Fibonacci Heaps
ed il loro utilizzo
nell'algoritmo di Prim

Paolo Larcheri 52 SI

26 novembre 2001

Indice

1 Analisi della Complessità Ammortizzata

Prima di affrontare il discorso *"Heap di Fibonacci"* è fondamentale capire cosa sia *l'analisi ammortizzata della complessità* di un algoritmo.
Sostanzialmente, nell'*analisi ammortizzata*, per tempo di esecuzione di una sequenza di operazioni, si intende la media dei costi di tali operazioni. È però importante fare attenzione a non confondersi con l'*analisi del costo medio* in cui si fa riferimento a concetti di teoria della probabilità per valutare il costo di esecuzione sull'istanza media (lo spazio campionario è definito sull'insieme delle istanze).

Esistono diverse tecniche usate per l'*analisi ammortizzata*:

- **Analisi aggregata:**
  si determina un limite superiore del costo totale $T(n)$ di una sequenza di operazioni per poi ottenere il costo ammortizzato: $T(n)/n$ dove n è il numero di operazioni della sequenza.

- **Metodo degli accantonamenti:**
  consiste nel creare un *"debito"* inteso come differenza tra costo reale e costo attribuito all'operazione. Il debito viene poi saldato definendo più alto il costo attribuito a altre operazioni facendo attenzione a far quadrare il bilancio. Alla "chiusura" è ammesso un bilancio negativo, in modo tale che ogni debito sia stato pagato. Se questo vincolo è stato rispettato, resta legittimato il ragionamento in termini di costi attribuiti anzichè di costi reali.

- **Metodo del potenziale:**
  è una tecnica molto simile al *Metodo degli accantonamenti* in cui però il "credito" accumulato è direttamente associato alla struttura di dati e tale "credito" è visto come "l'energia potenziale" della struttura stessa. Utilizzeremo questa tecnica per una valutazione ammortizzata della complessità delle operazioni sui *Fibonacci Heap*. Pertanto nella sottosezione *1.1* spieghiamo la tecnica in maggior dettaglio illustrando un esempio che ne facilita la comprensione.

1.1 Metodo del Potenziale

Data una struttura dati e una sequenza di n operazioni su di essa, si definisca come D_0 la struttura iniziale, per D_i la struttura dati che si ottiene eseguendo l'i-esima operazione sulla struttura D_{i-1}. Per ogni $1 \leq i \leq n$, sia c_i é il costo effettivo della i-esima operazione della sequenza. È possibile definire una *funzione potenziale* (o più semplicemente *potenziale*) $\Phi(D)$ che associa allo stato della struttura un numero reale detto appunto *potenziale*[1].
Il costo ammortizzato è cosí definito:

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

[1]per $\Phi(D_i)$ si intenderà in seguito il potenziale associato allo stato della struttura D dopo l'i-esima operazione.

Il costo ammortizzato delle n operazioni quindi sará:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

Per garantire che $\sum_{i=1}^{n} \hat{c}_i$ domini il costo effettivo totale deve valere che $\Phi(D_n) \geq \Phi(D_0)$ cioè che l'analisi della complessità è terminata con un "credito" anzichè con un "debito". Non essendo sempre noto il numero di operazioni che potrebbero essere eseguite, è indispensabile garantire che $\Phi(D_i) \geq \Phi(D_0)$ per tutti i valori di i. Per semplicità spesso si definisce $\Phi(D_0) = 0$ e basta poi dimostrare che $\Phi(D_i) \geq 0 \ \forall i$ per essere sicuri di avere ottenuto un limite superiore al costo effettivo totale.

- **Esempio**(Operazioni su pile): si consideri una *pila* con le operazioni Push, Pop e MultiPop e si definisca Φ come il numero di elementi presenti nella pila. Si osservi come Φ sia funzione del solo stato della struttura dati, ossia Φ è effettivamente una funzione potenziale. Nel caso iniziale la pila è vuota e si ha $\Phi(D_0) = 0$; dopo un numero indefinito i di operazioni il potenziale è $\Phi(D_0) \geq 0$. Questo fatto, come detto in precedenza, garantisce che il costo ammortizzato totale di n operazioni rispetto alla funzione potenziale Φ domini il costo effettivo delle operazioni stesse.
  Definiamo e il numero di elementi presenti nella pila.
  Non resta che calcolare i costi ammortizzati delle varie operazioni:

  Push: se la i-esima operazione è una Push si ha:
  $c_i = 1$;
  $\Phi(D_i) - \Phi(D_{i-1}) = e - (e-1) = 1$;
  Sostituendo in $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ si ha:
  $c_i = 1 + e - (e-1) = 2$;
  Il costo ammortizzato è ***costante***.

  Pop: se la i-esima operazione è una Pop si ha:
  $c_i = 1$;
  $\Phi(D_i) - \Phi(D_{i-1}) = e - (e+1) = -1$;
  Sostituendo in $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ si ha:
  $c_i = 1 + e - (e+1) = 0$;
  Il costo ammortizzato é ***costante***.

  MultiPop: se la i-esima operazione è una MultiPop si ha:
  $c_i = t'$ dove t' e' il numero di elementi tolti alla pila;
  $\Phi(D_i) - \Phi(D_{i-1}) = e - (e+t') = -t'$;
  Sostituendo in $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ si ha:
  $c_i = t' + e - (e+t') = 0$;
  Il costo ammortizzato è ***costante***.

  Il costo ammortizzato di tutte e tre le operazioni é $O(1)$, quindi il costo di una qualsiasi sequenza di n operazioni é $O(n)$.

2 Gli Heap

Per *Heap* in generale si intende una struttura dati atta a contenere e gestire un insieme di dati ordinabili. Le fondamentali operazioni esposte sono:

INSERISCI: per l'inserimento di un nuovo elemento nello Heap di Fibonacci.

TROVA-MIN: per la ricerca del nodo con chiave minima.

ESTRAI-MIN: per la ricerca e l'estrazione del nodo con chiave minima.

UNIONE: per l'unione di due Heap di Fibonacci.

DECREMENTA-CHIAVE: decrementa la chiave di un nodo e, se necessario, ristruttura lo Heap.

CANCELLA: per la cancellazione di nodo.

3 Gli Heap di Fibonacci

Gli Heap di Fibonacci furono introdotti verso la metà degli anni '70 da Fredman e Tarjan. In seguito lo stesso Tarjan, insieme a Driscoll, Sarnak e Sleator svilupparono gli "Heap rilassati" come alternativa agli Heap di Fibonacci. Nella nostra trattazione descriveremo solamentla prima delle due strutture di dati appena menzionate. I *Fibonacci Heaps* sono un'"ottima" implementazione di Heap in quanto le operazioni più costose hanno complessità (ammortizzata) $O(\log n)$ (e sono quelle che prevedono l'eliminazione di un elemento dall heap) mentre tutte le altre costano tempo (ammortizzato) costante[2] [3]:

	FIBONACCI HEAPS		**BINOMIAL HEAPS**
operazione	**costo reale**	**costo amm.**	**costo reale**
INSERISCI	$O(1)$	$O(1)$	$O(1)$
TROVA-MIN	$O(1)$	$O(1)$	$O(\log n)$
ESTRAI-MIN	$O(\log n + t(H))$	$O(\log n)$	$O(\log n)$
UNIONE	O(1)	O(1)	$O(\log n)$
DECREMENTA-CHIAVE	$O(\log n)$	$O(1)$	$O(\log n)$
CANCELLA	$O(\log n + t(H))$	$O(\log n)$	$O(\log n)$

3.1 Struttura degli Heap di Fibonacci

Gli *Heap di Fibonacci* sono un insieme di alberi con l'ordinamento parziale dello heap[4]; questi alberi non sono ordinati per grandezza come negli *Heap Bonimiali* e le loro radici sono poste in una lista bidirezionale (lo heap sarà sostanzialmente un puntatore

[2]In tabella è riportata anche la complessità reale nel caso pessimo.
[3]Per $t(H)$ si intende il numero di radici contenute nella lista dello heap
[4]Per ogni nodo vale che il padre ha chiave minore di ciascun figlio; in questo modo é anche garantito che il nodo con chiave minima dello heap è sicuramente la radice di un albero.

a tale lista). Ogni nodo ha un puntatore al nodo-padre, un puntatore alla lista (bidirezionale) dei figli, un puntatore al suo fratello destro ed uno al suo sinistro. Inoltre ogni elemento ha un campo intero[5] che contiene la sua *chiave*, un campo intero che contiene il suo *grado* (cioè il numero dei suoi figli) e un'altro campo booleano che indica se tale elemento è marcato, cioè se tale nodo ha perso un figlio dall'ultima volta in cui è diventato figlio di un altro nodo.
Uno *Heap di Fibonacci* è composto da un puntatore alla radice con chiave minima, un puntatore alla lista delle radici e due interi: uno che tiene conto del numero complessivo di elementi e l'altro del numero complessivo di alberi contenuti nello Heap. In Fig. 1 è mostrata una generica struttura di Heap di Fibonacci.

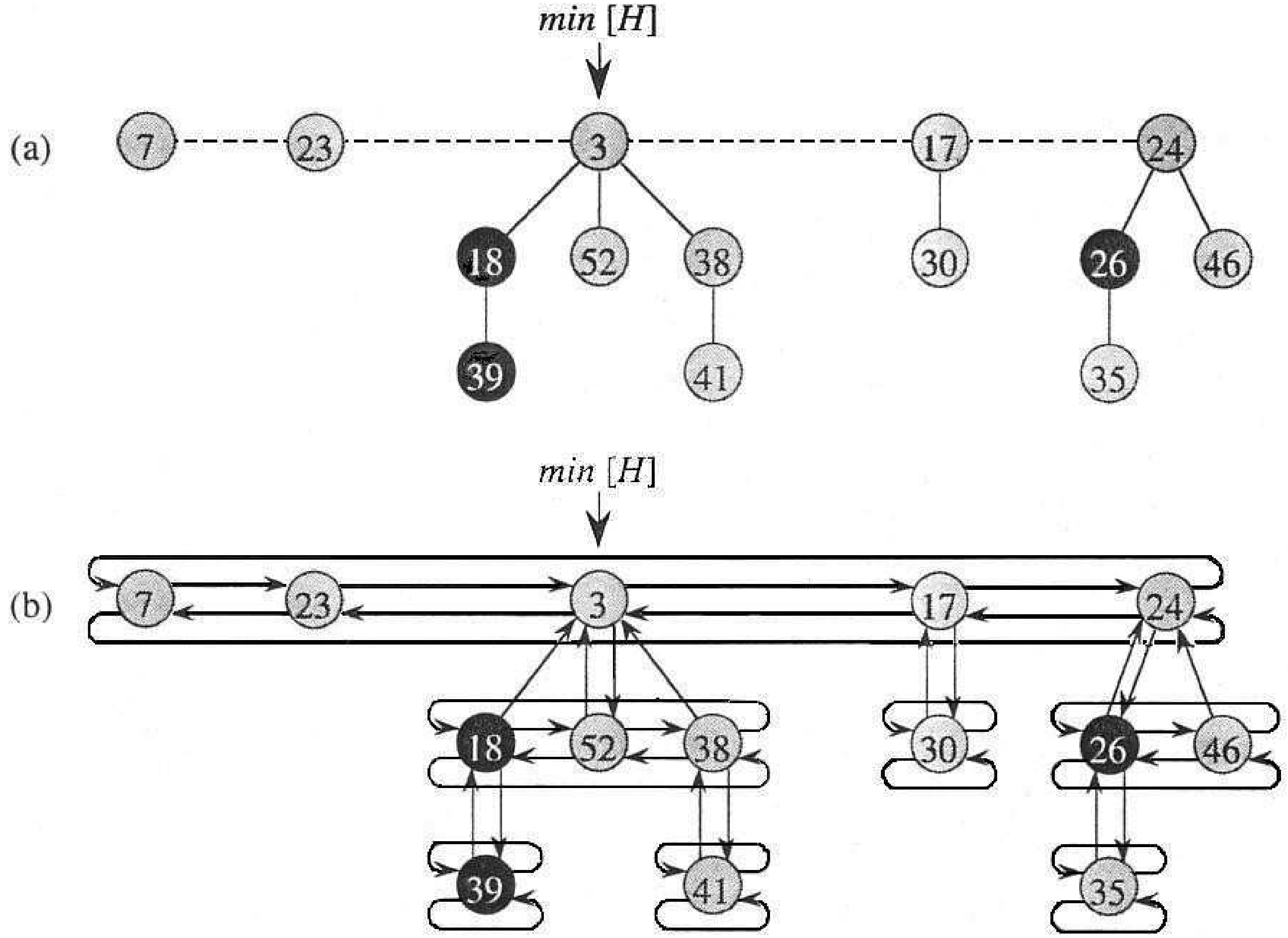


Figura 1: (a): rappresentazione di un possibile heap di Fibonacci; (b): lo stesso heap rappresentato con le liste delle radici e dei figli come nella reale implementazione; in nero sono rappresentati i nodi marcati.

[5]non è necessario che il campo chiave sia di tipo intero, ma è sufficiente che faccia parte di un insieme che goda della proprietà di ordinamento totale!

Lo *Heap di Fibonacci* possiede i seguenti attributi:

attributo	**descrizione**
ptesta	puntatore alla prima radice della lista
pmin	puntatore alla radice con chiave minima
nnodi	numero complessivo di nodi
nalberi	numero complessivo di alberi

Ogni *nodo* dello Heap possiede i seguenti attributi:

attributo	**descrizione**
ppadre	puntatore al padre
pfigli	puntatore alla lista dei figli
pdestro	puntatore al fratello destro
psinistro	puntatore al fratello sinistro
grado	numero di figli
chiave	valore del nodo
marcato	indica se il nodo è marcato o no

3.2 Operazioni - Creazione di un nuovo Heap di Fibonacci

La creazione di un nuovo *heap di Fibonacci* consisterà nel "settare" in questo modo gli attributi:

attributo	**valore**
ptesta	NULL
pmin	NULL
nnodi	0
nalberi	0

Il costo *reale* di questa operazione è $O(1)$.

3.3 Funzione Potenziale

Come detto in precedenza, il costo ammortizzato delle operazioni sugli *Heap di Fibonacci* è analizzato con il *metodo del potenziale*; è giunto quindi il momento di definire la *funzione potenziale* per questo tipo di struttura dati. Definiamo tale funzione:

$$\Phi(FH_i) = t(FH_i) + 2m(FH_i)$$

dove $t(FH_i)$ è il numero di alberi e $m(FH_i)$ è il numero di nodi marcati dello heap dopo la i-esima operazione.
Questa equazione garantisce che un limite superiore al costo totale ammortizzato è anche un limite superiore al costo totale reale poichè $\Phi(FH_0) = 0$ mentre $\Phi(FH_i) \geq 0$ $\forall i > 0$.

3.4 Operazioni - Inserimento di un nuovo nodo

La funzione di inserimento nello Heap prevede che sia passato come parametro il valore della chiave del nuovo elemento; all'interno della funzione viene allocato lo spazio in memoria per il nuovo nodo:

```
INSERISCI(nuova_chiave) {
  new = allocazioneNuovoNodo();
  new->chiave = nuova_chiave;
  new->grado = 0;
  new->ppadre = NULL;
  new->pfigli = NULL;
  new->marcato = false;
  inserisci new in testa alla lista delle radici;

  if(FHeap->min == NULL) oppure (FHeap->min->chiave > nuova_chiave)
    FHeap->min = new;

  FHeap->nnodi = FHeap->nnodi + 1;
  FHeap->nalberi = FHeap->nalberi + 1;
}
```

Il nuovo elemento viene inserito in testa alla lista delle radici e vengono opportunamente inizializzati tutti i suoi campi; se necessario viene aggiornato il *minimo* dello Heap e infine vengono incrementati i contatori del numero dei nodi e del numero di radici.

A differenza delle funzioni di inserimento di altri tipi di heap (come per esempio gli *Heap Binomiali*), questa funzione non si preoccupa di ristrutturare in nessun modo la struttura dati. In questo senso si dice che gli Heap di Fibonacci sono una struttura dati "pigra" poichè tende a rinviare il lavoro; questo aspetto dimostra come il progetto degli Heap di Fibonacci sia andato di pari passo con l'analisi ammortizzata delle complessità delle sue operazioni.
Cerchiamo di determinare il costo ammortizzato di questa operazione:
il potenziale abbiamo detto che vale

$$\Phi(FH_i) = t(FH_i) + 2m(FH_i)$$

supponiamo che la i-esima operazione di una sequenza sia di inserimento:

$$\hat{c_i} = c_i + \Phi(FH_i) - \Phi(FH_{i-1})$$

il suo costo ammortizzato sarà:

$$\hat{c_i} = c_i + [t(FH_i) + 2m(FH_i)] - [t(FH_{i-1}) + 2m(FH_{i-1})]$$

definiamo $t(FH_i) = k$ e $m(FH_i) = j$:

$$\hat{c_i} = O(1) + [k + 2j] - [k - 1 + 2j] = O(1) + 1 = \underline{O(1)}.$$

Il costo ammortizzato è quindi **<u>costante</u>**.

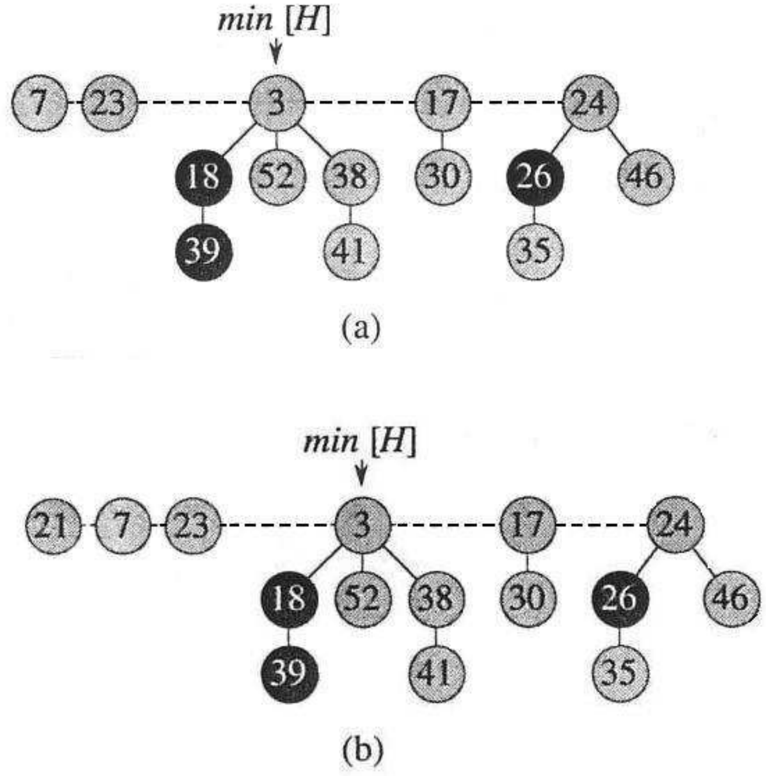


Figura 2: (a): lo heap prima dell'inserimento; (b): lo heap dopo l'inserimento.

3.5 Operazioni - Ricerca del minimo

Viene ritornato il puntatore al nodo con chiave minima (se lo Heap è vuoto viene ritornato NULL):

```
TROVA-MIN() {
  return FH->min;
}
```

Il costo *reale* di questa funzione è $O(1)$ e, dal momento che da prima a dopo l'esecuzione il potenziale dello Heap non cambia, anche il costo ammortizzato è **costante**.

3.6 Operazioni - Unione di due Heap di Fibonacci

Viene ritornato uno Heap che rappresenta l'unione dei due passati per indirizzo:

```
UNIONE(FH1, FH2) {
  FH = allocoNuovoFHeap();

  FH->min = FH1->min;
  concatena le liste di FH1 e FH2;
```

```
  if(FH->min == NULL) oppure ((FH2->min != NULL)and(FH2->min < FH1->min))
    FH->min = FH2->min;

  FH->nnodi = FH1->nnodi + FH2->nnodi;
  FH->nalberi = FH1->nalberi + FH2->nalberi;

  rilascia memoria occupata da FH1 e FH2;

  return FH;
}
```

Questa funzione prevede la creazione di un nuovo *Heap di Fibonacci*, la concatenazione delle liste dei due Heap passati per indirizzo e l'aggiornamento dei campi *min*, *nnodi* e *nalberi*.

Come nel caso dell' inserimento di un nuovo nodo, questa operazione non necessita della ristrutturazione dello Heap.
La differenza di potenziale è

$$\begin{aligned}&\Phi(FH) - (\Phi(FH1) + \Phi(FH2))\\&= (t(FH) + 2m(FH)) - [(t(FH1) + 2m(FH1)) + (t(FH2) + 2m(FH2))]\\&= 0\end{aligned}$$

dal momento che $t(FH) = t(FH1) + t(FH2)$ e $m(FH) = m(FH1) + m(FH2)$;
il costo ammortizzato dell'operazione di unione è quindi **costante**, come per altro il suo costo reale.

3.7 Operazioni - Estrazione del nodo con chiave minima

L'operazione di estrazione del nodo minimo costituisce, per quel che riguarda gli *Heap di Fibonacci*, il lavoro più complesso in quanto prevede la ristrutturazione dello Heap, che nelle altre funzioni non é presente.
Infatti la funzione di estrazione del minimo (come si puo' vedere dallo "pseudo-codice") utilizza due funzioni ausiliarie il cui funzionamento è spiegato in seguito:

```
ESTRAI-MIN() {
  minimo = FH->min;

  if(FH->min != NULL)
    for(ogni figlio x di FH->min)
      togli x dala lista dei figli di min;
      aggiungi x alla lista delle radici di FH;

      x->ppadre = NULL;
      FH->nalberi = FH->nalberi + 1;

    if(FH->nnodi == 1)
      FH->min = NULL;
```

```
      FH->nalberi = FH->nalberi - 1;
    else
      FH->min = FH->min->pdestro;
      CONSOLIDATE();     <- funzione di ristrutturazione dello Heap

    FH->nnodi = FH->nnodi - 1;

  return minimo;
}
```

La ESTRAI-MIN() accede direttamente al minimo grazie al campo *min* dello Heap, "promuove" ogni suo figlio a radice; se il minimo era l'unico nodo mette il campo *min* a NULL, altrimenti gli assegna il suo adiacente destro (che non necessariamente è realmente il nuovo minimo) ed esegue la funzione di restaurazione; infine viene appropriatamente aggiornato il campo contatore dei numeri di nodi dello Heap e viene ritornato il nodo con chiave minima. Chiaramente se lo Heap è vuoto viene ritornato NULL.

La CONSOLIDATE() ha il compito di ricompattere il più possibile gli alberi; tale operazione eseguirà ripetutamente i seguenti due passi:

- trovare due radici con lo stesso grado;
- "trasformare" quella con chiave più piccola in un figlio di quella con chiave maggiore.

Il secondo dei due passi è effettuato dalla LINK, un'altra funzione ausiliaria il cui (semplice) funzionamento è spiegato in seguito. Vediamo una possibile implementazione della CONSOLIDATE:

```
R[ ]:  vettore di puntatori alle radici (init: NULL),
       R[i] punta all'ultima radice di grado i incontrata
x, y:  puntatori alle radici correnti
g:     contiene il grado dell'ultima radice considerata

CONSOLIDATE() {
   R[i] = NULL per ogni i;

   for(ogni nodo nd della lista della radici di FH)
      g = nd->grado;
      x = nd;

      while(R[g] != NULL)
         y = R[g];

         if(x->chiave > y->chiave)
           swap(x, y);
```

```
            LINK(y, x);

            R[g] = NULL;
            g = g+1;

         R[g] = x;

      FH->min = NULL;

      for(ogni i fino alla fine di R[ ])
         if (R[i] != NULL)
           aggiungi R[i] alla lista delle radici;

           if(FH->min == NULL)oppure(R[i]->chiave < FH->min->chiave)
               FH->min = R[i];
   }
```

Per ogni radice dello Heap controlla se è stata precedentemente incontrata un'altra radice con lo stesso grado: in caso affermativo quella con chiave minore viene *"linkata"* all'altra (con la chiamata della funzione LINK), il grado della radice-padre viene incrementato e viene effettuato nuovamente lo stesso controllo. In caso negativo invece, la radice viene memorizzata nel vettore $R[]$ alla posizione i, dove i è il grado della radice stessa.
Infine viene aggiornato il campo *min* e ricreata la lista delle radici: viene scandito tutto il vettore $R[]$ inserendo una ad una ogni radice presente in esso.
La funzione LINK serve per fondere due alberi in uno; la radice di uno dei due (quella con *chiave* minore) diventa un nodo figlio della radice dell'altro:

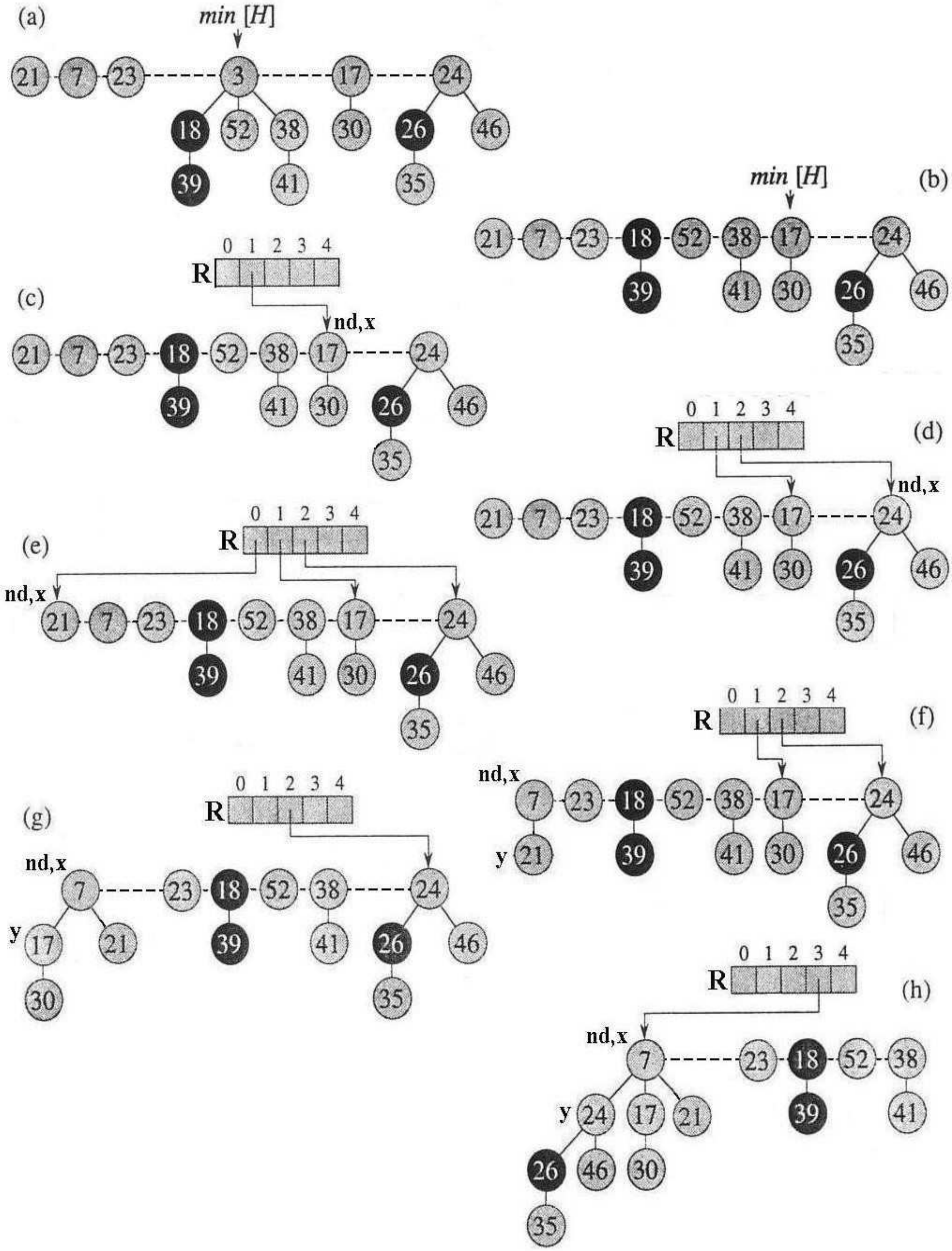
```
LINK(y, x) {
  togli y dalla lista delle radici di FH;

  aggiungi y alla lista dei figli di x;

  x->grado = x->grado + 1;
  y->marcato = false;
}
```

In Fig. 3 è mostrato il processo di estrazione del minimo(ESTRAI-MIN) e di restaurazione dello Heap (CONSOLIDATE e LINK).

Per il calcolo del costo della ESTRAI-MIN è indispensabile assumere che esiste un limite superiore $D(n) = \log n$ al grado massimo di un nodo qualsiasi dello *Heap di Fibonacci*; questo fatto verrà dimostrato in seguito (sottosezione *2.10*). Affinchè il costo della ESTRAI-MIN sia ottimale è ovvio che la dimensione di $R[]$ debba essere


(a)
min [H]
(b)
min [H]
(c)
R
0 1 2 3 4
nd,x
(d)
R
0 1 2 3 4
nd,x
(e)
R
0 1 2 3 4
nd,x
(f)
R
0 1 2 3 4
nd,x
y
(g)
R
0 1 2 3 4
nd,x
y
(h)
R
0 1 2 3 4
nd,x
y

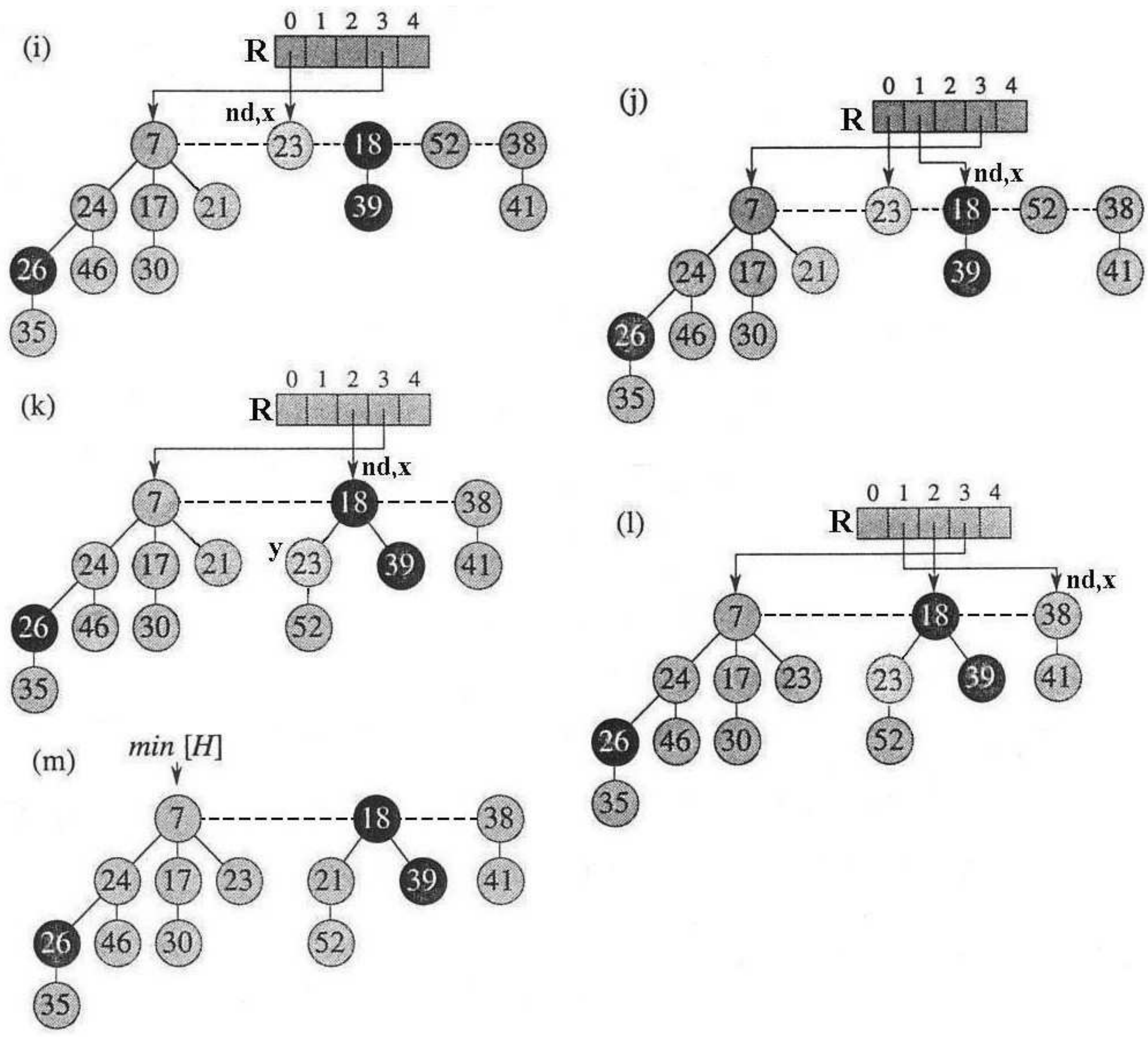


Figura 3: (a): lo heap prima dell'estrazione; (b): viene estratto il minimo e tutti i suoi figli sono promossi a radice; (c): inizia la scansione della lista delle radici: 17 ha grado 1; (d): 24 ha grado 2; (e): 21 ha grado 0; (f): 7 ha grado 0 $\Rightarrow$ 21 linkato a 7; (g): ora 7 ha grado 1 $\Rightarrow$ 17 linkato a 7; (h): ora 7 ha grado 2 $\Rightarrow$ 24 linkato a 7; (i): 23 ha grado 0; (j): 18 ha grado 1; (k): 52 ha grado 0 $\Rightarrow$ 52 linkato a 23; ora 23 ha grado 1 $\Rightarrow$ 23 linkato a 18; ' (l): 38 ha grado 1; (m): lo heap é finalmente restaurato!

anch'essa ottimale in quanto tale vettore viene più volte scandito e questo fatto potrebbe fare aumentare il costo totale; la sua dimensione sará $O(\log n)$ che è proprio il grado massimo che una radice può avere.
Dimostriamo ora che il costo ammortizzato della ESTRAI-MIN è $O(\log n)$: un contributo pari a $O(\log n)$ è dato dal ciclo che promuove a radice tutti gli $O(\log n)$ figli del nodo minimo; vi sono poi l'inizializzazione del vettore $R[]$ (all'inizio della CONSILIDATE) e la ricreazione della lista delle radici dello Heap (alla fine della CONSOLIDATE) che consitono entrambe nello scandire $R[]$ e che quindi portano un contributo pari anch'esso a $\log n$.
Resta ora da considerare la parte centrale della funzione CONSOLIDATE, quella incaricata di trovare e fondere gli alberi di uguale dimensione; il ciclo in questione opera una scansione della lista delle radici: tale lista è al più $D(n) + t(FH) - 1$ dal momento che è formata dalla vecchia lista delle radici ($t(FH)$) e dai figli del nodo con chiave minima (D(n)); inoltre ad essa è stato tolto il minimo (-1). Quindi il costo reale totale in questo caso è $\underline{O(D(n) + t(FH))}$. Trovato il costo reale non resta che il calcolo del potenziale per avere al termine il costo ammortizzato. Il potenziale prima:
$\Phi(FH_{i-1}) = t(FH_{i-1}) + 2m(FH_{i-1})$
e dopo:
$\Phi(FH_i) = log(n) + 2m(FH_{i-1})$
$\Phi(FH_i) = D(n) + 2m(FH_{i-1})$
Notiamo subito che il potenziale di FH_i è minore o uguale a quello di FH_{i-1} in quanto il numero di radici dopo la ristrutturazione dello Heap è diventato al più $D(n)$ (esattamente come il numero di radici in uno *Heap Binomiale*).
Il costo ammortizzato come al solito è:
$\hat{c}_i = c_i + \Phi(FH_i) - \Phi(FH_{i-1})$
sostituendo i due *potenziali*:
$\hat{c}_i = O(D(n) + t(FH_{i-1})) + D(n) + 2m(FH_{i-1}) - t(FH_{i-1}) - 2m(FH_{i-1})$
$\hat{c}_i = O(D(n) + t(FH_{i-1})) + D(n) - t(FH_{i-1})$
$\hat{c}_i = O(2D(n)) + t(FH_{i-1}) - t(FH_{i-1})$
$\hat{c}_i = O(D(n))$
dal momento che $D(n)$ è stato assunto come $\log n$, si ha:
$\mathbf{\hat{c}_i = O(\log(n))}$

3.8 Operazioni - Decremento di una chiave

Il vantaggio più grande offerto dallo *Heap di Fibonacci* è la possibilità di decrementare la chiave di un nodo in tempo (ammortizzato) costante. La funzione che decrementa la chiave di un nodo utilizza due funzioni ausiliarie il cui funzionamento è spiegato in seguito. Vediamo una possibile implementazione:

```
pnd: puntatore al nodo la cui chiave va decrementata
nk:  nuovo valore della chiave
ppd: puntatore al padre

DECREMENTA-CHIAVE(pnd, nk) {
   if (nk > pnd->chiave)
```

```
      Errore;

   pnd->chiave = nk;
   ppd = pnd->ppadre;

   if(ppd != NULL) e (pnd->chiave < ppd->chiave)
      TAGLIA(pnd, ppd);
      TAGLIO-A-CASCATA(ppd);

   if(pnd->chiave < FH->min->chiave)
     FH->min = pnd;
}
```

Il primo passo consiste nel confrontare il nuovo valore della chiave con quello vecchio: se il vecchio è minore del nuovo è chiaramente restituito errore; in caso contrario il campo *chiave* del nodo assume il nuovo valore; in seguito si controlla se il padre del nodo in questione ha chiave maggiore del figlio: in caso positivo va ristabilita la proprità degli *Heap* secondo cui un nodo ha sempre chiave maggiore o uguale al padre (se questo esiste). Per compiere questo vengono chiamate le funzioni ausiliarie TAGLIA e TAGLIO-A-CASCATA. Come ultima cosa viene, se necessario, aggiornato il puntatore al minimo.
Funzionamento della funzione TAGLIO:

```
TAGLIO(pnd, ppd) {
   toglie pnd dalla lista dei figli di ppd;

   ppd->grado =  ppd->grado -1;
   pnd->ppadre = NULL;
   pnd->marcato = false;
}
```

Funzionamento della funzione TAGLIO-A-CASCATA:

```
TAGLIO-A-CASCATA(pnd) {
  ppd = pnd->ppadre;

  if(ppd!=NULL)
    if(pnd->marcato == false)
       pnd->marcato = true;
    else
       TAGLIO(pnd, ppd);
       TAGLIO-A-CASCATA(ppd);
}
```

Per capire meglio il procedimento di decremento della chiave di un nodo e tutto quello che ne consegue è utile chiarirsi le idee con un esempio:

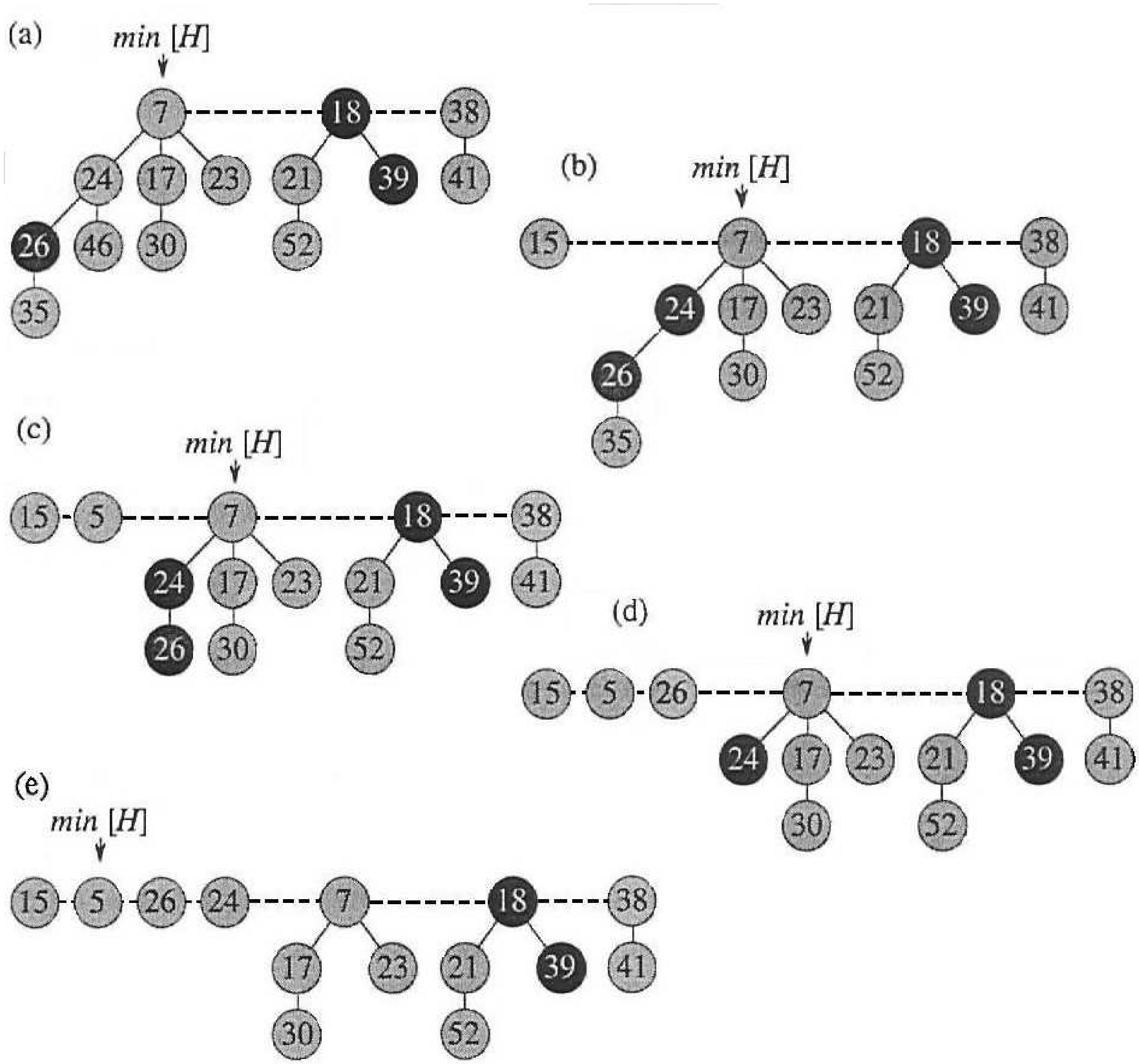


Figura 4: processo di decremento di una chiave e conseguenti tagli.

in Fig.4 sono raffigurati i vari stati dello Heap durante il decremento di due chiavi ed i successivi tagli:

(a) stato iniziale dello Heap;

(b) il nodo con chiave 46 prende chiave 15 e viene tagliato poichè la nuova radice è minore di quella del padre; la chiamata a TAGLIO-A-CASCATA marca a *true* il padre dal momento che ha appena perso il figlio;

(c) il nodo con chiave 35 prende chiave 5 e viene tagliato poichè la nuova radice è minore di quella del padre; la chiamata a TAGLIO-A-CASCATA “incontra” il padre del nodo tagliato (26) che era in precedenza stato marcato (non importa quando); per questo viene tagliato, marcato a *false* e viene richiamata a sua volta la TAGLIO-A-CASCATA su suo padre...

(d) la chiamata a TAGLIO-A-CASCATA “incontra” il padre del nodo tagliato (26) che era in precedenza stato marcato (non importa quando); per questo viene tagliato, marcato a *false* e viene richiamata a sua volta la TAGLIO-A-CASCATA su suo padre...

(e) ...dal momento che anche suo padre era stato marcato (vedi *(b)*) viene anch'esso tagliato e smarcato; a questo punto cessano i tagli a cascata perchè il nodo con chiave 7 è una radice;

A cosa serve fare tutti questi tagli se concretamente sarebbe stato sufficiente fare solo il taglio del nodo la cui radice è stata decrementata?
La risposta è semplicissima: il procedimento permette alla DECREMENTA-CHIAVE di avere costo ammortizzato pari a $O(1)$: supponiamo che la TAGLIO-A-CASCATA sia richiamata k volte dopo la chiamata a DECREMENTA-CHIAVE; il costo della DECREMENTA-CHIAVE stessa è $O(1)$ e cosí anche per la singola chiamata a TAGLIO-A-CASCATA (senza le chiamate ricorsive): quindi il *costo reale* della DECREMENTA-CHIAVE è $\underline{O(k)}$.
Il cambio di potenziale dello Heap:

$$\Phi(FH_{i-1}) = t(FH_{i-1}) + 2m(FH_{i-1})$$

Poichè il numero di alberi è aumentato di $k-1$ in k chiamate a TAGLIO-A-CASCATA più il taglio del nodo decrementato abbiamo che:

$$t(FH_i) = t(FH_{i-1}) + (k-1) + 1 = t(FH_{i-1}) + k$$

Poichè il numero di nodi marcati è diminuito di $k-1$ unità nei tagli a cascata e puó essere stato aumentato di 1 unità nell'ultima chiamata alla TAGLIO-A-CASCATA che può aver marcato un nodo, si ha:

$$m(FH_i) = m(FH_{i-1}) - k + 2$$

Dunque:

$$\Phi(FH_i) = t(FH_{i-1}) + k + 2m(FH_{i-1}) - k + 2$$

La differenza di potenziale è quindi:

$$\Phi(FH_i) - \Phi(FH_{i-1}) = k - 2k + 4 = 4 - k$$

e il costo ammortizzato:

$$\begin{aligned}\hat{c}_i &= c_i + (\Phi(FH_i) - \Phi(FH_{i-1})) \\ &= O(k) + 4 - k = \mathbf{O(1)}\end{aligned}$$

Ora risulta anche chiaro il motivo per cui la funzione potenziale degli Heap di Fibonacci sia stata definita con il doppio del numero di nodi marcati: in questo modo il costo ammortizzato della DECREMENTA-CHIAVE risulta essere **costante**.

3.9 Operazioni - Cancellazione di un nodo

Il processo di eliminazione di un nodo consiste nel decrementare a $-\infty$ la chiave del nodo da cancellare e quindi nell'estrarre il minimo:

```
DELETE(x) {
  DECREMENTA-CHIAVE(x, -infinito);
  ESTRAI-MIN();
}
```

Il costo ammortizzato della DELETE è data dalla somma dei costi ammortizzati della DECREMENTA-CHIAVE e della ESTRAI-MIN; quindi:

$$\hat{c} = O(log(n)) + O(1) = \mathbf{O(log(n))}$$

3.10 Limitazione del grado massimo

Affinchè il costo ammortizzato della fuzione DECREASE-KEY e DELETE sia realmente $O(\log n)$, dobbiamo dimostrare che il limite superiore al grado di un qualsiasi nodo di uno Heap di Fibonacci è $O(\log n)$.

Chiameremo $x \rightarrow size$ il numero di nodi dell'albero radicato in x (compreso x stesso). La dimostrazione mostra anche come $x \rightarrow size$ è esponenziale a $x \rightarrow grado$.

Lemma 1 *Sia x un nodo di un qualsiasi Heap di Fibonacci tale che abbia* (x→grado)=*k. Siano $y_1, y_2 \ldots, y_k$ i suoi figli ordinati in base al loro ordine di arrivo. Allora vale che* (y_1 →grado) ≥ 0 *e* (y_i →grado) $\geq i - 2$ *per i = 2,3,...k.*

Dimostrazione: per dire che $(y_1 \rightarrow grado) \geq 0$ non serve dimostrazione; dato $i \geq 2$ si nota che quando y_i è diventato figlio di x, x aveva i figli $y_1, y_2 \ldots, y_{i-1}$ e quindi $(x \rightarrow grado) \geq i - 1$; poichè y_i può essere stato aggiunto solo se aveva grado uguale a x, in quel momento si aveva $(y_i \rightarrow grado) \geq i - 1$. Da quell'istante y_i a perso al più un figlio perchè altrimenti sarebbe stato tagliato (vedi sottosezione *2.8* ed in particolare la funzione *taglio-a-cascata*). Ne consegue che $(y_i \rightarrow grado) \geq i - 2$. □

Con quello che stiamo per vedere si capisce finalmente perchè questo tipo di struttura si chiami *"Heap di Fibonacci"*. Come tutti sanno l' i-esimo numero di Fibonacci è:

$$F_i = \begin{cases} 0 & \text{se } i = 0 \\ 1 & \text{se } i = 1 \\ F_{i-1} + F_{i-2} & \text{se } i \geq 2 \end{cases}$$

Il seguente lemma dimostra che F_i può essere espresso in un'altro modo:

Lemma 2 *per tutti gli $i \geq 0$ si possono esprimere i numeri di Fibonacci come segue:*

$$F_{i+2} = 1 + \sum_{j=0}^{i} F_j$$

Dimostrazione: Per induzione su i:
Per $i = 0$

$$1 + \sum_{j=0}^{0} F_j = 1 + F_0 = 1 + 0 = 1 = F_2$$

Data l'ipotesi induttiva $\quad F_{i+1} = 1 + \sum_{j=0}^{i-1} F_j \quad$, si ha:

$$F_{i+2} = F_i + F_{i+1} = F_i + (1 + \sum_{j=0}^{i-1} F_j) = 1 + \sum_{j=0}^{i} F_j \qquad \square$$

Indichiamo con ϕ il cosiddetto *rapporto aureo* definito come

$$\phi = (1 + \sqrt{5})/2 = 1.61803\ldots$$

Il Lemma 4 che termina la dimostrazione utilizza il seguente Lemma 3.

Lemma 3 $\quad F_{i+2} \geq \phi^i$

Dimostrazione(per induzione su i):
<u>Caso Base:</u>
Semplice verifica per $i = 0$ e $i = 1$
<u>Passo Induttivo:</u>
per ipotesi

$$F_{i+2} = F_{i+1} + F_i \geq \phi^{i-1} + \phi^{i-2} = \phi^{i-2} \cdot (\phi + 1)$$

basta quindi dimostrare che

$$\phi^{i-2} \cdot (\phi + 1) \geq \phi^i$$

cioè che

$$\phi + 1 \geq \phi^2$$

ed infatti sostituendo il valore di Φ si verifica che

$$(3+\sqrt{5})/2 = (6+2\sqrt{5})/4 \qquad \square$$

Lemma 4 *sia x un nodo di uno Heap di Fibonacci e $k = x \rightarrow grado$. Allora vale che $(x \rightarrow size) \geq F_{k+2} \geq \phi^k$ dove $\phi = (1+\sqrt{5})/2$.*

Dimostrazione: Definiamo s_k il minimo valore possibile di $z \rightarrow size$ tra i nodi z t.c. $z \rightarrow grado = k$. Chiaramente si ha che $s_0 = 1, s_1 = 2$ e $s_2 = 3$.
Dato un nodo x, siano $y_1, y_2 \ldots, y_k$ i suoi figli ordinati in base al loro ordine di arrivo. Troviamo ora un limite inferiore su $s \rightarrow size$: sommiamo 1 per x stesso, 1 per il primo figlio y_1 e poi applichiamo via via il Lemma 1 per gli altri figli. Si ha quindi:

$$x \rightarrow size \geq s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

Resta ora da dimostrare per induzione su k che $s_k \geq F_{k+2}$. I casi di base (per $k = 0$ e $k = 1$) sono di verifica immediata.
Con $k \geq 2$ si prenda come ipotesi induttiva che $s_i \geq F_{i+2}$ per $i = 0, 1, 2, \ldots, k$.
Abbiamo quindi:

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=1}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2} \qquad \square$$

Abbiamo quindi dimostrato che:

$$x \rightarrow size \geq s_k \geq F_{k+2} \geq \phi^k$$

dove l'ultima disuguaglianza è data dal Lemma 3.
Con il Corollario 5 termina la parte relativa alla limitazione del grado massimo.

Corollario 5 *Il grado massimo $D(n)$ di un nodo di uno* Heap di Fibonacci *composto da n nodi é $O(logn)$.*

Dimostrazione: Sia x un nodo di heap di Fibonacci composto da n nodi e sia $x \rightarrow size = k$. Possiamo dire che $n \geq x \rightarrow size \geq \phi^k$ dato il Lemma 3. Utilizzando $\log_\phi$ si ottiene $\log_\phi n \geq \log_\phi \phi^k$, cioè $k \leq \log_\phi n$. Il grado massimo (quello che noi avevamo chiamato $D(n)$) di un qualunque nodo di uno heap di Fibonacci è dunque $O(\log n)$. $\square$

4 L'utilizzo degli Heap di Fibonacci nell'algoritmo di Prim

L'algoritmo di Prim, dato un grafo connesso, pesato e non orientato, trova il suo *mninimo albero di copertura* (=minimum spanning tree).
Prima di illustrare come gli heap di Fibonacci vengono utilizzati è indispensabile vedere come questo algoritmo lavora:

```
grafo G(N, A)  -> grafo                      (input)
albero T(N, A) -> minimo albero di copertura (init: vuoto)

PRIM(G) {
  considera T formato da un nodo e da nessun arco;

  while(esistono nodi in T adiacenti a un nodo non in T)
    seleziona l'arco di peso minimo che collega un nodo in T
       con uno non in T;
    aggiungi a T sia l'arco selezionato che il nodo che non
       era in T;

  restituisci T come il minimo albero di copertura di G;
}
```

L'algoritmo deve avere sempre a disposizione l'insieme degli archi[6] che collegano un nodo in T con uno non in T. Pertanto, ad un maggiore livello di dettaglio, la procedura dovrà interfacciarsi con una struttura dati per la gestione dell'insieme degli archi. La seguente pseudo-codifica rende esplicito l'utilizzo di tale struttura. Postponiamo invece, al momento, la sua implementazione, tenendo però presente che questa scelta influenza pesantemente le prestazioni dell'algoritmo:

```
grafo G(N, A)  -> grafo                      (input)
albero T(N, A) -> minimo albero di copertura (init: vuoto)
insieme I      -> insieme di archi che connettono  un nodo
                  non in T a un nodo in T
                                             (init: vuoto)

PRIM(G) {

  T->N = T->N + un nodo qualsiasi preso da G->N;
  I = I + tutti gli archi del nodo appena aggiunto;

  while (I != vuoto)
     estrai da I l'arco con peso minimo;
     elimina da I gli (eventuali) archi inutili (*);

     T->N = T->N + il nodo non appartenente a T fra i due
       congiunti dall'arco appena estratto;
     T->A = T->A + l'arco appena estratto;

     I = I + gli archi che collegano il nodo appena inserito
       in T con uno non in T;
```

[6] si ricorda che un arco non è altro che la tripla (nodo, nodo, peso)

```
    restituisci T come il minimo albero di copertura di G;
}
```

Ipotizziamo di avere un tipo di insieme in cui l'inserimento di un nuovo elemento costi $O(1)$ e l'estrazione del minimo costi $O(n)$ (come per esempio il banale caso di una lista). Il ciclo *while* verrebbe ripetuto $n-1$ volte ed a ogni iterazione la ricerca del minimo e l'eliminazione degli archi "inutili"[7] (*) costerebbe nel caso pessimo $O(n)$ (dovendo scandire tutta la lista). Inoltre l'inserimento degli archi del nodo appena aggiunto nell'albero, costerebbe anch'esso $O(n)$, in quanto verrebbero inseriti al più $n-1$ archi. Il costo totale sarebbe quindi $O(n^2)$. Quella che noi abbiamo chiamato brutalmente "eliminazione degli archi inutili" può essere evitata utilizzando un vettore di dimensione n che per ogni nodo non appartente a T tiene conto dell'arco presente in I che lo collega a T. In questo modo a ogni inserimento in I si controlla che non vi sia già nell'insieme un arco che collega T al dato nodo. Nel caso che questo esista viene sostituito dal nuovo arco solo se questo ha peso minore di quello già presente in I e, in tal caso, viene opportunamente aggiornato il vettore degli archi. Questa strategia permette di prevenire la presenza di archi "inutili" in tempo costante, mentre prima era necessario un tempo $O(n)$. Nella pratica comunque, la complessità dell'algoritmo resta $O(n^2)$, dal momento che abbiamo ipotizzato che l'estrazione del minimo costi $O(n)$.
Il metodo del vettore appena spiegato é utilizzato nell'implementazione dell'algoritmo di Prim con gli Heap di Fibonacci:

```
grafo G(N, A)      -> grafo                      (input)
albero T(N, A)     -> minimo albero di copertura (init: vuoto)
vettore FHtracer   -> vettore degli archi        (init: vuoto)
FibHeap FH         -> insieme di archi che connettono  un nodo
                      non in T a un nodo in T
                                                (init: vuoto)

PRIM(G) {

  T->N = T->N + un nodo qualsiasi preso da G->N;
  FH = FH + tutti gli archi del nodo appena aggiunto;
  aggiorna FHtracer;

  while(FH != vuoto)
      estrai da FH l'arco con peso minimo;
      T->A = T->A + l'arco appena estratto;
      T->N = T->N + il nodo non appartenente a T fra i due
        congiunti dall'arco appena estratto;
```

[7]Per arco inutile si intende l'arco che collega due nodi già appartenenti a T. La presenza di archi "inutili" è dovuta al fatto che in un dato momento possono essere presenti in I più archi comprendenti un dato nodo; quando tale nodo viene inserito in T, da I viene estratto un solo di questi archi mentre i restanti diventano per l'appunto "inutili".

```
        for each (arc = arco del nodo appena inserito in T)
          x = nodo congiunto da arc al nodo appena aggiunto a T;

          if(x non appartiene gia' a T)
           if(FHtracer[x] == NULL)
              FH->insert(arc);
              FHtracer[x] = arc;
           else
              FH->decrementaChiave(arc, FHtracer[x]);
              if(arc->chiave < FHtracer[x]->chiave)
                 FHtracer[x] = arc;

    restituisci T come il minimo albero di copertura di G;
}
```

La complessità dell'algoritmo con l'utilizzo degli heap di Fibonacci si abbassa a $O(m+n\log n)$ dove m è il numero degli archi del grafo: il ciclo *while* viene chiaramente ripetuto $n-1$ volte; l'estrazione dell'arco di peso minimo costa (vedi sottosezione *2.7*) $O(\log n)$ mentre il ciclo interno (quello effettuato per ogni arco di ogni nodo) costa dall'inizio alla fine della computazione, $O(m)$ in quanto in totale vengono eseguite $2m$ iterazioni, comprendenti o "inserimento" o "decremento di chiave", operazioni aventi entrambi costo ammortizzato costante (vedi sottosezione *2.4* e *2.8*).
É da notare però che l'utilizzo degli heap di Fibonacci nell'algoritmo di Prim fa dipendere la complessitá dal numero di archi del grafo; nel caso in cui il numero di archi sia $m \leq cn$ (dove c è una costante) la complessità risulta essere $O(n\log n)$; per grafi densi (o nel caso pessimo completi), in cui la complessità è $O(n^2)$, il loro utilizzo può essere sconveniente: si prenda per esempio il caso in cui $O(c_1 n^2)$ è il costo dell'algoritmo di Prim con l'utilizzo degli Heap di Fibonacci, $O(c_2 n^2)$ è il costo mediante l'utilizzo di un altro tipo di struttura dati e $c_2 \ll c1$.

Per renderne più chiaro il funzionamento in seguito è riportata una sequenza di figure che rappresentano i vari passaggi nel calcolo del minimo albero di copertura. In colore nero sono rappresentati gli archi e i nodi che fanno parte dell'albero di copertura, mentre in grigio quelli che non ne fanno parte; tratteggiati sono invece gli archi inseriti nello heap di Fibonacci.

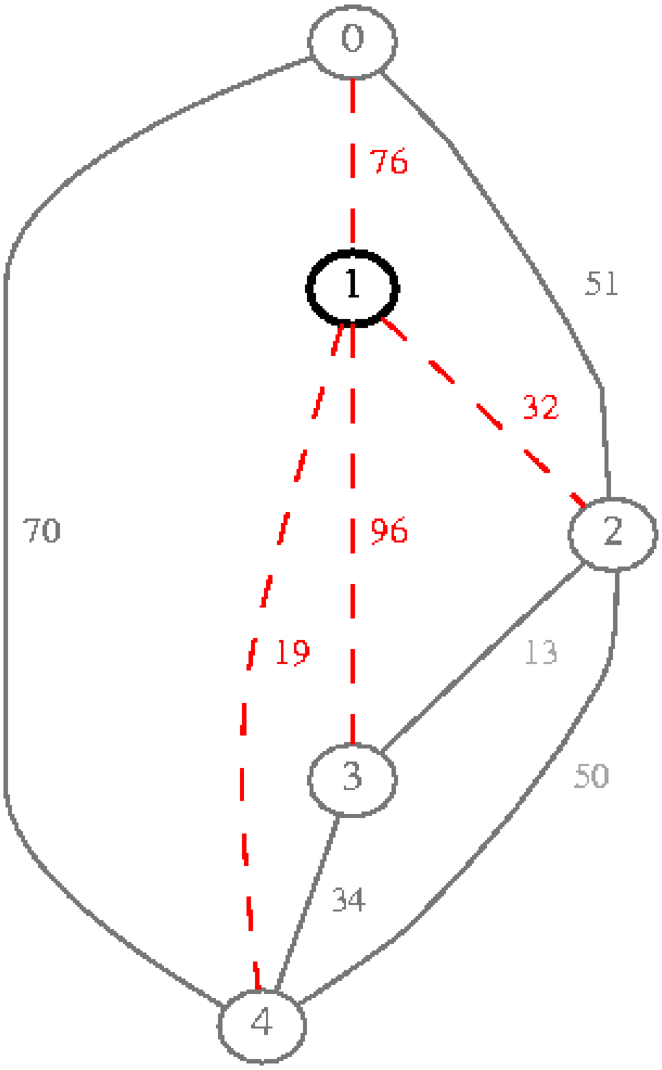


Figura 5: Come nodo di partenza per il calcolo del minimo albero di copertura è stato scelto (a caso) il nodo con chiave 1; tutti i suoi nodi sono stati inseriti nello heap.

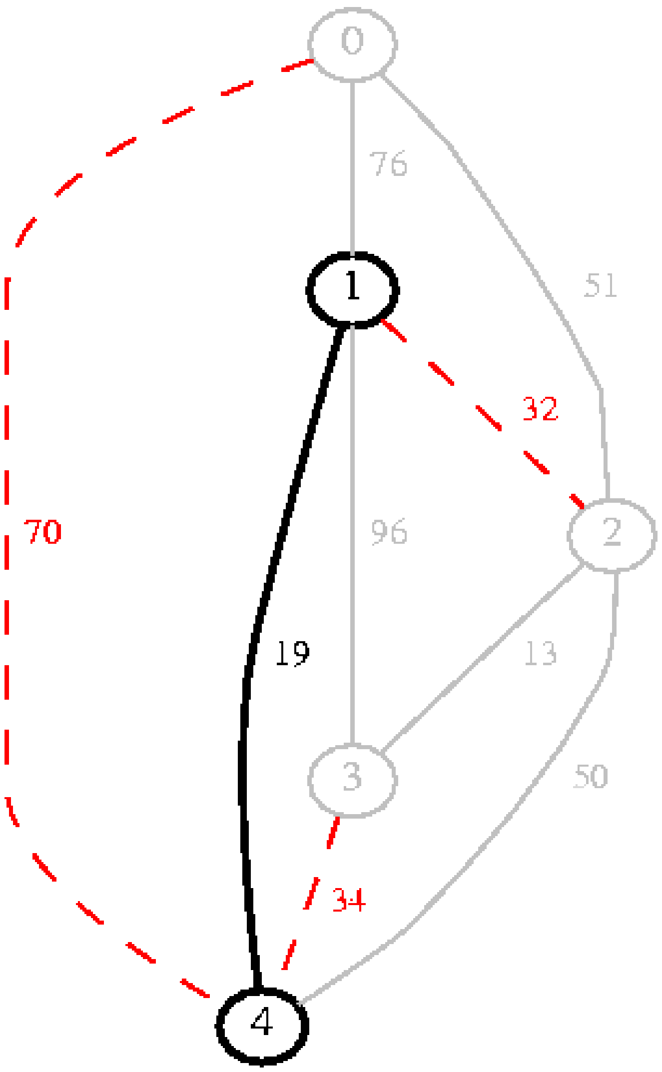


Figura 6: Fra gli archi presenti nello heap viene scelto quello con peso minore cioè l'arco *1-4* e che ha peso *19*; nell'albero vengono inseriti quindi l'arco *1-4* e il nodo *4*; nello heap andrebbero inseriti gli archi del nodo *4* che non appartengono già all'albero; essendo già presenti nello heap altri archi che arrivano ai nodi *0*, *2* e *3*, vengono effettuati dei "decrementi di chiave" e gli archi *1-0* e *1-3* vengono sostituiti rispettivamente con *4-0* e *4-3* in quanto hanno peso minore dei precedenti, mentre l'arco *1-2* non viene sostituito da *4-2*.

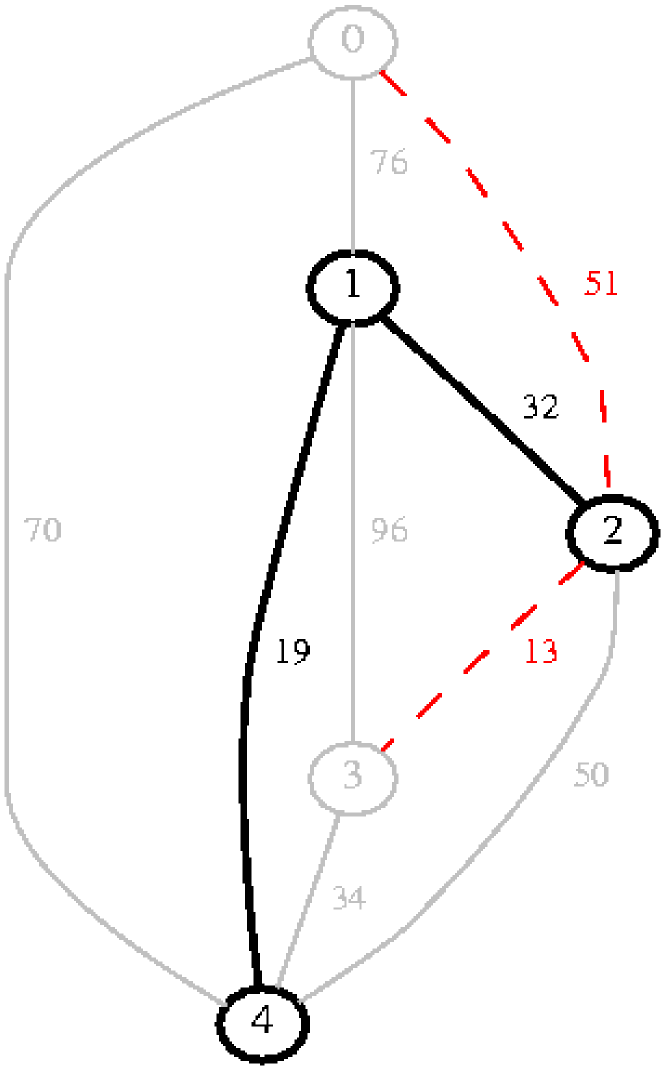


Figura 7: Dallo heap viene scelto l'arco *1-2*; nell'albero viene inserito il nodo *2* e l'arco *1-2*; nello heap gli archi *4-0* e *4-3* vengono sostituiti rispettivamente da *2-0* e *2-3* in quanto hanno peso minore.

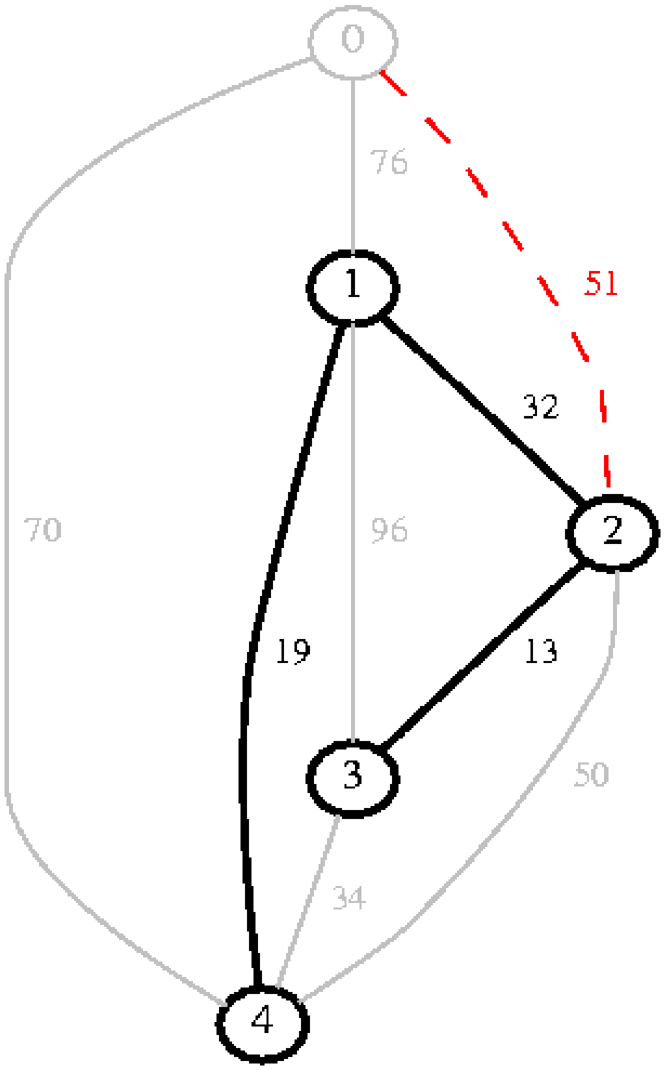


Figura 8: Dallo heap viene scelto l'arco *2-3*; nell'albero viene inserito il nodo *3* e l'arco *2-3*; nello heap non viene inserito nessun arco in quanto tutti gli archi del nodo *3* lo congiungono con nodi già appartenti all'albero.

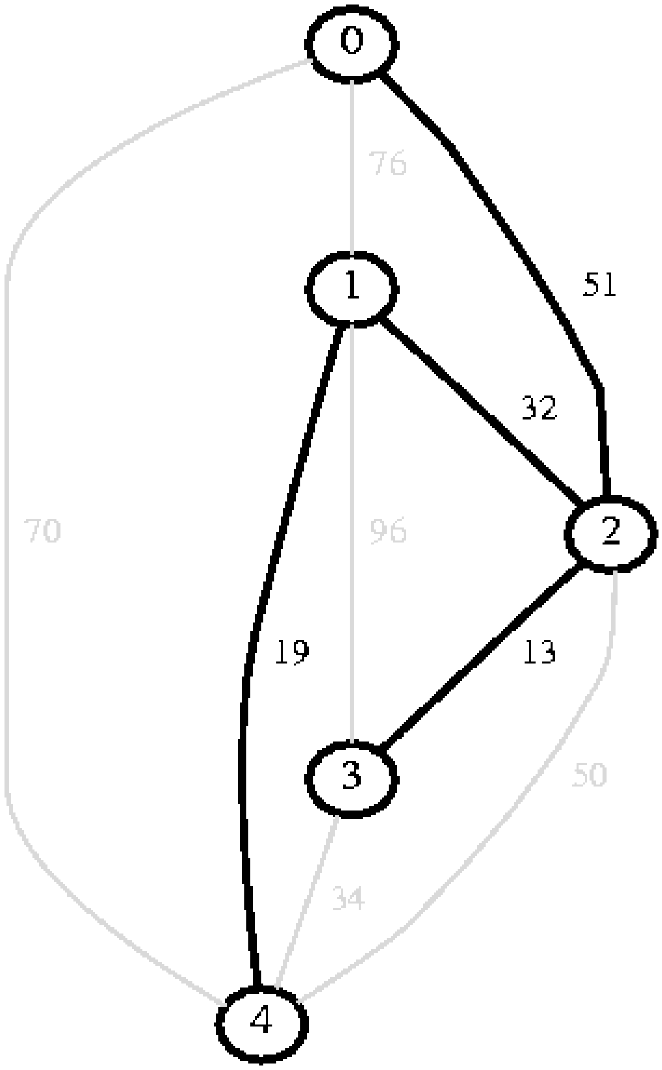


Figura 9: Dallo heap viene scelto l'arco *2-0*; nell'albero viene inserito il nodo *0* e l'arco *2-0*; lo heap è vuoto, di conseguenza il calcolo del minimo albero di copertura é terminato.