

Paolo Enrico Camurati Stefano Quer

Algoritmi e programmazione

Richiami di teoria con prove d'esame ed esercizi svolti

CLUT

I diritti di elaborazione, di traduzione o l'adattamento anche parziale in qualsiasi forma, di memorizzazione anche digitale, su supporti di qualsiasi tipo, di riproduzione e di adattamento totale o parziale con qualsiasi mezzo (compresi i microfilm e le copie fotostatiche) sono riservati per tutti i Paesi.

Fotocopie per uso personale (cioè provato ed individuale) nei limiti del 15% di ciascun volume possono essere effettuate negli esercizi che aderiscono all'accordo S.I.A.E. – S.N.S. e C.N.A. Confartigianato, C.A.S.A., Confcommercio del 18 Dicembre 2000, dietro pagamento del compenso previsto per tale accordo, conformemente alla legge n. 633 del 23.04.1941.

Per riproduzioni ad uso non personale l'Editore potrà concedere a pagamento l'autorizzazione a riprodurre un numero di pagine non superiore al 15% delle pagine del presente volume. Le richieste per tale tipo di riproduzione vanno inoltrate esclusivamente all'indirizzo dell'Editore.

La messa a punto di un libro è un' operazione complessa ed articolata, che necessita di studi, progettualità grafica, nonché di numerosi controlli di testo, immagine, stili grafici e di stampa. E' praticamente impossibile pubblicare un libro scevra da errori. La C.L.U.T. ringrazia sin d'ora i lettori che vorranno segnalare all'indirizzo dell'Editore eventuali errori riscontrati nella lettura del libro.

**L'eventuale errata corrigé aggiornata del presente Volume è disponibile on-line all'indirizzo:
<http://www.clut.it> nella pagina dedicata al libro.**

Ideazione e disegno copertina a cura di Andrea Ruffino

© 2013 C.L.U.T. Editrice
Proprietà letteraria riservata
Stampato in Italia da STAMPATRE -- Torino
Copyright C.L.U.T – Torino – Settembre 2013

ISBN 978-88-7992-346-0

Edizioni C.L.U.T. – Torino
Corso Duca degli Abruzzi 24 – 10129 Torino
tel. 011 090 79 80 - tel. e fax 011 54 21 92
e-mail: clut@inrete.it - www.clut.it

Presentazione

Questa pubblicazione raccoglie, organizza e risolve numerosi esercizi di teoria che coprono gli argomenti trattati nel Corso di “Algoritmi e programmazione”.

Ogni capitolo del volume è strutturato come segue:

- ▷ richiami dei fondamenti teorici
- ▷ svolgimento completo di almeno un esercizio significativo per argomento
- ▷ esercizi con soluzione, ma senza procedimento
- ▷ proposta di esercizi non risolti, che si pensa costituiscano un utile stimolo al lavoro individuale.

La suddivisione in capitoli segue lo schema successivo.

- ▷ Capitolo 1: algoritmi iterativi di ordinamento interno
- ▷ Capitolo 2: algoritmi ricorsivi di ordinamento interno
- ▷ Capitolo 3: analisi della complessità mediante le equazioni alle ricorrenze
- ▷ Capitolo 4: heap, heap sort e code a priorità
- ▷ Capitolo 5: alberi binari
- ▷ Capitolo 6: alberi binari di ricerca e loro estensioni
- ▷ Capitolo 7: tabelle di hash
- ▷ Capitolo 8: risoluzione dei problemi con il paradigma greedy
- ▷ Capitolo 9: visite dei grafi e loro applicazioni
- ▷ Capitolo 10: alberi ricoprenti minimi
- ▷ Capitolo 11: cammini minimi da una singola sorgente
- ▷ Capitolo 12: temi d'esame di teoria risolti.

I primi undici capitoli, quindi, presentano tutti argomenti specifici, suddivisi in maniera opportuna, riassumendone gli aspetti teorici e riportandone numerosi esercizi. Il capitolo dodici invece include tutti gli argomenti analizzati nei capitoli precedenti, includendo ogni tipologia di esercizio in maniera ortogonale, così come accade abitualmente nei temi d'esame del corso di “Algoritmi e programmazione” del Politecnico di Torino.

Argomenti e esercizi, sono stati selezionati in base alla lunga esperienza degli autori, da anni coinvolti in corsi di programmazione, algoritmi e strutture dati di base e

avanzati.

Chiunque rintracciasse errori sfuggiti agli autori è pregato di segnalarlo per posta elettronica agli autori stessi, utilizzando i seguenti indirizzi:

paolo.camurati@polito.it
stefano.quer@polito.it

Indice

1 Algoritmi iterativi di ordinamento interno	1
1.1 Insertion sort	1
1.1.1 Richiami di teoria	1
1.1.2 Esercizi svolti	2
1.1.3 Esercizi proposti	3
1.2 Exchange sort (o BUBBLE SORT)	3
1.2.1 Richiami di teoria	3
1.2.2 Esercizi svolti	4
1.2.3 Esercizi proposti	4
1.3 Selection sort	5
1.3.1 Richiami di teoria	5
1.3.2 Esercizi svolti	6
1.3.3 Esercizi proposti	6
1.4 Shell sort	7
1.4.1 Richiami di teoria	7
1.4.2 Esercizi svolti	8
1.4.3 Esercizi proposti	8
1.5 Counting sort	9
1.5.1 Richiami di teoria	9
1.5.2 Esercizi svolti	10
1.5.3 Esercizi risolti	13
1.5.4 Esercizi proposti	15
2 Algoritmi ricorsivi di ordinamento interno	17
2.1 Merge sort	17
2.1.1 Richiami di teoria	17
2.1.2 Esercizi svolti	18
2.1.3 Esercizi proposti	20
2.2 Quick sort	20
2.2.1 Richiami di teoria	20
2.2.2 Esercizi svolti	21

2.2.3	Esercizi risolti	26
2.2.4	Esercizi proposti	27
3	Equazioni alle ricorrenze	29
3.1	Metodo dello sviluppo (iterazione o unfolding)	29
3.1.1	Richiami di teoria	29
3.1.2	Esercizi svolti	30
3.1.3	Esercizi risolti	33
3.1.4	Esercizi proposti	34
4	Heap, heap sort e code a priorità	37
4.1	Heap	37
4.1.1	Richiami di teoria	37
4.2	Heap sort	38
4.2.1	Richiami di teoria	38
4.2.2	Esercizi svolti	39
4.2.3	Esercizi risolti	41
4.2.4	Esercizi proposti	41
4.3	Code a priorità	42
4.3.1	Richiami di teoria	42
4.3.2	Esercizi svolti	44
4.3.3	Esercizi risolti	44
4.3.4	Esercizi proposti	47
5	Alberi binari	49
5.1	Visita di alberi binari	49
5.1.1	Richiami di teoria	49
5.1.2	Esercizi svolti	50
5.1.3	Esercizi risolti	51
5.1.4	Esercizi proposti	53
5.2	Espressioni aritmetiche	54
5.2.1	Richiami di teoria	54
5.2.2	Esercizi svolti	55
5.2.3	Esercizi risolti	56
5.2.4	Esercizi proposti	57
6	Alberi binari di ricerca	59
6.1	Correttezza della struttura dati	59
6.1.1	Richiami di teoria	59
6.1.2	Esercizi svolti	60
6.1.3	Esercizi risolti	61
6.1.4	Esercizi proposti	61
6.2	Operazioni sui BST	62
6.2.1	Richiami di teoria	62
6.2.2	Esercizi svolti	68
6.2.3	Esercizi proposti	74
7	Le tabelle di hash	79

7.1	Hashing	79
7.1.1	Richiami di teoria	79
7.2	Linear Chaining	80
7.2.1	Richiami di teoria	80
7.2.2	Esercizi svolti	80
7.2.3	Esercizi risolti	81
7.2.4	Esercizi proposti	82
7.3	Open addressing con linear probing	83
7.3.1	Richiami di teoria	83
7.3.2	Esercizi svolti	83
7.3.3	Esercizi risolti	84
7.3.4	Esercizi proposti	85
7.4	Open addressing con quadratic probing	86
7.4.1	Richiami di teoria	86
7.4.2	Esercizi svolti	86
7.4.3	Esercizi risolti	87
7.4.4	Esercizi proposti	88
7.5	Open addressing con double hashing	89
7.5.1	Richiami di teoria	89
7.5.2	Esercizi svolti	89
7.5.3	Esercizi risolti	91
7.5.4	Esercizi proposti	92
8	Il paradigma greedy	95
8.1	Selezione di attività	95
8.1.1	Richiami di teoria	95
8.1.2	Esercizi svolti	96
8.1.3	Esercizi risolti	97
8.1.4	Esercizi proposti	98
8.2	Codici di Huffman	98
8.2.1	Richiami di teoria	98
8.2.2	Esercizi svolti	100
8.2.3	Esercizi risolti	103
8.2.4	Esercizi proposti	104
9	Le visite dei grafi e le loro applicazioni	105
9.1	Rappresentazione dei grafi	105
9.1.1	Richiami di teoria	105
9.1.2	Esercizi svolti	106
9.2	Visita in ampiezza (BFS, Breadth-First Search)	107
9.2.1	Richiami di teoria	107
9.2.2	Esercizi svolti	108
9.3	Visita in profondità (DFS, Depth-First Search)	110
9.3.1	Richiami di teoria	110
9.3.2	Esercizi svolti	112
9.4	Applicazioni della visita in profondità: le componenti connesse	114
9.4.1	Richiami di teoria	114
9.4.2	Esercizi svolti	115

9.5	Applicazioni della visita in profondità: le componenti fortemente connesse	116
9.5.1	Richiami di teoria	116
9.5.2	Esercizi svolti	116
9.6	Applicazioni della visita in profondità: i punti di articolazione	118
9.6.1	Richiami di teoria	118
9.6.2	Esercizi svolti	119
9.7	Applicazioni della visita in profondità: l'ordinamento topologico dei DAG	119
9.7.1	Richiami di teoria	119
9.7.2	Esercizi svolti	121
9.8	Esercizi risolti	123
9.9	Esercizi proposti	132
10	Gli alberi ricoprenti minimi	137
10.1	On-line Connectivity e algoritmi Union-Find	137
10.1.1	Richiami di teoria	137
10.1.2	Esercizi svolti	139
10.1.3	Esercizi risolti	140
10.1.4	Esercizi proposti	142
10.2	Alberi ricoprenti minimi	143
10.2.1	Richiami di teoria	143
10.2.2	Esercizi svolti	144
10.2.3	Esercizi risolti	148
10.2.4	Esercizi proposti	154
11	I cammini minimi da una singola sorgente	157
11.1	Richiami di teoria	157
11.1.1	L'algoritmo di Dijkstra	157
11.1.2	L'algoritmo per DAG pesati	158
11.1.3	L'algoritmo di Bellman-Ford	158
11.2	Esercizi svolti	159
11.3	Esercizi risolti	163
11.4	Esercizi proposti	169
12	Soluzione di temi d'esame	173
12.1	Tema d'esame 01	173
12.2	Tema d'esame 02	178
12.3	Tema d'esame 03	182
12.4	Tema d'esame 04	189
12.5	Tema d'esame 05	192
12.6	Tema d'esame 06	200
12.7	Tema d'esame 07	204
12.8	Tema d'esame 08	207
12.9	Tema d'esame 09	216
12.10	Tema d'esame 10	219
12.11	Tema d'esame 11	227

```

for (i=l; i<=r & trovato==false; i++)
    if (K==A[i])
        trovato=true;
if (trovato==false)
    return -1;
else
}

```

Capitolo 1

```

while (l<=r)
{
    m=(l+r)/2;
    if (A[m]==k)
        return m;
    if (A[m]<k)
        l=m+1;
    else // A[m]>k
        r=m-1;
}
return -1;

```

Algoritmi iterativi di ordinamento interno

Scopo di questo capitolo è quello di presentare gli algoritmi iterativi di ordinamento interno. Si dicono interni gli algoritmi che operano su dati contenuti in memoria centrale, esterni se operano su dati contenuti in memoria di massa. Si tratteranno gli algoritmi di complessità quadratica basati sul confronto, quali l'insertion sort, l'exchange (o bubble) sort e il selection sort. Verrà inoltre presentato il counting sort, un algoritmo di complessità lineare non basato sul confronto. Si tratterà infine lo shell sort, la cui complessità dipende dalla scelta di una sequenza.

Il problema dell'ordinamento può essere definito come segue. Dato in ingresso un sottoinsieme

$$< a_1, a_2, \dots, a_n >$$

di un insieme di simboli su cui è definita una relazione d'ordine totale (ad esempio \leq) trovare quella permutazione

$$< a'_1, a'_2, \dots, a'_n >$$

dei valori di ingresso per cui vale la relazione d'ordine¹

$$a'_1 \leq a'_2 \leq \dots \leq a'_n$$

Si dice *in loco* un algoritmo di ordinamento che utilizza un vettore di n dati e un numero di locazioni ausiliarie di memoria fisso.

Si dice *stabile* un algoritmo di ordinamento che mantiene invariato in uscita l'ordinamento relativo in ingresso di dati con la stessa chiave.

Per semplicità considereremo in questo capitolo come struttura dati un vettore di n valori interi.

1.1 Insertion sort COMPLESSITÀ: n^2 .

1.1.1 Richiami di teoria

L'insertion sort è un algoritmo iterativo di ordinamento interno di complessità quadratica basato sul confronto. La struttura dati è un vettore **A** di interi con indici compresi

¹ Si osservi che considerazioni analoghe valgono per ordinamenti decrescenti.

tra 1 e r. Concettualmente il vettore **A** è suddiviso in due sotto-vettori:

- quello di sinistra: ordinato, inizialmente **A[1]**
 - quello di destra: disordinato inizialmente tutto il resto del vettore.

Un vettore di un solo elemento è ordinato.

L'algoritmo segue un approccio incrementale. Al passo i -esimo si espande il sotto-vettore ordinato inserendovi l'elemento $x = A[i]$ effettuando le seguenti operazioni:

- ▷ scansione del sotto-vettore ordinato (da $A[i-1]$ a $A[1]$) fino a quando diventa vera la condizione $A[j] > A[i]$ con contemporaneo scalamento a destra di una posizione degli elementi da $A[j]$ a $A[i-1]$.
 - ▷ inserimento di $A[i]$ nella posizione corretta.

La terminazione dell'algoritmo si ha quando tutti gli elementi sono stati inseriti ordinatamente.

Una possibile implementazione dell'algoritmo è la seguente:

```

1 void insertionSort (int *A, int r) {
2     int i, j, x;
3
4     for (i=1; i<=r; i++) {
5         x = A[i];
6         j = i - 1;
7         while (j>=1 && x<A[j]) {
8             A[j+1] = A[j];
9             j--;
10        }
11        A[j+1] = x;
12    }
13    return;
14 }
15 }
```

void InsertionSort (int A[], int r)

{ int i, j, x;

for (i=1; i<=r; i++)

{ x = A[i];

for (j=i-1; j>=1 && x<A[j]; j--) { A[j+1] = A[j]; }

A[j+1] = x;

Si osservi che l'insertion sort è stabile e in loco.

1.1.2 Esercizi svolti

Esercizio

Si ordini in maniera ascendente la seguente sequenza di interi mediante insertion sort.

4 2 6 3 1 5

Si indichino i passaggi principali.

Soluzione

La Figura 1.1 fornisce una rappresentazione grafica dei passi dell'algoritmo. L'elemento evidenziato rappresenta quello selezionato (ed estratto) alla riga 5 del codice precedentemente riportato. Le frecce verso destra indicano le operazioni di scalamento (shift) eseguite dal ciclo alle righe $7 \div 10$. La freccia verso sinistra indica l'inserimento finale dell'elemento **x** eseguito alla riga 11.

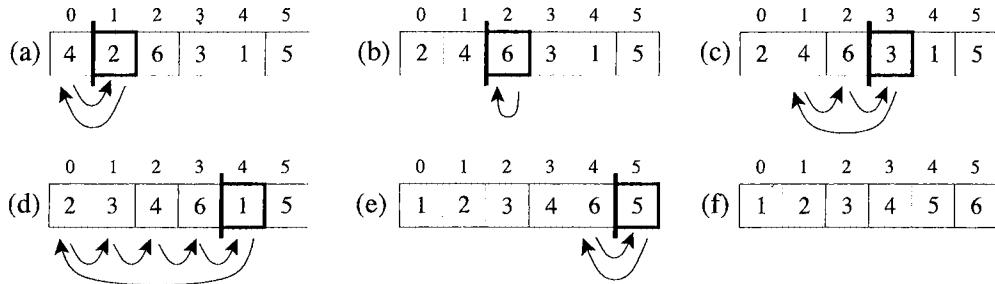


Figura 1.1 Ordinamento mediante inserzione.

1.1.3 Esercizi proposti

Esercizio

Si ordini in maniera ascendente la seguente sequenza di interi mediante insertion sort.

12 1 34 4 5 7 0 9 17 25 6 8 10

Si indichino i passaggi principali.

Esercizio

Si ordini in maniera discendente la seguente sequenza di interi mediante insertion sort.

2 11 17 3 9 4 17 15 6 10 28

Si indichino i passaggi principali.

1.2 Exchange sort (BUBBLE SORT)

1.2.1 Richiami di teoria

L'exchange sort (normalmente denominato bubble sort) è un algoritmo iterativo di ordinamento interno basato sul confronto di complessità quadratica. La struttura dati è un vettore \mathbf{A} di interi con indici compresi tra 1 e r . Concettualmente il vettore \mathbf{A} è suddiviso in due sotto-vettori:

- ▷ quello di sinistra: ordinato, inizialmente vuoto
- ▷ quello di destra: disordinato, inizialmente coincidente con \mathbf{A} .

L'operazione elementare consiste nel confronto tra elementi successivi del vettore $\mathbf{A}[j]$ e $\mathbf{A}[j+1]$, con il loro eventuale scambio nel caso in cui si abbia

$$\mathbf{A}[j] > \mathbf{A}[j + 1]$$

L'algoritmo segue un approccio incrementale: all'iterazione i -esima si assegna a $\mathbf{A}[r-i]$ il massimo del sotto-vettore sinistro $\mathbf{A}[1], \dots, \mathbf{A}[r-i]$ e si incrementa i .

Il sotto-vettore destro ordinato cresce di una posizione verso sinistra. In maniera duale quello destro decresce di una posizione a destra.

La condizione di terminazione si verifica quando il sotto-vettore sinistro coincide con il vettore A, e quello destro è vuoto.

Una possibile implementazione dell'algoritmo è la seguente:

```

1 void exchangeSort (int *A, int r) {
2     int i, j, tmp;
3
4     for (i=1; i<r; i++) {
5         for (j=0; j<r-i; j++) {
6             if (A[j] > A[j+1]) {
7                 tmp = A[j];
8                 A[j] = A[j+1];
9                 A[j+1] = tmp;
10            }
11        }
12    }
13    return;
14 }
15 }
```

```

void BubbleSort (int A[], int l, int r)
{
    int i, j, tmp;
    for (i=l; i<r; i++)
        for (j=l; j<r-i; j++)
            if (A[j] > A[j+1])
                {tmp = A[j];
                 A[j] = A[j+1];
                 A[j+1] = tmp;
                }
}
```

Sono possibili ottimizzazioni di vario tipo. Ad esempio è possibile utilizzare un flag per indicare se si sono effettuati degli scambi al passo precedente, terminando anticipatamente in caso negativo.

Il bubble sort è stabile e in loco.

1.2.2 Esercizi svolti

Esercizio

Si ordini in maniera ascendente la seguente sequenza di interi mediante bubble sort.

4 2 6 3 1 5

Si indichino i passaggi principali.

Soluzione

La Figura 1.2 riporta una rappresentazione grafica del procedimento del bubble sort. Le frecce indicano gli scambi effettuati alle righe 7 ÷ 9, dal codice precedentemente riportato, durante l'esecuzione del ciclo interno (righe 5 ÷ 11). La barra delimitatrice indica la separazione tra il sotto-vettore ordinato (a destra e inizialmente vuoto) e quello disordinato (a sinistra e inizialmente coincidente con l'intero vettore A).

1.2.3 Esercizi proposti

Esercizio

Si ordini in maniera ascendente la seguente sequenza di interi mediante bubble sort.

12 1 34 4 5 7 0 9 17 25 6 8 10

Si indichino i passaggi principali.

Esercizio

Si ordini in maniera discendente la seguente sequenza di interi mediante bubble sort.

2 11 17 3 9 4 17 15 6 10 28

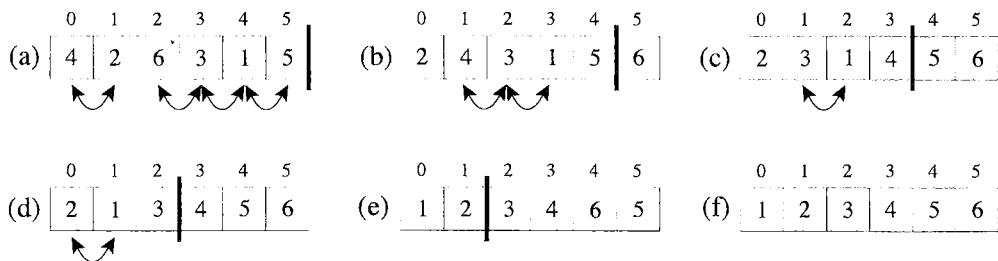


Figura 1.2 Ordinamento a bolla.

Si indichino i passaggi principali.

1.3 Selection sort

1.3.1 Richiami di teoria

Il selection sort è un algoritmo iterativo di ordinamento interno basato sul confronto di complessità quadratica. La struttura dati è un vettore \mathbf{A} di interi con indici compresi tra 1 e r .

Concettualmente il vettore \mathbf{A} è suddiviso in due sotto-vettori:

- ▷ quello di sinistra: ordinato, inizialmente vuoto
- ▷ quello di destra: disordinato, inizialmente coincidente con \mathbf{A} .

L'algoritmo segue un approccio incrementale: all'iterazione i -esima il minimo del sotto-vettore destro $\mathbf{A}[i], \dots, \mathbf{A}[r]$ è assegnato a $\mathbf{A}[i]$. Il valore di i viene incrementato. La ricerca del minimo nel sotto-vettore destro comporta la sua scansione.

Il sotto-vettore sinistro ordinato cresce di una posizione verso destra. In maniera duale quello destro decresce di una posizione.

La condizione di terminazione si verifica quando il sotto-vettore sinistro coincide con il vettore \mathbf{A} e quello destro è vuoto.

Una possibile implementazione dell'algoritmo è la seguente:

```

1 void selectionSort (int *A, int r) {
2     int i, j, min, tmp;
3
4     for (i=1; i<r; i++) {
5         min = i;
6         for (j=i+1; j<=r; j++) {
7             if (A[j]<A[min])
8                 min = j;
9         }
10        tmp = A[i];
11        A[i] = A[min];
12        A[min] = tmp;
13    }
14    return;
15 }
```

Il selection sort è stabile e in loco.

```

void SelectionSort(int A[], int l, int r)
{
    int i, min, tmp;
    for (i = l; i < r; i++) {
        min = i;
        for (j = i + 1; j <= r; j++)
            if (A[j] < A[min])
                min = j;
        tmp = A[i];
        A[i] = A[min];
        A[min] = tmp;
    }
}
```

1.3.2 Esercizi svolti

Esercizio

Si ordini in maniera ascendente la seguente sequenza di interi mediante selection sort.

4 2 6 3 1 5

Si indichino i passaggi principali.

Soluzione

La Figura 1.3 fornisce una rappresentazione grafica dei passi intermedi dell'algoritmo di ordinamento per selezione. Il minimo del sotto-vettore destro, selezionato dal ciclo delle

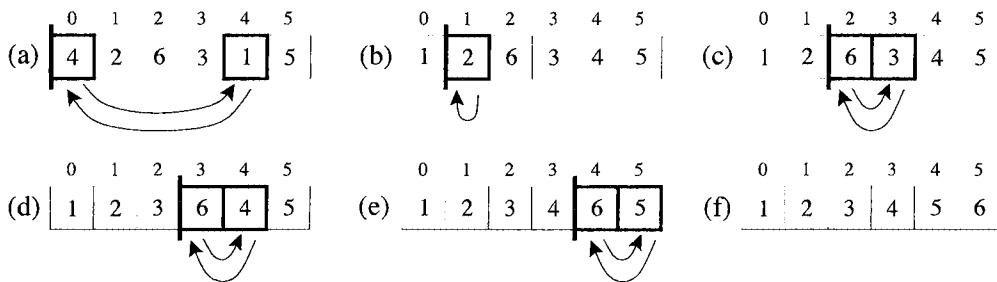


Figura 1.3 Ordinamento mediante selezione.

righe 3÷6 del codice precedente, è rappresentato riquadrato. La stessa rappresentazione viene utilizzata per l'elemento $A[i]$ scambiato con l'elemento minimo tmp alle righe 7 ÷ 9.

1.3.3 Esercizi proposti

Esercizio

Si ordini in maniera ascendente la seguente sequenza di interi mediante selection sort.

12 1 34 4 5 7 0 9 17 25 6 8 10

Si indichino i passaggi principali.

Esercizio

Si ordini in maniera discendente la seguente sequenza di interi mediante selection sort.

2 11 17 3 9 4 17 15 6 10 28

Si indichino i passaggi principali.

1.4 Shell sort

1.4.1 Richiami di teoria

Lo shell sort è un algoritmo iterativo di ordinamento interno basato sul confronto. L'insertion sort ha un evidente limite: il confronto, quindi lo scambio, avviene solo tra elementi adiacenti. Lo shell sort invece confronta, quindi eventualmente scambia, elementi a distanza h tra di loro definendo una sequenza decrescente di interi h che termina con 1.

La struttura dati coinvolta è un vettore A di interi con indici compresi tra 1 e r . Il numero di dati n è facilmente calcolabile come $n = r - l + 1$.

Per semplicità si considera la sequenza di Knuth 1, 4, 13, 40, 121, facilmente calcolabile a partire da $h = 1$ come $h = 3h + 1$.

Si dice h -ordinato un vettore formato da h sotto-sequenze non contigue ordinate composte da elementi che distano tra di loro h .

Ad esempio, il seguente vettore A

$$3 \ 0 \ 1 \ 0 \ 4 \ 6 \ 2 \ 1 \ 5 \ 7 \ 3 \ 4 \ 5 \ 7 \ 6 \ 8 \ 9 \ 8 \ 9$$

per $h = 4$ è suddiviso nelle seguenti 4 sotto-sequenze non contigue e ordinate, identificate dai numeri romani I , II , III e IV :

$$\left| \begin{array}{cccc} 3 & 0 & 1 & 0 \\ I & II & III & IV \end{array} \right| \left| \begin{array}{cccc} 4 & 6 & 2 & 1 \\ I & II & III & IV \end{array} \right| \left| \begin{array}{cccc} 5 & 7 & 3 & 4 \\ I & II & III & IV \end{array} \right| \left| \begin{array}{cccc} 5 & 7 & 6 & 8 \\ I & II & III & IV \end{array} \right| \left| \begin{array}{cc} 9 & 8 \\ I & II \end{array} \right| \left| \begin{array}{c} 9 \\ III \end{array} \right|$$

Una possibile implementazione dell'algoritmo è la seguente:

```

1 void shellSort (int *A, int l, int r) {
2     int i, j, tmp, h, n;
3
4     h = 1;
5     n = r-l+1;
6
7     while (h<(n/3)) {
8         h = 3*h+1;
9
10    while (h >= 1) {
11        for (i=h; i<=r; i++) {
12            j = i;
13            for (j=i; j>=h && A[j]<A[j-h]; j-=h) {
14                tmp = A[j];
15                A[j] = A[j-h];
16                A[j-h] = tmp;
17            }
18        }
19        h = h/3;
20    }
21    return;
22}

```

L'algoritmo innanzitutto determina il valore di h in funzione del numero di chiavi n , come indicato dal codice alle righe 7÷8. Quindi, fintanto che $h > 1$ applica l'insertion sort per generare le sotto-sequenze h -ordinate.

Lo shell sort è in loco, ma non stabile. La sua complessità con la sequenza di Knuth è $O(n^{\frac{3}{2}})$.

1.4.2 Esercizi svolti

Esercizio

Si ordini in maniera ascendente il seguente vettore di interi mediante shell sort con la sequenza di Knuth.

7 6 8 9 8 6 2 1 8 7 0 4 5 3 0 1 0 4 9

Si indichino i passaggi principali.

Soluzione

Data la dimensione del vettore, l'ultimo valore della sequenza di Knuth da considerare è $h = 13$. Applicando l'insertion sort ai sotto-vettori i cui elementi successivi distano tra di loro 13 si ottengono sotto-sequenze 13-ordinate:

$$\left| \begin{array}{cccccccccccccc} 3 & 0 & 1 & 0 & 4 & 6 & 2 & 1 & 8 & 7 & 0 & 4 & 5 & 3 & 0 & 1 & 0 & 4 & 9 \\ I & II & III & IV & V & VI & VII & VIII & IX & X & XI & XII & XIII & & & & & & \\ \end{array} \right| \left| \begin{array}{ccccccccc} 7 & 6 & 8 & 9 & 8 & 9 \\ I & II & III & IV & V & VI & \end{array} \right|$$

Proseguendo con $h = 4$, applicando l'insertion sort ai sotto-vettori i cui elementi successivi distano tra di loro 4 si ottengono sotto-sequenze 4-ordinate:

$$\left| \begin{array}{cccc} 3 & 0 & 0 & 0 \\ I & II & III & IV \end{array} \right| \left| \begin{array}{cccc} 4 & 6 & 1 & 1 \\ I & II & III & IV \end{array} \right| \left| \begin{array}{cccc} 5 & 7 & 2 & 4 \\ I & II & III & IV \end{array} \right| \left| \begin{array}{cccc} 8 & 7 & 6 & 8 \\ I & II & III & IV \end{array} \right| \left| \begin{array}{ccc} 9 & 8 & 9 \\ I & II & III \end{array} \right|$$

Infine con $h = 1$, applicando l'insertion sort ai sotto-vettori i cui elementi successivi distano tra di loro di una posizione, si ottiene il vettore ordinato:

0 0 0 1 1 2 3 4 4 5 6 6 7 7 8 8 8 9 9

1.4.3 Esercizi proposti

Esercizio

Si ordini in maniera ascendente il seguente vettore di interi mediante shell sort con la sequenza di Knuth.

17 16 18 19 8 6 2 11 8 7 10 4 5 3 10 1 0 4 9

Si indichino i passaggi principali.

Esercizio

Si ordini in maniera ascendente il seguente vettore di interi mediante shell sort con la sequenza di Knuth.

8 7 8 9 8 6 2 1 8 5 1 4 5 3 0 1 2 4 5

Si indichino i passaggi principali.

1.5 Counting sort

1.5.1 Richiami di teoria

Il counting sort è un algoritmo di ordinamento interno non basato sul confronto di complessità lineare. Dati n interi in ingresso compresi nell'intervallo $[0, k - 1]$, se $k = O(n)$, allora

$$T(n) = O(n)$$

L'algoritmo si basa sul calcolo della posizione finale di ogni chiave nell'ordinamento in base al numero di occorrenze di chiavi minori o uguali alla chiave stessa (occorrenze multiple).

Le strutture dati coinvolte sono un vettore **A** con n elementi, un vettore **C** con k elementi e un vettore **B** di n elementi. Gli elementi di tutti i vettori sono interi. Il vettore **A** contiene i dati in ingresso, **B** quelli in uscita, **C** le occorrenze semplici prima e multiple dopo. Una possibile implementazione dell'algoritmo è la seguente:

```

1 void countingSort (int *A, int l, int r, int k) {
2     int i, n, *B, *C;
3
4     n = r - l + 1;
5     B = malloc (n * sizeof(int));
6     C = malloc (k * sizeof(int));
7
8     for (i=0; i<k; i++)
9         C[i] = 0;
10
11    for (i=l; i<=r; i++)
12        C[A[i]]++;
13
14    for (i=1; i<k; i++)
15        C[i] += C[i-1];
16
17    for (i=r; i>=l; i--) {
18        B[C[A[i]]-1] = A[i];
19        C[A[i]]--;
20    }
21
22    for (i=l; i<=r; i++)
23        A[i] = B[i];
24
25    return;
26 }
```

Un primo passo consiste nell'inizializzazione del vettore **C** (righe 8 ÷ 9).

Successivamente, si costruisce il vettore **C** delle *occorrenze semplici* (righe 11 ÷ 12). In questo modo **C[i]** memorizza il numero di occorrenze di valore uguale a **i** in **A**.

Al terzo passo si costruisce il vettore delle *occorrenze multiple* mediante il ciclo riportato alle righe 14 ÷ 15. In questo modo **C[i]** memorizza il numero di occorrenze di chiavi di valore minore o uguale a **i** in **A**.

Al quarto passo, con una scansione da destra a sinistra del vettore **A**, che garantisce la stabilità dell'algoritmo, si inserisce ogni suo elemento nella posizione finale corretta nel vettore **B** sulla base dell'informazione contenuta nel vettore **C**. Ad ogni iterazione il vettore **C** viene opportunamente aggiornato. Il codice che effettua tali operazioni è quello delle righe 17 ÷ 20.

Un passo finale (righe 22 ÷ 23) si occupa di ricopiare il vettore **B** in **A**.

Il counting sort è stabile, ma non in loco.

1.5.2 Esercizi svolti

Esercizio

Si ordini in maniera ascendente mediante counting sort il seguente vettore **A** di interi:

1 4 5 1 0 4 2 4

Si indichino le strutture dati usate nei passi intermedi.

Soluzione

Nell'esempio si ha $n = 8$ e $k = 6$. La Figura 1.4 riporta in:

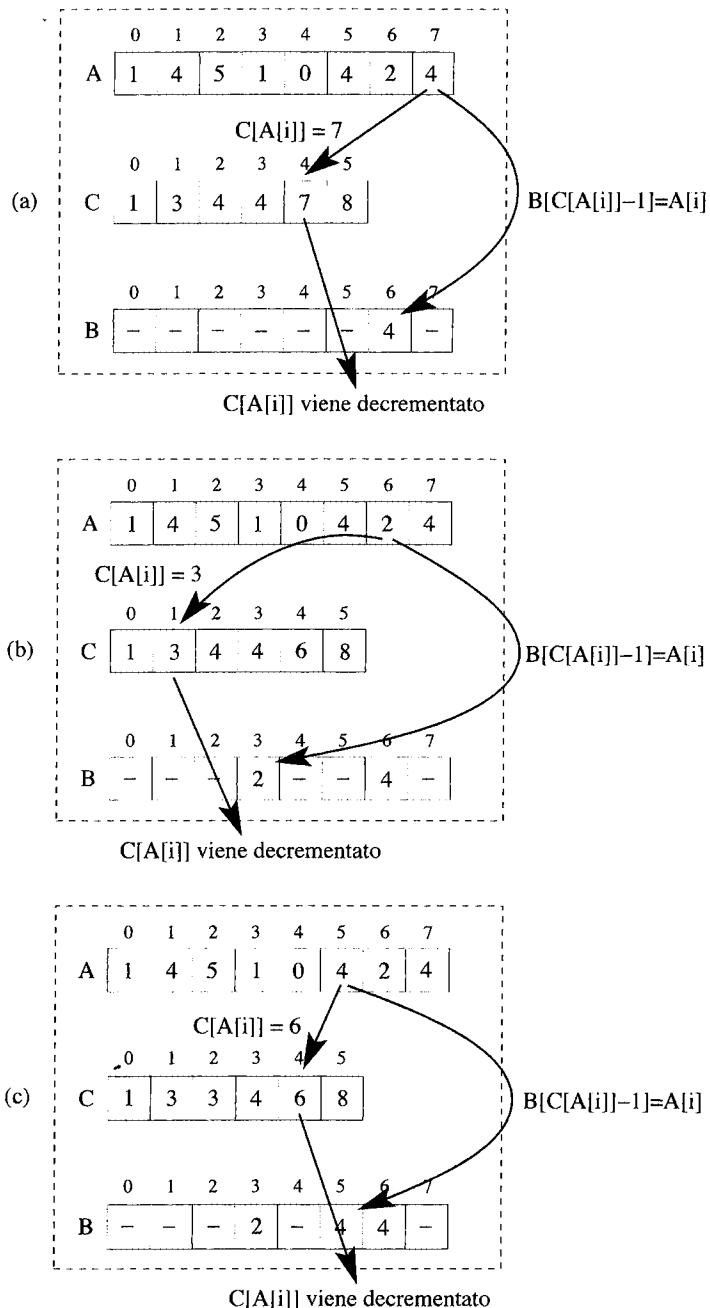
- ▷ (a) il vettore **A** di partenza
- ▷ (b) il vettore **C** dopo l'inizializzazione
- ▷ (c) il vettore **C** delle occorrenze semplici
- ▷ (d) il vettore **C** dopo il calcolo delle occorrenze multiple
- ▷ (e) il vettore risultato **B** all'inizio.

		0	1	2	3	4	5	6	7
(a)	A	1	4	5	1	0	4	2	4
(b)	C	0	0	0	0	0	0	0	0
(c)	C	1	2	1	0	3	1		
(d)	C	1	3	4	4	7	8		
(e)	B	–	–	–	–	–	–		

Figura 1.4 Ordinamento mediante conteggio: situazione iniziale.

I singoli elementi del vettore **A** sono quindi analizzati in successione, a partire dall'estremo destro procedendo verso quello sinistro. La Figura 1.5 riporta le prime tre fasi del procedimento.

Il primo valore esaminato è 4, usando il quale come indice si ricava dal vettore delle occorrenze multiple che il numero di elementi di **A** di valore minore o uguale a esso è pari a 7. Questo vuol dire che 4 può essere memorizzato direttamente nel settimo elemento (quello di indice 6) del vettore ordinato, proprio perché si sa già che tutti gli elementi del vettore ordinato di indice inferiore a 6 alla fine conterranno valori minori di (o al

**Figura 1.5** Ordinamento mediante conteggio: i primi 3 passi.

più uguali a) 4. Il vettore **B** assumerà dunque l'aspetto di Figura 1.5(a). Nel contempo, l'elemento di indice 4 del vettore delle occorrenze multiple viene decrementato, per cui il contenuto di **C** diventa quello riportato in Figura 1.5(b).

Si esamina poi il penultimo elemento di **A**, ovvero 2. In questo caso, il numero di elementi di **A** con valore minore o uguale a 2 è pari a 4, quindi 2 può essere ricopiatò nel quarto elemento (indice 3) di **B**; l'elemento di **C** di indice 2 viene inoltre decrementato. I vettori **B** e **C** corrispondenti sono riportati in Figura 1.5(b) e Figura 1.5(c), rispettivamente.

A questo punto si passa a considerare il terz'ultimo elemento di **A**, che è di nuovo 4. Questa volta si deduce dal vettore delle occorrenze multiple che il numero di elementi "rimasti" in **A** di valore minore o uguale a 4 è pari a 6, quindi 4 viene copiato nel sesto elemento di **B**, e il corrispondente elemento di **C** viene ancora decrementato: il vettore **B** è riportato in Figura 1.5(c).

Si procede in questo modo sino a quando tutti gli elementi di **A** non sono stati considerati. La situazione finale a cui si giunge è:

$$\begin{array}{rcl} B & = & 0 \ 1 \ 1 \ 2 \ 4 \ 4 \ 4 \ 5 \\ C & = & 0 \ 1 \ 3 \ 4 \ 4 \ 7 \end{array}$$

in cui il vettore **B** corrisponde esattamente al vettore **A** ordinato.

Si omette, in quanto banale, il passo finale in cui si ricopia il vettore **B** nel vettore **A**.

Esercizio

Si ordini in maniera ascendente mediante counting sort il seguente vettore **A** di interi:

$$10 \ 8 \ 7 \ 4 \ 6 \ 10 \ 5 \ 5 \ 1 \ 7 \ 4 \ 9 \ 10$$

Si indichino le strutture dati usate nei passi intermedi.

Soluzione

La Figura 1.4 riporta in:

- ▷ (a) il vettore **A** di partenza
- ▷ (b) il vettore **C** dopo l'inizializzazione
- ▷ (c) il vettore **C** delle occorrenze semplici
- ▷ (d) il vettore **C** dopo il calcolo delle occorrenze multiple
- ▷ (e) il vettore risultato **B** all'inizio.

I singoli elementi del vettore **A** sono quindi analizzati in successione, a partire dall'estremo destro procedendo verso quello sinistro. Il primo valore esaminato è pertanto 10, per il quale si ricava dal vettore delle occorrenze multiple che il numero di elementi di **A** di valore minore o uguale a esso è pari a 13. Questo vuol dire che 10 può essere memorizzato direttamente nel tredicesimo elemento (quello di indice 12) del vettore ordinato, proprio perché si sa già che tutti gli elementi del vettore ordinato di indice inferiore a 12 alla fine conterranno valori minori di (o al più uguali a) 10. Viene contestualmente aggiornato il numero di occorrenze multiple di 10 che decresce a 12. La Figura 1.7 riporta in (a) la situazione dopo le operazioni prima elencate.

		0	1	2	3	4	5	6	7	8	9	10	11	12
(a)	A	10	8	7	4	6	10	5	5	1	7	4	9	10
(b)	C	0	0	0	0	0	0	0	0	0	0	0	0	0
(c)	C	0	1	0	0	2	2	1	2	1	1	3		
(d)	C	0	1	1	1	3	5	6	8	9	10	13		
(e)	B	–	–	–	–	–	–	–	–	–	–	–	–	–

Figura 1.6 Ordinamento mediante conteggio: situazione iniziale.

Il secondo valore esaminato è 9, per il quale si ricava dal vettore delle occorrenze multiple che il numero di elementi di **A** di valore minore o uguale a esso è pari a 10. Questo vuol dire che 9 può essere memorizzato direttamente nel decimo elemento (quello di indice 9) del vettore ordinato, proprio perché si sa già che tutti gli elementi del vettore ordinato di indice inferiore a 9 alla fine conterranno valori minori di (o al più uguali a) 9. Viene contestualmente aggiornato il numero di occorrenze multiple di 9 che decresce a 9.

La Figura 1.7 riporta in (c) la situazione dopo le operazioni prima elencate.

Il terzo valore esaminato è 4, per il quale si ricava dal vettore delle occorrenze multiple che il numero di elementi di **A** di valore minore o uguale a esso è pari a 3. Questo vuol dire che 4 può essere memorizzato direttamente nel terzo elemento (quello di indice 2) del vettore ordinato, proprio perché si sa già che tutti gli elementi del vettore ordinato di indice inferiore a 3 alla fine conterranno valori minori di (o al più uguali a) 4. Viene contestualmente aggiornato il numero di occorrenze multiple di 4 che decresce a 2.

La Figura 1.7 riporta in (c) la situazione dopo le operazioni prima elencate.

Si lasciano al lettore i rimanenti passi.

1.5.3 Esercizi risolti

Esercizio

Si ordini in maniera ascendente mediante counting sort il seguente vettore di interi:

5 6 5 3 3 7 4 4 4 5 3 8 8

Si indichino le strutture dati usate nei passi intermedi.

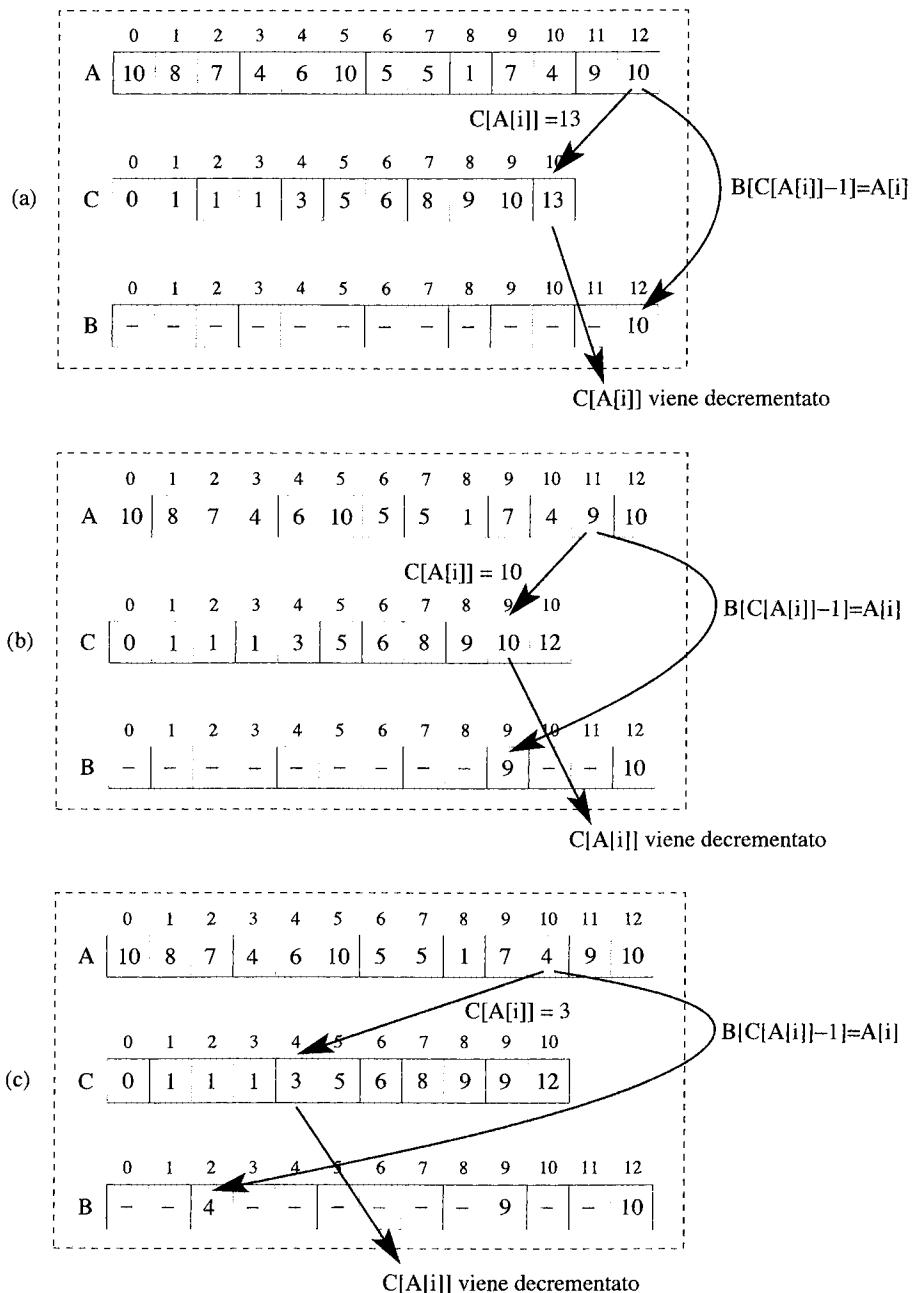


Figura 1.7 Ordinamento mediante conteggio: i primi 3 passi.

Soluzione

La Tabella 1.1 riporta nell'ordine: il vettore iniziale A , il vettore C dopo l'inizializzazione (1), il calcolo delle occorrenze semplici (2) e di quelle multiple (3) e il vettore finale B . Sono inoltre riportati i decrementi dei valori degli elementi di C durante le varie fasi di inserzione degli elementi originali nel vettore finale.

A	0	1	2	3	4	5	6	7	8	9	10	11	12
	5	6	5	3	3	7	4	4	4	5	3	8	8
C^1	0	1	2	3	4	5	6	7	8	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0
C^2	0	1	2	3	4	5	6	7	8	0	0	0	2
	0	0	0	3	3	3	1	1	2	0	0	0	0
C^3	0	1	2	3	4	5	6	7	8	0	0	0	13
	0	0	0	3	6	9	10	11	12	2	5	8	9
				1	4	7							11
				0	3	6							
B	0	1	2	3	4	5	6	7	8	9	10	11	12
	3	3	3	4	4	4	5	5	5	6	7	8	8

Tabella 1.1**1.5.4 Esercizi proposti****Esercizio**

Si ordini in maniera ascendente mediante counting sort il seguente vettore di interi:

7 1 3 1 2 4 5 7 2 4 3 9 9

Si indichino le strutture dati usate nei passi intermedi.

Esercizio

Si ordini in maniera ascendente mediante counting sort il seguente vettore di interi:

6 2 3 2 14 5 6 2 4 3 8 8

Si indichino le strutture dati usate nei passi intermedi.

Esercizio

Si ordini in maniera ascendente mediante counting sort il seguente vettore di interi:

11 1 4 9 8 9 5 13 8 7 3 6 1 7 4 5

Si indichino le strutture dati usate nei passi intermedi.

Esercizio

Si ordini in maniera ascendente mediante counting sort il seguente vettore di interi:

9 7 6 3 5 9 4 4 1 6 3 8 9

Si indichino le strutture dati usate nei passi intermedi.

Capitolo 2

Algoritmi ricorsivi di ordinamento interno

In questo capitolo si tratteranno alcuni tra i più efficienti e diffusi algoritmi di ordinamento ricorsivo.

In particolare si analizzeranno gli algoritmi denominati *merge sort* e *quick sort*. L'algoritmo denominato *heap sort* sarà invece presentato nel Capitolo 4.

Si ricorda infine che un oggetto si dice *ricorsivo* se è definito parzialmente (o completamente) in termini di se stesso. Inoltre la ricorsione si definisce usualmente *diretta* quando una funzione richiama direttamente se stessa, e si dice *indiretta* in caso contrario.

2.1 Merge sort

2.1.1 Richiami di teoria

Il merge sort è un algoritmo ricorsivo di ordinamento interno di complessità linearistica. La struttura dati coinvolta è un vettore **A** di interi con indici compresi tra **l** e **r**. Per ogni chiamata all'algoritmo `mergeSort(A, l, r)` si possono individuare tre fasi:

- ▷ divisione
- ▷ discesa ricorsiva nei sotto-problemi
- ▷ ricombinazione delle soluzioni.

Nella fase di divisione vengono individuati i due sotto-vettori sinistro e destro rispetto all'elemento centrale del vettore di indice **q**:

$$q = \frac{l+r}{2}$$

Nella fase di discesa ricorsiva si ricorre prima sul sotto-vettore sinistro, `mergeSort(A, l, q)`, e poi su quello destro `mergeSort(A, q+1, r)`.

La condizione di terminazione è che un vettore con uno ($l = r$) oppure zero ($l > r$) elementi è ordinato.

Nella fase di ricombinazione si fondono i due sotto-vettori ordinati in un vettore ordinato.

Una possibile implementazione dell'algoritmo è la seguente:

```

1 void mergeSort (int A[], int l, int r) {
2     int q;
3
4     if (r<=l)
5         return;
6
7     q = (l+r)/2;
8     mergeSort (A, l, q);
9     mergeSort (A, q+1, r);
10    merge (A, l, q, r);
11
12    return;
13 }
```

La fusione `merge(A, p, q, r)` di due vettori ordinati (sotto-vettori di `A` compresi tra $[l \div q]$ e tra $[q+1 \div r]$) genera un vettore ordinato `B`, che viene infine ricopiato in `A`. Il codice per la funzione `merge` è il seguente:

```

1 void merge (int A[], int l, int q, int r) {
2     int i, j, k, B[MAX];
3
4     i = l;
5     j = q+1;
6     for (k=l; k<=r; k++)
7         if (i>q)
8             B[k] = A[j++];
9         else if (j>r)
10            B[k] = A[i++];
11        else if (A[i]<=A[j])
12            B[k] = A[i++];
13        else
14            B[k] = A[j++];
15
16    for (k=l; k<=r; k++)
17        A[k] = B[k];
18
19    return;
20 }
```

Il merge sort non è in loco, in quanto utilizza un vettore ausiliario. Esso è stabile se, come nel codice presentato, in caso di chiavi uguali, la funzione `merge` seleziona la chiave dal sotto-vettore sinistro.

2.1.2 Esercizi svolti

Esercizio

Si ordini in maniera discendente la seguente sequenza di interi mediante merge sort.

12 1 34 4 5 7 0 9 17 25 6 8 14 10

Si indichino i passaggi principali.

Soluzione

La fase di divisione è rappresentata nella parte superiore, quella di ricombinazione nella parte inferiore della Figura 2.1. Si noti inoltre che, per mere ragioni di visualizzazione, la soluzione mostra impropriamente un comportamento in ampiezza sull'albero della ricorsione, anziché in profondità.

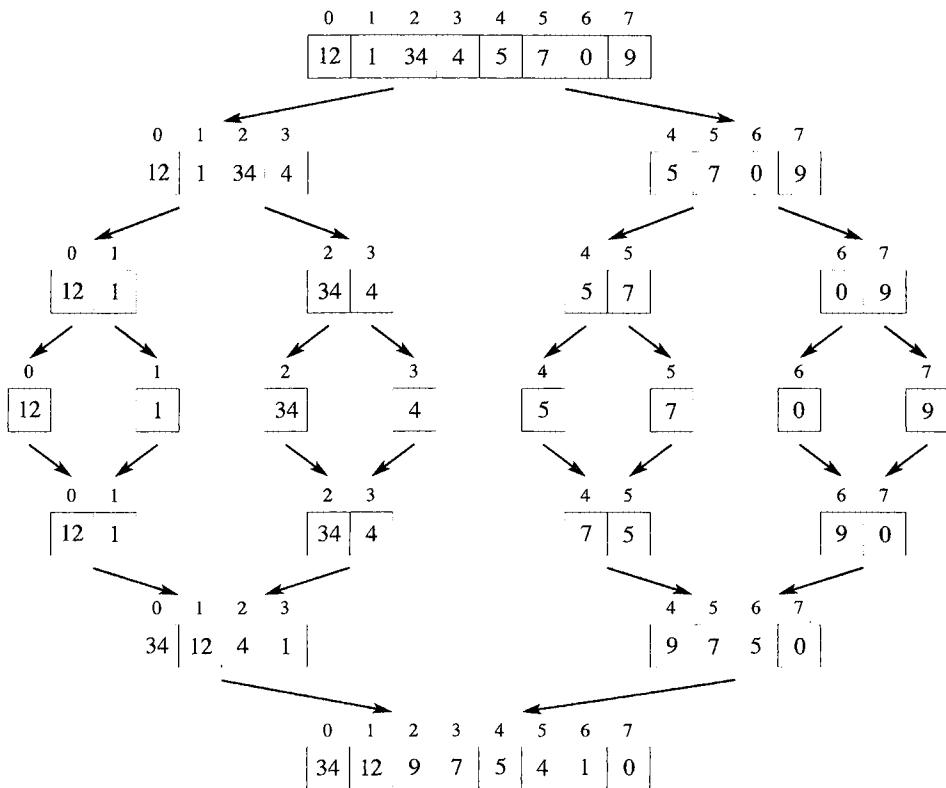


Figura 2.1 Ordinamento per fusione: fase di divisione (prime quattro righe) e fase di ricombinazione (ultime tre righe).

2.1.3 Esercizi proposti

Esercizio

Si ordini in maniera discendente la seguente sequenza di interi mediante merge sort.

22 15 34 40 51 7 90 19 17 25 16 83 14 100

Si indichino i passaggi principali.

2.2 Quick sort

2.2.1 Richiami di teoria

Il quick sort è un algoritmo ricorsivo di ordinamento interno di complessità quadratica nel caso peggiore, ma linearitmica nel caso medio. La struttura dati è un vettore **A** di interi con indici compresi tra 1 e **r**.

Si possono individuare 3 fasi:

- ▷ divisione mediante procedura **partition**
- ▷ discesa ricorsiva nei sotto-problemi
- ▷ ricombinazione delle soluzioni.

Una possibile implementazione dell'algoritmo è la seguente:

```
1 void quickSort (int A[], int l, int r) {  
2     int q;  
3  
4     if (r<=l)  
5         return;  
6  
7     q = partition (A, l, r);  
8     quickSort (A, l, q-1);  
9     quickSort (A, q+1, r);  
10    return;  
11 }  
12 }
```

Nella fase di discesa ricorsiva si ricorre prima sul sotto-vettore sinistro, **quickSort(A, l, q-1)**, poi su quello destro, **quickSort(A, q+1, r)**.

La condizione di terminazione è che un vettore con uno (**r = 1**) oppure zero (**r < 1**) elementi è ordinato.

La fase di ricombinazione non comporta operazioni.

Nella fase di divisione il vettore **A[l ÷ r]** viene partizionato in due sotto-vettori sinistro e destro mediante la funzione **partition**. Essa sceglie arbitrariamente un elemento **x**, che funge da pivot. Per convenzione si è selezionato **x = A[r]**. Mediante due cicli indipendenti, uno discendente da destra a sinistra e uno ascendente da sinistra a destra, si individua una coppia di elementi **A[i]** e **A[j]** fuori posto e li si scambia. Questa operazione è ripetuta fintanto che **i < j**. Infine si scambia **x** con **A[i]** e si ritorna **i**. Al termine delle operazioni precedenti di ha:

- ▷ il sotto-vettore sinistro **A[l ÷ q-1]** contiene tutti gli elementi $\leq x$
- ▷ il sotto-vettore destro **A[q+1 ÷ r]** contiene tutti gli elementi $\geq x$
- ▷ l'elemento **A[q]** si trova nella posizione finale corretta nell'ordinamento.

Il codice per la funzione `partition` è il seguente:

```

1 int partition (int A[], int l, int r ) {
2     int i, j;
3     int x = A[r];
4
5     i = l-1;
6     j = r;
7     for ( ; ; ) {
8         while (A[++i]<x);
9         while (A[--j]>x)
10             if (j==l)
11                 break;
12
13             if (i>=j)
14                 break;
15
16         swap (A, i, j);
17     }
18
19     Swap (A, i, r);
20
21     return i;
22
23 }
```

L'algoritmo di quick sort è in loco, ma non stabile. Il caso peggiore (vettore già ordinato, ma in senso opposto a quello richiesto) è di complessità quadratica, il caso medio è di complessità linearitmica.

2.2.2 Esercizi svolti

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

8 29 60 6 4 93 85 5 26 19 2 69 83 72 43 50

si eseguano i primi due passi dell'algoritmo di quick sort per ottenere un ordinamento ascendente, indicando ogni volta il pivot scelto. I passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le due partizioni del vettore originale e le due partizioni delle partizioni trovate al punto precedente.

Soluzione

Il primo pivot **x** è la chiave 50 (chiave più a destra). La funzione **partition** inizia a scandire il vettore dall'estremità destra mediante un indice **j** decrescente. La scansione si ferma quando si trova il primo valore minore del pivot 50:

8 29 60 6 4 93 85 5 26 19 2 69 83 72 43 50
 j

La funzione `partition` inizia ora a scandire il vettore dall'estremità sinistra mediante un indice i crescente. La scansione si ferma quando si trova il primo valore maggiore del pivot:

Ora vengono scambiati i due valori individuati dagli indici i e j , ottenendo:

8	29	43	6	4	93	85	5	26	19	2	69	83	72	60	50
	<i>i</i>												<i>j</i>		

Dato che $i < j$ si riprende prima la scansione verso sinistra e poi quella verso destra con le stesse modalità. L'indice j continua a spostarsi verso sinistra finché non incontra il prossimo elemento (2) minore del pivot, l'indice i si sposta verso destra fino a incontrare il prossimo elemento maggiore del pivot (93):

8 29 43 6 4 93 85 5 26 19 2 69 83 72 60 50
i *j*

Ora vengono scambiati i due valori individuati dagli indici i e j , ottenendo:

8	29	43	6	4	2	85	5	26	19	93	69	83	72	60	50
					<i>i</i>				<i>j</i>						

Procedendo sempre nello stesso modo si ha la seguente configurazione:

8 29 43 6 4 2 85 5 26 19 93 69 83 72 60 50
i *j*

Ora vengono scambiati i due valori individuati dagli indici i e j , ottenendo:

8 29 43 6 4 2 19 5 26 85 93 69 83 72 60 50
i *j*

Procedendo si arriva alla configurazione seguente, dove i e j risultano scambiati e questo comporta la terminazione di **partition**:

8 29 43 6 4 2 19 5 26 85 93 69 83 72 60 50
j i

Si scambiano quindi il pivot x e l'elemento $A[i]$ e si ritorna il valore di i , ottenendo:

8 29 43 6 4 2 19 5 26 50 93 69 83 72 60 85
 i i

→ Il procedimento sul sotto-vettore sinistro:

8 29 43 6 4 2 19 5 26

utilizza il pivot $x = A[r] = 26$, secondo i passi seguenti:

8 29 43 6 4 2 19 5 26
 i j

scambiando $A[i]$ con $A[j]$:

$$\begin{array}{cccccccccc} 8 & 5 & 43 & 6 & 4 & 2 & 19 & 29 & 26 \\ & i & & & & & j & & \end{array}$$

Si procede con i cicli per j e i :

$$\begin{array}{cccccccccc} 8 & 5 & 43 & 6 & 4 & 2 & 19 & 29 & 26 \\ & i & & & & & j & & \end{array}$$

Si scambiano gli elementi $A[i]$ e $A[j]$:

$$\begin{array}{cccccccccc} 8 & 5 & 19 & 6 & 4 & 2 & 43 & 29 & 26 \\ & i & & & & & j & & \end{array}$$

Si procede con i cicli per j e i :

$$\begin{array}{cccccccccc} 8 & 5 & 19 & 6 & 4 & 2 & 43 & 29 & 26 \\ & j & & & & i & & & \end{array}$$

Si scambia il pivot x e l'elemento $A[i]$ ritornando il valore di i , ottenendo:

$$8 \ 5 \ 19 \ 6 \ 4 \ 2 \ 26 \ 29 \ 43$$

→ Il procedimento sul sotto-vettore destro:

$$93 \ 69 \ 83 \ 72 \ 60 \ 85$$

utilizza il pivot $x = A[r] = 85$, secondo i passi seguenti:

$$\begin{array}{cccccccccc} 93 & 69 & 83 & 72 & 60 & 85 \\ & i & & & & j & & & \end{array}$$

Si scambiano quindi i valori di $A[i]$ e $A[j]$:

$$\begin{array}{cccccccccc} 60 & 69 & 83 & 72 & 93 & 85 \\ & i & & & j & & \end{array}$$

Si procede con i cicli per j e i :

$$\begin{array}{cccccccccc} 60 & 69 & 83 & 72 & 93 & 85 \\ & j & & & i & & \end{array}$$

Si scambiano il pivot x e $A[i]$ e si ritorna i , ottenendo:

$$60 \ 69 \ 83 \ 72 \ 85 \ 93$$

La risposta corretta all'esercizio proposto è perciò la seguente:

$$8 \ 5 \ 19 \ 6 \ 4 \ 2 \mid 26 \mid 29 \ 43 \parallel 50 \parallel 60 \ 69 \ 83 \ 72 \mid 85 \mid 93$$

dove con \parallel si separano pivot e le due partizioni del primo passo e con $|$ pivot e partizioni del secondo passo.

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

30 32 6 47 78 67 74 21 33 46 10 8 79 98 9 45

si eseguano i primi due passi dell'algoritmo di quick sort per ottenere un ordinamento ascendente, indicando ogni volta il pivot scelto. Si osservi che i passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le due partizioni del vettore originale e le due partizioni delle partizioni trovate al punto precedente.

Soluzione

Il primo pivot x è la chiave 45 (chiave più a destra). La funzione **partition** inizia a scandire il vettore dall'estremità destra mediante un indice j decrescente. La scansione si ferma quando si trova il primo valore (9) minore del pivot 45:

30 32 6 47 78 67 74 21 33 46 10 8 79 98 9 45
 j

La funzione `partition` inizia ora a scandire il vettore dall'estremità sinistra mediante un indice i crescente. La scansione si ferma quando si trova il primo valore (47) maggiore del pivot:

Ora vengono scambiati i due valori individuati dagli indici i e j , ottenendo:

30 32 6 9 78 67 74 21 33 46 10 8 79 98 47 45

Dato che $i < j$ si riprende prima la scansione verso sinistra e poi quella verso destra con le stesse modalità. L'indice j continua a spostarsi verso sinistra finché non incontra il prossimo elemento (8) minore del pivot, l'indice i si sposta verso destra fino a incontrare il primo elemento (78) maggiore del pivot:

30 32 6 9 78 67 74 21 33 46 10 8 79 98 47 45
i *j*

Ora vengono scambiati i due valori individuati dagli indici i e j , ottenendo:

30 32 6 9 8 67 74 21 33 46 10 78 79 98 47 45

Procedendo sempre nello stesso modo si arriva alla seguente configurazione:

30	32	6	9	8	67	74	21	33	46	10	78	79	98	47	45
					<i>i</i>					<i>j</i>					

Ora vengono scambiati i due valori individuati dagli indici i e j , ottenendo:

30 32 6 9 8 10 74 21 33 46 67 78 79 98 47 45

Procedendo sempre nello stesso modo si arriva alla seguente configurazione:

30 32 6 9 8 10 74 21 33 46 67 78 79 98 47 45
 i j

Ora vengono scambiati i due valori individuati dagli indici i e j , ottenendo:

30 32 6 9 8 10 33 21 74 46 67 78 79 98 47 45

Procedendo si arriva alla configurazione seguente, dove i e j risultano scambiati e questo comporta la terminazione di **partition**:

30 32 6 9 8 10 33 21 74 46 67 78 79 98 47 45
 j i

Si scambiano anche pivot x e $A[i]$ e si ritorna i , ottenendo:

30 32 6 9 8 10 33 21 45 46 67 78 79 98 47 74

Il procedimento sul sotto-vettore sinistro:

30 32 6 9 8 10 33 21

utilizza il pivot $x = A[r] = 21$, secondo i passi seguenti:

30 32 6 9 8 10 33 21
 i j

scambio di $A[i]$ e $A[j]$:

10 32 6 9 8 30 33 21

ricerca della prossima coppia di elementi fuori posto:

10 32 6 9 8 30 33 21
 i j

scambio di $A[i]$ e $A[j]$:

10 8 6 9 32 30 33 21

ricerca della prossima coppia di elementi fuori posto:

10 8 6 9 32 30 33 21
 j i

essendo i e j scambiati, avviene lo scambio tra pivot x e $A[i]$, si ritorna i , ottenendo:

10 8 6 9 21 30 33 32

Il procedimento sul sotto-vettore destro:

46 67 78 79 98 47 74

utilizza il pivot $x = A[r] = 74$, secondo i passi seguenti. Innanzitutto, si ricerca la prima coppia di elementi fuori posto:

46 67 78 79 98 47 74
i j

scambio di $A[i]$ e $A[j]$:

46 67 47 79 98 78 74

poi si procede con il ciclo su j e quello su i

46 67 47 79 98 78 74
j i

essendo i e j scambiati, avviene lo scambio di pivot x e $A[i]$, si ritorna i , ottenendo:

46 67 47 74 98 78 79

La risposta corretta all'esercizio proposto è perciò la seguente :

10 8 6 9 | 21 | 30 33 32 || 45 || 46 67 47 | 74 | 98 78 79

2.2.3 Esercizi risolti

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

12 1 34 4 5 7 0 9 17 25 6 8 4 10

si eseguano i primi due passi dell'algoritmo di quick sort per ottenere un ordinamento ascendente indicando ogni volta il pivot scelto. Si osservi che i passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le due partizioni del vettore originale e le due partizioni delle partizioni trovate al punto precedente.

Soluzione

Il risultato è il seguente:

4 1 0 4 5 | 6 | 8 9 7 || 10 || 17 12 | 25 | 34

2.2.4 Esercizi proposti

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

12 97 98 19 3 8 76 45 32 6 71 78 33 60 10 45

si eseguano i primi due passi dell'algoritmo di quick sort per ottenere un ordinamento ascendente indicando ogni volta il pivot scelto. Si osservi che i passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le due partizioni del vettore originale e le due partizioni delle partizioni trovate al punto precedente.

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

23 44 56 11 9 89 90 50 13 42 1 67 88 77 24 55

si eseguano i primi due passi dell'algoritmo di quick sort per ottenere un ordinamento ascendente, indicando ogni volta il pivot scelto. Si osservi che i passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le due partizioni del vettore originale e le due partizioni delle partizioni trovate al punto precedente.

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

13 34 65 11 9 98 90 5 31 24 1 76 88 77 48 55

si eseguano i primi due passi dell'algoritmo di quick sort per ottenere un ordinamento ascendente, indicando ogni volta il pivot scelto. Si osservi che i passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le due partizioni del vettore originale e le due partizioni delle partizioni trovate al punto precedente.

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

11 31 24 9 98 90 5 13 88 77 34 65 1 76 48 55

si eseguano i primi due passi dell'algoritmo di quick sort per ottenere un ordinamento ascendente, indicando ogni volta il pivot scelto. Si osservi che i passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le due partizioni del vettore originale e le due partizioni delle partizioni trovate al punto precedente.

Capitolo 3

Equazioni alle ricorrenze

Nel metodo di risoluzione *divide et impera* un problema viene risolto in maniera ricorsiva, suddividendo il problema in sotto-problemi più piccoli, risolvendo tali problemi singolarmente e infine combinandone le soluzioni.

Le *equazioni alle ricorrenze* sono strettamente legate al divide et impera, in quanto offrono un metodo naturale per caratterizzare i tempi di esecuzione degli algoritmi di tale tipo.

In questo capitolo si analizzerà il metodo di risoluzione delle equazioni alle ricorrenze detto dello sviluppo o di iterazione o di unfolding. Gli altri metodi di risoluzione, spesso presentati in letteratura, vengono lasciati all'approfondimento personale.

3.1 Metodo dello sviluppo (iterazione o unfolding)

3.1.1 Richiami di teoria

Il metodo detto di sviluppo (anche detto di iterazione o unfolding) utilizza lo sviluppo dell'equazione alle ricorrenze fino a un certo numero di passi per poi ricostruire una legge generale secondo il seguente schema:

- ▷ si calcolano i primi due o tre passi dello sviluppo
- ▷ si esprime $T(n)$ come sommatoria
- ▷ si determinano i limiti superiore e inferiore della sommatoria
- ▷ si calcola il valore della sommatoria.

Il metodo fornisce una indicazione di limite superiore lasco (notazione O) sulla complessità dell'algoritmo.

Di seguito vengono riportate alcune formule che si ritengono utili ai fini dei calcoli.

- ▷ Progressione geometrica

$$\sum_{i=0}^n x^i = \frac{x^{n+1}-1}{x-1} \quad (3.1)$$

- ▷ Relazioni notevoli

$$a^{\log_b n} = n^{\log_b a} \quad (3.2)$$

$$\log_b a = \frac{\log_c a}{\log_c b} \quad (3.3)$$

▷ Progressione aritmetica

$$\sum_{i=0}^{n-1} i = \frac{1}{2} \cdot n \cdot (n - 1) \quad (3.4)$$

Nel corso della trattazione ci si riferirà a tali formule tramite le etichette indicate di fianco a destra.

3.1.2 Esercizi svolti

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 3 \cdot T\left(\frac{n}{2}\right) + \frac{n}{2} & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Soluzione

Al passo 1, esprimiamo i primi passi dello sviluppo:

$$\begin{aligned} T(n) &= 3 \cdot T\left(\frac{n}{2}\right) + \frac{n}{2} \\ T\left(\frac{n}{2}\right) &= 3 \cdot T\left(\frac{n}{4}\right) + \frac{n}{4} \\ T\left(\frac{n}{4}\right) &= 3 \cdot T\left(\frac{n}{8}\right) + \frac{n}{8} \end{aligned}$$

Si ha quindi:

$$\begin{aligned} T(n) &= \frac{n}{2} + 3 \cdot T\left(\frac{n}{2}\right) \\ &= \frac{n}{2} + 3 \cdot \left(\frac{n}{4} + 3 \cdot T\left(\frac{n}{4}\right)\right) \\ &= \frac{n}{2} + 3 \cdot \left(\frac{n}{4} + 3 \cdot \left(\frac{n}{8} + 3 \cdot T\left(\frac{n}{8}\right)\right)\right) \\ &= \frac{n}{2} + 3 \cdot \frac{n}{4} + 9 \cdot \frac{n}{8} + 27 \cdot T\left(\frac{n}{8}\right) \end{aligned}$$

Al passo 2 in forma compatta si può scrivere, tenendo conto che n è costante rispetto alla sommatoria,

$$T(n) = \frac{n}{2} \cdot \sum_{i=0}^{\text{estremo superiore}} \left(\frac{3}{2}\right)^i$$

Al passo 3 per trovare l'estremo superiore si impone, ricorrendo alla condizione di terminazione, che l'insieme di dati su cui opera la funzione $T(n)$ al passo i -esimo della ricorsione abbia cardinalità 1: $\frac{n}{2^i} = 1$ da cui $i = \log_2 n$. Al passo 4 sfruttando le

relazioni 3.1 e 3.2 introdotte in precedenza si ottiene:

$$\begin{aligned}
 T(n) &= \frac{n}{2} \cdot \frac{\left(\frac{3}{2}\right)^{\log_2 n+1}-1}{\frac{3}{2}-1} \\
 &= \frac{n}{2} \cdot \left(\frac{\frac{3}{2} \cdot \left(\frac{3}{2}\right)^{\log_2 n}-1}{\frac{1}{2}} \right) \\
 &= n \cdot \left(\frac{3}{2} \cdot \frac{3^{\log_2 n}}{n} - 1 \right) \\
 &= \frac{3}{2} \cdot n^{\log_2 3} - n \\
 &= O(n^{\log_2 3})
 \end{aligned}$$

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned}
 T(n) &= 4 \cdot T\left(\frac{n}{2}\right) + n^3 & n \geq 2 \\
 T(1) &= 1
 \end{aligned}$$

Soluzione

Al passo 1, esprimiamo i primi passi dello sviluppo:

$$\begin{aligned}
 T(n) &= 4 \cdot T\left(\frac{n}{2}\right) + n^3 \\
 T\left(\frac{n}{2}\right) &= 4 \cdot T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^3 \\
 T\left(\frac{n}{4}\right) &= 4 \cdot T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^3
 \end{aligned}$$

Si ha quindi:

$$T(n) = n^3 + 4 \cdot \left(\left(\frac{n}{2}\right)^3 + 4 \cdot \left(\left(\frac{n}{4}\right)^3 + 4 \cdot T\left(\frac{n}{8}\right)\right) \right)$$

Al passo 2 in forma compatta si può scrivere, tenendo conto che n è costante rispetto alla sommatoria

$$\begin{aligned}
 T(n) &= n^3 + \frac{1}{2} \cdot n^3 + \frac{1}{4} \cdot n^3 + \dots \\
 &= n^3 \sum_{i=0}^{\text{estremo superiore}} \left(\frac{1}{2}\right)^i
 \end{aligned}$$

Al passo 3 per trovare l'estremo superiore si impone, ricorrendo alla condizione di terminazione, che l'insieme di dati su cui opera la funzione $T(n)$ al passo i -esimo della ricorsione abbia cardinalità: $\frac{n}{2^i} = 1$ da cui $i = \log_2 n$.

Al passo 4 la soluzione

$$T(n) = O(n^3)$$

è giustificata dal fatto che la ragione della sommatoria $(1/2)$ è < 1 e quindi la sommatoria produce un numero finito ben preciso che non è necessario conoscere in termini di ragionamento asintotico. Allo stesso risultato si perviene utilizzando le Formule 3.1, 3.2

e 3.3 ricordate sopra:

$$\begin{aligned} T(n) &= n^3 \cdot \frac{\left(\frac{1}{2}\right)^{\log_2 n+1}-1}{\frac{1}{2}-1} \\ &= 2 \cdot n^3 \cdot \left(1 - \frac{1}{2} \cdot \frac{1^{\log_2 n}}{n}\right) \\ &= 2 \cdot n^3 \cdot \left(\frac{2 \cdot n - 1}{2n}\right) \\ &= O(n^3) \end{aligned}$$

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 5 \cdot T\left(\frac{n}{2}\right) + n & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Soluzione

Al passo 1, esprimiamo i primi passi dello sviluppo:

$$\begin{aligned} T(n) &= 5 \cdot T\left(\frac{n}{2}\right) + n \\ T\left(\frac{n}{2}\right) &= 5 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2} \\ T\left(\frac{n}{4}\right) &= 5 \cdot T\left(\frac{n}{8}\right) + \frac{n}{4} \end{aligned}$$

Al passo 2 esprimiamo $T(n)$ come sommatoria

$$\begin{aligned} T(n) &= n + 5 \cdot \left(\left(\frac{n}{2}\right) + 5 \cdot \left(\left(\frac{n}{4}\right) + 5 \cdot T\left(\frac{n}{8}\right)\right)\right) \\ &= n \sum_{i=0}^{\text{estremo superiore}} \left(\frac{5}{2}\right)^i \end{aligned}$$

Al passo 3 troviamo l'estremo superiore della sommatoria, ponendo $\frac{n}{2^i} = 1$ da cui $i = \log_2 n$.

Al passo 4 la soluzione diventa

$$\begin{aligned} T(n) &= n \cdot \frac{\left(\frac{5}{2}\right)^{\log_2 n+1}-1}{\frac{5}{2}-1} \\ &= \frac{2 \cdot n}{3} \cdot \left(\frac{5}{2} \cdot \frac{5^{\log_2 n}}{n} - 1\right) \\ &= O(n^{\log_2 5}) \end{aligned}$$

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{4}\right) + n & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Soluzione

Al passo 1, esprimiamo i primi passi dello sviluppo:

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{4}\right) + n \\ T\left(\frac{n}{2}\right) &= 2 \cdot T\left(\frac{n}{16}\right) + \frac{n}{4} \\ T\left(\frac{n}{4}\right) &= 2 \cdot T\left(\frac{n}{64}\right) + \frac{n}{16} \end{aligned}$$

Si ha quindi:

$$T(n) = n + 2 \cdot \left(\frac{n}{4} + 2 \cdot \left(\frac{n}{16} + 2 \cdot T\left(\frac{n}{64}\right) \right) \right)$$

Al passo 2 esprimiamo $T(n)$ come sommatoria

$$\begin{aligned} T(n) &= n + \frac{n}{2} + \frac{n}{4} + \dots \\ &= n \cdot \sum_{i=0}^{\text{estremo superiore}} \left(\frac{1}{2}\right)^i \end{aligned}$$

Al passo 3 si ha che $\frac{n}{4^i} = 1$ da cui $i = \log_4 n$.

Al passo 4 la soluzione

$$T(n) = O(n)$$

è giustificata dal fatto che la ragione della sommatoria $(1/2)$ è < 1 e quindi la sommatoria produce un numero finito ben preciso che non ci è necessario conoscere in termini di ragionamento asintotico. Allo stesso risultato si perviene utilizzando le formule 3.1 e 3.2 ricordate sopra:

$$\begin{aligned} T(n) &= n \cdot \frac{\left(\frac{1}{2}\right)^{\log_4 n + 1} - 1}{\frac{1}{2} - 1} \\ &= 2 \cdot n \cdot \left(1 - \frac{1}{2} \cdot \frac{1^{\log_4 n}}{2^{\log_4 n}}\right) \\ &= 2 \cdot n \cdot \left(1 - \frac{1}{2 \cdot n^{1/2}}\right) \\ &= O(n) \end{aligned}$$

3.1.3 Esercizi risolti**Esercizio**

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 4 \cdot T\left(\frac{n}{2}\right) + n & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Soluzione

Si ottiene:

$$T(n) = O(n^2)$$

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 4 \cdot T\left(\frac{n}{2}\right) + n^2 & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Soluzione

Si ottiene:

$$T(n) = O(n^2 \cdot \log_2 n)$$

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 3 \cdot T\left(\frac{n}{2}\right) + n & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Soluzione

Si ottiene:

$$T(n) = O(n^{\log_2 3})$$

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{3}\right) + n & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Soluzione

Si ottiene:

$$T(n) = O(n)$$

3.1.4 Esercizi proposti

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 4 \cdot T\left(\frac{n}{2}\right) + n^3 & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 3 \cdot T\left(\frac{n}{2}\right) + n^2 & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 8 \cdot T\left(\frac{n}{2}\right) + n^3 & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 4 \cdot T\left(\frac{n}{3}\right) + n^2 & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Esercizio

Si risolva mediante il metodo dello sviluppo (unfolding) la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{3}\right) + n^2 & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Capitolo 4

Heap, heap sort e code a priorità

In questo capitolo si presenteranno innanzitutto gli *heap* binari, ovvero degli alberi binari con particolari proprietà e normalmente memorizzati in vettori lineari. In seguito saranno introdotti l'algoritmo di ordinamento ricorsivo *heap sort* e le *code a priorità* che si basano proprio sulla struttura degli heap. Su tali strutture dati verranno risolte e proposte diverse tipologie di esercizi.

4.1 Heap

4.1.1 Richiami di teoria

Uno heap è una struttura dati con:

- ▷ una proprietà strutturale: è un albero binario “quasi completo” (tutti i livelli tranne quelli delle foglie debbono essere completi e si riempiono per costruzione da sinistra a destra)
- ▷ una proprietà funzionale: data la terna `root`, `left_child`, `right_child`, ipotizzando chiavi distinte, valgono le seguenti relazioni:

$$\text{key}(\text{root}) > \text{key}(\text{left_child}) \quad \&\& \quad \text{key}(\text{root}) > \text{key}(\text{right_child}).$$

Di conseguenza, la radice contiene la chiave massima di tutto lo heap. Si noti però che ragionando in modo speculare, è possibile imporre che la radice contenga la chiave minore.

Gli heap sono usualmente implementati mediante vettori. Nel seguito si utilizzerà un vettore `A` con `n` elementi interi. La variabile `heapsize` indicherà il numero di elementi che fanno parte dello heap.

Dato il padre i , l'indice per accedere al figlio sinistro `left_child` può essere valutato mediante l'espressione:

$$\text{LEFT}(i) = 2 \cdot i + 1$$

Analogamente si esprime l'indice del figlio destro `right_child`:

$$\text{RIGHT}(i) = 2 \cdot i + 2$$

Infine, dato il figlio i per accedere al padre si può utilizzare l'espressione:

$$PARENT(i) = \frac{(i-1)}{2}.$$

La funzione ricorsiva fondamentale per mantenere la proprietà funzionale dello heap è la **Heapify**. Essa trasforma in uno heap la terna

$$(root_i, left_child, right_child)$$

nel caso in cui `left_child` e `right_child` siano già a loro volta degli heap. La funzione assegna ad `A[i]` il massimo tra `A[i]`, `A[Left(i)]` e `A[Right(i)]`. Se c'è stato scambio tra `A[i]` e `Left[i]`, applica ricorsivamente **Heapify** sul sotto-albero con radice `Left[i]`. Analogamente se c'è stato scambio tra `A[i]` e `Right[i]`, applica ricorsivamente **Heapify** sul sotto-albero con radice `Right[i]`. Una possibile implementazione della funzione **Heapify** è la seguente:

```

1 void Heapify (int A[], int i, int heapsize) {
2     int l, r, largest;
3
4     l = LEFT(i);
5     r = RIGHT(i);
6     if (l < heapsize && A[l] > A[i])
7         largest=l;
8     else
9         largest = i;
10    if (r < heapsize && A[r] > A[largest])
11        largest=r;
12
13    if (largest != i) {
14        Swap (A,i,largest);
15        Heapify (A, largest, heapsize);
16    }
17
18    return;
19 }
20 }
```

Dato un vettore di interi in cui è memorizzato un albero binario, la funzione **BuildHeap** provvede a trasformarlo in uno heap. Ricordando che le foglie sono heap, essa applica la procedura **Heapify** a partire dal padre dell'ultima foglia fino alla radice. Una possibile implementazione della funzione **BuildHeap** è la seguente:

```

1 void BuildHeap (int A[], int heapsize) {
2     int i;
3
4     for (i=(heapsize)/2-1; i>=0; i--)
5         Heapify(A, i, heapsize);
6
7     return;
8 }
```

4.2 Heap sort

4.2.1 Richiami di teoria

Heap sort è un algoritmo ricorsivo di ordinamento interno di complessità linearitmica. Esso procede secondo i seguenti passi:

- ▷ si trasforma il vettore A di interi in uno heap mediante la **BuildHeap**

- ▷ si ripete fino a che lo heap non è vuoto:
 - ◊ lo scambio tra la radice con l'ultima delle foglie (quella più a destra nell'ultimo livello)
 - ◊ la riduzione di una unità della dimensione dello heap
 - ◊ il ripristino delle proprietà dello heap mediante applicazione della funzione **Heapify**.

Una possibile implementazione dell'heap sort è la seguente:

```

1 void HeapSort (int A[], int heapsize) {
2     int i;
3
4     BuildHeap (A, heapsize);
5     for (i=heapsize-1; i>0; i--) {
6         Swap (A,0,i);
7         heapsize--;
8         Heapify (A, 0, heapsize);
9     }
10    return;
11 }
12 }
```

L'algoritmo di heap sort è in loco, ma non stabile.

4.2.2 Esercizi svolti

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

12 14 43 10 80 100 61 32 89 78 44 57 11 68 85 56

- ▷ la si trasformi in uno heap, ipotizzando di usare un vettore come struttura dati. Si riportino graficamente i diversi passi della costruzione dello heap e il risultato finale. Si ipotizzi che nella radice dello heap sia memorizzato il valore massimo
- ▷ si eseguano su tale heap i primi due passi dell'algoritmo di heap sort. Si osservi che la sequenza è già memorizzata nel vettore e rappresenta una configurazione intermedia per cui la proprietà di heap non è ancora soddisfatta.

Soluzione

Nel vettore è già memorizzato l'albero binario di Figura 4.1(a) che però non soddisfa le proprietà dello heap..

Per forzare il rispetto di tali proprietà si applica la **BuildHeap**, a partire dal padre dell'ultima foglia, cioè dall'elemento di indice 8 che contiene la chiave 32. L'applicazione di **Heapify** all'albero con radice 32 porta alla configurazione di Figura 4.1(b).

Continuando il ciclo di applicazione della **Heapify**, le Figure 4.1 (c), (d) ed (e) visualizzano le configurazioni che si hanno quando la **BuildHeap** ha terminato i livelli 2, 1 e 0 rispettivamente.

Sullo heap di Figura 4.1(e) vengono eseguiti i primi due passi dell'algoritmo di heap sort. Al primo passo si scambia la radice (100) con l'ultima delle foglie (14). Si decrementa quindi la dimensione dello heap e si esegue la **Heapify** il cui effetto è evidenziato in Figura 4.1(f).

Al secondo passo si scambia la radice (89) con l'ultima delle foglie (61), si decremente dimensione dello heap e si esegue la **Heapify**, ottenendo la configurazione finale della struttura dati di Figura 4.1(g). Le risposte sintetiche al quesito sono pertanto le

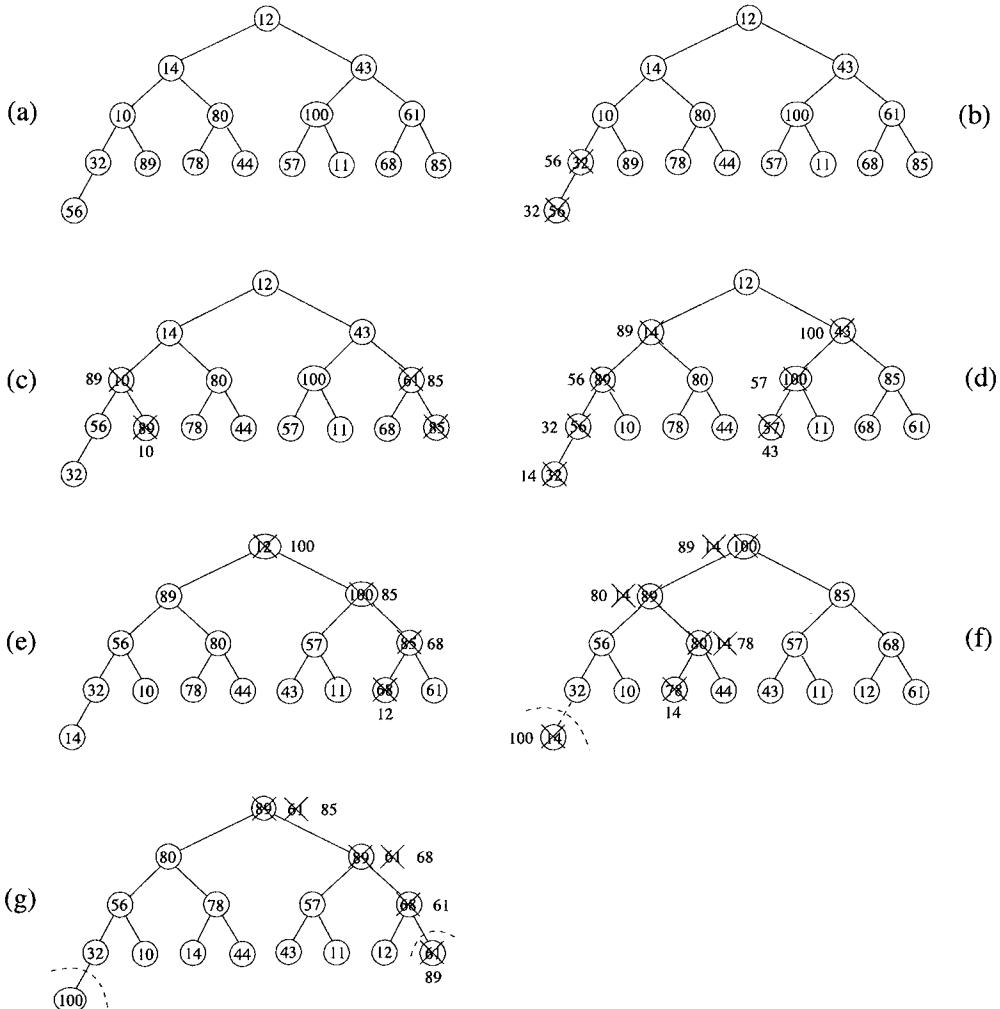


Figura 4.1 Heap sort.

seguenti:

```
100 89 85 56 80 57 68 32 10 78 44 43 11 12 61 14
89 80 85 56 78 57 68 32 10 14 44 43 11 12 61
85 80 68 56 78 57 61 32 10 14 44 43 11 12
```

4.2.3 Esercizi risolti

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

```
41 48 65 36 12 69 13 14 23 10 60 100 78 44 17 21
```

- ▷ la si trasformi in uno heap, ipotizzando di usare un vettore come struttura dati. Si riportino graficamente i diversi passi della costruzione dello heap e il risultato finale. Si ipotizzi che nella radice dello heap sia memorizzato il valore massimo
- ▷ si eseguano su tale heap i primi due passi dell'algoritmo di heap sort. Si osservi che la sequenza è già memorizzata nel vettore e rappresenta una configurazione intermedia per cui la proprietà di heap non è ancora soddisfatta.

Soluzione

```
100 60 78 36 48 69 44 21 23 10 12 41 65 13 17 14
78 60 69 36 48 65 44 21 23 10 12 41 14 13 17
69 60 65 36 48 41 44 21 23 10 12 17 14 13
```

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

```
21 18 15 6 22 96 13 14 33 101 65 10 8 64 17 53
```

- ▷ la si trasformi in uno heap, ipotizzando di usare un vettore come struttura dati. Si riportino graficamente i diversi passi della costruzione dello heap e il risultato finale. Si ipotizzi che nella radice dello heap sia memorizzato il valore massimo
- ▷ si eseguano su tale heap i primi due passi dell'algoritmo di heap sort. Si osservi che la sequenza è già memorizzata nel vettore e rappresenta una configurazione intermedia per cui la proprietà di heap non è ancora soddisfatta.

Soluzione

```
101 65 96 53 22 15 64 14 33 21 18 10 8 13 17 6
96 65 64 53 22 15 17 14 33 21 18 10 8 13 6
65 53 64 33 22 15 17 14 6 21 18 10 8 13
```

4.2.4 Esercizi proposti

Esercizio

Si eseguano i primo due passi dell'algoritmo di heap sort sullo heap di Figura 4.2:

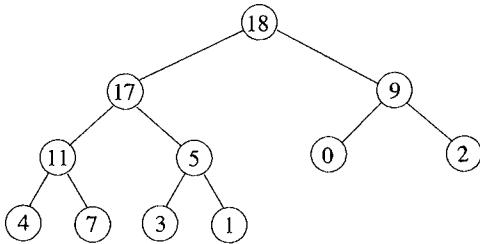


Figura 4.2 Heap iniziale.

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

21 41 34 10 8 12 16 23 97 86 44 75 11 80 85 65

- ▷ la si trasformi in uno heap, ipotizzando di usare un vettore come struttura dati. Si riportino graficamente i diversi passi della costruzione dello heap e il risultato finale. Si ipotizzi che nella radice dello heap sia memorizzato il valore massimo
- ▷ si eseguano su tale heap i primi due passi dell'algoritmo di heap sort. Si osservi che la sequenza è già memorizzata nel vettore e rappresenta una configurazione intermedia per cui la proprietà di heap non è ancora soddisfatta.

Esercizio

Sia data la sequenza di interi, supposta memorizzata in un vettore:

12 1 34 4 5 7 0 9 17 25 6 8 4 10

- ▷ la si trasformi in uno heap, ipotizzando di usare un vettore come struttura dati. Si riportino graficamente i diversi passi della costruzione dello heap e il risultato finale. Si ipotizzi che nella radice dello heap sia memorizzato il valore massimo
- ▷ si eseguano su tale heap i primi due passi dell'algoritmo di heap sort. Si osservi che la sequenza è già memorizzata nel vettore e rappresenta una configurazione intermedia per cui la proprietà di heap non è ancora soddisfatta.

4.3 Code a priorità

4.3.1 Richiami di teoria

Lo heap è anche una possibile implementazione di una coda a priorità, costruita attraverso inserzioni successive di elementi in uno heap inizialmente vuoto. Occorre però non confondere la richiesta di creazione di uno heap per inserzione da quella di correzione di una struttura dati monodimensionale (un banalissimo array) che si vorrebbe trasformare in uno heap. Nel primo caso si **deve** usare la procedura **PQUEUEinsert**, nel secondo caso la funzione **BuildHeap**.

La funzione **PQUEUEinsert** aggiunge una foglia all’albero (cresce per livelli da sinistra a destra, rispettando la proprietà strutturale), risale dalla foglia fino al più alla radice confrontando il valore del nodo padre del nodo corrente con la chiave da inserire (**item**) e facendo scendere la chiave del padre nel nodo corrente se la chiave da inserire è maggiore, altrimenti la inserisce nel nodo corrente. Una possibile implementazione è la seguente:

```

1 void PQinsert (PQ pq, int item) {
2     int i;
3     pq->heapsize++;
4     while (i>=1 && pq->A[PARENT(i)]<item) {
5         pq->A[i] = pq->A[PARENT(i)];
6         i = (i-1)/2;
7     }
8     pq->A[i] = item;
9     return;
10 }
```

La funzione **PQUEUEextract** modifica lo heap, estraendone il valore massimo, che è contenuto nella radice:

- ▷ scambia la radice con l’ultima delle foglie (quella più a destra nell’ultimo livello)
- ▷ riduce di una unità della dimensione dello heap
- ▷ ripristina le proprietà dello heap mediante applicazione di **Heapify**.

Una possibile implementazione è la seguente:

```

1 int PQUEUEextractMax (PQ pq) {
2     int item;
3     Swap (pq, 0, pq->heapsize-1);
4     item = pq->array[pq->heapsize-1];
5     pq->heapsize--;
6     Heapify (pq, 0);
7     return item;
8 }
```

La funzione **PQUEUEchange** modifica la priorità di un elemento dello heap e di conseguenza lo heap stesso. A partire dall’elemento con priorità cambiata, risale fino al più alla radice confrontando la chiave del padre con la chiave modificata, facendo scendere la chiave del padre nel figlio se la chiave da inserire è maggiore, altrimenti la inserisce nel nodo corrente. La funzione applica infine la procedura di **Heapify** a partire dalla posizione dell’elemento con priorità cambiata. Una possibile implementazione è la seguente:

```

1 void PQchange (PQ pq, int pos, int item) {
2     while (pos>=1 && pq->array[PARENT(pos)]<item) {
3         pq->array[pos] = pq->array[PARENT(pos)];
4         pos = (pos-1)/2;
5     }
6     pq->A[pos] = item;
7     Heapify (pq, pos);
8     return;
9 }
```

4.3.2 Esercizi svolti

Esercizio

Si inseriscano, in sequenza, le seguenti chiavi intere in una coda a priorità inizialmente vuota:

11 31 77 34 65 1 76 48 55 24 9 98 90 5 13 88

Si ipotizzi di usare uno heap come struttura dati. Si disegni la struttura ai diversi passi dell'inserzione. Su questa struttura si effettui un'estrazione della chiave a priorità massima (quella associata al valore minimo) e si disegni la struttura dopo l'operazione. Su di essa si cambi la priorità di 76 in 7 e se ne disegni la struttura risultante.

Si ipotizzi che la priorità massima sia associata alla chiave a valore minimo.

Soluzione

In ognuno dei primi 5 passi si crea una foglia, riempiendo per livelli da sinistra a destra. Poiché per ciascuno dei passi la chiave da inserire è maggiore di quella contenuta nel padre, l'inserzione avviene nel nodo foglia appena creato. Il risultato è riportato in Figura 4.3 (a).

Quando si tratta di inserire la chiave 1, il padre della foglia appena creata contiene 77 che è maggiore di 1. Pertanto 77 viene fatto scendere nella foglia appena creata e si risale di un livello nello heap, questa volta confrontando la chiave da inserire 1 con il padre che contiene 11. Anche in questo caso la chiave del padre viene fatta scendere, il confronto arriva alla radice, che, non avendo padre, viene riempita con la chiave da inserire. Il risultato è riportato in Figura 4.3 (b).

Di nuovo fino alla chiave 55 la chiave da inserire è maggiore di quella di suo padre e l'inserzione avviene direttamente nella foglia. Il risultato è riportato in Figura 4.3 (c).

Con la chiave 24 si percorre un cammino verso la radice fino a trovarne la corretta posizione, come mostrato in Figura 4.3 (d) e così via. La risposta corretta alla prima parte dell'esercizio è riportata in Figura 4.3 (e).

Sullo heap di Figura 4.3 (e) si procede con lo scambio tra l'elemento nella radice (1) e quello nell'ultima delle foglie (88). Applicando la **Heapify** si ottiene la configurazione di Figura 4.4.

Sullo heap di Figura 4.4 si modifica 76 in 7 e lo si fa risalire fino alla posizione corretta, cioè quella in cui il padre è minore. Si noti che in questo caso l'applicazione della **Heapify** non ha effetto. Si ottiene la configurazione di Figura 4.5.

4.3.3 Esercizi risolti

Esercizio

Si inserisca la seguente sequenza di interi in una coda a priorità inizialmente supposta vuota:

200 24 45 17 26 10 35 47 4 23 8 7 81 41 12 80

Si ipotizzi di usare uno heap come struttura dati. Si disegni la struttura ai diversi passi dell'inserzione. Sulla struttura risultante si cambi la priorità di 8 in 36 e si disegni lo heap che ne deriva.

Si ipotizzi che la priorità massima sia associata alla chiave a valore minimo.

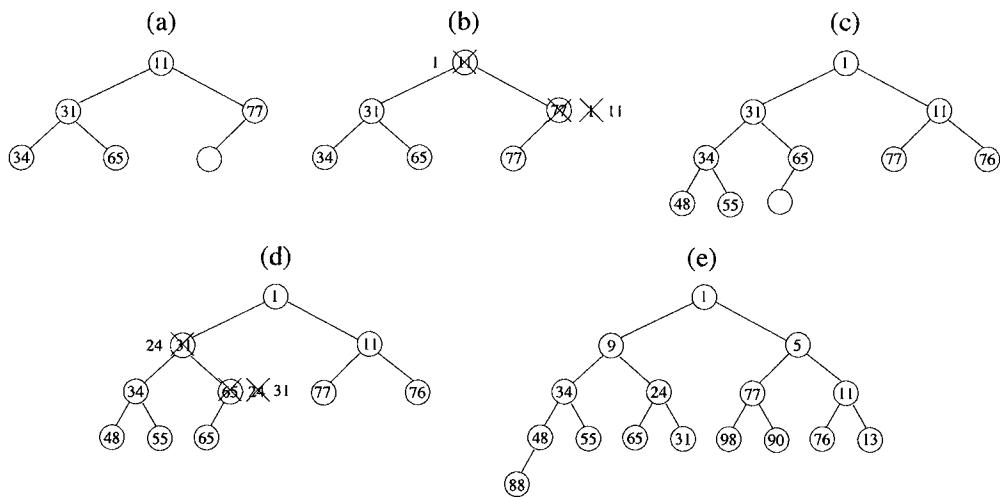


Figura 4.3 Inserzione in una coda a priorità.

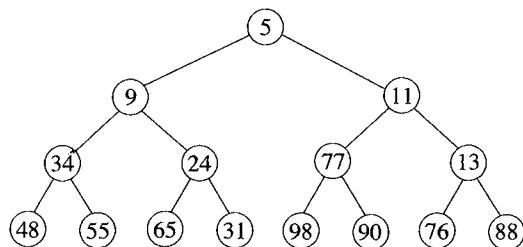


Figura 4.4 Estrazione da una coda a priorità.

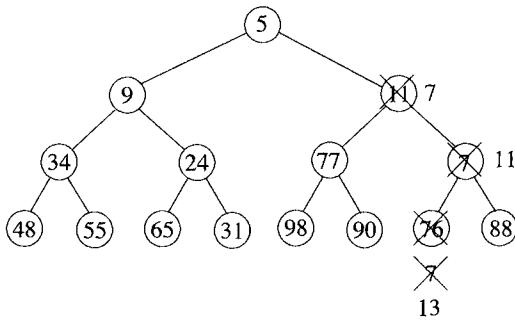


Figura 4.5 Cambio di priorità.

Soluzione

Lo heap dopo le inserzione è:

4 8 7 24 10 17 12 80 47 26 23 45 81 41 35 200

Lo heap dopo il cambio di priorità è:

4 10 7 24 23 17 12 80 47 26 36 45 81 41 35 200

Esercizio

Si inserisca la seguente sequenza di interi in una coda a priorità inizialmente supposta vuota:

11 31 24 9 98 90 5 13 88 77 34 65 1 76 48 55

Si ipotizzi di usare uno heap come struttura dati. Si disegni la struttura ai diversi passi dell'inserzione. Si ipotizzi che la priorità massima sia associata alla chiave a valore minimo.

Soluzione

La soluzione è la seguente:

1 11 5 13 34 9 24 31 88 98 77 90 65 76 48 55

Esercizio

Si inserisca la seguente sequenza di interi in una coda a priorità inizialmente supposta vuota:

10 30 76 33 64 0 75 47 54 23 8 97 89 4 12 87

Si ipotizzi di usare uno heap come struttura dati. Si disegni la struttura ai diversi passi dell'inserzione. Si ipotizzi che la priorità massima sia associata alla chiave a valore massimo.

Soluzione

La soluzione è la seguente:

97 87 89 64 33 76 30 54 47 23 8 0 75 4 12 10

4.3.4 Esercizi proposti**Esercizio**

Si inserisca la seguente sequenza di interi in una coda a priorità, inizialmente supposta vuota:

21 41 34 10 8 12 16 23 97 86 44 75 11 80 85 65

Si ipotizzi di usare uno heap come struttura dati. Si disegni la struttura ai diversi passi dell'inserzione. Si ipotizzi che la priorità massima sia associata alla chiave a valore massimo.

Esercizio

Si inserisca la seguente sequenza di interi in una coda a priorità, inizialmente supposta vuota:

21 41 34 10 8 12 16 23 97 86 44 75 11 80 85 65

Si ipotizzi di usare uno heap come struttura dati. Si disegni la struttura ai diversi passi dell'inserzione. Sulla struttura risultante si cambi la priorità di 41 in 36 e si disegni lo heap che ne deriva.

Si ipotizzi che la priorità massima sia associata alla chiave a valore minimo.

Capitolo 5

Alberi binari

Formalmente, un *albero* è un grafo aciclico, non orientato e connesso. In pratica, un albero è una struttura dati composta da *nodi*, contenenti usualmente dei campi di dato (tra cui un campo chiave) e quelli puntatore (i riferimenti), che individuano altri nodi della struttura. Nel caso di un *albero binario* ogni nodo può avere al più due puntatori, che individuano altrettanti nodi figli.

In questo capitolo analizzeremo questo tipo di struttura dati, gli algoritmi fondamentali per la loro gestione e una loro tipica applicazione.

5.1 Visita di alberi binari

5.1.1 Richiami di teoria

Visitare un albero significa elencare tutti i suoi nodi secondo una qualche strategia. Le strategie sono tre e si differenziano a seconda di quando è visitato il nodo radice rispetto ai sotto-alberi sinistro e destro. Le strategie possono essere implementate in forma ricorsiva o iterativa. Il codice sottostante descrive un'implementazione ricorsiva. Si individueranno il generico nodo dell'albero con un puntatore C denominato `link`. I figli sinistro e destro saranno quindi individuati dai puntatori `l` ed `r`, rispettivamente. Con `*visit` si farà riferimento a un'opportuna funzione di visita del nodo corrente. Si ricordi che l'albero binario è posizionale, quindi la visita del sotto-albero sinistro deve sempre precedere quella del sotto-albero destro, indipendentemente dal momento in cui si visita la radice.

▷ Strategia di visita in ordine infisso (in-order):

```
1 void inorder (link h, void (*visit) (link)) {
2     if (h==NULL)
3         return;
4     inorder (h->l, visit);
5     (*visit) (h);
6     inorder (h->r, visit);
7 }
8 }
```

▷ Strategia di visita in ordine anticipato (pre-order):

```

1 void preorder (link h, void (*visit) (link)) {
2     if (h==NULL)
3         return;
4
5     (*visit) (h);
6     preorder (h->l, visit);
7     preorder (h->r, visit);
8 }
```

▷ Strategia di visita in ordine posticipato (post-order):

```

1 void postorder (link h, void (*visit) (link)) {
2     if (h==NULL)
3         return;
4
5     postorder (h->l, visit);
6     postorder (h->r, visit);
7     (*visit) (h);
8 }
```

5.1.2 Esercizi svolti

Esercizio

Sia dato l'albero di Figura 5.1. Lo si visiti in ordine infisso (in-order), anticipato

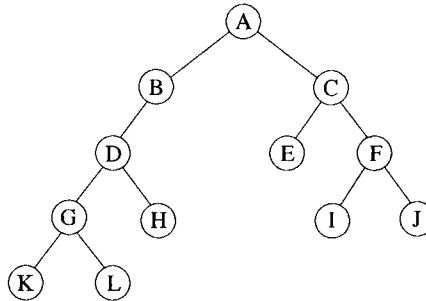


Figura 5.1

(pre-order), e posticipato (post-order). Si produca in uscita l'elenco dei nodi visitati.

Soluzione

Si esegue dapprima una visita in-order. Si incomincia visitando ricorsivamente il figlio sinistro di ciascun nodo fino a incontrare un figlio NULL. Questo accade quando si giunge al nodo K, che viene visitato (ovvero, in questo caso, semplicemente visualizzato). Si visita quindi il padre G, si scende al figlio destro L. Dato che i figli sono entrambi NULL, si stampa L. Avendo visitato il sotto-albero sinistro di D, si stampa D e si visita il sotto-albero destro. H non ha figli, lo si stampa. Il procedimento prosegue come descritto per ottenere la sequenza:

K G L D H B A E C I F J

Nell'esecuzione della visita in pre-order si stampa il valore del nodo corrente, prima di accedere al sotto-albero sinistro. Ne segue che la sequenza comincia con i valori A, B, D, G, K. Solo a questo punto si iniziano a visitare i sotto-alberi destri per completare la sequenza con i valori:

L H C E F I J

Nella visita in post-order, infine, la radice corrente viene stampata dopo aver fatto accesso ai sotto-alberi sinistro e destro. Si accede quindi in maniera ricorsiva al sotto-albero sinistro dei vari nodi a partire della radice A fino ad arrivare al nodo K. Il nodo K non ha figli e dunque è stampato. Si passa al sotto-albero destro di G, padre di K, che è L. L non ha figli e dunque viene stampato. Prima di stampare D si passa al suo sotto-albero destro, che è H e viene stampato. La sequenza prosegue in maniera analoga. La risposta corretta è la seguente:

K L G H D B E I J F C A

Esercizio

Sia dato un albero binario con 9 nodi. Nella visita si ottengono le tre seguenti sequenze di chiavi:

pre-order:	1 2 4 7 8 5 3 6 9
post-order:	7 8 4 5 2 9 6 3 1
in-order:	7 4 8 2 5 1 3 9 6

Si rappresenti graficamente l'albero binario di partenza.

Soluzione

Bisogna integrare le informazioni che si ricavano dalle tre visite, in modo da ricostruire l'albero originale. Dalla visita in pre-order si ricava che la radice dell'albero è 1. Da quella in-order si deduce che il sotto-albero sinistro contiene le chiavi 7, 4, 8, 2 e 5, quello destro le chiavi 3, 9 e 6. Tali considerazioni sono rappresentate in Figura 5.2(a).

Considerando il sotto-albero sinistro, dalla visita in pre-order si ricava che la radice dell'albero è 2, da quella in-order che il sotto-albero sinistro contiene le chiavi 7, 4, e 8, quello destro la chiave 5. Il risultato è rappresentato in Figura 5.2(b).

Considerando il sotto-albero sinistro, dalla visita in pre-order si ricava che la radice dell'albero è 4, da quella in-order che il sotto-albero sinistro contiene la chiave 7, quello destro la chiave 8. Il risultato è rappresentato in Figura 5.2(c).

Considerando il sotto-albero destro della radice 1, dalla visita in pre-order si ricava che la radice è 3, da quella in-order che il sotto-albero sinistro è vuoto e che quello destro contiene 9 e 6. Il risultato è rappresentato in Figura 5.2(d).

Considerando il sotto-albero destro, dalla visita in pre-order si ricava che la radice è 6, da quella in-order che il sotto-albero sinistro è contiene 9 e che quello destro è vuoto. Il risultato è rappresentato in Figura 5.2(e).

La visita in post-order conferma che l'albero costruito è corretto.

5.1.3 Esercizi risolti

Esercizio

Si visiti l'albero di Figura 5.3 in forma pre-order e post-order.

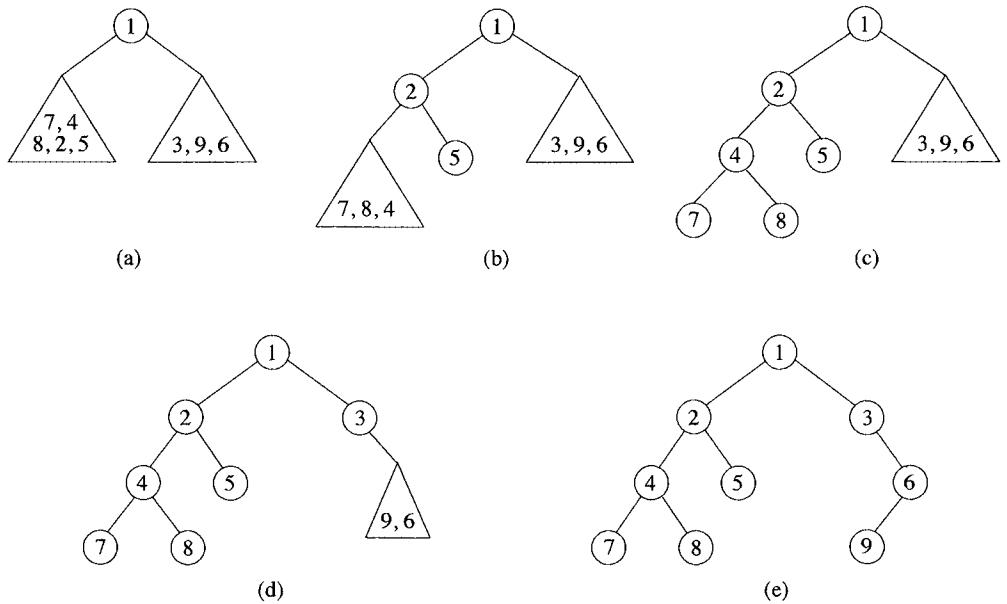


Figura 5.2 Procedimento di costruzione incrementale di un albero, dato il risultato delle sue visite in pre, post e in-order.

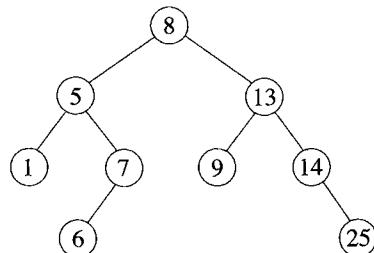


Figura 5.3

Soluzione

Si ottiene:

pre-order: 8 5 1 7 6 13 9 14 25
post-order: 1 6 7 5 9 25 14 13 8

5.1.4 Esercizi proposti

Esercizio

Si visiti l'albero di Figura 5.4. in pre-order, in-order e post-order. Si indichi l'elenco

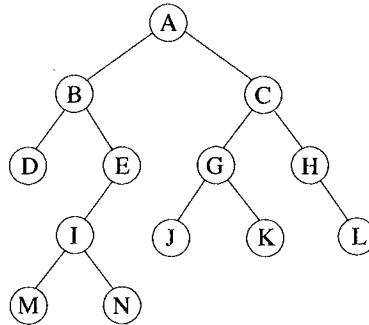


Figura 5.4

dei nodi in ordine di visita.

Esercizio

Si visiti l'albero di Figura 5.5. in pre-order, in-order e post-order. Si indichi l'elenco

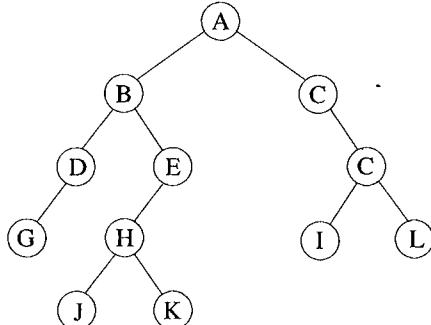
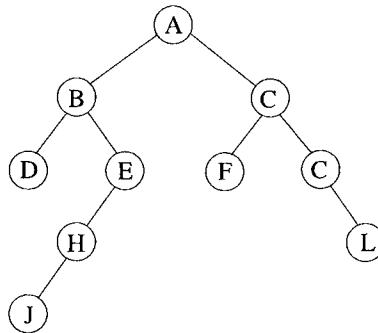


Figura 5.5

dei nodi in ordine di visita.

Esercizio

Si visiti l'albero di Figura 5.6. in pre-order, in-order e post-order. Si indichi l'elenco

**Figura 5.6**

dei nodi in ordine di visita.

Esercizio

Sia dato un albero binario con 12 nodi. Nella visita si ottengono le tre seguenti sequenze di chiavi:

pre-order:	10 15 20 7 40 9 1 0 134 21 5
post-order:	20 7 15 1 0 9 4 5 2113 40 10
in-order:	20 15 7 10 1 9 0 40 413 21 5

Si disegni l'albero binario di partenza.

5.2 Espressioni aritmetiche

5.2.1 Richiami di teoria

Un'applicazione rilevante delle visite degli alberi binari è la rappresentazione e eventuale valutazione di espressioni aritmetiche.

Si dice “arity” il numero di operandi relativi a un operatore.

Limitando a due la arity degli operatori $+$, $-$, $*$, $/$, viene spontaneo rappresentare espressioni aritmetiche tramite alberi binari, dove le foglie rappresentano gli operandi e i nodi interni gli operatori.

Facendo una visita in-order dell'albero si riottiene la stessa espressione aritmetica di partenza, ma senza parentesi. Facendo una visita in pre-order si ottiene un'espressione aritmetica in notazione pre-fissa, che è scarsamente utilizzata. Effettuando una visita in post-order si ottiene una rappresentazione in formato RPN (Reverse Polish Notation) che permette di valutare in maniera immediata operazioni aritmetiche se si ha a disposizione solo uno stack (struttura LIFO).

L'albero viene costruito dall'espressione in forma infissa sulla base della seguente grammatica:

```

> <exp> = <operand> | <exp> <op> <exp>
> <operand> = A ÷ Z
> <op> = + | * | - | /

```

5.2.2 Esercizi svolti

Esercizio

Si visita in pre-order e post-order l'albero binario corrispondente alla seguente espressione aritmetica:

$$\{(A/B) * (C + D)] * [E * (F - G)]\} / (H - I)$$

Soluzione

Innanzitutto si rappresenta l'espressione aritmetica come albero binario.

Si individuano `<exp>`, `<op>` e `<exp>` dove

$$\begin{aligned}
 <\text{op}> &= / \\
 <\text{exp}> &= \{(A/B) * (C + D)] * [E * (F - G)]\} \\
 <\text{exp}> &= (H - I)
 \end{aligned}$$

L'operatore / etichetta la radice, che ha come sotto-albero sinistro e destro rispettivamente le espressioni

$$\begin{aligned}
 &\{(A/B) * (C + D)] * [E * (F - G)]\} \\
 &(H - I)
 \end{aligned}$$

Ricorrendo sul sotto-albero sinistro, si individuano `<exp>` `<op>` `<exp>` dove

$$\begin{aligned}
 <\text{op}> &= * \\
 <\text{exp}> &= [(A/B) * (C + D)] \\
 <\text{exp}> &= [E * (F - G)].
 \end{aligned}$$

L'operatore * etichetta la radice, che ha come sotto-albero sinistro e destro rispettivamente le espressioni

$$\begin{aligned}
 &[(A/B) * (C + D)] \\
 &[E * (F - G)].
 \end{aligned}$$

Procedendo ricorsivamente si ottiene infine l'albero di Figura 5.7. A questo punto è su questa struttura dati che si deve effettuare la visita in pre-order e post-order, come spiegato nel paragrafo precedente. Il risultato che si ottiene è il seguente:

$$\begin{array}{l}
 \text{pre-order: } / * * / A B + C D * E F G - H I \\
 \text{post-order: } A B / C D + * E F G - * * H I - /
 \end{array}$$

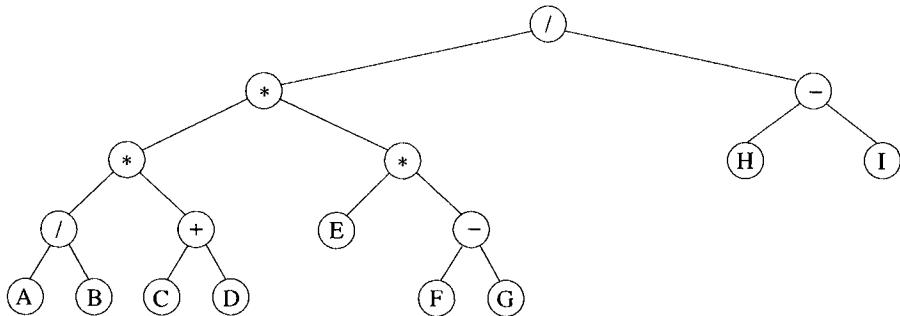


Figura 5.7 Costruzione di un albero data l'espressione aritmetica da esso rappresentata.

5.2.3 Esercizi risolti

Esercizio

Si visiti in pre-order, in-order e post-order l'albero binario corrispondente alla seguente espressione aritmetica:

$$[(A + B) * (C - D)] / [E + (F - G)]$$

Soluzione

Si ottiene:

pre-order:	/ * + A B - C D + E - F G
in-order:	A + B * C - D / E + F - G
post-order:	A B + C D - * E F G - + /

Esercizio

Si visiti in pre-order, in-order e post-order l'albero binario corrispondente alla seguente espressione aritmetica:

$$[(A - B) / (C - D)] / [E + (F - G)]$$

Soluzione

Si ottiene:

pre-order:	/ / - A B - C D + E - F G
in-order:	A - B / C - D / E + F - G
post-order:	A B - C D - / E F G - + /

Esercizio

Si visiti in pre-ordine, in-ordine e post-ordine l'albero binario corrispondente alla seguente espressione aritmetica:

$$[(A + B) * (C - D)] / [(E + F) * G]$$

Soluzione

Si ottiene:

pre-order: / * + A B - C D * + E F G
 in-order: A + B * C - D / E + F * G
 post-order: A B + C D - * E F + G * /

Esercizio

Si visiti in pre-order, in-order e post-order l'albero binario corrispondente alla seguente espressione aritmetica:

$$[(A - B) * (C - D)] - [(E + F)/(G - H)]$$

Soluzione

Si ottiene:

pre-order: - * - A B - C D / + E F - G H
 in-order: A - B * C - D - E + F / G - H
 post-order: A B - C D - * E F + G H - / -

5.2.4 Esercizi proposti

Esercizio

Sia data la seguente espressione in forma in-fissa.

$$A * \{[(B + C) * (D + E)] + F\}$$

La si trasformi in ordine pre-fisso e post-fisso visitando l'albero corrispondente. Si produca in uscita l'elenco dei nodi visitati.

Esercizio

Sia data la seguente espressione in forma in-fissa.

$$\{[(A + B) + C * (D + E)] + F\} * (G + H)$$

La si trasformi in ordine pre-fisso e post-fisso visitando l'albero corrispondente. Si produca in uscita l'elenco dei nodi visitati.

Esercizio

Sia data la seguente espressione in forma pre-fissa.

$$* A + B * C + D E$$

Si disegni l'albero corrispondente.

Esercizio

Sia data la seguente espressione in forma pre-fissa.

$$* A + * B + C D E$$

La si trasformi in ordine in-fisso e post-fisso visitando l'albero corrispondente. Si produca in uscita l'elenco dei nodi visitati.

Esercizio

Sia data la seguente espressione in forma in-fissa.

$$[(A * B + C) + D * (E + F * G)] * H + I$$

La si trasformi in ordine pre-fisso e post-fisso visitando l'albero corrispondente. Si produca in uscita l'elenco dei nodi visitati.

Capitolo 6

Alberi binari di ricerca

In questo capitolo si analizzeranno gli *alberi binari di ricerca*, ovvero alberi binari i cui nodi sono disposti ordinatamente a seconda del valore della chiave che li identifica. Su tale struttura dati presenteremo diverse tipologie di esercizi, che ne sfruttano e evidenziano le proprietà.

6.1 Correttezza della struttura dati

6.1.1 Richiami di teoria

Un albero binario di ricerca (BST, Binary Search Tree) è un tipo di dato astratto con due proprietà, una strutturale e l'altra funzionale:

- ▷ strutturalmente un BST è un albero binario
- ▷ funzionalmente per ogni terna root, left_child, right_child si ha che:

$$\text{key}(\text{root}) > \text{key}(\text{left_child}) \quad \&\& \quad \text{key}(\text{root}) < \text{key}(\text{right_child})$$

Ipotizzando il BST contenga dati di tipo `Item`, uno dei campi di `Item` è la chiave su cui è definita la relazione d'ordine. Per semplicità supporremo in tutti gli esempi seguenti che il tipo `Item` coincida con il tipo intero.

La funzione di ricerca di una chiave in un BST procede come segue. Si confronta la chiave cercata con quella del nodo corrente. Nel caso le chiavi coincidano la ricerca termina. In caso contrario, la procedura richiama se stessa ricorsivamente sul sotto-albero sinistro o destro del nodo corrente a seconda che la chiave cercata sia rispettivamente minore o maggiore della chiave del nodo corrente. La procedura di ricerca termina usualmente restituendo il puntatore al nodo con la chiave ricercata, nel caso essa sia presente nell'albero, oppure con un puntatore `NULL`, nel caso la chiave ricercata non sia presente nell'albero.

Dato un BST corretto contenente chiavi in un certo intervallo noto di valori si può verificare se una certa sequenza di chiavi esaminate alla ricerca di una certa chiave può esistere come cammino nel BST.

6.1.2 Esercizi svolti

Esercizio

Si supponga di aver memorizzato tutti i numeri compresi tra 1 e 1000 in un albero di ricerca binario e di stare cercando la chiave 363. Quali tra le seguenti sequenze non possono essere esaminate durante la ricerca?

```

924 220 911 244 898 258 362 363
925 202 911 240 912 245 363
    2 399 387 219 266 382 381 278 363
935 278 347 621 299 392 358 363
    2 252 401 398 330 344 397 363

```

Soluzione

La prima sequenza è possibile, in quanto la proprietà funzionale è rispettata come si vede dal cammino rappresentato in Figura 6.1(a).

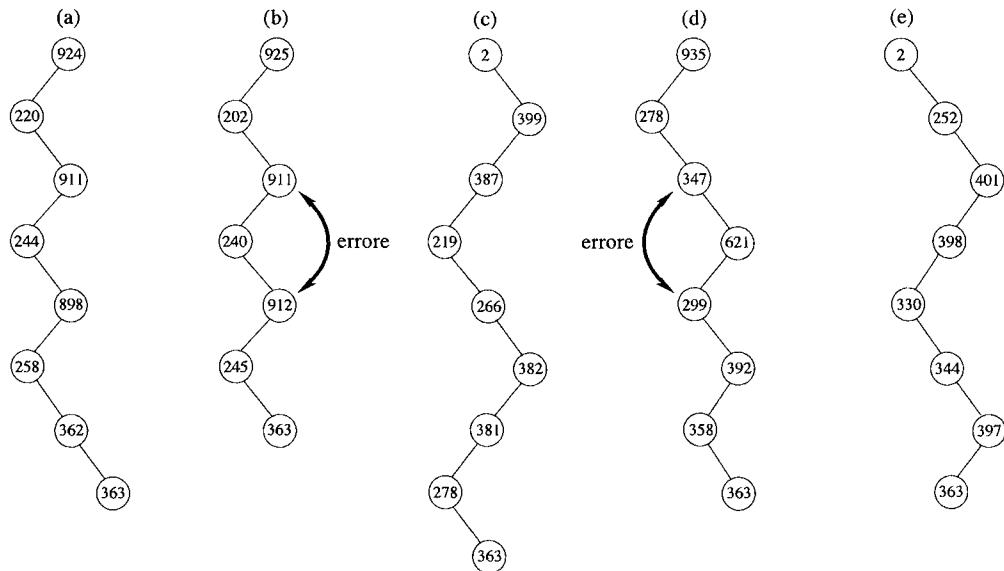


Figura 6.1 Valutazione di sequenze di ricerca lungo un BST.

Il cammino della seconda sequenza è rappresentato in Figura 6.1(b). Come si vede nel sotto-albero sinistro di 911, dove ci dovrebbero essere solo chiavi minori, appare 912. È stata violata la proprietà funzionale del BST e quella proposta non può essere una possibile sequenza di ricerca.

Nella terza sequenza, come si vede in Figura 6.1(c), non c'è nessuna violazione.

Per la penultima sequenza, analizzata in Figura 6.1(d), nel sotto-albero destro di 347, dove ci dovrebbero essere solo chiavi maggiori, appare 299. Anche in questo caso è stata violata la proprietà funzionale del BST e quella proposta non può essere una possibile sequenza di ricerca.

È facile osservare che con l'ultima sequenza, riportata in Figura 6.1(e), non c'è violazione di alcuna proprietà funzionale.

6.1.3 Esercizi risolti

Esercizio

Si supponga di aver memorizzato tutti i numeri compresi tra 1 e 500 in un albero di ricerca binario correttamente costruito e che si stia cercando la chiave 161. Quali tra queste non possono essere le sequenze esaminate durante la ricerca?

4 150 410 290 310 450 389 161
98 100 200 120 350 230 300 161

Soluzione

Entrambe.

Esercizio

Si supponga di aver memorizzato tutti i numeri compresi tra 1 e 1000 in un albero di ricerca binario correttamente costruito e che si stia cercando la chiave 261. Quali tra queste non possono essere le sequenze esaminate durante la ricerca?

4 250 410 390 310 350 389 261
998 100 800 120 750 200 300 261

Soluzione

La prima.

Esercizio

Si supponga di aver memorizzato tutti i numeri compresi tra 1 e 100 in un albero di ricerca binario correttamente costruito e che si stia cercando il numero 61. Quali tra queste non possono essere le sequenze esaminate durante la ricerca?

4 15 41 29 31 45 39 61
· 98 100 20 12 35 23 30 61

Soluzione

Entrambe.

6.1.4 Esercizi proposti

Esercizio

Si supponga di aver memorizzato tutti i numeri compresi tra 1 e 1000 in un albero di ricerca binario e che si stia cercando la chiave 363. Quali tra queste non possono essere

le sequenze esaminate durante la ricerca?

4	250	410	390	310	350	389	363	
998	100	800	120	750	200	300	363	
511	250	600	244	898	258	362	363	
1	500	410	120	251	380	369	271	363
700	210	357	891	299	392	358	363	

Esercizio

Si supponga di aver memorizzato tutti i numeri compresi tra 1 e 1000 in un albero di ricerca binario e che si stia cercando la chiave 213. Quali tra queste non possono essere le sequenze esaminate durante la ricerca?

14	250	490	390	310	371	389	213	
81	500	231	120	451	380	569	271	213
998	100	800	130	750	200	300	213	
700	210	357	891	287	392	388	213	
211	250	683	244	898	258	342	213	

Esercizio

Si supponga di aver memorizzato tutti i numeri compresi tra 1 e 1000 in un albero di ricerca binario e che si stia cercando la chiave 897. Quali tra queste non possono essere le sequenze esaminate durante la ricerca?

14	250	490	130	750	371	389	897	
81	500	451	380	231	120	569	271	897
998	100	800	390	310	200	300	897	
700	891	287	392	388	210	357	897	

6.2 Operazioni sui BST

6.2.1 Richiami di teoria

Ogni nodo del BST contiene l'oggetto memorizzato di tipo `Item`, di cui uno dei campi è la chiave, il puntatore al padre, i due puntatori ai figli sinistro e destro e un contatore del numero di nodi contenuti nell'albero ivi radicato. Il codice riportato in seguito fa uso di nodi sentinella (nodi finti nulli in fondo all'albero).

Ricerca di una chiave

La procedura di ricerca può essere espressa in forma iterativa o ricorsiva.

La procedura ricorsiva percorre l'albero a partire dalla radice. La condizione di terminazione è quella di avere trovato nel nodo corrente una chiave uguale a quella cercata o di essere giunti al fondo di un cammino, cioè di aver trovato un nodo sentinella. Altrimenti dal nodo corrente la ricorsione prosegue sul sotto-albero sinistro se la chiave cercata è minore di quella del nodo corrente, in quello destro se è maggiore.

Il codice seguente riporta la procedura ricorsiva.

```

1 Item BSTsearch (BST bst, Key k) {
2     return searchR (bst->head, k, bst->z);
3 }
4
5 Item searchR (link h, Key k, link z) {
6     if (h==z)
7         return NULLitem;
8     if (eq (k, key(h->item)))
9         return h->item;
10    if (less (k, key(h->item)))
11        return searchR (h->l, k, z);
12    else
13        return searchR (h->r, k, z);
14 }
```

Inserimento in una foglia

La procedura di inserimento in una foglia può essere espressa in forma iterativa o ricorsiva. Tali procedure fanno normalmente distinzione tra due possibili situazioni:

- ▷ se l'albero è vuoto, il nuovo elemento da inserire diventa la radice dell'albero.
- ▷ se l'albero già esiste, si identifica in quale posizione dovrebbe finire la chiave costruendo un percorso dalla radice alle foglie, confrontando la chiave da inserire con la chiave del nodo corrente e poi scendendo (iterativamente o ricorsivamente) nel sotto-albero sinistro o destro a seconda dell'esito del confronto.

Il codice seguente riporta la procedura iterativa:

```

1 void BSTinsert_leafI (BST bst, Item x) {
2     link p = bst->head, h = p;
3
4     if (bst->head==bst->z) {
5         bst->head = NEW (x, bst->z, bst->z, bst->z, 1);
6         bst->N++;
7         return;
8     }
9     while (h!=bst->z) {
10        p = h;
11        h->N++;
12        h = less (key(x), key (h->item)) ? h->l : h->r;
13    }
14    h = NEW (x, p, bst->z, bst->z, 1);
15    bst->N++;
16    if (less (key(x), key(p->item)))
17        p->l = h;
18    else
19        p->r = h;
20 }
```

Il codice seguente riporta la procedura ricorsiva:

```

1 void BSTinsert_leafR(BST bst, Item x) {
2     bst->head = InsertR(bst->head, x, bst->z); bst->N++;
3 }
4
5 link insertR(link h, Item x, link z) {
6     if (h == z)
7         return NEW(x, z, z, z, 1);
8     if (less(key(x), key(h->item))) {
9         h->l = insertR(h->l, x, z);
10        h->l->p = h;
11    }
12    else {
13        h->r = insertR(h->r, x, z); h->r->p = h;
14    }
```

```

15     (h->N)++;
16     return h;
17 }

```

Nell'esempio di Figura 6.2 si inserisce la chiave 13. Il cammino mostra, in funzione dei confronti effettuati, la localizzazione dove sarà inserita la chiave 13.

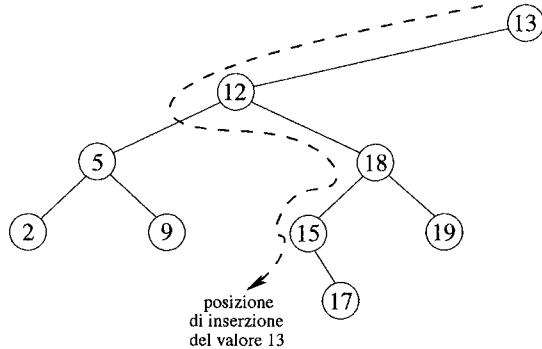


Figura 6.2 Sequenze di ricerca per l'inserzione della chiave 13.

Rotazione a sinistra o a destra

Le rotazioni a sinistra o a destra sono operazioni che modificano localmente la topologia del BST, mantenendo inalterata la proprietà funzionale. Sono utilizzate all'interno di altre operazioni di più alto livello, quali l'inserzione in radice, il partizionamento e la cancellazione.

La rotazione a sinistra fa sì che la vecchia radice y diventi il sotto-albero sinistro della nuova radice x , che era il vecchio sotto-albero destro di y . È inoltre necessario che il vecchio sotto-albero sinistro di x diventi il nuovo sotto-albero destro di y .

La Figura 6.3 illustra il funzionamento della procedura.

Si noti che il codice seguente, oltre alle modifiche topologiche, aggiorna anche il campo del nodo che contiene il numero di nodi del sotto-albero ivi radicato.

```

1 link rotL(link h) {
2     link x = h->r;
3     h->r = x->l;
4     x->l->p = h;
5     x->l = h;
6     x->p = h->p;
7     h->p = x;
8     x->N = h->N;
9     h->N = h->l->N + h->r->N +1;
10    return x;
11 }

```

La rotazione a destra fa sì che la vecchia radice y diventi il sotto-albero destro della nuova radice x , che era il vecchio sotto-albero sinistro di y . È inoltre necessario che il vecchio sotto-albero destro di x diventi il nuovo sotto-albero sinistro di y .

La Figura 6.4 illustra il funzionamento della procedura.

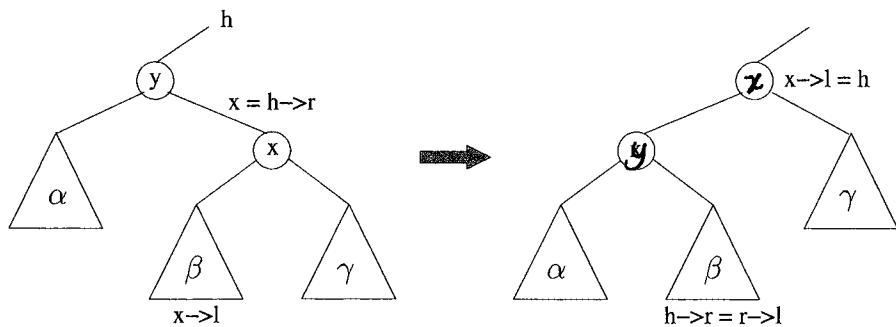


Figura 6.3 Rotazione destra. SINISTRA

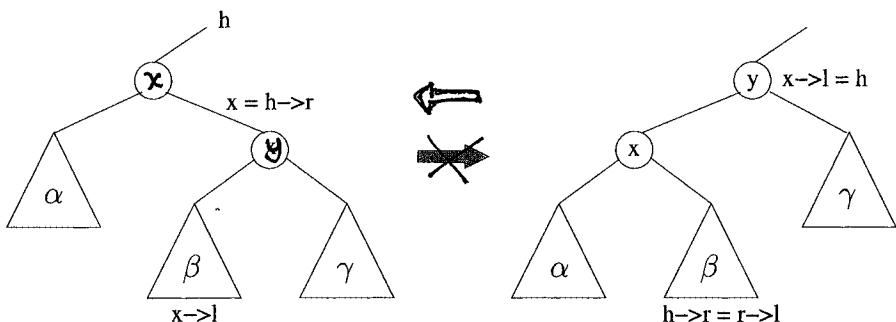


Figura 6.4 Rotazione destra.

Si noti che il codice seguente, oltre alle modifiche topologiche, aggiorna anche il campo del nodo che contiene il numero di nodi del sotto-albero ivi radicato.

```

1 link rotR(link h) {
2     link x = h->l;
3     h->l = x->r;
4     x->r->p = h;
5     x->r = h;
6     x->p = h->p;
7     h->p = x;
8     x->N = h->N;
9     h->N = h->r->N + h->l->N + 1;
10    return x;
11 }
```

Inserimento in radice

L'inserimento in radice procede ricorsivamente:

- ▷ se l'albero è vuoto, il nuovo elemento da inserire diventa la radice dell'albero
- ▷ se l'albero già esisteva, si confronta la chiave da inserire con quella del nodo corrente e poi si scende ricorsivamente nel sotto-albero sinistro o destro a seconda dell'esito del confronto. Se si è scesi nel sotto-albero sinistro, quando si ritorna dalla chiamata ricorsiva si applica una rotazione a destra, se si è scesi nel sotto-albero destro, quando si ritorna dalla chiamata ricorsiva si applica una rotazione a sinistra.

Il codice relativo è il seguente:

```

1 void BSTinsert_root (BST bst, Item x) {
2     bst->head = insertT (bst->head, x, bst->z); bst->N++;
3 }
4
5 link insertT (link h, Item x, link z) {
6     if (h==z)
7         return NEW (x, z, z, z, 1);
8
9     if (less (key (x), key (h->item))) {
10        h->l = insertT (h->l, x, z);
11        h = rotR(h);
12        h->N++;
13    } else {
14        h->r = insertT (h->r, x, z);
15        h = rotL (h);
16        h->N++;
17    }
18    return h;
19 }
```

Selezione

La funzione seleziona la k -esima chiave più piccola ($k = 0$ chiave minima) del BST. Utilizza l'informazione memorizzata in ciascun nodo sul numero di nodi dell'albero ivi radicato. Sia t il numero di nodi del sotto-albero sinistro. La condizione di terminazione è $t = k$: in questo caso si ritorna la radice del sotto-albero. Se $t > k$, si ricorre nel sotto-albero sinistro alla ricerca della k -esima chiave più piccola, altrimenti se $t < k$, si ricorre nel sotto-albero destro alla ricerca della $(k-t-1)$ -esima chiave più piccola. Il codice relativo è il seguente:

```

1 Item BSTselect (BST bst, int k) {
2     return selectR (bst->head, k, bst->z);
```

```

3 }
4
5 Item selectR (link h, int k, link z) {
6     int t;
7
8     if (h==z)
9         return NULLitem;
10    t = (h->l == z) ? 0 : h->l->N;
11    if (t>k)
12        return selectR (h->l, k, z);
13    if (t<k)
14        return selectR (h->r, k-t-1, z);
15
16    return h->item;
17 }
```

Partizionamento

La funzione riorganizza l'albero avendo la k -esima chiave più piccola nella radice. Si pone il nodo come radice di un sotto-albero:

- ▷ se $t > k$: si scende ricorsivamente nel sotto-albero sinistro, eseguendo un partizionamento rispetto alla k -esima chiave più piccola e al termine si esegue una rotazione a destra
- ▷ se $t < k$: si scende ricorsivamente nel sotto-albero destro, eseguendo un partizionamento rispetto alla $(k - t - 1)$ -esima chiave più piccola e al termine si esegue una rotazione a sinistra.

Il codice relativo è il seguente:

```

1 link partR (link h, int k) {
2     int t = h->l->N;
3
4     if (t>k) {
5         h->l = partR (h->l, k);
6         h = rotR (h);
7     }
8     if (t<k) {
9         h->r = partR (h->r, k-t-1);
10        h = rotL (h);
11    }
12
13    return h;
14 }
```

Cancellazione

Per cancellare da un albero binario di ricerca un nodo con data chiave bisogna discendere ricorsivamente sul sotto-albero opportuno fino a quando la chiave da cancellare non diventa la radice del sotto-albero corrente. A questo punto si cancella il nodo e si ricombinano i due sotto-alberi attraverso la funzione `joinLR`, che al suo interno richiama la funzione `part` per far diventare radice del sotto-albero destro il nodo che contiene il successore della chiave da cancellare. Il codice delle funzione `BSTdelete` e `deleteR` è il seguente:

```

1 void BSTdelete (BST bst, Item x) {
2     bst->head = deleteR (bst->head, x, bst->z);
3     bst->N--;
4 }
5
6 link deleteR (link h, Item x, link z) {
```

```

7     link y;
8
9     if (h==z)
10    return z;
11    if (less (key (x), key (h->item)))
12      h->l = deleteR (h->l, x, z);
13    if (less (key (h->item), key (x)))
14      h->r = deleteR (h->r, x, z);
15    (h->N)--;
16    if (eq (key (x), key (h->item))) {
17      y = h;
18      h = joinLR (h->l, h->r, z);
19      free (y);
20    }
21  return h;
22 }
```

Il codice delle funzione `joinLR` è il successivo:

```

1 link joinLR (link a, link b) {
2   if (b == z)
3     return a;
4
5   b = partR (b, 0);
6   b->l = a;
7   b->p = z;
8   a->p = b;
9   b->N = a->N + b->r->N + 1;
10
11  return b;
12 }
```

6.2.2 Esercizi svolti

Esercizio

Si effettuino secondo l'ordine specificato le seguenti operazioni di inserzione su un BST supposto inizialmente vuoto¹:

+10 + 63 + 11 + 15 + 62 + 84 + 9

Le inserzioni vengano effettuate nelle foglie.

Soluzione

Dopo aver creato la radice a partire dall'albero vuoto, a ogni passo successivo si determina dove deve finire la chiave da inserire e poi si procede all'inserimento. La Figura 6.5 illustra i vari passi.

Esercizio

Si effettuino secondo l'ordine specificato le seguenti operazioni di inserzione su un BST supposto inizialmente vuoto:

+5 + 17 + 0 + 23 + 26 + 50 + 9

Le inserzioni vengano effettuate nelle foglie.

Soluzione

La Figura 6.6 illustra il risultato finale.

¹ Per convenzione da questo momento in poi per tutto il resto del volume indicheremo con $+x$ e con $-x$ l'inserzione e l'estrazione della chiave x dal BST.

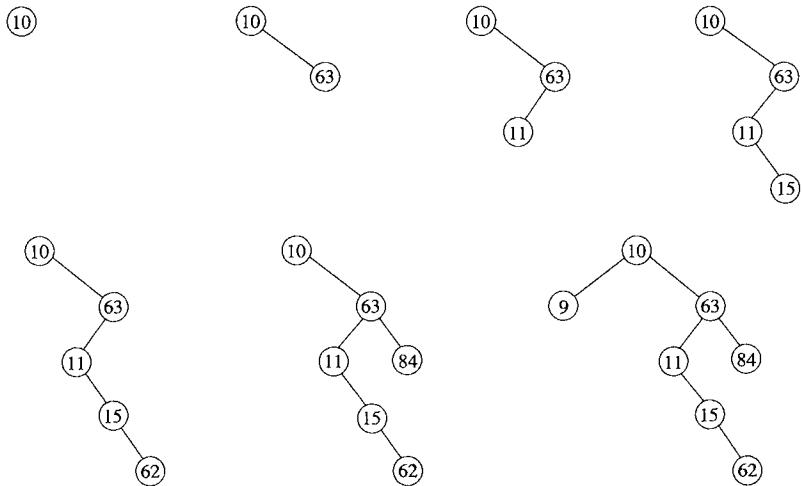


Figura 6.5

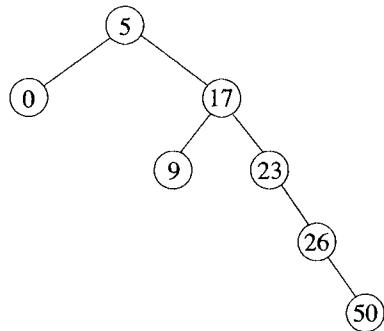


Figura 6.6

Esercizio

Si ruoti a destra il BST di Figura 6.7 attorno alla coppia di nodi con chiavi S e D.

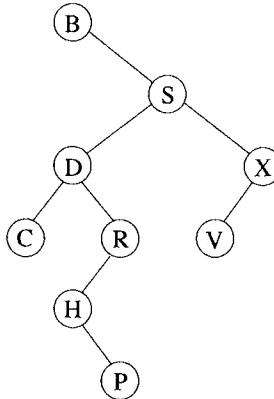


Figura 6.7

Soluzione

La rotazione è verso destra: nella Figura 6.8 a sinistra sono evidenziati i sotto-alberi α , β e γ . Nella figura Figura 6.8 a destra D è diventato padre di S e S ne è il figlio destro. I sotto-alberi evidenziati sono rimasti invariati. Il sotto-albero B è diventato figlio sinistro di S .

Esercizio

Si inserisca nella radice del BST di Figura 6.9 la chiave 11, riportando a ogni passo l'albero in Figura 6.9.

Soluzione

Come primo passo si procede con l'inserzione di 11 nella foglia appropriata. In seguito si applicano le rotazioni per far risalire la chiave 11 sino alla radice. La rotazione è verso destra se l'inserimento era avvenuto nel sotto-albero sinistro, verso sinistra se era avvenuta nel sotto-albero destro.

La soluzione è riportata in Figura 6.10.

Esercizio

Si partizioni il BST di Figura 6.11 attorno alla chiave R.

Soluzione

Si procede con una serie di rotazioni che coinvolgono R e il suo nodo padre: a sinistra per D-R, a destra per S-R, a sinistra per B-R. La soluzione è riportata in Figura 6.12.

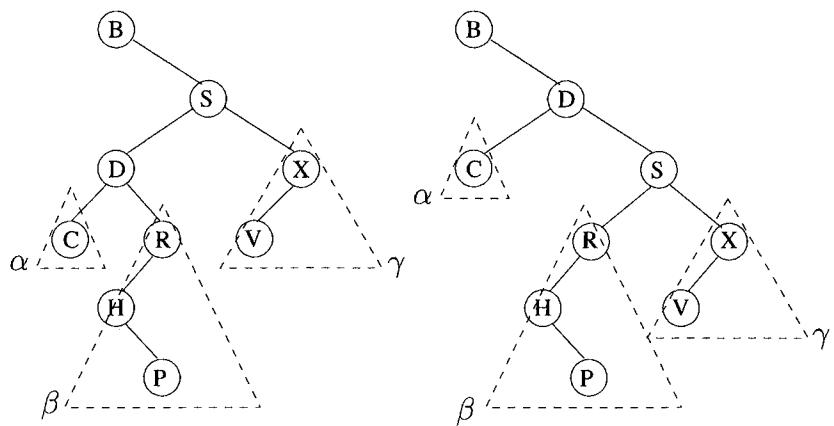


Figura 6.8

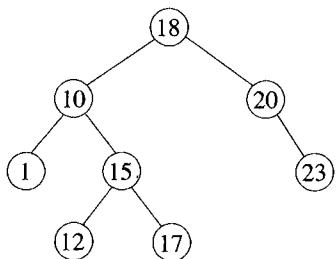


Figura 6.9

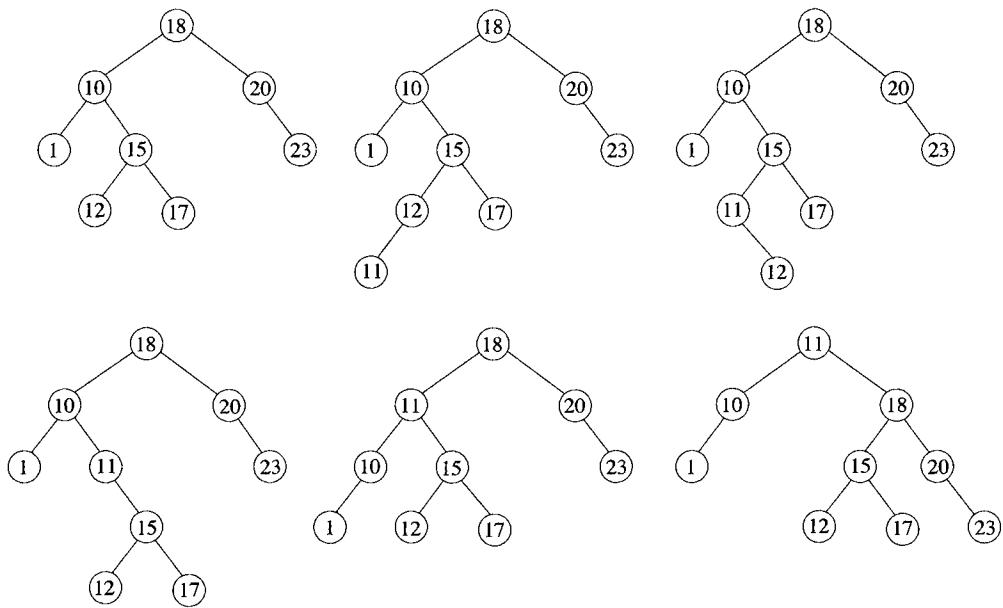


Figura 6.10

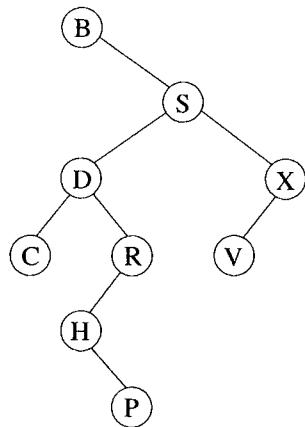


Figura 6.11

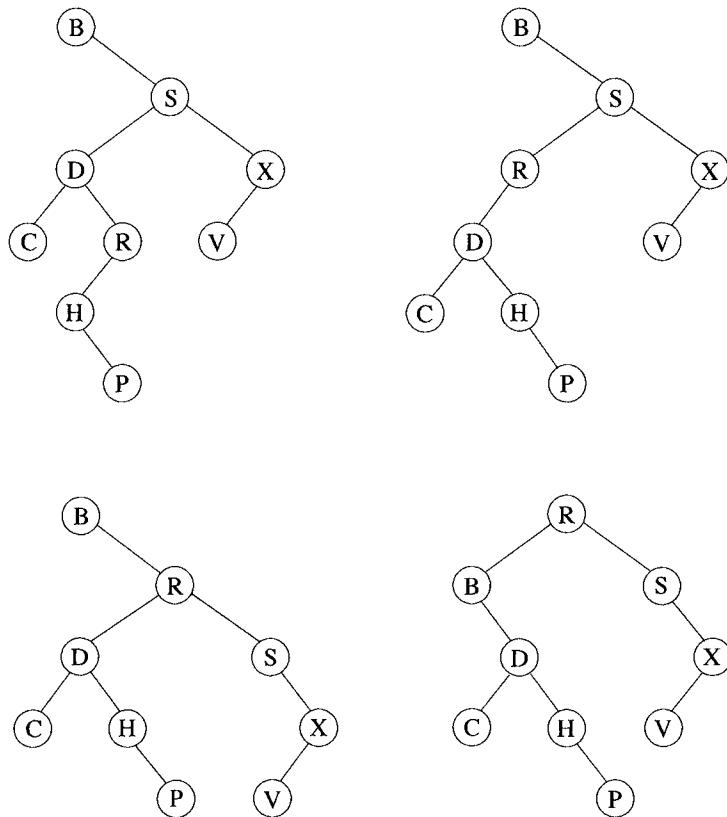


Figura 6.12

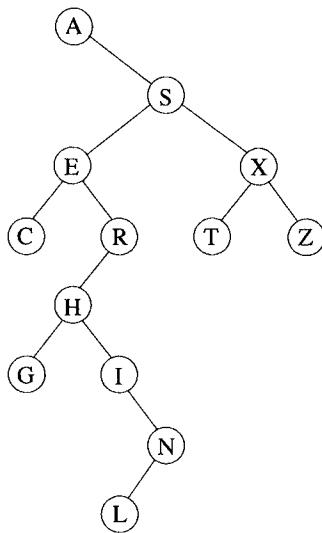


Figura 6.13

Esercizio

Si cancelli la chiave S dal BST di Figura 6.13.

Soluzione

Innanzitutto si cancella S, sconnettendo l'albero. Poi mediante rotazioni si porta T, il successore di S, nella radice, lo si collega ad A come suo figlio destro e infine si collega l'altro sotto-albero come figlio sinistro di T.

La soluzione è in Figura 6.14.

6.2.3 Esercizi proposti**Esercizio**

Sia dato l'albero di ricerca binario di Figura 6.15:

- ▷ lo si disegni dopo la cancellazione della chiave 8
- ▷ a partire dall'albero originale, lo si disegni dopo la cancellazione della chiave 14 e l'inserzione in una foglia della chiave 11.

Esercizio

Si effettuino secondo l'ordine specificato le seguenti operazioni di inserzione (+) ed estrazione (-) su un BST supposto inizialmente vuoto:

+19 + 21 + 4 + 13 + 17 + 2 - 19

Le inserzioni sono in foglia.

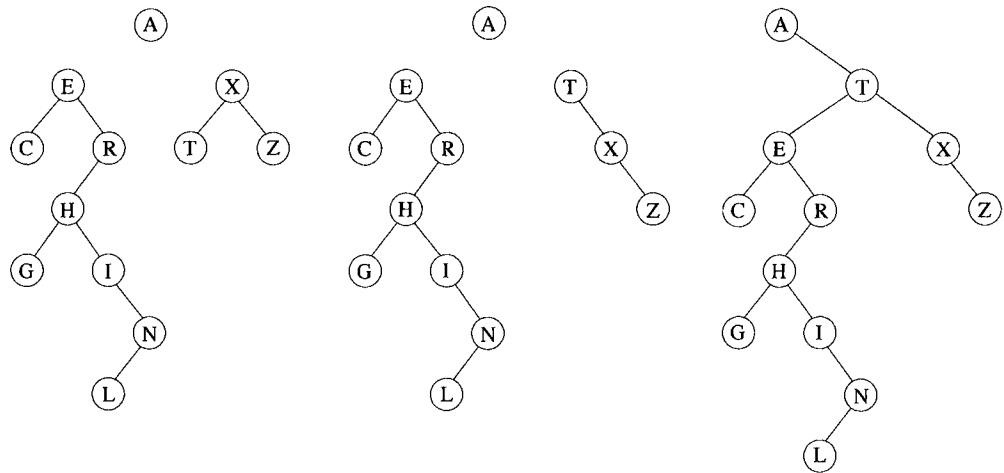


Figura 6.14

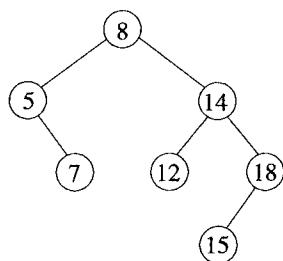


Figura 6.15

Esercizio

Si inseriscano in sequenza nella radice di un BST, inizialmente supposto vuoto, le chiavi:

+11 + 31 + 24 + 9 + 98 + 90 + 5 + 13 + 88 + 77

Esercizio

Si inseriscano in radice nel BST di Figura 6.16 la sequenza le chiavi:

+11 + 4 + 9 + 25

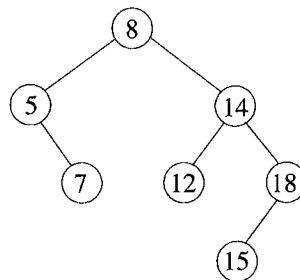


Figura 6.16

Esercizio

Si partizionino il BST di Figura 6.17 attorno alla chiave 14.

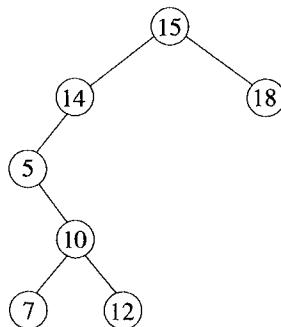


Figura 6.17

Esercizio

Si partizionino il BST di Figura 6.18 attorno alla chiave mediana.

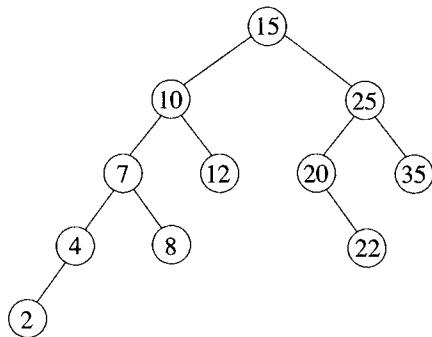


Figura 6.18

Esercizio

Si cancellino in sequenza le chiavi E, R e A dal BST di Figura 6.19.

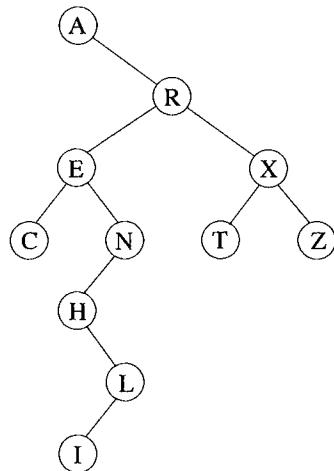


Figura 6.19

Capitolo 7

Le tabelle di hash

Le strutture dati analizzate nei capitoli precedenti seguono un meccanismo di funzionamento basato essenzialmente sul confronto di chiavi. Una ricerca basata su *hashing* offre invece la possibilità di accedere in modo diretto agli elementi memorizzati. L'idea è fondamentalmente quella di generalizzare le strutture dati vettoriali, in modo che, manipolando opportunamente la chiave di un dato, venga reperito l'indice dell'elemento in cui il dato associato alla chiave deve essere (oppure è stato) memorizzato. Analizzare tale generalizzazione sarà l'obiettivo principale di questo capitolo.

7.1 Hashing

7.1.1 Richiami di teoria

L'hashing è un metodo che consente di indirizzare in maniera diretta i record contenuti in una tabella eseguendo alcune trasformazioni aritmetiche sulle chiavi per ottenere gli indici relativi. Nei casi reali si deve come prima cosa determinare la dimensione M della tabella. Negli esercizi successivi M è fornito come dato in ingresso. Si ricordi comunque che per il *linear chaining* è opportuno che M sia il più piccolo numero primo maggiore o uguale al numero di chiavi massimo diviso 5 (o 10). In questo modo la lunghezza media delle liste sarà limitata a 5 (o 10) garantendo sufficiente efficienza. Per l'*open addressing* invece, M è bene sia il più piccolo numero primo maggiore o uguale al doppio del numero massimo di chiavi memorizzato nella tabella.

Bisogna poi individuare una funzione in grado di trasformare una chiave di ricerca (che può essere un intero, un reale, una carattere o una stringa) in un indice nella tabella (cioè in un valore compreso tra 0 e $M - 1$). La funzione dipende dal tipo della chiave. Negli esercizi successivi, se le chiavi sono singoli caratteri, la funzione ritorna il loro offset rispetto alla prima lettera dell'alfabeto inglese, se sono interi, ritorna il valore stesso.

Idealmente chiavi diverse dovrebbero essere trasformate in indici differenti, ma per quanto buona sia la funzione di hash prima o poi per $k_1 \neq k_2$ accadrà che $h(k_1) = h(k_2)$, si avrà cioè una "collisione".

Vi sono due strategie di alto livello per gestire le collisioni:

- ▷ il *linear chaining*

▷ l'*open addressing*.

Nel linear chaining gli elementi che collidono sono inseriti essenzialmente in liste concatenate. Nel caso di collisione, l'inserimento avviene in testa alla lista corretta per ridurre il costo di inserzione. Nell'*open addressing* tutte le chiavi sono memorizzate nella tabella di hash. Per inserire una nuova chiave nella tabella, se ne calcola la funzione di hash e si verifica se la cella corrispondente è piena o vuota. Se è vuota, l'inserzione ha immediatamente luogo, se è piena, quindi c'è collisione, si esaminano in successione (probing) gli slot del vettore secondo una certa legge fino a trovarne uno vuoto in cui mettere la chiave. Le tecniche di campionamento più comuni sono:

- ▷ *linear probing*
- ▷ *quadratic probing*
- ▷ *double hashing*.

Nei casi reali la tabella viene dimensionata in modo da garantire sempre l'esistenza di almeno uno slot libero per effettuare l'inserzione.

La funzione di hash primaria $h(k)$ viene specificata negli esercizi ed è tipicamente della forma:

$$h(k) = k \% M \quad (k \bmod M)$$

con M numero primo.

7.2 Linear Chaining

7.2.1 Richiami di teoria

Nel linear chaining concettualmente più elementi possono risiedere nella stessa locazione della tabella. Questa è quindi un vettore di puntatori a liste concatenate. Nel caso di collisione, l'inserimento avviene in testa alla lista opportuna per garantire complessità $O(1)$ all'operazione. Il codice successivo schematizza l'inserzione in tabella.

```

1 void STinsert (ST st, Item item) {
2     int i = hash (key(item), st->M);
3     st->heads[i] = NEW (item, st->heads[i]);
4     return;
5 }
```

7.2.2 Esercizi svolti

Esercizio

Sia data la sequenza di chiavi intere

5 28 19 15 20 33 12 16 10

Si riporti la struttura di una tabella di hash, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare il linear chaining, che la tabella abbia 9 posizioni e che la funzione di hash sia $h(k) = k \bmod 9$.

Soluzione

Si incomincia il procedimento generando una tabella vuota, come rappresentato in Figura 7.1(a).

Le prime due chiavi 5 e 28 vengono memorizzate in testa alle liste i cui puntatori di testa sono nelle locazioni $5 \bmod 9 = 5$ e $28 \bmod 9 = 1$. Il risultato è rappresentato in Figura 7.1(b).

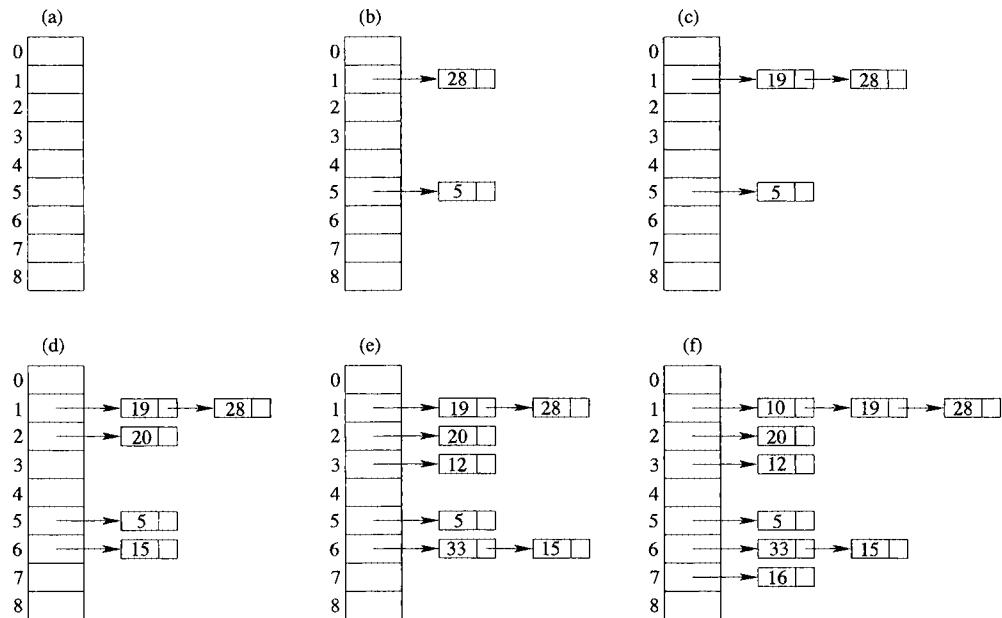


Figura 7.1 Inserzione di chiavi interi in un tabella di hash, gestita tramite linear chaining.

Con la chiave 19, poiché $19 \bmod 9 = 1$, si genera una collisione. Si accede alla lista opportuna e si inserisce in testa la chiave 19 (Figura 7.1(c)).

Non c'è alcuna collisione quando si inseriscono le chiavi 15 e 20 (Figura 7.1(d)).

Quando si tenta di inserire la chiave 33 si ha una collisione che viene risolta mediante inserzione in testa nella lista opportuna $33 \bmod 9 = 6$ (Figura 7.1(e)).

Si procede secondo questa strategia per le chiavi 12, 16 e 10. Il risultato finale è riportato in Figura 7.1(f).

7.2.3 Esercizi risolti

Esercizio

Sia data la sequenza di chiavi BELLEARTI, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash, inizialmente supposta vuota, in cui avvenga l'inserimento della

sequenza di cui sopra. Si supponga di utilizzare il linear chaining, che la tabella abbia 11 posizioni e che la funzione di hash sia $h(k) = k \bmod 11$.

Soluzione

In Tabella 7.1 si riporta per esteso l'alfabeto inglese in modo che sia immediato associare a ogni lettera il suo numero d'ordine.

A 1	B 2	C 3	D 4	E 5	F 6	G 7	H 8	I 9	J 10	K 11	L 12	M 13
N 14	O 15	P 16	Q 17	R 18	S 19	T 20	U 21	V 22	W 23	X 24	Y 25	Z 26

Tabella 7.1 Numero d'ordine dei caratteri alfabetici.

La soluzione è riportata in Figura 7.2. Si osservi che per distinguere le lettere identiche, tali lettere sono state numerate. Ad esempio, E_1 ed E_2 indicano la prima e la seconda lettera E della stringa BELLEARTI, rispettivamente.

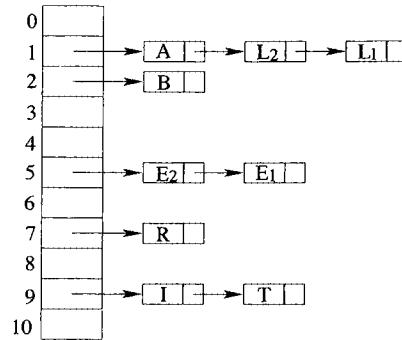


Figura 7.2

7.2.4 Esercizi proposti

Esercizio

Sia data la sequenza di chiavi MILLIONDOLLARBILL, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare il linear chaining, che la tabella abbia 13 posizioni e che la funzione di hash sia $h(k) = k \bmod 13$.

Esercizio

Sia data la sequenza di chiavi intere

27 14 23 1 19 7 85 14 53 19

Si riporti la struttura di una tabella di hash, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare il linear chaining, che la tabella abbia 19 posizioni e che la funzione di hash sia $h(k) = k \bmod 19$.

7.3 Open addressing con linear probing

7.3.1 Richiami di teoria

Si procede come segue:

- ▷ si calcola $i = h(k)$
- ▷ se la cella è libera, si inserisce la chiave. In caso contrario, si incrementa il valore di i di una unità, modulo la dimensione della tabella
- ▷ si ripete fino a trovare una cella vuota

Il codice seguente evidenzia tali operazioni:

```

1 #define full(A) (neq(key(st->a[A]), key(NULLitem)))
2
3 void STinsert (ST st, Item item) {
4     int i = hash (key (item), st->M);
5
6     while (full (i))
7         i = (i+1)%st->M;
8
9     st->a[i] = item;
10    st->N++;
11
12    return;
13 }
```

7.3.2 Esercizi svolti

Esercizio

Sia data la sequenza di chiavi SOLOINFOR, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 19, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con linear probing e che la funzione di hash primaria sia $h(k) = k \bmod 19$.

Soluzione

La prima lettera da considerare è S con numero d'ordine 19. Visto che $19 \bmod 19 = 0$ e la cella di indice 0 è vuota, in essa si memorizza S (Tabella 7.2(a)).

Non ci sono collisioni quando si inseriscono le successive due chiavi O e L, che vanno rispettivamente alla posizione 15 e 12 (Tabella 7.2(b)). Inserendo la successiva O avviene una collisione. La prima chiamata alla funzione di hash produce infatti di

(a)	(b)	(c)	(d)	(e)	(f)
0 S	0 S	0 S	0 S	0 S	0 S
1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
4	4	4	4	4	4
5	5	5	5	5	5
6	6	6	F	F	6 F
7	7	7	7	7	7
8	8	8	8	8	8
9	9	9	I	I	9 I
10	10	10	10	10	10
11	11	11	11	11	11
12	L	12 L	12 L	12 L	12 L
13	13	13	13	13	13
14	14	14	N	N	14 N
15	O	15 O	15 O ₁	15 O ₁	15 O ₁
16	16	16 O	16 O ₂	16 O ₂	16 O ₂
17	17	17	17	17 O ₃	17 O ₃
18	18	18		R	

Tabella 7.2

nuovo il valore 15. Il linear probing incrementa i di 1, ottenendo 16. Essendo la cella vuota, l'inserzione ha luogo e la situazione diventa quella di Tabella 7.2(c). Non ci sono collisioni con le 3 chiavi successive (Tabella 7.2(d)). Con la terza chiave O si ha di nuovo una collisione: per $i = 15$ c'è una prima collisione, si incrementa i di 1, ma si trova una collisione anche per $i = 16$, si incrementa i di 1 e si trova una cella libera in cui avviene l'inserzione per $i = 17$ (Tabella 7.2(e)). L'inserzione della chiave R all'indice 18 non genera collisioni. La soluzione corretta all'esercizio è riportata in Tabella 7.2(f).

7.3.3 Esercizi risolti

Esercizio

Sia data la sequenza di chiavi SDFGRSWD, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 17, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con linear probing e che la funzione di hash sia $h(k) = k \bmod 17$.

Soluzione

La soluzione è riportata in Tabella 7.3¹.

¹ Si osservi che in questo caso la tabella è disposta orizzontalmente, ottimizzando lo spazio occupato, mentre negli esercizi precedenti si era utilizzata una disposizione verticale, più ingombrante ma anche più comune.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	R	S ₁	S ₂	D ₁	D ₂	F	G ₁	W ₁								

Tabella 7.3

Esercizio

Sia data la sequenza di chiavi FONDAMENT, dove ciascun carattere è individuato dal suo ordine progressivo nell’alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 19, inizialmente supposta vuota, in cui avvenga l’inserimento della sequenza di cui sopra. Si supponga di utilizzare l’open addressing con linear probing e che la funzione di hash sia $h(k) = k \bmod 19$.

Soluzione

La soluzione è riportata in Tabella 7.4.

0	1	2	3	4	5	6	7	8
	A	T		D	E	F		

9	10	11	12	13	14	15	16	17	18
				M	N ₁	O	N ₂		

Tabella 7.4

Esercizio

Sia data la sequenza di chiavi TELECOMI, dove ciascun carattere è individuato dal suo ordine progressivo nell’alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 17, inizialmente supposta vuota, in cui avvenga l’inserimento della sequenza di cui sopra. Si supponga di utilizzare l’open addressing con linear probing e che la funzione di hash sia $h(k) = k \bmod 17$.

Soluzione

La soluzione è riportata in Tabella 7.5.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			T	C	E	E			I			L	M		O	

Tabella 7.5

7.3.4 Esercizi proposti**Esercizio**

Sia data la sequenza di chiavi EASYQUESTION, dove ciascun carattere è individuato dal suo ordine progressivo nell’alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura

di una tabella di hash di dimensione 29, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con linear probing e che la funzione di hash sia $h(k) = k \bmod 29$.

Esercizio

Sia data la sequenza di chiavi PASSEPARTOUT, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 29, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con linear probing e che la funzione di hash sia $h(k) = k \bmod 29$.

Esercizio

Sia data la sequenza di chiavi ADAYWITHOUTTRAIN, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 31, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con linear probing e che la funzione di hash sia $h(k) = k \bmod 31$.

7.4 Open addressing con quadratic probing

7.4.1 Richiami di teoria

Si procede come segue:

- ▷ si assegna 0 al contatore del numero di tentativi i
- ▷ si calcola $index = h(k)$
- ▷ se la cella è libera, si inserisce la chiave, altrimenti si incrementano i di 1 e $index$ di $c_1 i + c_2 i^2$ modulo la dimensione della tabella
- ▷ si ripete fino a trovare una cella vuota

come evidenziato dal codice seguente, dove $c_1 = 1$ e $c_2 = 1$:

```

1 #define full(A) (neq(key(st->a[A]), key(NULLitem)))
2
3 void STinsert (ST st, Item item) {
4     int index = hash (key (item), st->M);
5     int i = 0;
6
7     while (full (index)) {
8         i++;
9         index = (index + i + i*i)%st->M;
10    }
11    st->a[index] = item;
12    st->N++;
13
14    return;
15 }
```

7.4.2 Esercizi svolti

Esercizio

Sia data la sequenza di chiavi ADFAGDJD. Si riporti la struttura di una tabella di hash di dimensione 19, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con quadratic probing ($c_1 = 1$ e $c_2 = 1$) e che la funzione di hash sia $h(k) = k \bmod 19$.

Soluzione

La chiave A viene inserita in posizione 1 senza collisioni, come pure la chiave D in posizione 4 e la F in posizione 6 (Tabella 7.6(a)).

La successiva occorrenza di A genera una collisione all'indice 1. Si prova allora all'indice $1+1+1^2 = 3$, trovando la cella vuota, vi si memorizza la chiave (Tabella 7.6(b)). La chiave G viene inserita in posizione 7 senza collisioni (Tabella 7.6(c)).

La seconda occorrenza di D genera una collisione all'indice 4. Si prova allora all'indice $4+1+1^2 = 6$, trovando ancora una collisione. Al successivo campionamento all'indice $4+2+2^2 = 10$ la cella è vuota e vi si memorizza la chiave (Tabella 7.6(d)).

La chiave J genera una collisione all'indice 10, che viene risolta trovando una cella vuota all'indice $10+1+1^2 = 12$ (Tabella 7.6(e)).

La terza occorrenza di D genera una collisione agli indici 4, $4+1+1^2 = 6$ e $4+2+2^2 = 10$. Al successivo campionamento all'indice $4+3+3^2 = 16$ la cella è vuota e vi si memorizza la chiave (Tabella 7.6(f)).

(a)	(b)	(c)	(d)	(e)	(f)
0	0	0	0	0	0
1 A ₁	1 A ₁	1 A ₁	1 A ₁	1 A ₁	1 A ₁
2	2	2	2	2	2
3	3 A ₂	3 A ₂	3 A ₂	3 A ₂	3 A ₂
4 D ₁	4 D ₁	4 D ₁	4 D ₁	4 D ₁	4 D ₁
5	5	5	5	5	5
6 F	6 F	6 F	6 F	6 F	6 F
7	7	7 G	7 G	7 G	7 G
8	8	8	8	8	8
9	9	9	9	9	9
10	10	10	10 D ₂	10 D ₂	10 D ₂
11	11	11	11	11	11
12	12	12	12	12 J	12 J
13	13	13	13	13	13
14	14	14	14	14	14
15	15	15	15	15	15
16	16	16	16	16	16 D ₃
17	17	17	17	17	17
18	18	18	18	18	18

Tabella 7.6

7.4.3 Esercizi risolti

Esercizio

Sia data la sequenza di chiavi EOPNWVENBAY, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 23, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con quadratic probing ($c_1 = 1/2$ e $c_2 = 1/2$) e che la funzione di hash sia $h(k) = k \bmod 23$.

Soluzione

La soluzione è riportata in Tabella 7.7.

0	1	2	3	4	5	6	7	8	9	10
W	A	B	Y		E ₁	E ₂				
11	12	13	14	15	16	17	18	19	20	21
			N ₁	O	P	N ₂				V

Tabella 7.7

Esercizio

Sia data la sequenza di chiavi FGWDTS, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 13, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con quadratic probing ($c_1 = 0$ e $c_2 = 1$) e che la funzione di hash sia $h(k) = k \bmod 13$.

Soluzione

La soluzione è riportata in Tabella 7.8.

0	1	2	3	4	5	6	7	8	9	10	11	12
		S		D		F	G	T		W		

Tabella 7.8

7.4.4 Esercizi proposti

Esercizio

Sia data la sequenza di chiavi ILOOKATYOU, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 23, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con quadratic probing ($c_1 = 1$ e $c_2 = 1$) e che la funzione di hash sia $h(k) = k \bmod 23$.

Esercizio

Sia data la sequenza di chiavi SECONDCHANCE, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 29, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con quadratic probing ($c_1 = 0$ e $c_2 = 1$) e che la funzione di hash sia $h(k) = k \bmod 29$.

7.5 Open addressing con double hashing

7.5.1 Richiami di teoria

Nel double hashing, data una tabella di dimensione M :

- ▷ si calcola $i = h_1(k)$
- ▷ se la cella è libera, si inserisce la chiave, altrimenti si calcola $j = h_2(k)$, si aggiorna $i = (i + j) \bmod M$ e si prova per il valore di i così aggiornato
- ▷ si ripete fino a trovare una cella vuota, la cui esistenza è garantita dal corretto dimensionamento della tabella di hash rispetto al numero di chiavi da inserire.

Esempi di h_1 e h_2 :

$$\begin{aligned} h_1(k) &= k \bmod M \\ h_2(k) &= 1 + (k \bmod 97) \end{aligned}$$

con M il più piccolo numero primo maggiore o uguale al doppio del numero massimo di chiavi memorizzato nella tabella.

7.5.2 Esercizi svolti

Esercizio

Sia data la sequenza di chiavi ZABABBZ, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 23, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con double hashing. Sia a carico del candidato la scelta di $h_1(k)$ e $h_2(k)$.

Soluzione

La prima cosa da fare è scegliere le funzioni di hash $h_1(k)$ e $h_2(k)$. È evidente, come richiamato nel paragrafo precedente, che una possibile scelta sia:

$$\begin{aligned} h_1(k) &= k \bmod 23 \\ h_2(k) &= 1 + k \bmod 97 \end{aligned}$$

Le chiavi Z, A e B non generano collisioni (Tabella 7.9(a)).

La seconda chiave A genera collisione per $i = 1$, che si cerca di risolvere calcolando:

$$i = (1 + (1 + 1 \bmod 97)) \bmod 23 = 3$$

essendo piena la cella a questo indice, si passa al tentativo successivo

$$i = (3 + (1 + 1 \bmod 97)) \bmod 23 = 5$$

dove avviene l'inserimento (Tabella 7.9(b)).

La seconda chiave B genera collisione per $i = 2$, che si cerca di risolvere calcolando:

$$i = (2 + (1 + 2 \bmod 97)) \bmod 23 = 5$$

(a)	(b)	(c)	(d)	(e)
0	0	0	0	0
1 A ₁	1 A ₁	1 A ₁	1 A ₁	1 A ₁
2 B ₁	2 B ₁	2 B ₁	2 B ₁	2 B ₁
3 Z ₁	3 Z ₁	3 Z ₁	3 Z ₁	3 Z ₁
4	4	4	4	4
5	5 A ₂	5 A ₂	5 A ₂	5 A ₂
6	6	6	6	6
7	7	7	7	7 Z ₂
8	8	8 B ₂	8 B ₂	8 B ₂
9	9	9	9	9
10	10	10	10	10
11	11	11	11 B ₃	11 B ₃
12	12	12	12	12
13	13	13	13	13
14	14	14	14	14
15	15	15	15	15
16	16	16	16	16
17	16	16	16	16
18	16	16	16	16
19	16	16	16	16
20	16	16	16	16
21	16	16	16	16
22	16	16	16	16

Tabella 7.9

essendo piena la cella a questo indice, si passa al tentativo successivo

$$i = (5 + (1 + 2 \bmod 97)) \bmod 23 = 8$$

dove avviene l'inserimento (Tabella 7.9(c)).

La terza chiave B genera collisione per $i = 2$, $i = 5$ e $i = 8$. La collisione si risolve calcolando:

$$i = (8 + (1 + 2 \bmod 97)) \bmod 23 = 11$$

dove avviene l'inserimento (Tabella 7.9(d)).

La seconda chiave Z genera collisione per $i = 3$. La collisione si risolve calcolando:

$$i = (3 + (1 + 26 \bmod 97)) \bmod 23 = 7$$

dove avviene l'inserimento (Tabella 7.9(e)).

La soluzione finale è rappresentata in Tabella 7.9(e).

7.5.3 Esercizi risolti

Esercizio

Sia data la sequenza di chiavi HASHTABLE, dove ciascun carattere è individuato dal suo ordine progressivo nell’alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 23, inizialmente supposta vuota, in cui avvenga l’inserimento della sequenza di cui sopra. Si supponga di utilizzare l’open addressing con double hashing. Sia a carico del candidato la scelta di $h_1(k)$ e $h_2(k)$.

Soluzione

Utilizzando come funzioni di hash:

$$\begin{aligned} h_1(k) &= k \bmod 23 \\ h_2(k) &= 1 + k \bmod 97 \end{aligned}$$

si ottiene la soluzione riportata in Figura 7.10.

0	1	2	3	4	5	6	7	8	9	10	11
	A ₁	B	A ₂		E			H ₁			
12	13	14	15	16	17	18	19	20	21	22	22
L					H ₂		S	T			

Tabella 7.10

Esercizio

Sia data la sequenza di chiavi INFORMATICA, dove ciascun carattere è individuato dal suo ordine progressivo nell’alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 23, inizialmente supposta vuota, in cui avvenga l’inserimento della sequenza di cui sopra. Si supponga di utilizzare l’open addressing con double hashing. Sia a carico del candidato la scelta di $h_1(k)$ e $h_2(k)$.

Soluzione

Utilizzando come funzioni di hash:

$$\begin{aligned} h_1(k) &= k \bmod 23 \\ h_2(k) &= 1 + k \bmod 97 \end{aligned}$$

si ottiene la soluzione riportata in Figura 7.11.

Esercizio

Sia data la sequenza di chiavi TENTATIVO, dove ciascun carattere è individuato dal suo ordine progressivo nell’alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 19, inizialmente supposta vuota, in cui avvenga l’inserimento della sequenza di cui sopra. Si supponga di utilizzare l’open addressing con double hashing. Sia a carico del candidato la scelta di $h_1(k)$ e $h_2(k)$.

0	1	2	3	4	5	6	7	8	9	10	11
	A ₁		C		A ₂	F			I ₁		
12	13	14	15	16	17	18	19	20	21	22	
	M	N	O			R	I ₂	T			

Tabella 7.11**Soluzione**

Utilizzando come funzioni di hash:

$$\begin{aligned} h_1(k) &= k \bmod 19 \\ h_2(k) &= 1 + k \bmod 97 \end{aligned}$$

si ottiene la soluzione riportata in Figura 7.12.

0	1	2	3	4	5	6	7	8	9
I	T ₁		T ₂		E		A		T ₃
10	11	12	13	14	15	16	17	18	
	V			N	O				

Tabella 7.12**7.5.4 Esercizi proposti****Esercizio**

Sia data la sequenza di chiavi TWILIGHT, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 17, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare il double hashing e che la funzione di hash primaria sia $h(k) = k \bmod 17$. Sia a carico del candidato la scelta dell'altra funzione di hash.

Esercizio

Sia data la sequenza di chiavi ADELESKYFALL, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 29, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con double hashing. Si giustifichi la scelta di $h_1(k)$ e $h_2(k)$.

Esercizio

Sia data la sequenza di chiavi LIKEAPRAYER, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura

di una tabella di hash di dimensione 23, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con double hashing e che la funzione di hash primaria sia $h(k) = k \bmod 23$. Sia a carico del candidato la scelta dell'altra funzione di hash.

Esercizio

Sia data la sequenza di chiavi ALLATONCE, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A = 1, \dots, Z = 26$). Si riporti la struttura di una tabella di hash di dimensione 23, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare l'open addressing con double hashing. Sia a carico del candidato la scelta di $h_1(k)$ e $h_2(k)$.

Capitolo 8

Il paradigma greedy

Sovente negli algoritmi che affrontano problemi di ottimizzazione si possono individuare passi elementari in cui si tratta di scegliere tra diverse alternative mentre si sta ricercando la soluzione ottima. Gli algoritmi greedy permettono, tramite un approccio meno oneroso di quello del divide et impera o della programmazione dinamica, di raggiungere la soluzione, anche se non sempre ottima, scegliendo l'alternativa che localmente risulta essere minima in termini di funzione di costo o massima in termini di funzione di appetibilità.

È errato pensare che ogni problema ammetta una soluzione ottima di tipo greedy, ma esistono problemi per cui questo approccio risulta ottimo. Tra gli esempi vi sono due algoritmi che saranno presentati in questo capitolo, ovvero l'algoritmo di “selezione di attività” e “codici di Huffman”. Alcuni algoritmi sui grafi, quali la ricerca dell’albero ricoprente minimo di un grafo non orientato e pesato e l’algoritmo di *Dijkstra* per la ricerca dei cammini minimi da una sorgente singola quando il grafo orientato e pesato ha solo pesi positivi presentano inoltre le stesse caratteristiche, ovvero sono risolvibili in maniera ottima con l’utilizzo di algoritmi greedy.

8.1 Selezione di attività

8.1.1 Richiami di teoria

Lo scopo dell’algoritmo è di scegliere da un insieme di attività, ognuna eseguibile tra un tempo iniziale e uno finale noti, un sotto-insieme massimale che ne contenga il maggior numero di mutuamente compatibili. La funzione di appetibilità greedy è legata al tempo di fine esecuzione: tanto prima finisce l’attività, tante più altre attività potranno essere eseguite dopo. La funzione di appetibilità greedy è statica.

L’algoritmo procede nel modo seguente. Si ordina l’insieme delle attività in modo crescente rispetto al tempo di fine attività. Si inserisce quindi la prima attività nell’insieme che rappresenta la soluzione finale. Si analizzano, una alla volta, le attività successive: per ogni attività si confronta il tempo iniziale con il tempo finale dell’ultima attività inserita nella soluzione. Se questo è maggiore o uguale, l’attività è compatibile e può essere inserita nella soluzione, altrimenti si procede con quella seguente.

8.1.2 Esercizi svolti

Esercizio

Sia dato il seguente insieme di attività, dove la i -esima attività è identificata dalla coppia tempo di inizio, tempo di terminazione $[s_i, f_i]$ riportato in Tabella 8.1. Si

ATTIVITÀ	s_i	f_i
P_1	1	4
P_2	3	6
P_3	1	8
P_4	3	5
P_5	4	9
P_6	5	11
P_7	2	9

Tabella 8.1 Tempi di inizio e terminazione per le attività $P_1 \div P_7$.

determini, mediante un algoritmo greedy, l'insieme massimale di attività mutuamente compatibili.

Soluzione

Ogni attività P_i è contraddistinta da due attributi: il tempo di inizio s_i è il tempo di terminazione f_i e può essere rappresentata con un intervallo del tipo $P_i = [s_i, f_i]$. Si esegue come passo preliminare l'ordinamento in base al tempo di terminazione, ottenendo il vettore di Tabella 8.2.

ATTIVITÀ	s_i	f_i
P_1	1	4
P_4	3	5
P_2	3	6
P_3	1	8
P_5	4	9
P_7	2	9
P_6	5	11

Tabella 8.2 Passo 1.

Si include da subito la prima attività, che compare nel nuovo elenco, nell'insieme che rappresenta la soluzione finale. Evidenziando in neretto la soluzione temporanea si ottiene la Tabella 8.3.

Si analizza l'attività successiva: P_4 inizia nell'istante $s_4 = 3$, minore di $f_1 = 4$, quindi, non essendo compatibile con P_1 , non può essere inclusa nella soluzione. Si analizzano le attività P_2 e P_3 , e per gli stessi motivi della precedente, le si scarta. P_5 è compatibile, essendo $s_5 \geq f_1$, e viene inserita nella soluzione. La situazione è evidenziata in Tabella 8.4.

ATTIVITÀ	s_i	f_i
P_1	1	4
P_4	3	5
P_2	3	6
P_3	1	8
P_5	4	9
P_7	2	9
P_6	5	11

Tabella 8.3 Passo 2.

ATTIVITÀ	s_i	f_i
P_1	1	4
P_4	3	5
P_2	3	6
P_3	1	8
P_5	4	9
P_7	2	9
P_6	5	11

Tabella 8.4 Passo 3.

P_7 e P_6 sono incompatibili ($s_7 < f_2$ e $s_6 < f_2$), rispettivamente), e vengono scartate. La soluzione finale è quindi costituita dalle sole attività P_1 e P_5 .

8.1.3 Esercizi risolti

Esercizio

Sia dato l'insieme di attività di Tabella 8.5, dove la i-esima attività è identificata dalla coppia tempo di inizio, tempo di terminazione $[s_i, f_i]$. Si determini, mediante un

ATTIVITÀ	s_i	f_i
P_1	1	4
P_2	3	5
P_3	1	7
P_4	3	6
P_5	1	9
P_6	2	11
P_7	5	8
P_8	4	9

Tabella 8.5

algoritmo greedy, l'insieme massimale di attività mutuamente compatibili.

Soluzione

La soluzione è costituita da: $\{P_1, P_7\}$.

Esercizio

Sia dato l'insieme di attività di Tabella 8.6, dove la i-esima attività è identificata dalla coppia tempo di inizio, tempo di terminazione $[s_i, f_i]$. Si determini, mediante un

ATTIVITÀ	s_i	f_i
P_1	1	2
P_2	3	8
P_3	1	5
P_4	3	7
P_5	2	9
P_6	6	11
P_7	3	8
P_8	4	11

Tabella 8.6

algoritmo greedy, l'insieme massimale di attività mutuamente compatibili.

Soluzione

La soluzione è costituita da: $\{P_1, P_4\}$.

8.1.4 Esercizi proposti

Esercizio

Sia dato l'insieme di attività di Tabella 8.7, dove la i-esima attività è identificata dalla coppia tempo di inizio, tempo di terminazione $[s_i, f_i]$. Si determini, mediante un algoritmo greedy, l'insieme massimale di attività mutuamente compatibili.

8.2 Codici di Huffman

8.2.1 Richiami di teoria

I codici di Huffman costituiscono una tecnica particolarmente efficace per comprimere dati senza perdere informazioni. Piuttosto che un codice a lunghezza fissa, dove tutti i simboli sono codificati con lo stesso numero di bit, si utilizza un codice a lunghezza variabile libero da prefisso, in modo da assegnare codici più lunghi a simboli meno frequenti. Si dice libero da prefisso (“prefix-free”) un codice in cui nessuna parola di codice sia prefisso di un'altra parola di codice.

L'algoritmo proposto da Huffman parte da una tabella in cui viene associato a ciascun codice una frequenza, per poi costruire utilizzando un approccio greedy un codice ottimo a lunghezza variabile.

ATTIVITÀ	s_i	f_i
P_1	1	3
P_2	3	5
P_3	2	7
P_4	5	7
P_5	9	12
P_6	13	14
P_7	1	8
P_8	4	9

Tabella 8.7

Questo obiettivo viene raggiunto con un approccio bottom-up. Si costruisce un albero in cui le foglie sono i simboli e la loro distanza è messa in correlazione con l'inverso della frequenza (più il simbolo è frequente, più è vicino alla radice). Durante la costruzione dell'albero i nodi intermedi assumono come frequenza la somma delle frequenze dei figli. Ogni sotto-albero (d'ora in poi detto aggregato) rappresenta una codifica per i simboli che si trovano nelle sue foglie. La funzione di costo è legata alla frequenza con cui compare inizialmente il simbolo e in seguito l'aggregato di simboli. Si tratta quindi di una funzione di costo dinamica.

Per la costruzione dell'albero si sfrutta una coda a priorità, come indicato nel seguito. La priorità corrisponde alla frequenza, ovvero il valore minimo ha priorità massima. Si creano tanti nodi-foglia quanti sono i simboli, mantenendo l'informazione sulla frequenza, e si inseriscono nella coda. Si estraggono i due nodi con frequenza minore e si crea un aggregato, etichettato con una frequenza pari alla somma delle frequenze e con i due nodi estratti come figli. Le etichette 0 e 1 associate agli archi permettono di distinguere i due figli. Il nuovo nodo viene inserito nuovamente nella coda a priorità. Iterando sino a quando non rimane un unico nodo nella coda, si ottiene un albero.

Il codice per la costruzione dell'albero è il successivo:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "Item.h"
5 #include "PQUEUE.h"
6
7 void display (Item *n, char *code, int depth);
8
9 int main (int argc, char *argv[]) {
10    int i, freq, maxN = atoi(argv[1]);
11    char letter, code[maxN+1];
12    Item *root, *r, *l, *tmp;
13    PQ *pq;
14
15    if (argc<2) {
16        printf ("Error: missing argument\n");
17        printf ("correct format:\n");
18        printf ("%s maxN \n", argv[0]);
19        return 0;
20    }
21
22    pq = PQUEUEinit (maxN, ITEMcompare);
23
24    for (i=0; i<maxN; i++) {

```

```

25     printf ("Enter letter: ");
26     scanf ("%s", &letter);
27     printf ("Enter frequency: ");
28     scanf ("%d", &freq);
29     tmp = ITEMnew (letter, freq);
30     PQQUEUEinsert (pq, tmp);
31 }
32 while (PQUEUEsize (pq) > 1) {
33     l = PQUEUEextract (pq);
34     r = PQUEUEextract (pq);
35     tmp = ITEMnew ('!', l->freq+r->freq);
36     tmp->left = l;
37     tmp->right = r;
38     PQQUEUEinsert (pq, tmp);
39 }
40 root = PQUEUEextract (pq);
41 display (root, code, 0);
42 return 1;
43 }
```

La codifica di un simbolo è la sequenza di bit assegnata agli archi del cammino che va dalla radice alla foglia. Il codice per la visualizzazione è riportato di seguito:

```

1 void display (Item *n, char *code, int depth) {
2     if (n->left==NULL && n->right==NULL) {
3         code[depth] = '\0';
4         printf ("Huffman code for %c is %s\n", n->car, code);
5         return;
6     }
7     code[depth] = '0';
8     display (n->left, code, depth+1);
9     code[depth] = '1';
10    display (n->right, code, depth+1);
11
12    return;
13 }
```

L'algoritmo può portare a più soluzioni, in funzione dell'ordinamento e di come si assegnano i bit. È però garantita l'ottimalità della codifica così ottenuta.

8.2.2 Esercizi svolti

Esercizio

Si determini un codice di Huffman ottimo per i seguenti caratteri con le frequenze specificate:

G:13 F:29 H:35 I:8 J:20 K:60 L:27 M:50

Soluzione

Come primo passo si inseriscono le informazioni in una coda a priorità. Per pura comodità di rappresentazione si utilizza un vettore ordinato al posto della coda a priorità, come in Figura 8.1(a).

Si estraggono quindi i due elementi con frequenza minore e si effettua la loro fusione in un albero la cui radice ha come frequenza la somma delle frequenze (21) e si inserisce nella coda la struttura ottenuta, rappresentata in Figura 8.1(b).

Si compie una nuova estrazione, ottenendo il nodo **J** e il nodo radice dell'albero **I-G** creando un nuovo albero con frequenza alla radice pari a 41 (20 + 21). Tale albero si inserisce nuovamente nella coda (Figura 8.1(c)). Si procede quindi iterando. I passi

(a) I: 8 G: 13 J: 20 L: 27 F: 29 H: 35 M: 50 K: 60

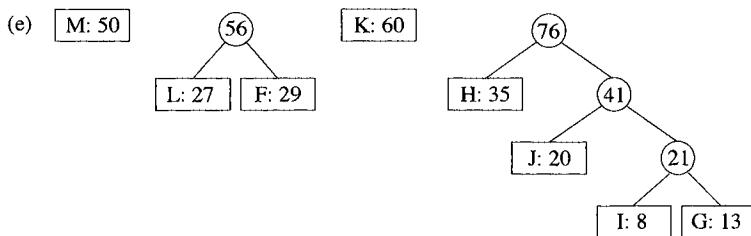
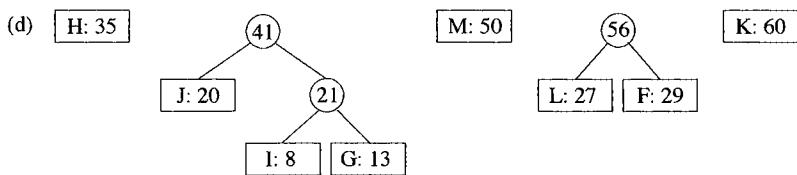
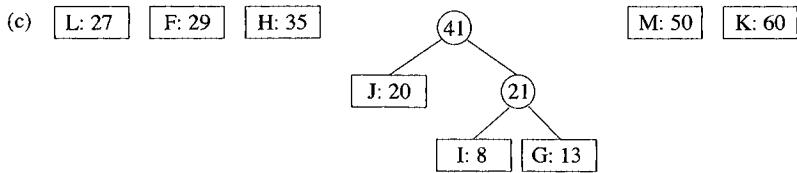
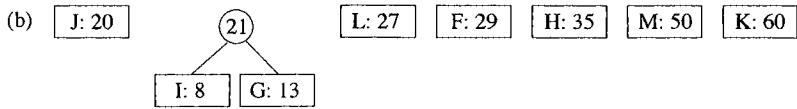
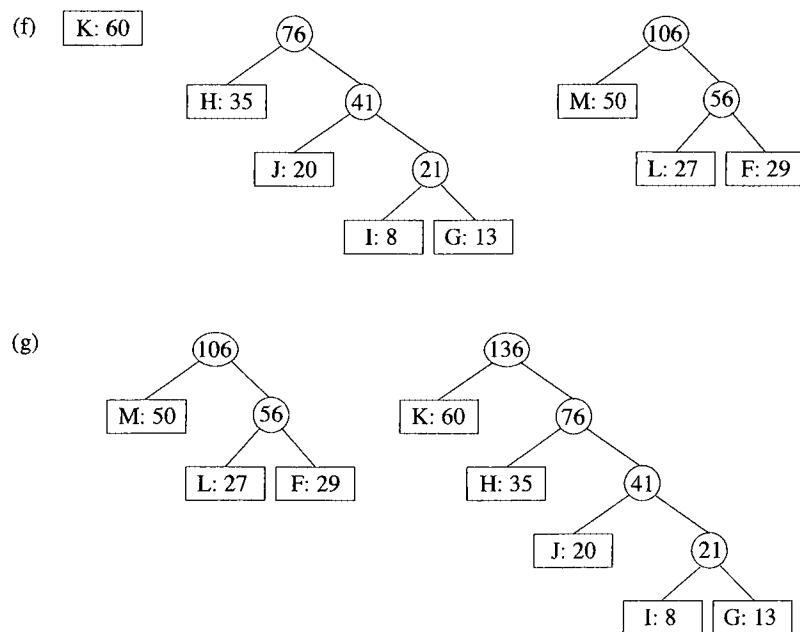


Figura 8.1

**Figura 8.2**

intermedi sono rappresentati in Figura 8.1(d) ed (e), il terzultimo e il penultimo nella Figura 8.2(f) e (g). Il risultato finale è riportato in Figura 8.3.

Al termine si ottiene un albero le cui foglie sono i nodi di partenza. La distanza delle foglie dalla radice, come già accennato, è in correlazione con la frequenza; si nota infatti che i simboli con frequenza minore sono a distanza maggiore.

L'etichettatura degli archi, sottintesa nei passi precedenti, assegna 1 all'arco sinistro e 0 a quello destro.

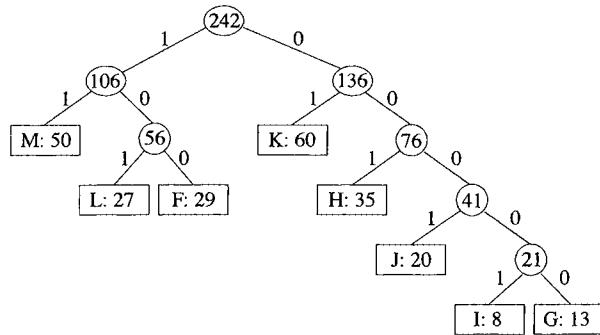


Figura 8.3

Il nuovo set di codici a lunghezza variabile è rappresentato in Tabella 8.8.

SIMBOLO	CODICE	l_i
G	00000	5
F	100	3
H	001	3
I	00001	5
J	0001	4
K	01	2
L	101	3
M	11	2

Tabella 8.8

8.2.3 Esercizi risolti

Esercizio

Si determini un codice di Huffman ottimo per i seguenti caratteri con le frequenze specificate.

A:1 B:1 C:2 D:3 E:5 F:8 G:11 H:21

Soluzione

Il risultato finale è il seguente:

A:101110 B:101111 C:10110 D:1010 E:100 F:110 G:111 H:0

Esercizio

Si determini un codice di Huffman ottimo per i seguenti caratteri con le frequenze specificate.

A:5 B:8 C:2 D:6 E:5 F:4 G:13 H:9

Soluzione

Il risultato è il seguente:

A:110 B:001 C:1001 D:101 E:111 F:1000 G:01 H:000

8.2.4 Esercizi proposti

Esercizio

Si determini un codice di Huffman ottimo per i seguenti caratteri con le frequenze specificate.

A:8 B:5 C:6 D:2 E:4 F:5 G:9 H:13

Esercizio

Si determini un codice di Huffman ottimo per i seguenti caratteri con le frequenze specificate.

A:50 B:17 C:60 D:40 E:48 F:15 G:39 H:33

Esercizio

Si determini un codice di Huffman ottimo per i seguenti caratteri con le frequenze specificate.

A:80 B:50 C:60 D:20 E:40 F:45 G:90 H:10

Capitolo 9

Le visite dei grafi e le loro applicazioni

Un grafo G è un coppia di insiemi (V, E) , dove V è un insieme non vuoto e finito di vertici ed E è un insieme finito di archi, che definiscono una relazione binaria su V .

In questo capitolo si tratteranno:

- ▷ le tecniche di rappresentazione dei grafi mediante
 - ◊ lista di adiacenza
 - ◊ matrice di adiacenza
- ▷ alcuni algoritmi classici della teoria dei grafi, quali:
 - ◊ la visita in ampiezza e in profondità
 - ◊ le applicazioni delle visite alla determinazione:
 - delle componenti connesse di un grafo non orientato
 - delle componenti fortemente connesse di un grafo orientato
 - dei punti di articolazione di un grafo non orientato
 - dell'ordinamento topologico di un DAG.

9.1 Rappresentazione dei grafi

9.1.1 Richiami di teoria

Matrice di adiacenza

Dato un grafo $G = (V, E)$, esso può essere memorizzato in una matrice di adiacenza A di $|V| \cdot |V|$ elementi, dove:

$$A[i,j] = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{se } (i, j) \notin E \end{cases}$$

Se il grafo non è orientato, la matrice A è simmetrica. La matrice di adiacenza ha complessità spaziale

$$S(n) = \Theta(|V|^2)$$

quindi questa rappresentazione è conveniente solo per grafi densi. Due suoi vantaggi sono l'assenza di costi aggiuntivi per rappresentare i pesi di un grafo pesato e l'accesso efficiente alla topologia del grafo, in quanto il costo di determinare se due nodi v e w sono connessi da un arco è $O(1)$.

Lista di adiacenza

Dato un grafo $G = (V, E)$, esso può essere memorizzato in una lista di adiacenza formata da una lista o da un vettore A di $|V|$ elementi tale che $A[i]$ contiene il puntatore alla lista dei vertici adiacenti a i . L'uso del vettore comporta che sia noto il numero di vertici.

Nei grafi non orientati, il numero complessivo di elementi contenuti nelle liste è $2 \cdot |E|$, nei grafi orientati è $|E|$. La complessità spaziale di una lista di adiacenza è la seguente:

$$S(n) = O(\max(|V|, |E|)) = O(|V + E|)$$

Questa rappresentazione è quindi vantaggiosa per grafi sparsi. La lista di adiacenza non garantisce un accesso efficiente alla topologia del grafo, in quanto il costo per determinare se due nodi v e w sono connessi da un arco è $O(n)$ dove n è la lunghezza di ciascuna delle liste di adiacenza, quindi nel caso peggiore è lineare nel numero di vertici $|V|$.

9.1.2 Esercizi svolti

Esercizio

Sia dato il grafo orientato di Figura 9.1. Lo si rappresenti come lista delle adiacenze e matrice delle adiacenze.

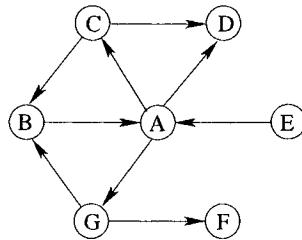


Figura 9.1

Soluzione

La soluzione è riportata in Figura 9.2. Più in dettaglio la Figura 9.2(a) riporta la rappresentazione mediante lista di liste, che fornisce altissima duttilità ma generalmente prestazioni inferiori. La Figura 9.2(b) riporta la rappresentazione mediante vettore di liste, meno dinamica ma che permette un più semplice accesso alla lista principale. Infine la Figura 9.2(c) riporta la matrice di adiacenza.

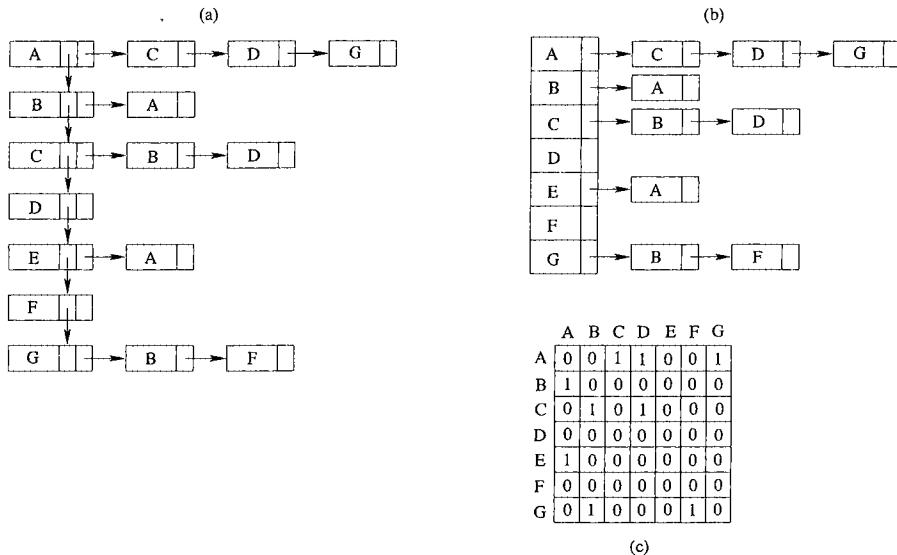


Figura 9.2 Rappresentazione di un grafo orientato mediante matrice e lista di adiacenze.

9.2 Visita in ampiezza (BFS, Breadth-First Search)

9.2.1 Richiami di teoria

Visitare un grafo $G = (V, E)$ significa, a partire da un vertice dato, seguendo gli archi con una certa strategia, elencare i vertici incontrati, eventualmente aggiungendo altre informazioni.

A partire da un vertice s , la visita in ampiezza:

- ▷ determina tutti i vertici raggiungibili da s , quindi non visita necessariamente tutti i vertici
- ▷ calcola la distanza minima da s di tutti i vertici da esso raggiungibili
- ▷ genera un elenco per livelli dei nodi scoperti dalla visita in ampiezza.

Si dice visita in ampiezza in quanto si espande tutta la frontiera tra i vertici già scoperti e quelli non ancora visitati. Si utilizza una coda per memorizzare i vertici man mano che vengono scoperti. L'algoritmo:

- ▷ estrae un vertice dalla coda
- ▷ mette in coda tutti i vertici non ancora scoperti e a esso adiacenti
- ▷ ripete tali operazioni finché la coda è vuota.

Una possibile implementazione dell'algoritmo è riportata di seguito:

```

1 void GRAPHbfs (Graph G) {
2     int v;
```

```

3   time = 0;
4   for (v=0; v<G->V; v++) {
5       pre[v] = -1;
6       st[v] = -1;
7   }
8   bfs (G, EDGE (0, 0));
9 }
10 }
11 void bfs (Graph G, Edge e) {
12     int v, w;
13     Q q = QUEUEinit ();
14     QUEUEput (q, e);
15     while (!QUEUEempty (q)) {
16         if (pre[(e = QUEUEget (q)).w] == -1) {
17             pre[e.w] = time++;
18             st[e.w] = e.v;
19             for (v = 0; v < G->V; v++)
20                 if (G->adj[e.w][v] == 1)
21                     if (pre[v] == -1)
22                         QUEUEput (q, EDGE(e.w, v));
23         }
24     }
25 }
26 }
```

Essa utilizza le seguenti strutture dati:

- ▷ grafo come matrice delle adiacenze
- ▷ coda **Q** dei vertici già scoperti
- ▷ vettore **st** dei padri nell'albero di visita in ampiezza
- ▷ vettore **pre** dei tempi di scoperta dei vertici
- ▷ contatore **time** del tempo.

9.2.2 Esercizi svolti

Esercizio

Dato il grafo orientato di Figura 9.3, se ne effettui una visita in ampiezza, considerando *A* come vertice di partenza. Qualora necessario, si trattino i vertici secondo l'ordine alfabetico.

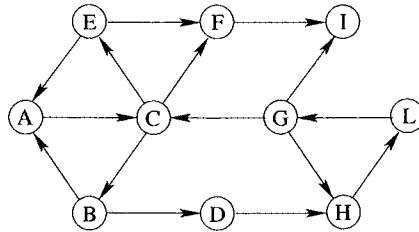


Figura 9.3

Soluzione

Si procede come segue:

- ▷ Passo 1: si scopre il vertice di partenza A , lo si inserisce nella coda Q e lo si elenca in uscita attribuendogli distanza 0 da se stesso (Figura 9.4(a))

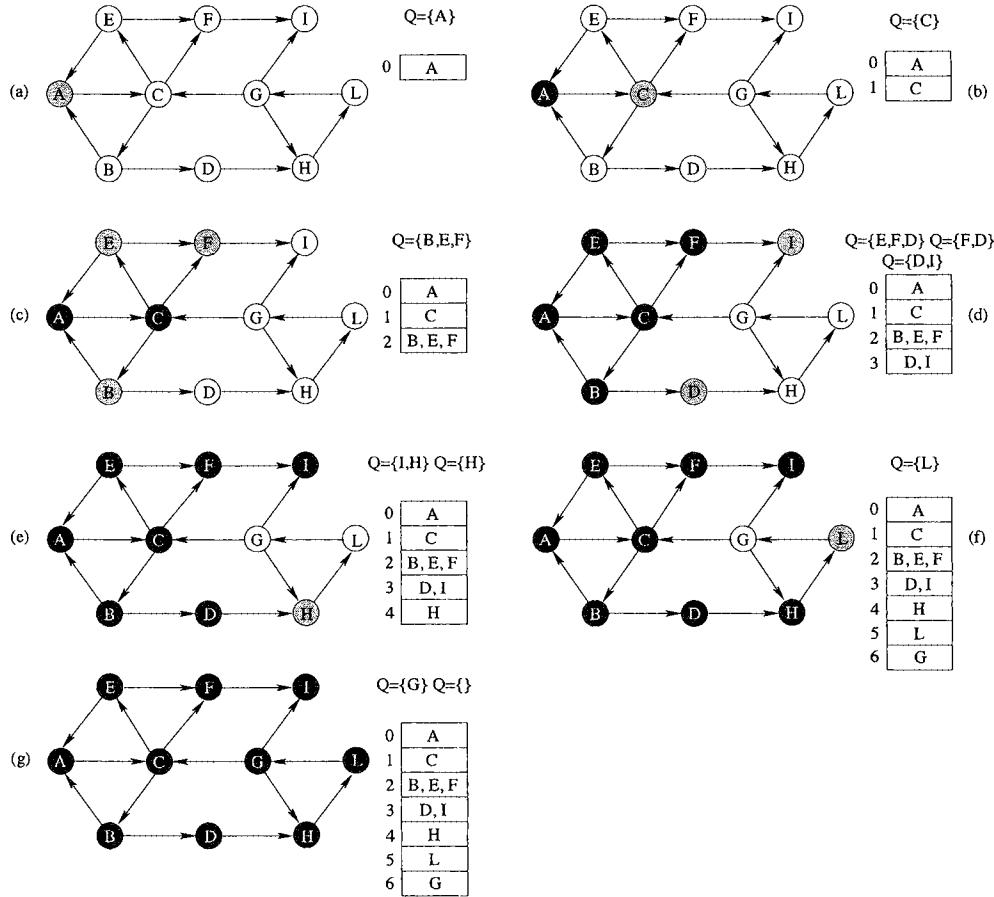


Figura 9.4

- ▷ Passo 2: si estrae A dalla coda Q , si identifica nel vertice C il solo vertice a esso adiacente non ancora scoperto, che viene messo nella coda e in uscita a distanza 1 da A (Figura 9.4(b))
- ▷ Passo 3: si estrae C dalla coda, si identificano i nodi B, E, F come nodi a esso adiacenti non ancora scoperti e li si inserisce nella coda e in uscita a distanza 2 da A (Figura 9.4(c))
- ▷ Passi 4 e 5: si estrae dalla coda B , si identifica nel vertice D il solo vertice a esso adiacente non ancora scoperto. Tale vertice viene messo nella coda Q e in uscita a distanza 3 da A . Si estrae dalla coda E , che non ha alcun vertice adiacente non

ancora scoperto. Infine si estrae dalla coda F , si identifica nel vertice I il solo vertice a esso adiacente non ancora scoperto. Tale vertice viene messo nella coda e in uscita a distanza 3 da A (Figura 9.4(d))

- ▷ Passi 6 e 7: si estrae dalla coda D , si identifica nel vertice H il solo vertice ad esso adiacente non ancora scoperto. Tale vertice viene messo nella coda e in uscita a distanza 4 da A . Si estrae dalla coda I , che non ha alcun vertice adiacente non ancora scoperto (Figura 9.4(e))
- ▷ Passo 8: si estrae dalla coda H , si identifica nel vertice L il solo vertice a esso adiacente non ancora scoperto. L viene messo nella coda e in uscita a distanza 5 da A (Figura 9.4(f))
- ▷ Passi 9 e 10: si estrae dalla coda L , si identifica nel vertice G il solo vertice a esso adiacente non ancora scoperto. G viene messo nella coda e in uscita a distanza 6 da A . Si estrae dalla coda G che non ha alcun vertice adiacente non ancora scoperto. La coda si è svuotata e l'algoritmo termina (Figura 9.4(g)).

9.3 Visita in profondità (DFS, Depth-First Search)

9.3.1 Richiami di teoria

A partire da un vertice s , la visita in profondità:

- ▷ visita tutti i vertici del grafo (sia che essi siano raggiungibili da s o meno)
- ▷ etichetta ogni vertice v con tempo di scoperta e tempo di fine elaborazione
- ▷ etichetta ogni arco in funzione della topologia del grafo:
 - ◊ per grafi orientati gli archi possono essere di tipo: T (Tree), B (Backward), F (Forward), C (Cross)
 - ◊ per grafi non orientati gli archi possono essere di tipo: T (Tree), B (Backward)
- ▷ genera una foresta di alberi della visita in profondità, memorizzata in un vettore.

La strategia di visita consiste nell'esplorare il grafo proseguendo finché possibile lungo lo stesso cammino e, quando non è più possibile, ritornando all'ultimo vertice sul quale si era effettuata una scelta sul percorso da intraprendere. Alcuni autori a questo punto considerano terminata la visita in profondità. In questo caso quindi non tutti i vertici vengono visitati, in quanto si scoprono solo i vertici raggiungibili a partire da quello di partenza. Altri autori proseguono invece con la visita nel caso rimangano dei vertici non scoperti. In tal caso uno di essi viene selezionato come nuovo vertice sorgente e la ricerca viene ripetuta a partire da esso. In questo caso l'intero processo viene ripetuto finché non vengono scoperti tutti i vertici del grafo.

Questa tecnica garantisce che ogni vertice finisce in uno e uno solo degli alberi della visita DFS.

Oltre a creare la foresta, la visita in profondità etichetta ogni vertice con informazioni temporali. Ogni vertice v ha due etichette: la prima (`pre[v]`) registra quando v è stato scoperto, mentre la seconda (`post[v]`) registra quando la visita ha finito di esaminare la lista di adiacenza di v .

Un'altra proprietà della visita in profondità è l'etichettatura degli archi del grafo.

Si possono definire quattro tipi di archi in base alla foresta prodotta da una visita in profondità su un grafo orientato G :

- ▷ etichetta **T**, arco tree o dell’albero (foresta) DFS. Un arco (u, v) è un arco dell’albero se v è stato scoperto esplorando l’arco (u, v)
- ▷ etichetta **B**, arco backward o all’indietro. Un arco (u, v) è un arco backward se connette un vertice u a un antenato proprio v in un albero DFS. I cappi vengono considerati come degli archi all’indietro
- ▷ etichetta **F**, arco forward o in avanti. Un arco (u, v) è un arco forward se connette un vertice u a un discendente proprio v in un albero DFS
- ▷ etichetta **C**, arco cross o trasversale. Un arco (u, v) che connette vertici nello stesso albero DFS, purchè un vertice non sia antenato dell’altro, oppure che connette vertici in alberi DFS distinti.

Il codice riportato in seguito illustra una visita in profondità e utilizza le seguenti strutture dati:

- ▷ grafo orientato G rappresentato mediante lista delle adiacenze
- ▷ vettore **st** dei padri per la costruzione della foresta della visita in profondità
- ▷ vettore **pre** dei tempi di scoperta dei vertici (numerazione in pre-ordine dei vertici)
- ▷ vettore **post** dei tempi di completamento dei vertici (numerazione in post-ordine dei vertici)
- ▷ contatore **time** del tempo.

```

1 void GRAPHdfs (Graph G) {
2     int v;
3
4     time = 0;
5     for (v=0; v<G->V; v++) {
6         pre[v] = -1;
7         post[v] = -1;
8         st[v] = -1;
9     }
10    for (v=0; v<G->V; v++)
11        if (pre[v]== -1)
12            dfsR (G, EDGE(v,v));
13    }
14
15 void dfsR (Graph G, Edge e) {
16     link t; int v, w = e.w, Edge x;
17
18     if (e.v != e.w)
19         printf("edge (%d, %d) is tree \n", e.v, e.w) ;
20
21     st[e.w] = e.v;
22     pre[w] = time++;
23     for (t=G->adj[w]; t!=NULL; t=t->next)
24         if (pre[t->v] == -1) {
25             dfsR(G, EDGE(w, t->v));
26         } else {
27             v = t->v;
28             x = EDGE(w, v);
29             if (post[v] == -1)
30                 printf("edge (%d, %d) is back \n", x.v, x.w);
31             else
32                 if (pre[v]>pre[w])
33                     printf ("edge (%d, %d) is forward \n", x.v, x.w);
34                 else
35                     printf ("edge (%d, %d) is cross \n", x.v, x.w);
36         }
37     post[w] = time++;
38 }
```

9.3.2 Esercizi svolti

Esercizio

Sia dato il grafo orientato di Figura 9.5. Considerando A come vertice di partenza:

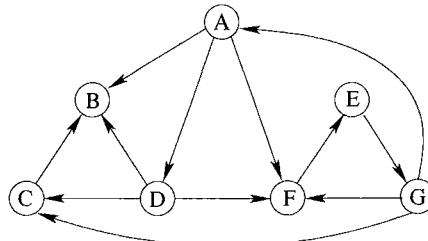


Figura 9.5

- ▷ se ne effettui una visita in profondità. Si elenchino i nodi nell'ordine risultante dalla visita e si indichino per ognuno di essi i tempi di scoperta e di fine elaborazione nel formato $\text{tempo}_1/\text{tempo}_2$.
- ▷ lo si ridisegni, etichettando ogni suo arco come T (tree), B (backward), F (forward), C (cross).

Qualora necessario, si trattino i vertici secondo l'ordine alfabetico e si assuma che la lista delle adiacenze sia anch'essa ordinata alfabeticamente.

Soluzione

Il procedimento considera inizialmente il vertice A , cui si assegna come tempo di scoperta 0 (Figura 9.6(a)). A ha tre nodi adiacenti non ancora scoperti B , D e F . In ordine alfabetico si sceglie B , che viene scoperto al tempo 1 (Figura 9.6(b)). Da B non ci sono archi uscenti, quindi gli si assegna 2 come tempo di fine elaborazione (Figura 9.6(c)).

Ritornando all'ultimo punto di scelta, dal vertice A si prosegue selezionando il vertice D , che viene scoperto al tempo 3 (Figura 9.6(d)). Da D si scopre C al tempo 4. Non essendoci altri nodi non ancora scoperti a esso adiacenti, C viene terminato al tempo 5 (Figura 9.6(e)).

Ritornando a D , si scopre F al tempo 6 (Figura 9.6(f)).

Proseguendo, si scoprono in sequenza E al tempo 7 e poi G al tempo 8 (Figura 9.6(g)).

Non essendoci più altri nodi ancora da scoprire, si ritorna indietro fino ad A , incrementando per ciascun vertice il contatore di tempo per assegnare il corretto tempo di fine elaborazione ai nodi G , E , F , D e A (Figura 9.6(h)).

Per effettuare l'etichettatura degli archi del grafo, uno dei metodi più facili è quello di ridisegnare prima l'albero (o la foresta) ottenuta dalla visita in profondità come rappresentato in Figura 9.7(a). Tutti gli archi sono etichettati tree.

Si aggiungono poi i restanti archi:

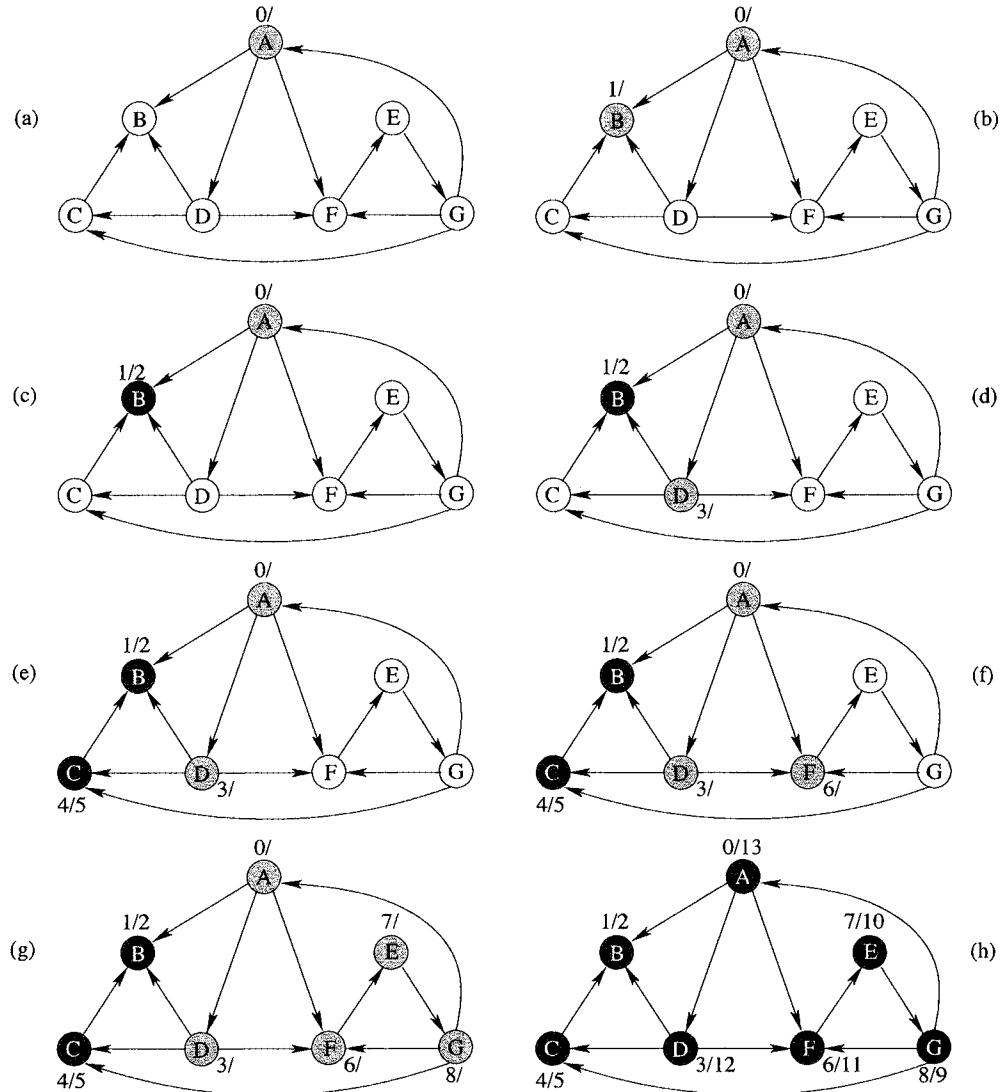


Figura 9.6

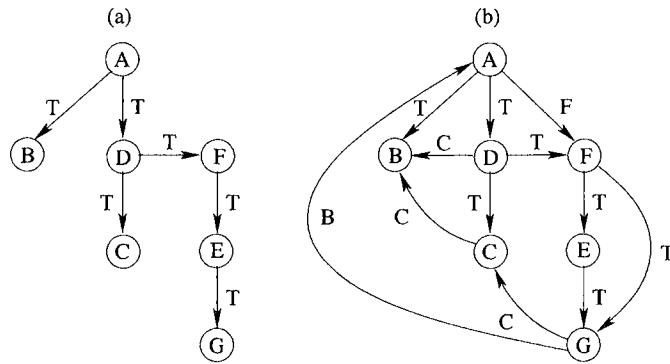


Figura 9.7

- ▷ (D, B) , (C, B) e (G, C) risultano essere archi cross, infatti collegano nodi che non sono in relazione di antenato/descendente sull'albero della visita in profondità
- ▷ (A, F) e (F, G) sono archi forward in quanto connettono un vertice ad un suo discendente
- ▷ (G, A) è invece un arco backward in quanto connette un discendente a un suo antenato (Figura 9.7(b)).

9.4 Applicazioni della visita in profondità: le componenti connesse

9.4.1 Richiami di teoria

In un grafo non orientato si dice *componente connessa* (CC, Connected Component) un sotto-grafo connesso massimale, cioè tale per cui per ogni coppia di vertici del sotto-grafo esiste un cammino che li connette e tale sotto-grafo non è incluso in alcun altro sotto-grafo per cui vale la stessa proprietà di connessione.

In un grafo non orientato, ogni albero della foresta della visita in profondità è una componente连通的.

L'algoritmo che valuta le componenti connesse utilizza quali struttura dati un array $G->cc[v]$ che memorizza, per ogni vertice, un intero che identifica la componente connessa di appartenenza. I vertici fungono da indici dell'array. Il codice per un grafo non orientato rappresentato come lista delle adiacenze è il seguente:

```

1 int GRAPHcc (Graph G) {
2     int v, id = 0;
3     G->cc = malloc (G->V * sizeof(int));
4
5     for (v=0; v<G->V; v++)
6         G->cc[v] = -1;
7     for (v=0; v<G->V; v++)
8         if (G->cc[v] == -1)
9             dfsRCC(G, v, id++);
10
11    return id;
12 }
```

```

13 void dfsRcc (Graph G, int v, int id) {
14     link t;
15
16     G->cc[v] = id;
17     for (t=G->adj[v]; t!=NULL; t=t->next)
18         if (G->cc[t->v] == -1)
19             dfsRccG, t->v, id);
20
21 }
```

9.4.2 Esercizi svolti

Esercizio

Si determinino le componenti connesse del grafo orientato di Figura 9.8. Qualora

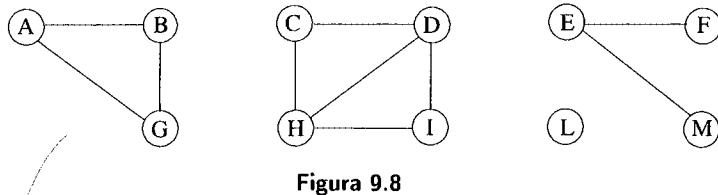


Figura 9.8

necessario, si trattino i vertici secondo l'ordine alfabetico e si assuma che la lista delle adiacenze sia anch'essa ordinata alfabeticamente.

Soluzione

Da *A* si visita *B* e da questo *G*, individuando la prima componente连通的. Si riparte da *C*, visitando in ordine *D*, *H* e *I* trovando la seconda componente连通的. Quindi si riparte da *E*, vistando in ordine *F* e poi *M* e trovando la terza componente连通的. Infine si visita *F* che costituisce la quarta componente连通的. Le componenti connesse rintracciate sono evidenziate in Figura 9.9.

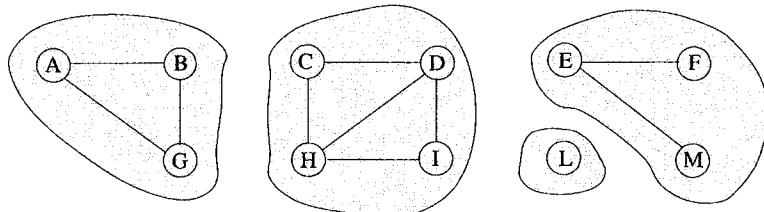


Figura 9.9

9.5 Applicazioni della visita in profondità: le componenti fortemente connesse

9.5.1 Richiami di teoria

In un grafo orientato si dice *componente fortemente connessa* (SCC, Strongly Connected Component) un sotto-grafo fortemente connesso massimale, cioè tale per cui i vertici di ogni coppia sono mutuamente raggiungibili tramite cammini e tale sotto-grafo non è incluso in alcun altro sotto-grafo per cui vale la stessa proprietà di forte connessione.

Uno degli algoritmi noti dalla letteratura per il calcolo delle componenti fortemente connesse è quello di *Kosaraju*. Si riporta di seguito il relativo pseudo-codice, preferito al codice C per ragioni di complessità.

Strongly-Connected-Components (G)

1. calcola il grafo trasposto G^T
2. esegui $DFS(G^T)$ per calcolare i tempi di fine elaborazione per ogni vertice u
3. esegui $DFS(G)$, ma nel ciclo principale della visita, considera i vertici in ordine decrescente di tempo di fine elaborazione determinato dalla di G^T
4. le componenti fortemente connesse sono gli alberi della visita in profondità del passo 3.

9.5.2 Esercizi svolti

Esercizio

Si determinino le componenti fortemente connesse del grafo orientato di Figura 9.10. Qualora necessario, si trattino i vertici secondo l'ordine alfabetico e si assuma che la

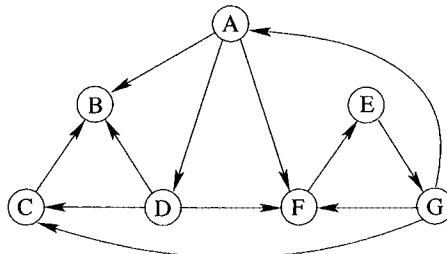


Figura 9.10

lista delle adiacenze sia anch'essa ordinata alfabeticamente.

Soluzione

Il grafo trasposto G^T è rappresentato in Figura 9.11(a). La sua visita in profondità a partire dal vertice A etichetta con i tempi di scoperta e di fine elaborazione i vertici rappresentati in Figura 9.11(b).

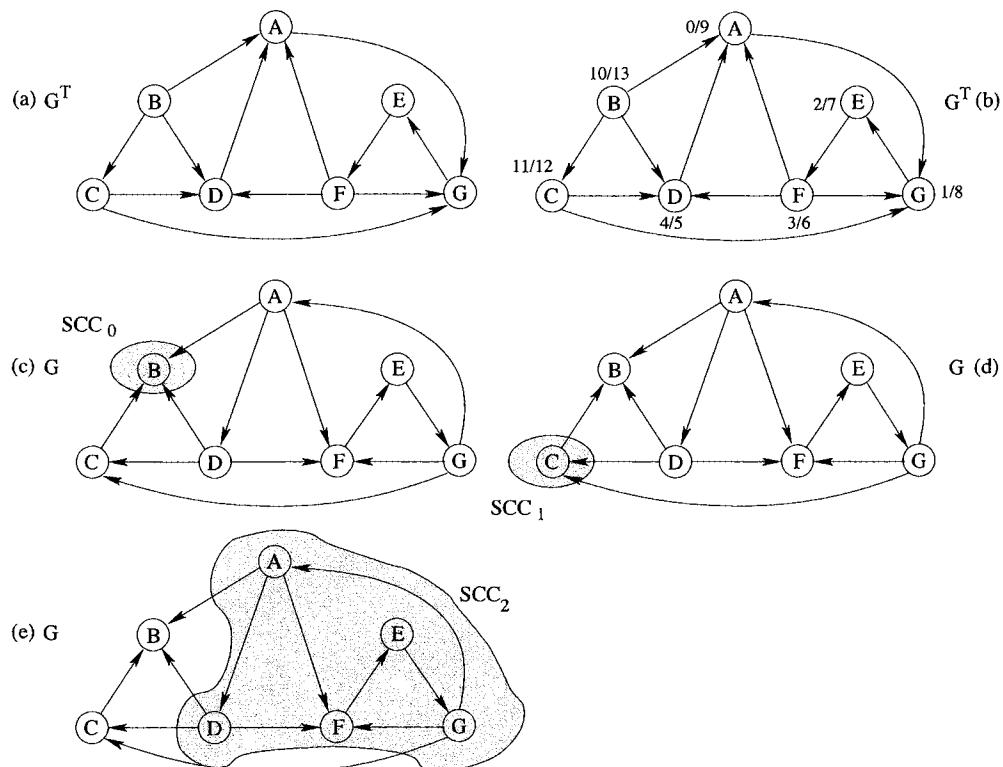


Figura 9.11

Si effettua quindi una visita in profondità del grafo G , partendo dal vertice B , in quanto esso ha tempo di fine elaborazione massimo. La visita termina immediatamente, non essendovi archi uscenti e si identifica la prima SCC (Figura 9.11(c)).

La visita riparte dal vertice C , che ha il secondo massimo tempo di fine elaborazione. Anche in questo caso la visita termina immediatamente, non essendovi nodi adiacenti ancora da scoprire e si identifica la seconda SCC (Figura 9.11(d)).

La visita riparte dal vertice A , che ha il terzo massimo tempo di fine elaborazione. Al termine si identifica la terza SCC, che include i nodi A, D, E, F e G (Figura 9.11(e)).

9.6 Applicazioni della visita in profondità: i punti di articolazione

9.6.1 Richiami di teoria

Sia $G = (V, E)$ un grafo non orientato e connesso. Un punto di articolazione di G è un vertice la cui rimozione disconnette G . Per determinare se un vertice è o meno punto di articolazione, si visita in profondità il grafo, che, essendo il grafo connesso per definizione, ritornerà l'albero della visita in profondità.

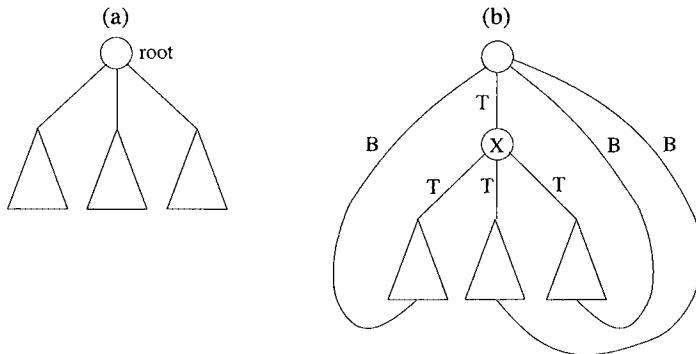


Figura 9.12 Regole di identificazione dei punti di articolazione per un grafo non orientato e connesso.

Su detto albero si fanno le seguenti considerazioni (rappresentate in Figura 9.12):

- ▷ la radice **root** è un punto di articolazione se ha almeno due figli nell'albero della visita in profondità (Figura 9.12(a))
- ▷ per i nodi intermedi (foglie escluse) un vertice è punto di articolazione se esiste almeno un sotto-albero dell'albero di visita in profondità ivi radicato che non presenta nessun arco backward che punta a un antenato proprio del vertice. Nella Figura 9.12(b) il vertice x non è punto di articolazione, in quanto da ciascuno dei tre sotto-alberi della DFS radicati in x si dipartisce un arco backward che punta a un antenato proprio di x .
- ▷ le foglie non sono mai punti di articolazione.

In taluni esercizi si richiede di valutare i punti di articolazione per grafi orientati. In tali casi il procedimento viene applicato dopo avere trasformato il grafo da orientato a non orientato. La trasformazione è banale. Basta infatti eliminare il verso degli archi e, individuate eventuali coppie di archi orientati (a, b) e (b, a) , sopprimere uno dei due archi non orientati che ne risultano, in modo da non trasformare il grafo in multi-grafo.

9.6.2 Esercizi svolti

Esercizio

Si determinino i punti di articolazione del grafo non orientato e connesso di Figura 9.13.

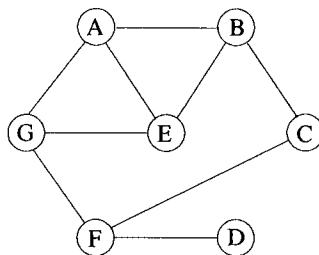


Figura 9.13

Soluzione

Partendo dal vertice A ed effettuando una visita in profondità, si ottiene l’albero di Figura 9.14. Si ricordi che nei grafi non orientati le etichette degli archi sono solo tree e backward. Si esaminano i nodi uno alla volta, secondo i criteri precedentemente citati:

- ▷ la radice A non è punto di articolazione visto che ha un solo figlio nell’albero della DFS
- ▷ B, C, G non sono punti di articolazione visto che esistono degli archi backward che collegano tutti i sotto-alberi in essi radicati con loro antenati propri
- ▷ F è un punto di articolazione. In esso sono radicati due sotto-alberi, ma quello con radice D non ha archi backward che lo collegano a un antenato proprio di F
- ▷ D, E : sono foglie quindi non possono essere punti di articolazione.

9.7 Applicazioni della visita in profondità: l’ordinamento topologico dei DAG

9.7.1 Richiami di teoria

I grafi orientati e aciclici si dicono DAG (Directed Acyclic Graph). Di un DAG è possibile determinare un ordinamento topologico: si tratta di un riordino dei vertici

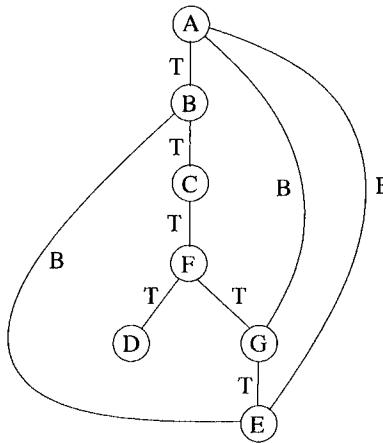


Figura 9.14

secondo una linea orizzontale, per cui se esiste l'arco (u, v) il vertice u compare a sinistra di v e gli archi vanno tutti da sinistra a destra.

È altresì possibile determinare un ordinamento topologico inverso. Si tratta di un riordino dei vertici secondo una linea orizzontale, per cui se esiste l'arco (u, v) il vertice u compare a destra di v e gli archi vanno tutti da destra a sinistra.

Per trovare l'ordinamento topologico inverso di un DAG si usano i tempi di fine elaborazione calcolati in una visita DFS: non appena di un vertice viene terminata l'elaborazione, il vertice viene inserito nel vettore `ts` la cui visualizzazione mostra l'ordinamento topologico inverso. Il codice per un DAG rappresentato come lista delle adiacenze è il seguente:

```

1 void DAGrts (Dag D) {
2     int v;
3
4     time = 0;
5     for (v=0; v < D->V; v++) {
6         pre[v] = -1; ts[v] = -1;
7     }
8     for (v=0; v < D->V; v++)
9         if (pre[v]== -1)
10            TSdfsR(D, v, ts);
11     printf("nodes in reverse topological order \n");
12     for (v=0; v < D->V; v++) printf("%d ", ts[v]);
13     printf("\n");
14 }
15
16 void TSdfsR (Dag D, int v, int ts[]) {
17     link t;
18
19     pre[v] = 0;
20     for (t=D->adj[v]; t!=NULL; t=t->next)
21         if (pre[t->v] == -1)
22             TSdfsR (D, t->v, ts);
23
24     ts[time++] = v;
25
26     return;

```

27 }

Per l’ordinamento topologico di un DAG rappresentato come matrice delle adiacenze si procede come nel caso precedente, ma scambiando i riferimenti degli indici, come mostrato nel codice seguente:

```

1 void TSdfsR (Dag D, int v, int ts[]) {
2     int w;
3
4     pre[v] = 0;
5     for (w=0; w<D->V; w++)
6         if (D->adj[w][v]!=0)
7             if (pre[w]==-1)
8                 TSdfsR (D, w, ts);
9
10    ts[time++] = v;
11 }
```

9.7.2 Esercizi svolti

Esercizio

Sia dato il DAG Figura 9.15. Si trovi l’ordine topologico e l’ordine topologico inverso di tutti i suoi vertici. Qualora necessario, si trattino i vertici secondo l’ordine alfabetico e si assuma che la lista delle adiacenze sia anch’essa ordinata alfabeticamente.

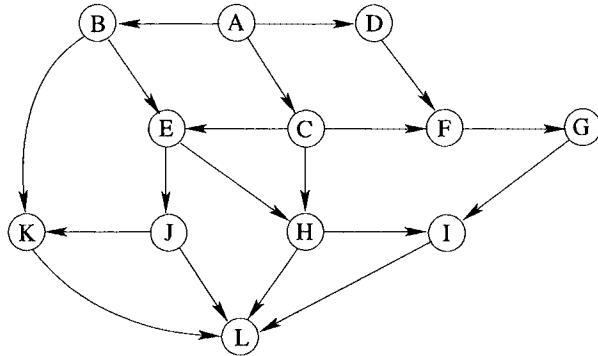


Figura 9.15

Soluzione

Partendo dal vertice *A* ed effettuando una visita in profondità, si ottengono i tempi di scoperta e fine elaborazione indicati in Figura 9.16.

Linearizzando i vertici da destra a sinistra man mano che termina la loro elaborazione e aggiungendo gli archi si ottiene l’ordinamento topologico rappresentato in Figura 9.17(a).

Linearizzando i vertici da sinistra a destra a man mano che termina la loro elaborazione e aggiungendo gli archi si ottiene l’ordinamento topologico inverso di Figura 9.17(b).

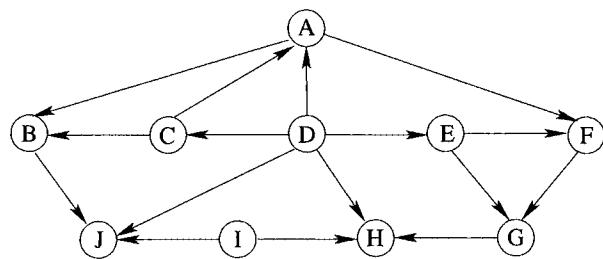


Figura 9.33

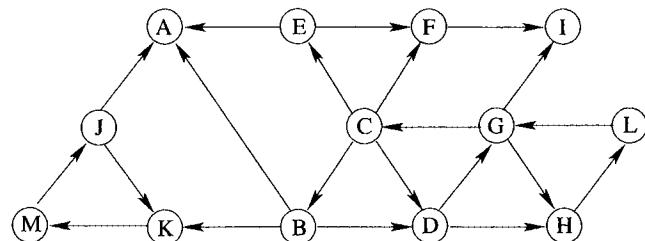


Figura 9.34

- ▷ se ne effettui una visita in profondità. Si elenchino i nodi nell'ordine risultante dalla visita e si indichino per ognuno di essi i tempi di scoperta e di fine elaborazione nel formato $\text{tempo}_1/\text{tempo}_2$
- ▷ lo si ridisegni, etichettando ogni suo arco come T (tree), B (backward), F (forward), C (cross)
- ▷ se ne determinino le componenti fortemente connesse
- ▷ se ne effettui una visita in ampiezza
- ▷ lo si trasformi nel corrispondente grafo non orientato e se ne determinino i punti di articolazione.

Qualora necessario, si trattino i vertici secondo l'ordine alfabetico e si assuma che la lista delle adiacenze sia anch'essa ordinata alfabeticamente.

Esercizio

Sia dato il grafo orientato di Figura 9.35. Considerando *A* come vertice di partenza:

- ▷ se ne effettui una visita in profondità. Si elenchino i nodi nell'ordine risultante dalla visita e si indichino per ognuno di essi i tempi di scoperta e di fine elaborazione nel formato $\text{tempo}_1/\text{tempo}_2$
- ▷ lo si ridisegni, etichettando ogni suo arco come T (tree), B (backward), F (forward), C (cross)
- ▷ se ne determinino le componenti fortemente connesse
- ▷ se ne effettui una visita in ampiezza.

Qualora necessario, si trattino i vertici secondo l'ordine alfabetico e si assuma che la lista delle adiacenze sia anch'essa ordinata alfabeticamente.

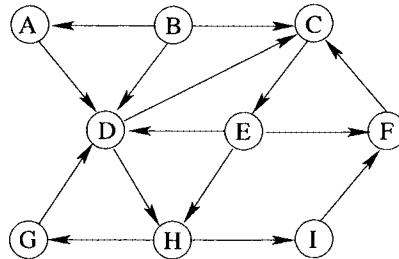


Figura 9.35

Esercizio

Sia dato il grafo orientato di Figura 9.36. Considerando *A* come vertice di partenza:

- ▷ se ne effettui una visita in profondità. Si elenchino i nodi nell'ordine risultante dalla visita e si indichino per ognuno di essi i tempi di scoperta e di fine elaborazione nel formato $\text{tempo}_1/\text{tempo}_2$

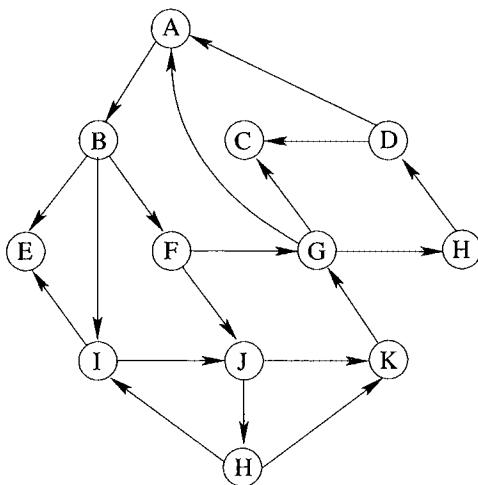


Figura 9.36

- ▷ lo si ridisegni, etichettando ogni suo arco come T (tree), B (backward), F (forward), C (cross)
- ▷ se ne determinino le componenti fortemente connesse
- ▷ se ne effettui una visita in ampiezza
- ▷ lo si trasformi nel corrispondente grafo non orientato e se ne determinino i punti di articolazione.

Qualora necessario, si trattino i vertici secondo l'ordine alfabetico e si assuma che la lista delle adiacenze sia anch'essa ordinata alfabeticamente.

Esercizio

Sia dato il grafo orientato di Figura 9.37. Lo si ordini topologicamente. Sia *A* il vertice di partenza. Qualora necessario, si trattino i vertici secondo l'ordine alfabetico e si assuma che la lista delle adiacenze sia anch'essa ordinata alfabeticamente.

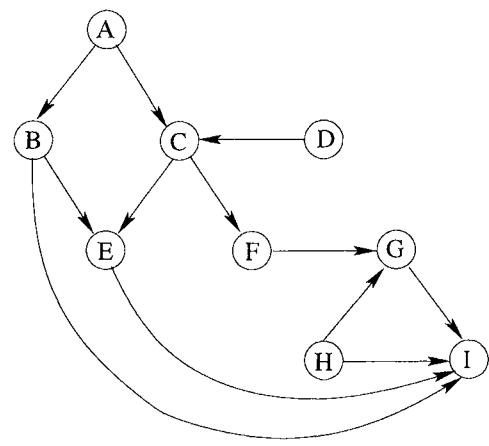


Figura 9.37

Capitolo 10

Gli alberi ricoprenti minimi

In questo capitolo saranno presentati il problema della *on-line connectivity* e quello degli *alberi di connessione minimi*. Si analizzeranno quindi gli algoritmi di *union-find* (normalmente denominati di *quick-union* e *quick-find*) e gli algoritmi di *Kruskal* e *Prim*.

10.1 On-line Connectivity e algoritmi Union-Find

10.1.1 Richiami di teoria

Data in ingresso una sequenza di coppie di interi (p, q) , che rappresenta una connessione tra p e q , il problema della on-line connectivity consiste nel determinare quali coppie (p, q) rappresentano connessioni ignote in precedenza o non implicate transitivamente dalle precedenti, senza esplicitamente costruire il grafo.

Dal punto di vista concettuale, data una coppia formata da due interi, si tratta di capire a quali insiemi appartengano i due interi. Se i due insiemi sono diversi, allora la coppia li unisce, altrimenti la coppia viene scartata.

Dal punto di vista pratico, il problema è quello di rappresentare degli insiemi e di definire su di essi le operazioni di ricerca (**find**) e unione (**union**).

L'algoritmo ripete per tutte le coppie (p, q)

- ▷ leggi la coppia (p, q)
- ▷ esegui **find** su p : trova S_p tale che $p \in S_p$
- ▷ esegui **find** su q : trova $q \in S_q$
- ▷ se S_p e S_q coincidono, passa alla coppia successiva, altrimenti esegui **union** di S_p e S_q .

In termini di complessità si può privilegiare l'operazione **find** (**quick-find**) oppure quella di **union** (**quick-union**).

Quick-find

Gli insiemi S_i delle coppie connesse sono rappresentati mediante un vettore **id** di N interi, dove N è il numero di elementi distinti tra 0 e $N - 1$ che possono comparire nelle coppie:

- ▷ inizialmente `id[i] = i` per indicare che non c'è nessuna connessione
- ▷ se `p` e `q` sono connessi, allora `id[p] = id[q]`.

L'operazione `find` è un semplice riferimento a una cella del vettore `id[indice]` e ha costo unitario. L'operazione `union` consiste nella scansione del vettore per cambiare gli elementi che valgono `p` in `q` con costo lineare nella dimensione del vettore.

Alcuni degli N oggetti rappresentano l'insieme cui essi stessi appartengono, mentre gli altri oggetti puntano al rappresentante del loro insieme. Questa informazione può essere rappresentata sotto forma di albero.

Il codice dell'on-line connectivity con `quick-find` è il seguente:

```

1 #include <stdio.h>
2
3 #define N 10
4
5 int main() {
6     int i, t, p, q, id[N];
7
8     for (i=0; i<N; i++)
9         id[i] = i;
10
11    printf("Input pair p q:  ");
12
13    while (scanf("%d %d", &p, &q) == 2) {
14        if (id[p] == id[q]) {
15            printf("pair %d %d already connected\n", p,q);
16        } else {
17            for (t = id[p], i = 0; i < N; i++)
18                if (id[i] == t)
19                    id[i] = id[q];
20            printf("pair %d %d not yet connected\n", p, q);
21        }
22        printf("Input pair p q:  ");
23    }
24 }
```

Quick-union

Gli insiemi S_i delle coppie connesse sono rappresentati mediante un vettore `id` di N interi dove N è il numero di elementi distinti tra 0 e $N - 1$ che possono comparire nelle coppie:

- ▷ inizialmente `id[i] = i`: tutti gli oggetti puntano a se stessi per indicare che non c'è nessuna connessione
- ▷ ogni oggetto punta a un oggetto cui è connesso oppure a se stesso (non vi sono cicli). Indicando con

$$(id[i])^* = id[id[id[\dots id[i]]]]$$

se gli oggetti `i` e `j` sono connessi allora

$$(id[i])^* = (id[j])^*.$$

L'operazione `find` consiste in un percorrimento di una “catena” di oggetti, con costo al massimo lineare nel numero di oggetti, in generale inferiore. L'operazione `union` è semplice in quanto è sufficiente fare in modo che un oggetto punti all'altro e ha costo unitario.

Il codice dell'on-line connectivity con quick-union è il seguente:

```

1 #include <stdio.h>
2
3 #define N 10
4
5 int main() {
6     int i, j, p, q, id[N];
7
8     for(i=0; i<N; i++)
9         id[i] = i;
10    printf ("Input pair p q: ");
11
12    while (scanf ("%d %d", &p, &q) == 2) {
13        for (i = p; i!= id[i]; i = id[i]);
14        for (j = q; j!= id[j]; j = id[j]);
15        if (i==j) {
16            printf ("pair %d %d already connected\n", p,q);
17        } else {
18            id[i] = j;
19            printf ("pair %d %d not yet connected\n", p, q);
20        }
21        printf ("Input pair p q: ");
22    }
23 }

```

10.1.2 Esercizi svolti

Esercizio

Sia data la seguente sequenza di coppie:

3-2 3-4 5-1 7-3 5-7 9-1

dove la relazione $i-j$ indica che il nodo i è adiacente al nodo j . Si applichi un algoritmo di on-line connectivity con quick-find, riportando a ogni passo il contenuto del vettore e la foresta di alberi al passo finale. I vertici siano denominati con interi compresi tra 0 e 9.

Soluzione

La Tabella 10.1 riporta il vettore **id** a ogni passo. La Figura 10.1 riporta la configu-

	0	1	2	3	4	5	6	7	8	9
3-2	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	2	4	5	6	7	8	9
5-1	0	1	4	4	4	5	6	7	8	9
7-3	0	1	4	4	4	1	6	7	8	9
5-7	0	4	4	4	4	4	6	4	8	9
9-1	0	4	4	4	4	4	6	4	8	4

Tabella 10.1 Algoritmo di quick-find: procedimento passo-passo.

razione finale dell'albero.

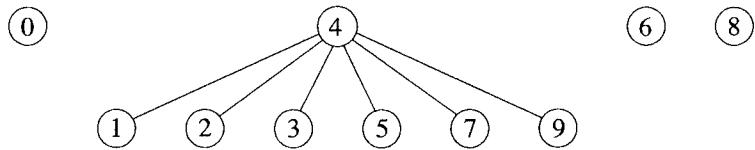


Figura 10.1 Algoritmo di quick-find: risultato finale.

Esercizio

Sia data la seguente sequenza di coppie:

1-3 3-4 5-1 4-0 7-3 5-7 9-4 2-6

dove la relazione $i-j$ indica che il nodo i è adiacente al nodo j . Si applichi un algoritmo di on-line connectivity con quick-union, riportando a ogni passo il contenuto del vettore e la foresta di alberi al passo finale. I vertici siano denominati con interi compresi tra 0 e 9.

Soluzione

La Tabella 10.2 riporta il vettore `id` a ogni passo. La Figura 10.2 riporta la configu-

	0	1	2	3	4	5	6	7	8	9
1-3	0	1	2	3	4	5	6	7	8	9
3-4	0	3	2	3	4	5	6	7	8	9
5-1	0	3	2	4	4	4	6	7	8	9
4-0	0	3	2	4	0	4	6	7	8	9
7-3	0	3	2	4	0	4	6	0	8	9
5-7	0	3	2	4	0	4	6	0	8	9
9-4	0	3	2	4	0	4	6	0	8	0
2-6	0	3	6	4	0	4	6	0	8	0

Tabella 10.2 Algoritmo di quick-union: procedimento passo-passo.

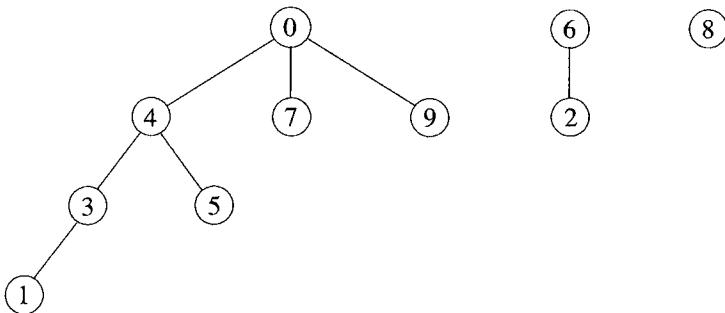
razione finale dell'albero.

10.1.3 Esercizi risolti

Esercizio

Sia data la seguente sequenza di coppie:

1-2 3-5 0-1 7-4 5-1 9-0 6-5

**Figura 10.2** Algoritmo di quick-union: risultato finale.

dove la relazione $i-j$ indica che il nodo i è adiacente al nodo j . Si applichi un algoritmo di on-line connectivity con quick-find, riportando a ogni passo il contenuto del vettore e la foresta di alberi al passo finale. I vertici siano denominati con interi compresi tra 0 e 9.

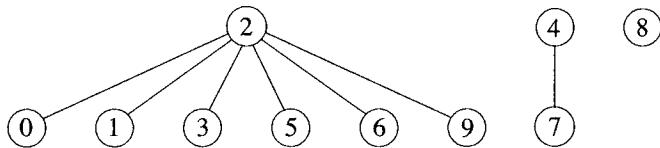
Soluzione

La Tabella 10.3 riporta la configurazione finale del vettore `id`. La Figura 10.3 riporta

0	1	2	3	4	5	6	7	8	9
2	2	2	2	4	2	2	4	8	2

Tabella 10.3

la configurazione finale dell'albero.

**Figura 10.3**

Esercizio

Sia data la seguente sequenza di coppie:

1-2 3-5 0-1 7-4 5-1 9-0 6-5

dove la relazione $i-j$ indica che il nodo i è adiacente al nodo j . Si applichi un algoritmo di on-line connectivity con quick-union, riportando a ogni passo il contenuto del vettore e la foresta di alberi al passo finale. I vertici siano denominati con interi compresi tra 0 e 9.

Soluzione

La Tabella 10.4 riporta la configurazione finale del vettore **id**. La Figura 10.4 riporta

0	1	2	3	4	5	6	7	8	9
0	3	2	4	0	4	6	0	8	0

Tabella 10.4

la configurazione finale dell'albero.

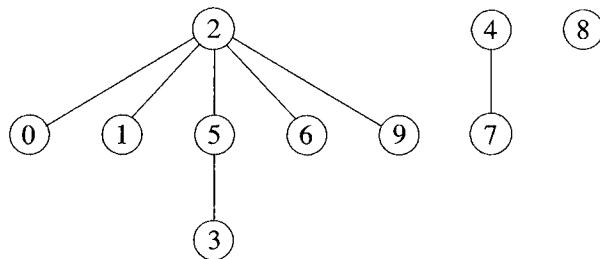


Figura 10.4

10.1.4 Esercizi proposti

Esercizio

Sia data la seguente sequenza di coppie:

$$0-2 \quad 1-4 \quad 5-1 \quad 7-0 \quad 5-6 \quad 9-3 \quad 6-2$$

dove la relazione $i-j$ indica che il nodo i è adiacente al nodo j . Si applichi un algoritmo di on-line connectivity con quick-find, riportando a ogni passo il contenuto del vettore e la foresta di alberi al passo finale. I vertici siano denominati con interi compresi tra 0 e 9.

Esercizio

Sia data la seguente sequenza di coppie:

$$0-2 \quad 1-4 \quad 5-1 \quad 7-0 \quad 5-6 \quad 9-3 \quad 6-2$$

dove la relazione $i-j$ indica che il nodo i è adiacente al nodo j . Si applichi un algoritmo di on-line connectivity con quick-union, riportando a ogni passo il contenuto del vettore e la foresta di alberi al passo finale. I vertici siano denominati con interi compresi tra 0 e 9.

10.2 Alberi ricoprenti minimi

10.2.1 Richiami di teoria

Dato un grafo $G = (V, E)$ non orientato, pesato con una funzione $w : E \rightarrow R$ e connesso, si definisce albero ricoprente minimo un grafo $G' = (V, T)$ dove $T \subseteq E$, aciclico e tale da minimizzare

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

La condizione di aciclicità e la copertura di tutti i vertici implicano che G' sia un albero. L'albero ricoprente minimo è unico se e solo se tutti i pesi degli archi sono distinti.

Un algoritmo generico per trovare un albero ricoprente minimo considera un sottoinsieme A dell'albero ricoprente minimo, inizialmente vuoto e, fintanto che A non è un albero ricoprente minimo, aggiunge ad A un arco "sicuro".

Un arco (u,v) è *sicuro* se e solo se aggiunto a un sottoinsieme di un albero ricoprente minimo produce ancora un sottoinsieme di un albero ricoprente minimo (invarianza della proprietà).

I due algoritmi per trovare un albero ricoprente minimo, detti algoritmo di *Kruskal* e algoritmo di *Prim*, sono particolarizzazioni dell'algoritmo generico, in quanto ognuno di essi usa una specifica regola per determinare un arco sicuro.

Algoritmo di Kruskal

Nell'algoritmo di Kruskal l'insieme di partenza A è una foresta di alberi, inizialmente formati da singoli vertici. L'arco sicuro aggiunto ad A è sempre un arco di peso minimo che connette due alberi distinti della foresta. A questo scopo gli archi vengono ordinati per pesi crescenti e si usano tecniche tipo union-find per rappresentare insiemi e per determinare se due elementi appartengano o meno allo stesso insieme.

Il codice dell'algoritmo di Kruskal è il seguente:

```

1 int GRAPHmstE (Graph G, Edge mst[]) {
2     int i, k;
3     Edge a[maxE];
4
5     E = GRAPHedges (a, G);
6     sort (a, 0, E-1);
7     UFInit (G->V);
8
9     for (i=0, k=0; i<E && k<G->V-1; i++ )
10        if (!UFind (a[i].v, a[i].w)) {
11            UUnion (a[i].v, a[i].w);
12            mst[k++] = a[i];
13        }
14
15    return k;
16 }
```

Algoritmo di Prim

Nell'algoritmo di Prim invece l'insieme A forma un singolo albero, che inizialmente consiste solo del vertice di partenza. L'arco sicuro che è aggiunto ad A è sempre un

arco di peso minimo che connette l'albero a un vertice che non appartiene all'albero, attraversando quindi il "taglio" indotto da A sull'insieme dei vertici V .

Per *taglio* si intende una partizione dei vertici V in due sotto-insiemi, quello che fa già parte della soluzione e quello che non ne fa ancora parte.

Si utilizzano come strutture dati:

- ▷ un vettore **st** per registrare il padre di un vertice che appartiene ad A
- ▷ un vettore **fr** per registrare per ogni vertice di $V - A$ quale è il vertice di A più vicino
- ▷ un vettore **wt** per registrare:
 - ◊ per vertici di A il peso dell'arco al padre
 - ◊ per vertici di $V - A$ il peso dell'arco verso il vertice di A più vicino
- ▷ una variabile **min** per il vertice in $V - A$ più vicino a vertici di A .

Quando si aggiunge ad A un nuovo arco (e un nuovo vertice):

- ▷ si controlla se il nuovo arco ha portato qualche vertice di $V - A$ più vicino a vertici di A
- ▷ si determina il prossimo arco da aggiungere.

Il codice dell'algoritmo di Prim è il seguente:

```

1 void GRAPHmstV (Graph G, int *st, int *wt) {
2   int v, w, min;
3
4   for (v=0; v<G->V; v++) {
5     st[v] = -1;
6     fr[v] = v;
7     wt[v] = maxWT;
8   }
9
10  st[0] = 0;
11  wt[0] = 0;
12  wt[G->V] = maxWT;
13  for (min=0; min!=G->V; ) {
14    v = min;
15    st[min] = fr[min];
16    for (w=0, min=G->V; w<G->V; w++) {
17      if (st[w] == -1) {
18        if (G->adj[v][w]<wt[w]) {
19          wt[w] = G->adj[v][w];
20          fr[w] = v;
21        }
22        if (wt[w]<wt[min]) {
23          min = w;
24        }
25      }
26    }
27  }
28
29  return;
30 }
```

10.2.2 Esercizi svolti

Esercizio

Dato il grafo non orientato, pesato e connesso di Figura 10.5, se ne determini un albero ricoprente minimo mediante l'applicazione dell'algoritmo di Kruskal.

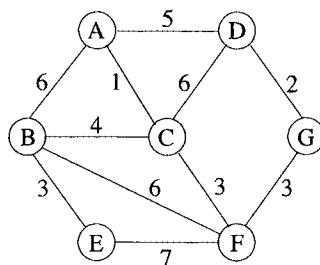


Figura 10.5

Soluzione

Per prima cosa si considerano tutti i nodi come singoli alberi che formano una foresta. Il passo successivo consiste nell'ordinare in maniera crescente tutti gli archi del grafo, poi, considerando gli archi in ordine crescente di peso, si aggiungono alla soluzione tutti quelli che uniscono due alberi distinti.

Il primo arco da considerare è (A, C) . Poiché esso ha peso correntemente minimo (1) e unisce due alberi distinti, viene aggiunto alla soluzione (Figura 10.6(a)).

L'arco (D, G) è a peso correntemente minimo (2) e unisce due alberi distinti, quindi viene aggiunto alla soluzione (Figura 10.6(b)).

Esistono tre archi di peso 3, ognuno dei quali può unire coppie di alberi distinti. Si aggiunge alla soluzione prima (F, G) , poi (C, F) , e infine (B, E) (Figura 10.6(c)–(e)).

Passando agli archi di peso pari a 4, l'arco (B, C) può essere preso, poiché collega due alberi distinti (Figura 10.6(f)).

L'algoritmo termina in quanto tutti i vertici sono stati inclusi nell'albero ricoprente minimo che ha peso 16 ($1 + 2 + 3 + 3 + 3 + 4$). Si noti che tutti gli altri archi non possono essere considerati visto che uniscono vertici appartenenti allo stesso albero.

Esercizio

Dato il grafo non orientato, pesato e connesso di Figura 10.7, se ne determini un albero ricoprente minimo mediante l'applicazione dell'algoritmo di Prim a partire dal vertice A .

Soluzione

Il vertice di partenza A identifica un taglio, attraversato dagli archi (A, B) e (A, C) . Tra i due archi si sceglie quello a peso minore, dunque (A, B) . La Figura 10.8(a) mostra i vertici A e B e l'arco (A, B) che già appartengono alla soluzione.

Si prosegue scegliendo a ogni passo un nuovo arco a peso minimo, che attraversa il taglio congiungendo un altro vertice non ancora appartenente alla soluzione, fino a quando tutti vertici sono stati aggiunti alla soluzione (Figure 10.8(b)–(g)). Le linee tratteggiate (1) ÷ (7) rappresentano i vari tagli, che permettono la scelta dell'arco sicuro.

L'algoritmo termina in quanto tutti i vertici sono stati inclusi nell'albero ricoprente minimo che ha peso 11 ($2 + 2 + 1 + 2 + 1 + 1 + 2$).

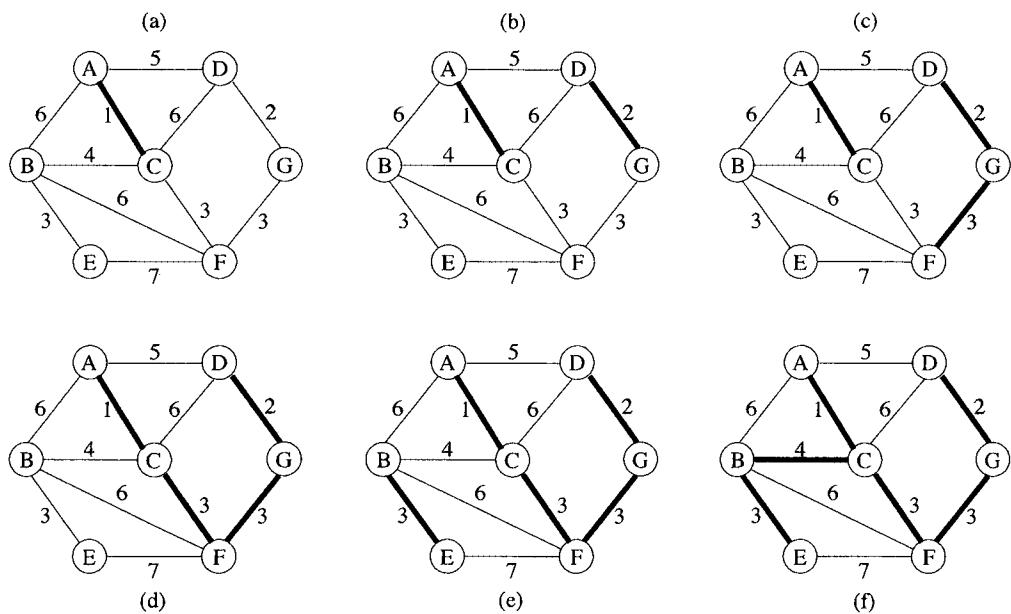


Figura 10.6

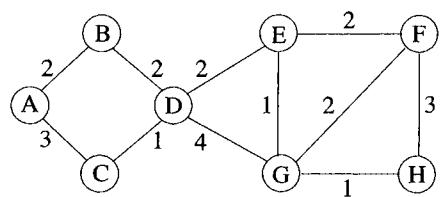


Figura 10.7

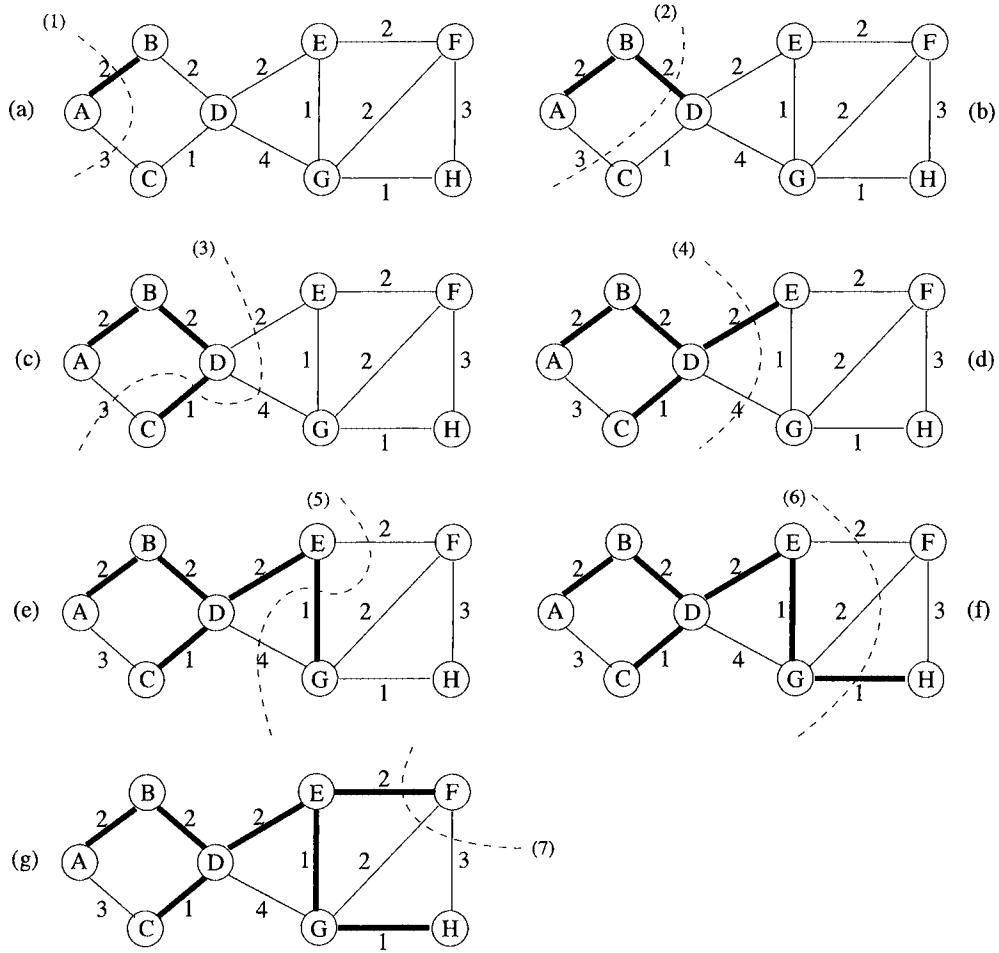


Figura 10.8

10.2.3 Esercizi risolti

Esercizio

Si determini un minimum spanning tree del grafo non orientato, pesato e connesso di Figura 10.9. Si ritorni come risultato l'albero e il valore del peso minimo applicando

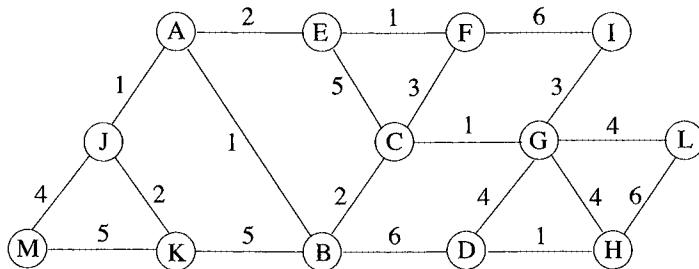


Figura 10.9

l'algoritmo di Kruskal e quello di Prim, a partire dal vertice *A*.

Soluzione

La soluzione è riportata in Figura 10.10. Il peso minimo è 26. L'algoritmo di Kruskal

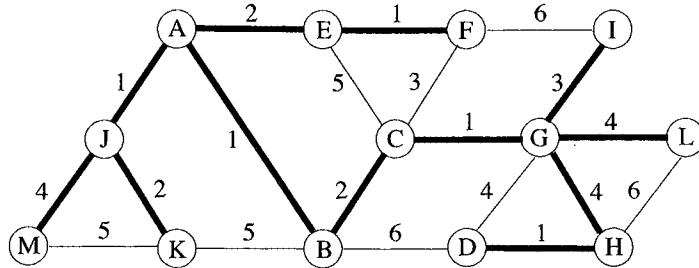


Figura 10.10

seleziona i seguenti archi:

- ▷ peso 1: (A, J) , (A, B) , (E, F) , (C, G) , (D, H)
- ▷ peso 2: (A, E) , (C, B) , (J, K)
- ▷ peso 3: (G, I)
- ▷ peso 4: (J, M) , (G, L) , (G, H)

L'algoritmo di Prim aggiunge a ogni passo gli archi come riportato in Tabella 10.5.

passo	I	II	III	IV	V	VI
arco	(A, J)	(A, B)	(A, E)	(E, F)	(J, K)	(B, C)
passo	VII	VIII	IX	X	XI	XII
arco	(C, G)	(G, I)	(J, M)	(G, D)	(D, H)	(G, L)

Tabella 10.5

Esercizio

Si determini un minimum spanning tree del grafo non orientato, pesato e connesso di Figura 10.11. Si ritorni come risultato l'albero e il valore del peso minimo applicando

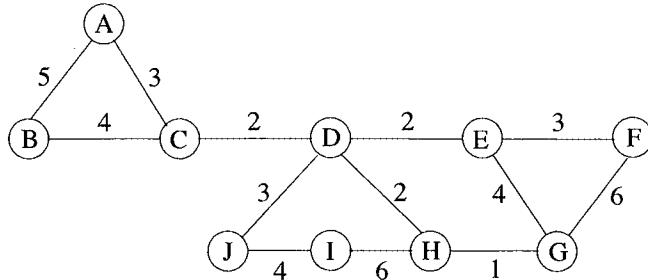


Figura 10.11

l'algoritmo di Kruskal.

Soluzione

La soluzione è riportata in Figura 10.12. Il peso minimo è 24. L'algoritmo di Kruskal seleziona i seguenti archi:

- ▷ peso 1: (H, G)
- ▷ peso 2: $(D, H), (D, E), (D, C)$
- ▷ peso 3: $(A, C), (E, F), (D, J)$
- ▷ peso 4: $(B, C), (J, I)$

Esercizio

Si determini un minimum spanning tree del grafo non orientato, pesato e connesso di Figura 10.13. Si ritorni come risultato l'albero e il valore del peso minimo applicando l'algoritmo di Prim a partire dal vertice A.

Soluzione

La soluzione è riportata in Figura 10.14. Il peso minimo è 15. L'algoritmo di Prim aggiunge a ogni passo gli archi come riportato in Tabella 10.6.

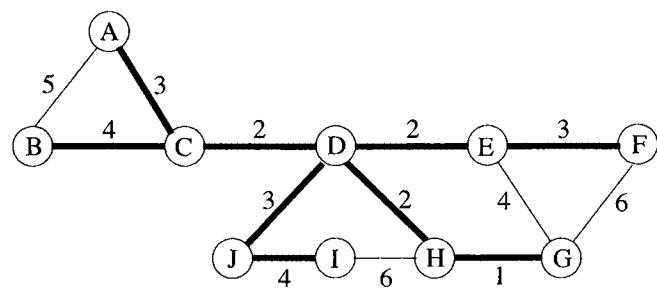


Figura 10.12

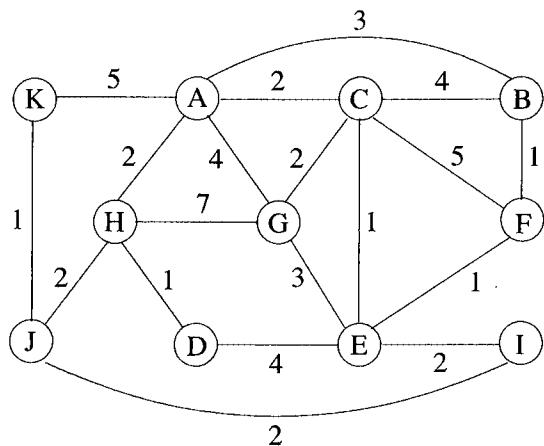


Figura 10.13

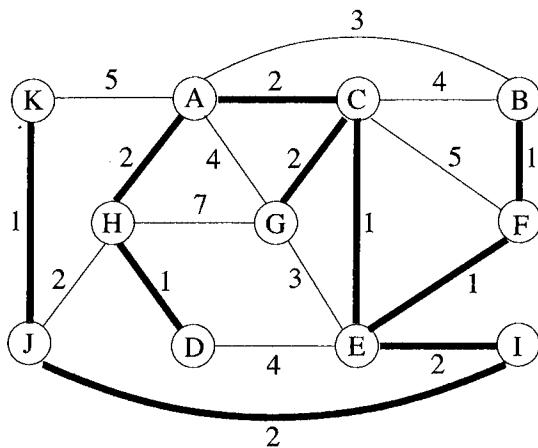


Figura 10.14

passo	I	II	III	IV	V
arco	(A, C)	(C, E)	(E, F)	(B, F)	(E, I)

passo	VI	VII	VIII	IX	X
arco	(A, H)	(H, D)	(C, G)	(J, I)	(J, K)

Tabella 10.6

Esercizio

Si determini un minimum spanning tree del grafo non orientato pesato di Figura 10.15. Si ritorni come risultato l'albero e il valore del peso minimo applicando l'algoritmo di Prim a partire dal vertice I .

Soluzione

La soluzione è riportata in Figura 10.16. Il peso minimo è 30. L'algoritmo di Prim aggiunge a ogni passo gli archi come riportato in Tabella 10.7.

Esercizio

Si determini un minimum spanning tree del grafo non orientato, pesato e connesso di Figura 10.17. Si ritorni come risultato l'albero e il valore del peso minimo applicando l'algoritmo di Prim a partire dal vertice I .

Soluzione

La soluzione è riportata in Figura 10.18. Il peso minimo è 58. L'algoritmo di Prim aggiunge a ogni passo gli archi come riportato in Tabella 10.8.

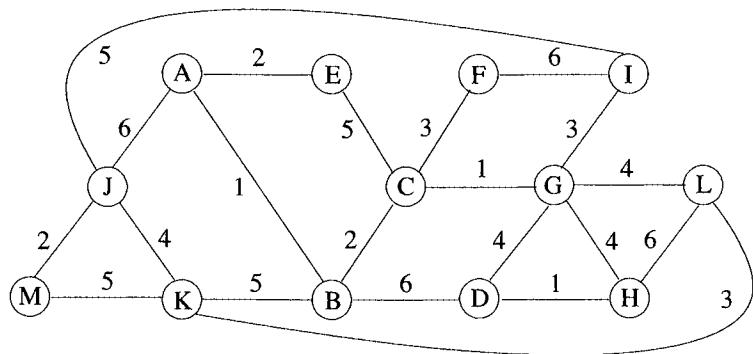


Figura 10.15

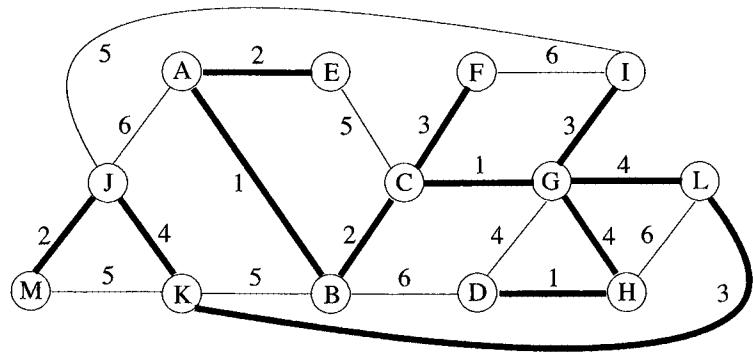


Figura 10.16

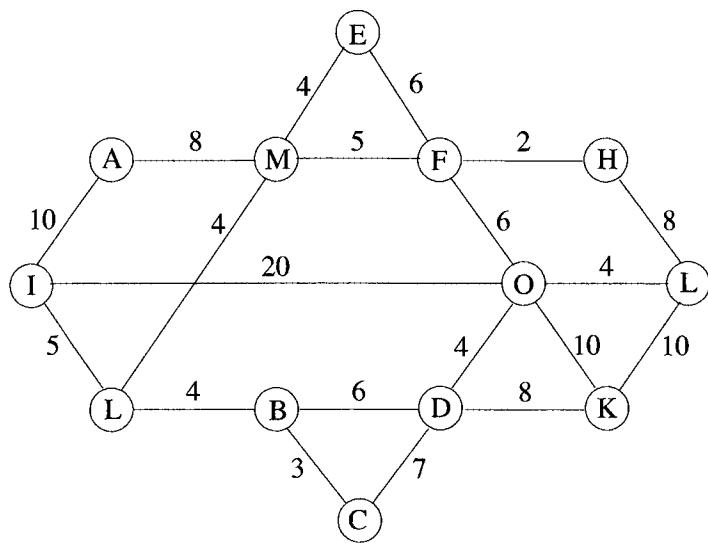


Figura 10.17

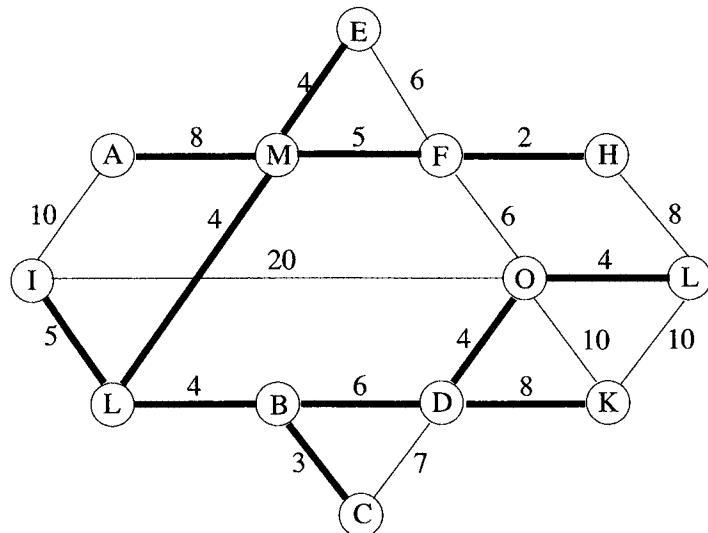


Figura 10.18

passo	I	II	III	IV	V	VI
arco	(I, G)	(C, G)	(C, B)	(B, A)	(A, E)	(C, F)

passo	VII	VIII	IX	X	XI	XII
arco	(G, L)	(L, K)	(J, K)	(J, M)	(G, H)	(H, D)

Tabella 10.7

passo	I	II	III	VI	V	VI
arco	(I, L)	(L, B)	(B, C)	(L, M)	(M, E)	(M, F)

passo	VII	VIII	IX	X	XI	XII
arco	(F, H)	(B, D)	(D, O)	(O, L)	(A, M)	(D, K)

Tabella 10.8

10.2.4 Esercizi proposti

Esercizio

Si determini un minimum spanning tree del grafo non orientato, pesato e connesso di Figura 10.19. Si ritorni come risultato l'albero e il valore del peso minimo applicando

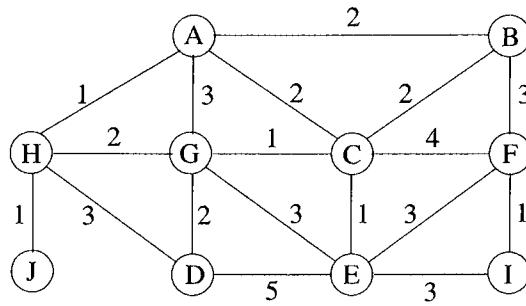


Figura 10.19

l'algoritmo di Kruskal e quello di Prim a partire dal vertice *A*.

Esercizio

Si determini un minimum spanning tree del grafo non orientato pesato di Figura 10.20. Si ritorni come risultato l'albero e il valore del peso minimo applicando l'algoritmo di Kruskal e quello di Prim a partire dal vertice *Q*.

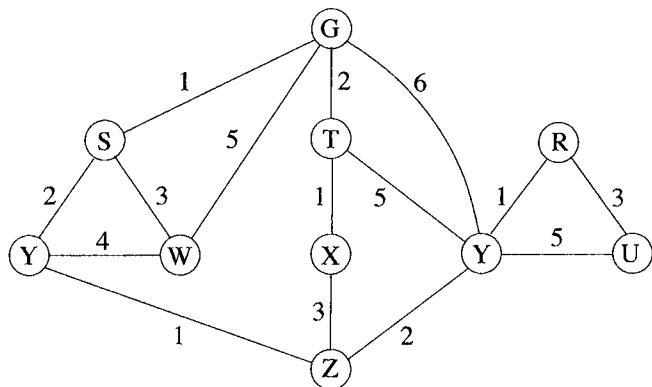


Figura 10.20

Esercizio

Si determini un minimum spanning tree del grafo non orientato, pesato e connesso di Figura 10.21. Si ritorni come risultato l'albero e il valore del peso minimo applicando

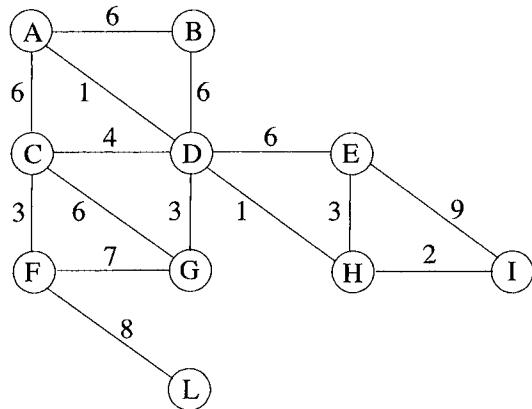


Figura 10.21

l'algoritmo di Kruskal e quello di Prim a partire dal vertice A.

Esercizio

Si determini un minimum spanning tree del grafo non orientato, pesato e connesso di Figura 10.22. Si ritorni come risultato l'albero e il valore del peso minimo applicando l'algoritmo di Kruskal e quello di Prim a partire dal vertice A.

A	B	C	D	E	F	G	H	I	J	K	L	M
1	2	3	4	5	6	7	8	9	10	11	12	13
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
14	15	16	17	18	19	20	21	22	23	24	25	26

	step = 0	step = 1	step = 2
C	3		
A	1		
D	4		
O ₁	15		
N	14		
O ₂	15	2	
L ₁	12		
E ₁	5		
F	6		
O ₃	15	2	18
G	7		
L ₂	12	25	
I	9		
E ₂	5	11	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	A	O ₂	C	D	E ₁	F	G		I		E ₂	L ₁		N
15	16	17	18	19	20	21	22	23	24	25	26	27	28	
O ₁			O ₃								L ₂			

Tabella 12.6

Soluzione

L'ordinamento topologico è il seguente:

A C E B F J D G I L H K

Esercizio 7 (2.0 punti)

Sia dato il grafo orientato pesato di Figura 12.17. Si determinino i valori di tutti i cammini minimi che collegano il vertice A con ogni altro vertice mediante l'algoritmo di Dijkstra. Si assuma, qualora necessario, un ordine alfabetico per i vertici e gli archi. Si riportino i passaggi rilevanti.

Soluzione

La soluzione è riportata in Figura 12.18. L'ordine con cui i nodi vengono estratti dalla coda a priorità è:

A C E F B D J I G H K

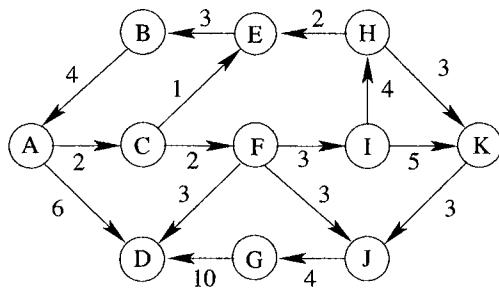


Figura 12.17

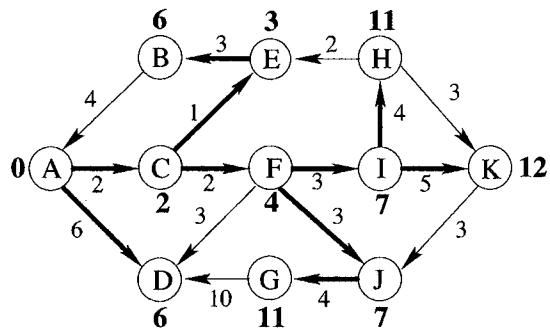


Figura 12.18

12.4 Tema d'esame 04

Esercizio 1 (1.0 punti)

Sia data la sequenza di interi, supposta memorizzata in un vettore:

15 11 2 19 9 6 8 13 16 17

La si ordini in ordine decrescente utilizzando l'algoritmo di exchange sort. Si riportino graficamente i passaggi rilevanti dell'algoritmo e il risultato finale.

Soluzione

La Figura 12.19 riporta i passaggi e la configurazione finale.

0	1	2	3	4	5	6	7	8	9
15	11	2	19	9	6	8	13	16	17
0	1	2	3	4	5	6	7	8	9
15	11	19	9	6	8	13	16	17	2
0	1	2	3	4	5	6	7	8	9
15	19	11	9	8	13	16	17	6	2
0	1	2	3	4	5	6	7	8	9
19	15	11	9	13	16	17	8	6	2
0	1	2	3	4	5	6	7	8	9
19	15	11	13	16	17	9	8	6	2
0	1	2	3	4	5	6	7	8	9
19	15	13	16	17	11	9	8	6	2
0	1	2	3	4	5	6	7	8	9
19	15	16	17	13	11	9	8	6	2
0	1	2	3	4	5	6	7	8	9
19	16	17	15	13	11	9	8	6	2
0	1	2	3	4	5	6	7	8	9
19	17	16	15	13	11	9	8	6	2
0	1	2	3	4	5	6	7	8	9
19	17	16	15	13	11	9	8	6	2

Figura 12.19

Esercizio 2 (2.0 punti)

Si inseriscano, in sequenza, le seguenti chiavi intere in una coda a priorità:

1 11 7 4 5 10 6 8 24 9 89 0 51 13 18

Si ipotizzi di usare uno heap come struttura dati. Si disegni la struttura ai diversi passi dell'inserzione. Su questa struttura si effettui un'estrazione della chiave a priorità massima e si disegni la struttura dopo l'operazione. Su di essa si cambi la priorità di 10 in 100 e si disegni la struttura risultante. Si ipotizzi che la priorità massima sia associata alla chiave a valore massimo.

Soluzione

Dopo le inserzioni si ha:

89 24 51 8 11 10 18 1 5 4 9 0 7 6 13

Dopo l'estrazione del massimo si ottiene:

51 24 18 8 11 10 13 1 5 4 9 0 7 6

Infine, modificando 10 in 100 si ha:

100 24 51 8 11 18 13 1 5 4 9 0 7 6

Esercizio 3 (2.0 punti)

Sia data la sequenza di interi, supposta memorizzata in un vettore:

7 11 19 2 5 15 6 13 16 18 1 9

Si eseguano i primi due passi dell'algoritmo di quick sort per ottenere un ordinamento ascendente. I passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le 2 partizioni del vettore originale e le due partizioni delle partizioni trovate al punto precedente. Si specifichino i pivot scelti e si indichino i passaggi rilevanti.

Soluzione

Il pivot al primo passo è $x = 9$. Al secondo passo il pivot è $x = 5$ per la partizione di sinistra e $x = 15$ per la partizione di destra. Il risultato è il seguente:

2 1 | 5 | 7 6 || 9 || 11 13 | 15 | 18 19 16

Esercizio 4 (2.0 punti)

Si determini mediante un algoritmo greedy un codice di Huffman ottimo per i seguenti caratteri con le frequenze specificate:

A:10 B:20 C:15 D:5 E:25 F:11 G:24 H:9 I:17 L:7

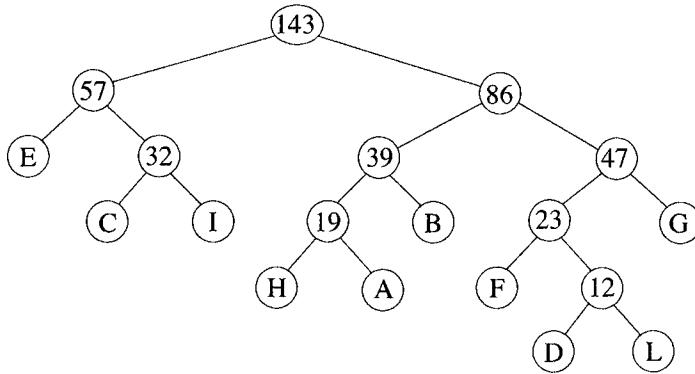


Figura 12.20

Soluzione

La Figura 12.20 riporta la soluzione.

Esercizio 5 (5.0 punti)

Sia dato il grafo orientato rappresentato in Figura 12.21. Considerando *A* come vertice di partenza:

- ▷ se ne effettui una visita in profondità, etichettando i vertici con i tempi di scoperta e di fine elaborazione nel formato tempo₁/tempo₂ (2.0 punti). Qualora necessario, si trattino i vertici secondo l'ordine alfabetico
- ▷ lo si ridisegni, etichettando gli archi come T (tree), B (back), F (forward), C (cross) (1.5 punti)
- ▷ se ne effettui una visita in ampiezza etichettando i vertici con la loro distanza dalla sorgente (1.5 punti).

Soluzione

La Figura 12.22 riporta il risultato della visita in profondità. La visita in ampiezza fornisce:

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 \\ A & | & B \ D & | & C \ E \ I & | \ L & M \end{array}$$

12.5 Tema d'esame 05**Esercizio 1 (1.0 punti)**

Si ordini in maniera ascendente mediante merge sort il seguente vettore di interi:

$$3 \ 9 \ 2 \ 11 \ 15 \ 4 \ 7 \ 3 \ 4 \ 10 \ 2 \ 10 \ 8 \ 6 \ 8$$

Si indichino i passaggi rilevanti.

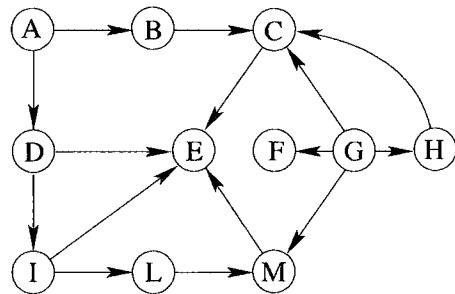


Figura 12.21

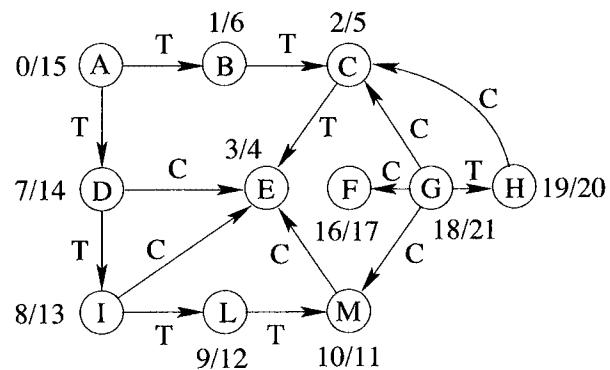


Figura 12.22

Soluzione

La Figura 12.23 riporta la soluzione. Si osservi che la figura è stata ruotata di 90 gradi per ottenerne una maggiore leggibilità.

Esercizio 2 (1.0 punti)

Si ordini in maniera ascendente il seguente vettore di interi mediante shell sort con la sequenza di Knuth. Si indichino i passaggi rilevanti.

5 4 6 7 6 4 0 1 6 5 0 2 3 1 0 10 0 2

Soluzione

Sotto-sequenze 13-ordinate:

$$\left| \begin{array}{ccccccccc} 1 & 0 & 6 & 0 & 2 & 4 & 0 & 1 & 6 \\ I & II & III & IV & V & VI & VII & VIII & IX \\ . & . & . & . & . & . & . & . & . \end{array} \right| \left| \begin{array}{ccccccccc} 6 & 5 & 0 & 2 & 3 & 1 & 0 & 10 & 7 \\ X & XI & XII & XIII & XIV & XV & XVI & XVII & XVIII \\ . & . & . & . & . & . & . & . & . \end{array} \right| \left| \begin{array}{cccc} 5 & 4 & 10 & 7 \\ I & II & III & IV \\ . & . & . & . \end{array} \right| \left| \begin{array}{cc} 6 & 6 \\ V & . \end{array} \right|$$

Sotto-sequenze 4-ordinate:

$$\left| \begin{array}{cccccc} 1 & 0 & 0 & 0 & 2 & 4 \\ I & II & III & IV & I & II \\ . & . & . & . & . & . \end{array} \right| \left| \begin{array}{cccccc} 0 & 1 & 1 & 2 & 2 & 3 \\ III & IV & I & II & III & IV \\ . & . & . & . & . & . \end{array} \right| \left| \begin{array}{cccccc} 3 & 5 & 4 & 2 \\ I & II & III & IV \\ . & . & . & . \end{array} \right| \left| \begin{array}{cccccc} 6 & 5 & 6 & 10 \\ I & II & III & IV \\ . & . & . & . \end{array} \right| \left| \begin{array}{cc} 7 & 6 \\ V & . \end{array} \right|$$

Sotto-sequenze 1-ordinate:

0 0 0 0 1 1 2 2 3 4 4 5 5 6 6 6 7 10

Esercizio 3 (1.5 punti)

Sia dato l'albero binario di Figura 12.24. Lo si visiti in pre-order (0.5 punti), in-order (0.5 punti) e post-order (0.5 punti) elencando l'ordine di visita dei nodi.

Soluzione

pre-order: $a \ b \ d \ g \ h \ l \ m \ e \ i \ c \ f \ j \ k$
 in-order: $g \ d \ l \ h \ m \ b \ i \ e \ a \ c \ j \ f \ k$
 post-order: $g \ l \ m \ h \ d \ i \ e \ b \ j \ k \ f \ c \ a$

Esercizio 4 (2.0 punti)

Si inseriscano nella *radice* del BST di Figura 12.25 in sequenza le chiavi:

11 4 9

riportando il risultato a ogni passo.

Soluzione

La Figura 12.26 riporta la soluzione in (a) dopo l'inserzione di 11, in (b) dopo l'inserzione di 4 e in (c) dopo l'inserzione di 9.

Esercizio 5 (2.5 punti)

Si determinino mediante l'algoritmo di Kosaraju le componenti fortemente connesse del grafo orientato di Figura 12.27.

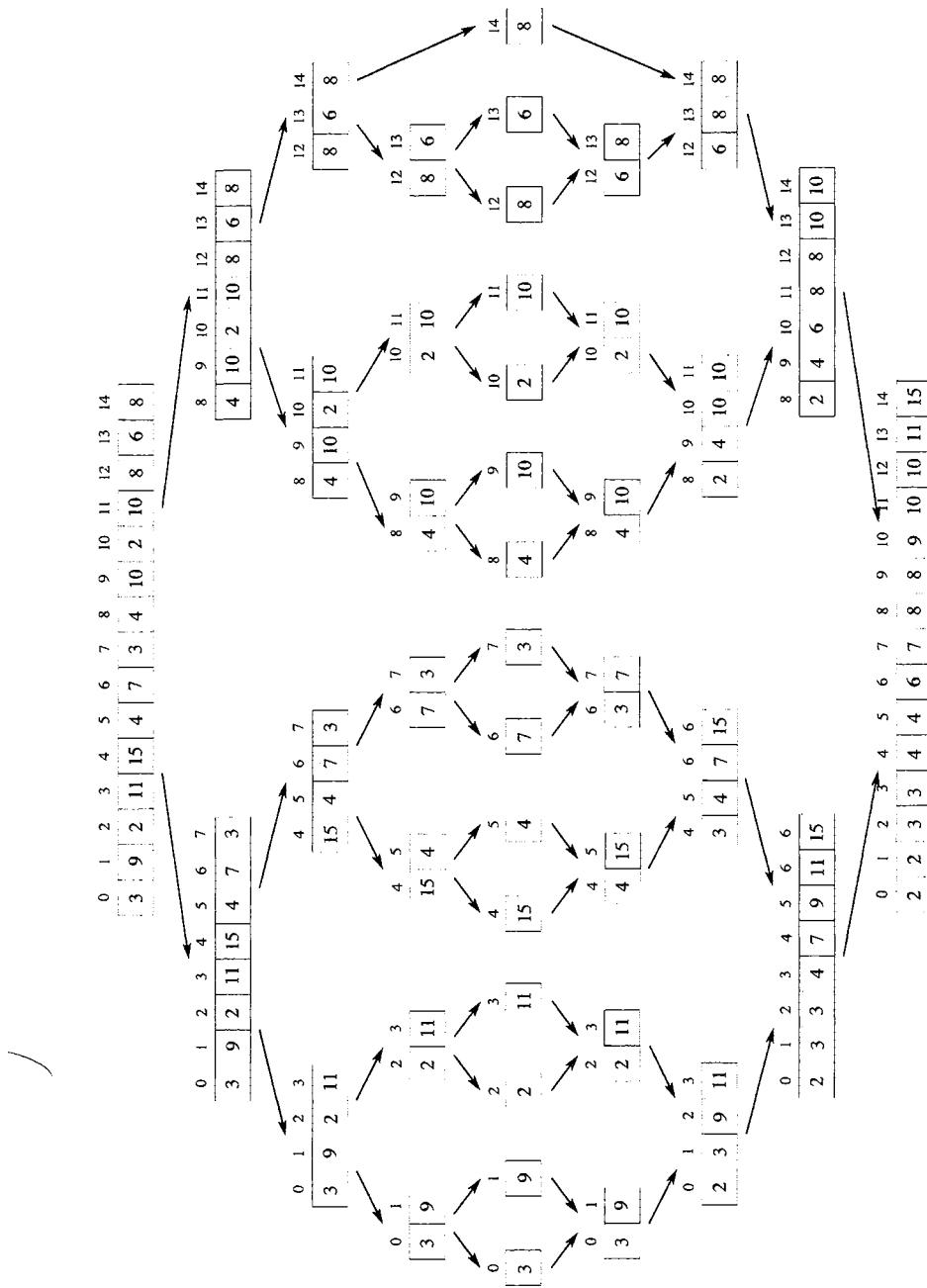
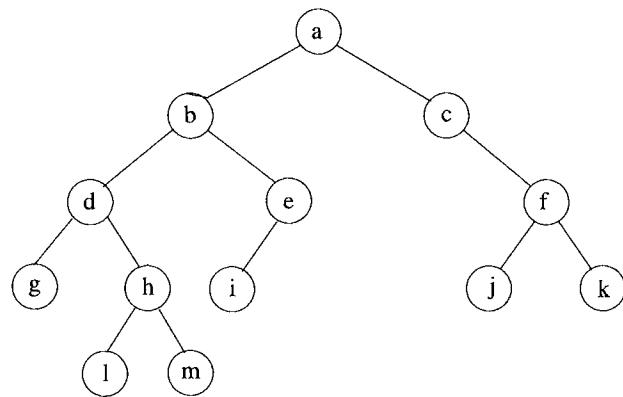
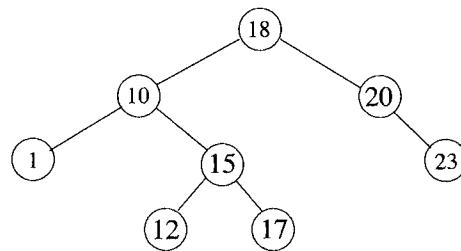
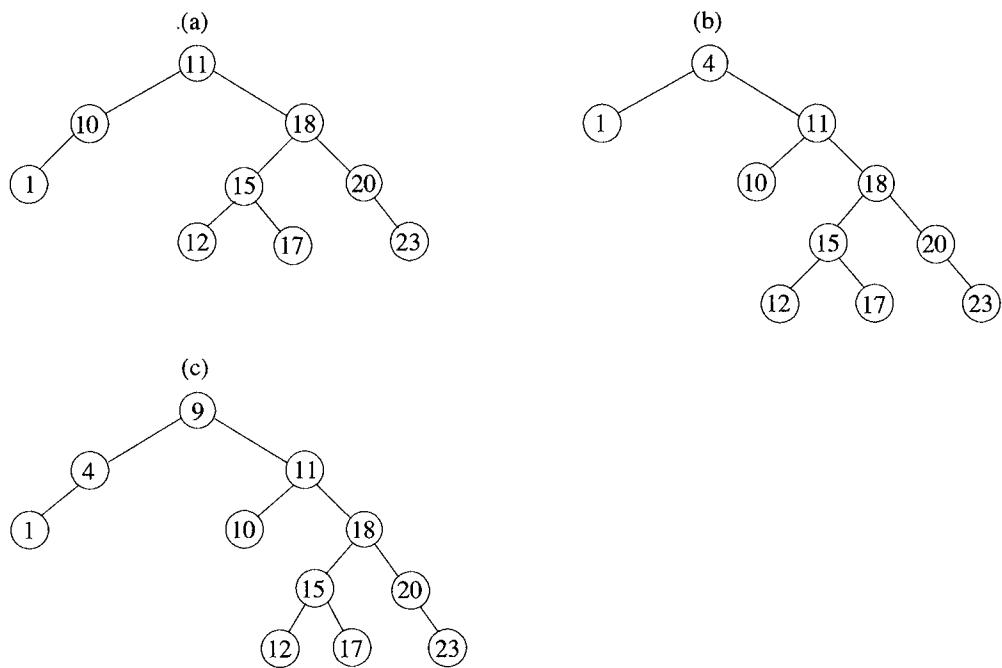
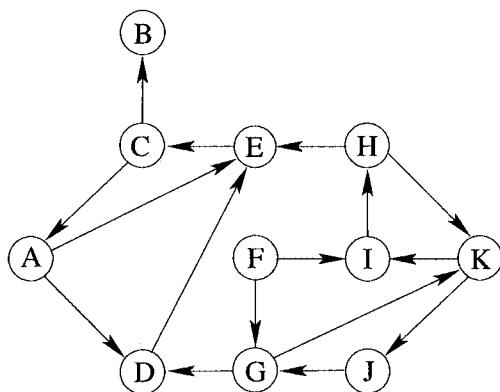


Figura 12.23

**Figura 12.24****Figura 12.25**

**Figura 12.26****Figura 12.27**

Soluzione

La Figura 12.28(a) riporta il risultato della visita in profondità del grafo trasposto a partire dal vertice A. In Figura 12.28(b) sono invece riportate le componenti fortemente connesse individuate sul grafo di partenza.

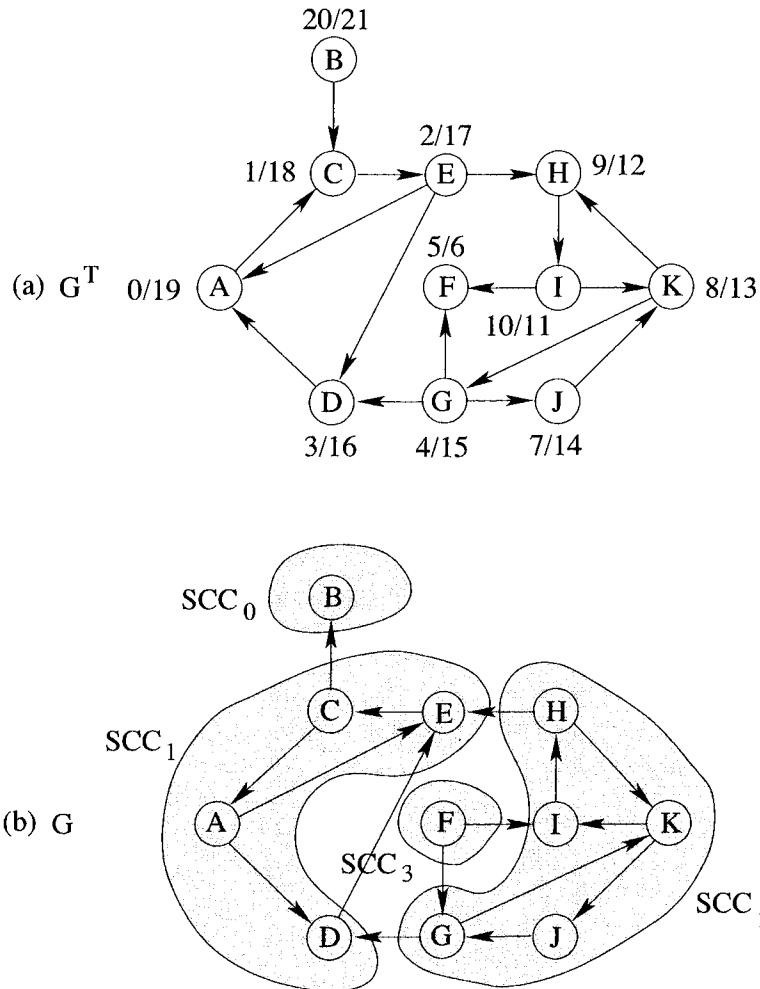


Figura 12.28

Esercizio 6 (4.0 punti)

Sia dato il grafo non orientato pesato di Figura 12.29. Si determini un albero ricoprente minimo applicando l'algoritmo di Kruskal (2.0 punti) e l'algoritmo di Prim (2.0 punti) dal vertice A. Si riportino i passaggi rilevanti, si disegni l'albero ricoprente minimo e si fornisca il valore del peso minimo.

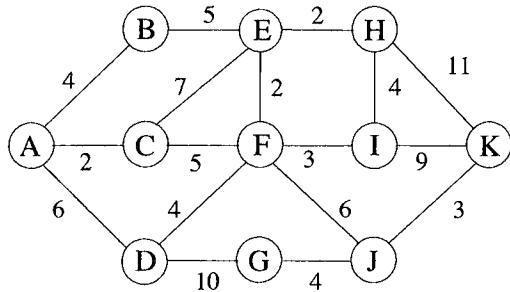


Figura 12.29

Soluzione

La Figura 12.30 riporta la soluzione per l'algoritmo di Kruskal e la Figura 12.31 quella per l'algoritmo di Prim. In entrambi i casi il peso totale del cammino è uguale a 35 e

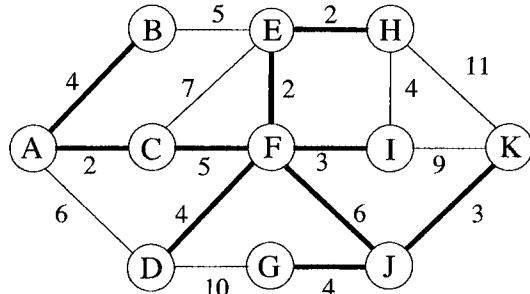


Figura 12.30

l'ordine di scelta degli archi per Kruskal è il seguente:

$$(A, C):2 \quad (E, F):2 \quad (E, H):2 \quad (F, I):3 \quad (J, K):3$$

$$(A, B):4 \quad (F, D):4 \quad (G, J):4 \quad (C, F):5 \quad (F, J):6$$

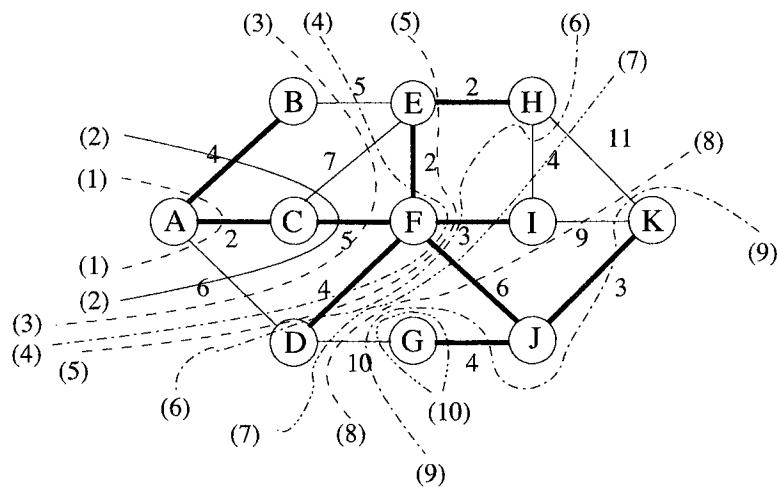


Figura 12.31

12.6 Tema d'esame 06

Esercizio 1 (1.0 punti)

Sia data la seguente sequenza di coppie, dove la relazione $i-j$ indica che il nodo i è adiacente al nodo j :

0-2 2-4 5-1 4-8 7-3 5-9 9-4 5-6 6-3

si applichi un algoritmo di on-line connectivity con quick-union, riportando a ogni passo il contenuto del vettore e la foresta di alberi al passo finale. I vertici sono denominati con interi tra 0 e 9.

Soluzione

La Tabella 12.7 riporta la configurazione finale del vettore `id`, la Figura 12.32 quella dell'albero.

0	1	2	3	4	5	6	7	8	9
2	9	4	3	8	1	3	3	6	8

Tabella 12.7

Esercizio 2 (2.0 punti)

Sia data la sequenza di interi, supposta memorizzata in un vettore:

28 19 29 31 18 43 21 15 12 39 35 25

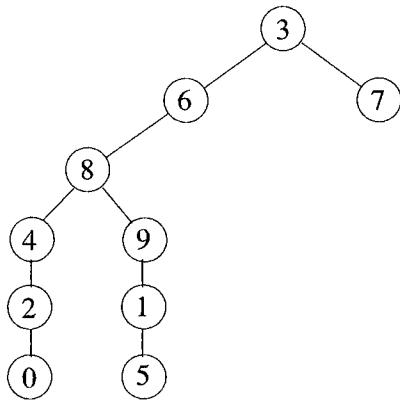


Figura 12.32

Si eseguano i primi due passi dell'algoritmo di quick sort per ottenere un ordinamento ascendente. I passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le 2 partizioni del vettore originale e le 2 partizioni delle partizioni trovate al punto precedente. Si specifichino i pivot scelti e si indichino i passaggi rilevanti.

Soluzione

Il pivot al primo passo è $x = 25$. Al secondo passo il pivot è $x = 18$ per la partizione di sinistra e $x = 43$ per la partizione di destra. Il risultato è il seguente:

12 15 | 18 | 21 19 || 25 || 31 29 28 39 35 | 43

Esercizio 3 (2.0 punti)

Si supponga di aver memorizzato tutti i numeri compresi tra 1 e 1000 in un albero di ricerca binario e che si stia cercando il numero 538. Quali tra queste non possono essere le sequenze esaminate durante la ricerca?

500 600 550 525 535 545 537 540 539 538
 500 600 550 525 535 545 537 538
 100 200 300 400 500 550 540 530 538
 400 700 651 625 325 555 550 523 535 538

Soluzione

La soluzione è riportata in Figura 12.33.

Esercizio 5 (2.0 punti)

Sia data la sequenza di chiavi intere THEDEATHLY, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A=1, \dots, Z=26$). Si riporti la struttura

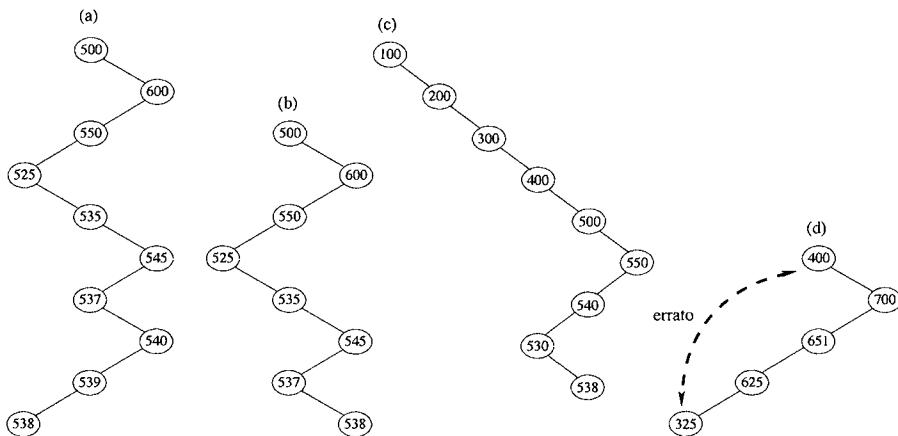


Figura 12.33

di una tabella di hash di dimensione 23, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza indicata. Si supponga di utilizzare l'open addressing con linear probing.

Soluzione

La Tabella 12.8 riporta in sequenza: l'ordine progressivo dei caratteri alfabetici, la valutazione iniziale della funzione di hash ($step = 0$), quella a seguito delle varie collisioni ($step = 1$), e la tabella di hash definitiva, ottenuta al termine dell'inserzione delle chiavi.

Esercizio 6 (5.0 punti)

Sia dato il grafo orientato rappresentato in Figura 12.34. Considerando A come vertice di partenza:

- ▷ se ne effettui una visita in profondità, etichettando i vertici con i tempi di scoperta e di fine elaborazione nel formato $\text{tempo}_1/\text{tempo}_2$ (2.0 punti). Qualora necessario, si trattino i vertici secondo l'ordine alfabetico.
- ▷ lo si ridisegni, etichettando gli archi come T (tree), B (back), F (forward), C (cross) (1.5 punti).
- ▷ se ne effettui una visita in ampiezza etichettando i vertici con la loro distanza minima dalla sorgente (1.5 punti).

Soluzione

La Figura 12.35 riporta la soluzione per la visita in profondità. La visita in ampiezza fornisce:

$$\begin{array}{ccccccc} 0 & & 1 & & 2 & & 5 \\ A & | & B F & | & C G I & | & H \\ & & & & & & \end{array}$$

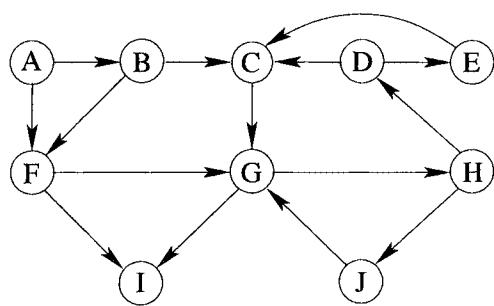


Figura 12.34

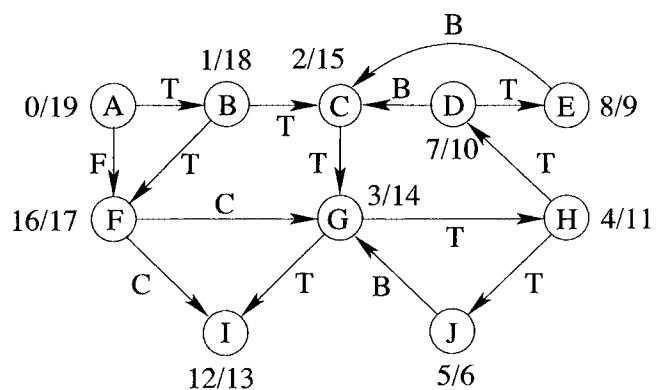


Figura 12.35

A 1	B 2	C 3	D 4	E 5	F 6	G 7	H 8	I 9	J 10	K 11	L 12	M 13
N 14	O 15	P 16	Q 17	R 18	S 19	T 20	U 21	V 22	W 23	X 24	Y 25	Z 26

	step = 0	step = 1
T_1	20	
H_1	8	
E_1	5	
D	4	
E_2	5	6
A	1	
T_2	20	21
H_2	8	9
L	12	
Y	2	

0	1	2	3	4	5	6	7	8	9	10
	A	Y		D	E_1	E_2		H_1	H_2	

11	12	13	14	15	16	17	18	19	20
	L						T_1	T_2	

Tabella 12.8

12.7 Tema d'esame 07

Esercizio 1 (2.0 punti)

Si risolva, mediante il metodo dello sviluppo (unfolding), la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 3 \cdot T\left(\frac{n}{2}\right) + n^2 & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Soluzione

$$T(n) = O(n^2)$$

Esercizio 2 (2.0 punti)

Si partizionino il BST di Figura 12.36 attorno alla chiave R.

Soluzione

La Figura 12.37 riporta la soluzione.

Esercizio 3 (2.0 punti)

Sia data la sequenza di chiavi intere SECONDCHANCE, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto (A=1, ..., Z=26). Si riporti la

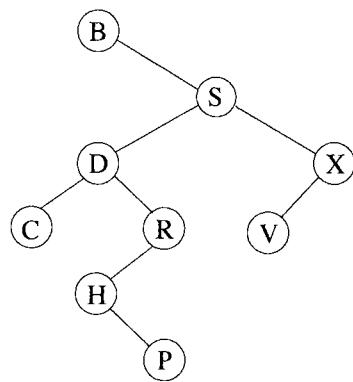


Figura 12.36

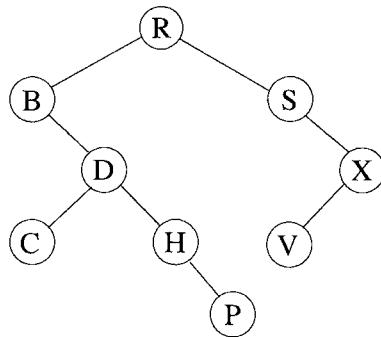


Figura 12.37

struttura di una tabella di hash di dimensione 29, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza indicata. Si supponga di utilizzare l'open addressing con quadratic probing (con $c_1 = 0.5$ e $c_2 = 0.5$) e che la funzione di hash primaria sia $h(k) = k \bmod 29$.

Soluzione

La Tabella 12.9 riporta in sequenza: l'ordine progressivo dei caratteri alfabetici, la valutazione iniziale della funzione di hash ($step = 0$), quella a seguito delle varie collisioni ($step = 1, 2, 3$), e la tabella di hash definitiva, ottenuta al termine dell'inserzione delle chiavi.

A 1	B 2	C 3	D 4	E 5	F 6	G 7	H 8	I 9	J 10	K 11	L 12	M 13
N 14	O 15	P 16	Q 17	R 18	S 19	T 20	U 21	V 22	W 23	X 24	Y 25	Z 26

	$step = 0$	$step = 1$	$step = 2$	$step = 3$
S	2			
E ₁	5			
C ₁	3			
O	15			
N ₁	14			
D	4			
C ₂	3	4	6	
H	8			
A	1			
N ₂	14	15	17	
C ₃	3	4	6	9
E ₂	5	6	8	11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	A		C ₁	D	E ₁	C ₂		H	C ₃		E ₂			N ₁
15	16	17	18	19	20	21	22	23	24	25	26	27	28	
O		N ₂		S										

Tabella 12.9

Esercizio 4 (6.0 punti)

Sia dato il grafo orientato rappresentato in Figura 12.38. Lo si rappresenti come lista delle adiacenze (0.5 punti) e come matrice delle adiacenze (0.5 punti). Considerando A come vertice di partenza:

- ▷ se ne effettui una visita in profondità, etichettando i vertici con i tempi di scoperta e di fine elaborazione nel formato tempo₁/tempo₂ (2.0 punti). Qualora necessario, si trattino i vertici secondo l'ordine alfabetico

- ▷ lo si ridisegni, etichettando gli archi come T (tree), B (back), F (forward), C (cross) (1.5 punti)
- ▷ se ne effettui una visita in ampiezza etichettando i vertici con la loro distanza minima dalla sorgente (1.5 punti).

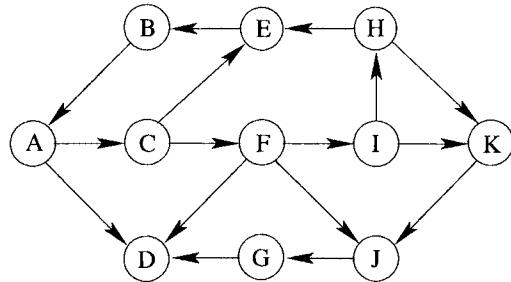


Figura 12.38

Soluzione

La Figura 12.39 riporta la soluzione. La visita in ampiezza fornisce:

$$\begin{array}{c|c|c|c|c} 0 & 1 & 2 & 3 & 4 \\ \hline A & C\ D & E\ F & B\ I\ J & H\ K\ G \end{array}$$

12.8 Tema d'esame 08**Esercizio 1 (1.0 punti)**

Si ordini in maniera ascendente la seguente sequenza di interi mediante insertion sort, indicando i passaggi rilevanti:

5 1 15 31 8 24 18 13 6 9

Soluzione

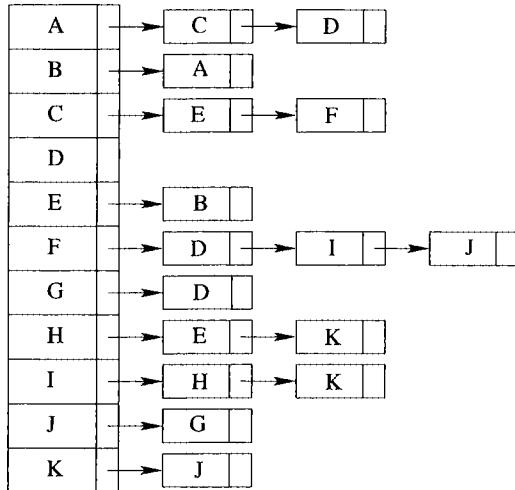
La soluzione è riportata in Figura 12.40.

Esercizio 2 (1.0 punti)

Si ordini in maniera ascendente mediante counting sort il seguente vettore di interi:

7 1 7 2 7 6 3 2 0 9 4 9 8 0 4

Si indichino i passaggi rilevanti.



	A	B	C	D	E	F	G	H	I	J	K
A	0	0	1	1	0	0	0	0	0	0	0
B	1	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0	0
E	0	1	0	0	0	0	0	0	0	0	0
F	0	0	0	1	0	0	0	0	1	1	0
G	0	0	0	1	0	0	0	0	0	0	0
H	0	0	0	0	1	0	0	0	0	0	1
I	0	0	0	0	0	0	0	1	0	0	1
J	0	0	0	0	0	0	1	0	0	0	0
K	0	0	0	0	0	0	0	0	0	1	0

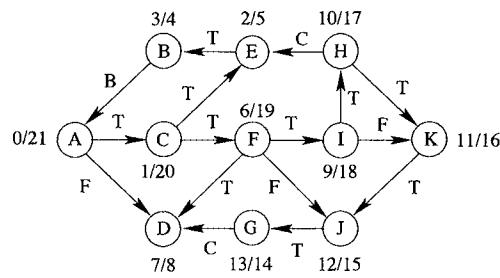


Figura 12.39

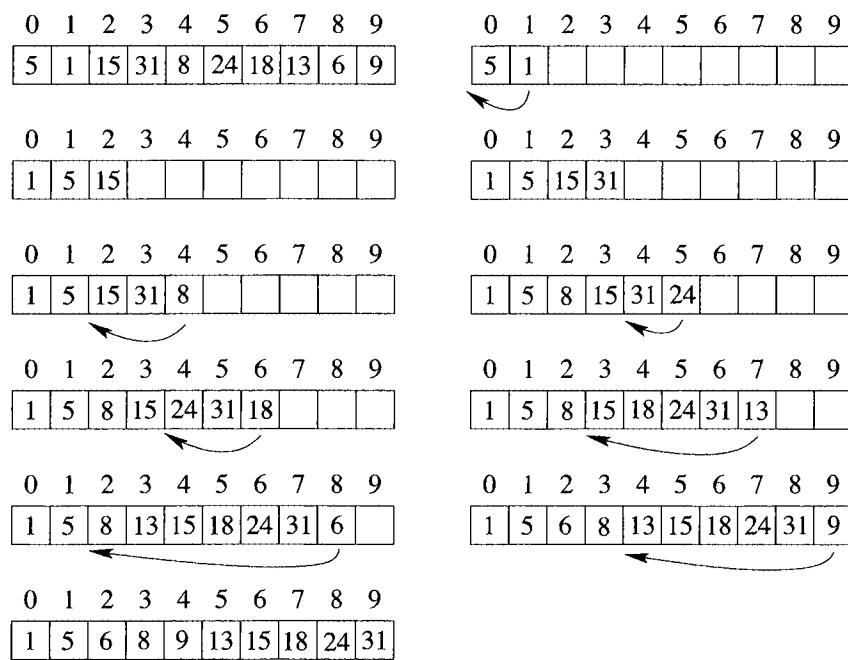


Figura 12.40

Soluzione

La Tabella 12.10 riporta nell'ordine: il vettore iniziale A , il vettore C dopo l'inizializzazione (C^1), il calcolo delle occorrenze semplici (C^2) e di quelle multiple (C^3) e il vettore finale B . Sono inoltre riportati i decrementi dei valori degli elementi di C durante le varie fasi di inserzione degli elementi originali nel vettore finale.

A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	7	1	7	2	7	6	3	2	0	9	4	9	8	0	4
C^1	0	1	2	3	4	5	6	7	8	9					
	0	0	0	0	0	0	0	0	0	0					
C^2	0	1	2	3	4	5	6	7	8	9					
	2	1	2	1	2	0	1	3	1	2					
C^3	0	1	2	3	4	5	6	7	8	9					
	2	3	5	6	8	8	9	12	13	15					
	1	2	4	5	7	8	9	11	12	14					
	0		3		6			10		13					
								9							
B	0	λ	5	4	6	7	8	9	10	11	12	13	14	15	
	0	0	1	2	2	3	4	4	6	7	7	7	8	9	9

Tabella 12.10

Esercizio 3 (1.5 punti)

Sia dato l'albero binario di Figura 12.41. Lo si visiti in ordine pre-order (0.5 punti), in-order (0.5 punti) e post-order (0.5 punti) elencando l'ordine di visita dei nodi.

Soluzione

pre-order: $a \ b \ d \ g \ h \ l \ e \ i \ m \ j \ c \ f \ k$
 in-order: $g \ d \ l \ h \ b \ i \ m \ e \ j \ a \ c \ f \ k$
 post-order: $g \ l \ h \ d \ m \ i \ j \ e \ b \ k \ f \ c \ a$

Esercizio 4 (2.0 punti)

Si inseriscano in sequenza nella *radice* di un BST, inizialmente supposto vuoto, le chiavi:

19 12 13 48 21 37 85

riportando il risultato a ogni passo.

Soluzione

La Figura 12.42 riporta la soluzione dopo le inserzioni in sequenza di 19 (a), 12 (b), 13 (c), 48 (d), 21 (e), 37 (f) e 85 (g).

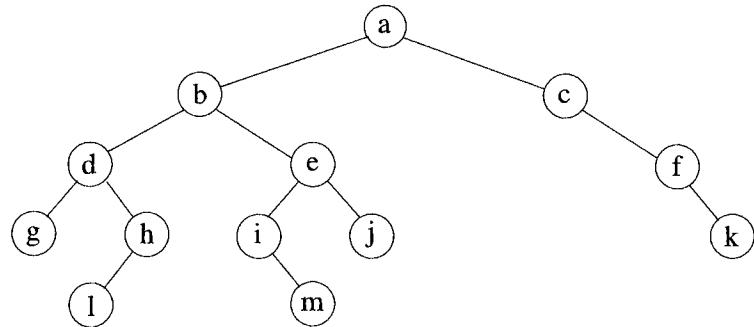


Figura 12.41

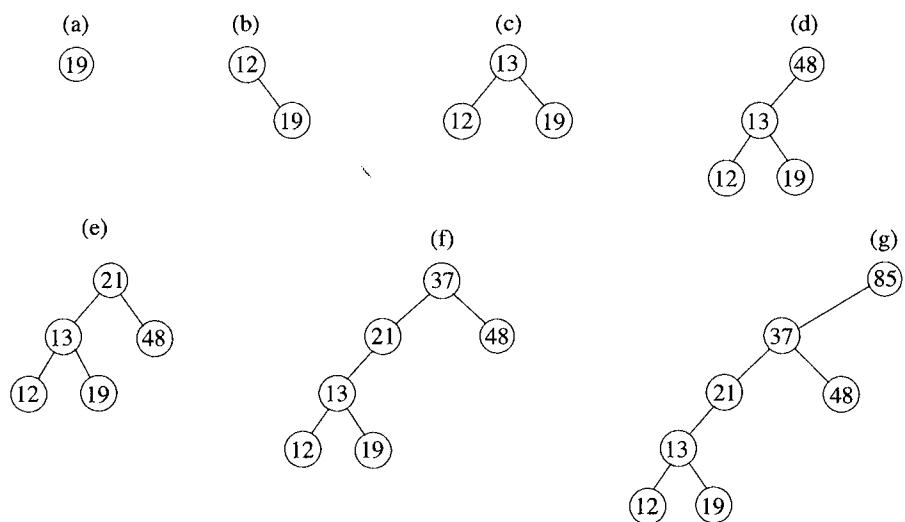


Figura 12.42

Esercizio 5 (2.0 punti)

Sia dato il grafo non orientato rappresentato in Figura 12.43. Considerando A come vertice di partenza:

- ▷ se ne effettui una visita in profondità, etichettando i vertici con i tempi di scoperta e di fine elaborazione nel formato $\text{tempo}_1/\text{tempo}_2$. Qualora necessario, si trattino i vertici secondo l'ordine crescente
- ▷ se ne determinino le componenti connesse.

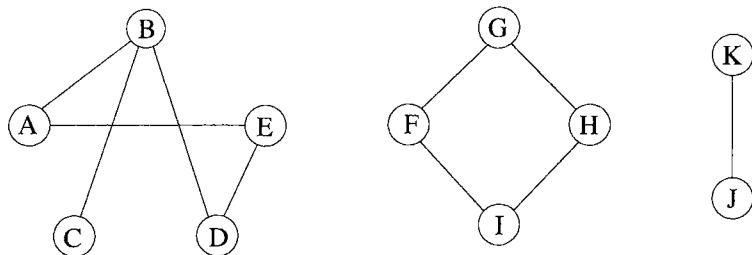


Figura 12.43

Soluzione

La Figura 12.44 riporta la soluzione.

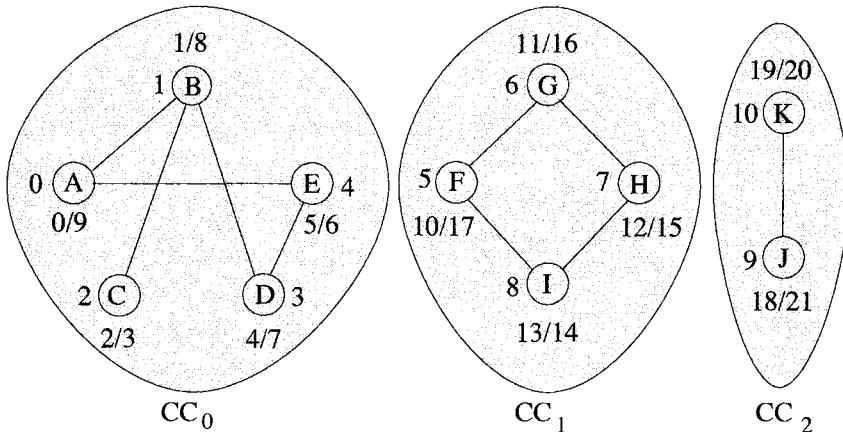


Figura 12.44

Esercizio 6 (2.5 punti)

Sia dato il grafo orientato rappresentato in Figura 12.45. Se ne trovino le componenti fortemente connesse mediante l'algoritmo di Kosaraju. Si consideri A come vertice di partenza e, qualora necessario, si trattino i vertici secondo l'ordine alfabetico

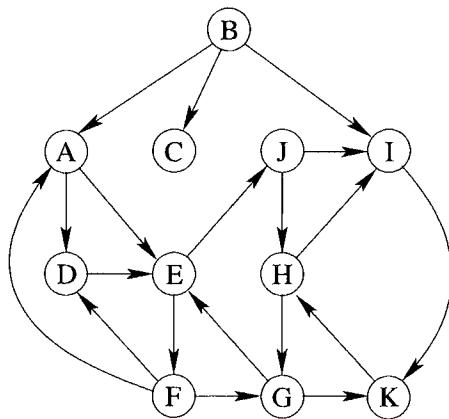


Figura 12.45

Soluzione

La Figura 12.46 in (a) riporta il risultato della visita in profondità del grafo trasposto, in (b) le componenti fortemente connesse individuate sul grafo di partenza.

Esercizio 7 (2.0 punti)

Sia dato il grafo orientato pesato di Figura 12.47. Si determinino i valori di tutti i cammini minimi che collegano il vertice **a** con ogni altro vertice mediante l'algoritmo di Dijkstra. Si assuma, qualora necessario, un ordine alfabetico per i vertici e gli archi.

Soluzione

La soluzione è riportata in Figura 12.48. L'ordine con cui i nodi vengono estratti dalla coda a priorità è il seguente:

A C B G H I J F E D

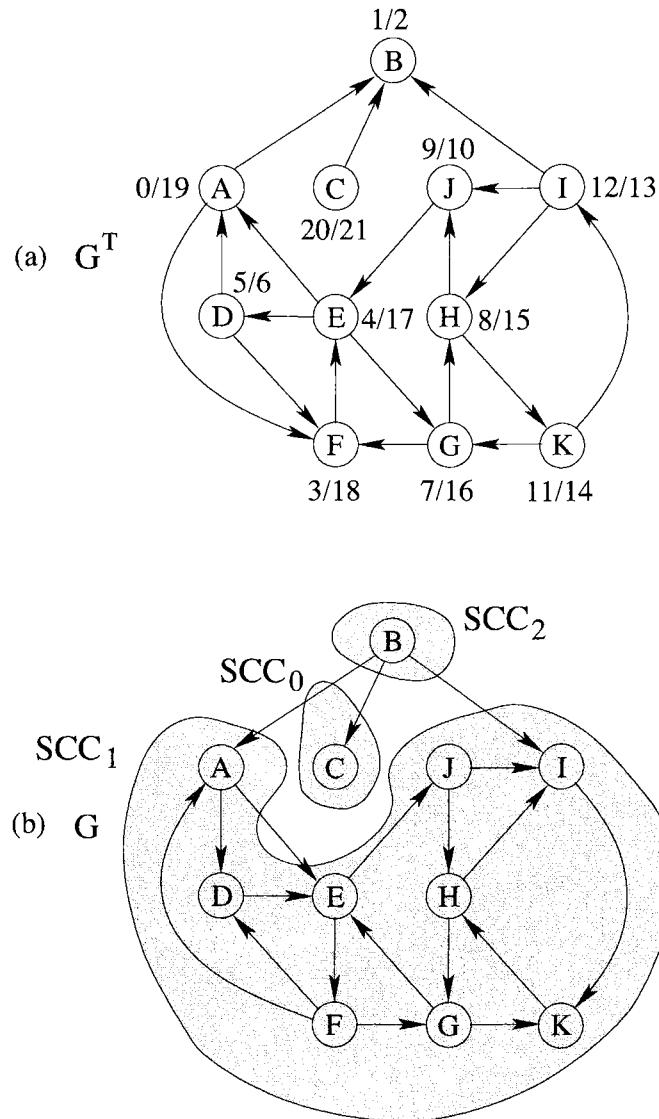


Figura 12.46

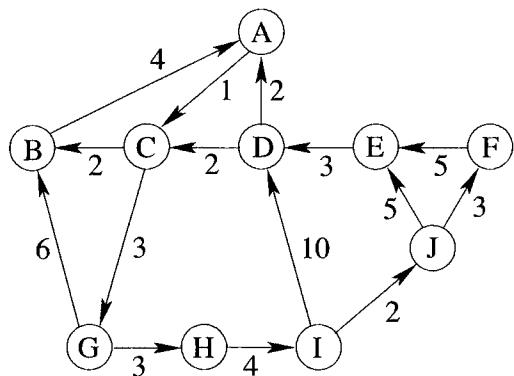


Figura 12.47

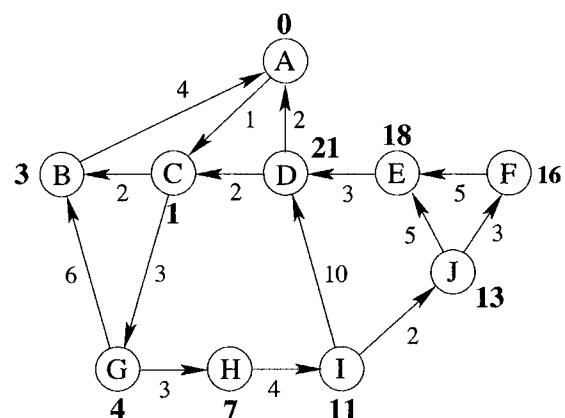


Figura 12.48

12.9 Tema d'esame 09

Esercizio 1 (2.0 punti)

Si risolva, mediante il metodo dello sviluppo (unfolding), la seguente equazione alle ricorrenze:

$$\begin{aligned} T(n) &= 3 \cdot T\left(\frac{n}{2}\right) + n & n \geq 2 \\ T(1) &= 1 \end{aligned}$$

Soluzione

$$T(n) = O(n^{\log_2 3})$$

Esercizio 2 (2.0 punti)

Sia data la sequenza di interi, supposta memorizzata in un vettore:

8 9 99 13 38 53 11 65 22 79 31 40

Si eseguano i primi due passi dell'algoritmo di quick sort per ottenere un ordinamento ascendente. I passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le 2 partizioni del vettore originale e le 2 partizioni delle partizioni trovate al punto precedente. Si specifichino i pivot scelti e si indichino i passaggi rilevanti.

Soluzione

Il pivot al primo passo è $x = 40$. Al secondo passo il pivot è $x = 11$ per la partizione di sinistra e $x = 65$ per la partizione di destra. Il risultato è il seguente:

8 9 | 11 | 13 38 22 31 || 40 || 53 | 65 | 99 79

Esercizio 3 (2.0 punti)

Si inseriscano in sequenza nella *radice* di un BST, inizialmente supposto vuoto, le chiavi:

9 24 31 68 11 13 58

riportando il risultato a ogni passo.

Soluzione

La Figura 12.49 riporta la soluzione dopo le inserzioni in sequenza di 9 (a), 24 (b), 31 (c), 68 (d), 11 (e), 13 (f) e 58 (g).

Esercizio 4 (2.0 punti)

Sia data la sequenza di chiavi intere ANDWINTERCAME, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A=1, \dots, Z=26$). Si riporti la struttura di una tabella di hash di dimensione 29, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza indicata. Si supponga di utilizzare l'open addressing con double hashing. Si definiscano opportune funzioni di hash.

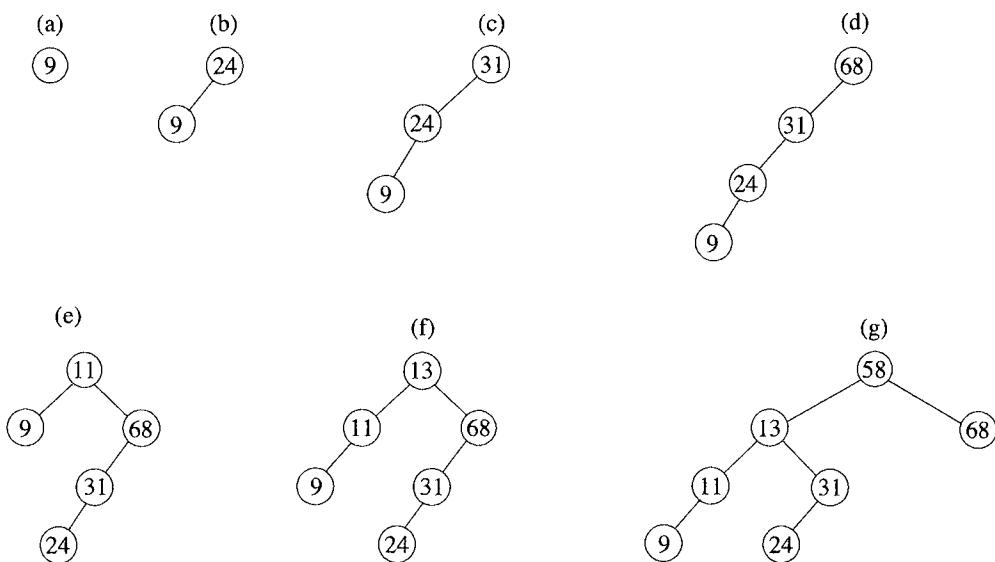


Figura 12.49

Soluzione

La Tabella 12.11 riporta in sequenza: l'ordine progressivo dei caratteri alfabetici, le funzioni di hash h_1 e h_2 , la valutazione iniziale della funzione di hash ($step = 0$), quella a seguito delle varie collisioni ($step = 1, 2, 3$), e la tabella di hash definitiva, ottenuta al termine dell'inserzione delle chiavi.

Esercizio 5 (2.0 punti)

Si determinino i punti di articolazione del grafo non orientato e connesso di Figura 12.50. Si consideri **Z** come vertice di partenza e, qualora necessario, si trattino i vertici secondo l'ordine alfabetico.

Soluzione

La Figura 12.51 riporta il risultato della visita in profondità del grafo. L'unico punto di articolazione è **Y**.

Esercizio 6 (2.0 punti)

Si ordini in modo topologico il DAG di Figura 12.52. Qualora necessario, si trattino i vertici secondo l'ordine alfabetico e si assuma che la lista delle adiacenze sia anch'essa ordinata alfabeticamente. La visita inizi dal vertice **A**.

Soluzione

L'ordinamento topologico è il successivo:

A D C F G B E J K H I L

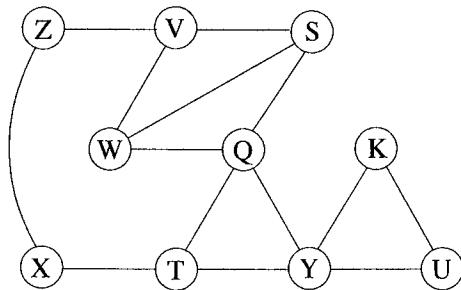


Figura 12.50

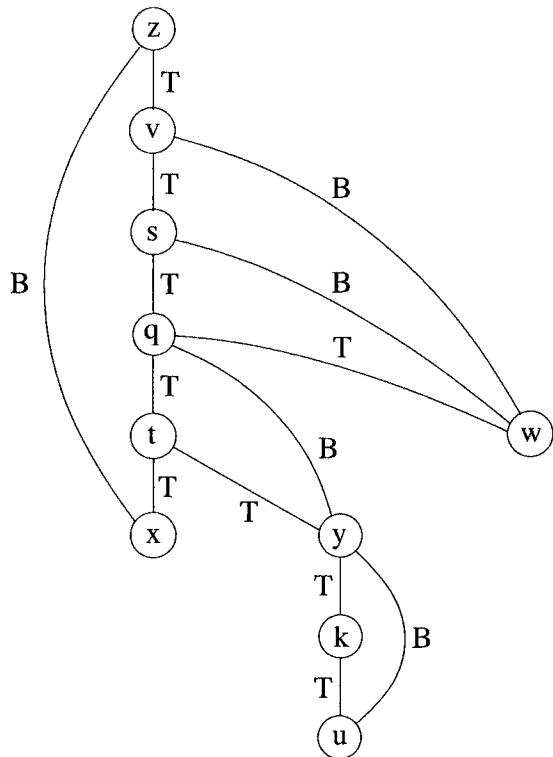


Figura 12.51

A 1	B 2	C 3	D 4	E 5	F 6	G 7	H 8	I 9	J 10	K 11	L 12	M 13
N 14	O 15	P 16	Q 17	R 18	S 19	T 20	U 21	V 22	W 23	X 24	Y 25	Z 26

$$h_1(k) = k \bmod 29 \quad h_2(k) = (1 + k \bmod 97) \bmod 29$$

	step = 0	step = 1	step = 2	step = 3
A ₁	1			
N ₁	14			
D	4			
W	23			
I	9			
N ₂	14	0		
T	20			
E ₁	5			
R	18			
C	3			
A ₂	1	3	5	7
M	13			
E ₂	5	11		

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
N ₂	A ₁		C	D	E ₁		A ₂		I		E ₂		M	N ₁
15	16	17	18	19	20	21	22	23	24	25	26	27	28	
			R		T			W						

Tabella 12.11

12.10 Tema d'esame 10

Esercizio 1 (1.0 punti)

Sia data la seguente sequenza di coppie, dove la relazione $i-j$ indica che il nodo i è adiacente al nodo j :

$$0-3 \ 3-4 \ 5-0 \ 2-8 \ 7-3 \ 4-7 \ 9-0 \ 5-6 \ 9-3 \ 4-0$$

si applichi un algoritmo di on-line connectivity con quick-find, riportando a ogni passo il contenuto del vettore e la foresta di alberi al passo finale. I vertici sono denominati con interi tra 0 e 9.

Soluzione

La Tabella 12.12 riporta la configurazione finale del vettore id. La Figura 12.53 riporta la configurazione finale dell'albero.

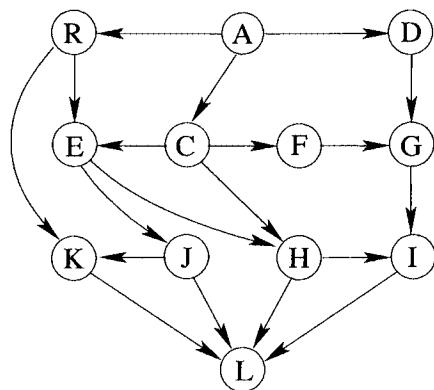


Figura 12.52

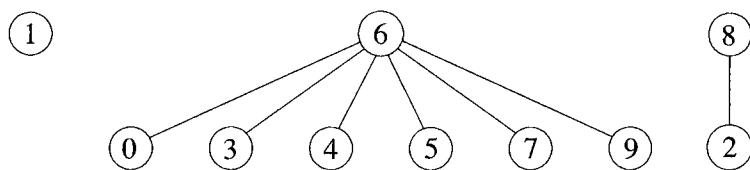


Figura 12.53

.0	1	2	3	4	5	6	7	8	9
6	1	8	6	6	6	6	6	8	6

Tabella 12.12

Esercizio 2 (1.0 punti)

Sia data la sequenza di interi, supposta memorizzata in un vettore:

15 11 2 19 9 6 8 13 16 7

La si ordini in ordine crescente utilizzando l'algoritmo di selection sort. Si riportino graficamente i passi rilevanti dell'algoritmo e il risultato finale.

Soluzione

La Figura 12.54 riporta i passaggi e la configurazione finale.

Esercizio 3 (2.0 punti)

Sia dato un albero binario con 12 nodi. Nella visita si ottengono le 3 seguenti sequenze di chiavi:

pre-order:	10	15	20	7	40	9	1	0	13	4	21	5
in-order:	20	15	7	10	1	9	0	40	4	13	21	5
post-order:	20	7	15	1	0	9	4	5	21	13	40	10

Si disegni l'albero binario di partenza.

Soluzione

La Figura 12.55 riporta la soluzione.

Esercizio 4 (2.0 punti)

Sia data la sequenza di chiavi intere ADAYWITHOUTRAIN, dove ciascun carattere è individuato dal suo ordine progressivo nell'alfabeto ($A=1, \dots, Z=26$). Si riporti la struttura di una tabella di hash di dimensione 19, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza di cui sopra. Si supponga di utilizzare una tabella di hash con linear chaining.

Soluzione

La Tabella 12.13 riporta l'ordine progressivo dei caratteri alfabetici, la Figura 12.56 la soluzione.

Esercizio 5 (2.0 punti)

Sia dato il grafo non orientato rappresentato in Figura 12.57. Considerando L come vertice di partenza:

- ▷ se ne effettui una visita in profondità, etichettando i vertici con i tempi di scoperta e di fine elaborazione nel formato $\text{tempo}_1/\text{tempo}_2$. Qualora necessario, si trattino i vertici secondo l'ordine alfabetico
- ▷ se ne determinino le componenti connesse.

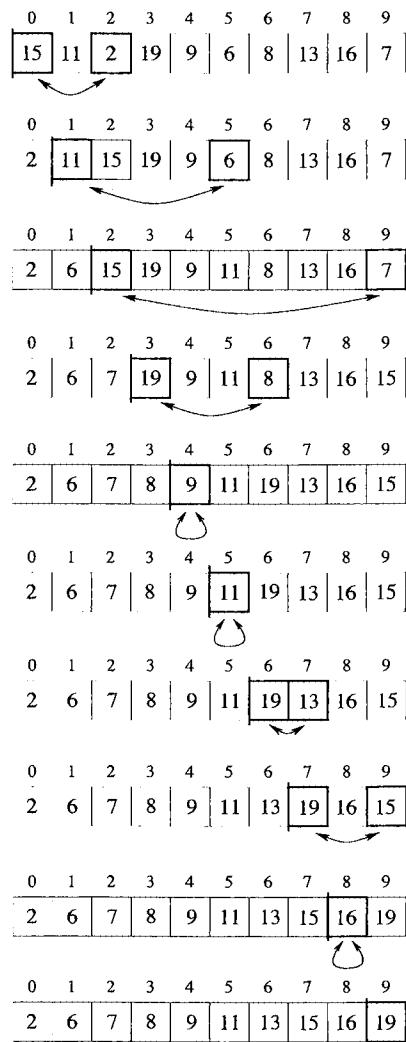


Figura 12.54

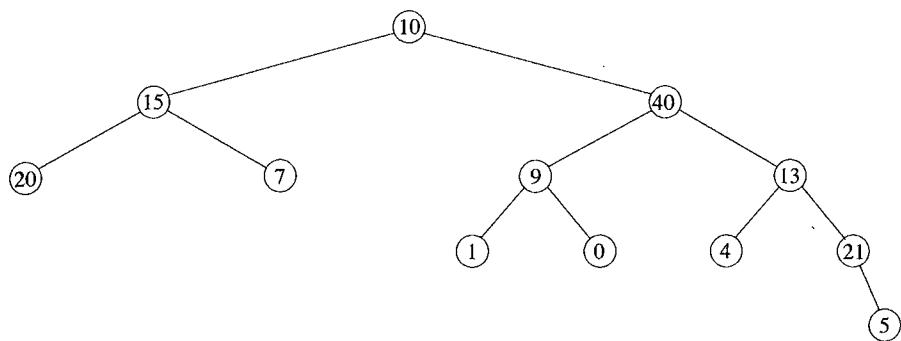


Figura 12.55

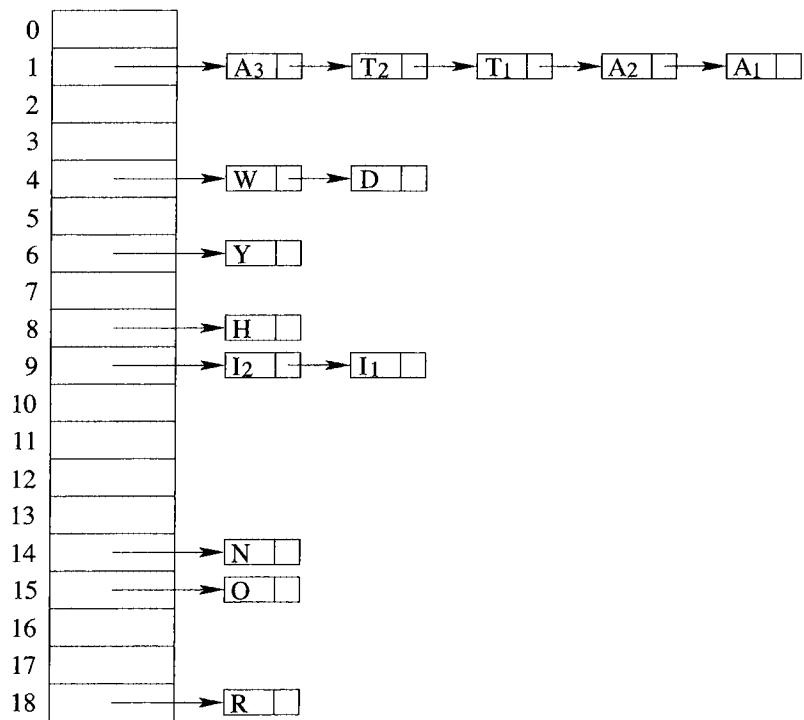


Figura 12.56

A 1	B 2	C 3	D 4	E 5	F 6	G 7	H 8	I 9	J 10	K 11	L 12	M 13
N 14	O 15	P 16	Q 17	R 18	S 19	T 20	U 21	V 22	W 23	X 24	Y 25	Z 26

Tabella 12.13

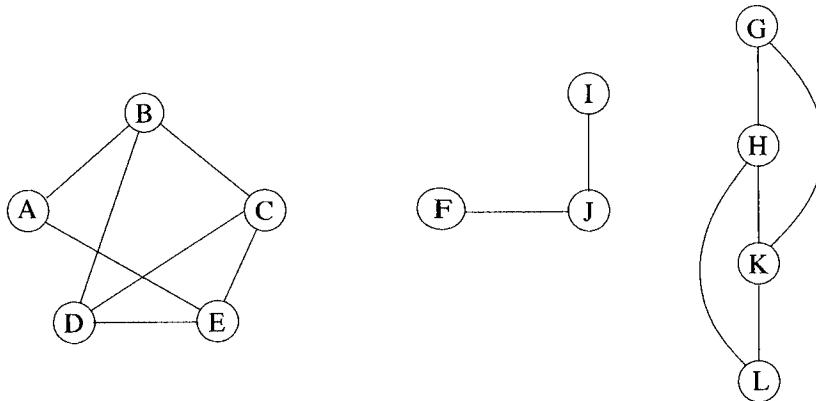


Figura 12.57

Soluzione

La Figura 12.58 riporta la soluzione.

Esercizio 6 (4.0 punti)

Sia dato il grafo non orientato pesato di Figura 12.59. Si determini un albero ricoprente minimo applicando l'algoritmo di Kruskal (2.0 punti) e l'algoritmo di Prim (2.0 punti) dal vertice C . Si riportino i passaggi rilevanti, si disegni l'albero ricoprente minimo e si fornisca il valore del peso minimo.

Soluzione

La Figura 12.60 riporta la soluzione con l'algoritmo di Prim. L'ordine di scelta degli archi per Kruskal è il seguente:

$$\begin{array}{lllll}
 (A, C):1 & (D, E):1 & (E, I):1 & (F, J):1 & (A, B):2 \\
 (B, E):2 & (D, H):2 & (K, G):2 & (B, F):3 & (F, G):5
 \end{array}$$

In entrambi i casi il peso totale del cammino è uguale a 20.

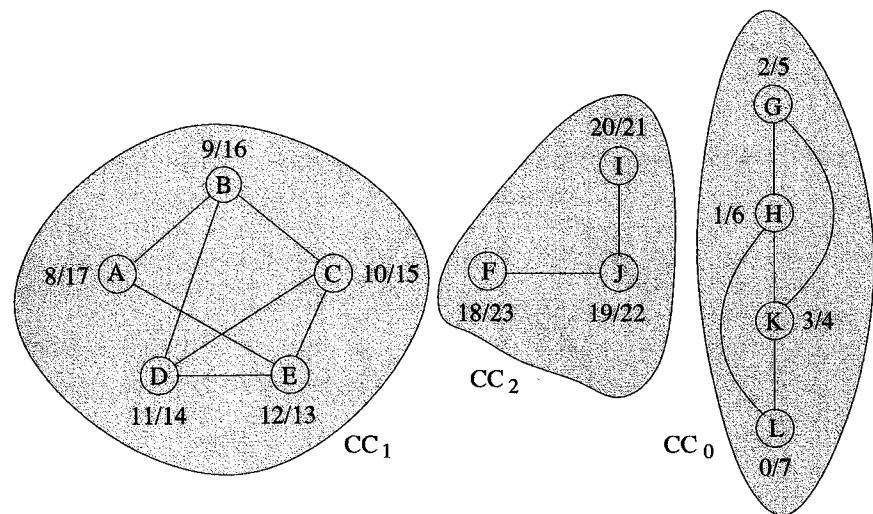


Figura 12.58

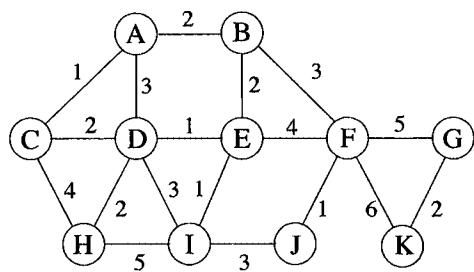


Figura 12.59

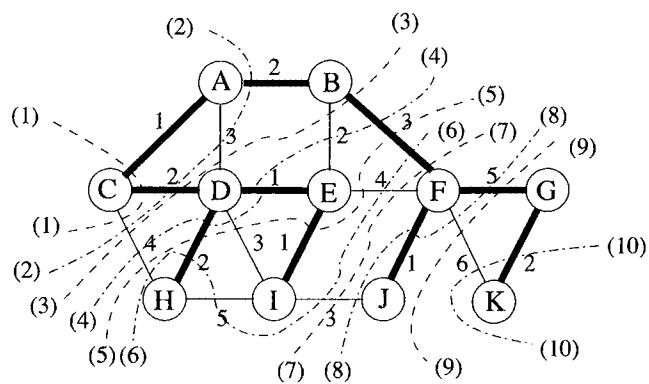
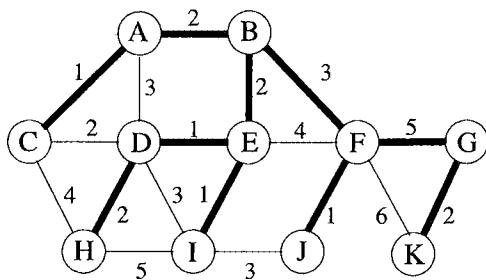


Figura 12.60

12.11 Tema d'esame 11

Esercizio 1 (1.5 punti)

Si ordini in maniera ascendente mediante merge sort il seguente vettore di interi:

8 7 3 8 3 7 5 3 6 3 0 9 7 1 9 2

Si riportino graficamente i passaggi rilevanti dell'algoritmo e il risultato finale.

Soluzione

La Figura 12.61 riporta la soluzione (si osservi che la figura è stata ruotata di 90 gradi per ottenerne una maggiore leggibilità).

Esercizio 2 (2.0 punti)

Sia data la sequenza di interi, supposta memorizzata in un vettore:

24 5 1 21 6 17 3 8 18 15 13 12

La si trasformi in uno heap. Si riportino graficamente i diversi passi della costruzione dello heap e il risultato finale. Si ipotizzi che, alla fine, nella radice dello heap sia memorizzato il valore massimo. Si eseguano su tale heap i primi 2 passi dell'algoritmo di heap sort.

Soluzione

La Figura 12.62 mostra nell'ordine: la situazione iniziale specificata nel testo (a), lo heap all'inizio (b) e dopo i primi due passi di heap sort (c) e (d).

Esercizio 3 (1.0 punti)

Si esprima in notazione infissa, prefissa e postfissa la seguente espressione aritmetica mediante visita dell'albero binario corrispondente:

$$((A * B) + C) * (D * (E + F))$$

Soluzione

La Figura 12.63 riporta l'albero.

pre-order:	*	+	*	A	B	C	*	D	+	E	F
in-order:	A	*	B	+	C	*	D	*	E	+	F
post-order:	A	B	*	C	+	D	E	F	+	*	*

Esercizio 4 (1.5 punti)

Si effettuino, secondo l'ordine specificato, le seguenti operazioni su un BST supposto inizialmente vuoto (+ indica una inserzione in foglia, – una cancellazione):

+31 +20 +27 +40 +35 +44 +37 +26 +45 +21 +12 +43 -40 -31 -20

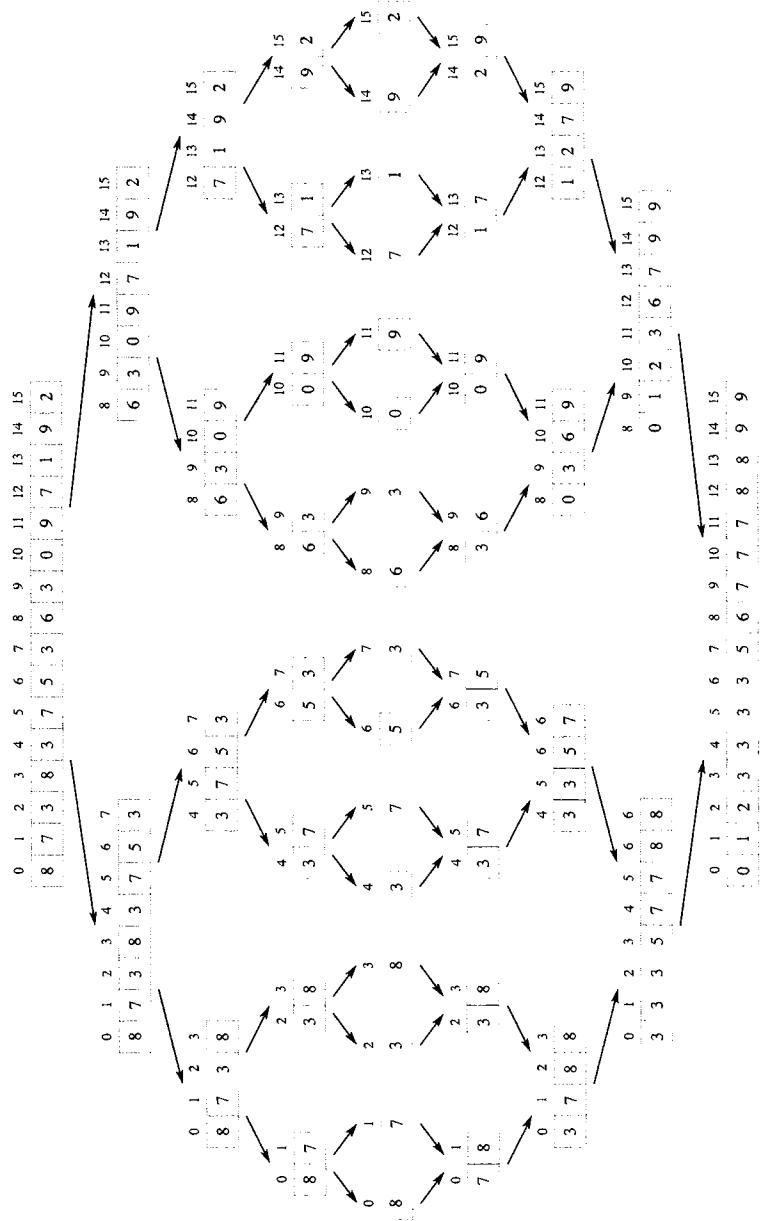


Figura 12.61

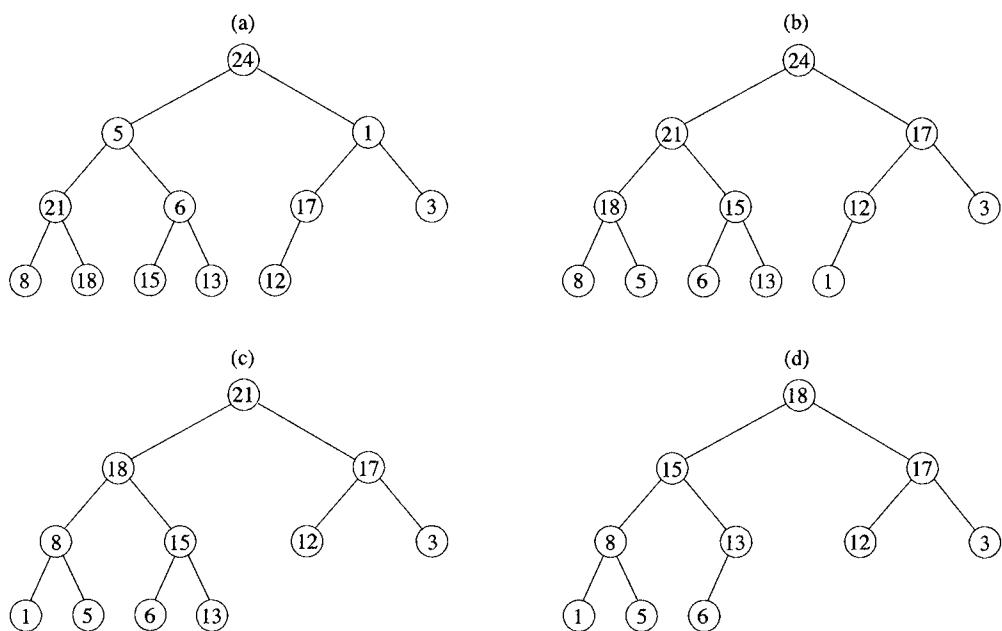


Figura 12.62

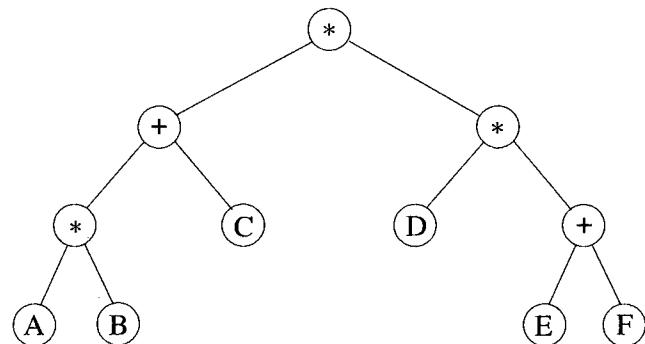


Figura 12.63

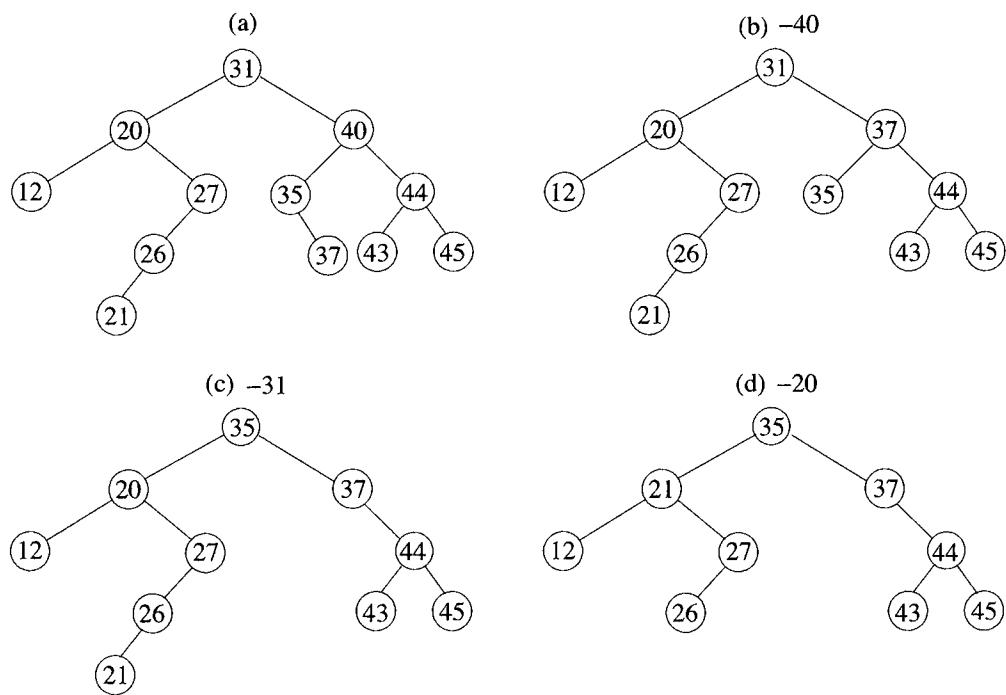


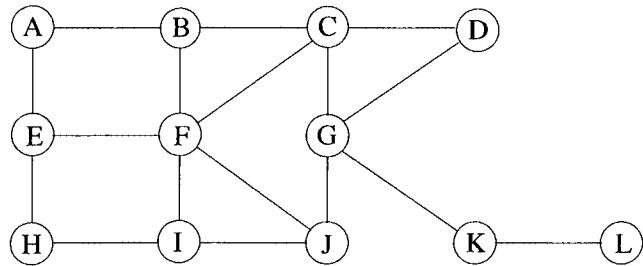
Figura 12.64

Soluzione

La Figura 12.64 riporta il BST ottenuto dopo tutte le inserzioni (a) e la configurazione risultante dopo la cancellazione dei valori 40 (b), 31 (c) e 20 (d).

Esercizio 5 (2.0 punti)

Si determinino i punti di articolazione del grafo non orientato e connesso di Figura 12.65. Si consideri A come vertice di partenza e, qualora necessario, si trattino i vertici secondo l'ordine alfabetico.

**Figura 12.65****Soluzione**

La Figura 12.66 riporta il risultato della visita in profondità del grafo. I punti di articolazione sono G e K.

Esercizio 6 (2.0 punti)

Si ordini in ordine topologico inverso il DAG di Figura 12.67. Qualora necessario, si trattino i vertici secondo l'ordine alfabetico e si assuma che la lista delle adiacenze sia anch'essa ordinata alfabeticamente. La visita inizi dal vertice A.

Soluzione

L'ordinamento topologico inverso è il seguente:

K H L I G D J F B E C A

Esercizio 7 (2.0 punti)

Sia dato il grafo orientato pesato di Figura 12.68. Si determinino i valori di tutti i cammini minimi che collegano il vertice A con ogni altro vertice mediante l'algoritmo di Dijkstra. Si assuma, qualora necessario, un ordine alfabetico per i vertici e gli archi.

Soluzione

La soluzione è riportata in Figura 12.69. L'ordine con cui i nodi vengono estratti dalla coda a priorità è:

A B D C E H L G I F

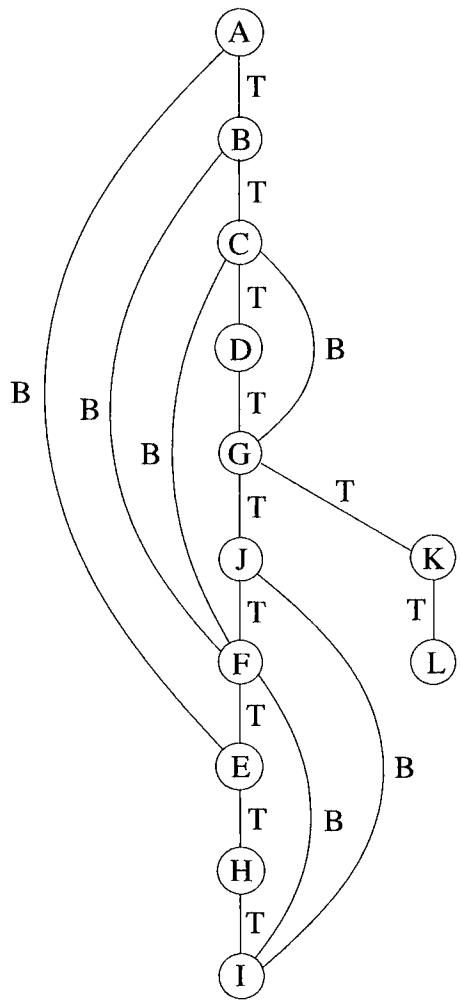


Figura 12.66

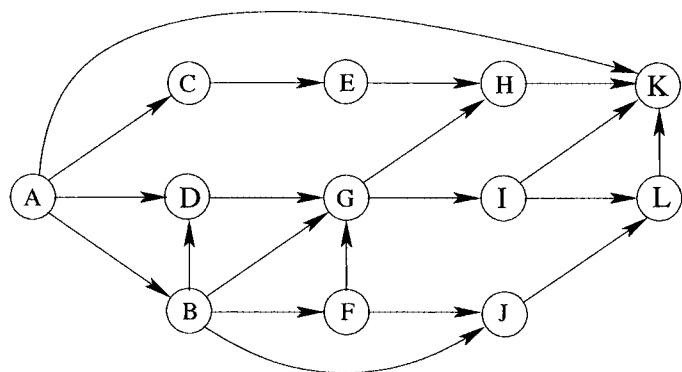


Figura 12.67

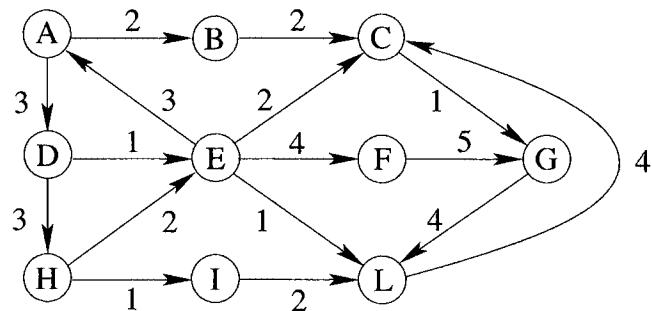


Figura 12.68

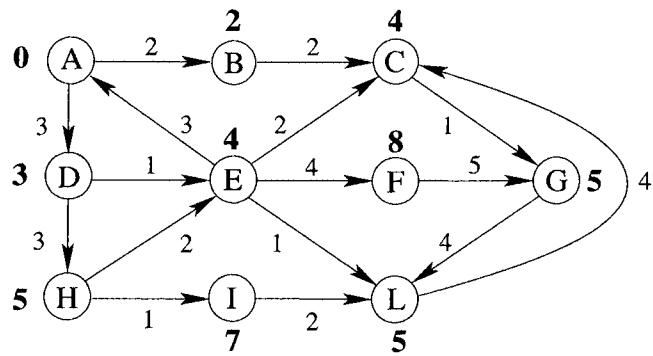


Figura 12.69

