Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Strutture dati fondamentali: Visite di alberi

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Sulla base delle slides a cura di T. Calamoneri e G. Bongiovanni per il corso di informatica generale AA 2019/2020

Visite di Alberi

Un'operazione basilare sugli alberi è l'accesso a tutti i suoi nodi, uno dopo l'altro, al fine di poter effettuare una specifica operazione (che dipende ovviamente dal problema posto) su ciascun nodo.

Tale operazione sulle liste si effettua con una semplice iterazione, ma sugli alberi la situazione è più complessa dato che la loro struttura è ben più articolata.

L'accesso progressivo a tutti i nodi di un albero si chiama ***visita dell'albero***.

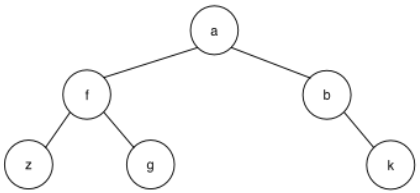
Visite di Alberi

Facendo riferimento all'ordine col quale si accede ai nodi dell'albero, è evidente che esiste più di una possibilità.

Nel caso degli alberi binari, nei quali i figli di ogni nodo (e quindi i sottoalberi) sono al massimo due, e volendo comportarsi nello stesso modo su tutti i nodi, le possibili decisioni in merito a questa scelta danno luogo a tre diverse visite:

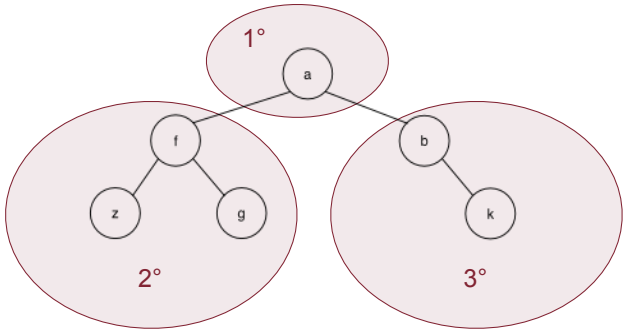
- ***visita in preordine (preorder)***: il nodo è visitato prima di proseguire la visita nei suoi sottoalberi;
- ***visita inordine (inorder)***: il nodo è visitato dopo la visita del sottoalbero sinistro e prima di quella del sottoalbero destro;
- ***visita postordine (postorder)***: il nodo è visitato dopo entrambe le visite dei sottoalberi.

Le tre visite dell'albero



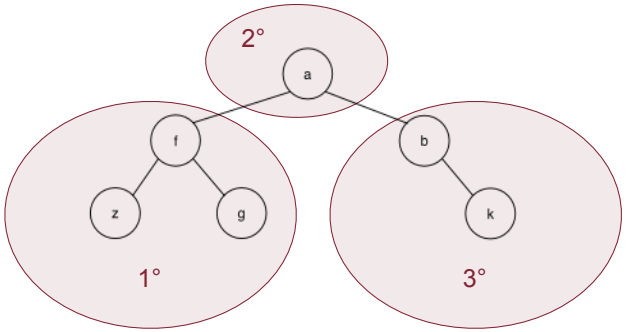
- visita in preordine: **a f z g b k**
- visita in inordine: **z f g a b k**
- visita in postordine: **z g f k b a**

Visita in preordine



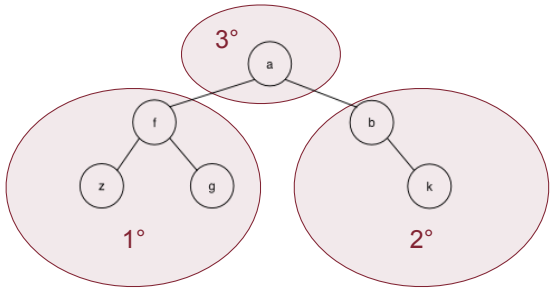
- visita in preordine: **a f z g b k**

Visita in inordine



- visita in inordine: **z f g a b k**

Visita postordine



- visita in postordine: **z g f k b a**

Visite di Alberi

Se l'albero è memorizzato tramite record e puntatori ed è quindi dato tramite il puntatore p alla sua radice:

```
def VisitaPreordine(p):
   if p != None:
      #accesso al nodo e operazioni conseguenti
      VisitaPreordine(p.left)
      VisitaPreordine(p.right)
```

Le altre visite sono analoghe.

L'unica differenza fra i tre casi è la posizione della pseudo-istruzione relativa all'accesso al nodo per effettuarvi le operazioni desiderate.

Visite di Alberi

L'unica differenza fra i tre casi è la posizione della pseudo-istruzione relativa all'accesso al nodo per effettuarvi le operazioni desiderate.

```
def VisitaInordine(p):
   if p != None:
      VisitaInordine(p.left)
      #accesso al nodo e operazioni conseguenti
      VisitaInordine(p.right)

def VisitaPostordine(p):
   if p != None:
      VisitaPostordine(p.left)
      VisitaPostordine(p.right)
      #accesso al nodo e operazioni conseguenti
```

Costo computazionale delle visite

E', ovviamente, lo stesso per tutte e tre.

Esso varia al variare della struttura dati utilizzata per memorizzare l'albero.

Nel caso di memorizzazione tramite record e puntatori, detto k il numero di nodi del sottoalbero sinistro della radice, l'equazione di ricorrenza è:

$$T(n) = T(k) + T(n-k-1) + \Theta(1)$$
$$T(0) = \Theta(1)$$

Facciamoci un'idea della possibile soluzione…

Costo computazionale delle visite – caso migliore

Si verifica quando l'albero è completo, poiché la suddivisione tra le due chiamate ricorsive è la più equa possibile.

In tal caso, se h è il numero dei suoi livelli, si ha $n = 2^{h+1} - 1$ ed entrambi i sottoalberi di ogni nodo sono completi. L'equazione quindi diviene:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$$

che ricade nel caso 1 del teorema principale, ed ha quindi soluzione:

$$T(n) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

Costo computazionale delle visite – caso peggiore

Si verifica quando $k = 0$ (oppure, simmetricamente, quando $n - k - 1 = 0$), per il quale si ottiene:

$$T(n) = T(n-1) + \Theta(1)$$

che ha banalmente, ad esempio col metodo iterativo, la soluzione:

$$T(n) = n \cdot \Theta(1) = \Theta(n)$$

Costo computazionale delle visite – caso generale

Il costo computazionale nel caso generale è limitato inferiormente da quello del caso migliore, quindi è $\Omega(n)$, e superiormente da quello del caso peggiore, quindi è $O(n)$.

Abbiamo dunque $\Omega(n) \leq T(n) \leq O(n)$

Di conseguenza: $T(n) = \Theta(n)$.

Dimostriamolo formalmente, risolvendo l'equazione con il metodo di sostituzione.

Costo computazionale delle visite – caso generale (segue)

Eliminiamo prima la notazione asintotica:

- $T(n) = T(k) + T(n-1-k) + b$
- $T(1) = a$

per due costanti positive *a* e b.

Tentiamo la soluzione $T(n) \leq c \cdot n$, dove c è una costante da determinare.

Passo base: $T(1) = a \leq c$ che è vero se prendiamo $c > a$

Passo induttivo: $T(n) \leq ck + c(n-1-k) + b$
$= c(n-1) + b$
$= c \cdot n - c + b$
$\leq c \cdot n$ *che è vero se prendiamo* $c > b$

Abbiamo dimostrato che $T(n) = O(n)$

Si dimostra analogamente che $T(n) \geq c' \cdot n$, quindi che $T(n) = \Omega(n)$

Ne segue che $T(n) = \Theta(n)$

Applicazioni delle visite

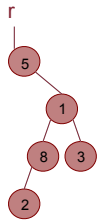
Le visite sono estremamente utili per ispezionare l'albero e dedurne delle proprietà.

A seconda delle proprietà che si vuole esaminare può essere più utile una delle tre visite considerate.

Esempio: conta il numero di nodi presenti nell'albero.

```
def ContaNodi(p):
      if p == None: return 0
      ns = ContaNodi(p.left)
      nd = ContaNodi(p.right)
      return ns + nd + 1 # accesso al nodo
```

```
>>> contaNodi(r)
5
```



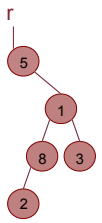
N.B. segue la filosofia della visita in postordine!

Al posto delle 3 istruzioni dopo l'if possiamo scrivere semplicemente :

```
return ContaNodi(p.left)+contaNodi(p.right) +1
```

Esempio: Ricerca la chiave k in un albero

```
def cerca(p, k):
      if p == None: return False
      if p.key == k: return True
      if cerca(p.left,k): return True
      return cerca(p.right,k)

>>> cerca1(r,8)
True
>>> cerca1(r,7)
False
```



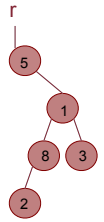
N.B. segue la filosofia della visita in preordine!
Si può sfruttare la compattezza di python per riscrivere la funzione:

```
def cerca1(p, k):
 if p == None: return False
 return p.key==k or cerca1(p.left,k)or cerca1(p.right,k)
```

***Esercizio proposto**: modificare il codice in modo da restituire il puntatore al nodo che contiene la chiave k oppure `None` se la chiave k non è presente.*

Esempio: calcolo dell'altezza dell'albero (stabiliamo che l'albero vuoto ha altezza-1, ricorda che l'albero di un solo nodo ha altezza 0.).

```
def altezza(p):
  if p == None: return -1  #albero vuoto
  h = max(altezza(p.left), altezza(p.right))
  return h + 1

>>> altezza(r)
3
```

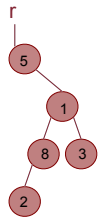


N.B. segue la filosofia della visita in postordine!

Esempio: conteggio dei nodi presenti nel livello k (ricorda che la radice è a livello zero).

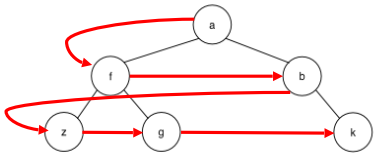
```
def contaLiv(p, k):
   if p == None: return 0 #albero vuoto
   if k==0: return 1
   ks = contaLiv(p.left, k-1)
   kd = contaLiv(p.right, k-1)
   return ks + kd

>>> contaLiv(r,2)
2
>>> contaLiv(r,5)
0
>>> contaLiv(r,0)
1
```

N.B. il costo computazionale è:
Θ(numero di nodi che si trovano ad un livello $\leq k$)



Visita per livelli

E se volessimo accedere ai nodi per livelli, dalla radice in giù?



Nessuna delle visite ricorsive che abbiamo illustrato per gli alberi implementati mediante puntatori permette di farlo.

E' necessario utilizzare una coda d'appoggio, nella quale inserire opportunamente i nodi, estraendoli poi per visitarli.

Vogliamo stampare le chiavi dei nodi dell'albero visitandoli per livelli.

Per semplicità, supponiamo che l'implementazione della coda sia fatta in modo tale da inserire ed estrarre direttamente puntatori a nodi dell'albero.

```
def visitaPerLivelli(r)
   if r == None: return
   head,tail = inserisciCoda(None, None, r)
   while head!=None:
      p, head, tail = estraiCoda(head, tail)
      print(p.key)
      if p.left: head, tail =inserisciCoda(head, tail, p.left)
      if p.right: head, tail=inserisciCoda(head, tail, p.right)
```

Usiamo qui:

- una funzione di inserimento in coda che dati il puntatore alla testa, alla coda e ad un nodo inserisce il puntatore al nodo in coda e restituisce la testa e la coda:

  head,tail = inserisciCoda(head, tail, r)

- una funzione di estrazione dalla coda che data la testa e la coda restituisce il puntatore in testa alla coda e i puntatori alla testa ed alla coda.

  p, head, tail = estraiCoda(head, tail)

La caratteristica fondamentale di questa implementazione è che vengono inseriti nella coda tutti i nodi di livello *i* prima di inserire anche un solo nodo di livello *(i + 1)*.

Poiché l'ordine di estrazione è lo stesso (la coda è una struttura FIFO) e il lavoro sul nodo (la stampa) segue immediatamente l'estrazione, il risultato di scandire l'albero per livelli è raggiunto.

Il costo computazionale è $\Theta(n)$ perché per ognuno degli *n* nodi si effettuano:

- il suo inserimento in coda , costo $\Theta(1)$;
- la sua estrazione dalla coda, costo $\Theta(1)$;
- un numero costante di operazioni elementari, $\Theta(1)$.

Esercizio (1)

Scrivere la stampa in pre-ordine delle chiavi dei nodi di un albero binario dato tramite vettore dei padri P. Possiamo fruttare una funzione $TrovaFigli(P,i)$ che restituisce la coppia di indici fs e fd , $fs < fd$, dei nodi figli di sinistra e di destra di i (oppure $None$ se il figlio corrispondente non c'è).

```
def PreOrder(P, ind_r):
   print(V[r])
   fs,fd = TrovaFigli(P, ind_r)
   if fs: PreOrder(P,fs)
   if fd: PreOrder(P,fd)
```

Costo computazionale:

Osserviamo che `Trova_Figli(P, i)` ha un costo di $\Theta(n)$, indipendentemente da quale sia *i*, per cui $T(n)$ non decresce spostandosi verso i sottoalberi.

Per questo non è possibile scrivere l'equazione di ricorrenza (o meglio, non è possibile scriverla semplicemente come sappiamo fare).

Procediamo quindi con questo ragionamento:

- per ogni nodo si fa lavoro $\Theta(n)$, da cui $T(n) = \Theta(n^2)$.

Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Esercizi per casa

Esercizi proposti

- Scrivere lo pseudocodice ITERATIVO della visita in preordine.
- Calcolare il costo computazionale delle visite quando l'albero venga memorizzato tramite rappresentazione posizionale.
- Scrivere lo pseudocodice della funzione `Trova_Figli` usata nell'esercizio precedente.
- In quell'esercizio, se usassimo un vettore ausiliario in cui memorizzare in fase di pre-processing i figli di ciascun nodo, come diventerebbe lo pseudo-codice? ed il costo computazionale?