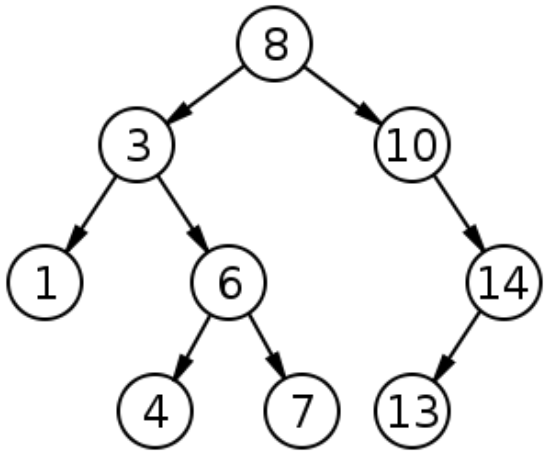
Marco Virgulin
13 Gennaio 2015

Esercizio di riepilogo.

- Generare N numeri reali in [0,1[e scriverli su file.
- Leggere i valori dal file e inserirli in un *albero binario di ricerca*.
- Definire una funzione che cerca nell'albero un valore contenuto in un intervallo specifico passato come argomento (se esiste ritorna il primo che trova).
- Verificare il "numero di chiamate ricorsive medio" necessario per la ricerca di un valore contenuto in un intervallo (eseguendo più volte la funzione precedente, per intervalli diversi).

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.



Alberi binari di ricerca

Tale proprietà permette di implementare la ricerca di un valore in un BST in modo più efficiente rispetto ad una semplice ricerca con visita dell'albero:

per ogni nodo dell'albero, possiamo verificare se tale nodo contiene il valore che stiamo cercando (nel qual caso lo si ritorna), altrimenti la ricerca continua nel solo sotto-albero sinistro o destro a seconda se il valore cercato è minore o maggiore del valore del nodo.

Se l'albero è bilanciato, i.e. ha altezza proporzionale la logaritmo nel numero di nodi $h \approx log(n)$, allora la complessità della ricerca si riduce da n (ricerca lineare su albero binario) a $log_2(n)$.

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")
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")
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"]
```

Parte 1 - Scrittura su File

- Definire una funzione `genera_valori(N)` che generi N numeri reali in [0,1[e li scriva su un file, uno per riga.

Parte 1 - Scrittura su File

- Definire una funzione `genera_valori(N)` che generi N numeri reali in [0,1[e li scriva su un file, uno per riga.

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

genera_valori(100000)
```

Parte 1 - Lettura da File

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

Parte 1 - 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 = float(v)
    print val
```

Parte 2 - Classe Albero

Definire la classe Albero (binario). È sufficiente definire il solo costruttore.

Parte 2 - Classe Albero

Definire la classe Albero (binario). È sufficiente definire il solo costruttore.

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

Parte 2 - 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.

Parte 2 - 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
```

Parte 2 - Generazione BST

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

Parte 2 - Generazione BST

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

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

Parte 3 - Funzione di stampa

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

Parte 3 - Funzione di stampa

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

Parte 3 - 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.

Parte 3 - 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)
```

Parte 3 - 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).

Parte 3 - 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)
```

Parte 4 - 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`.

Parte 4 - 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
    ...
```

Parte 4 - 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`.

Parte 4 - 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)
```

Parte 4 - Calcolo empirico della complessità

- Calcolare il numero di chiamate ricorsive **medio** necessario per la ricerca di un valore in un intervallo di ampiezza fissa 0.00001 contenuto in [0,1]: replicare più volte (≈ 10000) 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, ...).

Parte 4 - Calcolo empirico della complessità

- Calcolare il numero di chiamate ricorsive **medio** necessario per la ricerca di un valore in un intervallo di ampiezza fissa 0.00001 contenuto in [0,1]: replicare più volte (≈ 10000) 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, ...).

```
n_prove = 10000
for i in range(n_prove):
    a = random.random()
    b = a + 0.00001
    if b > 1.0:
        b,a = 1.0,a - 0.00001
    out = bst_search_intervallo(A,a,b)
print conta_iterazioni/float(n_prove)
```