

GRAFI

Sono utilizzati per rappresentare insiemi di oggetti e le relazioni tra esse.

GRAFI ORIENTATI

Un grafo orientato (o direct graph) è una coppia $G = (V, E)$, con V insieme finito di elementi detti nodi (o vertex), ed E insieme finito di coppie ordinate di nodi detti archi (o Edges).

Il numero di nodi si indica con $|V| = n$.

Il numero di vertici si indica con $|E| = m$.

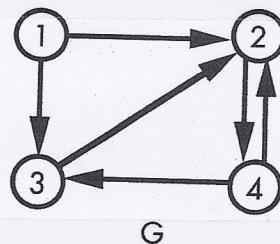
Un nodo v si dice adiacente a w se esiste un arco $(v, w) \in E$ (arco incidente).

Il grado entrante/uscente di un nodo è il numero di archi entranti/uscenti.

ESEMPIO: $G = (V, E)$, $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (1, 3), (2, 3), (3, 2), (4, 2), (4, 3)\}$

$$|V| = 4$$

$$|E| = 6$$



Cammino: è una sequenza di nodi v_0, \dots, v_k t.c. $(v_i, v_{i+1}) \in E$ per $i = 0, \dots, k-1$. Il cammino ha lunghezza uguale al numero di archi attraversati.

Cammino semplice: se non ci sono nodi ripetuti, cioè $v_i \neq v_j \forall 0 \leq i < j \leq k$.

Cammino chiuso: se $v_0 = v_k$

Cammino ciclico: se semplice e chiuso.

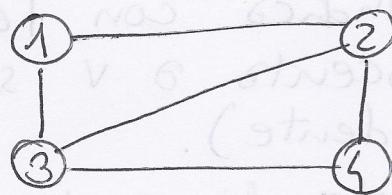
GRAFI NON ORIENTATI

Se gli archi di un grafo G non formano coppie non ordinate $[u,v]$, allora G è un grafo non orientato.

Grafo non orientato: (u,v) e (v,u) due archi distinti.

Grafo non orientato: $[u,v]$ e $[v,u]$ stesso arco.

ESEMPIO: $G = (V, E)$, $V = \{1, 2, 3, 4\}$, $E = \{\overrightarrow{1,2}, \overrightarrow{1,3}, \overrightarrow{2,4}, \overrightarrow{3,2}, \overrightarrow{4,3}\}$



Catena: sequenza di nodi v_0, \dots, v_k t.c. $[v_i, v_j] \in E$ per $i=0, \dots, k-1$.

Lunghezza della catena: numero di archi attraversati (k).

Catena semplice: sequenza senza nodi ripetuti.

Catena chiusa: sequenza con $v_0 = v_k$

Circuito: se sequenza è semplice e chiusa.

ESEMPIO: (con grafo sopra)

sequenza $1-2-4-2-3-1 \Rightarrow$ catena chiusa

sequenza $2-4-3-2 \Rightarrow$ circuito

sequenza $1-2-1 \Rightarrow$ catena chiusa

(non è un circuito perché l'arco $\overrightarrow{1,2}/\overrightarrow{2,1}$ è ripetuto)

CONNESSIONE E CONNESSIONE FORTE

- Sia $G = (V, E)$ un grafo orientato.
 G è fortemente connesso se per ogni coppia di nodi $u, v \in V$ esiste almeno un cammino da u a v e viceversa.
- Un grafo $G' = (V', E')$ è una componente fortemente connessa di G se è un sottografo di G fortemente connesso e massimale.
- Sia $G = (V, E)$ un grafo non orientato.
 G è connesso se per ogni coppia di nodi distinti $u, v \in V$ esiste una catena tra u e v .
Un grafo $G' = (V', E')$ è una componente connessa di G se è un sottografo di G connesso e massimale.

G' è sottografo massimale di G se:

- G' è sottografo di G ($G' \subseteq G$) se $V' \subseteq V$ e $E' \subseteq E$
- G' è massimale se non è possibile trovare un sottografo G'' di G che sia fortemente connesso/connesso e più grande di G' , in altri termini $G' \subseteq G'' \subseteq G$.

Oss. $G' = G$ è sottografo massimale

SPECIFICA

Sarà fornita solo la specifica per grafi orientati, chiamati solo grafi, in quanto un grafo non orientato G può essere visto come un grafo orientato G' , ottenuto da G sostituendo ogni arco $[u,v]$ con due archi (u,v) e (v,u) .

GRAPH	
Graph()	% Crea un grafo vuoto
insertNode(NODE u)	% Aggiunge il nodo u al grafo
insertEdge(NODE u , NODE v)	% Aggiunge l'arco (u, v) al grafo
deleteNode(NODE u)	% Rimuove il nodo u dal grafo
deleteEdge(NODE u , NODE v)	% Rimuove l'arco (u, v) nel grafo
SET adj(NODE u)	% Restituisce l'insieme dei nodi adiacenti ad u
SET V()	% Restituisce l'insieme di tutti i nodi

Supponiamo che l'insieme statico dei nodi si possa rappresentare con i numeri $1, \dots, n$.

```

foreach  $u \in G.V()$            // Scorre tutti i nodi  $u$  del grafo
    foreach  $v \in G.adj(u)$    // Scorre tutti i nodi adiacenti ad  $u$ 
        [ fai operazione su arco  $(u, v)$  ]
    ]
}

```

Abbreviazione: $G.n = G.V().size()$

REALIZZAZIONI CON MATERICI

Dato un grafo $G = (V, E)$, la matrice di adiacenza (nodi-nodi) $M = [m_{uv}]$ è una matrice di dimensione $n \times n$ t.c.

$$m_{uv} = \begin{cases} 1 & \text{se } (u, v) \in E \\ 0 & \text{se } (u, v) \notin E \end{cases}$$

Se il grafo non è orientato, M è simmetrica.

Se il grafo è pesato, si utilizzano pesi sugli archi al posto degli elementi binari. Se p_{uv} è il peso dell'arco (u, v) allora M diventa:

$$m_{uv} = \begin{cases} p_{uv} & \text{se } (u, v) \in E \\ +\infty/\infty & \text{se } (u, v) \notin E \end{cases}$$

Verifica se arco è presente: $O(1)$

Ricavare $G.\text{adj}(v)$: $O(n)$

Spatio di memoria: $O(n^2)$

Esaminare tutti gli archi: $O(n^2)$

Un grafo $G = (V, E)$ può essere rappresentato come una matrice di incidente (nodi - archi) di dimensione $n \times m$, dove ciascuna riga rappresenta il nodo e ciascuna colonna rappresenta l'arco.

$$b_{uv} = \begin{cases} -1 & \text{se l'arco } k\text{-esimo esce dal nodo } u \\ +1 & \text{se l'arco } k\text{-esimo entra nel nodo } v \\ 0 & \text{altrimenti} \end{cases}$$

Se il grafo non è orientato:

$$b_{uv} = \begin{cases} 1 & \text{se l'arco } k\text{-esimo è incidente in } u \\ 0 & \text{altrimenti} \end{cases}$$

Spazio di memoria: $\Theta(nm)$

Ricavare $G.\text{adj}(v)$: $\Theta(nm)$

ESPLORAZIONE DI UN GRAFO

Una visita di un grafo è una procedura per esplorare almeno una volta ogni suo nodo e arco. Genericamente una visita di un grafo può essere descritta:

visita(Graph G, NODE r)

```
SET S ← Set()
S.insert(r)
{marca il nodo r come "scoperto"}
while S.size() > 0 do
    NODE u ← S.remove()
    {esamina il nodo u}
    foreach v ∈ G.adj(u) do
        {esamina l'arco (u, v)}
        if v non è già stato scoperto then
            {marca il nodo v come "scoperto"}
            S.insert(v)
```

L'insieme S contiene nodi che sono stati scoperti ma non ancora visitati. Con l'if ogni nodo è inserito in S una sola volta.

Nel caso di grafi orientati e fortemente connessi, o non orientati e connessi, la visita parte da un generico nodo e raggiunge ogni altro nodo in una sola passata; in caso contrario è necessario fare più passate. L'algoritmo è ottimo e richiede $\Theta(n+m)$.

L'ordine in cui vengono visitati i nodi dipende dalla politica di estrazione della procedura remove(). Esistono due tipi di approcci:

- Breadth-First-Search (BFS) : visita in ampiezza
- Depth-First-Search (DFS) : visita in profondità

VISITA BFS

Nella visita BFS i nodi sono visitati in ordine di distanza crescente dal nodo di partenza r , dove la distanza da r ad un generico nodo v è il minimo numero di archi di un cammino da r a v . Questo metodo di visita è un'estensione della visita per livelli di un albero radicato in r . Nell'algoritmo l'insieme S è una coda FIFO e il processo di marcatura dei nodi già visitati è realizzato con un vettore di booleani.

bfs(GRAPH G , NODE r)

```
QUEUE  $S \leftarrow$  Queue()
 $S.\text{enqueue}(r)$ 
boolean[] visitato  $\leftarrow$  new boolean[1 ...  $G.n$ ]
foreach  $u \in G.V() - \{r\}$  do  $\text{visitato}[u] \leftarrow \text{false}$ 
 $\text{visitato}[r] \leftarrow \text{true}$ 
while not  $S.\text{isEmpty}()$  do
    NODE  $u \leftarrow S.\text{dequeue}()$ 
    {esamina il nodo  $u$ }
    foreach  $v \in G.\text{adj}(u)$  do
        {esamina l'arco  $(u, v)$ }
        if not  $\text{visitato}[v]$  then
             $\text{visitato}[v] \leftarrow \text{true}$ 
             $S.\text{enqueue}(v)$ 
```

ALBERI DI COPERTURA BFS

L'albero dei cammini BFS è un albero di copertura del grafo G radicato r . In questo albero, i figli di un nodo v sono tutti i nodi di v scoperti durante la visita dell'arco (u, v) .

VISITA DFS

Il metodo di visita DFS è una diretta estensione della visita in ordine anticipato di un albero binario. Quando viene visitato un nuovo nodo u , ci si allontana il più possibile lungo il cammino (o catena), fino a raggiungere un nodo da cui non è possibile visitare nodi non ancora visitati.

L'algoritmo viene chiamato sul grafo G a partire da un nodo r , mentre "visitato" è il vettore di booleani che rappresenta i nodi già scoperti.

```
dfs(GRAPH G, NODE u, boolean[] visitato)
    visitato[u] ← true
    (1) {esamina il nodo  $u$  (caso previsita)}
        foreach  $v \in G.\text{adj}(u)$  do
            {esamina l'arco  $(u, v)$ }
            if not visitato[v] ← then
                visitato[v] ← true
                | dfs(G, v, visitato)
    (2) {esamina il nodo  $u$  (caso postvisita)}
```

APPLICATIONS DFS: COMPONENTI CONNESSE

Modificando la DFS è possibile risolvere in tempo $\Theta(nm)$ molti problemi fondamentali relativi alla connessione dei grafici:

- (1) verificare se un grafo non orientato G è连通的 oppure no.
- (2) trovare le componenti connesse di cui G è composto.

Le soluzioni sono:

- (1) un grafo è connesso se al termine della visita DFS, tutti i nodi sono stati marcati.
- (2) in questo caso si deve far partire la visita del grafo da tutti i nodi segnati come non marcati.

```
integer [] cc(GRAPH G, NODE[] ordine)
```

```
integer[] id ← new integer[1 ... G.n]
foreach  $u \in G.V()$  do  $id[u] \leftarrow 0$ 
integer conta ← 0
for integer  $i \leftarrow 1$  to  $n$  do
    if  $id[ordine[i]] = 0$  then
        conta ← conta + 1
        ccdfs( $G$ , conta,  $ordine[i]$ ,  $id$ )
return  $id$ 

ccdfs(GRAPH  $G$ , integer conta, NODE  $u$ , integer[]  $id$ )
 $id[u] \leftarrow conta$ 
foreach  $v \in G.adj(u)$  do
    if  $id[v] = 0$  then
        ccdfs( $G$ , conta,  $v$ ,  $id$ )
```

id contiene per ogni elemento u un intero che identifica la componente a cui appartiene.

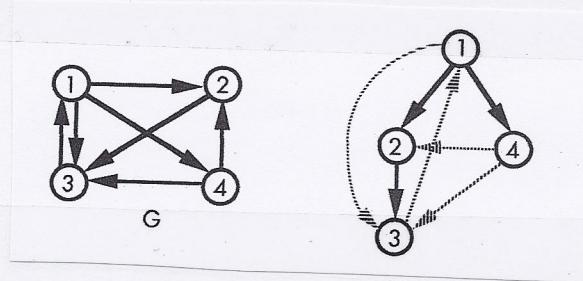
Ogni volta che viene trovato un nodo non ancora scoperto ($id[u]=0$) viene incrementato un contatore e si inizia una DFS da quel nodo.

Il vettore "ordine" contiene una permutazione di nodi di G che rappresenta l'ordine in cui far partire la visita sui nodi di G .

ALBERI DI COPERTURA DFS

È possibile definire l'albero dei cammini T generato dalla visita DFS. Tutte le volte che viene incontrato un arco che connette un nodo marcato da uno non marcato, esso viene inserito nell'albero T . Se non fosse possibile raggiungere tutti i nodi, allora la visita forma una foresta di copertura DFS. Gli archi non inclusi in T possono essere divisi:

- (1) se l'arco è esaminato passando da un nodo di T ad un altro nodo che è suo antenato in T , è detto arco all'indietro.
- (2) se l'arco è esaminato passando da un nodo T ad un suo discendente (che non sia figlio) in T , è detto arco in avanti.
- (3) altivamente è detto arco di attraversamento.



Questa distinzione è importante perché definisce uno schema generale di visita: dfs-schema.

Nello schema si fa uso di due vettori dt e ft (rispettivamente discovery time e finish time) di dimensione n e di un intero time con funzione di orologio discreto.

Durante la visita nelle posizioni $dt[u]$ e $ft[u]$ sono registrati i valori dell'orologio, rispettivamente, all'inizio e alla fine della procedura di visita di u . Confrontando i valori di $dt[u]$, $dt[v]$ e $ft[v]$ è possibile stabilire il tipo di arco (u,v) .

Se $dt[u] = 0$ allora la visita del nodo v non è ancora iniziata, altrimenti se $dt[u] > dt[v]$ e $ft[v] = 0$ la visita di u non è iniziata ma non terminata dopo quella di v , quindi (u,v) è all'indietro; quando invece $dt[u] < dt[v]$ e $ft[v] \neq 0$ la visita di v è sia iniziata che finita dopo quella di u , che però non è ancora terminata, e quindi (u,v) è in avanti; nei rimanenti casi è di attraversamento.

dfs-schema(GRAPH G, NODE u)	
esamina il nodo u prima (caso pre-visita)	
$time \leftarrow time + 1$; $dt[u] \leftarrow time$	
foreach $v \in G.\text{adj}(u)$ do	
esamina l'arco (u, v) di qualsiasi tipo	
if $dt[v] = 0$ then	
esamina l'arco (u, v) in T	
dfs-schema(G, v)	
else if $dt[u] > dt[v]$ and $ft[v] = 0$ then	
esamina l'arco (u, v) all'indietro	
else if $dt[u] < dt[v]$ and $ft[v] \neq 0$ then	
esamina l'arco (u, v) in avanti	
else	
esamina l'arco (u, v) di attraversamento	
esamina il nodo u dopo (caso post-visita)	
$time \leftarrow time + 1$; $ft[u] \leftarrow time$	

APPLICATION = DFS-SCHEMA

Si consideri il problema di verificare se un grafo $G = (V, E)$ è aciclico.

Un grafo è aciclico se non esistono archi all'indietro. Infatti se esiste un arco all'indietro (v, v) , dove v è un antenato di v , allora esiste un cammino da $v \rightarrow v$ e un arco da $v \rightarrow v$, ovvero un ciclo.

Adattando opportunamente l'DFS schema, si ottiene l'algoritmo `ciclico()` che restituisce true se il grafo contiene un ciclo.

```
boolean ciclico(GRAPH G, NODE u)
    time ← time + 1; dt[u] ← time
    foreach v ∈ G.adj(u) do
        if dt[v] = 0 then
            | if ciclico(G, v) then return true
            | else if dt[u] > dt[v] and ft[v] = 0 then
                | return true
        time ← time + 1; ft[u] ← time
    return false
```

APPLICAZIONE DFS-SCHEMA: COMPONENTI FONTEMENTE CONNESE

Un algoritmo del '78 attribuito a Kosaraju funziona nel seguente modo:

- (1) esegue una visita DFS sul grafo G
- (2) calcola il grafo trasposto G^T di G
- (3) esegue l'algoritmo $scc()$ per ottenere le componenti del grafo G^T , ma nel ciclo principale seleziona i nodi in ordine inverso di tempo di fine rispetto la prima visita.

Ciascuno dei tre passaggi richiede tempo $O(n+m)$. Nell'algoritmo $scc()$ i tempi di inizio e di fine non vengono calcolati esplicitamente, al loro posto vengono inseriti in una pila al termine di ogni visita.

```
integer[] scc(GRAPH G)
```

```
STACK ← Stack()
boolean[] visitato ← new boolean[1 ... G.n]
foreach  $u \in G.V()$  do visitato[ $u$ ] ← false
foreach  $u \in G.V()$  do dfsStack( $G$ , visitato,  $S$ ,  $u$ )

GRAPH  $G^T$  ← Graph()
foreach  $u \in G.V()$  do  $G^T$ .insertNode( $u$ )
foreach  $u \in G.V()$  do
  foreach  $v \in G.\text{adj}(u)$  do
    |  $G^T$ .insertEdge( $v$ ,  $u$ )
```

```
integer[] ordine ← new integer[1 ... G.n]
for integer  $i \leftarrow 1$  to  $n$  do  $ordine[i] \leftarrow S.pop()$ 
return cc( $G^T$ , ordine)
```

```
dfsStack(GRAPH G, boolean[] visitato, STACK S, NODE u)
```

```
visitato[ $u$ ] ← true
foreach  $v \in G.\text{adj}(u)$  do
  if not visitato[ $v$ ] then
    | dfsStack( $G$ , visitato,  $S$ ,  $v$ )
  S.push( $u$ )
```

Dato un grafo $G = (V, E)$, il grafo trasposto $G^T = (V, E^T)$ è formato dagli stessi nodi mentre gli archi hanno direzione invertite, $E^T = \{(u, v) | (v, u) \in E\}$

TOP SORT

Si definisce ordine topologico (topsort) di un grafo orientato un ordinamento dei nodi t.c. se (u, v) è un arco del grafo, allora u precede v nell'ordinamento.

Solo i grafi aciclici possono essere ordinati topologicamente.

La procedura `top-sort()` costruisce un DAG (direct acyclic graph) attraverso una visita DFS che calcola l'istante $f[u]$ in cui finisce la visita di ciascun nodo u e ordinando i nodi per istanti di fine decrescenti.

```
topSort(GRAPH G, integer[] ordine)
```

```
boolean[] visitato ← boolean[1 ... G.n]
foreach  $u \in G.V()$  do  $visitato[u] \leftarrow \text{false}$ 
integer  $i \leftarrow 1$ 
foreach  $u \in G.V()$  do
    if not  $visitato[u]$  then
         $i \leftarrow \text{ts-dfs}(G, u, i, visitato, ordine)$ 
```

```
integer ts-dfs(GRAPH G, NODE  $u$ , integer  $i$ , boolean[] visitato, NODE[] ordine)
    visitato[ $u$ ] ← true
    foreach  $v \in G.\text{adj}(u)$  do
        if not  $visitato[v]$  then
             $i \leftarrow \text{ts-dfs}(G, u, i, visitato, ordine)$ 
             $ordine[G.n - i + 1] \leftarrow u$ 
    return  $i + 1$ 
```

CODE CON PRIORITÀ E INSIEMI DISGIUNTI

COD= CON PRIORITÀ

È una struttura dati dove ogni elemento è associato ad una priorità, valore numerico confrontabile tramite una relazione di " \leq ". Il concetto di priorità dipende da come viene usata la coda; in alcuni casi si dà priorità ai valori numerici più bassi (priorità decrescente), in altri a quelli più alti (priorità crescente).

PRIORITYQUEUE

% Crea una coda con priorità vuota
PriorityQueue()

% Restituisce true se la coda con priorità è vuota
boolean isEmpty()

% Restituisce l'elemento minimo di una coda con priorità non vuota
ITEM min()

% Rimuove e restituisce il minimo da una coda con priorità non vuota
ITEM deleteMin()

% Inserisce l'elemento x con priorità p in una coda con priorità non piena
% Restituisce un oggetto PRIORITYITEM che identifica x all'interno della coda
PRIORITYITEM insert(ITEM x , integer p)

% Diminuisce la priorità dell'oggetto identificato da y portandola a p
decrease(PRIORITYITEM y , integer p)

Sono ammesse le operazioni di insert(), min(), deleteMin() e decrease(). Se più elementi hanno la stessa priorità, questa specifica l'ordine in cui verranno estratti.

REALIZZAZIONE TRAMITE LISTE

Si può realizzare una coda con priorità utilizzando liste ordinate e non:

- Ordinate: $\text{min}()$, $\text{deletemin}()$ $O(1)$
 $\text{insert}()$, $\text{decrease}()$ $O(n)$

- Non ordinate: $\text{min}()$, $\text{deletemin}()$ $O(n)$
 $\text{insert}()$, $\text{decrease}()$ $O(1)$

REALIZZAZIONE CON ALBERI BILANCIATI

È possibile realizzare una coda con priorità tramite alberi bilanciati con costo $O(\log n)$ a discapito di una maggior complessità organizzativa della struttura dati e di un maggior consumo di memoria.

REALIZZAZIONE CON VETTORE HEAP

È possibile rendere uniforme la complessità di tutte le operazioni con un vettore HEAP.

Gli elementi della coda con priorità di dimensione n possono essere disposti in un vettore $H[1..n]$ che può essere interpretato come un albero B .

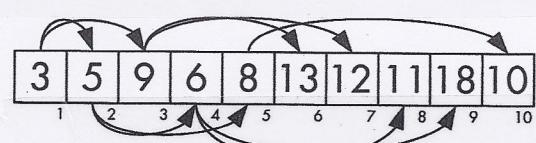
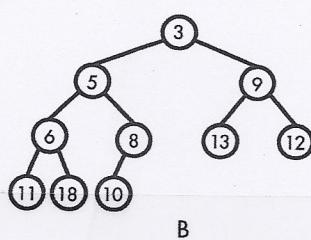
B deve verificare le seguenti proprietà:

- (1) se h è il livello massimo delle foglie, allora ci sono esattamente $2^h - 1$ nodi di livello minore di h
- (2) tutte le foglie di livello h sono addossate a sinistra
- (3) ogni nodo, diverso dalla radice, contiene un elemento della coda con priorità che è maggiore o uguale a quella contenuta nel padre.

Grazie a (1) e (2) ciascun livello (tranne l'ultimo) k di B contiene tutti i possibili 2^k nodi, mentre i nodi del livello h sono addossati sul lato sinistro dell'albero.

10.2 ESEMPIO – [Albero B]

L'albero B di Fig. 10.1 contiene gli elementi della coda con priorità $C = \{5, 10, 8, 11, 13, 12, 9, 18, 3, 6\}$ e verifica le proprietà (1) – (3). Il livello massimo delle foglie è 3, ci sono $2^3 - 1 = 7$ nodi di livello al più 2, e le 3 foglie di livello 3 sono addossate a sinistra.



È possibile passare da una rappresentazione ad albero ad una a vettore in modo semplice:

- si scorre l'albero per livelli, dall'alto verso il basso e da sinistra a destra.
- prima il nodo radice al livello zero
- poi i suoi figli al livello 1, 2, ...

Poiché ogni livello il numero di nodi raddoppia si ha:

- La radice di B viene memorizzata in $H[1]$
- i figli sinistro e destro del nodo i -esimo vengono memorizzati nelle posizioni $H[z_i]$ e $H[z_{i+1}]$
- il padre del nodo i è memorizzato nella posizione $H[i/2]$

Per semplicità:

- $p(i) = \lfloor i/2 \rfloor$ (padre)
- $l(i) = z_i$ (figlio sinistro)
- $r(i) = z_{i+1}$ (figlio destro)

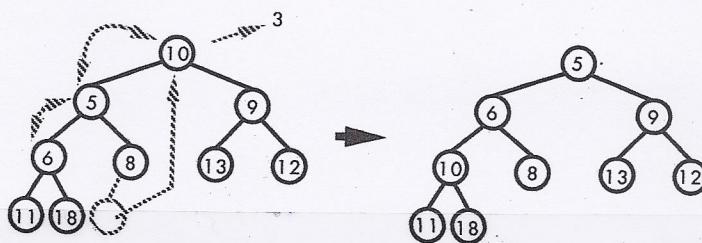
Questa struttura dati viene chiamata minifap.

Per eseguire la procedura `min()` basta leggere l'elemento contenuto nella radice.

Nentre per la `deleteMin()` si sposta la foglia di livello massimo più a destra al posto della radice, e si fa scendere tale elemento lungo un percorso radice-foglia, scambiandolo con il minimo degli elementi contenuti nei figli.

10.4 ESEMPIO – [deleteMin()]

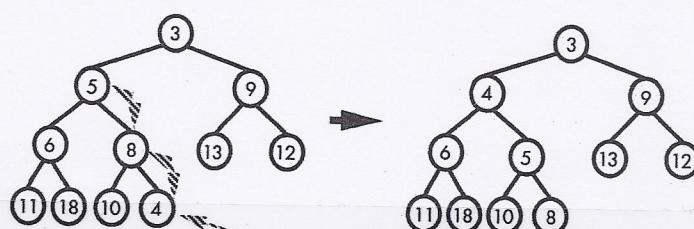
Volendo cancellare 3 dall'albero B di Fig. 10.1, si cancella la foglia contenente 10, si ricopia 10 nella radice, si scambia 10 con 5 e successivamente 10 con 6 (Fig. 10.3).



Per eseguire lo `insert()` si aggiunge una foglia al livello massimo più a sinistra, facendo poi salire tale elemento lungo un percorso foglia-radice, scambiandolo con il padre quando necessario.

10.5 ESEMPIO – [insert]

Volendo inserire l'elemento 4 nell'albero B di Fig. 10.1, si aggiunge un figlio, con 4, al nodo contenente 8, si scambia 4 con 8 e successivamente 4 con 5 (Fig. 10.4).



Numero nodi con heap di altezza h: 2^h e $2^h - 1$

Nella procedura `insert()` si aggiunge una foglia nel livello massimo di B , nella posizione più a sinistra.
In questa posizione l'elemento x potrebbe essere più piccolo di suo padre, quindi vengono scambiati dall'operazione `swap()`, e il cursore " i " risale alla posizione del padre $p[i]$.

Nel caso pessimo la risalita continua fino alla radice.
La funzione `swap()` tiene aggiornato anche la posizione dei record (pos è usato nella `decrease`)

PRIORITYITEM insert(ITEM x , integer p)

precondition: $dim < capacità$

```
dim ← dim + 1
H[dim] ← new PRIORITYITEM()
H[dim].valore ← x
H[dim].priorità ← p
H[dim].pos ← dim
integer i ← dim
while i > 1 and H[i].priorità < H[p(i)].priorità do
    swap(H, i, p(i))
    i ← p(i)
return H[i]
```

swap(PRIORITYITEM[] H , integer i , integer j)

```
H[i] ↔ H[j]
H[i].pos ← i
H[j].pos ← j
```

~~deleteMin()~~ si basa sull'idea di sovrascrivere la radice dell'albero con la foglia più a destra del livello massimo e diminuendo la dimensione del vettore di 1. In questo modo si può chiamare la procedura ~~minHeapRestore()~~ sulla radice, che ha il compito di controllare se le proprietà (1), (2) e (3) sono rispettate. Questa procedura parte dal presupposto che i figli destri e sinistro della radice siano dei min heap, il che è vero essendo che non si ha cancellato la foglia più a destra del livello massimo, ma è stata messa come nuova radice.

ITEM deleteMin(ITEM x)

precondition dim > 0

swap(H, 1, dim)

dim ← dim - 1

minHeapRestore(H, 1, dim)

return H[dim + 1]

minHeapRestore(PRIORITYITEM[] A, integer i, integer dim)

integer min ← i

if $l(i) \leq dim$ and $A[l(i)].priorità < A[min].priorità$ then $min \leftarrow l(i)$

if $r(i) \leq dim$ and $A[r(i)].priorità < A[min].priorità$ then $min \leftarrow r(i)$

if $i \neq min$ then

 swap(A, i, min)

 minHeapRestore(A, min, dim)

La procedura ~~decrease()~~ decrements la priorità dell'oggetto e lo ripositiona nell'heap, scambiandolo con il padre quando serve.

decrease(PRIORITYITEM x, integer p)

precondition $p < x.priorità$

$x.priorità \leftarrow p$

integer i ← x.pos

while $i > 1$ and $H[i].priorità < H[p(i)].priorità$ do

 swap(H, i, p(i))

 i ← p(i)

HEAP SORT

È possibile realizzare un algoritmo di ordinamento ottimo utilizzando la struttura dati heap.

Dato il vettore $A[n]$ è possibile permutare gli elementi richiamando `maxHeapRestore()` più volte in modo che al termine il vettore A soddisfi le proprietà max-heap. `maxHeapRestore()` lavora assumendo che gli alberi sottostanti siano già max-heap, quindi, parte dalle foglie e risale verso la radice.

Applicando `maxHeapRestore()` partendo da $\lfloor n/2 \rfloor$ e risalendo fino a 1, si restaurano le proprietà max-heap essendo certi che i sottosalberi siano max-heap (o sono foglie o sono max-heap).

`heapBuild(ITEM[] A, integer n)`

```
for integer  $i \leftarrow \lfloor n/2 \rfloor$  down to 1 do  
| maxHeapRestore(A,  $i, n$ )
```

`maxHeapRestore(ITEM[] A, integer $i, integer dim$)`

```
integer  $max \leftarrow i$   
if  $l(i) \leq dim$  and  $A[l(i)] > A[max]$  then  $max \leftarrow l(i)$   
if  $r(i) \leq dim$  and  $A[r(i)] > A[max]$  then  $max \leftarrow r(i)$   
if  $i \neq max$  then  
|  $A[i] \leftrightarrow A[max]$   
| maxHeapRestore(A,  $max, dim$ )
```

La complessità di `heapBuild` è $O(n)$

Terminata la ristrutturazione, l'elemento massimo di $A[-n]$ si trova in $A[1]$, ma la posizione che gli compete è $A[n]$, quindi ad ogni passo il primo e l'ultimo elemento vengono scambiati e maxHeapRestore() viene chiamata direttamente.

```
heapsort(ITEM[1 ... n] A)
  heapBuild(A, n)
  for integer i ← n downto 2 do
    A[i] ↔ A[1]
    maxHeapRestore(A, 1, i - 1)
```

Complessità: $\mathcal{O}(n \log n)$