Capitolo 4

ALGORITMI INGORDI (GREEDY)

4.1 Struttura dati Heap

La struttura dati chiamata **Heap** è un array A che può essere visto come un albero binario completo, che per ogni nodo tranne che per la radice soddisfa la **proprietà di Heap**:

$A[\text{Parent}(i)] >= A[i]$

La heap realizza una coda con priorità.

Dell'array A ci interessa conoscere il numero dei suoi elementi, che chiamiamo $lung[A]$, ed $heap\text{-}size[A]$ che è il numero degli elementi della struttura heap memorizzata in A. Come si vede nella figura 4.1, $A[1]$ è la radice dell'albero. Dato poi l'i-esimo nodo dell'albero, si ha che:

$\text{Parent}(i) = A[i/2]$

$\text{Left}(i) = A[2i]$

$\text{Right}(i) = A[2i+1]$

Sempre facendo riferimento all'esempio illustrato in fig. 4.1 avremo $heap\text{-}size[A] = 12$ e $lung[A] = 14$.

La funzione fondamentale per operare con una struttura heap è la cosiddetta funzione **Heapify**. Essa riceve in input l'array A e un indice i. Quando è chiamata, Heapify suppone che i sottoalberi che hanno radice in Left(i) e Right(i) godano della proprietà di heap, ma il nodo $A[i]$ potrebbe essere più piccolo dei suoi figli. Heapify "spinge verso il basso" $A[i]$ e ripristina la proprietà di heap su tutto l'albero.

Lo pseudo codice per Heapify è:

```
HEAPIFY(A, i)
   l ← Left(i);
   r ← Right(i);
   if l <= heap-size and A[l] > A[i];
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14
80	**34**	**46**	**8**	**16**	**23**	**23**	**5**	**7**	**12**	**9**	**8**		

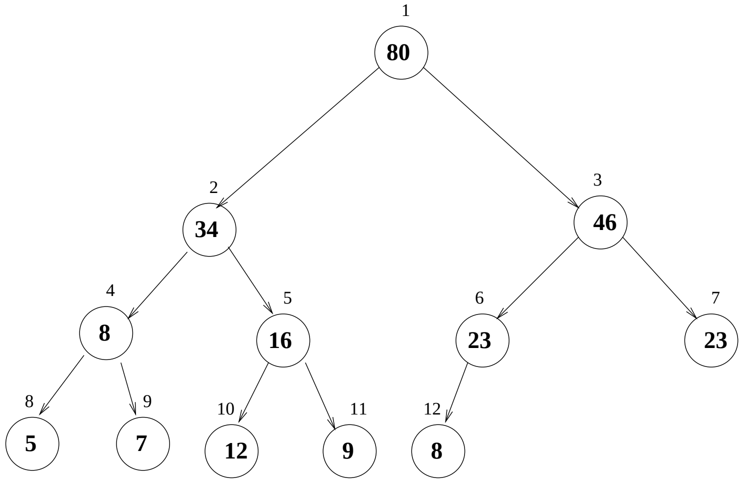


Figura 4.1: Heap visto come array e come albero.

then *largest* $\leftarrow l$;
else *largest* $\leftarrow i$;
if $r <= heap\text{-}size$ and $A[r] > A[largest]$;
then *largest* $\leftarrow r$;
if $largest\ != i$;
then exchange $A[i] \leftarrow A[largest]$;
HEAPIFY(A, *largest*);

La fig. 4.2 illustra il funzionamento della funzione Heapify chiamata per $i = 2$.

4.2 Codice di Huffman

Il **codice di Huffman** è una tecnica di compressione dati molto usata ed efficiente. Sfrutta una tabella della frequenza delle occorrenze di ogni carattere per arrivare ad una rappresentazione ottimale dei caratteri come stringa binaria.

Partiamo con un esempio: supponiamo di voler comprimere una stringa di 80 caratteri, formata solo con le lettere *a, b, c, d, e, f, g* , senza spazi bianchi.

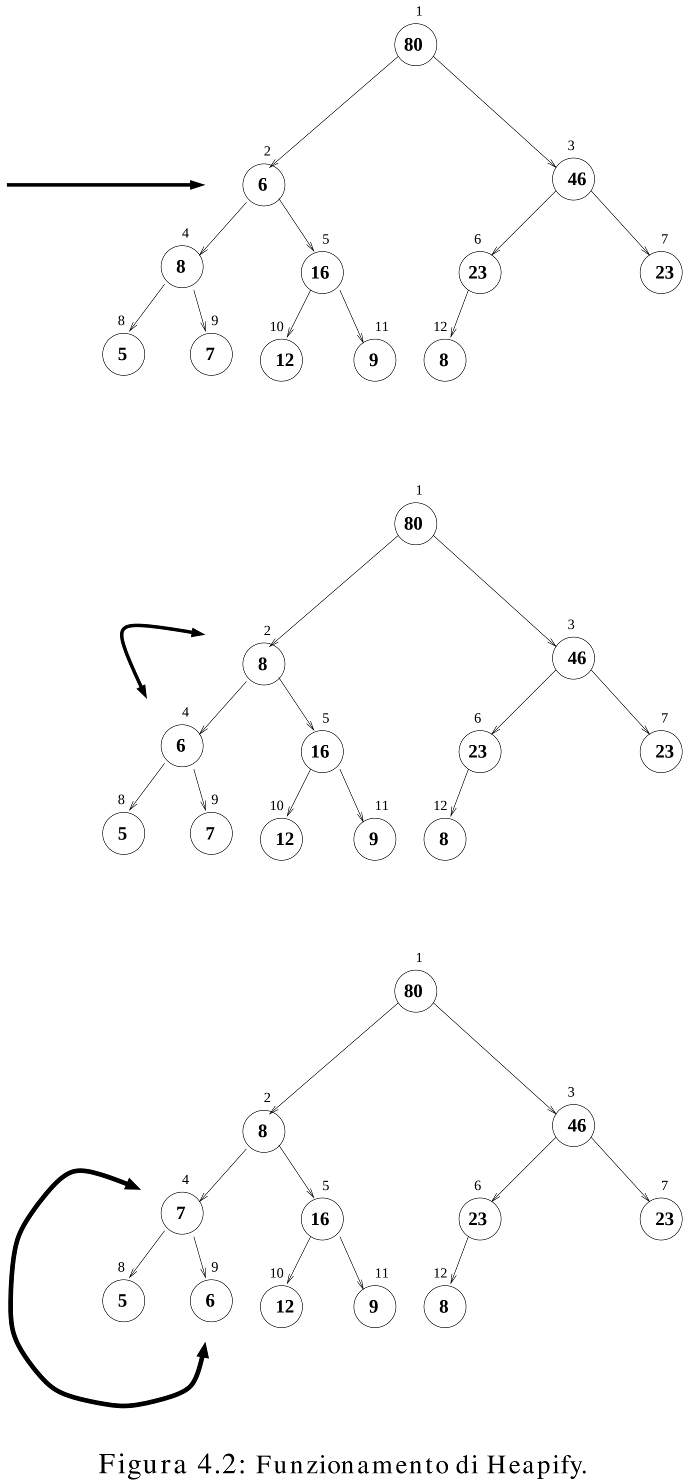


Figura 4.2: Funzionamento di Heapify.

Un primo modo per comprimere i dati consiste nello sfruttare il fatto che nel nostro esempio abbiamo solo 7 lettere. Possiamo rappresentarle usando solo tre bit: a = 000, b = 001, c = 010, d = 011, e = 100, f = 101, g = 111. In questo modo la stringa di 80 caratteri diventa una stringa di 240 bit (piú i separatori fra ogni terna di bit).

Si può fare di meglio, se sfruttiamo il fatto che alcuni caratteri compaiono piú frequentemente di altri. Come primo passo quindi si deve costruire una tabella di frequenza. Supponiamo di ottenere la tabella seguente:

lettera:	a	b	c	d	e	f	g
frequenza:	23	11	12	8	18	5	3

Tabella 4.1: Tabella di frequenza

Usiamo una rappresentazione a lunghezza variabile e rappresentiamo i caratteri piú frequenti con meno bit. Per esempio: a = 0, b = 10, c = 01, d = 000, e = 00, f = 11, g = 001. In questo caso abbiamo bisogno di 148 bit, ai quali però vanno sempre sommati dei separatori tra ogni gruppo di bit, per evitare ambiguità nella decodifica (il codice 01000 rappresenta *abe* oppure *cd* ?). Si può ulteriormente migliorare il risultato eliminando la necessità di separare (o comunque distinguere in qualche modo) ogni gruppo di bit. Lo si fa usando un **codice prefisso**, nel quale ogni codice rappresentante una lettera non è la prima parte (il prefisso) del codice di un'altra lettera. Questo permette di scrivere una unica stringa di bit senza interruzioni, con possibilità di decodificarla senza ambiguità. Ad esempio, nel nostro caso: a = 0, b = 101, c = 100, d = 1101, e = 111, f = 11000, g = 11001. Risultato: 218 bit.

Supponiamo di avere un insieme C di caratteri con relative frequenze. La struttura ideale per allocarli in vista della compressione con l'algoritmo di Huffman è una coda con priorità, che chiamiamo Q, i cui elementi sono nodi di albero binario ordinati in modo crescente. In ogni nodo viene memorizzato un carattere e la relativa frequenza. L'algoritmo costruisce un albero con strategia bottom-up.

Lo pseudo codice è:

HUFFMAN
  $n \leftarrow dim(C)$;
  $Q \leftarrow C$;
  for $i \leftarrow 1$ **to** n-1;
    $z \leftarrow CreaNodo()$;
    $x \leftarrow left[z] \leftarrow Min(Q)$;
    $y \leftarrow right[z] \leftarrow Min(Q)$;
    $Freq(z) \leftarrow Freq(x) + Freq(y)$;

```
    └ Insert(Q,z);
└ return Min(Q);
```

Nelle figure 4.3, 4.4 e 4.5 viene illustrato come lavora l'algoritmo. Ad ogni passo i due nodi con frequenza minore, per esempio nel primo passaggio g ed f, diventano i figli di un nuovo nodo che ha come valore la somma delle frequenze dei figli e la lista viene riordinata. In presenza di più di due nodi con frequenza minima, la scelta è arbitraria ed ininfluente sul risultato. Alla fine si rimane con un unico albero. Assegnando 0 ai rami sinistri ed 1 ai destri, percorrendo l'albero fino ad ogni carattere otteniamo la sua codifica, che si può dimostrare ottimale.

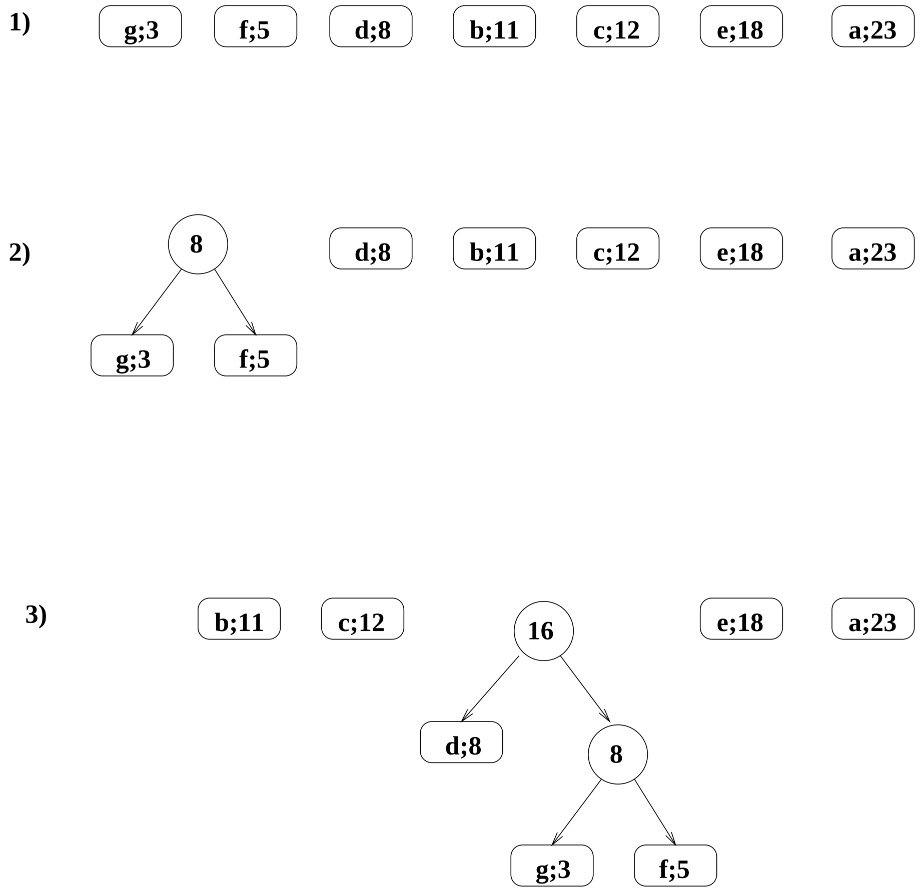


Figura 4.3: Primi 3 passi dell'algoritmo di Huffman.

Nel nostro esempio otteniamo che $a = 10$, $b = 110$, $c = 111$, $d = 010$, $e = 00$, $f = 0111$, $g = 0110$. La stringa (*begafa*) viene quindi codificata come (1100010011010011110).

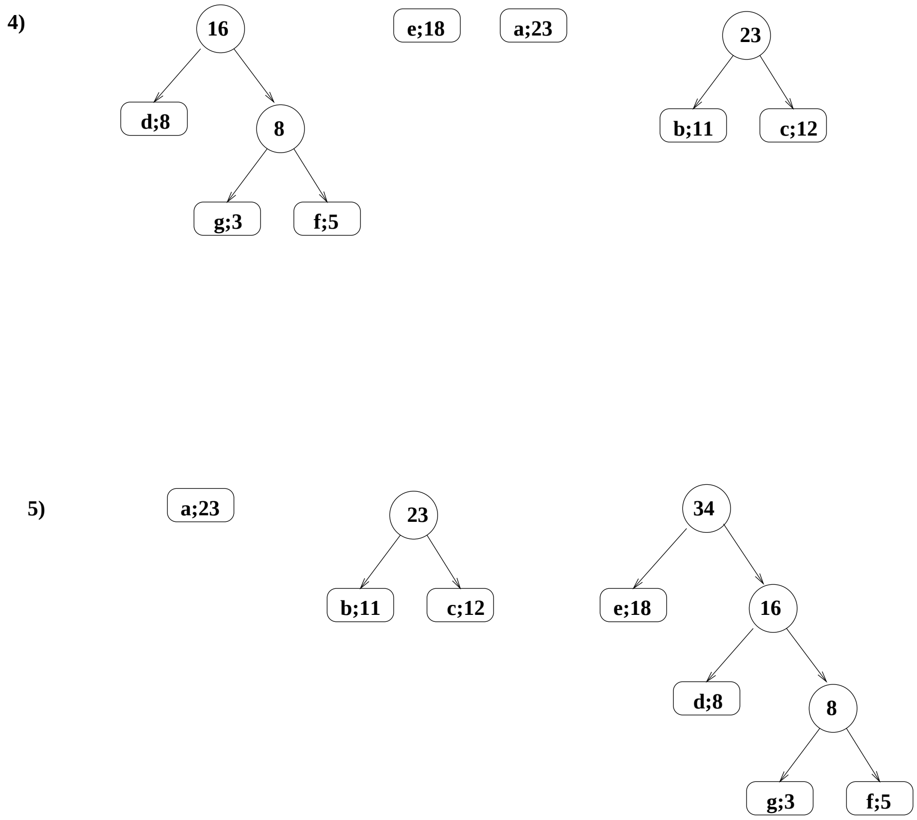
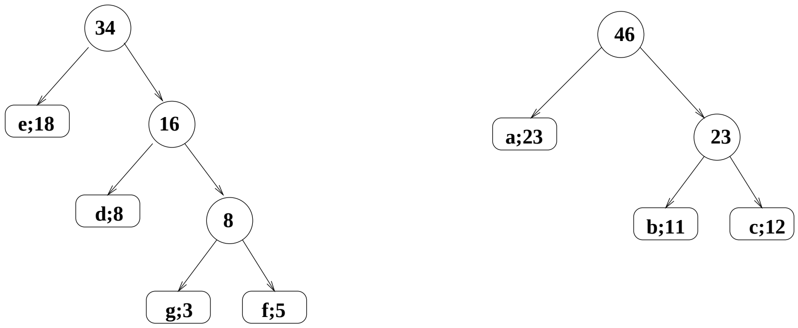


Figura 4.4: Passi 4 e 5 dell'algoritmo di Huffman.

6)



7)

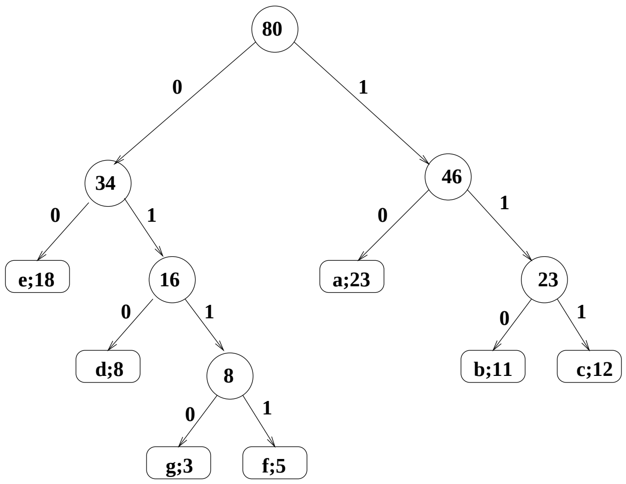


Figura 4.5: Passi 6 e 7 dell'algoritmo di Huffman.