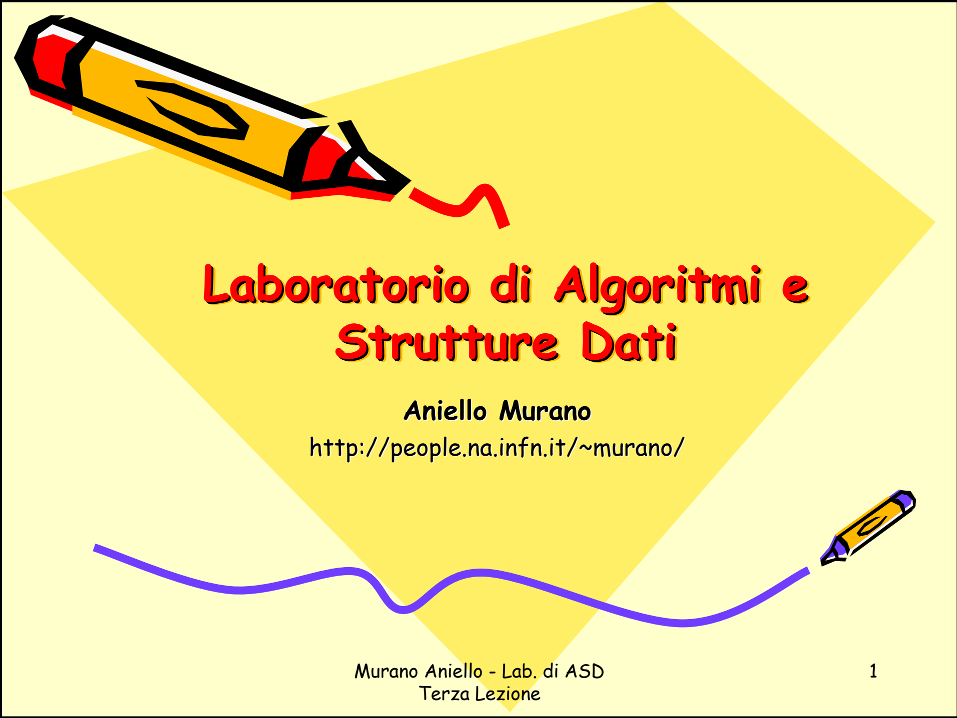
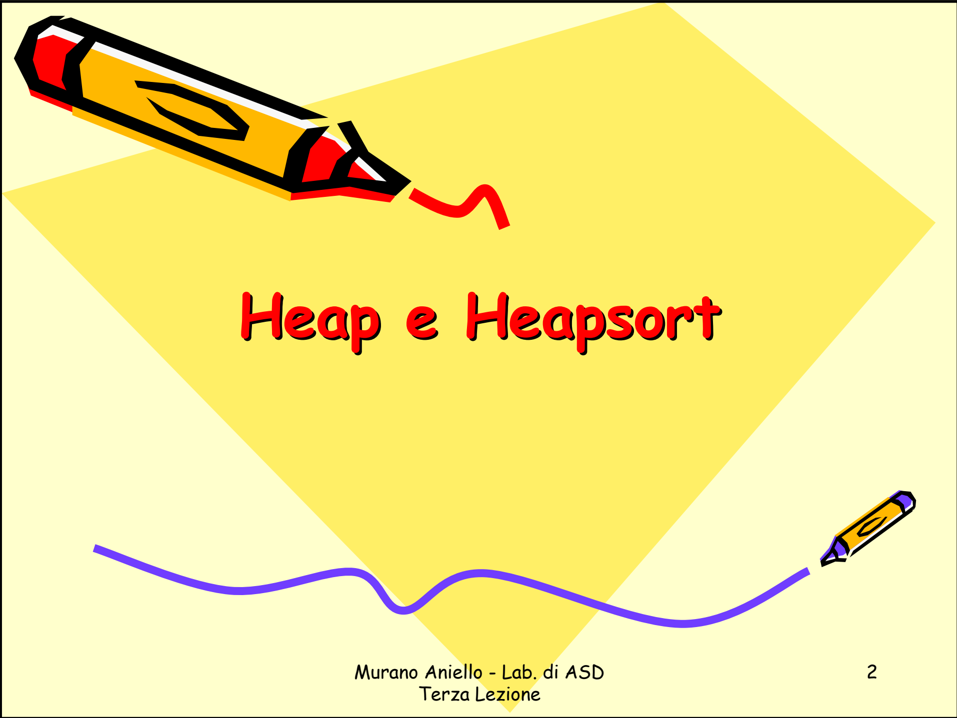

Laboratorio di Algoritmi e
Strutture Dati
Aniello Murano
http://people.na.infn.it/~murano/
Murano Aniello - Lab. di ASD
Terza Lezione
1


Heap e Heapsort
Murano Aniello - Lab. di ASD
Terza Lezione
2

Algoritmi di ordinamento

- Insertion Sort
- Quicksort
- Heapsort

Insertion Sort

- L'insertion Sort è uno algoritmo di ordinamento molto efficiente per ordinare un piccolo numero di elementi
- L'idea di ordinamento è simile al modo che un giocatore di bridge potrebbe usare per ordinare le carte nella propria mano.
- Si inizia con la mano vuota e le carte capovolte sul tavolo
- Poi si prende una carta alla volta dal tavolo e si inserisce nella giusta posizione
- Per trovare la giusta posizione per una carta, la confrontiamo con le altre carte nella mano, da destra verso sinistra. Ogni carta più grande verrà spostata verso destra in modo da fare posto alla carta da inserire.

Documentazione per Insertion Sort

- **Complessità di Tempo**
  - Nel caso peggiore la complessità asintotica di Insertion Sort è $O(n^2)$. Nel caso migliore (vettore ordinato) bastano N-1 confronti.
- **Complessità di spazio:**
  - La struttura dati utilizzata per implementare l'algoritmo è un ARRAY monodimensionale, contenente i valori da ordinare, di conseguenza la complessità di spazio è $O(n)$.

Quicksort

- Il Quicksort è un algoritmo di ordinamento molto efficiente.
- Il suo tempo asintotico medio è $O(n*logn)$
- È un metodo di ordinamento ricorsivo: si sceglie un elemento del vettore (pivot) e si partiziona il vettore in due parti. La prima partizione contiene tutti gli elementi minori o uguali al pivot, mentre la seconda gli elementi maggiori. A questo punto basta riapplicare il metodo (reinvocare ricorsivamente la procedura che lo realizza) ai due sotto-vettori costituiti dalle partizioni per ordinarli. (Si veda per una simulazione la lezione del Prof. Benerecetti "QuickSort.pdf")

Complessità del Quicksort

- Il caso migliore si ha quando la scelta del pivot crea, ad ogni livello della ricorsione, due partizioni di *uguale dimensione*. In tal caso si converge in k = logN passi. Al passo k si opera su M=2^k vettori di lunghezza L=N/(2^k)
- Il numero di confronti ad ogni livello e' L*M, pertanto, il numero globale di confronti e' **1*N+2*N/2 + 4*N/4 +... + N*1 = NlogN**
- Il caso più sfortunato si ha quando ad ogni passo si sceglie un pivot tale che un sotto-vettore ha lunghezza 1. In tal caso si converge in k=N-1 passi. Pertanto, il numero globale di confronti e': **N-1+N-2+ ... +2+1 = N(N/2)**

Heapsort

- L'Heapsort è un algoritmo di ordinamento molto efficiente:
- Come l'insertion Sort e il Quicksort, l'Heapsort ordina sul posto
- Meglio dell'Insertion Sort e del Quicksort, il running time dell'Heapsort è O(nlogn) nel caso peggiore
- L'algoritmo di Heapsort basa la sua potenza sull'utilizzo di una struttura dati chiamata **Heap**, che gestisce intelligentemente le informazioni durante l'esecuzione dell'algoritmo di ordinamento.

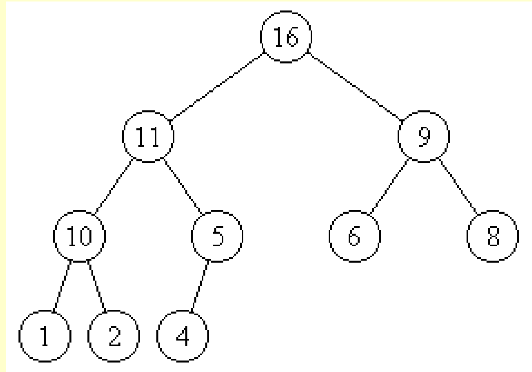
Heap

- La struttura dati Heap (binaria) è un array che può essere visto come un albero binario completo
- Proprietà fondamentale degli Heap è che il valore associato al nodo padre è sempre maggiore o uguale a quello associato ai nodi figli
- Un Array A per rappresentare un Heap ha bisogno di due attributi:
  - Lunghezza dell'array
  - Elementi dell'Heap memorizzati nell'array

Organizzazione dell'array

- La radice dell'Heap è sempre memorizzata nel primo elemento dell'array.
- Dato un nodo i, il nodo padre, figlio sinistro e destro di i possono essere calcolati velocemente nel modo seguente:
  - int left(int i) { return 2*i+1;}
  - int right(int i) { return 2*i+2;}
  - int parent (int i) {return (i-1)/2;}
- N.B. Nel C il primo elemento di un vettore A è A[0]. Nel libro di testo "Algoritmi e Strutture Dati", il primo elemento di un vettore A è invece A[1] e i valori precedenti sono dati dalle seguenti pseudo-funzioni:
  - Parent (i) return $\lfloor i/2 \rfloor$
  - Left (i) return 2i
  - Right(i) return 2i+1

Esempio di Heap



Heap con 10 vertici

0	1	2	3	4	5	6	7	8	9
16	**11**	**9**	**10**	**5**	**6**	**8**	**1**	**2**	**4**

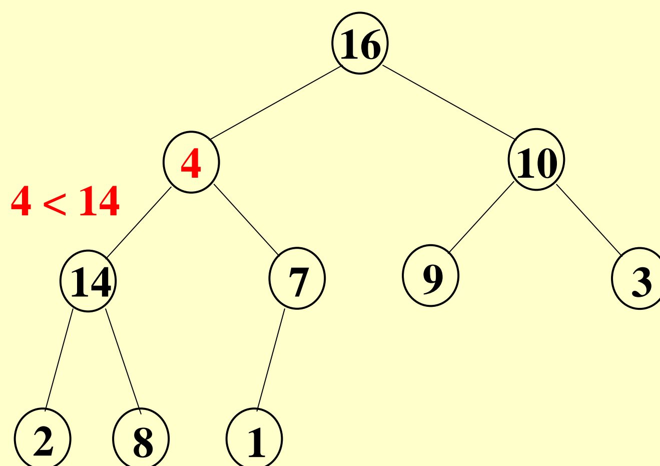
Heapify

- Una subroutine molto importante per la manipolazione degli Heap è Heapfy. Questa routine ha il compito di assicurare il rispetto della proprietà fondamentale degli Heap. Cioè, che il valore di ogni nodo non è inferiore di quello dei propri figli.
- Di seguito mostriamo una funzione ricorsiva Heapify che ha il compito di far scendere il valore di un nodo che viola la proprietà di Heap lungo i suoi sottoalberi.

Implementazione di Heapify

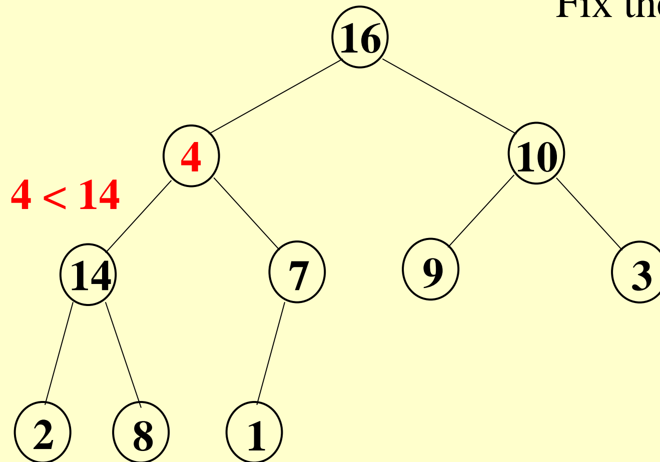
```
void Heapify(int A[MAX], int i)
{
    int l,r,largest;
    l = left(i);
    r = right(i);
    if (l < HeapSize && A[l] > A[i])
            largest = l;
    else largest = i;
    if (r < HeapSize && A[r] > A[largest])
            largest = r;

    if (largest != i) {
            swap(A, i, largest);
            Heapify(A, largest);
    }
}
```


Example of heapify

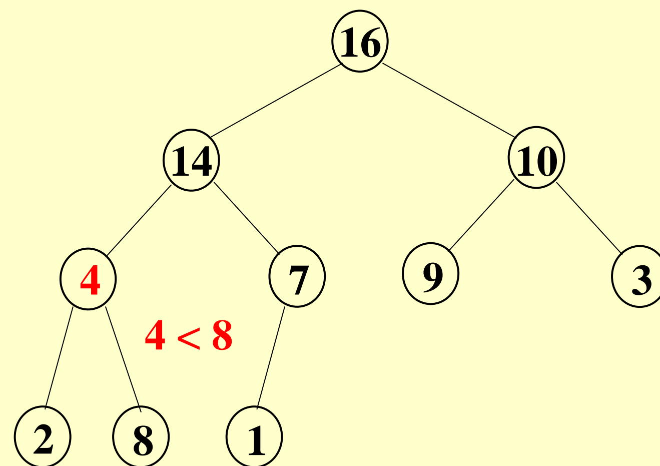


Example of heapify

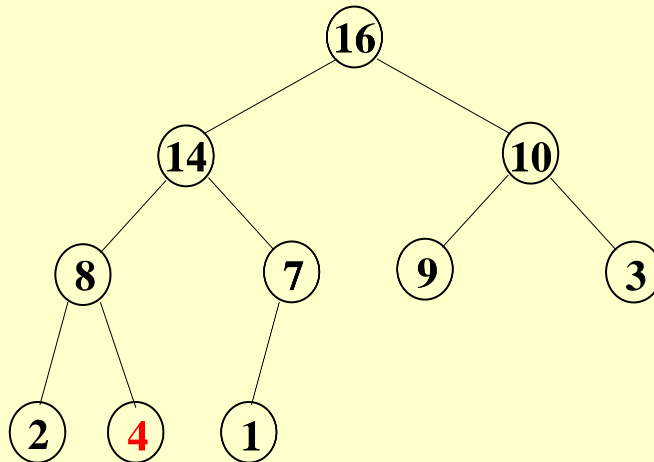
Fix the heap



Example of heapify



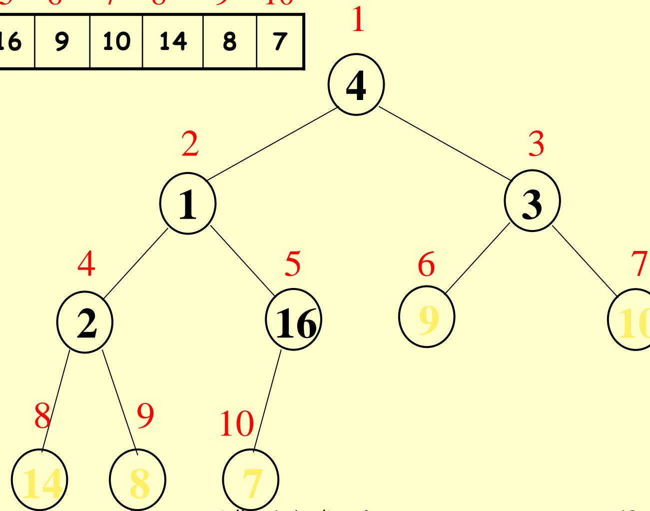
Example of heapify

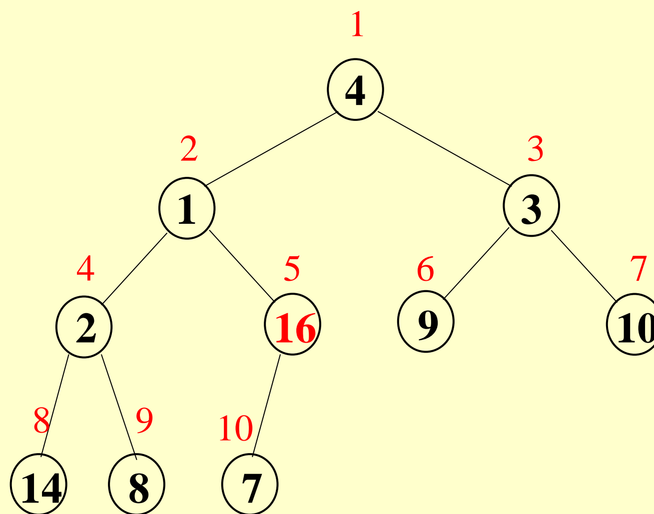


Costruire un Heap

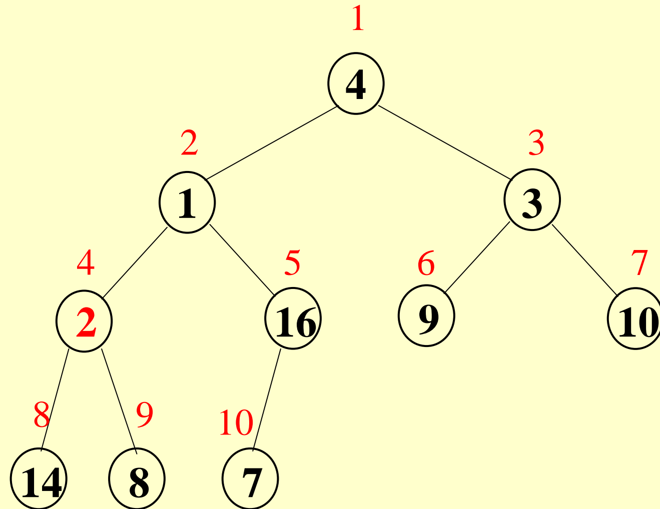
- La seguente procedura serve a costruire un Heap da un array:

```
void BuildHeap(int A[MAX])
{
int i;
    HeapSize = ArraySize;
    for (i=ArraySize/2; i>=0; i--)
      Heapify(A, i);
}
```

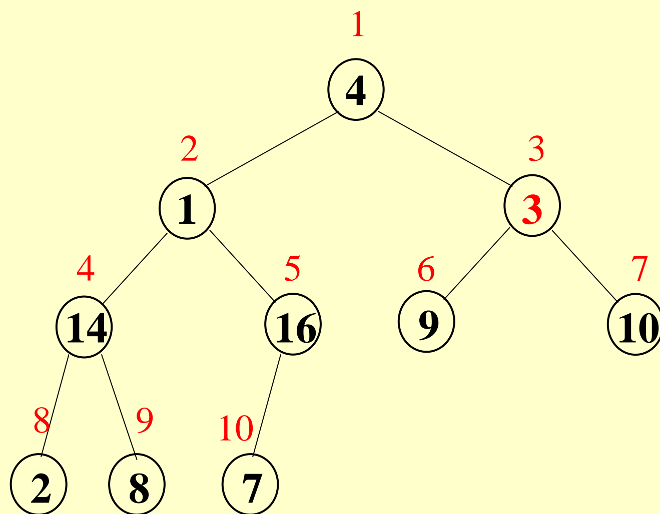

Example: building a heap (1)
1 2 3 4 5 6 7 8 9 10
4 1 3 2 16 9 10 14 8 7
1
4
2
1
3
3
4
2
5
16
6
9
7
10
8
14
9
8
10
7
Murano Aniello - Lab. di ASD
Terza Lezione
19


Example: building a heap (2)
1
4
2
1
3
3
4
2
5
16
6
9
7
10
8
14
9
8
10
7
Murano Aniello - Lab. di ASD
Terza Lezione
20

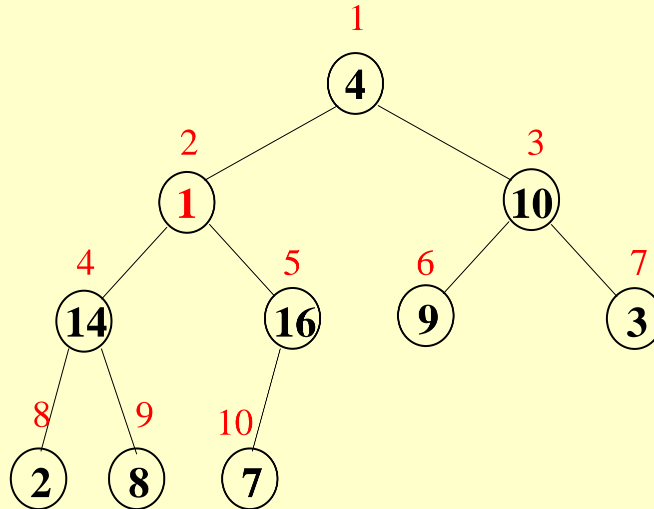
Example: building a heap (3)



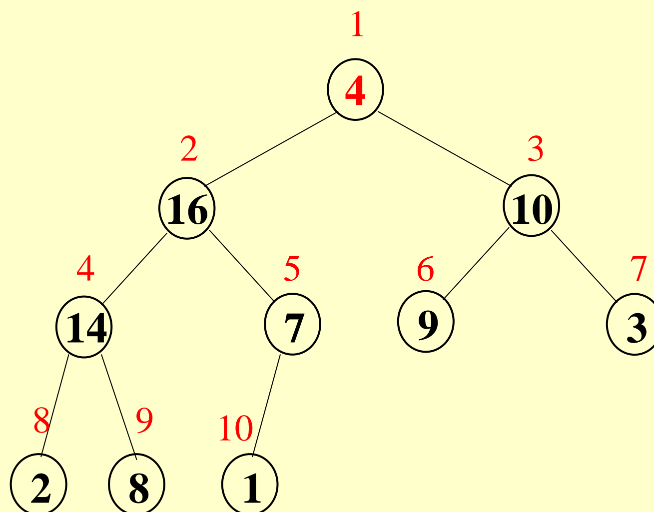
Example: building a heap (4)



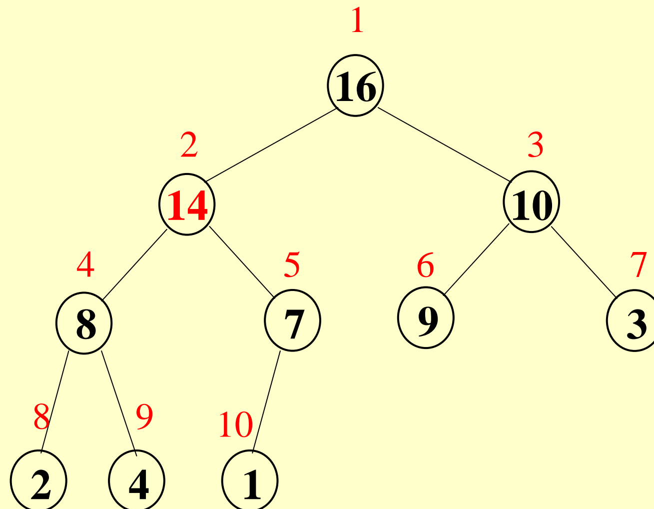
Example: building a heap (5)



Example: building a heap (6)



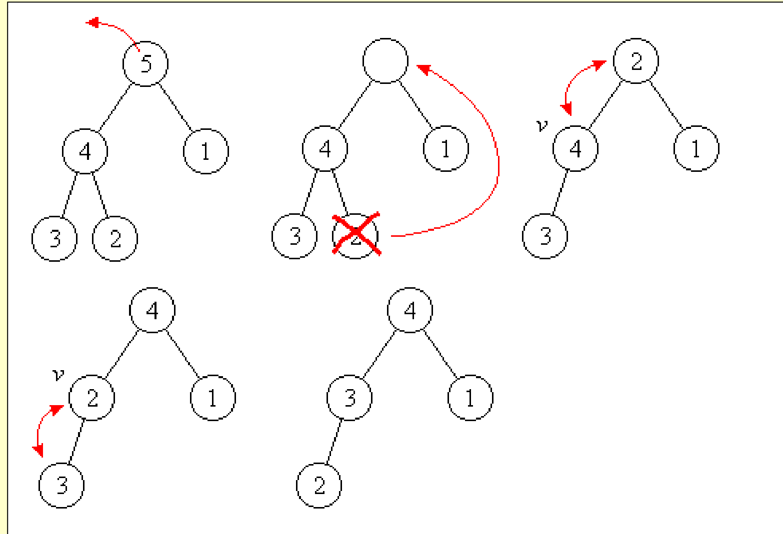
Example: building a heap (7)



Funzione HeapSort

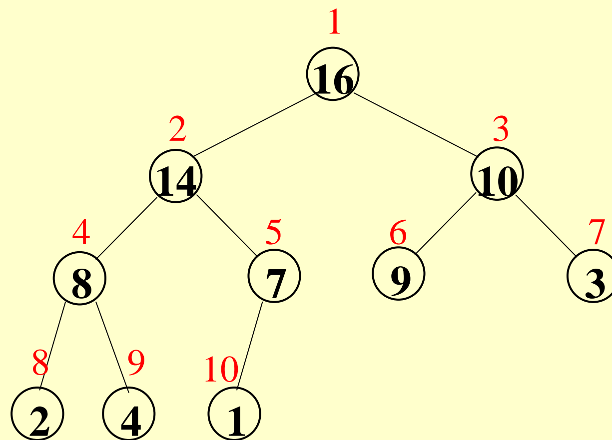
```
void HeapSort(int A[MAX])
{
int i;
    BuildHeap(A);
    for (i=ArraySize-1; i>=1; i--) {
          swap(A, 0, i);
          HeapSize--;
          Heapify(A, 0);
    }
}
```

Simulazione

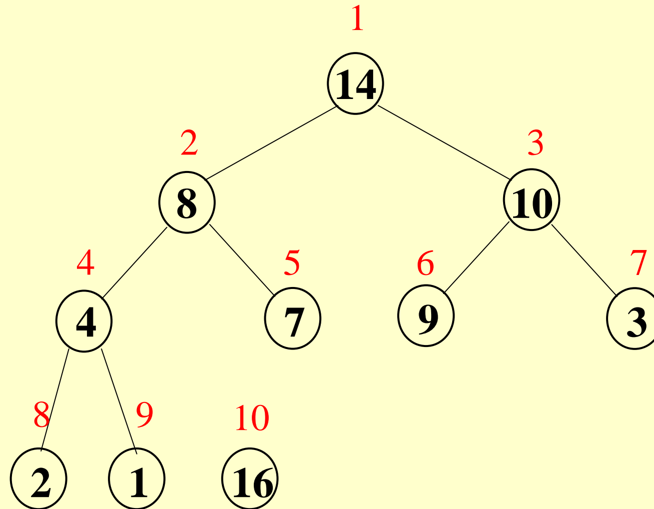


Example: Heap-Sort

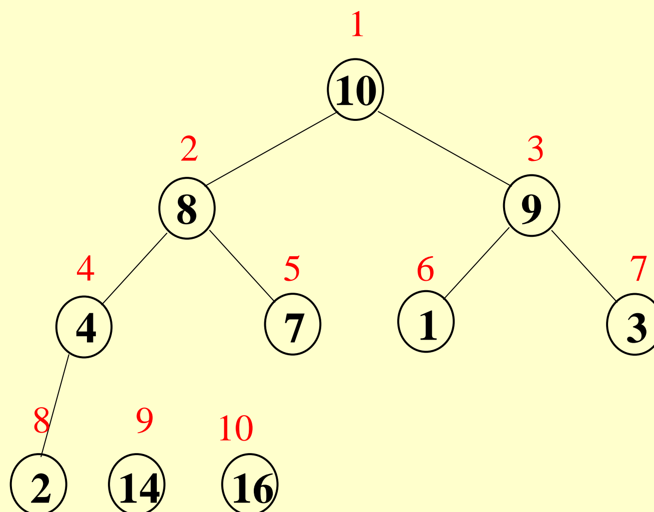
16	14	10	8	7	9	3	2	4	1

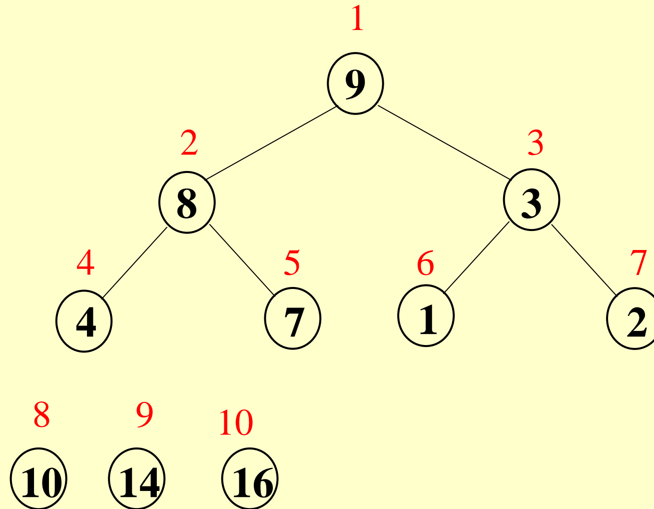


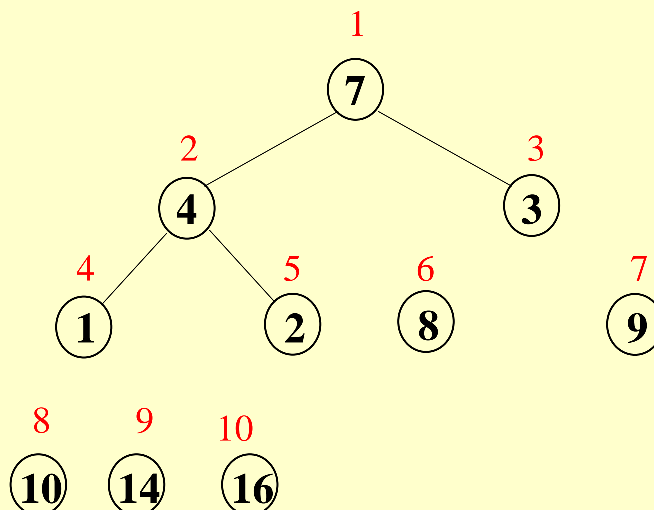
Example: Heap-Sort (2)



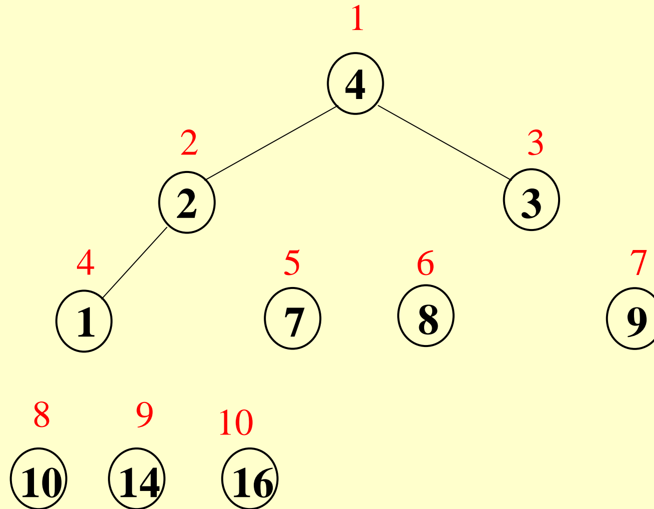
Example: Heap-Sort (3)



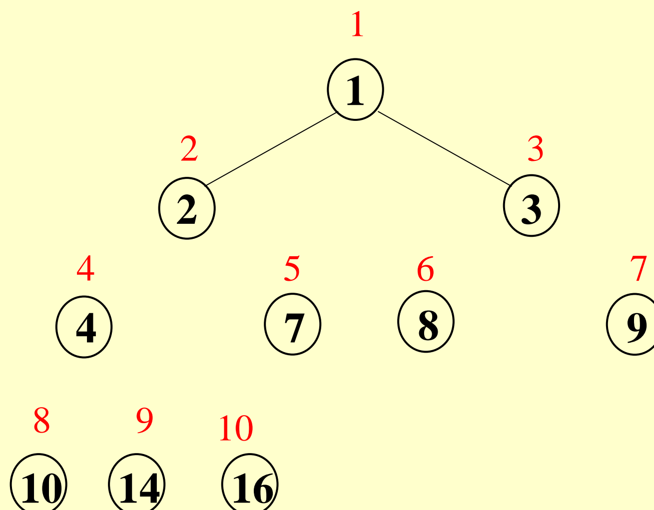

Example: Heap-Sort (4)
1
9
2
8
3
3
4
4
5
7
6
1
7
2
8
10
9
14
10
16
Murano Aniello - Lab. di ASD
Terza Lezione
31


Example: Heap-Sort (5)
1
7
2
4
3
3
4
1
5
2
6
8
7
9
8
10
9
14
10
16
Murano Aniello - Lab. di ASD
Terza Lezione
32

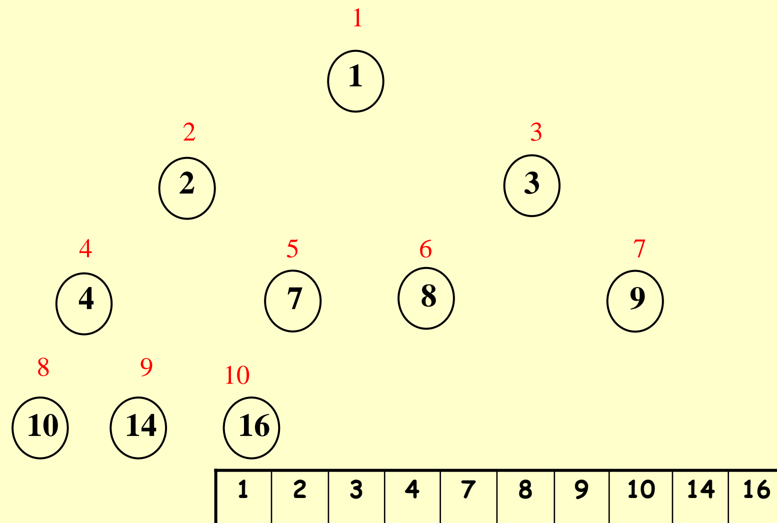
Example: Heap-Sort (6)



Example: Heap-Sort (7)



Example: Heap-Sort (8)



1	2	3	4	7	8	9	10	14	16

Algoritmo di HeapSort

```
#include <stdlib.h>
#define MAX 20
int ArraySize, HeapSize, tot;

int left(int i)   { return 2*i+1;}
int right(int i) { return 2*i+2;}
int p(int i)       {return (i-1)/2;}

void swap(int A[MAX], int i, int j)
    {int tmp = A[i];
    A[i] = A[j];
    A[j] =tmp;}

void Heapify(int A[MAX], int i);
void BuildHeap(int A[MAX]);
void HeapSort(int A[MAX]);
```

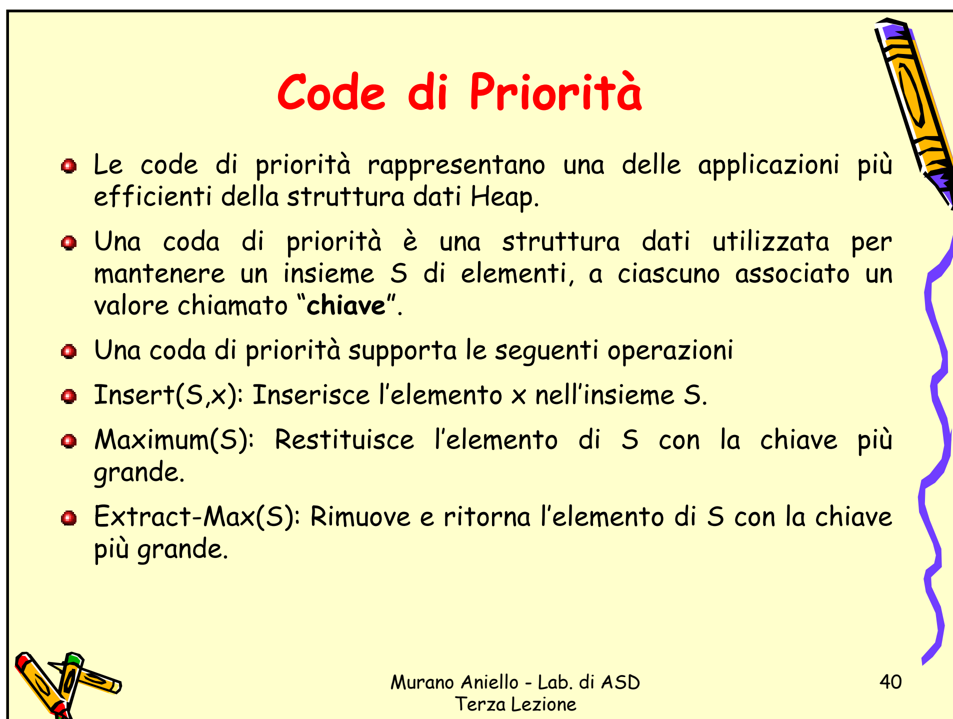
Main di HeapSort

```
main()
{
int A[MAX], k;
 printf("\nQuanti elementi deve contenere l'array: ");
 scanf("%d",&tot);
 while (tot>MAX)
    {printf("\n max 20 elementi: ");   scanf("%d",&tot);}
  for (k=0;k<tot;k++) {
      printf("\nInserire il %d° elemento: ",k+1);
      scanf("%d",&A[k]); }
  HeapSize=ArraySize=tot;
  HeapSort(A);
  printf("\nArray Ordinato:");
    for (k=0;k<tot;k++)
       printf(" %d",A[k]);
}
```

Complessità

- Il running time di Heapify è O(h) dove h è l'altezza dell'Heap. Siccome l'heap è un albero binario completo, il running time è O(logn). Più in dettaglio la sua complessità è la soluzione della ricorrenza $T(n) \leq T(2n/3) + \Theta(1)$ utilizzando il master method (caso 2).
- BuildHeap fa O(n) chiamate a Heapify. Per cui il running time di Buildheap è sicuramente O(nlogn). Si noti che le chiamate a Heapify avvengono su nodi ad altezza variabile minore di h. Da un'analisi dettagliata, risulta che il running time di Heapify è O(n).
- Heapsort fa O(n) chiamate a Heapify. Dunque il running time di Heapsort è O(nlogn)
- La complessità di spazio di Heapsort è invece O(n), visto che oltre il vettore di input necessita solamente di un numero costante di variabili per implementare l'algoritmo.

Un'applicazione degli Heap:
Code di Priorità
Murano Aniello - Lab. di ASD
Terza Lezione
39


Code di Priorità
Le code di priorità rappresentano una delle applicazioni più efficienti della struttura dati Heap.
Una coda di priorità è una struttura dati utilizzata per mantenere un insieme S di elementi, a ciascuno associato un valore chiamato "chiave".
Una coda di priorità supporta le seguenti operazioni
Insert(S,x): Inserisce l'elemento x nell'insieme S.
Maximum(S): Restituisce l'elemento di S con la chiave più grande.
Extract-Max(S): Rimuove e ritorna l'elemento di S con la chiave più grande.
Murano Aniello - Lab. di ASD
Terza Lezione
40

Una possibile applicazioni

- Una delle applicazioni più comuni delle code di priorità è quella della schedulazione dei lavori su computer condivisi (per esempio per gestire le code di stampa)
- La coda di priorità tiene traccia del lavoro da realizzare e la relativa priorità.
- Quando un lavoro viene eseguito o interrotto, il lavoro con più alta priorità è selezionato da quelli in attesa utilizzando la procedura Extract-Max.
- Ad ogni istante un nuovo lavoro può essere aggiunto alla coda.

Esercizio

- Scriver in C la funzione Extract-Max, che estrae il massimo da un heap.
- Scriver in C la funzione Insert, che permette di inserire un valore in un heap.
- Scrivere un programma che, dopo aver preso in input la grandezza di un heap e i valori contenuti nell'heap, dapprima costruisca l'heap, poi a scelta chiami una delle due funzioni precedenti e infine stampi l'heap risultante.
- N.B. La funzione Extract-Max deve essere eseguita soltanto se l'heap contiene almeno un elemento e Insert soltanto se l'heap non ha raggiunto la massima taglia del vettore che lo contiene.
- Analizzare le complessità del programma e delle funzioni.