







4° lezione algoritmi

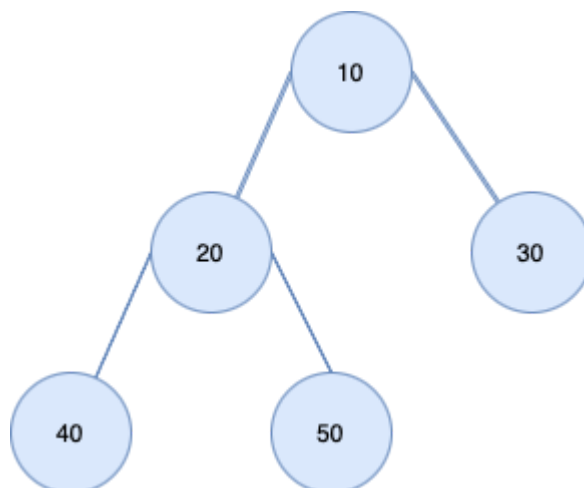
	Ⓛ Luigi Domenico Castano
 Due Date	@October 11, 2024
 Materia	Algoritmi e Laboratorio
 Status	Done

HEAP(MUCCHIO)

Si tratta di una versione più avanzata di una struttura FIFO. La differenza, tuttavia, risiede nel fatto che gli elementi si dispongono in maniera più strutturata. L'obiettivo è avere una struttura di altezza minore e che si distribuisca in larghezza.

L'heap viene utilizzato per implementare una coda con priorità, ossia una coda di tipo FIFO in cui gli elementi vanno insieme ad una informazione detta priorità dell'elemento. Verranno prima estratti, quindi, gli elementi con maggiore priorità.

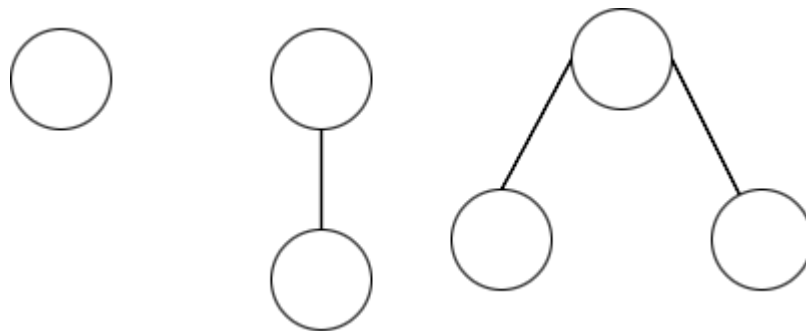
Dal punto di vista strutturale la coda con priorità può essere strutturata con un albero binario posizionale.



DEFINIZIONE FORMALE

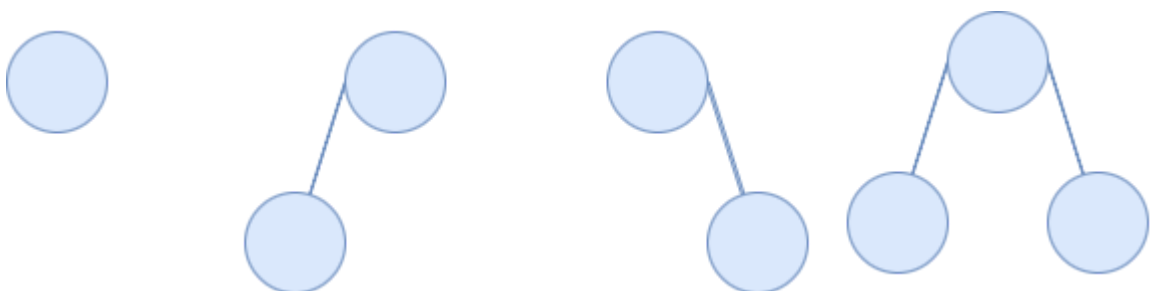
Un heap è un albero binario, è una struttura dati rappresentata da un grafo in cui ogni nodo può avere al massimo due figli e un "genitore". Può essere posizionale o non posizionale.

▼ ALBERO NON POSIZIONALE



- **singolo nodo**
- **un solo figlio**
- **due figli**

▼ ALBERO POSIZIONALE

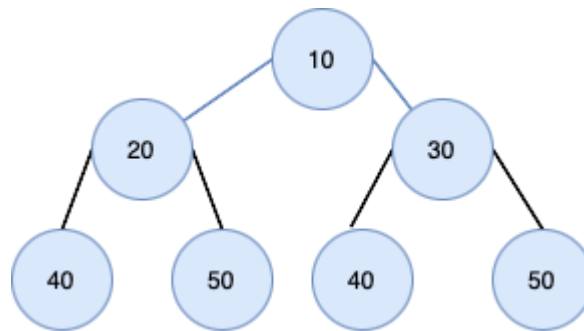


- **singolo nodo**
- **figlio dx**
- **figlio sx**
- **due figli dx e sx**

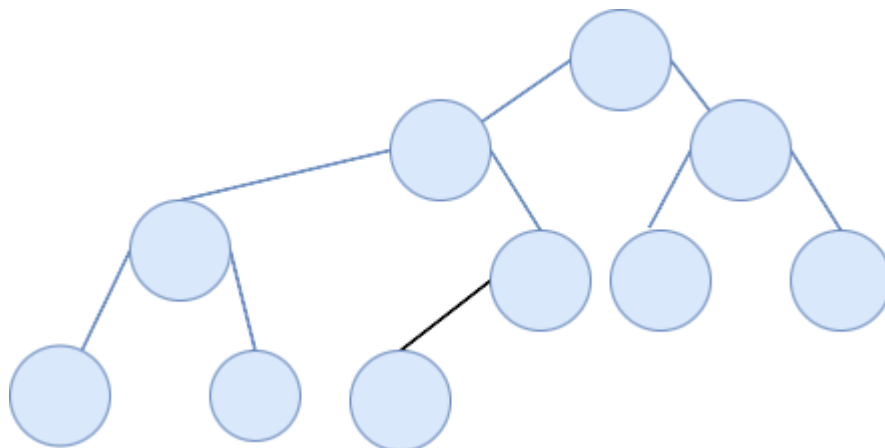
▼ ALBERO COMPLETO

Si ha quando tutti i nodi hanno due figli (dx e sx). Per ogni nodo il sotto albero sinistro deve avere lo stesso numero di nodi del sotto albero destro se si verifica questa condizione si dice che l'albero è bilanciato. **Tutti i nodi devono essere alla stessa altezza e tutti devono avere due figli o anche si**

ha un albero binario completo quando tutti i livelli sono completi cioè non manca nessun nodo in ogni livello.



Si ha una variazione che sta nella possibilità che **l'ultimo livello dell'albero può non essere completo. Gli elementi si tolgono da sx verso dx.**



La struttura heap è un albero binario posizionale completo.

Un albero binario si dice completo se in ogni livello ogni nodo ha due figli, oppure se ogni nodo ha due figli e tutte le foglie sono allo stesso livello.

In un albero binario completo, il numero di nodi è sempre dispari. Se numero di nodi di un albero binario completo, il numero di foglie è:

$$\frac{n}{2} + 1$$

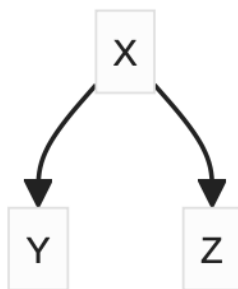
Se ogni nodo contiene una chiave e la struttura dati è sempre un albero binario completo, per aggiungere una chiave devo aggiungere molte chiavi. Quindi ammetteremo che l'ultimo livello può non essere completo; tuttavia, tutti gli elementi dell'ultimo livello sono allineati a sinistra (o inseriti da sinistra a destra).

Quindi:

Un heap è un albero binario posizionale completo a meno dell'ultimo livello, a patto che tutti i suoi elementi sono allineati a sinistra.

COME SONO INSERITI GLI ELEMENTI IN UN HEAP?

Sono inseriti secondo una proprietà chiamata ordinamento parziale.



Dato un nodo con due figli, la priorità di x y, z x dev'essere maggiore o uguale della priorità di y e

.

- $P(X) \geq P(Y)$
- $P(X) \geq P(Z)$

La priorità del padre è sempre più alta di quella dei due figli.

MIN-HEAP

- $key(x) \leq key(y)$
- $key(x) \leq key(z)$

MAX-HEAP

Nel Max-heap, invece, vale:

$$\begin{aligned}P(x) &\geq P(y) \\ P(x) &\geq P(z)\end{aligned}$$

In entrambi i casi possiamo dire sicuramente che l'altezza è data da:
 $h \in (O \log n)$

Procedure in un min-heap

Potremmo eseguire queste operazioni:

1. **Heapify**
2. **find-minimum;**
3. **extract-minimum;**
4. **insertion;**
5. **decrease-key;**
6. **delete-key.**

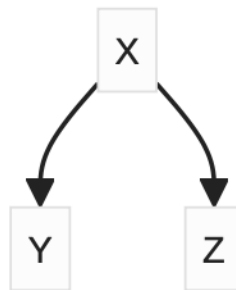
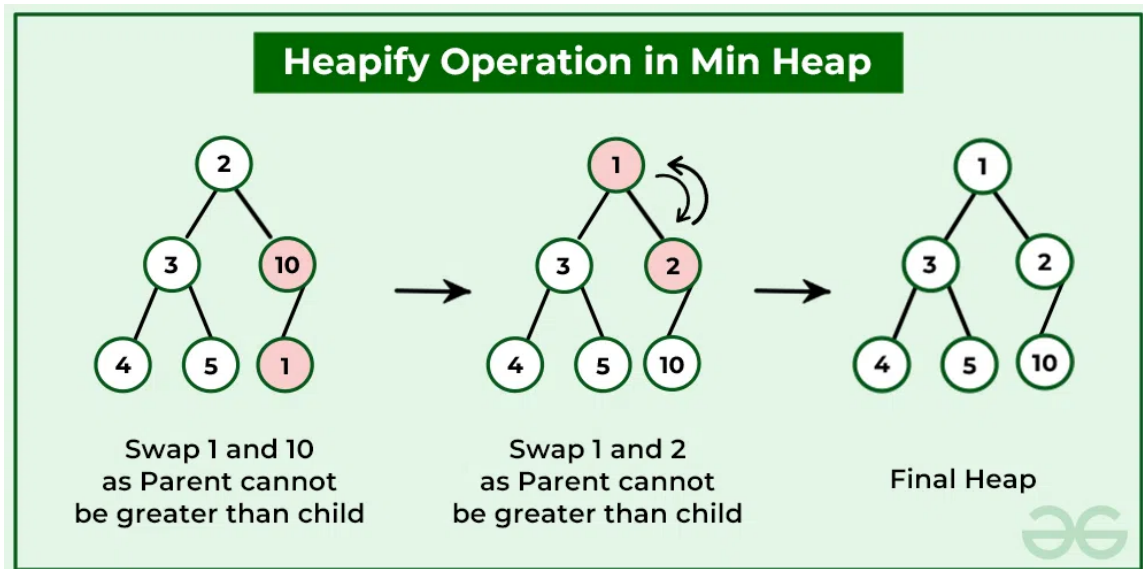
Nel caso in cui implementiamo la coda con priorità, la prima operazione può essere eseguita in tempo costante, perché la radice dell'albero è il nodo con chiave minore.

Le altre operazioni richiedono più operazioni.

▼ Heapify

La procedura di heapify rende un heap qualcosa. In particolare, sia x un nodo con due sottoalberi y, z con al di sotto heap. Chiamando heapify su x , rendiamo l'albero con radice x un heap. Se c'è una violazione tra x, y, z allora basta che scambiamo la chiave x con y o con z .

Avremo che la violazione è rimossa tra x, y, z e richiamiamo la funzione sul nodo in cui abbiamo scambiato il valore: questo perché non è detto che questo sottoalbero abbia mantenuto la proprietà della struttura heap.



▼ find-minimum

Nel caso del min heap si applica la medesima procedura e nel caso in cui ci siano due figli dx e sx si sceglie il minimo o il massimo elemento tra Y e Z e si mette in X

```

heapify(H, x):
  y = left(x)
  z = right(x)
  min = x
  if(y != null and key(y) < min) then
    min = y
  if(z != null and key(z) < min) then
    min = z
  if min != x then
    swap(x, min)
    heapify(H, min)
  
```

Tale procedura ha complessità, nel caso peggiore, pari a equazione di ricorrenza è:

$$T(n) = \frac{n}{3}$$

perché la funzione `heapify` viene chiamata solamente o a destra o a sinistra. Tuttavia, questa analisi è parzialmente corretta, perché stiamo supponendo che il numero di foglie sia uguale a destra o a sinistra. Nel caso peggiore, in realtà, tutte le foglie sono nel sottoalbero sinistro. Inoltre, il numero di foglie di un albero di dimensione n è $\frac{n}{2} + 1$, quindi se $\frac{n}{3}$ è la dimensione dell'albero, allora avremo che l'ultimo livello del sottoalbero destro e il sottoalbero sinistro hanno lo stesso numero di nodi, ossia $\frac{n}{3}$. Quindi sarebbe più corretto considerare:

$$T(n) = T\left(\frac{2}{3}n\right) + O(1)$$

▼ **extract-minimum;**

Nel min-heap per rimuovere il minimo, che risiede nella radice, è necessario mantenere la struttura generale dell'albero. Un modo facile per farlo è prendere un nodo dall'ultimo livello, dato che esso può anche non essere bilanciato e quindi la sua rimozione non provoca alcun problema. Una volta rimosso, mettiamo il valore del nodo rimosso nella radice. Quindi chiamiamo `heapify()`

```
removeMin(heap):  
    if heap == NULL THEN  
        return null
```

```
    Step 1: Get the minimum value from the root  
    min_value = heap[0]
```

```
    Step 2: Replace the root with the last element  
    heap[0] = heap[heap.size - 1]
```

```
    Step 3: Reduce the size of the heap  
    reduceSize(heap)
```

```
Step 4: Restore the min-heap property
heapify(heap, 0)

return min_value
```

▼ insertion;

Nel momento in cui inserisco una chiave k nella struttura heap, creiamo intanto il nuovo nodo in cui inserire la chiave k ed eseguiamo ciò. Successivamente, controllo se la chiave inserita è più piccola della chiave contenuta nel nodo padre. In tal caso, scambio i valori tra nodo figlio e nodo padre. Continuo così fino ad arrivare alla radice.

```
insert(H, k):
    x = newNode(H) // identifico nuovo nodo x che contiene
    key(x) = k
    p = parent(x)
    while(p != NULL and key(p) > key(x)) do
        swap(x, p)
    x = p
    p = parent(x)
```

Tale procedura ha una complessità **$O(\log n)$**

▼ decrease-key;

Il decrease key prevede che io prenda un nodo qualsiasi e diminuisca il valore della sua priorità, ad esempio diventa il 7 diventa 3. Notiamo che se diminuisco una chiave, il problema non si pone con i suoi figli, ma solo con il genitore.

Ma come abbiamo fatto nell'inserimento, scambiare un figlio con il padre non porta a problemi verso il basso.

▼ delete-key.

Nel momento in cui devo eliminare un nodo, prendo una foglia e scambio i valori con il nodo target. Rimuovo il nodo foglia, e richiamo `heapify()`.

Procedure per il max-heap

Per il max-heap le procedure sono le medesime, basta invertire i segni delle diseguaglianze.

Differenze tra albero binario di ricerca e heap

A meno dell'operazione di minimo o massimo, l'albero binario di ricerca non ha complessità differenti rispetto alla struttura heap. Possiamo facilmente, tuttavia, implementare la struttura heap come un array. Gli elementi verranno inseriti sequenzialmente per livello, ossia avremo che il primo elemento dell'array è la radice, i due successivi sono i suoi figli, i quattro successivi a questi ultimi saranno quelli che costituiscono il livello 2 e così via.

La domanda che ci poniamo è la seguente:

come possiamo ricostruire la struttura heap a partire dall'array?