Strutture Dati

Lezione 12
L'heap

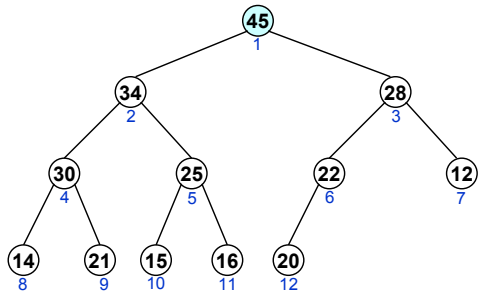
Oggi parleremo di …

- Heap
  - ADT
  - rappresentazione
  - operazioni
    - inserimento
    - cancellazione
- Code con priorità

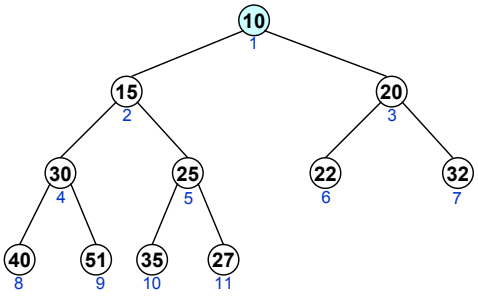
Heap: definizione

- Un **albero massimo** è un albero in cui il valore della chiave di ogni nodo non è minore del valore della chiave dei suoi figli
  - un **heap massimo** è un albero binario completo che è anche un albero massimo.
- Un **albero minimo** è un albero in cui il valore della chiave di ogni nodo non è maggiore del valore della chiave dei suoi figli
  - un **heap minimo** è un albero binario completo che è anche un albero minimo.

Heap max: esempio



Heap min: esempio



Il tipo di dati astratto *MaxHeap*

```
Struttura MaxHeap
   oggetti: un albero binario completo di n > 0 elementi organizzati in
        modo che il valore in un nodo qualsiasi sia maggiore o uguale a
        quelli contenuti nei figli.

   funzioni: per ogni heap ∈ MaxHeap, item ∈ elemento, n, max_size
        interi

     MaxHeap Create(max_size)   ::=   crea un heap vuoto capace di
                                      contenere max_size elementi.
     Booleano HeapFull(heap,n)  ::=   if(n == max_size) return TRUE
                                      else return FALSE.
     Booleano HeapEmpty(heap,n) ::=   if(n > 0) return FALSE
                                      else return TRUE.
     MaxHeap Insert(heap,item,n) ::=  if(!HeapFull(heap,n)) inserisci
                                      item in heap e restituisci l'heap
                                      risultante, else return errore.
     elemento Delete(heap,n)    ::=   if(!HeapEmpty(heap,n)) return
                                      l'elemento massimo dell'heap ed
                                      eliminalo dall'heap, else errore.
end MaxHeap
```

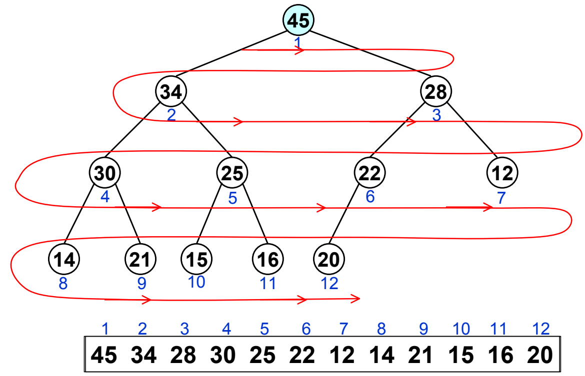
Heap: rappresentazione

- Un heap può essere implementato
  - come un albero con puntatori
  - come un array

Un heap è un albero binario completo

Conviene utilizzare un array!

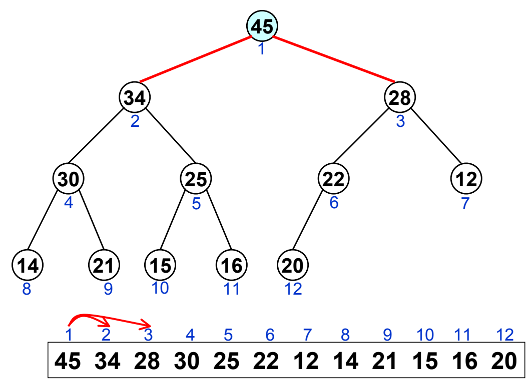
Heap: rappresentazione



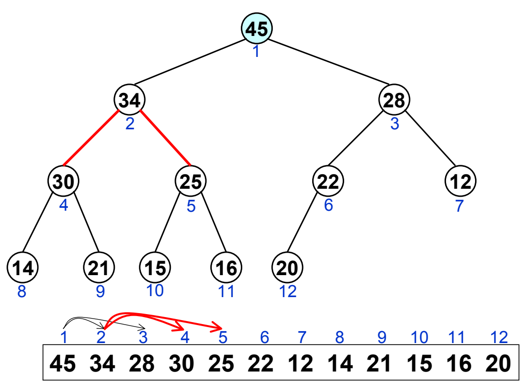
Heap: rappresentazione

- Un heap può essere implementato come un array in cui:
  - la radice dell'heap si trova nella posizione 1 dell'array
  - se il nodo *i* dello heap si trova nella posizione *i* dell'array,
    - il figlio sinistro di *i* si trova nella posizione 2*i*
    - il figlio destro di *i* si trova nella posizione 2*i*+1

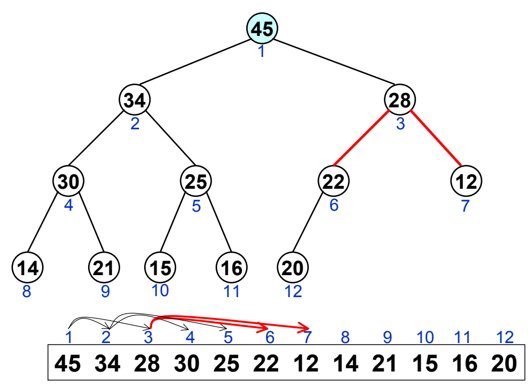
Heap: rappresentazione



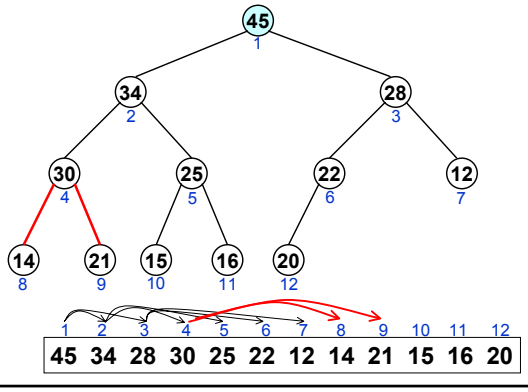
Heap: rappresentazione



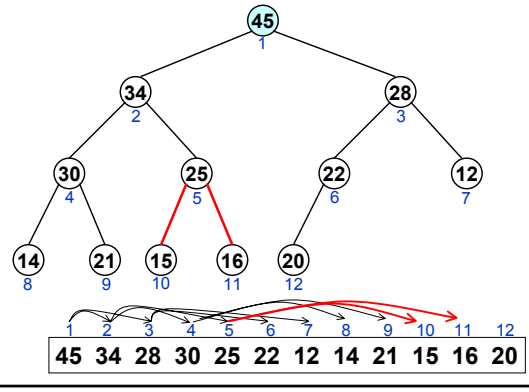
Heap: rappresentazione



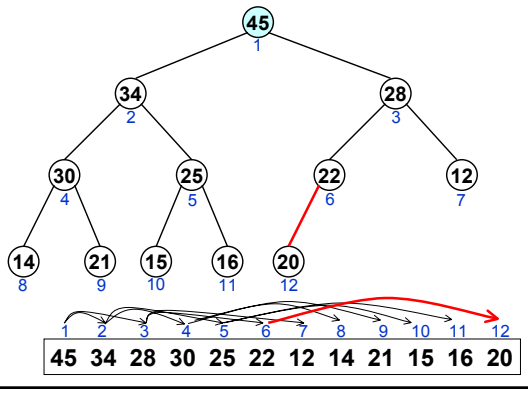
Heap: rappresentazione



Heap: rappresentazione



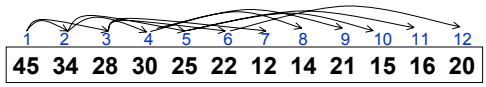
Heap: rappresentazione



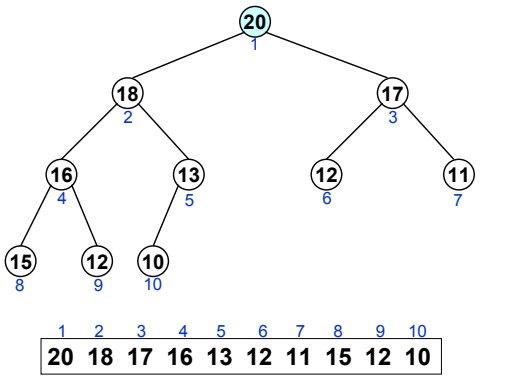
Heap: proprietà

- Un **albero heap** è un albero binario completo tale che per ogni nodo *i*:
  - entrambi i nodi *j* e *k* figli di *i* sono NON maggiori di *i*.
- Un **array *A*** è un heap se

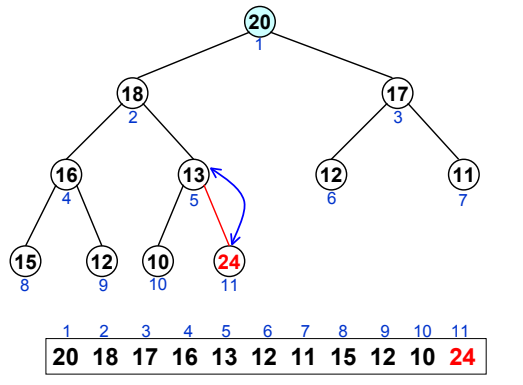
$$A[i] \geq A[2i] \quad \text{e} \quad A[i] \geq A[2i+1]$$



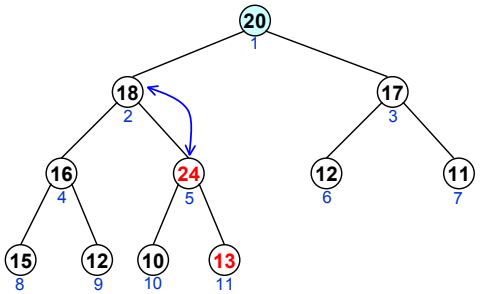
Heap: inserimento



Heap: inserimento

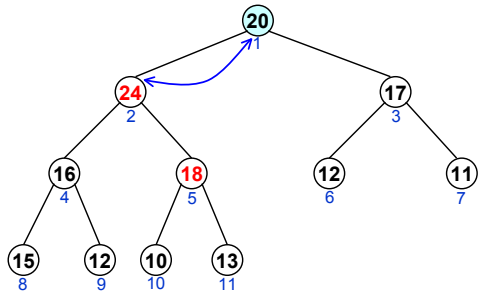


Heap: inserimento



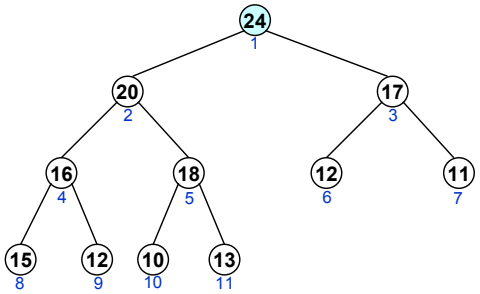
1	2	3	4	5	6	7	8	9	10	11
20	18	17	16	24	12	11	15	12	10	13

Heap: inserimento



1	2	3	4	5	6	7	8	9	10	11
20	24	17	16	18	12	11	15	12	10	13

Heap: inserimento



1	2	3	4	5	6	7	8	9	10	11
24	20	17	16	18	12	11	15	12	10	13

Heap: inserimento

- Si inserisce il nuovo elemento in fondo all'heap.
- Per determinarne la corretta posizione nell'heap, si percorre l'albero verso l'alto finché
  - si raggiunge la radice
  - una posizione *i* tale che il valore nella posizione del padre *i*/2 sia almeno pari al valore da inserire.

Heap: inserimento

```
#define MAX_ELEMENTI 200  // dimensione dell'heap + 1
#define Heap_full(n)      (n == MAX_ELEMENTI-1)
#define Heap_empty(n)     (!n)

typedef struct {
        int chiave;
        // altri elementi
        } elemento;

elemento heap[MAX_ELEMENTI];
int n = 0;
```

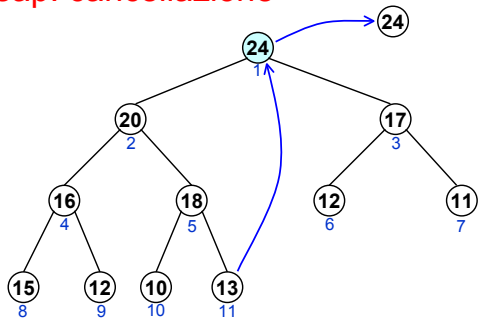
```
void insert_max_heap(elemento item, int *n)
{
  int i;

  if(Heap_full(*n))  { printf("L'heap e' pieno\n");
                       return; }
  i = ++(*n);
  while ((i!=1) && (item.chiave > heap[i/2].chiave))
    {
      heap[i] = heap[i/2];
      i = i/2;
    }
  heap[i] = item;
}
```

$O(\log_2 n)$

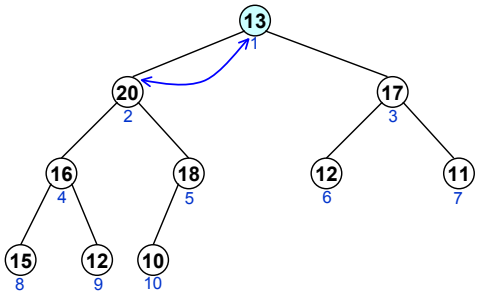
Altezza dell'heap è $\lceil \log_2 (n+1) \rceil$

Heap: cancellazione



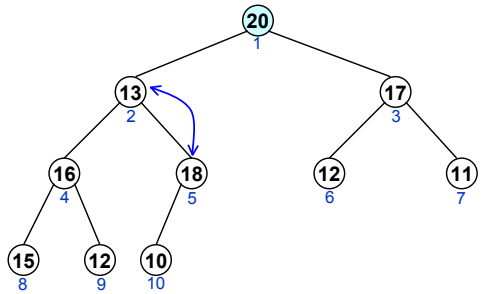
1	2	3	4	5	6	7	8	9	10	11
24	20	17	16	18	12	11	15	12	10	13

Heap: cancellazione



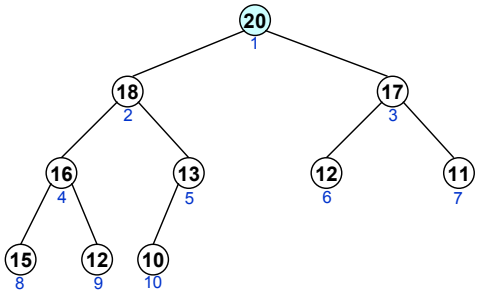
1	2	3	4	5	6	7	8	9	10
13	20	17	16	18	12	11	15	12	10

Heap: cancellazione



1	2	3	4	5	6	7	8	9	10
20	13	17	16	18	12	11	15	12	10

Heap: cancellazione



1	2	3	4	5	6	7	8	9	10
20	18	17	16	13	12	11	15	12	10

Heap: cancellazione

- Si "sostituisce" il nodo radice con l'ultimo nodo dell'heap.
- Per ripristinare l'heap, si percorre l'albero verso il basso
  - si confronta il nodo padre con i suoi figli
  - si scambiano gli elementi fuori posto.

```
elemento delete_max_heap(int *n)
{
 elemento item, temp;
 int padre, figlio;

 item = heap[1];
 temp = heap[(*n) --];
 padre = 1;
 figlio = 2;
 while (figlio<=*n) {
     if ((figlio < *n) &&
       heap[figlio].chiave<heap[figlio+1].chiave))
           figlio ++;
     if (temp.chiave >= heap[figlio].chiave)
           break;
     heap[padre] = heap[figlio];
     padre = figlio;
     figlio *= 2;
   }
 heap[padre] = temp;
 return (item);
}
```

$O(\log_2 n)$

Code con priorità

- Una coda con priorità elimina l'elemento con la priorità più elevata o meno elevata

Rappresentazione	Inserimento	Cancellazione
Array non ordinato	$O(1)$	$O(n)$
Lista concatenata non ordinata	$O(1)$	$O(n)$
Array ordinato	$O(n)$	$O(1)$
Lista concatenata ordinata	$O(n)$	$O(1)$
Heap massimo	$O(\log_2 n)$	$O(\log_2 n)$