Federica

Facoltà di Scienze Matematiche Fisiche Naturali

# Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Ordinamento, Ricorsione e Code di Priorità

| | | |
|---|---|---|
| Corso di Laurea | Informatica | Lezione numero: 3 |
| Codice insegnamento | 13917 | |
| Email docente | murano@na.infn.it | Parole chiave: Ordinamento, Insertion sort, Heapsort, |
| Anno accademico | 2007/2008 | |

next

e-Learning

Federica 16/10/2008 2

Facoltà di Scienze Matematiche Fisiche Naturali

## La Ricorsione

- Il concetto di ricorsione nasce dalla possibilità di eseguire un compito applicando lo stesso algoritmo ad un dominio ridotto rispetto a quello originale **fondendo** i risultati.
- Le funzioni C possono essere usate ricorsivamente, cioè una funzione può chiamare se stessa sia direttamente che indirettamente.
- Nella ricorsione è importante la condizione di uscita.
- Il problema deve poter essere suddiviso in sotto-problemi più piccoli fino ad arrivare ad un sotto-problema banale di cui si conosce immediatamente la soluzione.

back next

## Soluzione ricorsiva di un problema

PROBLEMA: Voglio lavare una pila di 15 piatti.

**Ho un sistema che riesce a lavare i piatti se ha in input una pila più piccola:**
**(intuitivamente, il calcolatore è in grado di lavare 14 piatti)**

**Algoritmo** *RICORSIVO:* ***un po' più semplice*** *del problema intero*

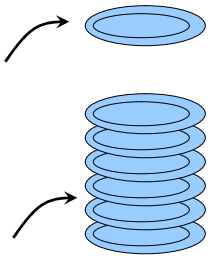
**prendo un piatto,**

**resta una pila di 14 piatti:**

**il calcolatore li lava**

**lavo il mio piatto**

***facilissimo***







## Esempio

```
void lava_piatti(n)
{
        if (nessun_piatto_da_lavare)        // caso base
            riposati!
        else            // passo ricorsivo
        {
            Prendi un piatto
            Lavalo
            lava_piatti(n-1)
        }
}
```

Condizione di terminazione

Chiamata Ricorsiva

il numero di piatti da lavare decresce:
la terminazione e' garantita!

Una funzione ricorsiva:

- Ha una o più condizioni di terminazione (casi base)
- Chiama se stessa ricor-sivamete.
- Ad ogni chiamata ci si avvicina alla condizione di terminazione

## Stampa dei primi N interi positivi

**Sia N=3.**
**Assumiamo che il calcolatore sappia fare la stampa di N-1 interi (I primi due interi).**
**Il mio risultato è dato da:**
**dopo aver stampato i primi N-1 interi (2 interi) stampa l'ennesimo intero (l'intero 3)**
**Attenzione: Gli elementi devono essere stampati in ordine crescente!**

**Considerazioni:**

- Assumiamo sempre che la chiamata ricorsiva svolge sempre il suo compito correttamente.
- Bisogna concentrarsi solo sull'ultimo passo di elaborazione per ottenere la soluzione corretta.

back next



```
void StampaN (int n)
{if (n==0)
   return;
 else
   { StampaN(n-1);
     Printf(" %d ",n);}
}
```

Stampa(3)

N=3;
Stampa (3-1);

Stampa(2)

N=2;
Stampa (2-1);

Stampa(1)

N=1;
Stampa (1-1);

Stampa(0)

back next

## Esempio di ricorsione

**Calcolo del fattoriale di un numero:**
**n! = n*(n-1)*(n-2)* .. .* (n - (n-1))**

```
int fact(int num)
{
  int product=1;
  for(; num>1; --num)
     product*=num;
  return product;
}
```

```
int fact(int num)
{
  if ( num <= 1)
     return 1;
  else
     return (num* fact(num -1));
}
```

back next



## Esempio di ricorsione

**Calcolo del numero di Fibonacci**

- f(n) = f(n-1)+f(n-2);
- f(0) = 0; f(1) =1;

**MAIN:**

```
#include <stdio.h>
int num;
main()
{
 printf("\n numero? ");
 scanf("%d",&num);
 printf("ris.= %d",fib(num));
}
```

**ITERATIVO:**

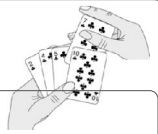
```
int fib(int num)
{
 int tmp=0, ris=1, prec=0;
 switch (num)
 {
    case 0: return 0;break;
    case 1: return 1;break;
    default:
    {
      for (; num>1;--num)
      {
          tmp=ris;
          ris+=prec;
          prec=tmp;
      }
      return ris;
    }
 }
}
```

**RICORSIVO:**

```
int fib(int num)
{
 switch (num)
 {
  case 0: return 0;break;
  case 1: return 1;break;
  default:
   return (fib(num-1) +
   fib(num-2));
 }
}
```

back next

# Algoritmi di ordinamento



## Insertion Sort

- L'Insertion Sort è uno algoritmo di ordinamento molto efficiente per ordinare un piccolo numero di elementi
- L'idea di ordinamento è quella che potrebbe usare un persona 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.

back next



# Funzione Insertion Sort

```
void insertion(int interi[20],int tot)
{
   int  temp;       /* Indice temporaneo per scambiare elementi  */
   int  prossimo;
   int  attuale;
   for (prossimo=1; prossimo<tot; prossimo++)
  {
          temp=interi[prossimo];
          attuale=prossimo-1;
          while ((attuale>=0) && (interi[attuale]>temp))
          {
             interi[attuale+1]=interi[attuale];
             attuale=attuale-1;
          }
          interi[attuale+1]=temp;
  }
}
```

| 2 | 4 | 5 | 3 | 6 | 9 | 8 | 1 |
|---|---|---|---|---|---|---|---|

back next

## Programma per Insertion Sort

```
# include <stdio.h>
# define MAX 20
int  i,j;                /* Indici di scorrimento dell'array */
void insertion(int interi[MAX],int tot);
main()
{
    int  interi[MAX]   /* Array che contiene i valori da ordinare */
    int  tot;              /* Numero totale di elementi nell'array */
    printf("\n Quanti elementi deve contenere l'array: ");
    scanf("%d",&tot);
    while (tot>20)
       { printf("\n max 20 elementi: ");       scanf("%d",&tot);  }
    for (i=0;i<tot;i++)
       { printf("\nInserire il %d° elemento: ",i+1);    scanf("%d",&interi[i]); }
    insertion(interi,tot);
    printf("\nArray Ordinato:");
    for (i=0; i<tot; i++)
       printf(" %d",interi[i]);
}
```

back next



## Ingegneria del software...

Autore: **Nome, Cognome, matricola ...**
Titolo: **nome della funzione (o modulo) implementata.**
Scopo: **obiettivi dell'algoritmo implementato(sintetico)**
Specifiche: **Nomi di funzioni, array, variabili importanti**
Descrizione: **Informazioni sull'algoritmo implementato.**
Lista dei Parametri: **Parametri input e output**
Complessità di Tempo e Di Spazio
Altri parametri eventualmente vuoti:

- **Indicatori di Errore:**
- **Routine Ausiliarie**
- **Indicazioni sull'utilizzo**

Implementazione

back next

## Documentazione per Insertion Sort

- **Scopo** : Ordinamento di un array di numeri interi
- **Specifiche:**
  - array di interi "interi[MAX]"
  - void insertion(int interi[MAX], int tot);
- **Descrizione** : L'insertion sort è un algoritmo molto efficiente per ordinare pochi numeri. Questo algoritmo è simile al modo che si potrebbe usare per ordinare un mazzo di carte…
- **Lista dei Parametri**
- *Input:*
  - interi[ ]: vettore contenente gli elementi da ordinare
  - tot numero degli elementi contenuti nel vettore
- *Output:* array interi[ ] ordinato in ordine crescente.





## Documentazione per Insertion Sort

### Complessità di Tempo

- L'algoritmo inserisce il componente **interi[i]** nel vettore già ordinato di componenti **interi[0]…interi[i-1]** spostando di una posizione tutti i componenti che seguono quello da inserire.
- Ad ogni passo la procedura compie nel caso peggiore (vettore ordinato al contrario), N-1 confronti, essendo N la lunghezza del vettore corrente e i-1 spostamenti.
- Le operazioni nel caso peggiore sono dunque (1+2+…+N-1), cioè, nel caso peggiore la complessità asintotica di tempo è $O(n^2)$. Nel caso migliore (vettore ordinato) bastano N-1 confronti.

## Documentazione per Insertion Sort

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).**

**Esempi di esecuzione:**

- Dati in ingresso i numeri: 20 11 45, si ottiene in uscita: 11 20 45

Risorse:

- Documentazione Insertion Sort
- QuickSort





## 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 0(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

back next



## Organizzazione dell'array

La radice dell'Heap è sempre memorizzata nel primo elemento dell'array.

Dato un nodo i, il suo nodo padre, figlio sx e dx possono essere calcolati 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;}

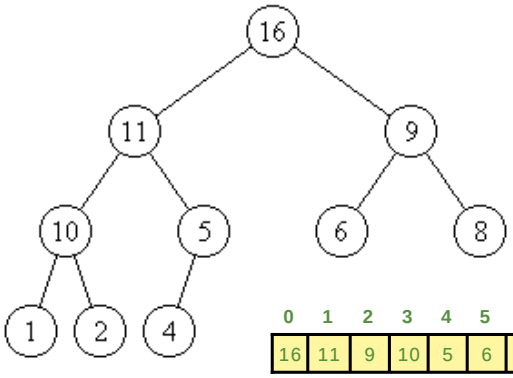
N.B. Nel C il primo elemento di un vettore A è A[0]. Nel libro di testo "Algoritmi e Strutture Dati", il primo elemento è invece A[1] e i valori precedenti sono dati dalle seguenti pseudo-funzioni:

- Parent (i) return$\lfloor i/2 \rfloor$
- Left (i) return 2i
- Right(i) return 2i+1

back next

## 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 |

back next



## 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.

back next

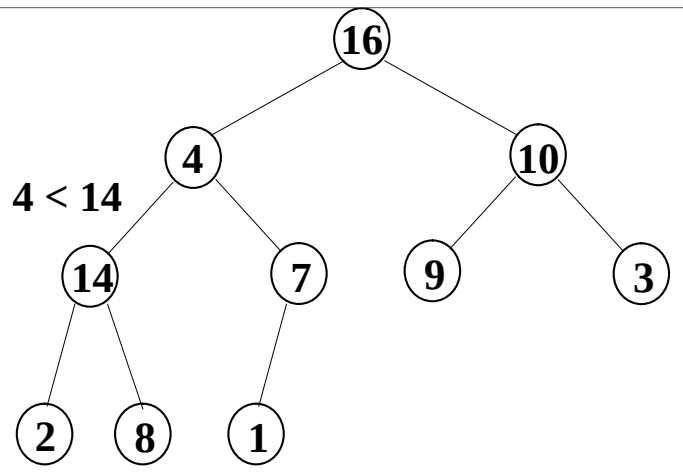
## Implementazione di Heapify

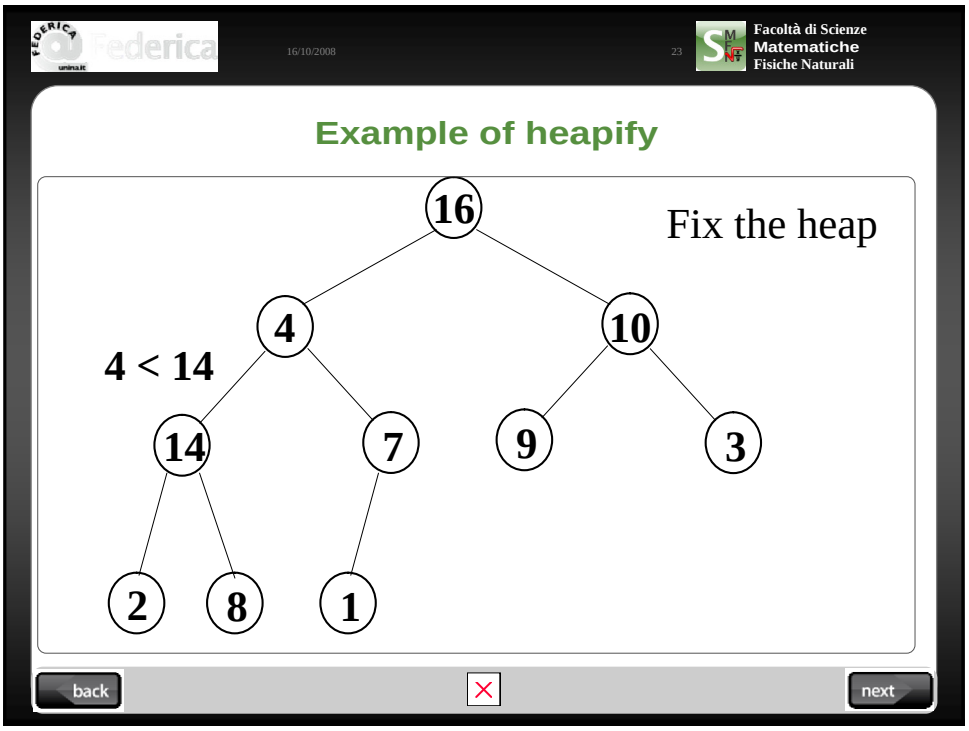
```
void Heapify(int A[MAX], int i)
{
        int l,r,largest;
        l = left(i);
        r = right(i);
        if (l < HeapSize && A[l] > A[i])
                largest = l;
        else largest = i;
        if (r < HeapSize && A[r] > A[largest])
                largest = r;

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

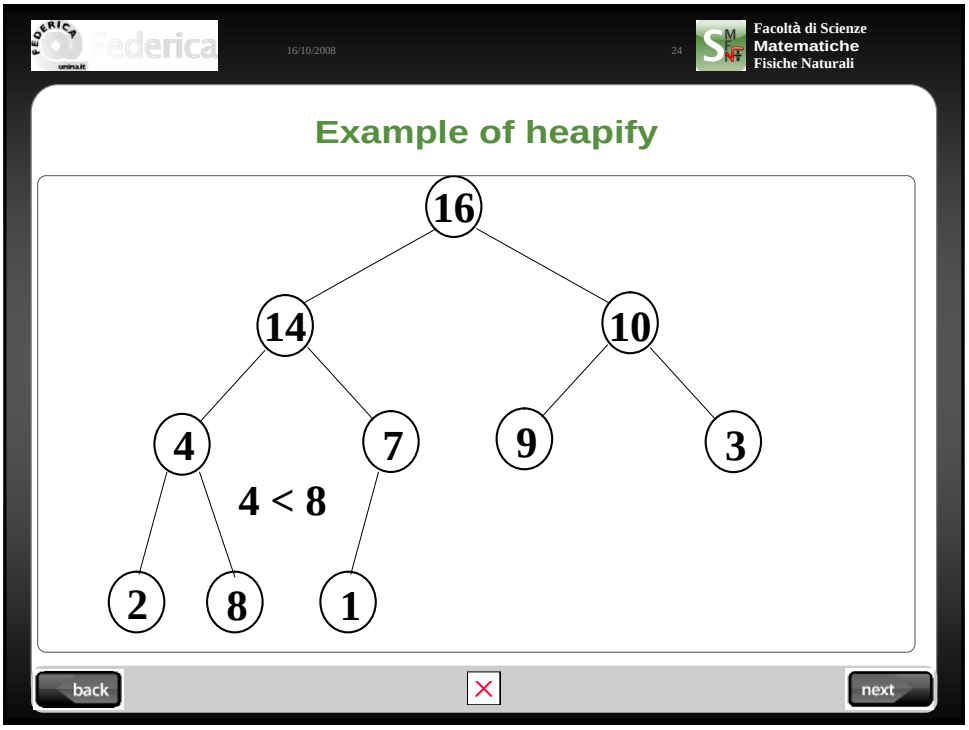




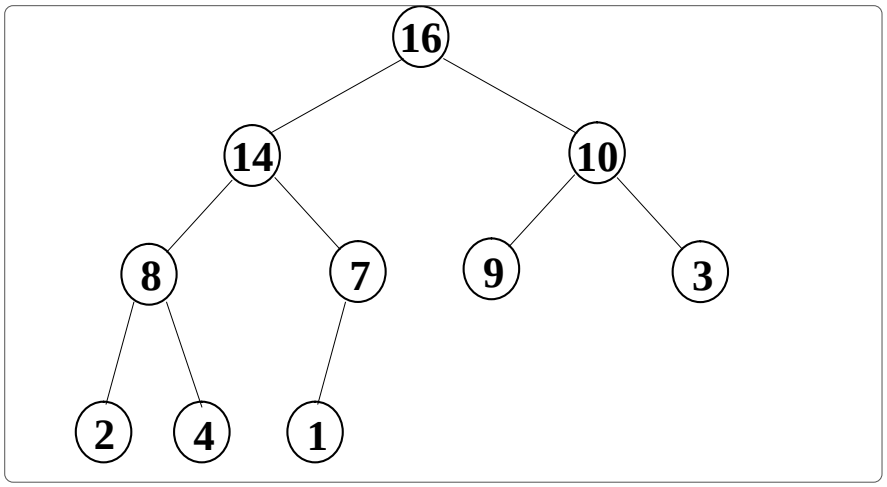
## Example of heapify




Federica
16/10/2008
23
Facoltà di Scienze
Matematiche
Fisiche Naturali
Example of heapify
Fix the heap
4 < 14
16
4
10
14
7
9
3
2
8
1
back
next


Federica
16/10/2008
24
Facoltà di Scienze
Matematiche
Fisiche Naturali
Example of heapify
16
14
10
4
7
9
3
4 < 8
2
8
1
back
next

## 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);
}
```

Risorse:

Esempio di Costruzione dell'heap

## 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);
        }
}
```

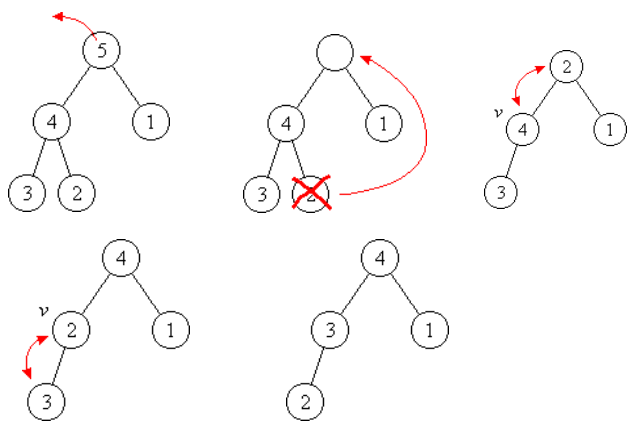
Risorse:

Esempio di HeapSort

back next



## Simulazione




back next

## 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]);
```

back next



## 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]);
}
```

back next

## 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.

back next



## 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.**

back next

## Una possibile applicazione

- 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.