II - STRUTTURE DI DATI

2.0 Strutture di dati

Nel presente capitolo verranno prese in esame alcune strutture di dati in vista di una efficiente implementazione degli algoritmi. In particolare verranno presentate strutture orientate alla rappresentazione ed alla gestione di insiemi, le cui caratteristiche (cardinalità, eventuale ordine interno, ecc.) varino nel corso della esecuzione degli algoritmi.

2.1 Liste

Una *lista* $q = [x_1, x_2,\ldots, x_n]$ è una sequenza di elementi arbitrari (non necessariamente distinti). Gli elementi x_1 e x_n, le *estremità* della lista, sono anche detti rispettivamente la *testa* e la *coda* della lista stessa. Denoteremo con $|q|$ la cardinalità n della lista q. Una *coppia ordinata* $[x_1, x_2]$ è considerabile come una lista di due elementi; il simbolo [ ] denota la lista vuota. Nel seguito indicheremo con $q[i]$ l'elemento i^{esimo} della lista q, cioè x_i, e con $q[i,\ldots, j]$, con $i \le j$, la sottolista $[x_i,\ldots, x_j]$.

Definiamo sulle liste le seguenti operazioni fondamentali:

Concatenazione date le liste $q = [x_1, x_2,\ldots, x_n]$ e $p = [y_1, y_2,\ldots, y_m]$ restituisce la lista $q\&p = [x_1, x_2,\ldots, x_n, y_1, y_2,\ldots, y_m]$;

Accesso

in posizione i	data la lista q restituisce l'elemento $q[i]$;
in testa	data la lista q restituisce l'elemento $q[1]$;
in coda	data la lista q restituisce l'elemento $q[\|q\|]$;

Inserzione

in posizione i	data la lista q e l'elemento x restituisce la nuova lista $q[1,\ldots, i\text{-}1]\&[x]\&q[i,\ldots, \|q\|]$;
in testa	data la lista q e l'elemento x restituisce la nuova lista $[x]\&q$;
in coda	data la lista q e l'elemento x restituisce la nuova lista $q\&[x]$;

Estrazione

dalla posizione i	data la lista q restituisce l'elemento $q[i]$, e sostituisce q con $q[1,\ldots, i\text{-}1]\&q[i\text{+}1,\ldots, \|q\|]$;
dalla testa	data la lista q restituisce l'elemento $q[1]$ e sostituisce q con $q[2,\ldots, \|q\|]$;
dalla coda	data la lista q restituisce l'elemento $q[\|q\|]$ e sostituisce q con $q[1,\ldots, \|q\|\text{-}1]$.

Una lista su cui siano definite le operazioni di accesso ed inserzione in testa e di estrazione dalla testa è detta una *pila*; essa funziona in modo LIFO (*last in first out*). Una lista su cui siano definite le operazioni di accesso in testa, inserzione in coda e di estrazione dalla testa è detta una *fila* (o anche spesso *coda*); una fila funziona in modo FIFO (*first in first out*).

Una lista in cui accessi, inserzioni ed estrazioni sono definite sia in testa che in coda sarà chiamata *doppia fila* (*deque*: double ended queue).

Una lista può essere rappresentata sia per mezzo di un *vettore* che di una *struttura a puntatori*.

Ad esempio una pila q può facilmente essere rappresentata per mezzo del vettore Q, mantenendo nell'intero k l'ultima posizione non vuota nel vettore; per cui la corrispondenza tra pila e vettore è data dalla

$$q[i] = Q[k\text{+}1\text{-}i];$$

se la pila è vuota si pone $k = 0$. Chiaramente, con questa rappresentazione, le operazioni proprie dello stack richiedono tempo $O(1)$.

Una doppia fila q può essere rappresentata per mezzo di un vettore Q, mantenendo in due interi, j e k le due estremità della deque, e consentendo un uso *circolare* del vettore. La corrispondenza tra q e Q è data dalla

$$q[i] = Q[((j+i-2) \bmod n)+1]$$

dove si è supposto Q un vettore ad n componenti; anche in questo caso le operazioni hanno complessità costante.

L'uso di vettori è una scelta ragionevole per rappresentare liste, purché si disponga di una valutazione (per eccesso) sufficientemente accurata del massimo numero di elementi che esse possono contenere, e purché non si debbano effettuare troppe operazioni quali la concatenazione di liste o estrazioni di elementi in posizioni diverse dalle estremità.

Ci sono molti modi per rappresentare una lista per mezzo di strutture a puntatori. Una struttura di questo tipo è detta *endogena* se i puntatori che ne costituiscono l'ossatura sono contenuti negli elementi stessi della lista; ad esempio in figura 2.1. è riportata la lista [5, 7, 2, 3, 1] rappresentata per mezzo di una struttura endogena.

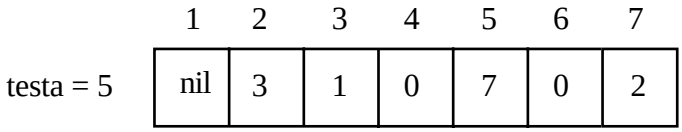


Fig.2.1

Una struttura è di tipo *esogeno* se c'è una distinzione tra la struttura stessa e gli elementi che vi sono rappresentati.

Le liste, sia quelle endogene che quelle esogene, possono essere *lineari* o *circolari*, *singole* o *doppie*. In Fig. 2.2 sono indicate nell'ordine, per il caso di liste endogene, una lista lineare, una circolare, una doppia lineare ed una doppia circolare.

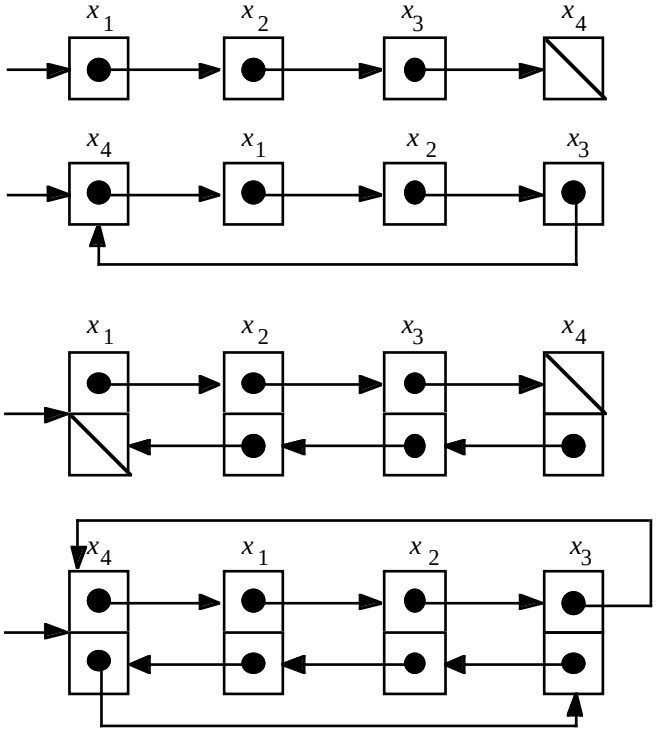


Fig. 2.2

In Fig. 2.3 sono riportate, per il caso di liste esogene, una lista lineare ed una circolare, mentre in Fig. 2.4 sono riportate una lista doppia lineare ed una doppia circolare, sempre di tipo esogeno.

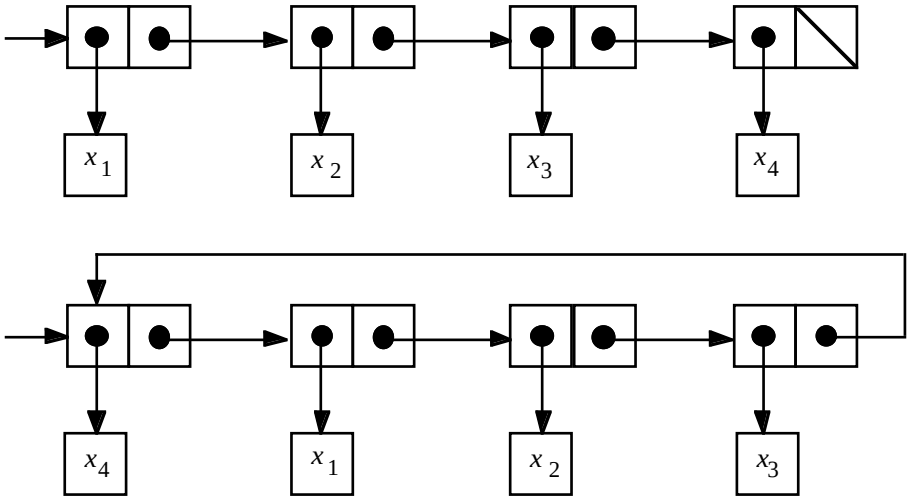


Fig.2.3

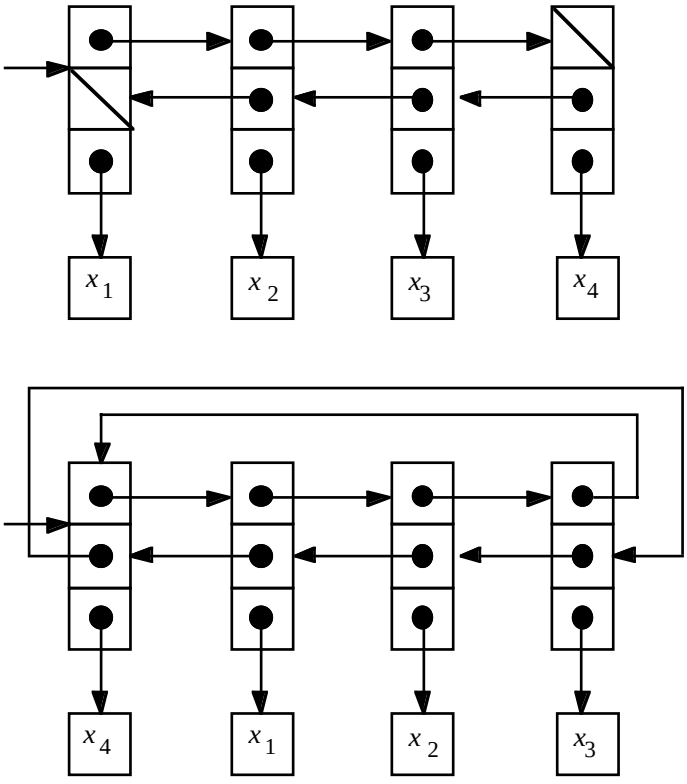
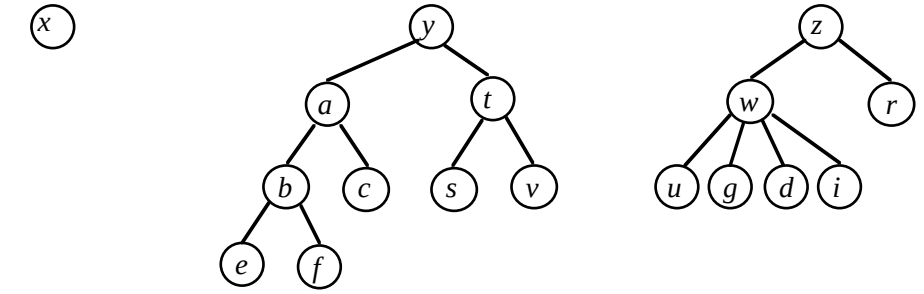


Fig. 2.4

La scelta del particolare tipo di lista da usare sarà guidata dall'analisi delle operazioni che su di essa dovranno essere eseguite. Ad esempio, su liste lineari gli accessi in testa hanno complessità $O(1)$, mentre gli accessi in coda hanno complessità $O(|q|)$. Entrambi i tipi di accesso hanno invece complessità $O(1)$ su liste circolari. In liste semplici, l'inserzione di un nuovo elemento in posizione immediatamente successiva a quella di un prefissato elemento ha complessità $O(1)$, mentre l'inserzione di un nuovo elemento in posizione immediatamente precedente a quella di un prefissato elemento ha complessità $O(|q|)$. Entrambe le operazioni hanno complessità $O(1)$ nel caso di liste doppie.

2.2 Alberi ed insiemi disgiunti

Sia $I = \{1, 2,..., n\}$ un insieme di elementi, e $S_1, S_2,..., S_m$ sottoinsiemi disgiunti di I. Gli insiemi $S_1, S_2,..., S_m$ possono essere rappresentati per mezzo di alberi. Ogni insieme è rappresentato per mezzo di un albero i cui nodi sono gli elementi dell'insieme stesso. Ogni elemento, x, punta al suo predecessore $p(x)$. L'elemento radice dell'albero punta a se stesso; tale elemento viene anche detto *elemento canonico* dell'insieme. Un esempio è illustrato in Fig. 2.5.



$S_1 = \{x\}$ $S_2 = \{a, b, c, e, f, s, t, v, y \}$ $S_3 = \{d, g, i, r, u, w, z \}$

Fig. 2.5

Definiamo le seguenti operazioni:

Makeset(x): costruisce un nuovo insieme x, con x non appartenente a nessuno degli insiemi già esistenti; viene effettuata ponendo $p(x) := x$ e costa $O(1)$.

Find(x): restituisce l'elemento canonico dell'insieme contenente x; comporta il percorrere il cammino da x fino alla radice, e costa $O(n)$.

Union(x,y): costruisce un nuovo insieme unione degli insiemi aventi x ed y come elementi canonici ($x \neq y$) in sostituzione dei due vecchi insiemi (che si assumono disgiunti); infine restituisce l'elemento canonico del nuovo insieme; può essere realizzata ponendo $p(x) := y$ con costo $O(1)$.

L'operazione **Union** è descritta in Fig. 2.6.

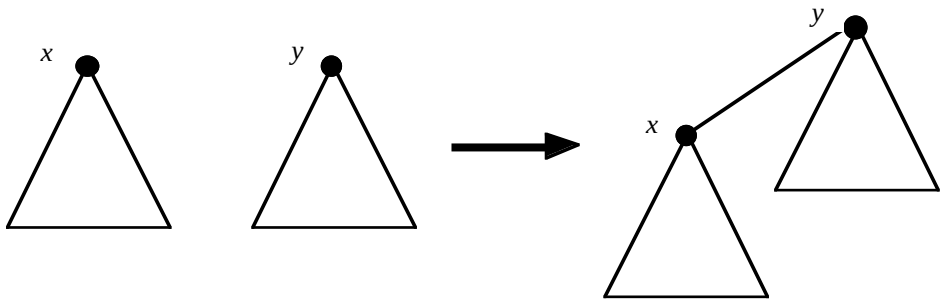


Fig. 2.6

Sia T un albero con radice x, e v un suo nodo. Definiamo:

altezza(v) = { 0,,se v è una foglia,,max{altezza(w): $p(w)=v$}+1,,altrimenti

altezza(x) ≡ altezza dell'albero T;

rango(v) ($\geq$ altezza(v)) ≡ una valutazione per eccesso di altezza(v).

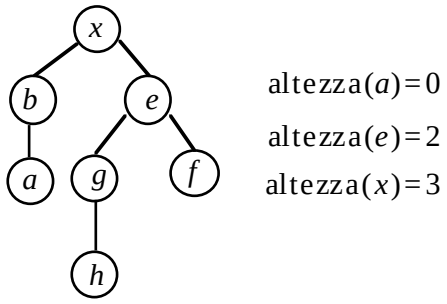


Fig. 2.7

Per rendere basso il costo dell'operazione **Find** bisogna operare in modo da ridurre l'altezza degli alberi. Un primo risultato in questa direzione può essere ottenuto effettuando l'unione tra due insiemi in modo che sia l'albero meno alto ad essere collegato a quello più alto e non viceversa. Questa tecnica è illustrata nella Fig. 2.8.

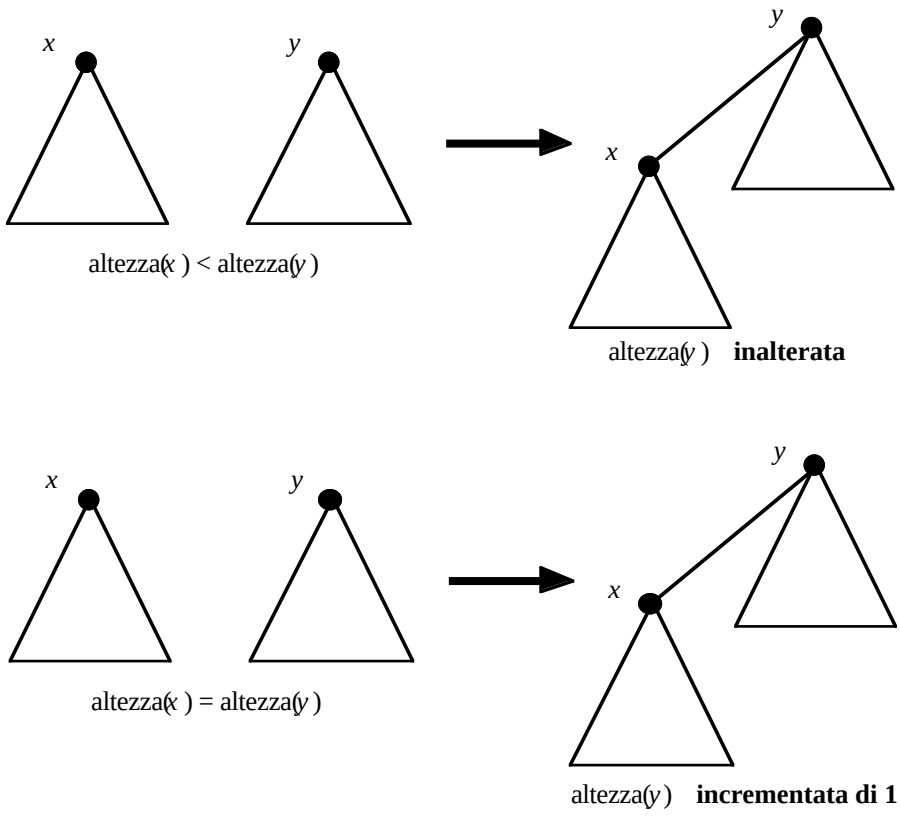


Fig. 2.8

Naturalmente l'uso dell'altezza come criterio per l'operazione di unione non evita la creazione di alberi molto alti se è grande il numero di operazioni da effettuarsi. Una tecnica che consente di limitare consistentemente l'altezza degli alberi è quella del *compattamento lungo cammini* (*path compression*), che viene effettuata nella **Find**. L'idea è quella di ristrutturare l'albero ad ogni chiamata di **Find** in modo da ridurne l'altezza.

Quest'idea è illustrata nella Fig. 2.9, dove è considerato il caso di una chiamata di **Find** applicata all'elemento *a*.

La tecnica di compattamento lungo i cammini rende costoso l'uso dell'altezza per le operazioni di unione. Bisognerebbe dopo ogni chiamata di **Find** effettuare un aggiornamento dell'altezza, operazione costosa perché implica la visita dell'albero. Si preferisce allora usare invece dell'altezza la funzione *rango*.

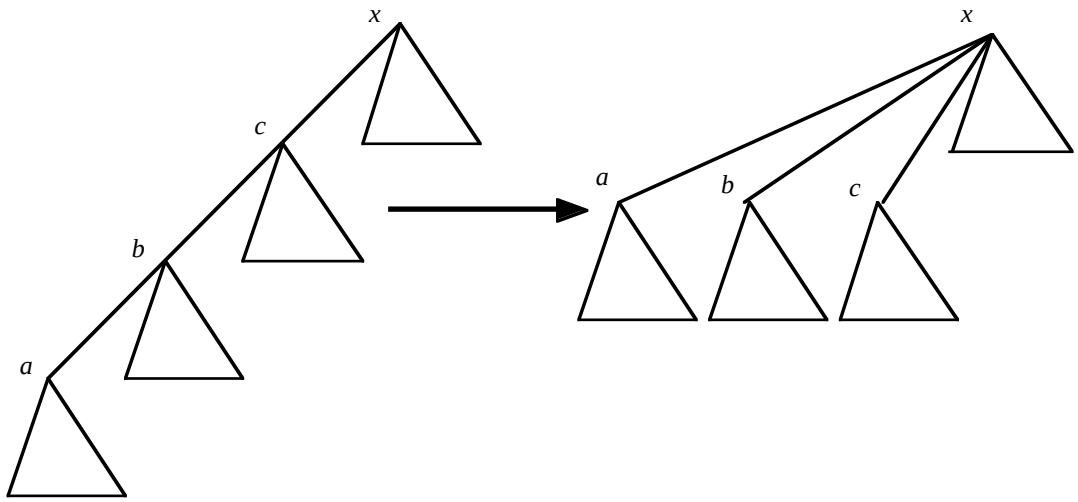


Fig. 2.9

Si dice rango(x) una valutazione per eccesso dell'altezza di x, ottenuta ponendo rango(x) = altezza(x) = 0 all'inizio quando gli insiemi sono formati da elementi singoli; quindi successivamente ad ogni operazione di unione il rango viene aggiornato esattamente come si faceva con l'altezza. Praticamente l'effetto è quello di trascurare il compattamento delle operazioni **Find**.

Esaminiamo ora gli effetti in termini di complessità computazionale di questo modo di realizzare l'unione tra insiemi. Utilizzando il rango, è possibile valutare la complessità nel caso peggiore di ciascuna operazione **Find**, come si ricava dalle seguenti proposizioni.

Proposizione 2.1 - Il numero di nodi nell'albero di radice x è almeno $2^{\text{rango}(x)}$.

Dimostrazione: La prova è per induzione sul numero delle operazioni di unione. La tesi è certamente vera dopo la prima operazione di unione. Assumiamo che sia ancora vera dopo la k^{esima}, e si effettui la $(k+1)^{\text{esima}}$ unione sugli alberi di radice x e y. Se rango(x) ≠ rango(y), la proprietà continua ad essere vera dopo l'operazione. Se rango(x) = rango(y), il nuovo albero avrà rango pari a rango(x)+1, e conterrà almeno $2^{\text{rango}(x)} + 2^{\text{rango}(y)} = 2^{\text{rango}(y)+1}$ nodi, e quindi la proprietà continua ad essere vera. La tesi è così dimostrata. ◊

Proposizione 2.2 - La complessità di una operazione **Find** è O(log$n$).

Dimostrazione: Il costo computazionale di una operazione **Find(x)** è dato dalla lunghezza del cammino dal nodo x alla radice r dell'albero a cui x appartiene, e dunque è limitato da altezza(r). Dalla proposizione 2.1 risulta evidente che ogni nodo ha rango al più $\lfloor \log n \rfloor$, e quindi per ogni nodo radice r si ha:

$$\text{altezza}(r) \leq \text{rango}(r) \leq \log n \ ;$$

dunque nel caso pessimo l'operazione **Find** avrà complessità O(log$n$).◊

Analizziamo ora una sequenza di n operazioni **Union**, n operazioni **Makeset** ed m operazioni **Find**. Ricordando che la complessità di **Union** e **Makeset** è O(1), si ha una complessità

$$O(n + m\log n),$$

e nel caso in cui è $m = O(n)$ la complessità diventa $O(n \log n)$.

Si può tuttavia dimostrare la seguente:

Osservazione - Una sequenza di n **Makeset**, $m \geq n$ **Find**, e n-1 **Union**, ha una complessità $O(m\alpha(m,n))$ $[\Theta(m\ \alpha(m,n))]$, con $\alpha(m,n)$ inversa della funzione di Ackerman [per $n < 2^{16}$ è $\alpha(m,n) \leq 3$; agli effetti pratici possiamo assumere $\alpha(m,n)$ una costante ≤ 4].

Nel seguito viene fornita una descrizione formale delle procedure che consentono di implementare le operazioni definite sopra.

```
Procedure MAKESET(i):
   begin
      p[i] := i; rango[i] := 0
   end.

Function FIND(i):
   begin
      if i ≠ p[i] then p[i] := FIND(p[i]);
      return i
   end.

Procedure UNION(x,y):
   begin
      if rango[x] > rango[y]
         then begin w := y; y := x; x := w end
         else if rango[x] = rango[y] then rango[y] := rango[y]+1;
      p[x] := y
   end.
```

2.3 Heaps

Una coda di priorità è una struttura di dati astratta consistente di un insieme finito di elementi ciascuno con associato un valore reale, detto *chiave*, val(i), $i = 1, 2, \ldots, n$.

Operazioni principali:

Insert(i,Q): inserisce l'elemento i nella coda di priorità Q non contenente i;

Deletemin(Q): restituisce e cancella l'elemento di valore minimo fra quelli in Q (se Q è vuoto restituisce *null*);

Make(Q,S): costruisce e restituisce una nuova coda di priorità Q contenente gli elementi dell'insieme S;

Ulteriori operazioni:

Findmin(Q): restituisce senza cancellarlo l'elemento di valore minimo fra quelli in Q (se Q è vuoto restituisce *null*);

Delete(i,Q): cancella l'elemento i da Q;

Link(Q_1,Q_2): restituisce la coda di priorità ottenuta combinando insieme Q_1 e Q_2, che vengono distrutte;

Change(i,v): cambia il valore dell'elemento i nella coda di priorità ponendo val(i)=v.

Un *heap* è una coda di priorità organizzata ad albero, in cui:

- l'elemento di valore minimo si trova nella radice dell'albero;
- se x e $p(x)$ sono un nodo ed il suo predecessore, allora $\text{val}(x) \geq \text{val}(p(x))$.

Un esempio di heap è riportato in Fig. 2.11 in cui ai nodi dell'albero sono associati direttamente i valori.

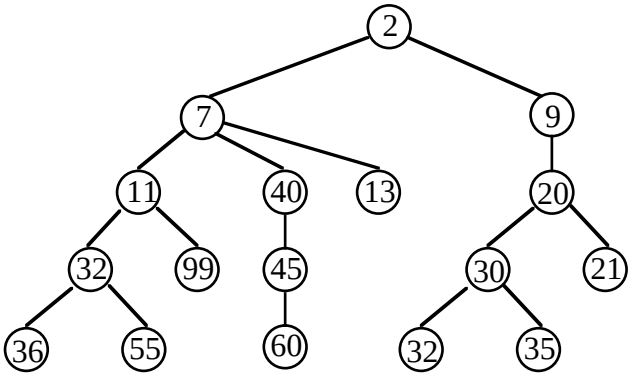


Fig.2.11

2.3.1 d-heaps

Un *d*-heap è un heap caratterizzato dalle seguenti proprietà:

- ogni nodo ha al più *d* figli (un caso particolarmente interessante è lo heap binario);
- l'albero è bilanciato, ovvero i nodi foglia hanno profondità compresa tra l e l-1 (la *profondità* o *livello* di un nodo è il numero di archi nel cammino che lo unisce alla radice). Il livello massimo l è compreso tra $\lfloor \log_d n \rfloor$ e $(\lfloor \log_d n \rfloor + 1)$;
- i nodi dell'ultimo livello sono addensati a sinistra.

Si noti che un *d*-heap contenente n elementi ha un nodo a livello 0 (la radice), d nodi a livello 1, d^2 nodi a livello 2, sino a d^{l-1} al penultimo livello e non più di d^l nodi nell'ultimo livello l. Pertanto, il numero di nodi dal livello 0 al livello l-1 è $n' = (d^l-1)/(d-1)$; e il numero di nodi $n \leq (d^{l+1}-1)/(d-1)$. Essendo $n' < n$, si ha:

$$d^l - 1 < n(d-1) \leq d^{l+1} - 1,$$

e quindi

$$l < \log_d (n(d-1)+1) \leq l+1.$$

Da cui si ottiene:

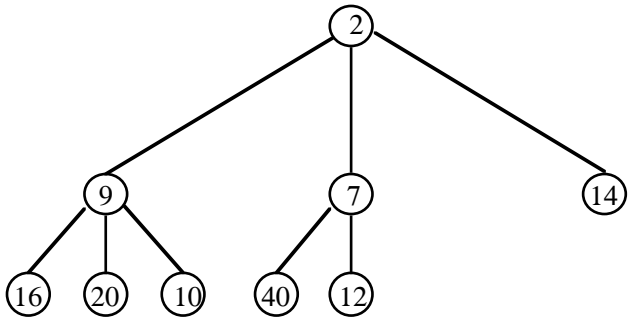
$$l = \lceil \log_d (n(d-1)+1) \rceil - 1.$$

Il numero di nodi (foglie) nell'ultimo livello l è dato da $n - n' = n - (d^l-1)/(d-1)$. Essendo addensati a sinistra, essi saranno figli dei primi $n'' = \lceil (n - n')/d \rceil$ nodi del livello precedente. Quindi, il numero di foglie del *d*-heap è dato da $nf = n - n' + d^{l-1} - n''$.

Un *d*-heap con n nodi può essere facilmente implementato per mezzo di un vettore con n componenti, $(x_1, x_2, ..., x_n)$, come segue:

- i nodi sono numerati da 1 ad n;
- 1 è la radice;
- il nodo i ha come figli i nodi da $d(i-1)+2$ a $\min\{di+1, n\}$;
- il predecessore del nodo i è il nodo $\lceil (i-1)/d \rceil$;
- x_i è l'elemento contenuto nel nodo i.

In Fig. 2.12 è riportato un esempio di 3-heap con il vettore che lo implementa.



[2, 9, 7, 14, 16, 20, 10, 40, 12]

Fig. 2.12

Esercizio: Calcolare per il 3-heap in figura i valori l, n', n'' e nf.

Nel seguito descriviamo in dettaglio l'implementazione delle diverse operazioni nel caso in cui il d-heap sia stato implementato per mezzo di un vettore H contenente $|H| = h$ elementi. Associato al vettore H viene utilizzato un *dizionario* che fornisce la posizione da esso occupata da ogni elemento nel d-heap H; nel seguito indicheremo la posizione dell'elemento x mediante $H^{-1}[x]$.

Insert(x,H): inserisce in ultima posizione (posizione h+1 del vettore) dello heap H l'elemento x con chiave val(x); quindi x viene fatto risalire fino a trovare la posizione corretta:

```
Procedure INSERT(x,H):
   begin
      h := h+1; u := h; MOVEUP(x,u,H)
   end.

Procedure MOVEUP(x,u,H):
   begin
      p := ⌈(u-1)/d⌉;
      while p ≠ 0 and val[H[p]] > val[x] do
         begin H[u] := H[p]; u := p; p := ⌈(u-1)/d⌉ end;
      H[u] := x
   end.
```

La complessità dell'operazione di **Insert** dipende dall'altezza dell'albero; essa è $O(k)$ dove k è l'altezza dell'albero. Con una implementazione di H come vettore ad n componenti che rappresenta un albero bilanciato, è $k = O(\log_d n)$, e pertanto la complessità di **Insert** è $O(\log_d n)$.

Delete(x,H): Cancella l'elemento x dalla sua posizione corrente $H^{-1}[x]$, e lo sostituisce con l'elemento in ultima posizione; questo viene fatto salire verso la radice o scendere, a seconda del suo valore.

```
Procedure DELETE(x,H):
   begin
      u := H^-1[x]; y := H[h]; H[h] := null; h := h-1;
      if x ≠ y then
         if val[x] ≥ val[y] then MOVEUP(y,u,H)
                            else MOVEDOWN(y,u,H)
   end.
```

```
Procedure MOVEDOWN(x,u,H):
   begin
      c := MINCHILD(u,H);
      while c ≠ 0 and val[H[c]] < val[x] do
         begin H[u] := H[c]; u := c; c := MINCHILD(u,H) end;
      H[u] := x
   end.

Function MINCHILD(u,H):
   begin
      if d(u-1) + 2 > h
         then return 0
         else return argmin{val[H[i]]: i = d(u-1)+2,..., min{du+1,h}}
   end.
```

La complessità di **Delete** è $O(d \log_d n)$; infatti il numero di passi è limitato dall'altezza dell'albero e, ad ogni passo, nel caso di chiamata di **Movedown**, devono essere esaminati tutti i *d* figli del nodo corrente mediante la **Minchild**.

Findmin(*H*): restituisce la radice dell'albero.

```
Function FINDMIN(H):
   begin
      if h = 0 then return null
               else return H[1]
   end.
```

La complessità di **Findmin** è $O(1)$.

Deletemin(*H*): Opera utilizzando successivamente la funzione **Findmin** e la procedure **Delete**:

```
Function DELETEMIN(H):
   begin
      x := FINDMIN(H);
      if x ≠ null then DELETE(x,H);
      return x
   end.
```

La funzione **Deletemin** ha la stessa complessità della procedura **Delete**.

Change(*i*,*v*): cambia il valore dell'elemento *i* nella coda di priorità ponendo val(*i*)=*v*.

```
Procedure CHANGE(x,v,H):
   begin
      u := H^-1[x];
      if v < val[x] then begin val[x] := v; MOVEUP(x,u,H) end
                    else begin val[x] := v; MOVEDOWN(x,u,H) end
   end.
```

La funzione **Change** ha la stessa complessità della procedura **Delete**.

MakeHeap(*S*): Costruisce un *d*-heap contenente gli elementi di *S*, con $|S| = n$. Può essere effettuata inizializzando un *d*-heap vuoto e quindi effettuando *n*

inserzioni, con una complessità $O(n \log_d n)$.

Un modo più efficiente di costruire il *d*-heap è quello di costruire prima un albero *d*-ario con gli elementi in un ordine arbitrario; quindi si effettuano, per tutti gli elementi associati a nodi non foglia iniziando da quelli al livello *l*-1. Si opereranno perciò $n - nf$ operazioni **Movedown**$(H(p),p,H)$, con $p = n\text{-}nf, n\text{-}nf\text{-}1,\ldots, 2, 1$. Nel caso peggiore, per ciascun nodo a livello *l*-1 l'operazione **Movedown** avrà complessità $O(d)$ in quanto tutti i figli sono delle foglie. In generale per ciascun nodo a livello *l-i* la complessità di **Movedown** è $O(di)$; poiché vi sono al più n/d^i nodi a livello *l-i* in un albero *d*-ario completo la complessità globale, calcolata per livelli, è:

$$\sum_{i=1}^{l-1} \frac{n\ d\ i}{d^i} = \sum_{i=0}^{l-2} \frac{n\ (i+1)}{d^i} \le \sum_{i=0}^{\infty} \frac{n\ (i+1)}{d^i} = O(n);$$

l'ultima uguaglianza deriva dal fatto che la serie

$$\sum_{i=0}^{\infty} \frac{i+1}{d^i}$$

è convergente. È infatti facile verificare che, essendo $d \ge 2$, si ha:

$$\sum_{i=0}^{\infty} \frac{i+1}{d^i} = d \sum_{i=0}^{\infty} \frac{i+1}{d^{i+1}} \le d \sum_{i=0}^{\infty} \frac{i}{2^i}$$

e

$$\sum_{i=0}^{\infty} \frac{i}{2^i} \le \sum_{i=0}^{\infty} \frac{2i}{2^{2i}} + \sum_{i=0}^{\infty} \frac{2i+1}{2^{2i+1}} \le \sum_{i=0}^{\infty} \frac{1}{2^i} + \sum_{i=0}^{\infty} \frac{1}{2^i} = 4.$$

Riferimenti bibliografici

* Per approfondimenti si rimanda a (Tarjan 1983), capitoli 1, 2 e 3.

Tarjan, R. E. (1983). *Data structures and network algorithms*. SIAM.