Lezione_15_DeA

5.3. Implementazione di Priority Queue tramite liste

5.3.1. Lista non ordinata

Sia P una lista *non ordinata* di n entry `(PositionalList<Entry<K,V>>)`. I metodi della Priority Queue vengono implementati nelle seguenti maniere:

Metodo `min()`

```
v <- P.first();
e <- v.getElement();
while (P.after(v) != null) do {
     v <- P.after(v);
     if (v.getElement().getKey() < e.getKey()) then {
          e <- v.getElement();
     }
}
return e;
```

Complessità: $\Theta(n)$, infatti devo vedere tutte le entry per trovare quella con chiave minima.

Metodo `insert(k,x)`

```
e <- (k,x);
P.addLast(e);
return e;
```

Complessità: $\Theta(1)$, perché posso inserire l'entry dove voglio.

Metodo `removeMin()`

```
v <- P.first();
e <- v.getElement();
while (P.after(v) != null) do {
     v <- P.after(v);
     if (v.getElement().getKey() < e.getKey()) then {
          e <- v.getElement();
```

```
    }
}
P.remove(v);
return e;
```

Complessità: $\Theta(n)$ come il caso di `min`.

5.3.2. Lista non ordinata

Sia P una lista *ordinata* (in senso crescente) di n entry `(PositionalList<Entry<K,V>>)`. I metodi della Priority Queue vengono implementati nelle seguenti maniere:

Metodo `min()`

```
return P.first().getElement();
```

Complessità: $\Theta(1)$, infatti la entry con chiave minima è in testa.

Metodo `removeMin()`

```
v <- P.first();
P.remove(P.first());
return v;
```

Complessità: $\Theta(1)$, come il caso di `min`.

Metodo `insert(k,x)`

```
e <- (k,x);
v <- P.first();
while (v != null) do{
    if (v.getElement().getKey() >= k) then {
        P.addBefore(v,e);
        retunr e;
    }
    else{
        v <- P.after(v);
    }
}
P.insertLast(e);
return e;
```

Complessità: $\Theta(n)$, perché devo trovare la posizione in cui inserire la entry.

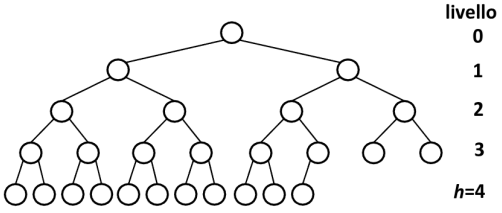
Riepilogo implementazione Priority Queue tramite Liste

Lista	`insert`	`min`	`removeMin`
Non ordinata	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Ordinata	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Per le strutture dati non discutiamo l'implementazione dei metodi `size` e `isEmpty` in quanto sono banali.

5.4. Implementazione di Priority Queue tramite Heap

> Un *albero binario completo* T è un albero binario di altezza $h \geq 0$ tale che:
>
> - $\forall i,\ 0 \leq i \leq h-1$, il livello i ha 2^i nodi (ovvero il massimo numero);
> - Al livello $h-1$ tutti i nodi interni sono alla sinistra delle eventuali foglie e hanno tutti due figli tranne, eventualmente, quello più a destra che, se ha un solo figlio, ha il figlio sinistro.



Proposizione

Un albero binario completo con n nodi ha altezza $h = \lfloor \log_2 n \rfloor$.

Osservo che al livello h (l'ultimo livello dell'albero) il numero di nodi è compreso tra 1 e 2^h.

$1 + \sum_{i=0}^{h-1}(2^i) \leq n \leq 2^h + \sum_{i=0}^{h-1}(2^i) = \sum_{i=0}^{h}(2^i)$

$\iff 1 + 2^h - 1 = 2^h \leq n \leq 2^{h+1} - 1$

$\iff h \leq log_2(n) < h + 1$

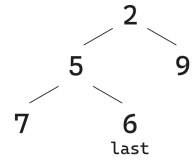
$\implies \lfloor log_2(n) \rfloor = h$

□

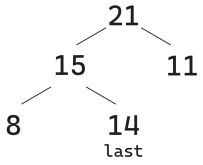
Osservazione

$\forall n \geq 1$ esiste un unico albero binario completo con n nodi.

> Un *min-heap* (o semplicemente *heap*) è un albero binario completo in cui ogni nodo v memorizza una entry e soddisfa la *heap-order property*: la chiave in v è *minore o uguale* della chiave in ciascun figlio di v.



> Un *max-heap* è uno heap in cui la definizione della heap-order property cambia: la chiave in v è *maggiore o uguale* della chiave in ciascun figlio di v.



> Il *nodo last* di uno heap di altezza h è il nodo più a destra del livello h.

Osservazione

Uno heap è caratterizzato da due proprietà: è un *albero binario completo* e segue la *heap-order property* (per ogni nodo).

5.4.1. Proprietà

Sia P uno heap con n entry. Dalla definizione si ricavano facilmente le seguenti proprietà.

1. Le chiavi incontrate lungo un cammino dalla radice verso le foglie formano una sequenza non decrescente;
2. Per qualsiasi discendente u di un nodo $v \in P$ si ha che $e_u.getKey() \geq e_v.getKey()$;
3. *La radice contiene una entry con chiave minima*;
4. Se le chiavi sono tutte distinte, la entry con chiave massima (e_{max}) si trova in una foglia di P.

La prima e seconda proprietà sono conseguenze immediate della heap-order property, mentre la terza è un corollario della seconda in quanto ogni nodo è discendente della radice.
Si prova la quarta proprietà per assurdo, infatti se e_{max} avesse un discendente e, si avrebbe che la chiave di e dovrebbe essere maggiore della chiave di e_{max}, che contraddice l'ipotesi.
□

Osservazione

Da queste proprietà si evince che uno heap non assicura necessariamente un ordinamento totale tra le entry, ma assicura l'ordinamento solo lungo percorsi radice-foglia.

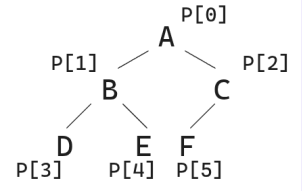
5.4.2. Implementazione di alberi binari su array

> Il seguente schema (*level numbering*) consente di mappare un albero binario su un array $P = P[0], P[1], \ldots$:
>
> - Radice -> $P[0]$;
> - Figli di $P[i]$ -> $P[2i+1], P[2i+2]$;
> - Padre di $P[i]$ -> $P\left[\left\lfloor \frac{i-1}{2} \right\rfloor\right]$.

Osservazione

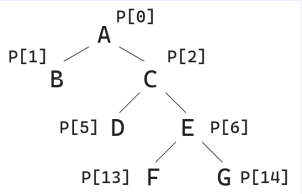
La rappresentazione è *space-efficient per alberi molto bilanciati* (ad esempio per alberi binari completi), ma non lo è affatto per alberi sbilanciati, come si vede nei seguenti esempi.

5.4.2.1. Albero molto bilanciato



`P ≡ A B C D E F`, il numero di nodi dell'albero è 6, come $|P| = 6$, quindi c'è la massima space-efficiency.

5.4.2.2. Albero molto sbilanciato



`P ≡ A B C [] [] D E [] [] [] [] [] [] F G`, il numero di nodi dell'albero è 7, mentre $|P| = 15$; ho una space-efficiency minima.

5.4.3. Implementazione di alberi binari completi su array

È facile dimostrare che usando il level numbering per mappare un albero binario completo con $n \geq 1$ nodi e altezza h su un array P, si ha che:

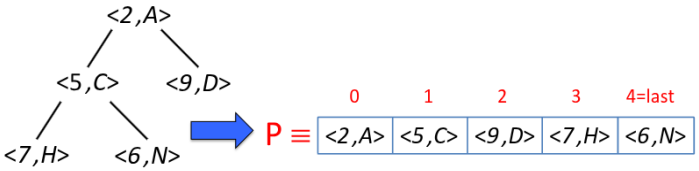
- $\forall i,\ 0 \leq i < h$, i 2^i nodi del livello i presi da sinistra a destra, sono mappati in $P[2^i - 1], P[2^i], \ldots, P[2^{i+1} - 2]$;
- I nodi del livello h, presi da sinistra a destra, sono mappati in $P[2^h], \ldots, P[n-1]$. Il nodo last è quindi mappato in $P[n-1]$.

Livello	Indici
0	0
1	$1 \div 2$
2	$3 \div 6$
$\vdots$	$\vdots$
j	$2^j - 1 \div 2^{j+1} - 2$
$\vdots$	$\vdots$
$h-1$	$2^{h-1} - 1 \div 2^h - 2$
h	$2^h - 1 \div n - 1$

> **Osservazione**
>
> Il level numbering definisce quindi una corrispondenza 1:1 (biunivoca) tra array e alberi binari completi.

5.4.4. Implementazione di heap su array

Se non diversamente specificato, assumeremo sempre che uno heap sia realizzato tramite array.



5.4.4.1. Esercizio

> Sia P un array di n entry $P[0], P[1], \ldots, P[n-1]$ le cui chiavi sono in ordine non decrescente. P rappresenta uno heap? Motivare la risposta.

In generale è vero dato che se vedo l'albero binario completo che corrisponde a P, le entry nel nodo associato a $P[i]$ avrà chiave minore o uguale delle entry nei nodi associati ai figli di $P[i]$ ($P[2i+1]$, $P[2i+2]$) grazie all'ordinamento di P.

Osservazione

Non è vero il viceversa, ovvero che uno heap sia sempre associato a un array ordinato.