# Code di priorità - ADT

A volte utilizzare una coda del tipo FI-FO potrebbe non essere sufficiente! Es, se costruiamo una coda d. processi sul kernel, è importante che alcuni di essi (es. movimento del mouse) vengano computati dalla CPU prima di altri, meno importanti. A tal scopo si implementano le ADT.

Quando si aggiungerà un elemento, dunque, l'utente della coda associerà all'elemento una priorità sotto forma di chiave.
L'elemento a chiave minima sarà il prossimo ad essere rimosso.

Si costruirà l'oggetto voce (entry), che avrà come valore l'oggetto aggiunto e chiave la sua relativa priorità
Si costruiranno funzioni a supporto di esso.

| | |
|---|---|
| insert (k,v) | inserisce nella coda una voce con chiave k e val v |
| min () | restituisce la voce della coda che ha chiave minima, senza eliminarla (analogo a peek()) |
| removeMin() | Elimina e restituisce la voce relativa alla chiave minore, null se vuota |
| size() | Restituisce il numero di voci presenti nella ADT |
| isEmpty() | Restituisce il boolean true se è vuota, false altrimenti |

Serve implementazione di comparatore per trovare il minimo

Varie implementazioni costruite dall'

```
public abstract class AbStract PriorityQueue <K,V> {

    // implementazione <key, val>
    // comparator salvato
    // size
    //
       :
```

- LISTA NON ordinata (gli elementi sono inseriti alla fine)
- Lista ORDINATA (primo el. è il minimo)

Confronto Costi asintotici

| | sortedListPriorityQueue | unsortedListPriorityQueue |
|---|---|---|
| size | O(1) | O(1) |
| isEmpty | O(1) | O(1) |
| insert | O(n) | O(1) |
| min | O(1) | O(n) |

Ottimi compromessi da entrambe le parti

↳ Costruzione Heap.

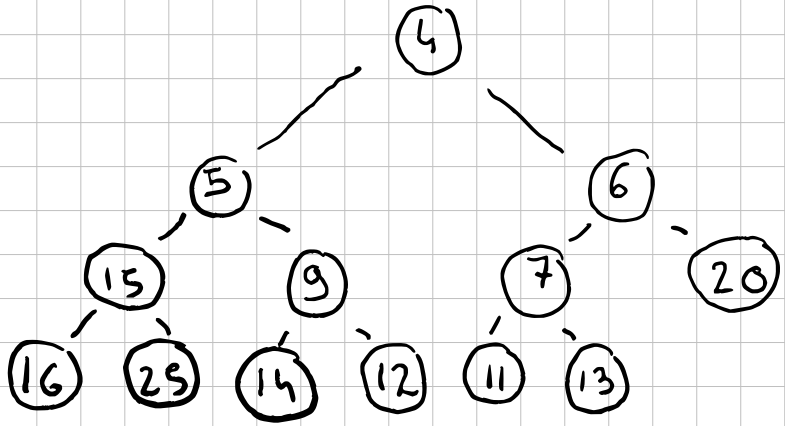
# HEAP

Con essa sia gli inserimenti che le rimozioni saranno logaritmiche, infatti sfrutterà il concetto dell' ALBERO BINARIO

In uno heap T, per ogni posizione p diversa dalla radice, la chiave memorizzata in p è NON MINORE della chiave memorizzata nel genitore di p

per questioni di efficienza, i livelli da 0 ad h-1 saranno sempre completi, e al livello h i nodi occupano la posizione a sinistra

ES min Heap




Ora la voce con la chiave minima è il nodo radice (4)

L'implementazione della struttura dati delle ADT come Heap porta ad alcuni vantaggi.
Gli algoritmi interessanti consistono nell'insert e removeMin

## Aggiunta di voce ad uno Heap

Dovendo rispettare la condizione di completezza, il nodo Entry può essere aggiunto unicamente nella posizione p subito alla destra dell'ultimo nodo.

Una volta aggiunto, è importante fare si che la regola dello Heap continui ad essere rispettata.

## Risalita dello Heap dopo un aggiunta

A meno che l'albero in cui si è aggiunto l'Entry fosse vuoto, possiamo sempre confrontarlo con il genitore.
Se $k_p$ (posizione) < $k_q$ (genitore) allora: il nodo p e il suo genitore devono essere SCAMBIATI. Lo scambio continua finche o si è arrivati alla radice oppure finche la proprietà NON RISULTA DI NUOVO SODDISFATTA ($k_p \geq k_q$)
La tecnica si chiama UP-HEAP bubbling

Nel caso peggiore, ovvero nel caso in cui la procedura UP-HEAP termini col nodo nella radice, il tempo asintotico sarà dell'Altezza dell'Albero, ovvero

$$O(\log n)$$

# Eliminazione di una entry avente chiave minima

Il metodo dell'eliminazione non consiste solamente nell'eliminazione della radice, in quanto ciò porterebbe a 2 alberi disconnessi.
Dopo la rimozione quindi si copia la foglia nell'ultima posizione sulla radice

# Discesa dello heap dopo una rimozione

Nonostante ora T è completo, la proprietà di T non è sicuramente rispettata!
Se T ha un solo nodo la procedura termina, altrimenti:

- Se p non ha figlio destro, si chiama c il figlio sinistro di p
- Altrimenti chiamamo c il figlio di avente chiave minore

Se $k_p \leq k_c$ la proprietà di ordinamento è soddisfatta e l'algoritmo termina, altrimenti scambia le voci di c e p.
L'algoritmo continua finché non ci sono punti in cui la proprietà di ordinamento è violata. Il processo è detto down-heap bubbling

Il caso peggiore consiste nell'applicazione del down-heap bubbling finché non si è una foglia, quindi quando si è arrivati alla max altezza. Il costo asintotico è

$$O(\log n)$$

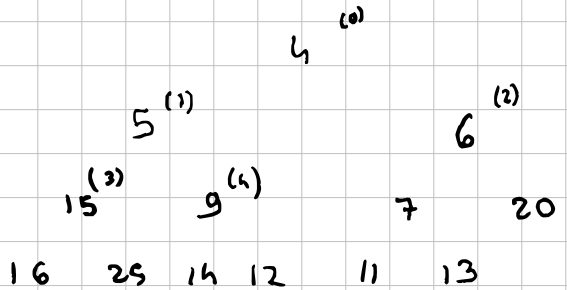
# RAPPRESENTAZIONE HEAP BINARIO MEDIANTE ARRAY

In generale non è possibile rappresentare facilmente un albero attraverso un array, ma sfruttando la completezza dello Heap T ci è possibile

L'elemento in posizione p quindi conterrà il nodo A secondo la seguente regola

- Se p è la radice, allora $f(p) = 0$
- Se è il figlio sinistro della posizione q, allora $f(p) = 2f(q) + 1$
- Se è il figlio destro della posizione q allora $f(p) = 2f(q) + 2$

Si prenda un esempio




| 4 | 5 | 6 | 15 | 9 | 7 | 20 | 16 | 25 | 14 | 12 | 11 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

questo rimuove alcune complicazioni date dalle linked list, e permette operazioni come up-heap e down heap in maniera veloce

# COSTRUZIONE DI HEAP Bottom-up

dato uno heap vuoto e un set d. dati, è possibile costruire uno heap in tempo nlogn attraverso n insert, ma esiste un modo più efficace attraverso una costruzione dal basso verso l'alto (bottom-up) in tempo O(n).

Assumendo che lo heap sia un albero completo pieno possiamo sapere che il numero di chiavi sarà $2^{h+1}-1$

Ci saranno quindi $h+1$ fasi

1. Nella fase 1 si costruiscono $(n+1)/2$ heap elementari ad una sola voce

2. Nella fase 2 si compongono $(n+1)/4$ heap, ciascuno dei quali ha 3 voci, ovvero i 2 heap elementari uniti, assunta una nuova voce. Si dovrà forse applicare uno scambio down-heap per ripristinare le proprietà di heap

3. Nella terza fase si compongono $(n+1)/8$ heap, con 7 elementi ciascuno, composta da 2 coppie unite + la radice. È possibile dover fare down-heap

⋮

$h+1$ nell'ultima fase si compone lo heap conclusivo
[...]

16 15 4 12 6 7 23 20 1°

15 4 6 17

16 25 5 12 11 7 23 20 2°

4 6

15 5 7 17

16 25 9 12 11 8 23 20 3°

4

5 6

15 9 7 17

16 25 14 12 11 8 23 20 4

0 nuovo

9 numero scambiato con...

# Implementazione Java e heapify

Il merging di 2 heap aventi stessa dimensione può essere effettuato semplicemente applicando il down-heap all'entità presente in p, che sarebbe la posizione in comune dei 2 alberi (quindi la nuova radice)

Utilizzando l'implementazione basata su array, si possono inserire tutti gli elementi in ordine arbitrario, applicando poi la procedura di down heap per ciascuna posizione dell'albero, partendo dalla posizione più interna più profonda (non foglia)

Si aggiungerà il metodo Heapify all'implementazione dello HeapPriorityQueue, che invoca downHeap per ogni posizione che non sia foglia, a partire dal basso.

```
protected void heapify() {
    int startIndex = parent(size-1)      // inizia da genitore dell'
    for(int j = startIndex; j >= 0; j--) {   ultima voce
        downHeap(j);
    }
```

La costruzione bottom-up di uno heap avente n entità richiede un tempo O(n) nell'ipotesi che 2 chiavi possono essere confrontate in tempo O(1)

dim: somma dei percorsi fatti con downHeap sono O(n), in quanto se si visitano tutti i rami secondo dx-sx..sx e si sommano tutti i percorsi, univoci, si hanno limitazioni O(n)

(guarda doc classroom Heap - parte 2)

# SORTING HEAP QUEUE

1. Nella prima fase, si inseriscono gli elementi di S come chiavi in una coda prioritaria P vuota, eseguendo n operazioni insert, una per ogni elemento

2. Nella seconda fase, si estraggono gli elementi di P in ordine non decrescente, eseguendo n operazioni removeMin, per memorizzarli di nuovo in S in ordine di estrazione

Lo schema è il paradigma seguito da diversi algoritmi di ordinamento, come il selectionSort e l'heapSort

```
public static <E> void pqSort(List<E> S){
    HeapQueue<E,?> P;
    int n = S.size();
    for(int j=0, j<n; j++){
      E elem = S.remove(S.first());
      P.insert(elem, null);
    }
    for(int j=0, j<n; j++){
      E elem = P.removeMin.getKey();
      S.addLast(element)
    }
}
```

O(n log n)

O(n log n)

Il costo dello heapSort è $O(n \log n)$