Appunti lezione – Capitolo 10
Code con priorità e insiemi disgiunti

Alberto Montresor

12 Agosto, 2019

1 Correttezza di heapBuild()

Invariante di ciclo: All'inizio di ogni iterazione del ciclo **for**, ogni nodo $i+1, i+2, \ldots, n$ è la radice di uno heap.

Dimostrazione.

- **Inizializzazione**: All'inizio, $i = \lfloor n/2 \rfloor$. Per prima cosa, dimostriamo che tutti i nodi da $\lfloor n/2 \rfloor + 1$ fino a n sono foglie. Supponiamo per assurdo che $\lfloor n/2 \rfloor + 1$ non sia una foglia. Allora $2\lfloor n/2 \rfloor + 2$ fa parte dello heap. Ma tale elemento è uguale a $n+1$ o a $n+2$, che sono entrambi fuori dal vettore. Quindi tutti i nodi $i+1 \ldots n$ sono heap (banali).
- **Conservazione**: Per ipotesi induttiva, supponiamo che al passo i, tutti i nodi $i+1, i+2, \ldots, n$ siano radici di heap. E' quindi possibile applicare maxHeapRestore() al nodo i, perché $2i < 2i+1 < n$ sono entrambi radici di heap. Quindi al termine del ciclo, tutti i nodi $i, i+1, \ldots, n$ sono radici di heap.
- **Conclusione**: Al termine, $i = 0$. Quindi il nodo 1 è una radice di uno heap.

□

2 Correttezza di heapsort()

Invariante di ciclo: Al passo i, il sottovettore $A[i+1 \ldots n]$ è ordinato; $A[1 \ldots i] \leq A[i+1 \ldots n]$ e $A[1]$ è la radice di un vettore heap di dimensione i.

Dimostrazione. Per induzione su i.

- **Inizializzazione**: Al passo $i = n$, il sottovettore $A[n+1 \ldots n]$ è vuoto. Dopo aver eseguito heapBuild(), $A[1]$ è la radice di un vettore heap di dimensione n.
- **Conservazione**: Ad ogni passo i, il nodo in posizione $A[1]$ è il massimo fra i nodi rimanenti $A[1 \ldots i]$ ed è minore di tutti i nodi in $A[i+1 \ldots n]$.
- **Conclusione**: L'algoritmo termina quando $i = 1$. Questo significa che il vettore è dato dai nodi $A[2 \ldots n]$ è ordinato, e $A[1] \leq A[2 \ldots n]$. Quindi il vettore è ordinato.

□

3 Complessità di heapsort()

Vengono compiuti n passi, ognuno dei quali richiede $O(\log n)$ passi. Quindi il costo totale è $O(n \log n)$. Notare la differenza con l'analisi di complessità di heapBuild(): poiché heapBuild() funziona dal basso, solo una volta viene costruito uno heap di altezza $\log n$. In questo caso, invece la funzionalità di maxHeapRestore() viene eseguita $n/2$ volte con alberi dell'altezza $h = O(\log n)$. Quindi il costo totale è $O(n \log n)$.

4 Liste + Euristica sul peso

Dato una struttura dati Merge-Find contenente inizialmente n oggetti, vogliamo valutare il costo delle operazioni merge().
Consideriamo solo operazioni fra insiemi disgiunti (l'unione fra insiemi uguali possono essere "rilevate" in tempo $O(1)$ con un semplice confronto fra i rappresentanti). Ci possono essere al più $n-1$ di queste operazioni (ogni operazione riduce di uno il numero degli insiemi, fino a raggiungere un insieme unico).
Utilizzando l'euristica con peso, ogni volta che cambia rappresentante un oggetto si ritroverà in un insieme che contiene almeno il doppio degli elementi dell'insieme originale.
Utilizziamo il metodo dell'analisi ammortizzata basata sui crediti, e assegniamo ad ogni oggetto un numero di crediti pari a $\lfloor \log n \rfloor$ al momento della creazione dell'insieme corrispondente.
Ad ogni operazione merge(), consumiamo un credito per ognuno degli oggetti che cambia rappresentante; ogni singolo oggetto può essere coinvolto in al più $\lfloor \log n \rfloor$ (per via del raddoppiamento), quindi un totale di $O(n \log n)$ crediti è sufficiente a pagare qualunque sequenza di unioni.

5 Complessità: Euristica sul rango

Lemma 1. Un albero MFSet bilanciato in altezza con radice x ha almeno $2^{rank[x]}$ nodi.

Dimostrazione. Procediamo per induzione sul numero di operazioni effettuate.

- All'inizio (operazioni = 0) tutti gli alberi hanno rango 0 e un elemento; poiché $2^0 = 1$, la proprietà è rispettata.
- Consideriamo un'operazione merge(x, y): siano X e Y gli insiemi rappresentati da x e y, rispettivamente. Sia:
  - $rank[x], rank[y]$ il rango dei nodi x, y prima dell'unione;
  - $|X|, |Y|$ il numero di nodi negli insiemi X, Y prima dell'unione;
  - $rank[r]$ il rango della radice dell'insieme risultante dopo l'unione
  - $|R| = |X \cup Y| = |X| + |Y|$ il numero di nodi nell'insieme risultante dopo l'unione;

  Ci sono tre casi:
  - $rank[y] < rank[x]$: abbiamo che $rank[r] = rank[x]$; inoltre, per ipotesi induttiva:

    $$|R| = |X \cup Y| = |X| + |Y| \geq 2^{rank[x]} + 2^{rank[y]} > 2^{rank[x]} = 2^{rank[r]}$$

    (questo perché $2^{rank[y]} > 0$)
  - $rank[x] < rank[y]$: simmetrico del precedente;
  - $rank[x] = rank[y]$: l'albero con radice x prende y come nuovo figlio, e il suo rank viene incrementato di uno: $rank[r] = rank[x] + 1$. Dall'ipotesi induttiva, abbiamo che:

    $$|R| = |X \cup Y| = |X| + |Y| \geq 2^{rank[x]} + 2^{rank[x]} = 2^{rank[x]+1} = 2^{rank[r]}$$

□

Corollario 1. L'altezza di un albero in una struttura Merge-Find con n elementi è limitata superiormente da $\lfloor \log n \rfloor$.

Dimostrazione. Sia x la radice dell'albero e X l'insieme di nodi. Dal lemma precedente, $|X| \geq 2^{rank[x]}$. Applicando il logaritmo ad entrambi i lati, dimostriamo il corollario. □

Da cui segue che il numero di passi per effettuare un'operazione find() è limitato superiormente da $\log n$, dove n è il numero totale di elementi.

6 QuickUnion + Euristica sul rango + Compressione dei cammini

E' possibile dimostrare (ma non lo faremo qui) che il costo ammortizzato dell'esecuzione di m operazioni merge-find in un insieme di n elementi è $O(m \cdot \alpha(n))$, dove $\alpha(n)$ (funzione inversa di Ackermann) è una funzione che cresce molto lentamente (praticamente costante per tutti gli usi pratici): $\alpha(n) = 4$ per $n \approx 10^{80}$, che è la stima del numero di atomi presenti nell'universo.