

# Algoritmi e Strutture Dati

## Strutture Elementari

Maria Rita Di Berardini, Emanuela Merelli<sup>1</sup>

<sup>1</sup>Dipartimento di Matematica e Informatica  
Università di Camerino

A.A. 2006/07

# Insertion Sort

```
for  $j \leftarrow 2$  to  $\text{lenght}[A]$   
  do  $\text{key} \leftarrow A[j]$ 
```

▷ Si inserisce  $A[j]$  nella sequenza ordinata  $A[1..j-1]$

```
   $i \leftarrow j-1$ 
```

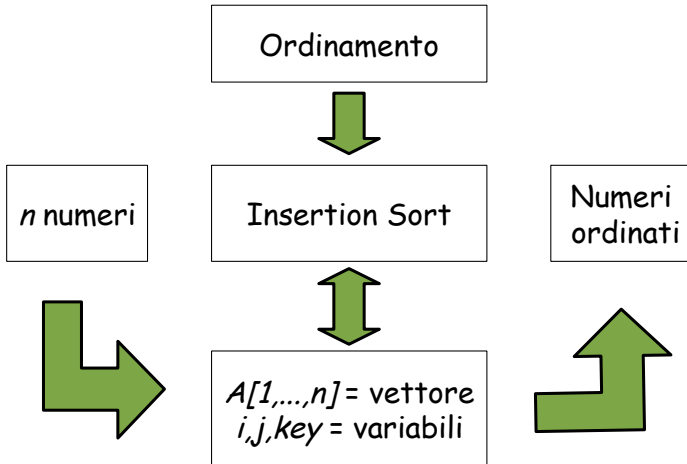
```
  while  $i > 0$  e  $A[i] > \text{key}$ 
```

```
    do  $A[i+1] \leftarrow A[i]$ 
```

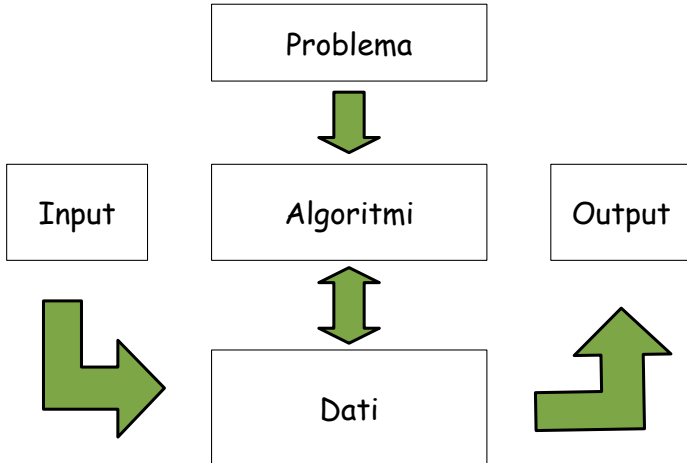
```
       $i \leftarrow i - 1$ 
```

```
   $A[i+1] \leftarrow \text{key}$ 
```

# Schema dell'Insertion-Sort



# Schema di un generico algoritmo

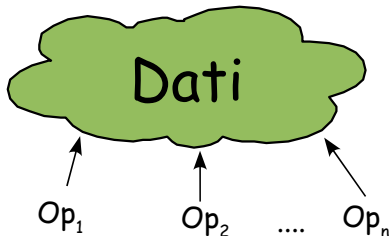


# Strutture Dati Astratte (SDA)

Dati: insieme di valori – oggetti

+

Operazioni che consentono di manipolare i dati



## Esempio di SDA

**Dati** = insieme  $S$  di numeri

$OP_1$  = estrai il minimo

$OP_2$  = estrai il massimo

$OP_3$  = restituisci la dimensione di  $S$

$OP_4$  = inserisci un nuovo numero in  $S$

# Insertion Sort

**for**  $j \leftarrow 2$  **to**  $\text{size}(A)$

▷ Si inserisce  $A[j]$  nella sequenza ordinata  $A[1..j-1]$

$i \leftarrow j-1$

**while**  $i > 0$  e  $\text{read}(j) > \text{key}$

**do**  $\text{modify}(i+1, \text{read}(i))$

$i \leftarrow i-1$

$\text{modify}(i+1, \text{key})$

# Insertion Sort

$$ADS = \left\{ \begin{array}{l} \text{Insieme } S \text{ di numeri} \\ + \\ \text{Read, Size, Modify} \end{array} \right.$$

$$DS = \left\{ \begin{array}{l} S = "A[1, \dots, n]" \text{ vettore} \\ \text{Read}(i) = "A[i]" \\ \text{Size}(A) = "n" \\ \text{Modify}(i, x) = "A[i] = x" \end{array} \right.$$

# ADS vs DS

ADS = che cosa vogliamo ?

DS = come lo implementiamo ?

# Quando una struttura dati è buona??

Quando non usa troppe risorse

$$Risorse = \left\{ \begin{array}{l} \text{Tempo} \\ \text{Spazio di memoria} \\ \text{Numero di processori} \\ \dots \end{array} \right.$$

# Il concetto di Dato

Un **dato** è tutto ciò su cui agisce un calcolatore

A livello hardware tutti i dati sono rappresentati  
come sequenza di cifre binarie

Linguaggi ad alto livello ci permettono di usare  
astrazioni tramite il concetto di tipo di dato

# Tipo di dato

Nei linguaggi di programmazione

- il **tipo di dato** determina l'insieme dei valori (oggetti) che una variabile può rappresentare
- il **tipo di dato**, specifica le operazioni di interesse su un insieme di valori - o collezione di oggetti (es. inserisci, cancella, cerca, etc.)

Ogni operatore accetta argomenti di uno o più tipi di dato fissato e produce risultati di un tipo di dato fissato

# Gestione di collezioni di oggetti

**Struttura dati:** un organizzazione dei dati che permette di supportare le operazioni di un tipo di dato in modo efficiente

# Tipo di dato Dizionario

**Dati:** un insieme di coppie (*elem*, *chiave*)

**Operazioni:**

`insert(elem e, chiave k)`

aggiunge ad *S* una nuova coppia (*e*, *k*)

`delete(elem e, chiave k)`

cancella da *S* la coppia con chiave *k*

`search(chiave k)`

se la chiave *k* è presente in *s* restituisce l'elemento  
e ad esso associato altrimenti restituisce `null`

# Pile e Code

## Pila:

- insieme di elementi gestiti secondo una politica LIFO (**L**ast **I**n **F**irst **O**ut)
- l'ultimo elemento inserito è il primo estratto

## Coda:

- insieme di elementi gestiti secondo una politica FIFO (**F**irst **I**n **F**irst **O**ut)
- il primo elemento inserito è il primo estratto

# Il tipo di dato Pila

**Dati:** una sequenza  $S$  di  $n$  elementi

**Operazioni:**

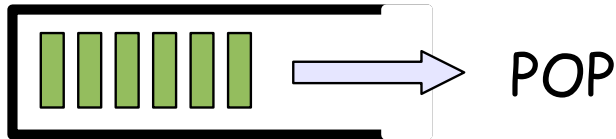
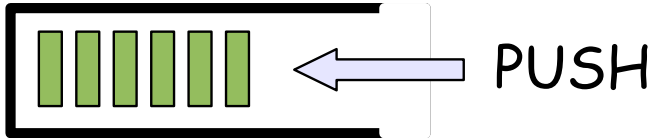
$\text{isEmpty}() \rightarrow \text{boolean}$   
restituisce *true* se  $S$  è vuota, *false* altrimenti

$\text{push}(\textit{elem } e)$   
aggiunge  $e$  come ultimo elemento di  $S$

$\text{pop}() \rightarrow \textit{elem}$   
toglie da  $S$  l'ultimo elemento e lo restituisce

$\text{top}() \rightarrow \textit{elem}$   
restituisce l'ultimo elemento di  $S$  (senza eliminarlo)

# Il tipo di dato Pila



# Tipo di dato Coda

**Dati:** una sequenza  $S$  di  $n$  elementi

**Operazioni:**

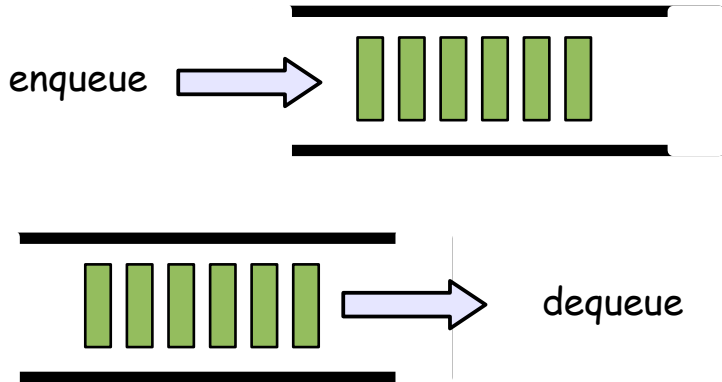
`isEmpty()`  $\rightarrow$  boolean  
restituisce true se  $S$  è vuota, false altrimenti

`enqueue(elem e)`  
aggiunge  $e$  come ultimo elemento di  $S$

`dequeue()`  $\rightarrow$  *elem*  
toglie da  $S$  l'ultimo elemento e lo restituisce

`first()`  $\rightarrow$  *elem*  
restituisce il primo elemento di  $S$  (senza eliminarlo)

# Il tipo di dato Coda



# Tecniche di rappresentazione dei dati

Rappresentazioni indicizzate:

- I dati sono contenuti in array

Rappresentazioni collegate:

- I dati sono contenuti in record collegati fra loro mediante puntatori

## Pro e contro

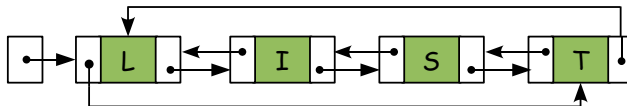
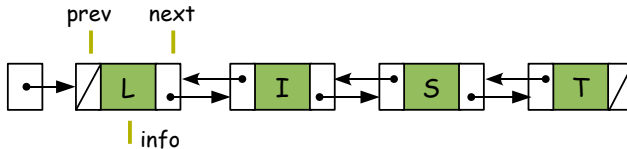
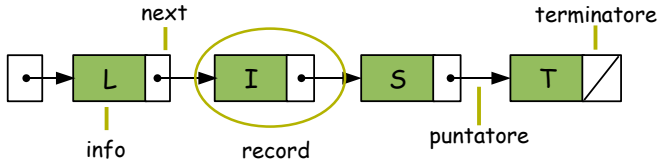
Rappresentazioni indicizzate:

- **Pro:** accesso diretto ai dati mediante indici
- **Contro:** dimensione fissa (riallocazione array richiede tempo lineare)

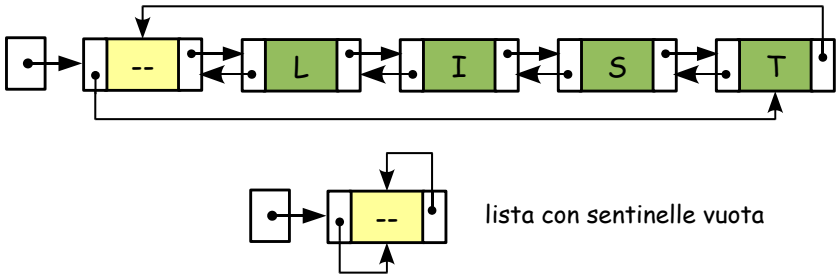
Rappresentazioni collegate:

- **Pro:** dimensione variabile (aggiunta e rimozione record in tempo costante)
- **Contro:** accesso sequenziale ai dati

# Esempi di strutture collegate: le liste



# Liste con “sentinelle”



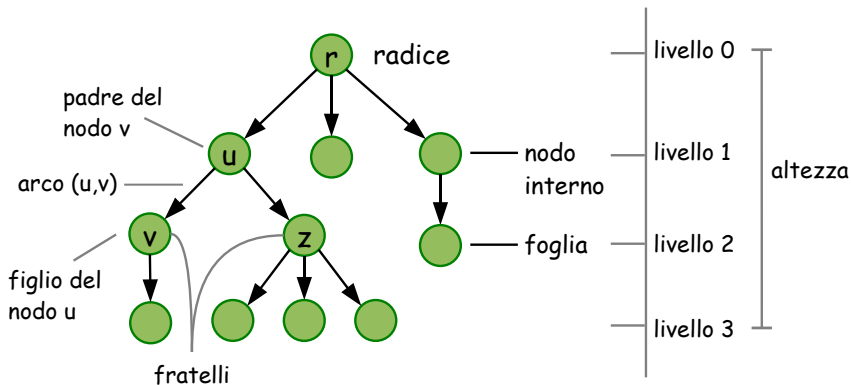
# Organizzazione gerarchica dei dati: alberi

Un albero radicato è una coppia  $T = (N, A)$  costituita da un insieme  $N$  di **nodi** ed un insieme  $A \subseteq N \times N$  di coppie di nodi, detti **archi**. In un albero:

- ogni nodo  $v$  (tranne la radice) ha un solo **padre**  $u$  tale che  $(u, v) \in A$ . Nodi con lo stesso padre sono detti **fratelli**
- un nodo  $u$  può avere zero o più **figli**  $v$  tali che  $(u, v) \in A$  e, il loro numero è detto grado del nodo
- un nodo senza figli è detto **foglia**, mentre i nodi che non sono nè foglie nè la radice sono detti **nodi interni**
- la profondità (o livello) di un nodo è dato dal numero di archi che bisogna attraversare per raggiungerlo dalla radice
- **altezza** di un albero: massima profondità a cui si trova una foglia

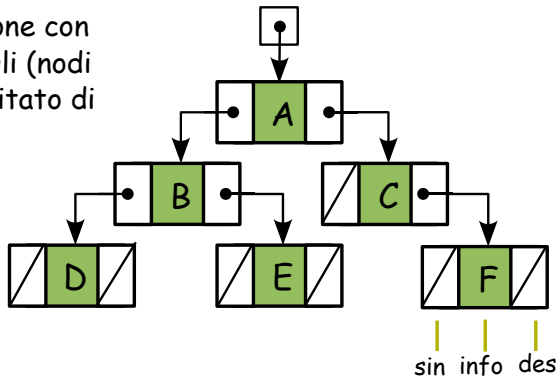
# Organizzazione gerarchica dei dati: alberi

**Dati** contenuti nei nodi, **relazioni gerarchiche** definite dagli archi che collegano coppie di nodi

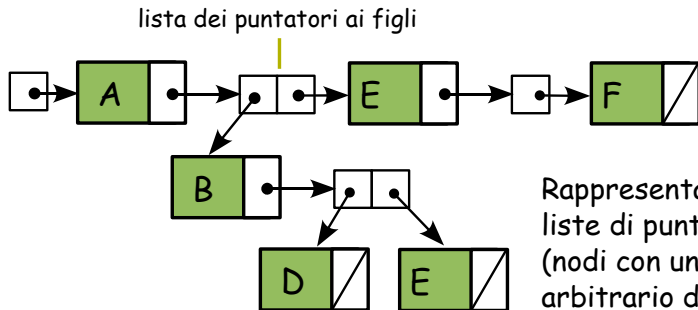


# Rappresentazioni collegate di alberi

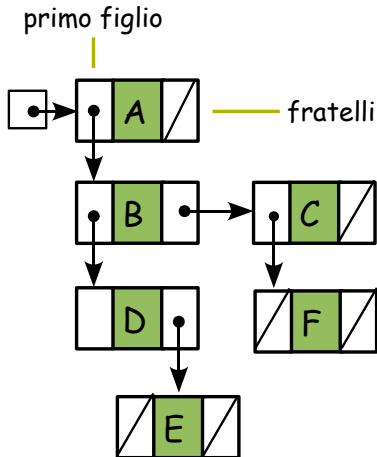
Rappresentazione con  
puntatori ai figli (nodi  
con numero limitato di  
figli)



# Rappresentazioni collegate di alberi



# Rappresentazioni collegate di alberi



Rappresentazione  
con primo figlio -  
fratello successivo  
(numero arbitrario di  
figli)

# Visite di Alberi

Algoritmi che consentono l'accesso sistematico ai nodi e agli archi di un albero

Gli algoritmi di visita si distinguono in base al particolare ordine di accesso ai vari nodi dell'albero

# Algoritmo di visita generica

L'algoritmo **visitaGenerica**(*nodo*  $r$ ) visita il nodo  $r$  e tutti i suoi discendenti in un albero

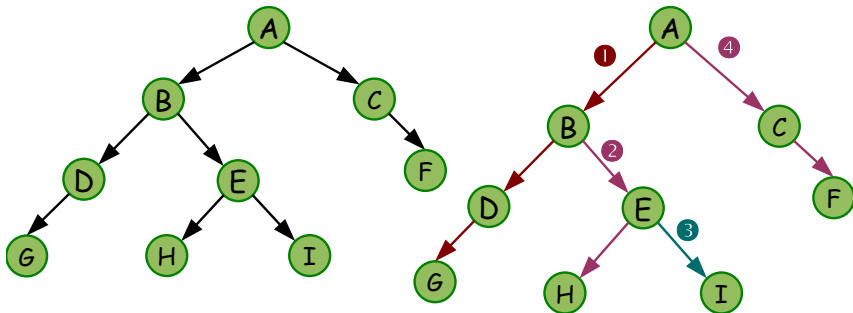
```
visitaGenerica(nodo  $r$ )  
     $S \leftarrow \{r\}$   
    while ( $S \neq \emptyset$ ) do  
        estrai un nodo  $u$  da  $S$   
        visita il nodo  $u$   
         $S \leftarrow S \cup \{\text{figli di } u\}$ 
```

# Algoritmo di visita in profondità

L'algoritmo di **visita in profondità** (DFS, Depth-First-Search) parte da  $r$  e procede visitando i nodi di figlio in figlio fino a raggiungere una foglia

Retrocede poi al primo antenato che ha figli non ancora visitati (se esiste) e ripete il procedimento a partire da uno di quei figli

# Algoritmo di visita in profondità



$\text{DFS}(A) = A, B, D, G, E, H, I, C, F$

# Algoritmo di visita in profondità

## Versione iterativa (per alberi binari)

**visitaDFS**(*nodo r*)

Pila *S*

*S.push(r)*

**while** (not *S.isEmpty()*) **do**

$u \leftarrow S.pop()$

**if** ( $u \neq null$ ) **then**

        visita il nodo *u*

*S.push*(figlio destro di *u*)

*S.push*(figlio sinistro di *u*)

# Algoritmo di visita in profondità

Versione ricorsiva (per alberi binari)

```
visitaDFSricorsiva(nodo r)  
  if (r = null) return  
  visita il nodo r  
  visitaDFSricorsiva(figlio sinistro di r)  
  visitaDFSricorsiva(figlio destro di r)
```

Questa versione dell'algoritmo di visita in profondità viene anche detta **visita in preordine**

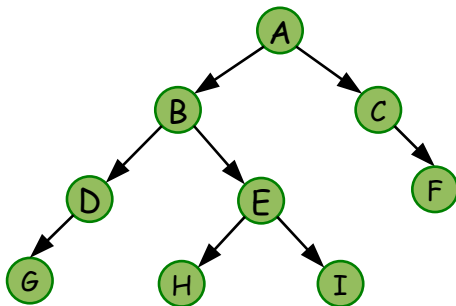
## Tre varianti classiche dell'algoritmo di visita in profondità

**Visita in preordine:** si visita prima la radice e poi si effettuano le chiamate ricorsive sul figlio sinistro e destro

**Visita simmetrica:** si effettua prima la chiamata ricorsiva sul figlio sinistro, poi si visita la radice ed, infine, si effettua la chiamata ricorsiva sul figlio destro

**Visita in postordine:** si effettuano prima le chiamate ricorsive sul figlio sinistro e destro e poi si visita la radice

# Tre varianti classiche dell'algoritmo di visita in profondità



visita in preordine: A, B, D, G, E, H, I, C, F

visita simmetrica: G, D, B, H, E, I, A, C, F

visita in postordine: G, D, H, I, E, B, F, C, A

# Visita simmetrica

## Versione ricorsiva (per alberi binari)

**visitaSimmetrica**(*nodo r*)

- ▷ Rispetto a **visitaDFSricorsiva**, basta invertire la chiamata
- ▷ **ricorsiva sul sottoalbero sinistro e la visita della radice**

**if** ( $r = \text{null}$ ) **return**

**visitaDFSricorsiva**(figlio sinistro di  $r$ )

visita il nodo  $r$

**visitaDFSricorsiva**(figlio destro di  $r$ )

# Visita in postordine

## Versione ricorsiva (per alberi binari)

**visitaInPostOrdine**(*nodo r*)

- ▷ Rispetto a **visitaDFSricorsiva**, basta effettuare la visita
- ▷ della radice solo dopo le chiamate ricorsive sui figli

**if** ( $r = \text{null}$ ) **return**

**visitaDFSricorsiva**(figlio sinistro di  $r$ )

**visitaDFSricorsiva**(figlio destro di  $r$ )

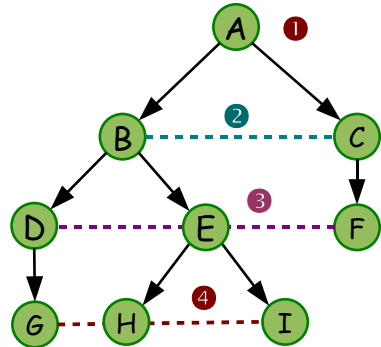
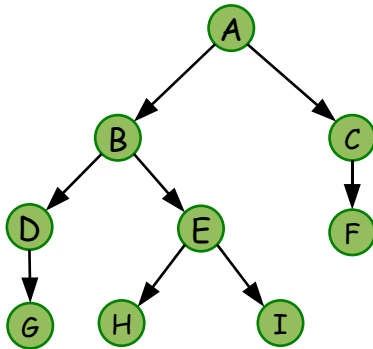
visita il nodo  $r$

## Algoritmo di visita in ampiezza

L'algoritmo di **visita in ampiezza** (BFS, Breadth-First-Search) parte da  $r$  e procede visitando i nodi per livelli successivi

Un nodo sul livello  $i$  può essere visitato solo se tutti i nodi sul livello  $i - 1$  sono stati visitati

# Algoritmo di visita in profondità



$BFS(A) = A, B, C, D, E, F, G, H, I$

# Algoritmo di visita in ampiezza

Versione iterativa (per alberi binari)

```
visitaDFS(nodo r)  
  Coda C  
  C.enqueue(r)  
  while (not C.isEmpty()) do  
    u  $\leftarrow$  C.dequeue()  
    if (u  $\neq$  null) then  
      visita il nodo u  
      C.enqueue(figlio sinistro di u)  
      C.enqueue(figlio destro di u)
```

# Riepilogo

Nozione di **tipo di dato** come specifica delle operazioni su una collezione di oggetti

Rappresentazioni indicizzate e collegate di collezioni di dati: pro e contro

Organizzazione gerarchica dei dati mediante alberi

Rappresentazioni collegate classiche di alberi

Algoritmi di esplorazione sistematica dei nodi di un albero (algoritmi di visita)