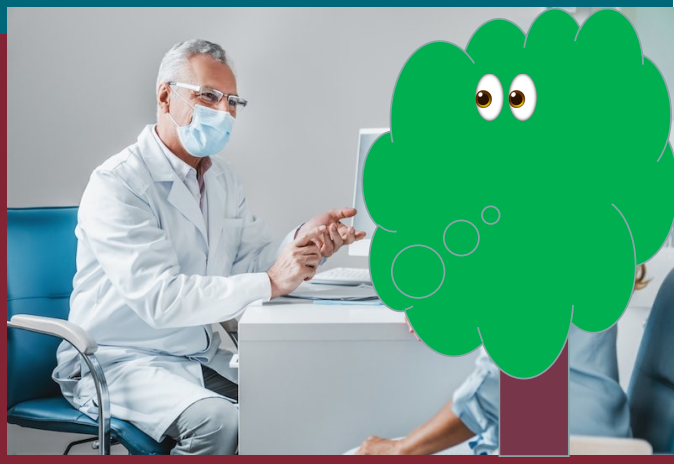
Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2024/25

Strutture dati fondamentali: Visite di Alberi



Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Visite di Alberi (1)

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.

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

Visite di Alberi (2)

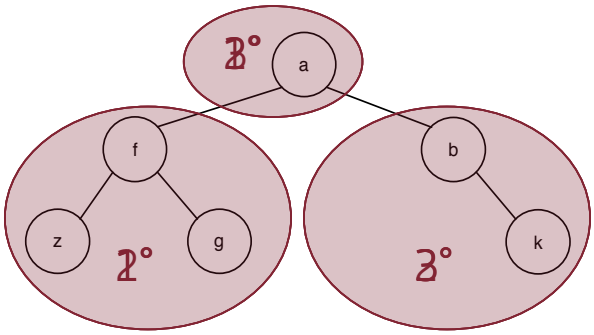
Facendo riferimento all'ordine col quale si accede ai nodi dell'albero, è evidente che esiste più di una possibilità.

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, abbiamo tre diverse visite:

Visite di Alberi (3)

- *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.

Visite di Alberi (4)



- 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**

Visite di Alberi (5)

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

```
def Visita_preordine (p):
   if (p ≠ None)
      accesso al nodo e operazioni conseguenti
      Visita_preordine (p->left)
      Visita_preordine (p->right)
   return
```

inordine

postordine

Le altre visite sono analoghe.
L'unica differenza fra i tre casi è la posizione dell'istruzione relativa all'accesso al nodo per effettuarvi le operazioni desiderate.

Visite di Alberi (6)

Costo computazionale

E' lo stesso per tutte e tre e varia con la struttura dati utilizzata per memorizzare l'albero.
Nel caso di record e puntatori, detto k il numero di nodi del sottoalbero sinistro della radice, l'equazione di ricorrenza è:

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

Non siamo in grado di risolverla se non con il metodo di sostituzione.
Facciamoci un'idea della possibile soluzione…

Visite di Alberi (7)

Costo computazionale – caso bilanciato

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:

$$T(n) = 2T(n/2) + \Theta(1)$$

che ricade nel caso 1 del teorema principale, ed ha quindi soluzione: $T(n) = \Theta(n^{\log 2}) = \Theta(n)$

Visite di Alberi (8)

Costo computazionale – caso sbilanciato

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

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

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

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

Visite di Alberi (9)

Costo computazionale – caso generale

Visto che le due soluzioni coincidono, possiamo ipotizzare che anche il costo computazionale nel caso generale sia anch'esso $\Theta(n)$.

Con questa idea di soluzione, risolviamo l'equazione con il metodo di sostituzione.

Visite di Alberi (10)

Costo computazionale delle visite – caso generale (segue)

Eliminiamo prima la notazione asintotica:

- $T(n)=T(k)+T(n-1-k)+c$
- $T(1)=d$

per due costanti positive c e d note.
Tentiamo la soluzione $T(n) \leq an$, dove a è una costante da determinare.

Passo base: $d = T(1) \leq a$ vera sse $\underline{a \geq d}$

…

Visite di Alberi (11)

Costo computazionale delle visite – caso generale (segue)

…

Passo induttivo: $T(n) \leq ak + a(n-1-k) + c =$

$= a(n-1) + c = an - a + c \leq an$ *vera* sse $\underline{a \geq c}$

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

Si dimostra analogamente che $T(n) \geq bn$, quindi che $T(n)=\Omega(n)$.

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

Visite di Alberi (12)

Costo computazionale delle visite – caso generale (segue)

Domanda: avremmo potuto scrivere l'equazione di ricorrenza in termini di h? In altre parole, è giusto scrivere:

- $T(h)=2T(h-1)+\Theta(1)$
- $T(1)=\Theta(1)$?

Questa equazione ha costo $T(h) = \Theta(2^h)$ ma non siamo in grado di mettere in relazione esatta n ed h in un albero binario qualsiasi; sappiamo solo che $h=O(n)$, che porterebbe a $T(n) = O(2^n)$, certamente corretto ma inutile!

Applicazioni delle visite (1)

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.

Applicazioni delle visite (2)

Esercizio: conteggio del numero dei nodi.

```
def Calcola_n (p):
   if (p ≠ None):
      num_l = Calcola_n (p->left)
      num_r = Calcola_n (p->right)
      num = num_l + num_r + 1      #accesso al nodo
      return num
   return 0
```

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

Al posto delle 4 istruzioni nell'if possiamo scrivere:

```
return 1+Calcola_n(p->left)+Calcola_n(p->right)
```


Applicazioni delle visite (3)

Esercizio: ricerca in un albero.

```
def Cerca (p):
   if (p ≠ None):
      if p->info== k return TRUE
      else:
         if Cerca(p->left,k)==TRUE:
                return TRUE
           else: return Cerca(p->right,k)
   return FALSE
```

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

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.*

Applicazioni delle visite (4)

Esercizio: calcolo dell'altezza dell'albero (stabiliamo che un albero costituito di un singolo nodo abbia altezza zero).

```
def Calcola_h (p):
   if (p==None): return -1          #albero vuoto
   if (p->left==None AND p->right==None): return 0
   h = max{Calcola_h(p->left),Calcola_h(p->right)}
   return h + 1
```

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

Applicazioni delle visite (5)

Esercizio: conteggio dei nodi presenti nel livello *k* (la radice è a livello 0).

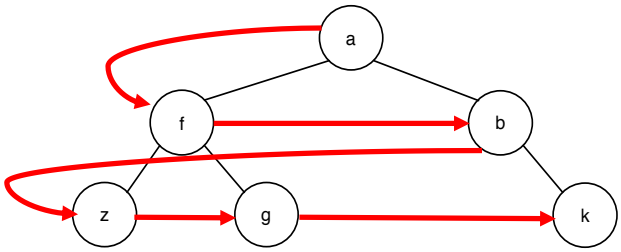
```
def Conta_k (p; k,i: interi):
   if (p == None): return 0          #albero vuoto
   if (k==i): return 1
   k_left = Conta_k(p->left,k,i+1)
   k_right = Conta_k(p->right,k,i+1)
   return k_left + k_right
```

La chiamata iniziale nel main() sarà `nk = Conta_k(radice,k,0)`

N.B. segue la filosofia della visita in postordine; il costo computazionale è Θ(numero di nodi che si trovano ad un livello $\leq$ k)

Visita per livelli (1)

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...

Visita per livelli (2)

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

La medesima tecnica, con alcuni ulteriori accorgimenti, permetterà (come vedrete il prossimo anno...) di realizzare un importante tipo di visita su grafi.

Visita per livelli (3)

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

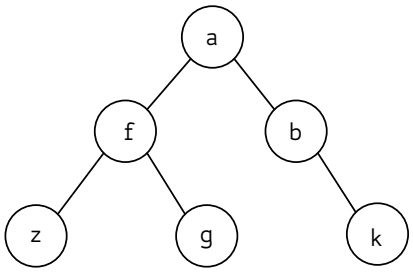
Esercizio: riscrivere le funzioni che operano sulla pila per consentire tale implementazione.

Visita per livelli (4)

Vogliamo stampare le chiavi dei nodi per livelli:

```
def Visita_per_livelli (r; head, tail):
   if (r==None): return
   Enqueue(head, tail, r)
   while (!CodaVuota(head))
      p = Dequeue(head, tail)
      print(p->key)
      if (p->left≠None):  Enqueue(head, tail, p->left)
      if (p->right≠None): Enqueue(head, tail, p->right)
   return
```

Visita per livelli (5) - esempio




```
def Visita_per_livelli (r; head, tail):
   if (r==None): return
   Enqueue(head, tail, r)
   while (!CodaVuota(head))
      p = Dequeue(head, tail)
      print (p->key)
      if (p->left≠None)
         Enqueue(head, tail, p->left)
      if (p->right≠None)
         Enqueue(head, tail, p->right)
return
```

Stampa: a f b z g k

Visita per livelli (6)

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.

Visita per livelli (7)

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

- una `Enqueue`, costo $\Theta(1)$;
- una `Dequeue`, costo $\Theta(1)$;
- un numero costante di operazioni elementari, costo $\Theta(1)$.

Esercizio svolto 1 (1)

Esercizio 1. Scrivere la visita in pre-ordine di un albero dato tramite vettore dei padri *P*, supponendo di avere una funzione `Trova_Figli(P, v, sx, dx)` che restituisce in sx e dx i figli di *v* (oppure -1 se il figlio non c'è).

Esercizio svolto 1 (2)

Soluzione.

```
def Pre_Order_Vett_Padri(P,ind_radice):
   sx = -1; dx = -1;
   #visita qui il nodo corrisp. a ind_radice;
   Trova_Figli(P, ind_radice, sx, dx;)
   if sx≠-1: Pre_Order_Vett_Padri(P,sx);
   if dx≠-1: Pre_Order_Vett_Padri(P,dx);
```


Esercizio svolto 1 (3)

Costo computazionale:

`Trova_Figli(P, v, sx, dx)` ha un costo di $\Theta(n)$, per qualsiasi v, per cui $T(n)$ non decresce spostandosi verso i sottoalberi.

Perciò non è possibile scrivere l'equazione di ricorrenza semplicemente come sappiamo fare...

Procediamo quindi con un ragionamento più intuitivo (che va evitato, se è possibile essere più rigorosi):

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

Esercizio svolto 2 (1)

Esercizio 2. Dato un albero binario non vuoto memorizzato tramite vettore dei padri, crearne uno identico memorizzato tramite notazione posizionale. Calcolare il costo computazionale.

Soluzione

Indichiamo con:

- A: array risultato, albero nella notaz. posizionale
- B: chiavi dei nodi nella memorizzazione con vettore dei padri
- P: padri dei nodi nella memorizzazione con vettore dei padri

Esercizio svolto 2 (2)

IDEA

- Prima di tutto identifichiamo la radice - funzione `Trova_radice()`:
- `Trova_radice(P)` effettua una scansione dell'array P dei padri e restituisce l'indice dell'unico elemento a None (o a -1), che è quello relativo alla radice.

Esercizio svolto 2 (3)

IDEA (segue)

- Poi, per ogni nodo in posizione `i`, identifichiamo i suoi figli e li sistemiamo nelle posizioni `2i` e `2i+1` tramite la ricorsione.
- Supponiamo che la funzione `Trova_figli()` restituisca due valori, **`left, right,`** con gli indici di ciascuno dei due figli (oppure None per ogni figlio che non esiste). Si assume per convenzione che il primo indice trovato da `Trova_figli()` sia quello del figlio sinistro.

Esercizio svolto 2 (4)

IDEA (segue)

- `Trova_figli(P;i)` cerca le posizioni dei figli del nodo collocato in posizione `i`. Effettua una scansione dell'array P da sinistra a destra:
  - Il primo indice j trovato (se esiste) per cui P[j] = `i` viene assegnato a `left` (se non viene trovato: None);
  - Il secondo indice k trovato (se esiste) per cui P[k] = `i` viene assegnato a `right` (se non viene trovato: None);

Esercizio svolto 2 (5)

P è il vettore dei padri; **ind** è l'indice del nodo di cui si vogliono trovare i figli.

```
def Trova_Figli(P; ind):
   num_figli= 0
   left=None
   right=None
   for i in range len(P):
      if (P[i]==ind)
         num_figli=num_figli+1
         if (num_figli == 1): left=i
            else: right=i
   return {left,right}
```

Costo computazionale: $\Theta(n)$

N.B. Si può facilmente modificare questa funzione perché funzioni in $O(n)$.

Esercizio svolto 2 (6)

IDEA (segue)

- `Sistema_Nodo()` è ricorsiva:
  - Copia un nodo dall'array delle chiavi (vettore dei padri, sottoarray B) all'array posizionale (A);
  - Chiama ricorsivamente se stessa sui due figli del nodo appena sistemato, quindi alla fine sistema i nodi di tutto l'albero.

Esercizio svolto 2 (7)

Nella funzione chiamante si avrà:

```
IndiceRadice =Trova_radice(P)
SistemaNodo(A,B,P,0,Indiceradice)
```

```
def Sistema_Nodo (A,B,P;indA,indB):
   if (indA ≤ n-1):
    A[indA]=B[indB]
    {left,right} = Trova_Figli(P,indB)
    if (left≠None): Sistema_Nodo(A,B,P,2*indA+1,left)
    if (right≠None): Sistema_Nodo(A,B,P,2*indA +2,right)
   return
```

- indA: si muove sull'array A (memorizz. posizionale)
- indB: si muove sugli array B e P (memorizz. vett. dei padri)

Esercizio svolto 2 (8)

Il costo computazionale è dato dalla somma dei costi di `Trova_radice()` e `Sistema_Nodo()`:

1. `Trova_radice()` ha banalmente costo $\Theta(n)$
2. `Sistema_Nodo()` è di fatto una visita, nella quale però il lavoro su ciascun nodo ha costo $\Theta(n)$. La visita fa tale lavoro sempre sull'intero array P per ciascun nodo => il suo costo è $T(n) = \Theta(n^2)$.

Quindi il costo computazionale complessivo è

$$T(n) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

N.B. visto che il lavoro sul nodo non decresce con n NON si può scrivere l'eq. di ricorrenza!

Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2024/25

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi proposti (1)

- Scrivere lo pseudocodice ITERATIVO della visita in preordine.
- Calcolare il costo computazionale delle visite quando l'albero venga memorizzato tramite rappresentazione posizionale.

Esercizi proposti (2)

- Calcolare il costo computazionale delle visite quando l'albero venga memorizzato tramite vettore dei padri (si può usare la funzione `Trova_Figli`).
- In questo esercizio, se usassimo un array ausiliario in cui memorizzare in fase di pre-processing i figli di ciascun nodo, come diventerebbe lo pseudo-codice? ed il costo computazionale?