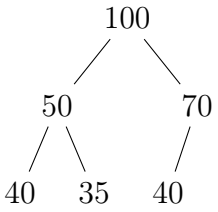
Appunti Algoritmi e Strutture Dati

Luca Seggiani

10 Aprile 2024

1 Struttura dati Heap

Un heap è un albero binario quasi bilanciato con la proprietà di avere i nodi dell'ultimo livello addossati a sinistra, e sopratutto che in ogni sottoalbero l'etichetta della radice è maggiore o uguale a quella di tutti i discendenti (la cosiddetta *heap property*). Vediamo un'esempio:



La visita in ordine anticipato di tale albero restituisce l'array:

$$100, 50, 70, 40, 35, 40$$

Il primo elemento dell'array sarà sempre la radice dell'heap. Si nota inoltre che il figlio sinistro di un qualsiasi elemento di indice i si trova con $2i + 1$, il figlio destro con $2i + 2$, e infine il padre con $\frac{i-1}{2}$.
Sugli heap sono definite operazioni di inserimento e estrazione dell'elemento maggiore. L'estrazione prevede complessità di $O(\log n)$ nel caso medio, e $O(n)$ nel caso peggiore (albero linearizzato). Vediamo intanto una possibile implementazione:

```
1 class Heap {
2   int* h;
3   int last; //indice dell'ultimo elemento
4   void up(int);
5   void down(int);
6   void exchange(int i, int j) {
7     int k = h[i]; h[i] = h[j]; h[j] = k;
```

```
  }
public:
  Heap(int);
  ~Heap(int);
  void insert(int);
  int extract();
}
```

Vediamo nello specifico costruttore e distruttore.

```
Heap::Heap(int n) {
  h = new int[n];
  last = -1;
}
Heap::~Heap() {
  delete[] h;
}
```

è chiaro come la struttura dati viene realizzata effettivamente con un vettore di interi, vettore che rappresenta la vista in ordine anticipato dell'albero che forma l'heap.

Inserimento in Heap

La strategia adottata per l'inserimento è quella di memorizzare l'elemento nella prima posizione libera dell'array, e far poi risalire l'elemento tramite scambi figlio padri finché l'heap property non è rispettata:

```
void Heap::insert(int x) {
  h[++last] = x;
  up(last);
}
```

Notiamo come viene sfruttata la funzione up() definita precedentemente per far risalire l'elemento. Vediamone l'implementazione:

```
void Heap::up(int i) {
  if(i > 0) {
    if(h[i] > h[(i - 2) / 2] {
      exchange(i, (i - 2) / 2);
      up((i - 2) / 2);
    }
  }
}
```

dove si sfrutta la proprieta per cui il padre di i è sempre $\frac{i-1}{2}$ per assicurare l'heap property.

Estrazione da Heap

La strategia adottata per l'estrazione è quella di restituire il primo elemento dell'array, mettere l'ultimo elemento al suo posto (e decrementare il valore last), e a questo punto far scendere l'elemento tramite scambi padre figlio per mantenere l'heap property. In codice:

```
int Heap::extract() {
  int r = h[0];
  h[0] = h[last--];
  down(0);
  return r;
}
```

Vediamo l'implementazione della down:

```
void Heap::down(int i) {
  int child = 2 * i + 1;
  if(child == last) {
    if(h(son) > h(i))
      exchange(i, last);
  } else if(child < last) {
    if(h[child] < h[child + 1])
      child++;
    if(h(child) > h(i)) {
      exchange(i, child);
      down(child);
    }
  }
}
```

Applicazioni dell'Heap

La struttura dati heap torna particolarmente utile nella realizzazione del tipo di dato astratto **coda con proprità**. Essa si tratta di una coda in cui gli elementi contengono, oltre all'informazione, un'intero che ne definisce la priorità. In caso di estrazioni l'elemento da estrarre dovrà essere ovviamente quello con maggiore priorità.

2 Heap Sort

Il meccanismo dell'heap permette di realizzare algoritmi di ordinamento. Basterà infatti creare un heap a partire da un certo vettore, estrarre il primo elemento (che andremo a disporre in fondo all'heap) e ripetere finché il vettore non è completamente ordinato. L'ordinamento risulterà come conseguenza dell'operazione ripetuta di estrazione, che ricordiamo comporta la preservazione dell'*heap property* attraverso scambi successivi. Più schematicamente, si trasforma un'array in un heap, e si segue n volte l'estrazione scambiando ogni volta il primo elemento dell'array con quello puntato da last. In codice:

```
void heapSort(int* vett, int n) { //n = dimensione dell'array
  buildHeap(vett, n);
  int i = n - 1;
  while(i > 0) {
```

```
    extract(vett, i);
  }
}
```

Vediamo quindi la funzione buildHeap() usata per costruire l'heap. Inizieremo dall'eseguire la funzione down() sulla prima metà degli elementi dell'array (gli elementi della seconda metà saranno tutti foglie, verificare dal disegno). Si esegue poi down() sull'altra metà, partendo dall'elemento centrale e tornando indietro fino al primo.

```
void buildHeap(int* vett, int n) {
  for(int i = (n / 2) - 1; i >= 0; i--) {
    down(A,i, n);
  }
}
```

Questo richiederà una rielaborazione sia della down che della extract, che riportiamo in seguito:

```
//RIPORTA VARIAZIONI (E MAGARI COMPRENDILE!)
```

La complessità dell'inserimento a questo punto sarà di $O(n)$, mentre la complessità delle estrazioni ripetute darà una complessità finale di $O(n \log n)$, che non è affatto male.