Programmazione Orientata Agli Oggetti

Struttura dati Heap e HeapSort

Libero Nigro

Alcune osservazioni sull'albero binario di ricerca (classe ABR)

- Se un ABR contiene n nodi ed ha altezza h, è facile verificare che:

$$h + 1 \leq n < 2^{h+1}$$

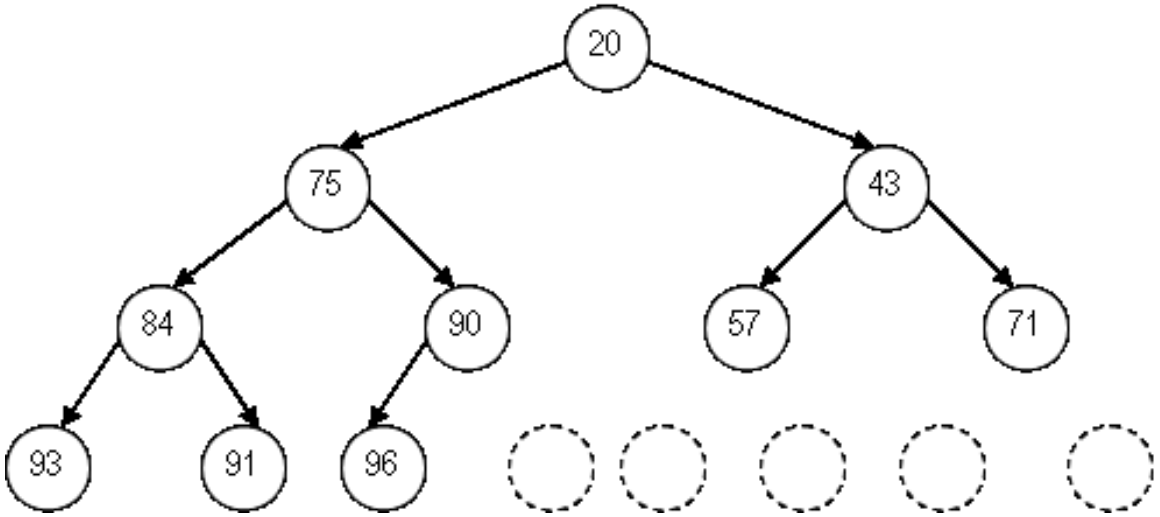
Infatti, se un percorso ha h archi, il percorso possiede h+1 nodi e, naturalmente, l'albero potrebbe ridursi a quel solo percorso.

D'altra parte, se l'altezza è h, h è anche il livello massimo nell'albero, e nel caso tutti i livelli siano pieni, $n = 2^h + 2^{h-1} + \cdots + 2 + 1$.
Es. se h=3, l'albero completo avrebbe $8 + 4 + 2 + 1 = 15$ nodi, dunque $< 2^{h+1} = 2^4 = 16$.
Si nota ancora che se n è la cardinalità di un ABR, l'altezza dell'albero è al massimo: $h = \log_2 n$. Es. Per n=15, h=3.

Heap – Definizioni e proprietà

Un **heap** (letteralmente “mucchio”) è una *struttura dati* del tipo collezione *parzialmente ordinata*. Essa è definita naturalmente su un albero binario. Gode delle seguenti due proprietà:

1) un heap è un albero binario quasi completo. Ogni suo livello è completo dei suoi nodi tranne l’ultimo livello, dove possono mancare nodi nella parte destra:
2) *ogni nodo* contiene un valore che *è minore di ogni suo discendente*, ossia il sotto albero sinistro e quello destro contengono nodi con valori maggiori o uguale alla radice (e cosi via ricorsivamente). La figura che segue mostra un heap. La radice dell’albero contiene 20 che è minore di tutti i suoi discendenti e cosi via ricorsivamente per ogni nodo.

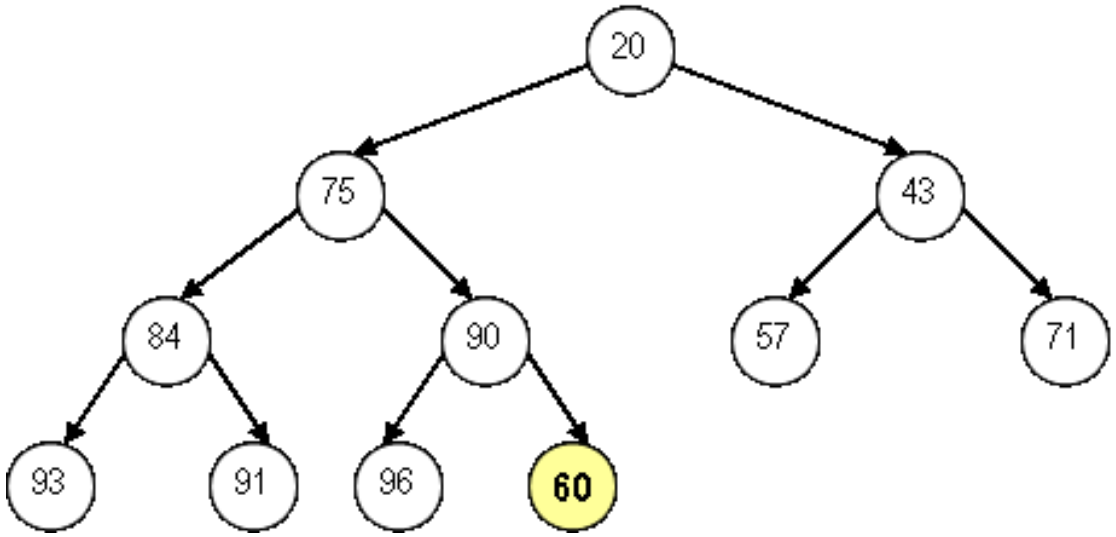


Le differenze dall'albero binario di ricerca sono evidenti, sia nel riempimento dei livelli sia, soprattutto, nella disposizione dei valori dei nodi.

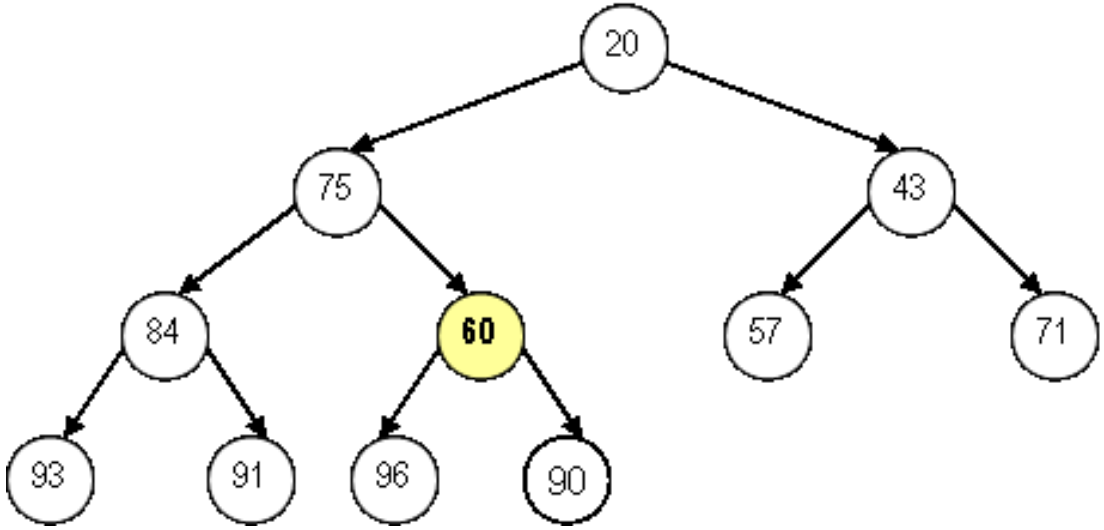
A causa delle sue proprietà, *un heap ammette operazioni di inserimento e rimozione molto efficient*i. È importante riflettere che di momento in momento, *la radice dell'albero contiene il minimo* e che un nuovo inserimento potrà avvenire *riempiendo il primo buco libero sull'ultimo livello*.

Aggiunta di un nuovo elemento

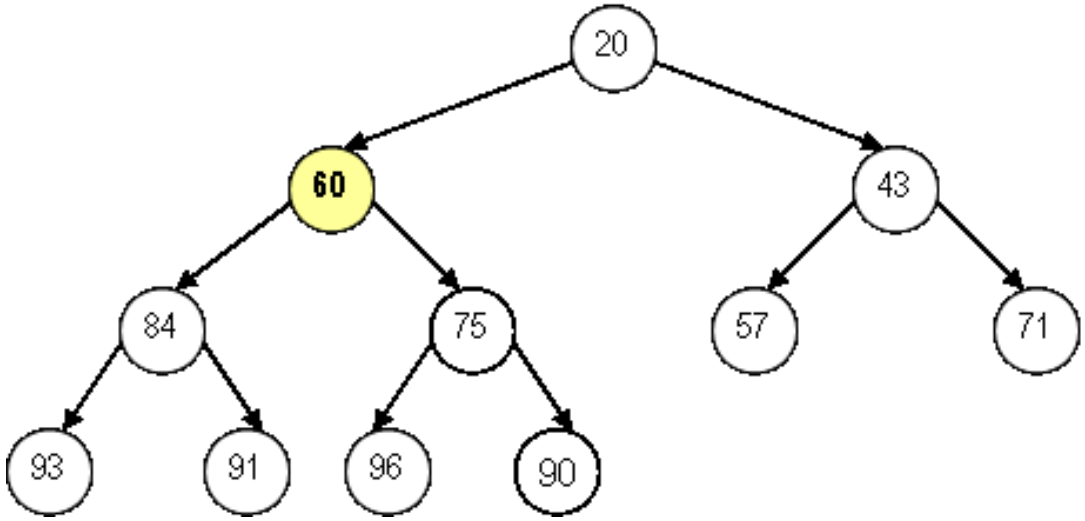
Si vuole aggiungere 60 all'heap precedente. Si ha:



Si crea un nodo con 60 e lo si attacca (nel caso particolare dell'esempio) come figlio destro di 90. Ovviamente, un inserimento per essere accettabile deve mantenere le proprietà dell'heap. Tuttavia a causa dell'inserimento di 60, l'heap non è verificato localmente al sotto albero di radice 90, in quanto 90 non è minore di ogni suo discendente. Per riaggiustare l'heap si scambiano 90 e 60:



Si vede che nonostante lo scambio di 90 (padre originario di 60) col 60, ancora l'heap è violato. Infatti il padre di 60 (cioè 75) non è minore di tutti i suoi discendenti. Dunque si prosegue scambiando il 75 col 60.

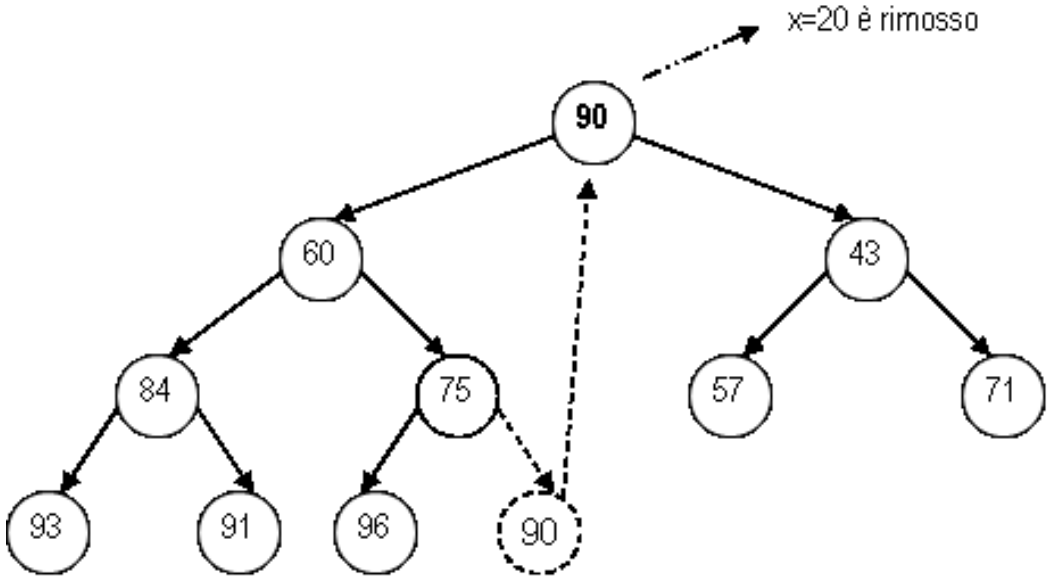


A questo punto l'heap è ricostituito. Per riassumere: quando si aggiunge un elemento, si riempie il primo buco libero e si riaggiusta l'heap *upward* (dalla foglia verso la radice) confrontandosi col nodo padre e scambiando subito se la proprietà dell'heap non è verificata (un nodo radice deve risultare minore di ogni suo discendente). Si continua cosi, sino a che il nodo corrente è non minore del suo nodo radice.

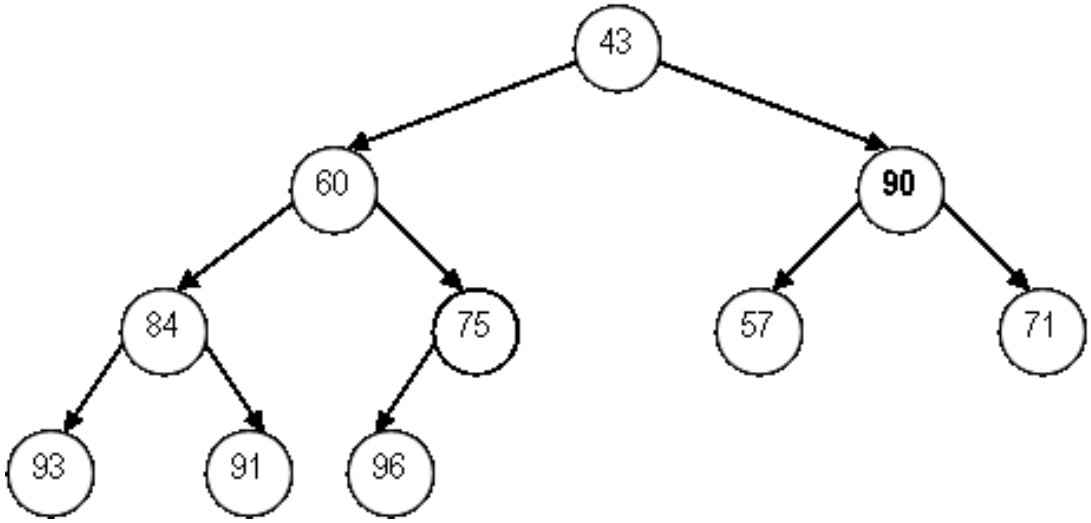
Rimozione del minimo

In un heap l'elemento che viene rimosso è di norma la radice, ossia il minimo in tutto l'albero. Trattandosi di un nodo con entrambi i figli, si provvede a colmare la lacuna promuovendo l'ultimo nodo nell'ultimo livello al posto della radice, ed eliminando nel contempo l'ultimo nodo.

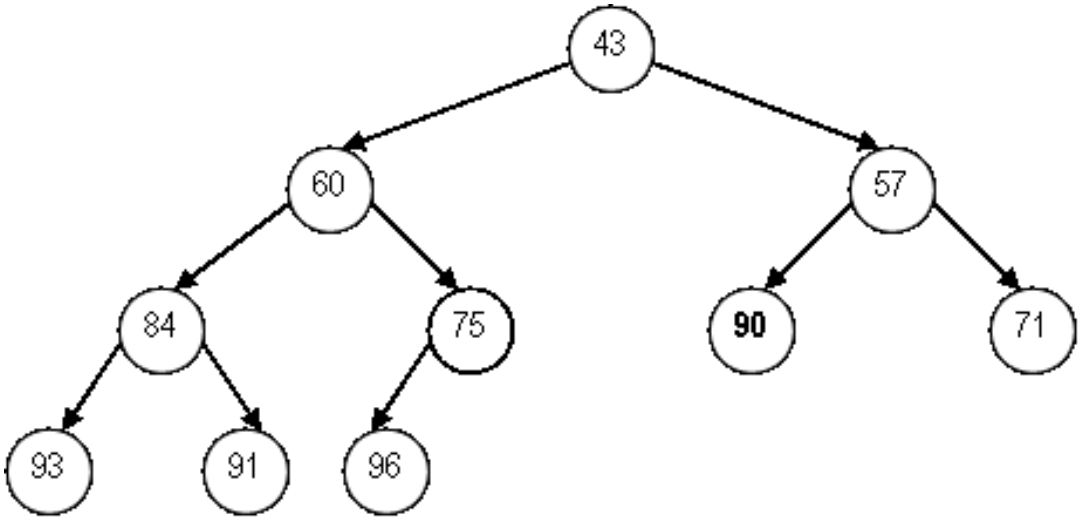
Nell'albero risultante di cui sopra, l'operazione di rimozione restituisce x=20. Quindi si promuove 90 al posto della vecchia radice e si distrugge il vecchio nodo 90. Naturalmente l'operazione distrugge tipicamente la proprietà dell'heap per cui occorre riaggiustarlo ma stavolta procedendo *downward* (dalla radice verso le foglie).



Il riaggiusto downward dell'heap si può ottenere confrontando la radice con entrambi i nodi figli immediati. È sufficiente trovare il minimo tra i due figli della radice. Occorre procedere ad uno scambio se la radice è maggiore di questo minimo. Nel caso di cui sopra, il minimo tra 60 e 43 è 43. Siccome 90 (la nuova radice) è maggiore di 43, si scambiano i nodi con 90 e 43 determinando la nuova situazione mostrata di seguito:



Come si vede, l'heap è a posto dal punto di vista della radice dell'albero, ma non dal punto di vista del nodo radice del sotto albero destro. Infatti 90 non è minore di tutti i suoi discendenti! Si trova il minimo tra 57 e 71 (cioè 57) e si scambiano il 90 col 57:



A questo punto tutto l'heap è ricomposto e nella radice esiste il nuovo minimo (43) che una successiva operazione di estrazione provvederà a restituire etc.

Efficienza delle operazioni di inserimento/rimozione

Dall'analisi della dinamica delle operazioni che si devono compiere per ricomporre l'heap dopo un'aggiunta o una rimozione, risulta che esse interessano solo un percorso (es. quello che dalla foglia-ultimo nodo porta sino alla radice nel caso di inserimento, o quello che dalla radice porta ad una foglia nel caso di rimozione). Pertanto, il numero di operazioni effettuate, nel caso peggiore, è pari alla lunghezza di un percorso, cioè è pari all'altezza dell'albero binario (massima lunghezza di un percorso dalla radice ad un nodo foglia), vale a dire $h = \log_2 n = \log n$.

In altre parole, le operazioni si inserimento e rimozione, costando O(h) costano in realtà O(log n) dunque sono estremamente efficienti. Il prezzo da pagare per questa efficienza è quello di accettare una struttura dati parzialmente ordinata, con la garanzia che la radice è il minimo nell'albero.

Come sfruttare l'efficienza dell'heap ?

Una struttura dati heap si può agevolmente mappare su un array ed essere manipolata direttamente sull'array. E' sufficiente depositare sull'array gli elementi dell'albero heap secondo la *visita per livelli*.

Rinunciando ad utilizzare (per semplicità) il primo elemento (indice 0), riferendoci all'ultimo albero heap, si ha:

Array heap:

	43	60	57	84	75	90	71	93	91	96					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	...	
	liv-0	**liv-1**		**liv-2**				**liv-3**							

Il nodo radice è posto in posizione 1. I suoi figli sono collocati in posizione 2 e 3.
In generale, un nodo collocato in posizione j, avrà i suoi figli posti negli indici 2j e 2j+1.
A causa di questa semplicissima regola, i livelli dell'albero sono disposti come mostrato sulla figura dell'array. Il livello 3, l'ultimo, è incompleto. Si nota come il primo buco libero sull'albero viene a coincidere col primo buco libero sull'array (in questo caso con la posizione 11).

È utile riesprimere le proprietà dell'heap sull'array.

1. La radice (cioè il minimo) è in posizione 1.
2. La proprietà heap interpretata downward (cioè, dalla radice in giù) si esprime dicendo: a

$$heap[i] \leq heap[2i] \;\&\&\; heap[2i+1], \forall i \epsilon [1, n/2]$$

   ammesso che esista l'elemento heap[2i+1]
3. La proprietà heap interpretata upward (cioè, da una foglia in su nell'albero) si esprime dicendo:

$$heap[i] \geq heap[i/2] \;\; \forall i, da\ n\ a\ 2$$

Direttamente fondata su queste osservazioni è la classe Heap appartenente al package poo.heap.

Osservazioni: mentre le operazioni di inserimento e rimozione costano O(log n)
se n è il numero degli elementi dell'heap, la ricerca di un elemento costa O(n). Inoltre, la rimozione di un elemento, dal secondo in poi, costa il tempo necessario a ricomporre l'heap considerando la necessità di dover ri-collocare nell'heap tutti gli elementi che seguono quello rimosso.

Possibili usi di una struttura dati heap

Un heap può essere direttamente sfruttato per ottenere una *coda a priorità*, cioè una coda nella quale gli elementi che arrivano non escono secondo l'ordine di arrivo (comportamento FIFO delle normali code) ma in base ad un criterio di urgenza (o priorità). Si pensi ad un pronto soccorso in cui le persone che giungono possono non essere servite secondo l'ordine d'arrivo ma piuttosto secondo l'urgenza o gravità dei singoli casi.

L'arrivo in coda (operazione add(x)) aggiunge x all'heap ma preserva la proprietà dell'heap. Un'estrazione (operazione remove()) toglie e restituisce il minimo dell'heap, ossia la sua radice in posizione 1, dopo di che si ricostituisce l'heap.

L'implementazione della classe Heap può essere banalmente letta come implementazione di una coda a priorità.

Un secondo uso della struttura dati Heap è come supporto all'ordinamento di un array, ciò che è noto come HeapSort, richiamato di seguito (si fa riferimento ad un metodo static ad esempio posto nella classe poo.util.Array):

```
public static <T extends Comparable<? super T>> void heapSort( T[] v ){
    Heap<T> h=new Heap<T>(v.length);
    //prima fase: riempimento heap
    for( T e: v ) h.add(e);
    //seconda fase: svuotamento heap
    for( int i=0; i<v.length; i++ ) v[i]=h.remove();
}//heapSort
```

Il metodo di utilità heapSort() consiste di due fasi: nella prima si riempie un heap di appoggio con gli elementi dell'array. Nella seconda si provvede ripetutamente a rimuovere la testa dell'heap e a metterla nella prima posizione libera dell'array.

Complessità di HeapSort

$T_{HeapSort}(n)=T(\text{prima-fase})+T(\text{seconda-fase})$

dove n è la dimensione del problema, ossia il numero di elementi dell'array da ordinare. È pressocchè evidente che i due contributi al tempo di esecuzione, cioè della prima parte e della seconda sono uguali. Valutiamo dunque il tempo della prima fase. Nel caso peggiore, ogni nuovo elemento aggiunto in posizione j dell'array, migra in tutti i modi possibili, finendo nella radice (in posizione 1). Ossia le operazioni sono log j (log in base 2). Quindi:

T(prima-fase)=log 1 + log 2 + log 3 + … + log (n-1) + log n = log (1*2*3*…(n-1)*n)=log n!

Utilizzando l'approssimazione di n! (per n in crescita asintotica) fornita dalla formula di Stirling:

$n! \approx (\frac{n}{e})^n \sqrt{2\pi n}$ dove e è il numero di Nepero (e≈2.718).

Si ha (prendendo i log in base 2):

log n!=n*log(n/e)+1/2*[log(2π)+log n] ≈n log n

e complessivamente:

$T_{HeapSort}$(n)=2T(prima-fase) ≈2 n log n =O(n log n).

Implementazione di una classe Heap

```
package poo.util;
public class Heap<T extends Comparable<? super T>>{
    private T[] heap;
    private int n, size;
    //size punta all'ultimo occupato: 1<=size<=n
    @SuppressWarnings("unchecked")
    public Heap( int n ){
        if( n<=0 )  throw new IllegalArgumentException();
        this.n=n; size=0;
        heap=(T[]) new Comparable[n+1];
    }//Heap
    public int size(){ return size; }//size
    public boolean contains( T elem ){
        for( int i=1; i<=size; i++ )
            if( heap[i].equals(elem) ) return true;
        return false;
    }//contains
```

```
public void add( T elem ){
     if( size==n ){//espandi
          heap=Arrays.copyOf( heap, 2*n+1 );
          n=2*n;
}
     size++;
     heap[size]=elem; //aggiunge elem in ultima posizione
     //aggiusta heap “upward”
     int i=size;
     while( i>1 ){
         if( heap[i].compareTo(heap[i/2])<0 ){
               //scambia heap[i] e heap[i/2]
               T park=heap[i]; heap[i]=heap[i/2];
               heap[i/2]=park;
               i=i/2;
          }
     else break;
     }
}//add
```

```
public T remove(){
    //rimuove il minimo e lo restituisce
    if( size==0 )  throw new RuntimeException("Heap empty!");
    T min=heap[1];
    heap[1]=heap[size]; //promozione ultimo elemento
    heap[size]=null; size--;
    //riaggiusto heap
    int i=1;
    while( i<=size/2 ){
        int j=2*i, k=j+1;
        //trova min tra heap[j] e heap[k], sia z l'indice del min
        int z=j;
        if( k<=size && heap[k].compareTo(heap[z])<0 ) z=k;
        if( heap[i].compareTo(heap[z])>0 ){
            //scambia heap[i] con heap[z]
            T park=heap[i]; heap[i]=heap[z]; heap[z]=park;
            i=z;
        }
        else break;
    }
    return min;
}//remove
```

```
    public void clear(){
        for( int i=1; i<=size; i++ )
            heap[i]=null;
        size=0;
    }//clear

    public String toString(){
        StringBuilder sb=new StringBuilder(200);
        sb.append('[');
        for( int i=1; i<=size; i++ ){
            sb.append( heap[i] );
            if( i<size ) sb.append(',');
        }
        sb.append(']');
        return sb.toString();
    }//toString

}//Heap
```

Modificare la classe Heap

- aggiungendo un metodo void remove(T elem) che rimuove, se esiste, la prima occorrenza di elem (fatto a lezione);
- prevedendo la struttura di iterazione (rendere Heap iterabile) (esercizio proposto).

Esercizio

Data la sequenza di arrivo di numeri: 23 -3 10 2 5 1 -4 12 7 6

1. Costruire graficamente l'ABR corrispondente.
2. Costruire graficamente l'albero heap corrispondente.
3. Nell'ipotesi di rimuovere il primo elemento dell'heap, dimostrare graficamente come cambia l'heap.

In ogni caso mostrare il contenuto di un heap anche in modo lineare con la visita a livelli.

La classe PriorityQueue<E> di java.util

Oltre che usare un proprio heap, es. un'istanza della classe Heap di poo.util, è possibile sfruttare la classe generica e parametrica PriorityQueue<E> di java.util.
Il tipo parametrico E si suppone dotato del confronto naturale (implementazione dell'interfaccia Comparable). In alternativa si può usare un costruttore di PriorityQueue passando un oggetto comparatore deputato a fare i confronti.

Costruttori (lista parziale)

PriorityQueue()	capacità di default: 11
PriorityQueue(int capacita)	
PriorityQueue(Collection<? extends E> c)	inizializza la priority queue con gli elementi provenienti da c
PriorityQueue(int capacita, Comparator<? super E> oggetto-comparatore)	

Metodi

boolean add(E e) o offer(E e)	aggiungono l'elemento e alla coda
void clear()	
boolean contains(Object x)	
E peek()	ritorna la testa della coda, senza rimuoverla
E poll()	ritorna la testa della coda, rimuovendola
int size()	
boolean remove(Object x)	rimuove x dalla coda
Iterator<E> iterator()	
Object[] toArray()	