POLITECNICO
DI TORINO
Dipartimento
di Automatica e Informatica

Code a Priorità e Heap

Gianpiero Cabodi e Paolo Camurati

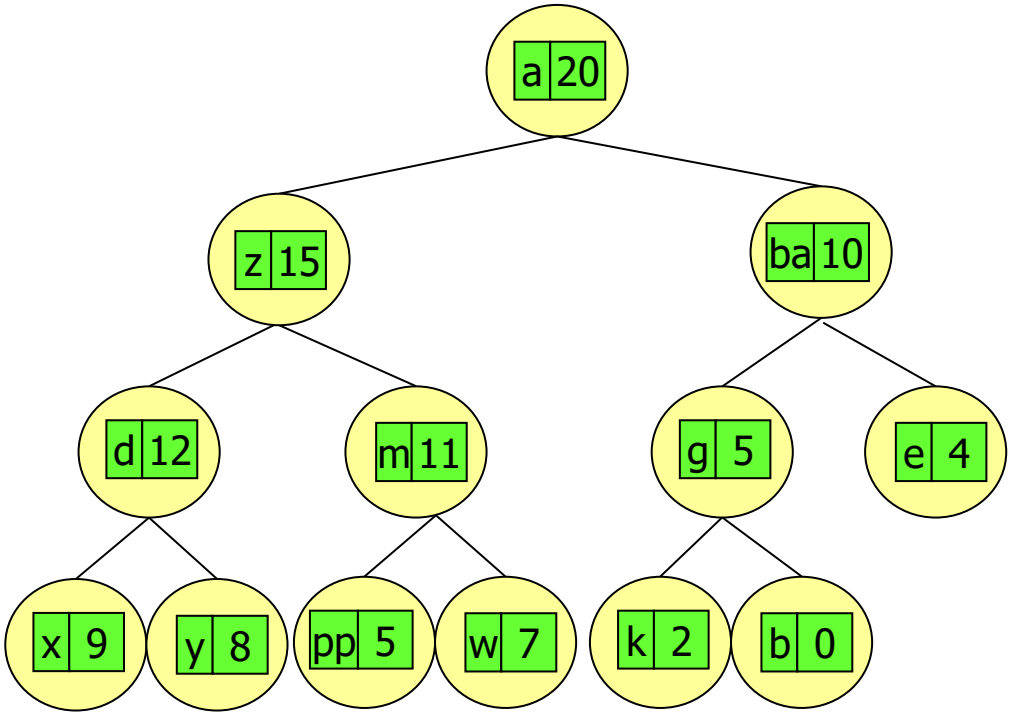
ADT Heap

- Definizione: albero binario con
  - proprietà strutturale: quasi completo (tutti i livelli completi, tranne eventualmente l'ultimo, riempito da SX a DX) $\Rightarrow$ quasi bilanciato
  - proprietà funzionale:

    $$\forall\ i \neq r \quad key(parent(i)) \geq key(i)$$

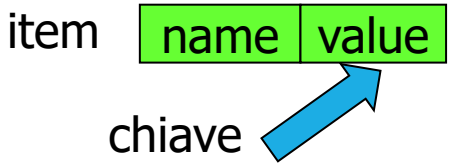
  - conseguenza: chiave max nella radice
- Implementazione: mediante vettore.

Esempio



Item

- Quasi ADT Item
- Dati:
  - Nome (stringa), valore (intero)
  - Chiave = valore
  - Tipologia 3



ADT di I classe Heap

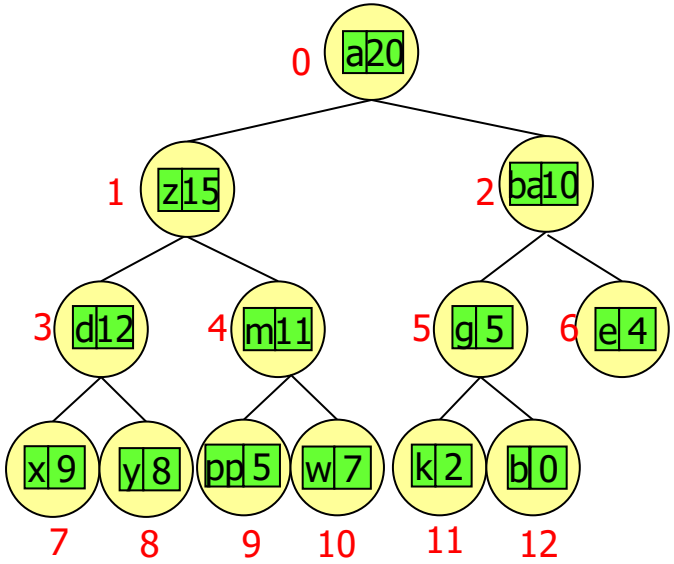
Heap.h

```
typedef struct heap *Heap;

Heap   HEAPinit(int maxN);
Void   HEAPfree(Heap h);
void   HEAPfill(Heap h, Item val);
void   HEAPsort(Heap h);
void   HEAPdisplay(Heap h);
```

Implementazione

- Struttura dati: vettore di Item h->A[0..maxN-1]
- h->heapsize: numero di elementi in heap h->A
- radice in h->A[0]
- dato h->A[i]:
  - il figlio SX è h->A[LEFT(i)] dove LEFT(i)= 2i+1
  - il figlio DX è h->A[RIGHT(i)] dove RIGHT(i) = 2i+2
  - il padre è h->A[PARENT(i)] dove PARENT(i)=(i-1)/2

Esempio



maxN = 15

h->A

a	z	ba	d	m	g	e	x	y	pp	w	k	b		
20	15	10	12	11	5	4	9	8	5	7	2	0		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

h->heapsize 13

Heap.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "Heap.h"

struct heap { Item *A; int heapsize; };

int LEFT(int i) { return (i*2 + 1); }
int RIGHT(int i) { return (i*2 + 2); }
int PARENT(int i) { return ((i-1)/2); }

Heap HEAPinit(int maxN) {
  Heap h;
  h = malloc(sizeof(*h));
  h->A = malloc(maxN*sizeof(Item));
  h->heapsize = 0;
  return h;
}
```

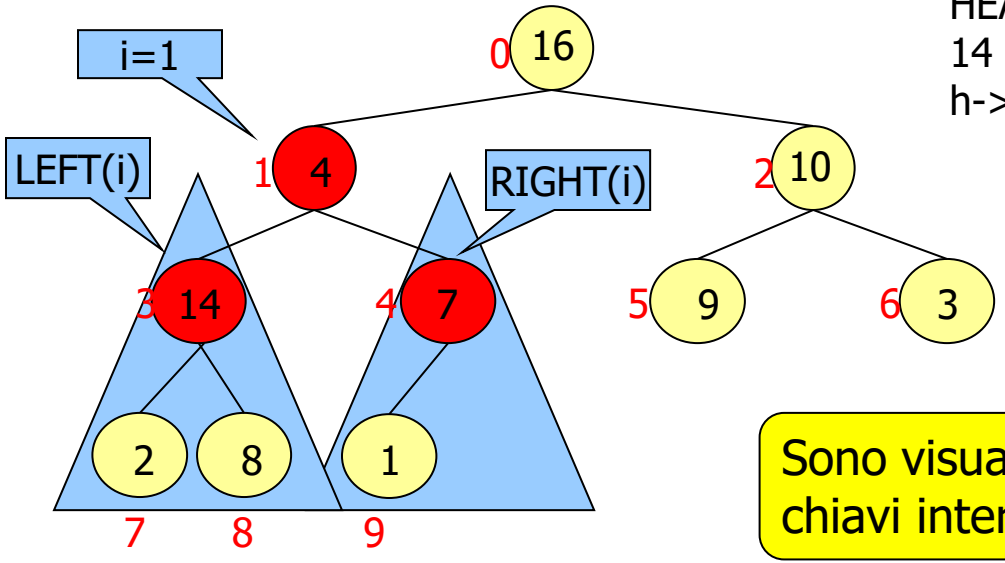
usata per inserire valori,
non necessariamente
il risultato sarà uno heap

```
void HEAPfree(Heap h) {
  free(h->A);
  free(h);
}

void HEAPfill(Heap h, Item item) {
  int i;
  i  = h->heapsize++;
  h->A[i] = item;
  return;
}

void HEAPdisplay(Heap h) {
  int i;
  for (i = 0; i < h->heapsize; i++)
    ITEMstore(h->A[i]);
}
```

Funzione HEAPify

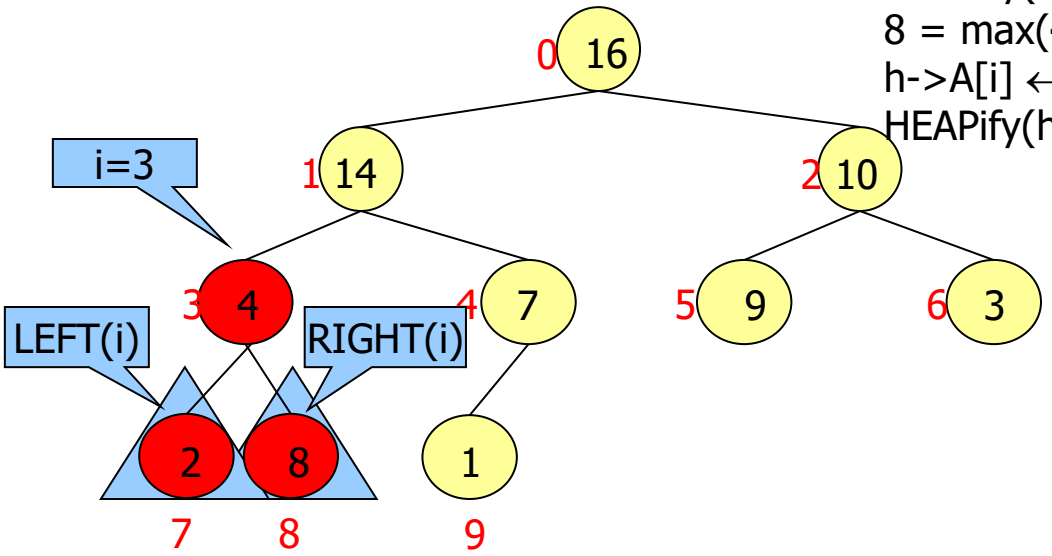
- Trasforma in heap i, LEFT(i), RIGHT(i), dove LEFT(i) e RIGHT(i) sono già heap
- assegna ad A[i] il max tra A[i], A[LEFT(i)] e A[RIGHT(i)]
- se c'è stato scambio A[i] $\leftrightarrow$ A[LEFT(i)], applica ricorsivamente HEAPify su sottoalbero con radice LEFT(i)
- analogamente per scambio A[i] $\leftrightarrow$ A[RIGHT(i)].
- Complessità: $T(n) = O(\lg n)$.

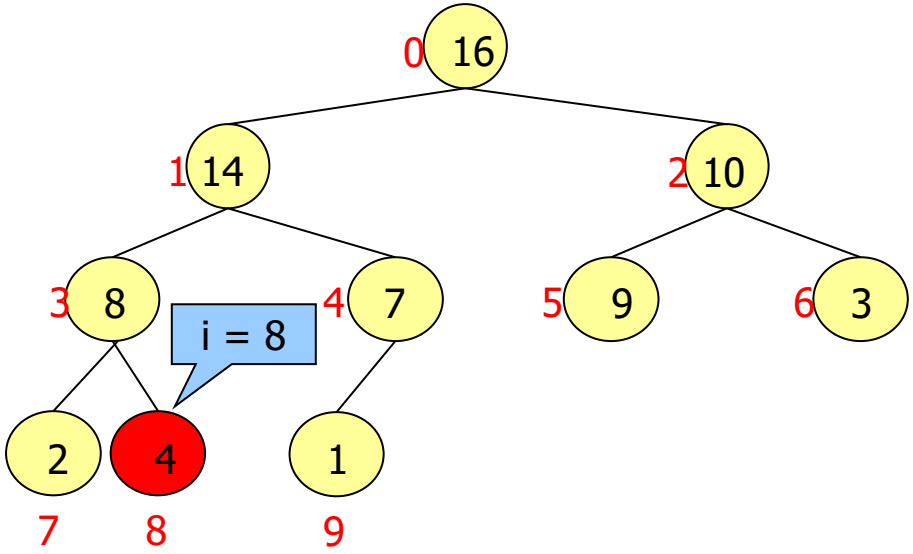
Esempio



HEAPify(h->A, 1)
14 = max(4, 14, 7)
h->A[i] ↔ h->A[LEFT(i)]

Sono visualizzate solo le chiavi intere, non gli item


HEAPify(h->A, 3)
8 = max(4, 2, 8)
h->A[i] ↔ h->A[RIGHT(i)]
HEAPify(h->A, 8)
0 16
i=3
1 14
2 10
3 4
4 7
5 9
6 3
LEFT(i)
RIGHT(i)
2
8
1
7
8
9



HEAPify(h->A, 8)
foglia
terminazione.

```
void HEAPify(Heap h, int i) {
  int l, r, largest;
  l = LEFT(i);
  r = RIGHT(i);
  if ((l<h->heapsize) &&
       KEYcmp(KEYget(h->A[l]),KEYget(h->A[i]))==1)
    largest = l;
  else
    largest = i;
  if ((r<h->heapsize) &&
       KEYcmp(KEYget(h->A[r]), KEYget(h->A[largest]))==1)
    largest = r;
  if (largest != i) {
    Swap(h, i,largest);
    HEAPify(h, largest);
  }
}
```

Funzione HEAPbuild

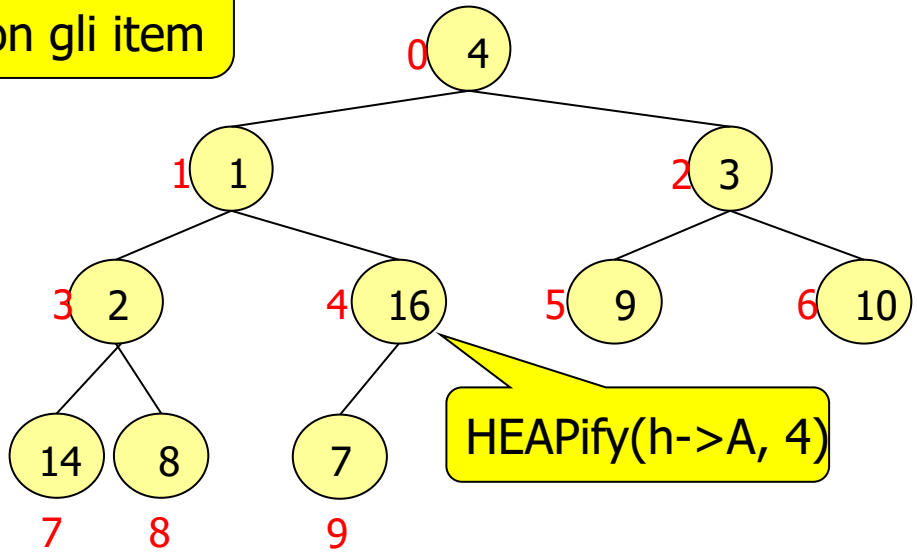
- Trasforma un albero binario memorizzato in vettore A in uno heap:
  - le foglie sono heap
  - applica HEAPify a partire dal padre dell'ultima foglia o coppia di foglie fino alla radice.

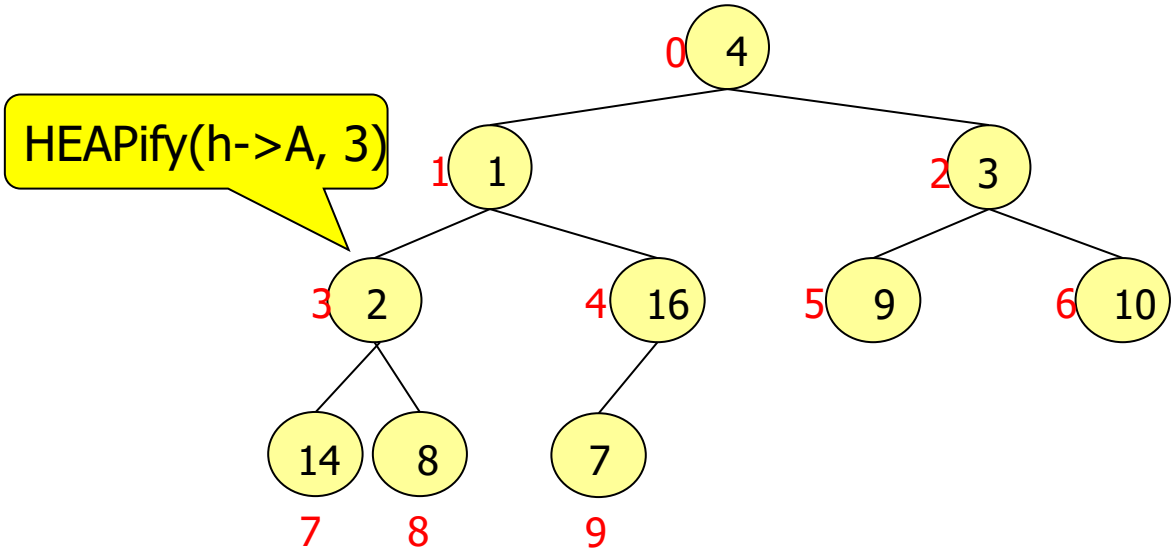
```
void HEAPbuild (Heap h) {
  int i;
  for (i=PARENT(h->heapsize-1); i >= 0; i--)
    HEAPify(h, i);
}
```

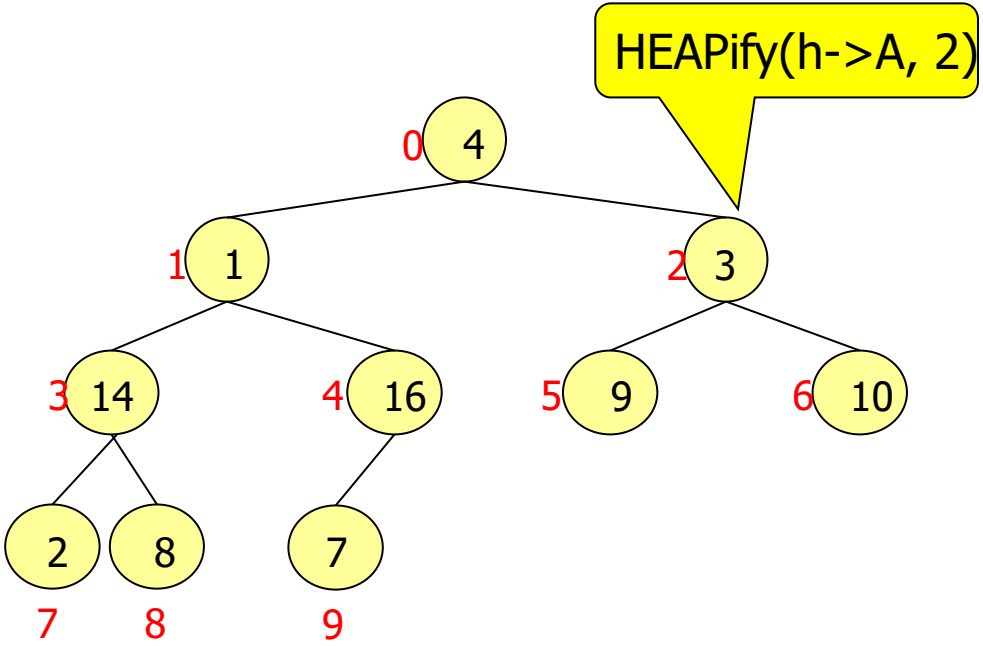
Esempio

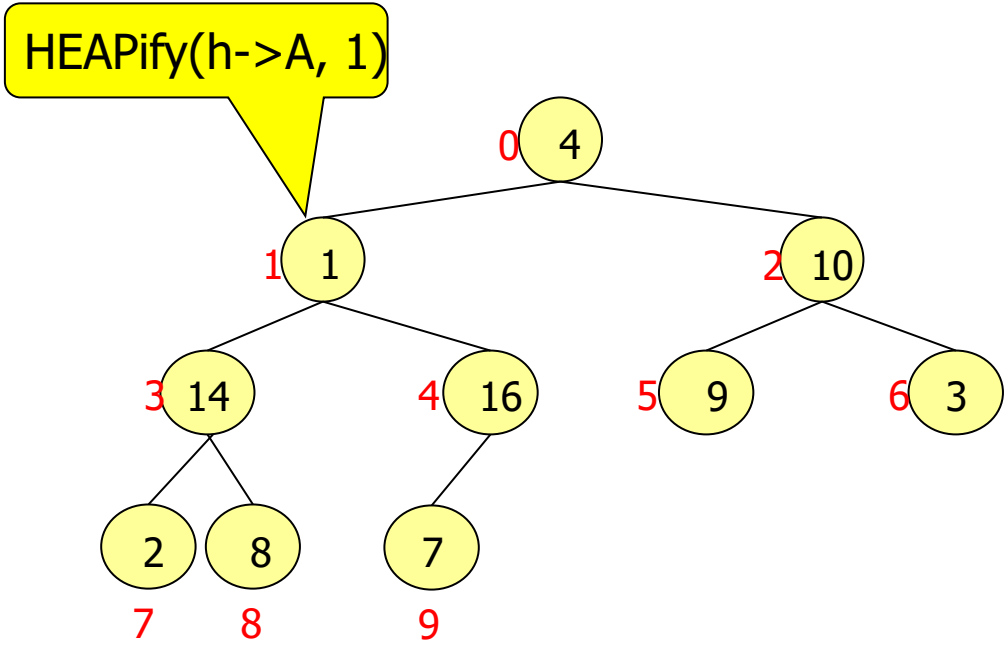
HEAPbuild(h->A)

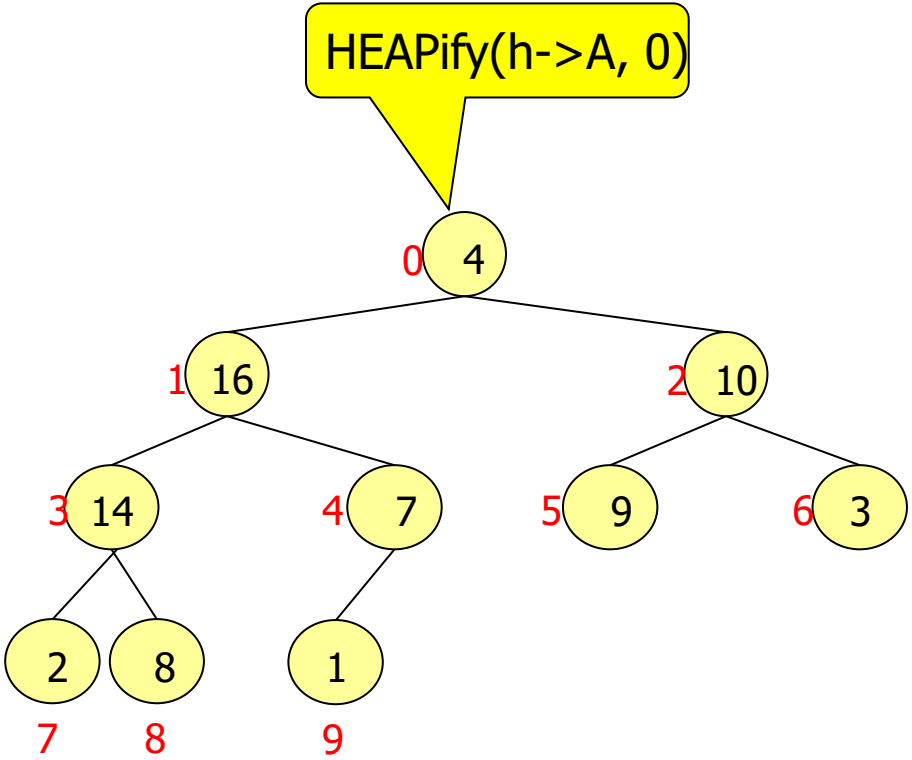
Sono visualizzate solo le chiavi intere, non gli item

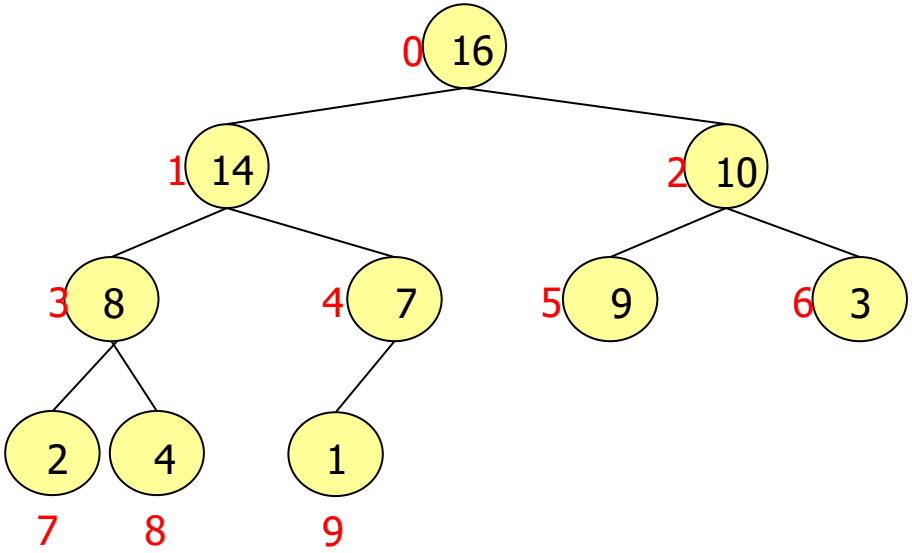



HEAPify(h->A, 3)
0
4
1
1
2
3
3
2
4
16
5
9
6
10
14
8
7
7
8
9


HEAPify(h->A, 2)
0
4
1
1
2
3
3
14
4
16
5
9
6
10
2
8
7
7
8
9


HEAPify(h->A, 1)
0
4
1
1
2
10
3
14
4
16
5
9
6
3
2
8
7
7
8
9


HEAPify(h->A, 0)
0
4
1
16
2
10
3
14
4
7
5
9
6
3
2
8
1
7
8
9


0
16
1
14
2
10
3
8
4
7
5
9
6
3
2
4
1
7
8
9

- Analisi di complessità:
  - intuitiva ed imprecisa: n passi ciascuno di costo logn, quindi T(n) = O(n lg n)
  - precisa: T(n) = O(n).

Risoluzione per sviluppo (unfolding).

$T(n) = 2T(n/2) + \log_2(n)$

$T(n/2) = 2T(n/4) + \log_2(n/2)$

$T(n/4) = 2T(n/8) + \log_2(n/4)$

2 sottoalberi

Heapify

Sostituendo in T(n):

$\sum_{i=0}^{k} ix^i = x\,(1-(k+1)x^k+kx^{k+1})/(1-x)^2$

$T(n) = \sum_{i=0}^{\log_2 n} 2^i \log_2(n/2^i)$

$= \log_2 n \sum_{i=0}^{\log_2 n} 2^i - \sum_{i=0}^{\log_2 n} i2^i$

$= \log_2 n(2n-1) - 2(1-(\log_2 n+1)n+2n\log_2 n)$

$= 2n - \log_2 n - 2$

$= O(n)$

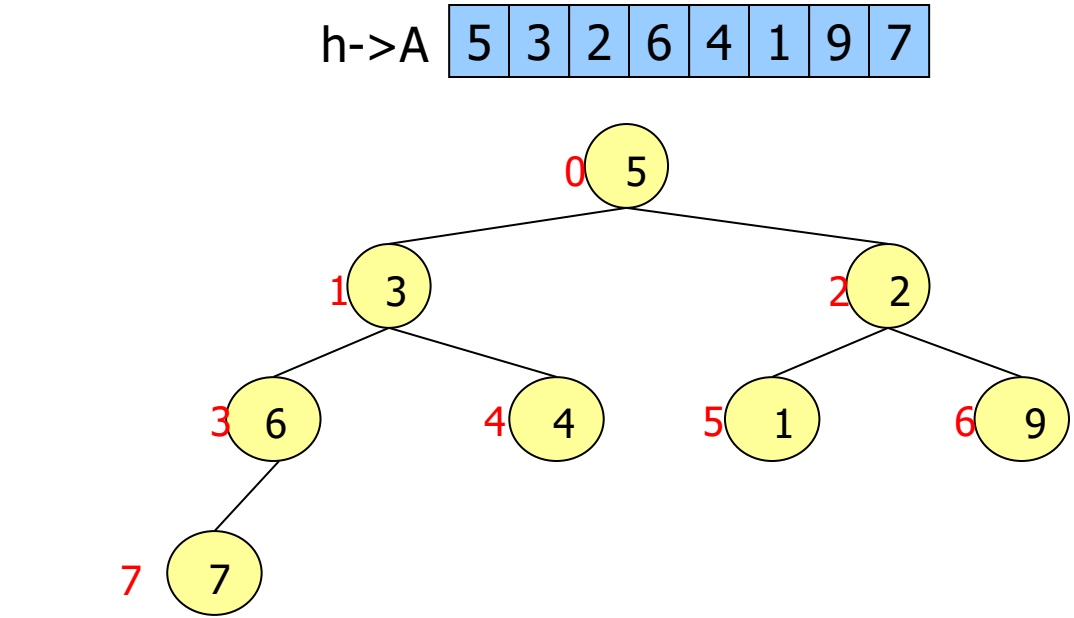
Funzione HEAPsort

- Trasforma il vettore in uno heap mediante HEAPbuild
- Scambia il primo e ultimo elemento
- Riduci la dimensione dello heap di 1
- Ripristina la proprietà di heap
- Ripeti fino a esaurimento dello heap.
- Caratteristiche:
  - complessità: T(n)= O(n lg n).
  - in loco
  - non stabile

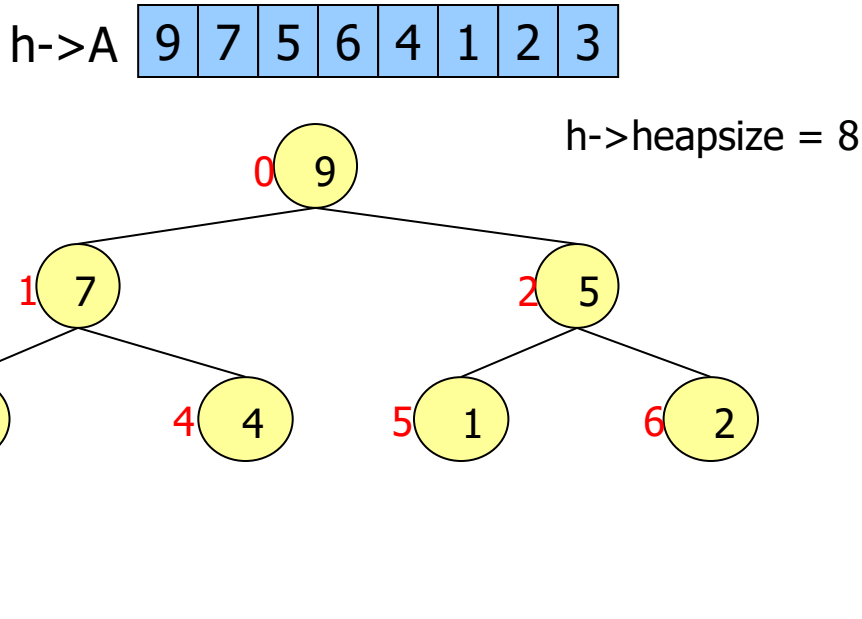
Esempio

Sono visualizzate solo le chiavi intere, non gli item

Configurazione iniziale



Applico HEAPbuild

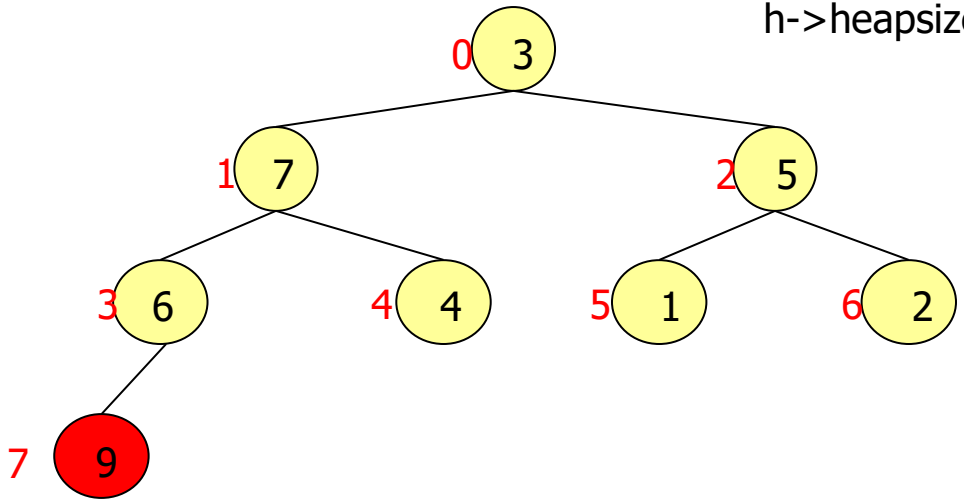


h->A[0] ↔ h->A[h->heapsize-1]
h->heapsize--

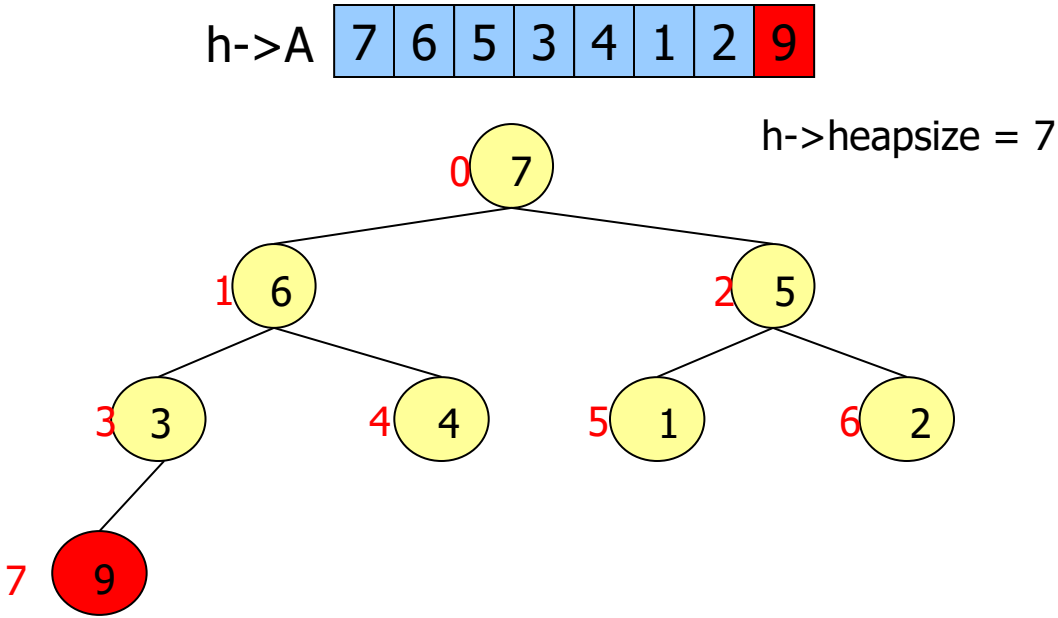
h->A

3	7	5	6	4	1	2	9

h->heapsize = 7

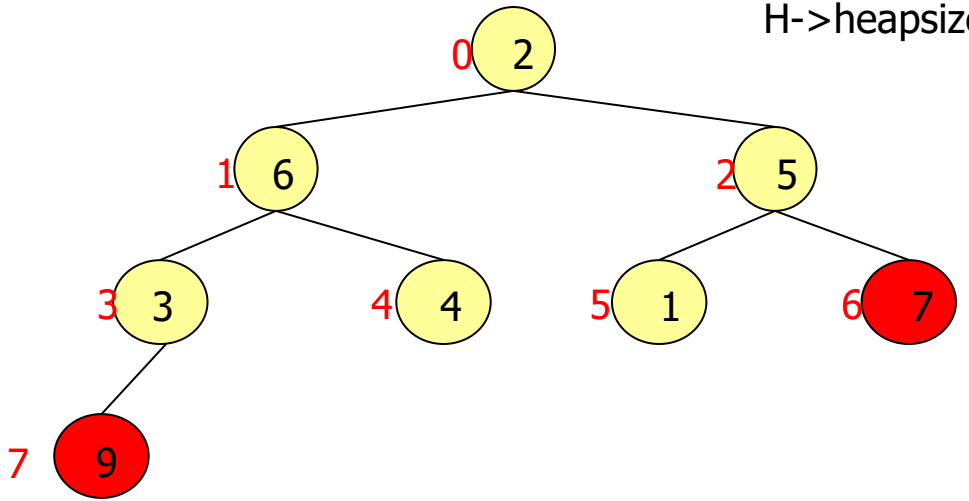


HEAPify(h->A, 0)



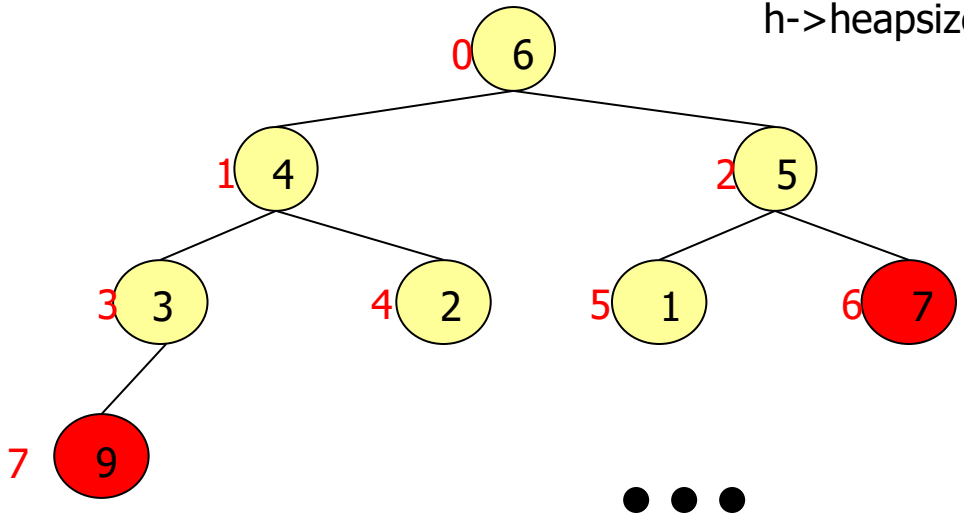
h->A[0] ↔ h->A[h->heapsize-1]
h->heapsize--

H->heapsize = 6



HEAPify(h->A, 0)

h->heapsize = 6



```
void HEAPsort(Heap h) {
  int i, j;
  HEAPbuild(h);
  j = h->heapsize;
  for (i = h->heapsize-1; i > 0; i--) {
    Swap (h,0,i);
    h->heapsize--;
    HEAPify(h,0);
  }
  h->heapsize = j;
}
```

Coda a priorità

Definizione:

- struttura dati PQ per mantenere un set di elementi di tipo Item, ciascuno dei quali include un campo priorità
- operazioni principali: inserzione, estrazione del massimo, lettura del massimo, cambio di priorità.

ADT di I classe Coda a Priorità

PQ.h

```
typedef struct pqueue *PQ;

PQ      PQinit(int maxN);
void    PQfree(PQ pq);
int     PQempty(PQ pq);
void    PQinsert(PQ pq, Item val);
Item    PQextractMax(PQ pq);
Item    PQshowMax(PQ pq);
void    PQdisplay(PQ pq);
int     PQsize(PQ pq);
void    PQchange(PQ pq, Item val);
```

discussione a parte

Implementazione della struttura dati:

- vettore/lista non ordinato
- vettore/lista ordinato
- heap di dati/indici.

} non considerati qui, cfr Tipi di Dato Astratto

Complessità

	PQinsert	PQshowMax	PQextractMax
Vettore non ordinato	1	N	N
Lista non ordinata	1	N	N
Vettore ordinato	N	1	1
Lista ordinata	N	1	1
Heap di item/indici	logN	1	logN

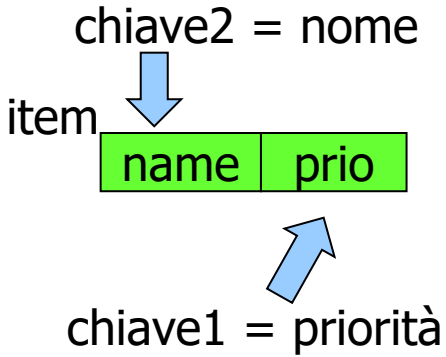
max in coda

max in testa

Cosa contiene l'ADT Coda a priorità?

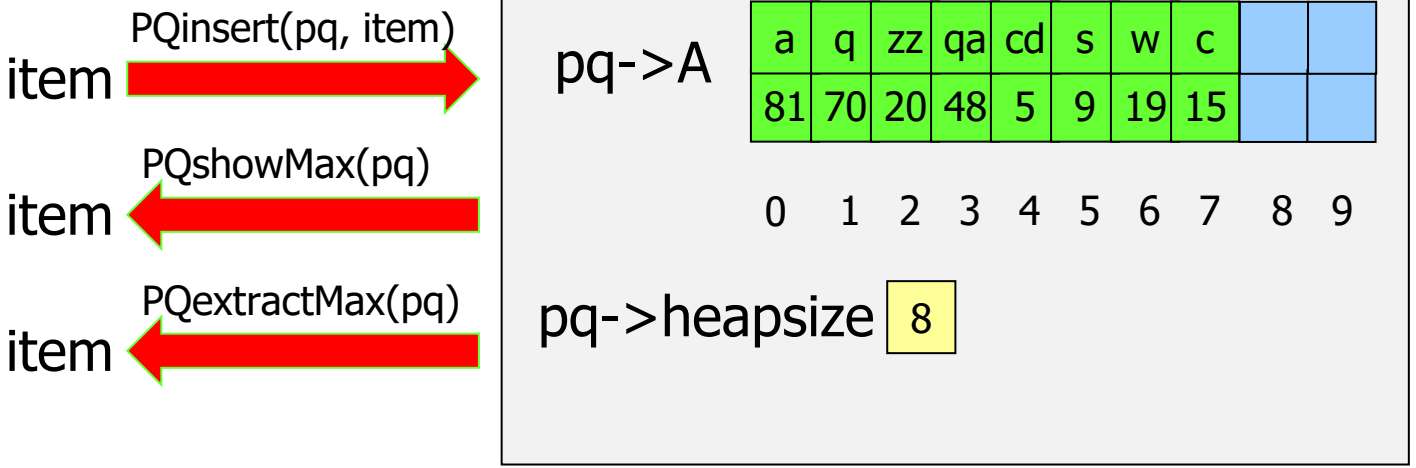
I soluzione: la coda a priorità contiene dati (lo heap che realizza la coda a priorità contiene i dati), l'ADT è una struct con:

1. la coda a priorità: vettore (heap) pq->A di dati di tipo Item (quasi ADT, tipologia 3)
2. heapsize: intero.



Utente

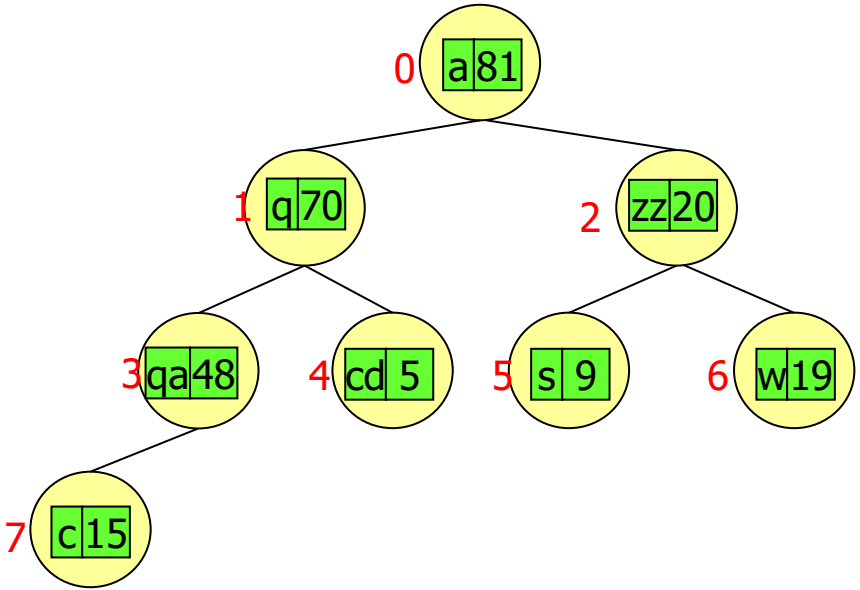
ADT I cat. coda a priorità di dati



pq->A

a	q	zz	qa	cd	s	w	c		
81	70	20	48	5	9	19	15		
0	1	2	3	4	5	6	7	8	9

pq->heapsize 8



ADT di I cat. Coda a priorità

PQ.c

```
#include <stdlib.h>
#include "Item.h"
#include "PQ.h"

struct pqueue { Item *A; int heapsize; };

static int LEFT(int i) { return (i*2 + 1); }
static int RIGHT(int i) { return (i*2 + 2); }
static int PARENT(int i) { return ((i-1)/2); }

PQ PQinit(int maxN){
  PQ pq = malloc(sizeof(*pq));
  pq->A = (Item *)malloc(maxN*sizeof(Item));
  pq->heapsize = 0;
  return pq;
}
```

```
void PQfree(PQ pq){
  free(pq->A);
  free(pq);
}

int PQempty(PQ pq) { return pq->heapsize == 0; }

int PQsize(PQ pq) { return pq->heapsize; }

Item PQshowMax(PQ pq) { return pq->A[0]; }

void PQdisplay(PQ pq) {
  int i;
  for (i = 0; i < pq->heapsize; i++)
    ITEMstore(pq->A[i]);
}
```

Funzione PQinsert

- Aggiunge una foglia all'albero (cresce per livelli da SX a DX, rispettando la proprietà strutturale)
- Risale dal nodo corrente (inizialmente la foglia appena creata) fino al più alla radice. Confronta la chiave del dato contenuto nel padre con la chiave del dato da inserire, facendo scendere il dato del padre nel figlio se la chiave da inserire è maggiore, altrimenti inserisce il dato nel nodo corrente.

Complessità: $T(n) = O(\lg n)$.

```
void PQinsert (PQ pq, Item val) {
  int i;
  i  = pq->heapsize++;
  while((i>=1) &&
      (KEYcmp(KEYget(pq->A[PARENT(i)]),KEYget(val))==-1)){
    pq->A[i] = pq->A[PARENT(i)];
    i = PARENT(i);
  }
  pq->A[i] = val;
  return;
}
```

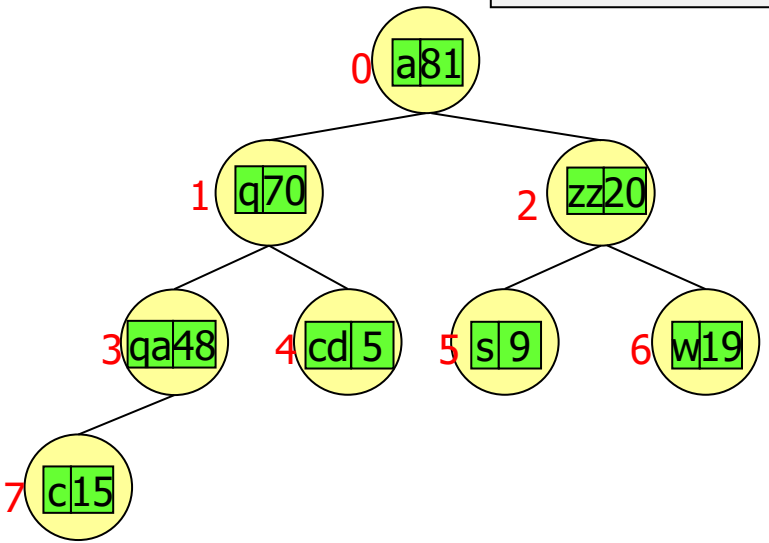
Esempio

Inserzione di r 75

pq->A

a	q	zz	qa	cd	s	w	c		
81	70	20	48	5	9	19	15		
0	1	2	3	4	5	6	7	8	9

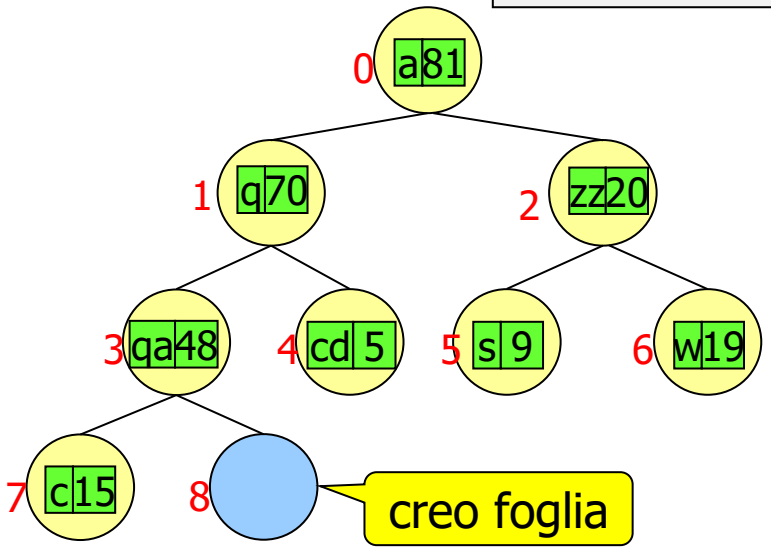
pq->heapsize 8



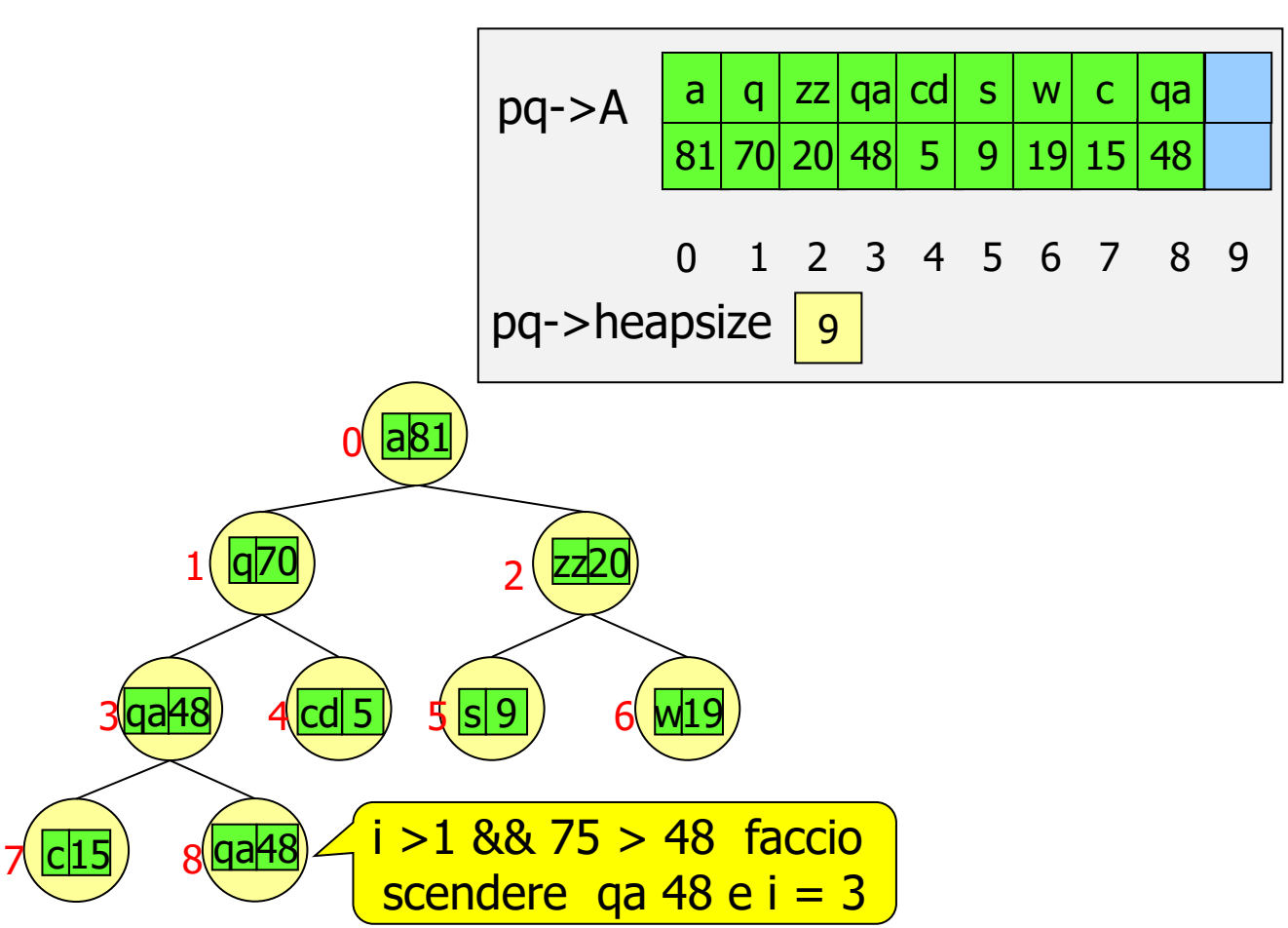
pq->A

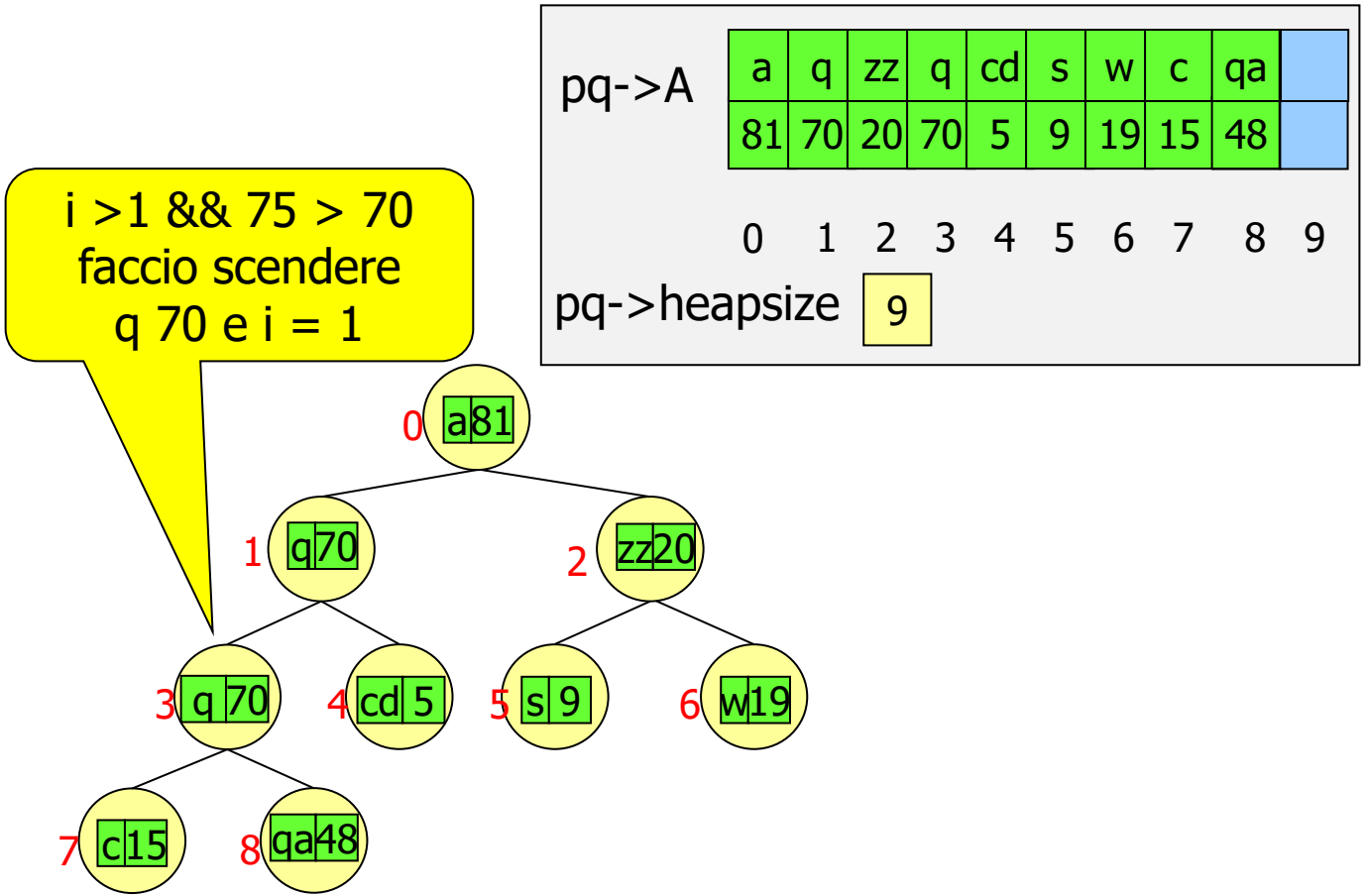
a	q	zz	qa	cd	s	w	c		
81	70	20	48	5	9	19	15		
0	1	2	3	4	5	6	7	8	9

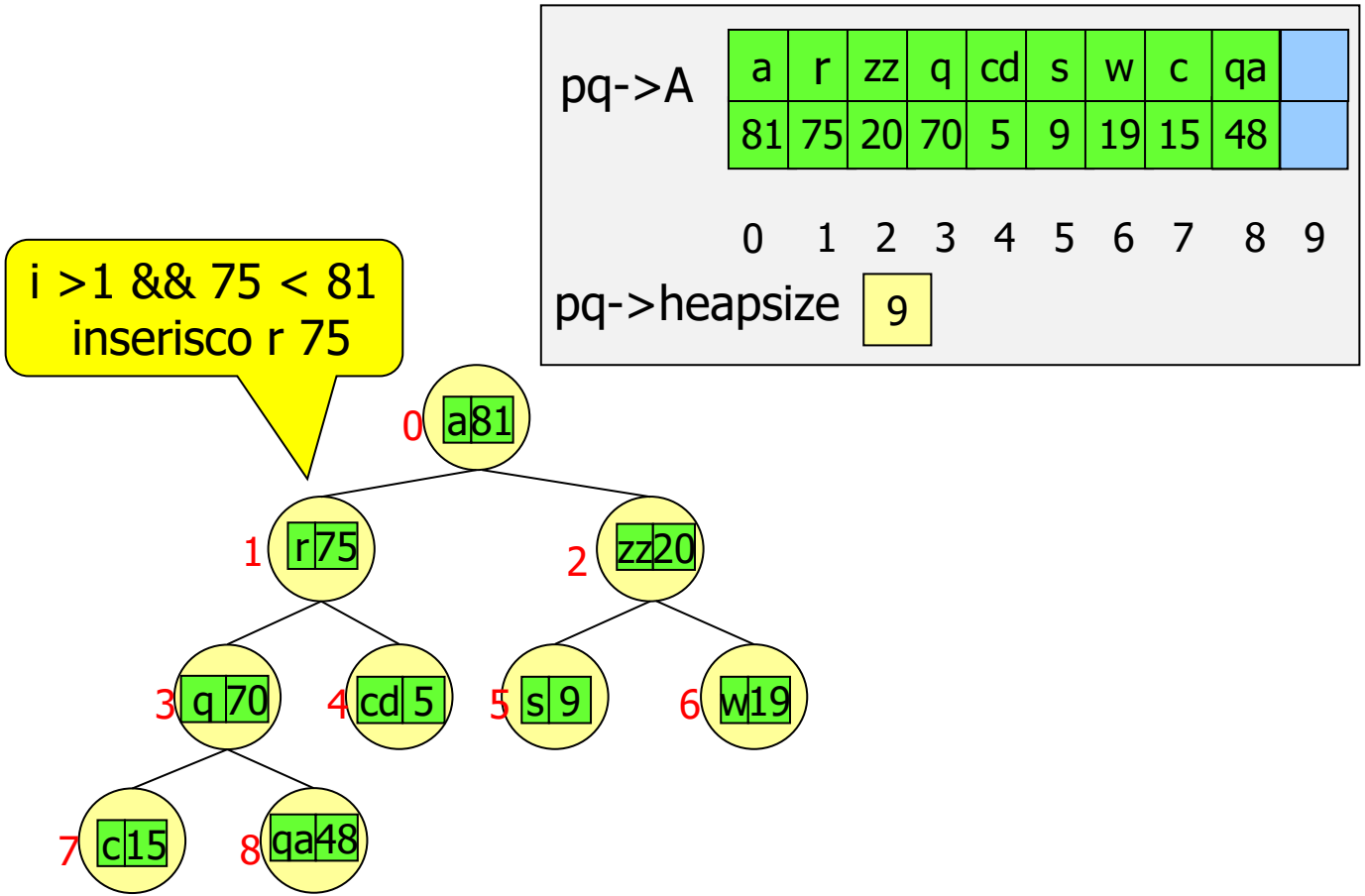
pq->heapsize 9



i = 8


pq->A
a q zz qa cd s w c qa
81 70 20 48 5 9 19 15 48
0 1 2 3 4 5 6 7 8 9
pq->heapsize
9
0 a 81
1 q 70
2 zz 20
3 qa 48
4 cd 5
5 s 9
6 w 19
7 c 15
8 qa 48
i >1 && 75 > 48 faccio
scendere qa 48 e i = 3


pq->A
a q zz q cd s w c qa
81 70 20 70 5 9 19 15 48
0 1 2 3 4 5 6 7 8 9
pq->heapsize
9
i >1 && 75 > 70
faccio scendere
q 70 e i = 1
0 a 81
1 q 70
2 zz 20
3 q 70
4 cd 5
5 s 9
6 w 19
7 c 15
8 qa 48


pq->A
a r zz q cd s w c qa
81 75 20 70 5 9 19 15 48
0 1 2 3 4 5 6 7 8 9
pq->heapsize
9
i >1 && 75 < 81
inserisco r 75
0 a 81
1 r 75
2 zz 20
3 q 70
4 cd 5
5 s 9
6 w 19
7 c 15
8 qa 48

Funzione PQextractMax

- Modifica lo heap, estraendone il valore massimo, che è contenuto nella radice:
  - scambia la radice con l'ultima delle foglie (quella più a destra nell'ultimo livello)
  - riduce di 1 della dimensione dello heap
  - ripristina le proprietà dello heap mediante applicazione di HEAPify.

Complessità: $T(n) = O(\lg n)$.

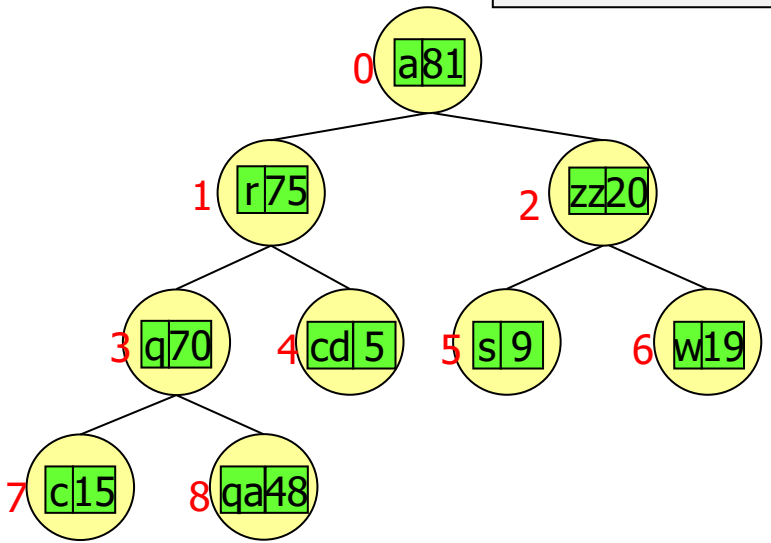
```
Item PQextractMax(PQ pq) {
  Item val;
  Swap (pq, 0,pq->heapsize-1);
  val = pq->A[pq->heapsize-1];
  pq->heapsize--;
  HEAPify(pq, 0);
  return val;
}
```

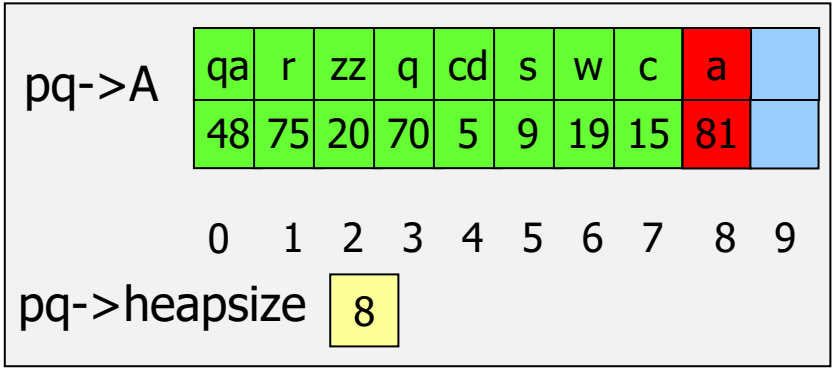
Esempio

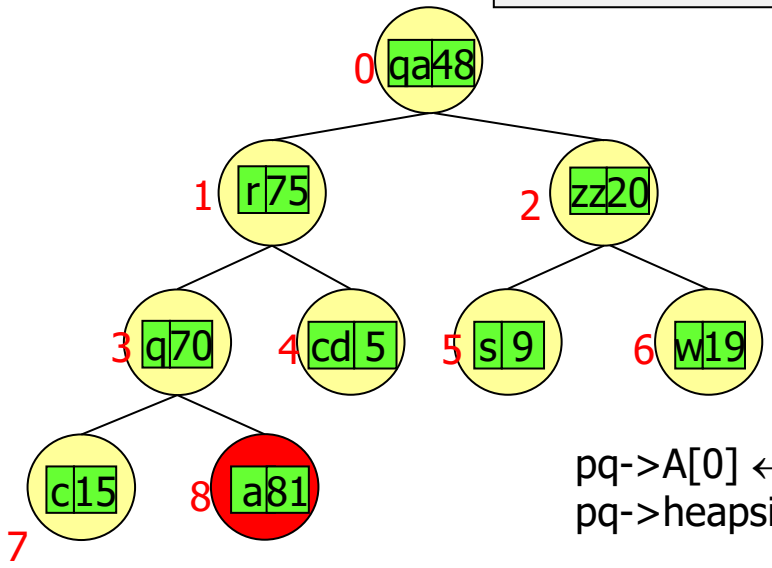
pq->A

a	r	zz	q	cd	s	w	c	qa	
81	75	20	70	5	9	19	15	48	
0	1	2	3	4	5	6	7	8	9

pq->heapsize 9







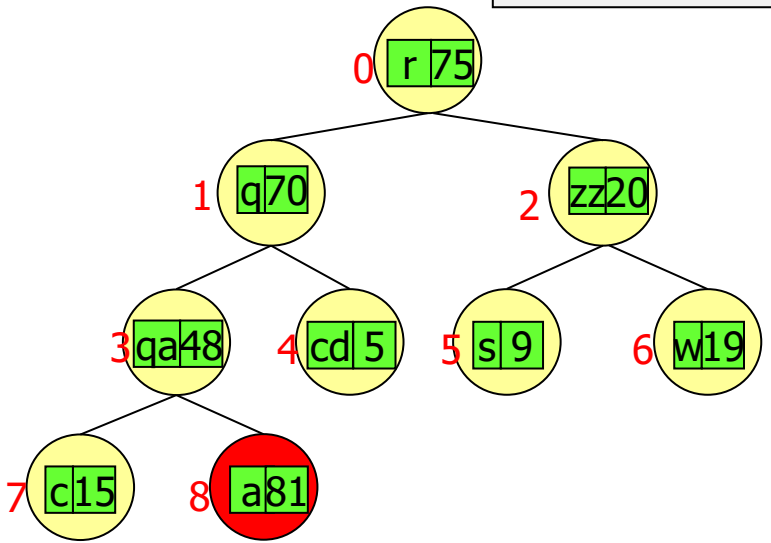
pq->A[0] ↔ pq->A[pq->heapsize-1]
pq->heapsize--

HEAPify(pq->A, 0)

pq->A

r	q	zz	qa	cd	s	w	c	a	
75	70	20	48	5	9	19	15	81	
0	1	2	3	4	5	6	7	8	9

pq->heapsize 8



Funzione PQchange

- Modifica la priorità di un elemento, la cui **posizione** (indice nello heap) viene calcolata con una scansione di costo lineare
- O risale dalla posizione data fino al più alla radice confrontando la chiave del padre con la chiave modificata, facendo scendere la chiave del padre nel figlio se la chiave modificata è maggiore, altrimenti la inserisce nel nodo corrente
- O applica HEAPify a partire dalla posizione data.

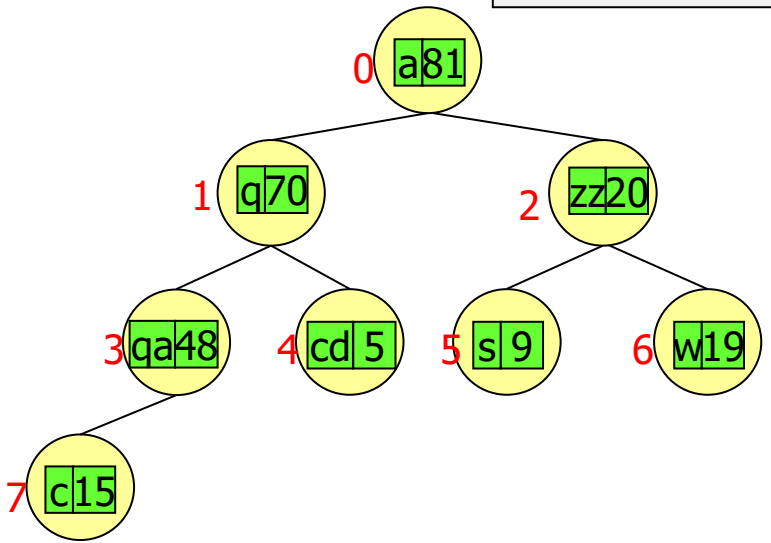
Complessità: $T(n) = O(n) + O(\lg n) = O(n)$.

Esempio

pq->A

a	q	zz	qa	cd	s	w	c		
81	70	20	48	5	9	19	15		
0	1	2	3	4	5	6	7	8	9

pq->heapsize 8

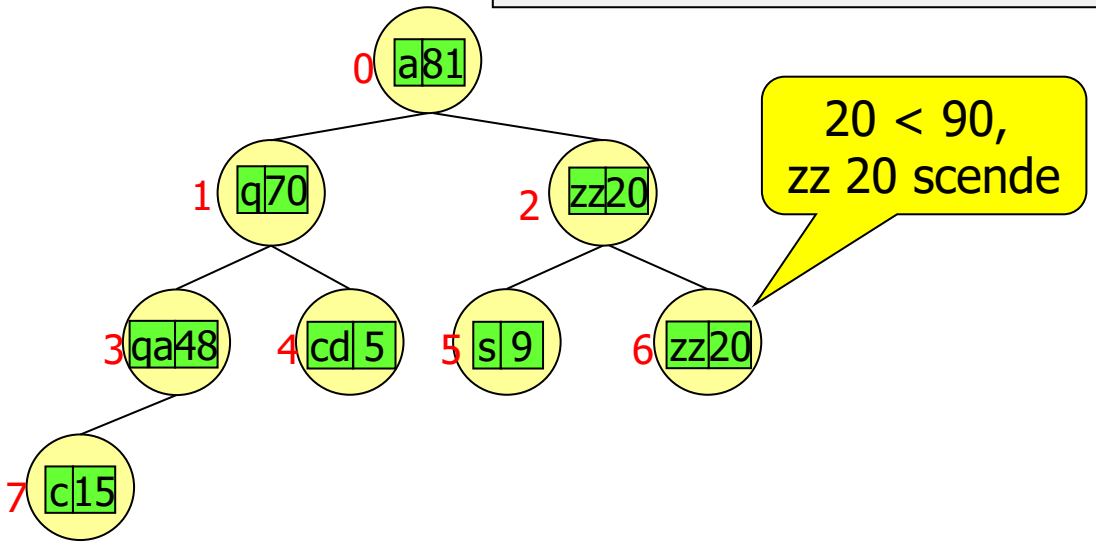


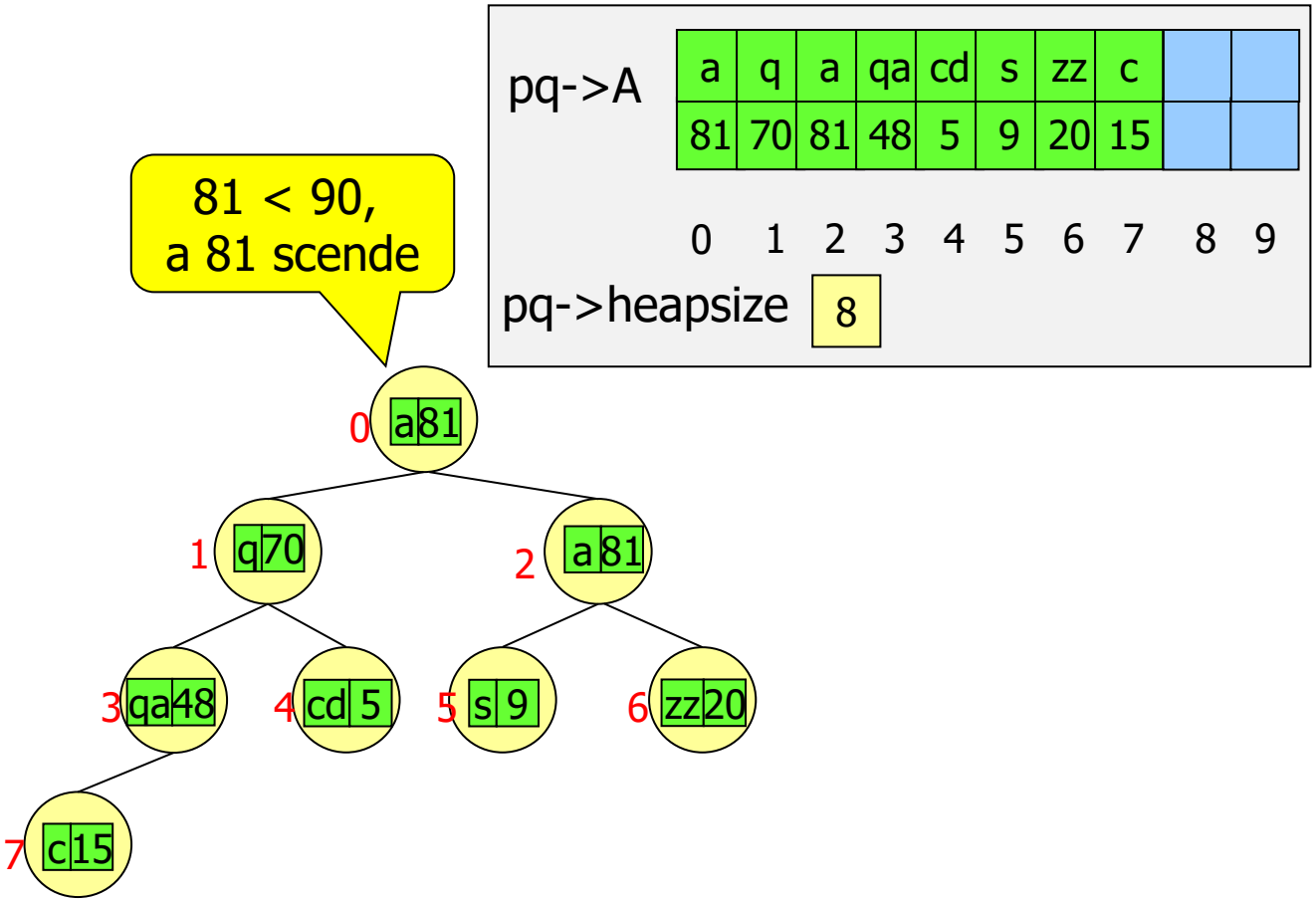
Cambio la priorità di w da 19 a 90. L'elemento si trova all'indice 6.

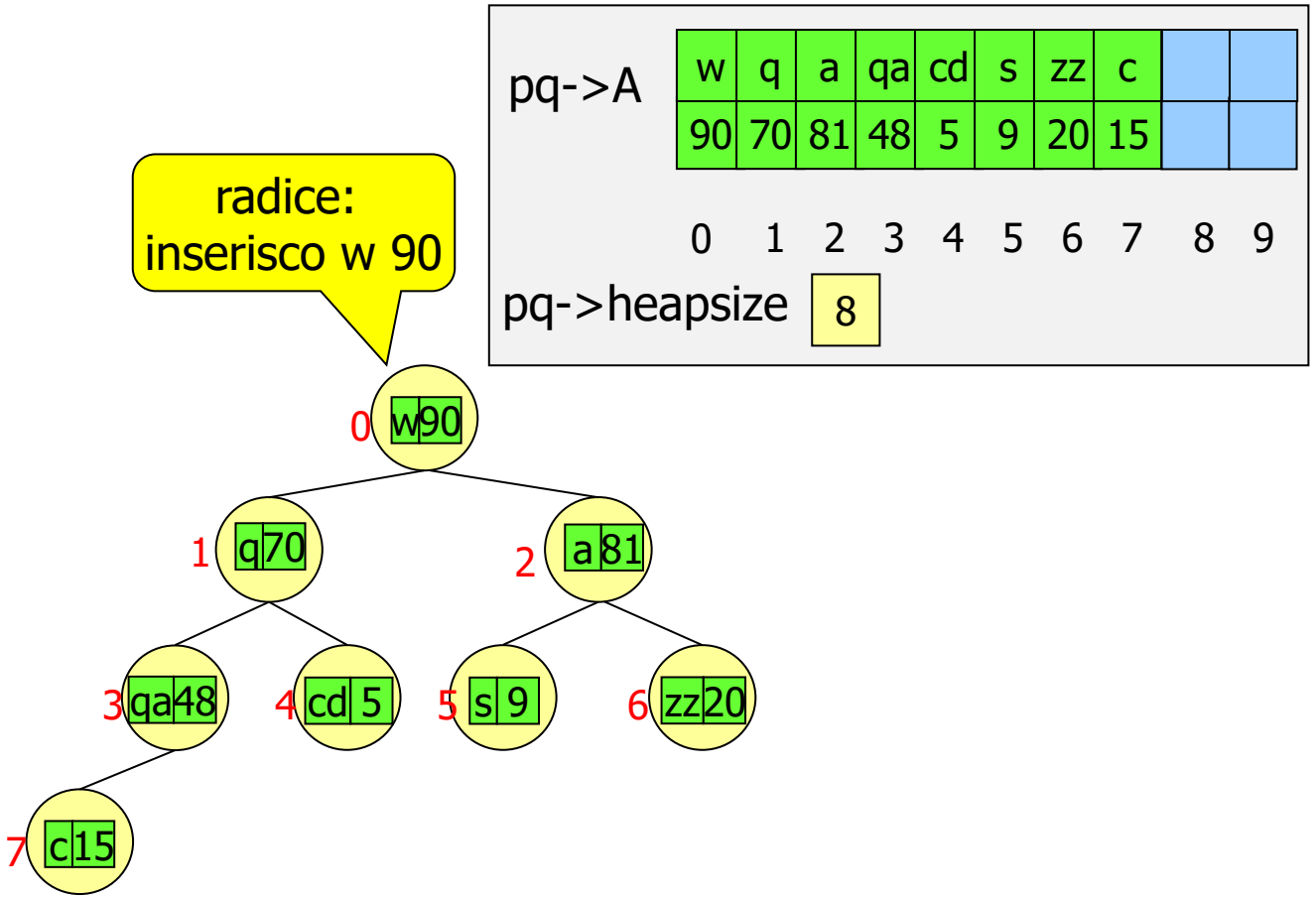
pq->A

a	q	zz	qa	cd	s	zz	c		
81	70	20	48	5	9	20	15		
0	1	2	3	4	5	6	7	8	9

pq->heapsize 8




pq->A
a q a qa cd s zz c
81 70 81 48 5 9 20 15
0 1 2 3 4 5 6 7 8 9
pq->heapsize
8
81 < 90,
a 81 scende
0 a 81
1 q 70
2 a 81
3 qa 48
4 cd 5
5 s 9
6 zz 20
7 c 15

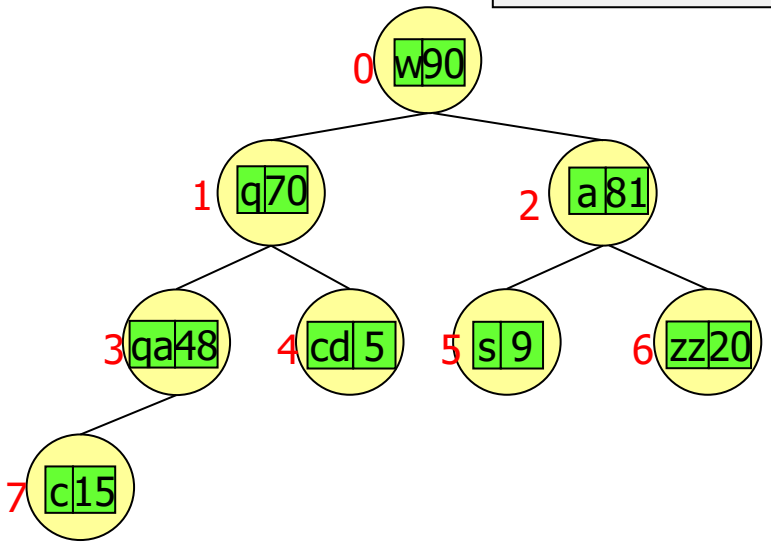

pq->A
w q a qa cd s zz c
90 70 81 48 5 9 20 15
0 1 2 3 4 5 6 7 8 9
pq->heapsize
8
radice:
inserisco w 90
0 w 90
1 q 70
2 a 81
3 qa 48
4 cd 5
5 s 9
6 zz 20
7 c 15

Esempio

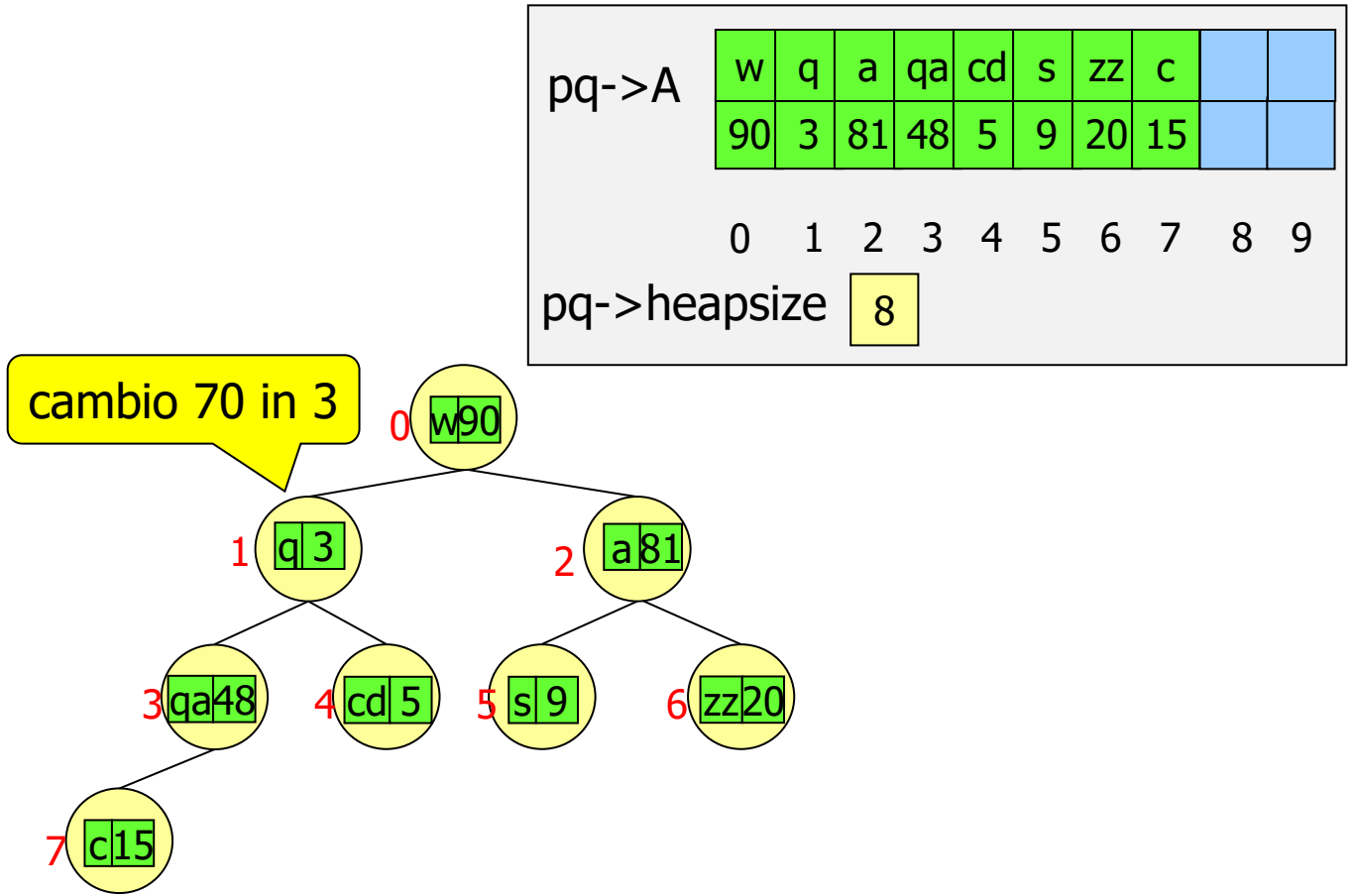
pq->A

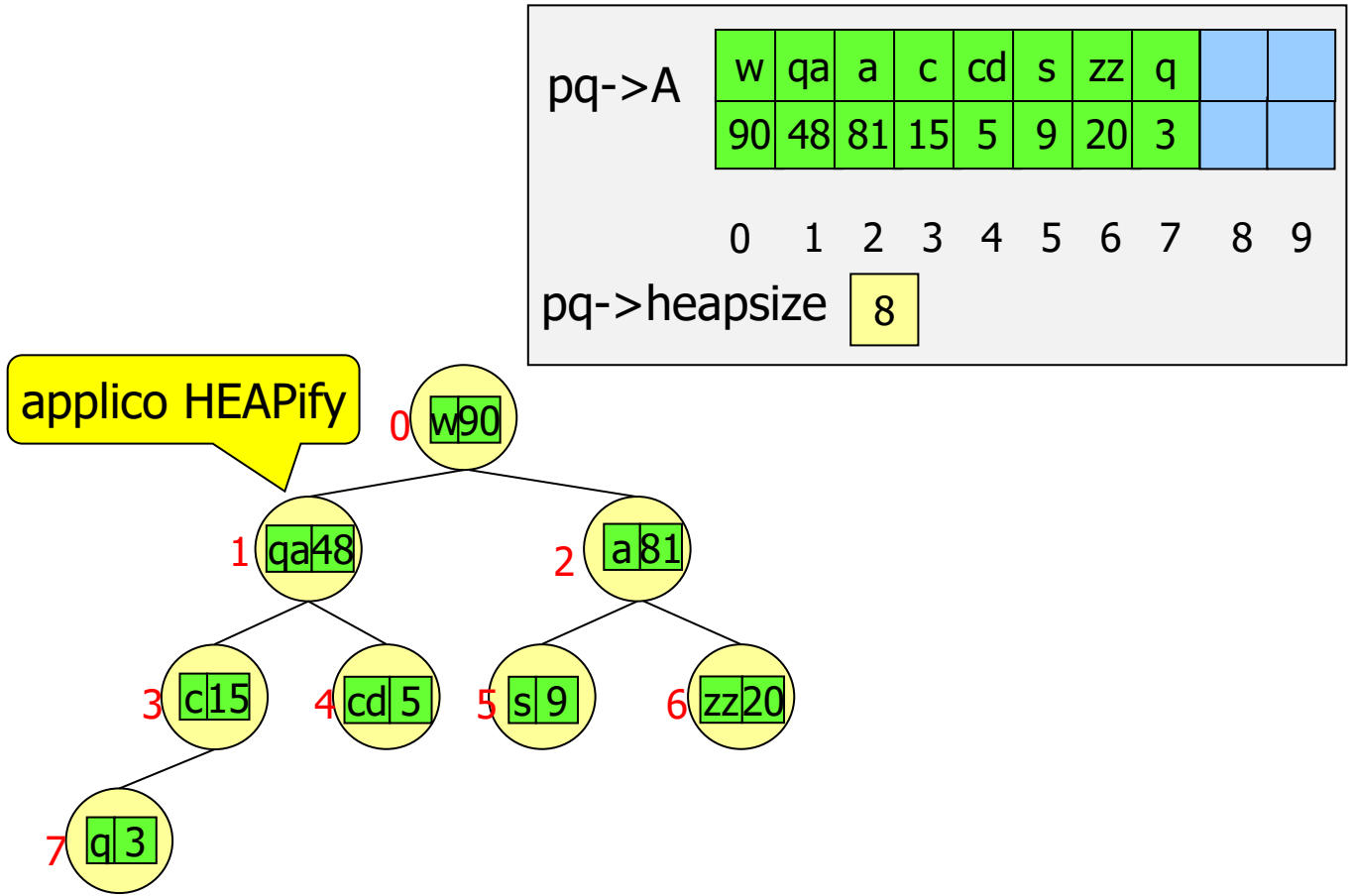
w	q	a	qa	cd	s	zz	c		
90	70	81	48	5	9	20	15		
0	1	2	3	4	5	6	7	8	9

pq->heapsize 8



Cambio la priorità di q da 70 a 3. L'elemento si trova all'indice 1.


pq->A
w q a qa cd s zz c
90 3 81 48 5 9 20 15
0 1 2 3 4 5 6 7 8 9
pq->heapsize
8
cambio 70 in 3
0 w 90
1 q 3
2 a 81
3 qa 48
4 cd 5
5 s 9
6 zz 20
7 c 15


pq->A
w qa a c cd s zz q
90 48 81 15 5 9 20 3
0 1 2 3 4 5 6 7 8 9
pq->heapsize
8
applico HEAPify
0 w 90
1 qa 48
2 a 81
3 c 15
4 cd 5
5 s 9
6 zz 20
7 q 3

```
void PQchange (PQ pq, Item val) {
  int i, found = 0, pos;
  for (i = 0; i < pq->heapsize && found == 0; i++)
    if (NAMEcmp(NAMEget(&(pq->A[i])), NAMEget(&val))==0) {
      found = 1;
      pos = i;
    }

  if (found==1) {
    while(pos>=1 &&
          PRIOget(pq->A[PARENT(pos)])<PRIOget(val)){
      pq->A[pos] = pq->A[PARENT(pos)];
      pos = PARENT(pos);
    }
    pq->A[pos] = val;
    HEAPify(pq, pos);
  }
  else
    printf("key not found!\n");
  return;
}
```

È possibile migliorare PQchange O(n)?

Occorre fare in modo che NON si debba **cercare** l'item nella coda

- Soluzione: L'Item ricorda/sa «dove» si trova nella coda (gestito dalle operazioni su PQ)

Possibili implementazioni:

A. In coda si inseriscono solo «riferimenti» ad Item (es. puntatori)
   - l'Item ha un campo pos (posizione in coda prioritaria)
   - Il modulo Item fornisce le operazioni ITEMsetPos ITEMgetPos che permettono di ottenere la posizione di un item con costo O(1)

B. L'Item è un indice (oppure ha come campo valore un indice in un vettore): in pratica è un riferimento a «dove» sono collocate le informazioni. La priorità può essere parte dell'item oppure può essere «affiancata» all'item/indice
   - La coda sfrutta internamente la corrispondenza dato-indice, associa a ogni item una casella unica di un vettore
   - E' possibile ottenere la posizione di un Item in coda con tempo O(1)

C. Non è possibile gestire un riferimento ad Item con puntatore o indice, si usa la chiave come riferimento (deve essere univoca, senza possibili duplicati)
   - il modulo PQ usa la chiave dell'item per gestire una corrispondenza chiave-posizione mediante una tabella di simboli efficiente (es. Hash O(1), BST bilanciato O(lg(n))

È possibile migliorare PQchange O(n)?

Occorre fare in modo che NON si debba **cercare** l'item nella coda

- Soluzione: L'Item ricorda/sa «dove» si trova nella coda (gestito dalle operazioni su PQ)

Possibili implementazioni:

A. In coda si inseriscono solo «riferimenti» ad Item (es. puntatori)
   - l'Item ha un campo pos (posizione in coda prioritaria)
   - Il modulo Item fornisce le operazioni ITEMsetPos ITEMgetPos che permettono di ottenere la posizione di un item con costo O(1)

B. L'Item è un indice (oppure ha come campo valore un indice in un vettore): in pratica è un riferimento a «dove» sono collocate le informazioni. La priorità può essere parte dell'item oppure può essere «affiancata» all'item/indice
   - La coda sfrutta internamente la corrispondenza dato-indice, associa a ogni item una casella unica di un vettore
   - E' possibile ottenere la posizione di un Item in coda con tempo O(1)

C. Non è possibile gestire un riferimento ad Item con puntatore o indice, si usa la chiave come riferimento (deve essere univoca, senza possibili duplicati)
   - il modulo PQ usa la chiave dell'item per gestire una corrispondenza chiave-posizione mediante una tabella di simboli efficiente (es. Hash O(1), BST bilanciato O(lg(n))

Coda prioritaria di indici

Non si inseriscono in coda gli item ma coppie (indice, priorità), quindi si adotta la versione di «chiave affiancata al dato» (la priorità è un parametro aggiuntivo) invece che «chiave parte del dato»

- Il vettore pq->qp (posizione in coda) serve per implementare Pqchange efficiente, identificando la posizione dell'elemento nello heap con costo O(1) (l'elemento è un indice) senza bisogno di una scansione lineare.

PQ.h

```
typedef struct pqueue *PQ;

PQ      PQinit(int maxN);
void    PQfree(PQ pq);
int     PQempty(PQ pq);
int     PQsize(PQ pq);
void    PQinsert(PQ pq, int index, int prio);
int     PQshowMax(PQ pq);
int     PQextractMax(PQ pq);
void    PQdisplay(PQ pq);
void    PQchange(PQ pq, int index, int prio);
```

PQ.h

```
typedef struct pqueue *PQ;

PQ      PQinit(int maxN);
void    PQfree(PQ pq);
int     PQempty(PQ pq);
int     PQsize(PQ pq);
void    PQinsert(PQ pq, int index, int prio);
int     PQshowMax(PQ pq);
int     PQextractMax(PQ pq);
void    PQdisplay(PQ pq);
void    PQchange(PQ pq, int index, int prio);
```

Scompare il tipo Item
Si gestiscono indici e priorità

PQ.c

```
...
typedef struct {int index; int prio} heapItem;
struct pqueue {heapItem *A; int heapsize; int *qp};

PQ PQinit(int maxN) {
  int i;
  PQ pq = malloc(sizeof(*pq));
  pq->A = malloc(maxN*sizeof(heapItem));
  pq->qp = malloc(maxN*sizeof(int));
  for (i=0; i < maxN; i++){
    pq->A[i].index = -1; pq->qp[i] = -1;
  }
  pq->heapsize = 0;
  return pq;
}
```

PQ.c

```
...
typedef struct {int index; int prio} heapItem;
struct pqueue {heapItem *A; int heapsize; int *qp;};

PQ PQinit(int maxN) {
  int i;
  PQ pq = malloc(sizeof(*pq));
  pq->A = malloc(maxN*sizeof(heapItem));
  pq->qp = malloc(maxN*sizeof(int));
  for (i=0; i < maxN; i++){
    pq->A[i].index = -1; pq->qp[i] = -1;
  }
  pq->heapsize = 0;
  return pq;
}
```

Item interno al modulo PQ per gestire la coppia (indice,priorità)

PQ.c

```
...
typedef struct {int index; int prio} heapItem;
struct pqueue {heapItem *A; int heapsize; int *qp};

PQ PQinit(int maxN) {
  int i;
  PQ pq = malloc(sizeof(*pq));
  pq->A = malloc(maxN*sizeof(heapItem));
  pq->qp = malloc(maxN*sizeof(int));
  for (i=0; i < maxN; i++){
    pq->A[i].index = -1; pq->qp[i] = -1;
  }
  pq->heapsize = 0;
  return pq;
}
```

Si assume che maxN sia, oltre che il massimo numero di dati in coda, il limite superiore agli indici

```
void PQfree(PQ pq) {
  free(pq->qp);
  free(pq->A);
  free(pq);
}

int PQempty(PQ pq){
  return pq->heapsize == 0;
}

int PQsize(PQ pq) {
  return pq->heapsize;
}
```

```
void PQinsert (PQ pq, int index, int prio){
  int i, j;
  i=pq->heapsize++;
  while((i>=1) &&
     (pq->A[PARENT(i)].prio)<prio)){
    pq->A[i] = pq->A[PARENT(i)];
    pq->qp[pq->A[i].index] = i;
    i = PARENT(i);
  }
  pq->A[i].index = index;
  pq->A[i].prio = prio;
  pq->qp[index] = i;
}
```

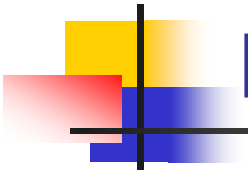
aggiorno pq->qp

```
static void Swap(PQ pq, int pos1, int pos2){
  heapItem temp;
  int index1, index2;
  temp = pq->A[pos1];
  pq->A[pos1] = pq->A[pos2];
  pq->A[pos2] = temp;
  // update correspondence index-pos
  index1 = pq->A[pos1].index;
  index2 = pq->A[pos2].index;
  pq->qp[index1] = pos1;
  pq->qp[index2] = pos2;
}
```

```
static void HEAPify(PQ pq, int i) {
  int l, r, largest;
  l = LEFT(i);
  r = RIGHT(i);
  if (l < pq->heapsize && (pq->A[l].prio > pq->A[i].prio))
    largest = l;
  else
    largest = i;
  if (r < pq->heapsize && (pq->A[r].prio > pq->A[largest].prio))
    largest = r;
  if (largest != i) {
    Swap(pq, i, largest);
    HEAPify(pq, largest);
  }
}
```

```
int PQextractMax(PQ pq) {
  int res;
  int j=0;

  Swap (pq, 0, pq->heapsize-1);
  res = pq->A[pq->heapsize-1].index;
  pq->qp[res]=-1;
  pq->heapsize--;
  pq->A[pq->heapsize].index=-1; // redundant
  HEAPify(pq, 0);
  return res;
}
```

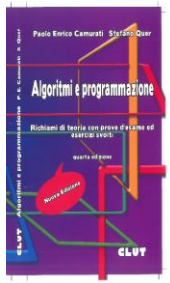
```
void PQchange (PQ pq, int index, int prio) {
  int pos = pq->qp[index];
  heapItem temp = pq->A[pos];
  temp.prio = prio; // new prio

  while ((pos>=1) && (pq->A[PARENT(pos)].index < prio) {
    pq->A[pos] = pq->A[PARENT(pos)];
    pq->qp[pq->A[pos].index] = pos;
    pos = PARENT(pos);
  }
  pq->A[pos] = temp;
  pq->qp[index] = pos;

  HEAPify(pq, pos);
}
```



Riferimenti

- Heap:
  - Cormen 7.2, 7.3
  - Sedgewick 9.2, 9.3
- Heapsort:
  - Cormen 7.4
  - Sedgewick 9.4
- Code a priorità:
  - Cormen 7.5
  - Sedgewick 9.1, 9.6

Esercizi di teoria



- 7. Code a priorità e heap
  - 7.1 Heap
  - 7.2 Heap Sort
  - 7.3 Code a priorità