Problema della Coda con priorità

```
tipo CodaPriorita
dati:
        Un insieme S di n elementi di tipo elem a cui sono associate chiavi
di tipo
        chiave prese da un universo totalmente ordinato.

operazioni:
        findMin() -> elem:
                Restituisce l'elemento in S con la chiave minima

        insert(elem e, chiave k):
                Aggiunge a S un nuovo elemento e con chiave k

        delete(elem e):
                Cancella da S l'elemento e

        deleteMin():
                Cancella da S l'elemento con chiave minima

        increaseKey(elem e, chiave d):
                Incrementa della quantità d la chiave dell'elemento e in S

        decreaseKey(elem e, chiave d):
                decrementa della quantità d la chiave dell'elemento e in S

        merge(CodaPriorita c1, CodaPriorita c1) -> CodaPriorita:
                Restituisce una nuova coda con priorità c3 = c1 U c2
```

Applicazioni:

- Gestione code in risorse condivise
- Gestione priorità in processi concorrenti
- Progettazione di algoritmi efficienti per diversi problemi (calcolo cammini minimi).

Tabella delle implementazioni elementari

	FindMin	Insert	Delete	DeleteMin
Array non ordinato	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$
Array ordinato	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Lista non ordinata	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$
Lista ordinata	$O(1)$	$O(n)$	$O(1)$	$O(1)$

Come si può notare abbiamo con ogni implementazione almeno una delle operazioni che ci costa in tempo lineare.

Implementazioni evolute

- **d-heap**
- **Heap binomiali**
- **Heap di Fibonacci**

d-heap

Un d-heap è un albero radicato d-ario con le seguenti proprietà:

1. Struttura: Completo almeno fino al penultimo livello, e tutte le foglie sull'ultimo livello sono compattate verso sinistra.
2. Contenuto informativo: Ogni nodo v contiene un elemento $elem(v)$ ed una chiave $chiave(v)$ presa da un dominio totalmente ordinato.
3. Ordinamento parziale (inverso) dell'heap (min-heap): $chiave(v) \geq chiave(parent(v))$ per ogni nodo v diverso dalla radice.

Proprietà:

1. Un d-heap ha altezza $\Theta(log_d(n))$
2. La radice conetiene l'elemento con chiave minima
3. Può essere rappresentato tramite vettore posizionale.

Procedure ausiliarie

Procedure che torneranno utili per implementare le altre operazioni:

```
procedura muoviAlto(v):         T(n) = O(log_d(n))
        while(v != radice(T) and chiave(v) < chiave(padre(v))) do
```

```
                scambia il posto v e padre(v) in T

procedura muoviBasso(v)         T(n) = O(log_d(n))
        repeat
                sia u il figlio di v con la minima chiave(u), se esiste
                if(v non ha figli o chiave(v) <= chiave(u)) break
                scambia di posto v e u in T
```

Osservazioni:

MuoviBasso è semplicemente fixHeap

findMin()

```
findMin() -> elem:
        restituisci l'elemento nella radice di T
```

Complessità costante.

insert(elem e, chiave k)

Creaimo un nodo v con elemento e e chiave k in modo che sia l'ultimo elemento dell'albero T. Rispristiniamo l'ordinamento dell'heap spingendo il nodo verso l'alto e scambiando nodi.

Complessità -> $O(log_d(n))$ per l'esecuzione di muoviAlto.

delete(elem e) e deleteMin

Scambia il nodo v contenente l'elemento e con una qualunque foglia u sull'ultimo livello di T, poi elimina v. Ripristina l'ordinamento spostando u verso la posizione corretta con le procedure ausiliarie.

Complessità -> $O(log_d(n))$ per l'esecuzione di muoviAlto() o muoviBasso()

decreaseKey(elem e, chiave d)

Decrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta d. RIpristina l'ordinamento spingendo il nodo v verso l'alto e scambiando i nodi.

⚠ **Complessità -> $O(log_d(n))$ per l'esecuzione di muoviAlto()**

increaseKey(elem e, chiave d)

Aumenta il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta d.
RIpristina l'ordinamento spingendo il nodo v verso il basso e scambiando i nodi.

Complessità -> $O(log_d(n))$ per l'esecuzione di muoviBasso()

merge(CodaPriorità c_1, CodaPriorità c_2)

Abbiamo due metodi diversi:

1. **Costruire da zero una nuova coda**: Distruggendo le due iniziali.

Come:

- Generalizzazione della procedura heapify
- Rendo i sottoalberi della radice heap ricorsivamente e chiamo muoviBasso sulla radice.

Complessità -> $T(n) = dT(n/d) + O(dlog_d(n))$ dove: $n = |c_1| + |c_2|$
Teorema Master -> $T(n) = \Theta(n)$

2. **Inserendo ripetutamente**: Gli elementi della coda più piccola in quella più grande.

Come:

- Inseriamo ad uno ad uno tutti gli elemento della coda più piccola nella coda più grande.

Sia $k = min[|c_1|, |c_2|]$ e $n = |c_1| + |c_2|$
Eseguiamo quindi k inserimenti nella coda più grande.

Complessità -> $O(klog(n))$ con $n = |c_1| + |c_2|$

L'approccio conviene quindi per $klog(n) = o(n)$ -> $k = o(n/log(n))$

Osservazione

Nel caso peggiore entrambe le operazioni hanno un costo di $\Omega(n)$

Riepilogo

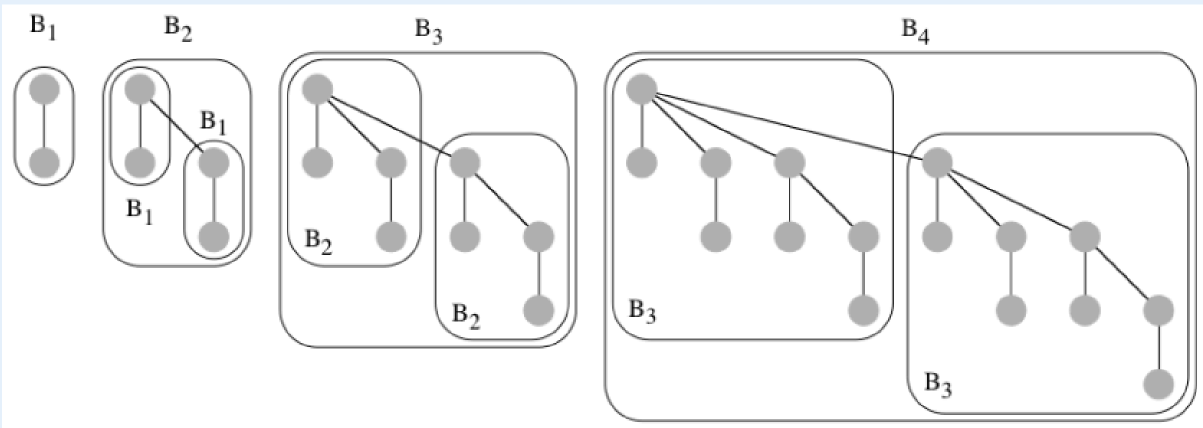
Per il momento non siamo ancora riusciti a creare una struttura in grado di eseguire tutte le operazioni in tempo minore di lineare.
Per riuscire a risolvere il problema dobbiamo introdurre nuovi tipi di heap:

Alberi Binomiali

Alberi binomiali:

Un albero binomiale B_i è definito ricorsivamente come segue:

1. B_0 consiste in un unico nodo.
2. Per $i > 0, B_{i+1}$ è ottenuto fondendo due alberi binomiali B_i ponendo la radice dell'uno come figlia della radice dell'altro.



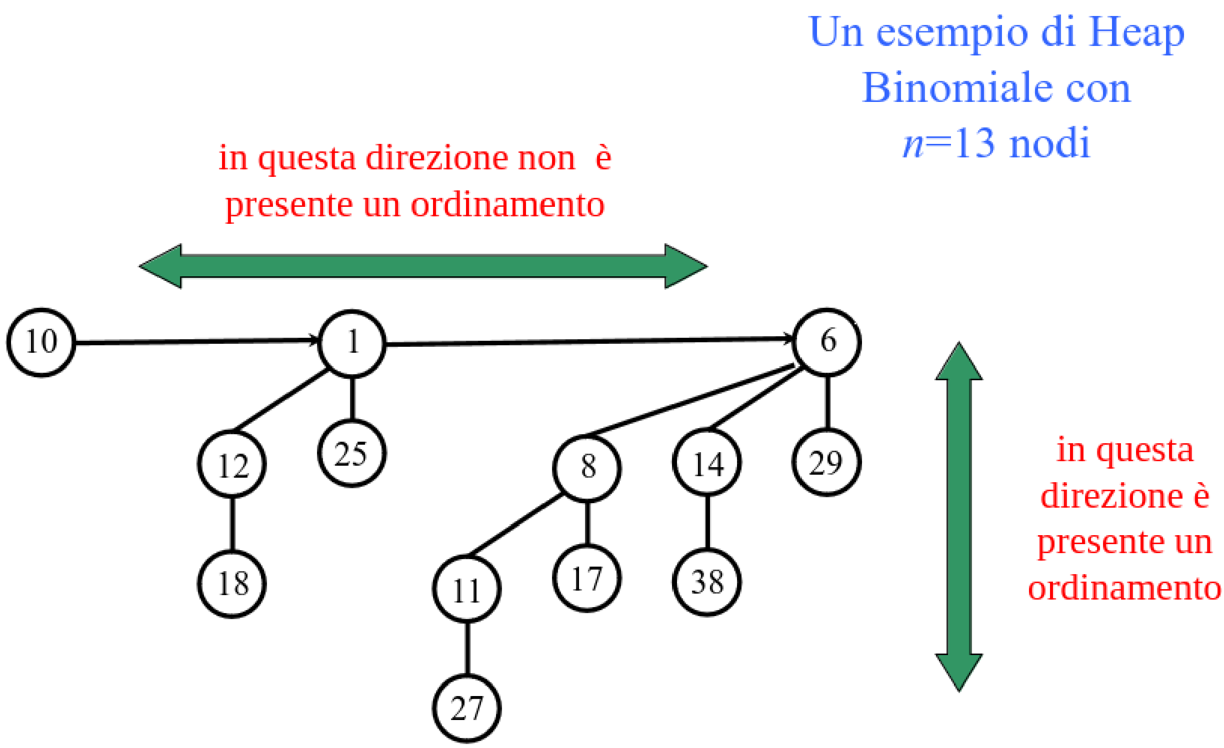
Proprietà:

1. Numero di nodo $(|B_h|) : n = 2^h$
2. Grado della radice: $D(n) = log_2(n)$
3. Altezza: $H(n) = h = log_2(n)$
4. Figli della radice: I sottoalberi radicati nei filgi della radice di B_h sono $B_0, B_1, \ldots, B_{h-1}$.

Heap Binomiali

Un heap binomiale è una foresta di alberi binomiali che gode delle seguenti proprietà:

- Unicità: Per ogni intero $i \geq 0$, esiste al più un B_i nella foresta.
- Contenuto informativo: Ogni nodo v contiene un lemento $elem(v)$ ed una chiave $chiave(v)$ presa da un dominio totalmente ordinato.
- Ordinamento a heap: $chiave(v) \geq chiave(parente(v))$ per ogni nodo v diversi da una delle radici.



> **Nota**
> Ci sta una correlazione tra gli alberi e il numero dei nodi presenti, infatti:
>
> $$13 = 2^0 + 2^2 + 2^3$$
>
> 13 in binario è 1101, quindi il nosto albero è composto da B_0, B_2, B_3.

Proprietà topologiche:

Dalla proprietà di unicità degli alberi binomiali che lo costituiscono, ne deriva che un heap binomiale di n elementi è formato dagli alberi binomiali $Bi0, Bi1, \dots, Bih$, dove $i0, i1, \dots, ih$ corrispondono alle posizioni degli 1. nella rappresentazione in base 2 di n.

Ne consegue che in un heap binomiale con n nodi, vi sono al più $\lfloor log \rfloor$ alberi binomiali, ciascuno con grado ed altezza O(log n).

Proprità ausiliaria

Utile per mantenere la struttura del heap binomiale:

```
Procedura ristruttura():  T(n): lineare nel numero di alberi binomiali in input
        i = 0
        while(esistono ancora due B_i) do
```

```
                Si fondono i due B_i per formare un albero B_i+1 ponendo la
radice con
                la chiave più piccola come genitore della radice con chiave
più grande
                i = i +1
```

Realizzazione

```
Classe HeapBinomiale implementa CodaPriorita:
dati:
        Una foresta H con n nodi, ciascuno contenente un lementeo di tipo
elem e e
        una chiave di tipo chiave presa da un universo totalmente ordinato.

operazioni:
        findMin() -> elem:
                Scorre le radici di H e restituisce le'lemento con chiave
minima

        insert(elem e , chiave k):
                Aggiunge ad H un nuovo B_0 con dati e e k. Ripristina poi la
proprietà
                di unicità in H mediante fusioni successive dei doppioni B_i

        delemteMin():
                Trova l'albero T_h con radice a chiave minima. Togliendo la
radice a T_h
                esso si spezza in h alberi binomiali, che vengono aggiunti
ad H.
                Ripristina poi la proprietà di unicità di H mediante fusioni
succesive
                dei doppioni B_i

        decreaseKey(elem e, chiave d):
                Decrementa di d la chiave nel nodo contenente l'elemento e.
Ripristina
                poi la proprietà dell'ordinamento a heap spiengendo verso
l'alto
                ripetutamente tramite ripetuti scambi di nodi.
```

```
        delete(elem e):
                Richiama decreaseKey(e, -inf) e poi deleteMin()

        increaseKey(elem e, chiave d):
                Richiama delete(e) e poi insert(elem,k+d), dove k è la
chiave associata
                all'elemento e.

        merge(codaprio.c_1,codaprio.c_2)->codaPrio:
                Unisce gli alberi c_1 e c_2 in un nuovo heap binomiale c_3.
                Ripristina poi la proprietà di unicità nell'heap binomiale
c_3 mediante
                fusioni successive dei doppioni B_i
```

Costo delle operazioni

Tutte le operazioni richiedono tempo $T(n) = O(log(n))$

Heap di Fibonacci (Fredman, Tarjan, 1987)

Definizione di Heap binomiale rilassato:

Si ottiente da un heap binomiale rilassando la proprietà di unicità dei B_i e usando un atteggiamento "pigro" nell'insert().

Gli heap di Fibonacci si ottendono da heap binomiali rilassati indebolendo la struttura dei B_i che non sono più necessariamente alberi binomiali.

	FindMin	Insert	Delete	DeleteMin	IncKey	DecKey	merge
d-heap	$O(1)$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(n)$
Heap Binomiali	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$
Heap di Fibonacci	$O(1)$	$O(1)$	$O(log(n))$ *	$O(log(n))$*	$O(log(n))$ *	$O(1)$*	$O(1)$

L'analisi del tempo svolta sul Heap di Fibonacci è ammortizzata.

Analisi ammortizzata

Il costo ammortizzato di un'operazione è il costo "medio" rispetto a una sequenza qualsiasi di operazioni.
E' diverso dal costo medio perché non ci sta nessuna distribuzione di probabilità.