```
22      FOR (i = m = 0; i < n; i = i + 1) {
23        IF (¦Py[i].x - p.x¦ <= d) {
24          Pd[m] = Py[i]; m = m+1;
25        }
26      }
27      FOR (i = 0; i < m; i = i + 1) {
28        FOR (j = i+1; j <= min{i+10, m}; j = j + 1) {
29          IF (Dist(Pd[i], Pd[j]) < d) {
30            d = Dist(Pd[i], Pd[j]); (p, q) = (Pd[i], Pd[j]);
31          }
32        }
33      }
34      RETURN (p, q);
35    }                                  ⟨post: la coppia (p,q) dei punti più vicini⟩
```

Si osservi che ogni passo dell'algoritmo ha un costo al più lineare quindi mettendo insieme l'Equazione (3.10), il Teorema 3.1 e il Lemma 3.1 otteniamo il seguente risultato.

Teorema 3.5 *L'algoritmo è corretto e la sua complessità è* $O(n \log n)$.

3.8 Algoritmi ricorsivi su alberi binari

Gli alberi binari, essendo definiti in modo ricorsivo, permettono di progettare naturalmente algoritmi ricorsivi seguendo la metodologia del divide et impera: nel discuterne alcuni esempi, introdurremo anche della terminologia aggiuntiva che, sebbene fornita per semplicità con riferimento agli alberi binari, è in generale applicabile anche ad alberi di tipo diverso.

Un parametro che caratterizza un albero è la sua **dimensione** n, data dal numero di nodi in esso contenuti: chiaramente, un albero di dimensione n ha esattamente n - 1 archi (che collegano un qualunque nodo diverso dalla radice al padre), come possiamo notare nella figura dell'Esempio 1.10 che ha dimensione 16 e contiene 15 archi. Osserviamo che la dimensione di un albero binario può essere definita ricorsivamente nel modo seguente: un albero vuoto ha dimensione 0, mentre la dimensione di un albero non vuoto è pari alla somma delle dimensioni dei suoi sottoalberi, incrementata di 1, per includere la radice. Il Codice 3.11 utilizza tale osservazione per realizzare un algoritmo che determina la dimensione di un albero binario e che si basa sul seguente schema di divide et impera.

Codice 3.11 Algoritmo ricorsivo per il calcolo della dimensione di un albero binario.

```
1  Dimensione( u ):
2    IF (u == null) {
3      RETURN 0;
4    } ELSE {
5      dimensioneSX = Dimensione( u.sx );
6      dimensioneDX = Dimensione( u.dx );
7      RETURN dimensioneSX + dimensioneDX + 1;
8    }
```

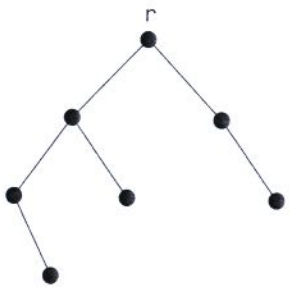
Decomposizione: se l'albero è vuoto, restituisci il valore `0` (riga 3), altrimenti suddividilo nei due sottoalberi radicati nei figli.

Ricorsione: calcola ricorsivamente la dimensione di ciascun sottoalbero (righe 5-6).

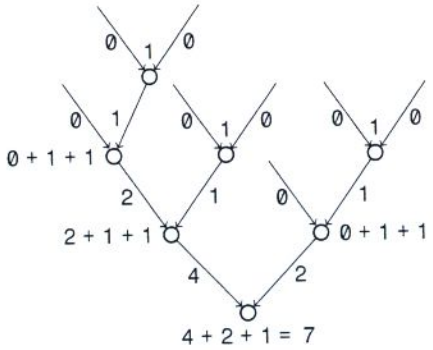
Ricombinazione: restituisci come risultato la somma delle dimensioni dei due sottoalberi incrementata di 1 (riga 7).

ESEMPIO 3.5

Eseguiamo l'algoritmo `Dimensione` sull'albero `T` nella figura avente radice `r`.



Nella figura che segue è rappresentato l'albero R delle invocazioni ricorsive di `Dimensione(r)`: i nodi di T e R sono in corrispondenza uno-a-uno, per comodità conviene usare gli stessi nomi per i nodi che sono in corrispondenza nei due alberi; ogni nodo u di R rappresenta la chiamata alla funzione `Dimensione(u)`. C'è un arco diretto tra u e v in R se `Dimensione(v)` ha invocato `Dimensione(u)`; l'etichetta sull'arco indica il valore restituito dalla funzione chiamata a quella chiamante. Infine l'etichetta di ogni nodo u in R rappresenta l'operazione di ricombinazione (riga 7) in cui viene calcolata la dimensione del sottoalbero di T con radice u sommando le dimensioni dei suoi due sottoalberi più uno.



La dimensione dell'albero T si legge nell'etichetta della radice dell'albero R.

Teorema 3.6 *La dimensione* n *di un albero binario può essere calcolata in tempo* O(n).

Dimostrazione Per dimostrare il Teorema 3.6, osserviamo che il problema della dimensione rientra nella famiglia dei problemi su alberi che possono essere risolti con uno schema di tipo divide et impera e che sono chiamati *problemi decomponibili*. La loro computazione ricalca il Codice 3.12, dove `Decomponibile(u)` rappresenta il valore da calcolare relativamente al sottoalbero radicato nel nodo u (che può essere una foglia oppure `null` nel caso base) e `Ricombina` permette di ricombinare i valori calcolati per i figli di u. Per esempio, nel calcolo della dimensione nel Codice 3.11, la funzione `Decomponibile(u)` rappresenta la dimensione del sottoalbero radicato in u e `Ricombina` rappresenta la somma incrementata di 1.

Codice 3.12 Algoritmo di tipo divide et impera per risolvere un problema decomponibile su alberi binari.

```
1  Decomponibile(u):
2    IF (u == null) {
3      RETURN Decomponibile(null);
4    } ELSE {
5      risultatoSX = Decomponibile(u.sx);
6      risultatoDx = Decomponibile(u.dx);
7      RETURN Ricombina(risultatoSX, risultatoDx);
8    }
```

Il vantaggio di avere uno schema di tipo divide et impera per gli alberi è che ci consente di scrivere la relazione di ricorrenza per il costo `T(n)`. Per semplicità, come accade in questo libro, ipotizziamo che il costo di divisione e ricombinazione per ogni nodo dell'albero sia limitato da una costante c. Preso un nodo u e il

suo sottoalbero con n nodi (incluso u), ipotizziamo di avere r - 1 nodi che discendono dal figlio sinistro di u e, quindi, n - r che discendono da quello destro, dove $1 \leq r \leq n$ e c_0 e c sono costanti positive:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ T(r-1) + T(n-r) + c & \text{altrimenti} \end{cases} \tag{3.11}$$

Teorema 3.7 *La relazione nella* (3.11) *ha soluzione* $T(n) = O(n)$.

Dimostrazione Non potendo applicare il teorema fondamentale, osserviamo che vale $T(0) \leq c_0 \leq c'$ e dimostriamo per induzione che $T(n) \leq 3c'n$ per ogni $n \geq 1$, dove $c' = \max\{c_0, c\}$. Se $n = 1$ (ovvero l'albero contiene un solo nodo), abbiamo che $r = 1$ e, quindi, $T(1) \leq 2T(0) + c \leq 2c_0 + c \leq 3c' = 3c'n$. Supponiamo che l'affermazione sia vera per $1 \leq n' < n$. Allora, $T(n) \leq T(r-1) + T(n-r) + c$ e, se $1 < r < n$, per ipotesi induttiva abbiamo che $T(n) \leq 3c'(r-1) + 3c'(n-r) + c \leq 3c'n - 2c < 3c'n$. Se invece $r = 1$, utilizziamo il fatto che $T(0) \leq c'$ e applichiamo l'ipotesi induttiva su $T(n-1) \leq 3c'(n-1)$, ottenendo $T(n) \leq c' + 3c'(n-1) + c \leq 3c'n - c < 3c'n$. Lo stesso ragionamento vale se $r = n$. In conclusione, $T(n) = O(n)$ e il teorema risulta essere dimostrato. ❑

Esercizio svolto 3.4 Ricordiamo che l'altezza di un albero misura la massima distanza di una foglia dalla radice dell'albero, in termini del numero di archi attraversati. Progettare un algoritmo ricorsivo che calcoli l'altezza di un albero binario in tempo $O(n)$, dove n denota la dimensione dell'albero.

Soluzione Osserviamo che l'albero composto da un solo nodo ha altezza pari a 0, mentre un albero con almeno due nodi ha altezza pari all'altezza del suo sottoalbero più alto, incrementata di 1 in quanto la radice introduce un ulteriore livello (da cui deriviamo che l'albero vuoto ha altezza pari a -1). Il seguente codice utilizza tale osservazione per realizzare un algoritmo che determina l'altezza di un albero.

```
Altezza( u ):
  IF (u == null) {
    RETURN -1;
  } ELSE {
    altezzaSX = Altezza( u.sx );
    altezzaDX = Altezza( u.dx );
    RETURN max( altezzaSX, altezzaDX ) + 1;
  }                          ⟨post: restituisce -1 se e solo se u è null⟩
```

Come si può notare, abbiamo usato l'accorgimento di considerare come caso base l'albero vuoto (a cui abbiamo assegnato un altezza pari a -1): in tal modo, il codice segue lo stesso schema del Codice 3.11, l'altezza calcolata per le foglie risulta correttamente pari a 0 (in quanto sottoalberi composti da un solo nodo) e, per induzione, è corretta anche l'altezza calcolata per tutti i sottoalberi.

Nello specifico, il codice precedente opera nel modo seguente: se l'albero è vuoto, la sua altezza è pari a -1. Se non lo è, le due chiamate ricorsive calcolano l'altezza dei sottoalberi radicati nei figli: di tali altezze viene restituita come risultato la massima incrementata di 1. L'analisi di complessità del codice ricalca quella del Codice 3.11 mostrata nella dimostrazione del Teorema 3.6.

Rimarchiamo che sia il Codice 3.11 che il codice dell'esercizio precedente hanno un caso base (albero vuoto) e un passo induttivo (albero non vuoto) in cui avvengono le chiamate ricorsive. A parte le differenze sintattiche dovute al fatto che i due codici calcolano quantità differenti, la struttura computazionale è quella dei problemi decomponibili (Codice 3.12): ciascuna invocazione restituisce un valore (la dimensione o l'altezza), che possiamo facilmente dedurre nel caso base di un albero vuoto. Nel passo induttivo, deleghiamo il calcolo delle rispettive quantità alla ricorsione sui due figli (sottoalberi): una successiva fase di combinazione di tali quantità, restituite dalle chiamate ricorsive sui figli, contribuisce a ottenere il risultato per il nodo corrente. Tale risultato va a sua volta restituito mediante l'istruzione `RETURN`, per far sì che l'induzione si propaghi attraverso la ricorsione: infatti, chi invoca le chiamate ricorsive deve a sua volta trasmettere il risultato così ottenuto. Notiamo che, in base a tale approccio, ogni nodo viene attraversato un numero costante di volte, per cui se il caso base e la regola di ricombinazione richiedono tempo costante, l'esecuzione richiede un tempo totale $O(n)$.

3.8.1 Visite di alberi

Lo schema ricorsivo del paradigma del divide et impera applicato ad alberi binari, permette anche di effettuare una **visita** di un albero binario a partire dalla sua radice. La visita equivale a esaminare tutti i nodi in modo sistematico, una e una sola volta, analogamente alla scansione di sequenze lineari, dove procediamo dall'inizio alla fine o viceversa. Per semplicità, durante la visita facciamo corrispondere l'esame di un nodo all'operazione di stampa del suo contenuto. Tale visita permette di operare varie scelte che dipendono dall'ordine in cui viene esaminato l'elemento memorizzato nel nodo corrente e vengono invocate le chiamate ricorsive nei suoi figli.

Visita anticipata (*preorder*): *stampa l'elemento contenuto nel nodo*; visita ricorsivamente il sottoalbero sinistro; visita ricorsivamente il sottoalbero destro.

Visita simmetrica (*inorder*): visita ricorsivamente il sottoalbero sinistro; *stampa l'elemento contenuto nel nodo*; visita ricorsivamente il sottoalbero destro.

Visita posticipata (*postorder*): visita ricorsivamente il sottoalbero sinistro; visita ricorsivamente il sottoalbero destro; *stampa l'elemento contenuto nel nodo.*

In modo analogo a quanto fatto nella dimostrazione del Teorema 3.6, possiamo dimostrare che il costo di ciascuna delle tre visite è $O(n)$ per un albero di di-

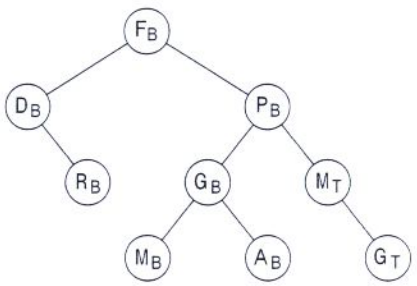
mensione n (cambia soltanto l'ordine in cui l'elemento nel nodo corrente viene stampato). Il codice per tali visite è una semplice variazione del Codice 3.11: per esempio, il Codice 3.13 realizza la visita anticipata. Osserviamo che esso non restituisce alcun valore in questa forma e che può essere trasformato nel codice di una visita simmetrica o posticipata molto semplicemente, spostando l'istruzione di stampa (riga 3).

Codice 3.13 Visita anticipata di un albero binario. Le altre due visite, simmetrica e posticipata, sono ottenute spostando l'istruzione di stampa dalla riga 3 in una delle due righe successive.

```
1  Anticipata( u ):
2    IF (u != null) {
3      print u.dato;
4      Anticipata( u.sx );
5      Anticipata( u.dx );
6    }
```

ESEMPIO 3.6

Per apprezzare la differenza delle tre visite, consideriamo l'esempio mostrato nella parte sinistra della seguente figura.



anticipata:
F_B D_B R_B P_B G_B M_B A_B M_T G_T
simmetrica:
D_B R_B F_B M_B G_B A_B P_B M_T G_T
posticipata:
R_B D_B M_B A_B G_B G_T M_T P_B F_B
ampiezza:
F_B D_B P_B R_B G_B M_T M_B A_B G_T

Nella parte destra della figura, oltre alle tre visite suddette viene illustrato anche il risultato di una quarta visita che illustreremo più avanti.

3.8.2 Alberi completamente bilanciati

Tornando allo schema del Codice 3.11, possiamo notare che esso rappresenta un modo di effettuare una visita posticipata in cui viene raccolta l'informazione necessaria alla computazione di `Dimensione(u)`, a partire dal basso verso l'alto. Per risolvere alcuni problemi su alberi binari, è necessario raccogliere più informazione di quanta ne serva apparentemente: studiamo, per esempio, il caso degli alberi completamente bilanciati.

A tale scopo, ricordiam
esattamente due figli nc
essere completo, tutte le
bilanciato di altezza h
relazione tra altezza h e
introdurre la definizion
relazione h = O(log n
operazioni fornite da di
bilanciato è bilanciato,

Volendo usare lo s
è completamente bila
il valore restituito sia
è completamente bila
come al solito con u_s
completamente bilanc
quanto i due sottoalbe
completamente bilanci
bilanciati, hanno anch

Nel Codice 3.14 ri
in cui il primo è TRUI
secondo è l'altezza d
regola di ricombinazi

- La prima compon
se lo sono le due p
CompletamenteB
componenti sono
- La seconda compo
simo tra le due sec
CompletamenteB

Codice 3.14 Algoritm

```
1  Completamente
2    IF (u == nu
3      RETURN <T
4    } ELSE {
5      <bilSX,al
6      <bilDX,al
7      completam
8      altezza =
9      RETURN <c
10   }     ⟨post: r
              bilanc
```

A tale scopo, ricordiamo che un albero binario è completo se ogni nodo interno ha esattamente due figli non vuoti. L'albero è **completamente bilanciato** se, oltre a essere completo, tutte le foglie hanno la stessa profondità. Un albero completamente bilanciato di altezza h ha quindi $2^h - 1$ nodi interni e 2^h foglie: ne deriva che la relazione tra altezza h e numero di nodi $n = 2^{h+1} - 1$ è $h = \log(n+1) - 1$. Possiamo introdurre la definizione di albero binario **bilanciato**: per un tale albero vale la relazione $h = O(\log n)$, che risulta essere interessante per la complessità delle operazioni fornite da diverse strutture di dati. Notiamo che un albero completamente bilanciato è bilanciato, mentre il viceversa non sempre vale.

Volendo usare lo schema del Codice 3.11 per stabilire se un albero binario è completamente bilanciato, possiamo valutare cosa succede ipotizzando che il valore restituito sia un valore booleano, che risulta TRUE se e solo se $T(u)$ è completamente bilanciato, dove $T(u)$ indica l'albero radicato in u. Indicati come al solito con u_S e con u_D i due figli di u, il fatto che $T(u_S)$ e $T(u_D)$ siano completamente bilanciati, non comporta purtroppo che anche $T(u)$ lo sia, in quanto i due sottoalberi potrebbero avere altezze diverse: in altre parole, $T(u)$ è completamente bilanciato se e solo se $T(u_S)$ e $T(u_D)$, oltre a essere completamente bilanciati, hanno anche la *stessa* altezza.

Nel Codice 3.14 richiediamo che il valore restituito sia una coppia di valori, in cui il primo è TRUE se e solo se $T(u)$ è completamente bilanciato, mentre il secondo è l'altezza di $T(u)$ (calcolata come nel codice dell'Esercizio 3.4). La regola di ricombinazione diventa quindi quella riportata di seguito.

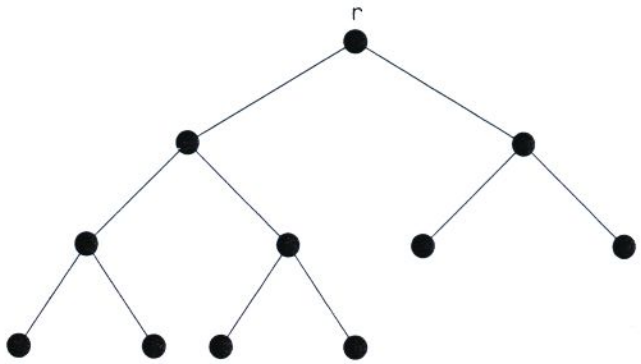
- La prima componente di `CompletamenteBilanciato(u)` è TRUE se e solo se lo sono le due prime componenti di `CompletamenteBilanciato(`u_S`)` e di `CompletamenteBilanciato(`u_D`)` sono entrambe TRUE e se le due seconde componenti sono uguali (riga 7).
- La seconda componente di `CompletamenteBilanciato(u)` è uguale al massimo tra le due seconde componenti di `CompletamenteBilanciato(`u_S`)` e di `CompletamenteBilanciato(`u_D`)` incrementate di 1 (riga 8).

Codice 3.14 Algoritmo ricorsivo per stabilire se un albero binario è completamente bilanciato.

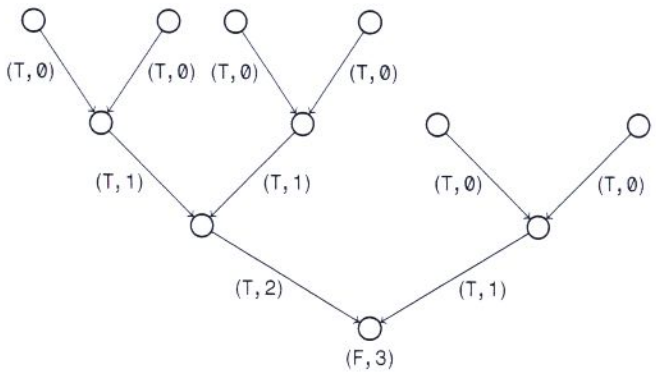
```
1  CompletamenteBilanciato( u ):
2    IF (u == null) {
3      RETURN <TRUE, -1>;
4    } ELSE {
5      <bilSX,altSX> = CompletamenteBilanciato( u.sx );
6      <bilDX,altDX> = CompletamenteBilanciato( u.dx );
7      completamenteBil = bilSX && bilDX && (altSX == altDX);
8      altezza = max(altSX, altDX) + 1;
9      RETURN <completamenteBil,altezza>;
10   }     ⟨post: restituisce TRUE come prima componente ⇔ T(u) è completamente
            bilanciato⟩
```

ESEMPIO 3.7

Eseguiamo l'algoritmo `CompletamenteBilanciato` sull'albero T, avente radice r, nella figura.



Come per l'Esempio 3.5 rappresentiamo R, l'albero delle chiamate ricorsive dell'algoritmo `CompletamenteBilanciato`. Anche in questo caso l'arco diretto tra u e v indica che `CompletamenteBilanciato(u)` restituisce il proprio output a `CompletamenteBilanciato(v)`: l'output è indicato come etichetta dell'arco da una coppia (B, h), dove B ∈ {T, F}, dove T sta per TRUE e F per FALSE.



L'algoritmo restituisce FALSE in quanto i sottoalberi sinistro e destro di r, pur essendo bilanciati, hanno altezze diverse.

ALVIE **Codice 3.15** Algoritmo ricorsivo per individuare i nodi cardine in un albero binario. La chiamata iniziale ha come parametri la radice e la sua profondità pari a 0.

```
1  Cardine( u, p ):                          ⟨pre: p è la profondità di u⟩
2    IF (u == null) {
3      RETURN -1;
4    } ELSE {
5      altezzaSX = Cardine( u.sx, p+1 );
6      altezzaDX = Cardine( u.dx, p+1 );
```

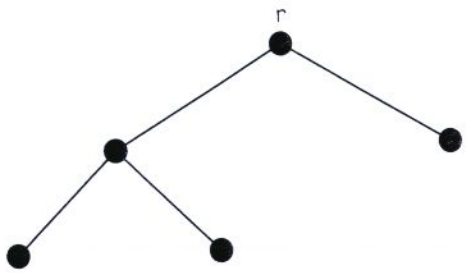
```
    altezza = max( altezzaSX, altezzaDX ) + 1;
    IF (p == altezza) print u.dato;
    RETURN altezza;
}                                    ⟨post: stampa i nodi cardine di T(u)⟩
```

3.8.3 Nodi cardine di un albero binario

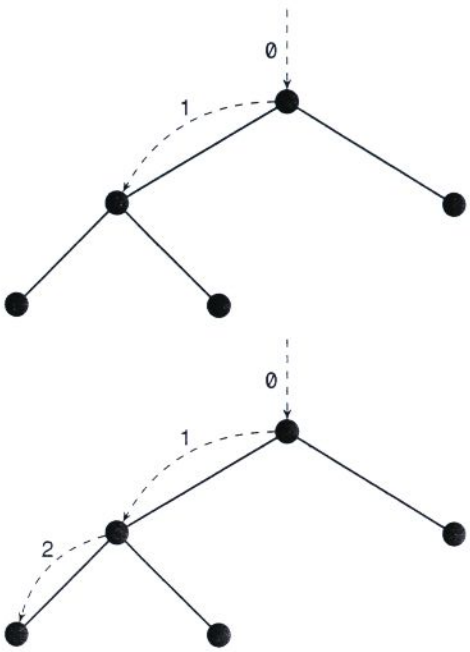
Per completare il quadro dello schema generale di sviluppo di algoritmi ricorsivi su alberi binari descritto in questo paragrafo, discutiamo un algoritmo in cui le chiamate non solo raccolgono informazione dai sottoalberi, ma propagano simultaneamente informazione proveniente dagli antenati, passando opportuni parametri alle chiamate. Un problema di questo tipo riguarda l'identificazione dei nodi cardine. Dato un nodo u, sia p_u la sua profondità e h_u l'altezza di T(u). Diciamo che u è un nodo **cardine** se e solo se $p_u = h_u$. Vogliamo progettare un algoritmo ricorsivo che stampi il contenuto di tutti i nodi cardine presenti in un albero binario.

In questo caso, possiamo presumere che il valore restituito dalla chiamata ricorsiva sia h_u, analogamente a quanto fatto nel codice dell'Esercizio 3.4: tuttavia, al momento di invocare la chiamata ricorsiva su u dobbiamo garantire di passare p_u come parametro. Il Codice 3.15 ha quindi due parametri in ingresso per questo scopo: il primo indica il nodo corrente e il secondo la sua profondità. Inizialmente, questi parametri sono la radice r dell'albero e la sua profondità $p_r = 0$. Le successive chiamate ricorsive provvedono a passare i parametri richiesti (righe 5 e 6): ovvero, se il nodo corrente ha profondità p, i figli avranno profondità p + 1. La verifica che la profondità sia uguale all'altezza nella riga 8 stabilisce infine se il nodo corrente è un nodo cardine: in tal caso, la sua informazione viene stampata. Da notare che la complessità temporale dell'algoritmo rimane O(n) in quanto si tratta di una semplice variazione della visita posticipata implicitamente adottata nel Codice 3.11.

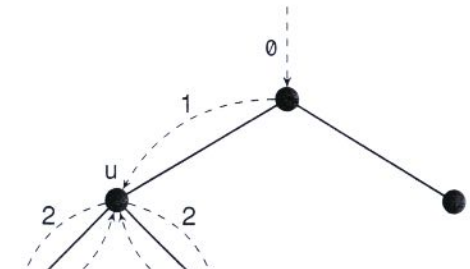
Eseguiamo l'algoritmo Cardine con input il nodo r radice dell'albero rappresentato nella figura sottostante.



Rappresentiamo l'invocazione di una chiamata ricorsiva su un nodo u con un arco tratteggiato orientato verso le foglie. L'etichetta dell'arco rappresenta la profondità del nodo u. Ovviamente la prima invocazione è sulla coppia (r, 0).



Arrivati alle foglie inizia il processo di ricostruzione delle altezze dei sottoalberi in quanto per le foglie questo valore è noto.



L'istanza della funzione eseguita con input (u, 1), dopo che le chiamate ricorsive sui nodi figli hanno restituito le altezze di questi (entrambi 0), calcola l'altezza di T(u) (riga 7) e poiché questa risulta uguale alla profondità di u ricevuta in input, stampa il dato contenuto nel nodo. Il procedimento continuerà per tutti gli altri nodi dell'albero.