

Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Stack e Code

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 5

Parole chiave: **LIFO, FIFO**

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Realizzata con il coinvolgimento del Istituto Universitario Lattesco - Missa 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006

Stack(pile) e Code

- Stack e code sono insiemi dinamici in cui l'elemento rimosso dall'insieme con l'operazione di cancellazione è sempre predeterminato.
- In uno stack, l'elemento cancellato è quello più recentemente inserito. Gli stack rispettano la politica LIFO (**last-in, first-out**).
- In una coda, l'elemento cancellato è sempre quello che è rimasto più a lungo nell'insieme. Le code rispettano la politica FIFO (**first-in, first out**)
- In uno stack un nuovo elemento è sempre posto in testa agli altri, mentre nella coda esso è posto dopo tutti gli altri.
- Ci sono molti modi per implementare stack e code su un computer. Cominciamo con una implementazione di stack utilizzando array.



Stack

Le operazioni fondamentali su Stack sono:

- **Push(S,x)** che serve a inserire l'elemento **x** al top dello stack
- **Pop(S)** che serve a cancellare il top dello stack **S**

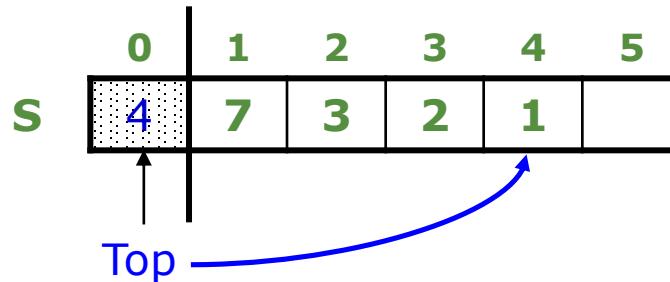
Un esempio di stack può essere dato da una pila di piatti posta su un tavolo.

- Si noti che l'ordine in cui i piatti sono tolti dallo stack è inversa all'ordine in cui essi sono stati inseriti nello stack, visto che solo il top dello stack è accessibile.
- Un'altra operazione importante sugli stack è **Empty-Stack**, necessaria per controllare se uno stack è vuoto



Implementazione di Stack con Array

- Per implementare uno stack di n elementi, si può utilizzare un array S di dimensione n .
- L'array S utilizzerà l'attributo TOP che indicherà l'indice dell'elemento più recente immesso nello stack. In una implementazione si potrebbe pensare di memorizzare questo indice nel primo elemento dell'array che dunque avrà come dimensione non più n ma **$n+1$** .
- Esempio:



- Un'operazione di Pop sullo stack dell'esempio precedente, restituirà l'elemento 1 e il TOP diventerà 3.
- Quando $\text{TOP}=0$, lo stack è vuoto, cioè non contiene elementi.

Implementazione di Push e Pop

- Utilizzando un array **S[MAX+1]** per l'implementazione di uno stack, nel modo descritto precedentemente, le operazioni di **Push** e **Pop** possono essere realizzate semplicemente nel modo seguente:

Push

```
void push(int S[], int valore)
{
    S[0] = S[0]+ 1;
    S[S[0]] = valore;
}
```

Pop

```
int pop(int S[])
{
    S[0] = S[0]-1;
    return S[S[0]+1];
}
```



Implementazione di Empty_stack e Full_stack

- Utilizzando un array **S[MAX+1]** per l'implementazione di uno stack di **MAX** elementi, empty_stack può essere realizzata controllando se Top=0. Inoltre, si può utilizzare una procedura full_stack per controllare se lo stack è pieno, cioè se Top=MAX.

empty_stack

```
int empty_stack(int S[])
{
    return S[0]==0;
}
```

full_Stack

```
int full_stack(int S[])
{
    return S[0]==MAX;
}
```



Implementazione efficiente di Push e Pop

- Facoltativo: Utilizzando empty_stack e full_stack è possibile realizzare una versione più efficiente di Pop e Push con controllo di errore, nel modo seguente:

Push_check

```
void push_c(int S[], int val, int *err)
{
    if (full_stack(S))
        *err=1;
    else
    {
        S[0] = S[0]+ 1;
        S[S[0]]= valore;
        *err=0;
    }
}
```

Pop_check

```
int Pop_c(int S[],int *err)
{
    int val=0;
    if (empty_stack(S))
        *err=1;
    else
    {
        S[0] = S[0]-1;
        val=S[S[0]+1];
        *err=0;
    }
    return val;
}
```



Costruzione di uno Stack

- La seguente procedura permette di costruire uno stack **S** di taglia **num_elementi**, sapendo che **S** può contenere al più **MAX** valori

```
void new_stack(int S[])
{
    int num_elementi, valore;
    printf("\n Quanti elementi (max %d
elementi): ", MAX);
    scanf("%d",&num_elementi);
    while (num_elementi >MAX) {
        printf("\n max %d elementi: ", MAX);
        scanf("%d",&num_elementi);
    }
    while(num_elementi) {
        printf("\n Inserire un valore: ");
        scanf("%d",&valore);
        push(S, valore);
        --num_elementi;
    }
}
```

- Cosa succede se si inserisce **0** per num_elementi?



Stampa di uno Stack

- La stampa di uno stack **S** può essere fatta nel modo seguente:

Stampa di uno Stack

```
void stampa_stack (int S[])
{
    int valore;
    if (!empty_stack)
    {
        valore=pop(S);
        printf(" %d ",valore);
        stampa_stack(S);
        push(S,valore);
    }
}
```



Gestione di uno Stack (1)

- Il seguente programma gestisce uno stack **S** di **MAX** valori. Si noti come i controlli siano indipendenti da MAX. Questo è utile quando MAX e S[MAX+1] sono dati esternamente al programma (come solitamente avviene)

```
#define MAX 20
main()
{
    int S[MAX+1],scelta,valore;
    do
    {
        printf("\n scelta: 0-Crea, 1-Stampa, 2-Pop, 3-Push, 4-uscita : ");
        scanf("%d ",&scelta);
        switch (scelta)
            { ..... /* vedi prossima slide */
        }
    while(scelta==0||scelta==1 ||scelta==2||scelta==3);
} /* fine main() */
```



Gestione di uno Stack (2): Implement. Switch

```
switch (scelta)
{
    case 0:
        new_stack(S);      break;
    case 1:
        stampa(S);       break;
    case 2:
        if (!empty_stack(S))
            printf("\n Top dello Stack %d", pop(S));
        else
            printf("\n spiacente, stack vuoto");
        break;
    case 3:
        if (!full_stack(S)) {
            printf("\n valore da inserire nello stack: ");
            scanf("%d",&valore);
            push(S,valore);
        }
        else
            printf("\n spiacente, stack pieno");
} /* fine switch */
```



Esercizio su Stack

- Si consideri uno Stack **S**, implementato con array **S[MAX+1]**.
- Si implementi la funzione **ricorsiva**

void togli_da_Stack(int S[], int el)

che elimini dallo stack S tutti gli elementi uguali ad **el** lasciando invariato l'ordine degli altri elementi.

- Non utilizzare altre strutture dati di appoggio.
- Non utilizzate accessi diretti all'array, ma servitevi solo delle funzioni implementate per la gestione degli stack.
- Si ricordi che lo stack è una struttura dati che permette l'accesso ai suoi dati solo dal top.



Code

A differenza dello stack, una coda usa due attributi:

- Inizio coda (Head)
- Fine coda (Tail)

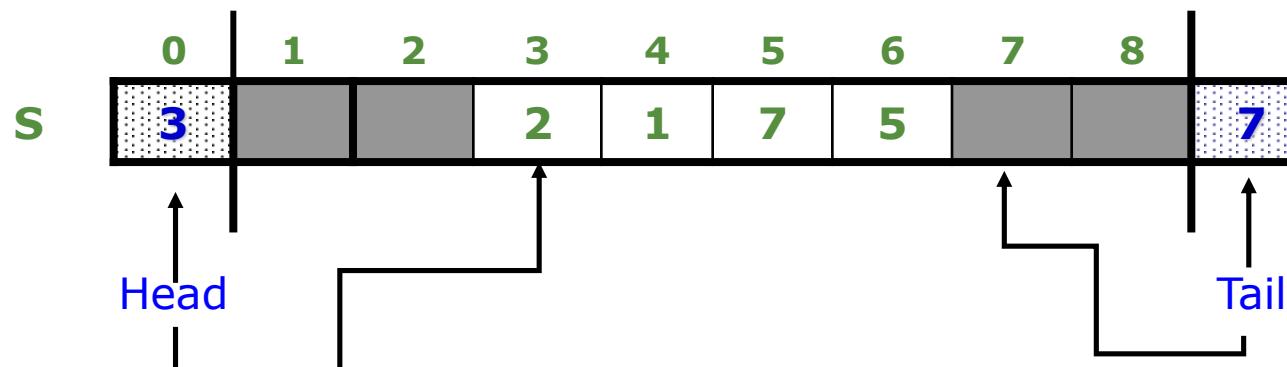
In pratica, usando come esempio di coda una fila ad uno sportello. L'inizio della coda è rappresentato dalla prima persona della fila (quella la prossima ad essere servita), mentre la fine della coda è rappresentata dall'ultima persona che si è aggiunta alla coda (cioè, l'ultima persona tra quelle attualmente in fila ad essere servita)

Un inserimento nella coda sarà fatto sempre alla sua fine mentre una cancellazione alla sua testa



Implementazione di Code

- Come per gli stack, anche per le code possiamo avere diversi modi di rappresentazione su un computer.
- Iniziamo con l'utilizzo di array.
- Per memorizzare una coda con massimo **n** elementi, possiamo utilizzare uno stack di dimensione **n** più due variabili che memorizzano costantemente l'indice della testa e della coda.
- Come per gli stack, si può anche scegliere di memorizzare l'indice della testa e della coda nel vettore stesso.
- Per esempio:



Implementazione di Empty_queue e Full_queue

- Utilizzando un array **Q[MAX+2]** per l'implementazione di una coda di **MAX** elementi, **empty_queue** può essere realizzata controllando se Head=0 (in questo caso Tail sarà uguale a 1).
- Inoltre, si può utilizzare una procedura **full_queue** per controllare se la coda è piena, cioè se Head=Tail.

empty_queue

```
int empty_stack(int Q[])
{
    return Q[0]==0;
}
```

full_queue

```
int full_queue(int Q[])
{
    return Q[0]==Q[MAX+1];
}
```

- Dunque, per creare una coda vuota basterà avere Head= 0 e Tail=1



Implementazione di Dequeue e Enqueue

Usando l'array Q precedente per l'implementazione di una coda, le operazioni di Cancellazione (Dequeue) e Inserimento (Enqueue) possono essere realizzate come segue:

Enqueue

```
void enqueue(int Q[], int valore)
{
    Q[Q[MAX+1]] = valore;
    if (Q[0] == 0)
        Q[0]=1;
    Q[MAX+1] = (Q[MAX+1] % MAX) + 1;
}
```

Dequeue

```
int dequeue(int Q[])
{
    int valore=Q[Q[0]];
    Q[0] = (Q[0] % MAX) + 1;
    if (Q[0] == Q[MAX+1]) {
        Q[0]=0;
        Q[MAX+1]=1;    }
    return valore;
}
```



Stampa di una Coda

- Per la stampa di una coda, non possiamo usare esattamente la stessa procedura vista per gli stack. Infatti questa produrrebbe una inversione della coda. Per risolvere questo problema, provvediamo a invertire ulteriormente la coda. Dunque la funzione stampa prima chiama **stampa_queue** e poi **rewerse**

stampa_queue

```
void stampa_queue(int Q[])
{
    int val;
    if (!empty_queue(Q))
    {
        val=dequeue(Q);
        printf(" %d |\",val);
        stampa_queue(Q);
        enqueue(Q,val);
    }
}
```

rewerse

```
void rewerse (int Q[])
{
    int val;
    if (!empty_queue(Q))
    {
        val=dequeue(Q);
        rewerse(Q);
        enqueue(Q,val);
    }
}
```



Costruzione di una Coda

- La seguente procedura permette di costruire una coda **Q** di taglia **num_elementi**, sapendo che **Q** può contenere al più **MAX** valori

```
void new_queue(int Q[])
{
    int num_elementi, valore;
    printf("\n Quanti elementi (max %d elementi): ", MAX);
    scanf("%d",&num_elementi);
    while (num_elementi >MAX) {
        printf("\n max %d elementi: ", MAX);
        scanf("%d",&num_elementi);
    }
    while(num_elementi) {
        printf("\n Inserire un valore: ");
        scanf("%d",&valore);
        enqueue(Q, valore);
        num_elementi--;
    }
}
```



Gestione di una Coda (1)

- Il seguente programma gestisce una coda **Q** di **MAX** valori. Si noti come i controlli siano indipendenti da MAX. Questo è utile per le stesse motivazioni date per gli stack

```
#define MAX 20
main()
{
    int Q[MAX+1],scelta,valore;
    do
    {
        printf("\n scelta: 0-Crea, 1-Stampa, 2-Deq, 3-Enq,, 4-uscita : ");
        scanf("%d ",&scelta);
        switch (scelta)
            { ..... } /* vedi prossima diapositiva */
    }
    while(scelta==0||scelta==1 ||scelta==2||scelta==3);
} /* fine main()
```



Gestione di una Coda(2): Implement. switch

```
switch (scelta)
{
    case 0:
        new_queue(Q);           break;
    case 1:
        stampa_queue(Q);      break;
    case 2:
        if (!empty_queue(Q))
            printf("\n Head della coda %d", dequeue(S));
        else
            printf("\n spiacente, coda vuoto");
        break;
    case 3:
        if (!full_queue(Q)))
            {
                printf("\n Valore da inserire nello stack: ");
                scanf("%d",&valore);
                enqueue(Q,valore);
            }
        else
            printf("\n spiacente, coda piena");
} /* fine switch */
```



Esercizio su Code

- Si consideri una coda **Q**, implementata con array **Q[MAX+2]**.
- Si implementi la funzione **ricorsiva**

void togli_dispari(int Q[])

che elimini dalla coda tutti i numeri dispari, lasciando invariato l'ordine degli elementi.

- Non utilizzare altre strutture dati di appoggio.
- Non utilizzate accessi diretti all'array, ma servitevi solo delle funzioni implementate per la gestione delle code.
- Si ricordi che la coda è una struttura dati che permette il reperimento dei dati dalla testa e l'inserimento in coda.



Esercizio

Si consideri un grafo G con V vertici ed E archi. Si implementino in linguaggio C le seguenti operazioni utilizzando sia una rappresentazione con liste di adiacenza che con matrice di adiacenza:

- Creazione di una struttura dati grafo pesata contenente tutti i vertici del grafo G .
- Modifica di un peso.
- Aggiunta di un arco.
- Cancellazione di un arco
- Calcolo del grado uscente e entrante di un nodo.
- Calcolo dell'arco con peso maggiore

Scrivere in linguaggio C un programma che implementi le operazioni precedenti indipendentemente dal fatto che la struttura dati di appoggio sia un grafo rappresentato con liste di adiacenza o con matrice di adiacenza.

Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Introduzione al Corso - Il Linguaggio C (I parte)

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 1

Parole chiave: **Introduzione, Linguaggio C**

next



Percorsi di Formazione a Distanza
e-Learning

Università degli Studi di Napoli Federico II



Centro di Ricerca e Sviluppo
CNR-IPCF

Realizzata con il coinvolgimento del Istituto Universitario Lattesano. Missa 3.12. Mhassane anima I – P.D.R. Campagna 2007-2008



European Project



Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II

Introduzione al Corso

Informazioni Generali sul Corso:

- Esame: Laboratorio di Algoritmi e strutture dati (6 CFU)
- Libri di testo:
 - [Brian W. Kernighan](#) e [Dennis M. Ritchie](#), "Linguaggio C ", Jackson libri
- Approfondimenti:
 - Al Kelley e Ira Pohl, "C Didattica e Programmazione", Addison Wesley Italia.
 - Dantona e Damiani "Il linguaggio C++ ", Addison Wesley Italia.
- Modalità d'esame: Una prova di laboratorio a gruppi (di tre persone) e una prova scritta.



Introduzione al Corso

Informazioni sul Docente:

- Prof. Dr. Aniello Murano, ricercatore universitario presso la Sezione di Informatica del Dipartimento di Fisica - Università degli Studi di Napoli "Federico II"
- Sito web: <http://people.na.infn.it/~murano/>
- Ricevimento: Studio OF29b – Edificio del Dip. di Biologia, nei giorni e orari indicati sul sito web del docente
- E-mail: murano@na.infn.it



Introduzione al Corso

Obiettivi del Corso:

- Familiarizzare lo studente con la progettazione di algoritmi e strutture dati.
- Particolare enfasi verrà posta sullo stile di programmazione utile per produrre codice chiaro, modulare, efficiente e facilmente modificabile.
- Dopo una breve introduzione al linguaggio di programmazione C, si procederà all'implementazione di alcune strutture dati fondamentali quali alberi, heap, code con priorità, insiemi disgiunti e grafi.
- Le lezioni sono basate su lezioni frontali e esercitazioni pratiche in laboratorio.

Introduzione al Corso

Finalità del Corso:

- Al termine del corso gli studenti dovranno essere in grado di realizzare un progetto completo, comprensivo dei seguenti passi:
 - Analisi del problema
 - Individuazione di una soluzione efficiente
 - Stesura del codice
 - documentazione delle scelte effettuate e del codice prodotto.



Introduzione al Corso

Programma del Corso:

- Breve Introduzione al C:
 - Origini del C e sue relazioni con altri linguaggi di programmazione, Librerie, Tipi di dati, Espressioni ed istruzioni, Operazioni di input/output, Procedure e funzioni.
- Tecniche di progetto (divide-et-impera, ordinamento, ricerca, selezione) e analisi asintotica
- Implementazione delle seguenti strutture dati:
 - Stack e Code, Liste semplici, doppiamente puntate e circolari, Heap binari e code di priorità, Alberi binari di ricerca.
- Definizione della struttura dati grafo e sua rappresentazione in memoria
- Algoritmi su grafi
 - Creazione, interrogazione e modifica di grafi rappresentati con matrici e liste di adiacenza Visita di grafi (BFS e DFS).



Introduzione al Linguaggio C (Prima parte)

Cenni storici

Caratteristiche generali

- Struttura di un programma C
- Variabili e costanti

Istruzioni elementari

- Espressioni ed operatori
- Assegnazioni
- Input/output

Istruzioni di controllo del flusso

- Istruzioni condizionali
- Iterazioni



Introduzione al C

Cenni Storici

- Il linguaggio C è stato sviluppato intorno al 1972 nei Bell Laboratories AT&T americani, da Dennis Ritchie.
- E' nato come linguaggio di sviluppo del Sistema Operativo UNIX.
- Gli antenati del C possono essere riuniti in linea genealogica:
 - **Algol 60** 1960 (Comitato Int.)
 - regolarità della sintassi, struttura in moduli, particolarmente complesso
 - **CPL** (Combined Programming Lan.) 1963 e **BCPL** 1967 (Cambridge)
 - migliorarono le caratteristiche dell'Algol ma non la complessità.
 - **B** 1970 (Thompson)
 - molto legato alla struttura dell'hardware.
 - **C** 1972 (Ritchie)
 - riassume le migliori caratteristiche dei precedenti
- 1986: Objective C (Cox), C++ (Stroustrup)
- 1999: Ultima release dello standard ANSI



Introduzione al C

Perché programmare in C

- Portabilità del Codice e del Compilatore
- Codice generato molto efficiente
- Facilità di accesso al livello “macchina”
- Interfacciamento completo al S.O. UNIX
- Varietà di operatori di linguaggio
- Strutture dati potenti
- Non complesso (poche keywords)
- Modularità e Riuso



Introduzione al C

Passi fondamentali:

- Scrivere un codice sorgente in linguaggio C con un **editor**
- Produzione di codice eseguibile utilizzando un **compilatore**
- Dev C++ è un compilatore Free (under the GNU General Public License)

Risorse:
[DevC++](#)
[Html.it](#)



Introduzione al C

Struttura del Programma Sorgente C

- Un **programma sorgente C** è formato da uno o più blocchi chiamati **funzioni**
- La definizione di una funzione rappresenta la specificazione delle operazioni che dovranno essere svolte all'atto della chiamata.
- La definizione è costituita da due parti:
 - ***intestazione dichiarativa***
 - ***Corpo***
- **L'intestazione** definisce le regole di interfaccia.
- Il **corpo** specifica le operazioni da eseguire ed è formata da un insieme di istruzioni delimitato utilizzando le parentesi graffe

{ corpo }



Introduzione al C

Le istruzioni

- Le istruzioni C terminano con un punto e virgola (;)
- Esempio: `printf("Stampa questa riga \n");`
- Generalmente, un'istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata chiaramente dal punto e virgola finale.
- Raggruppamenti di istruzioni si fanno utilizzando le parentesi graffe { }
- Esempio: `<istruzione>; {<istruzione>; <istruzione>;}`



Introduzione al C

Struttura del Sorgente (2)

- Altri oggetti fondamentali di un codice sorgente in linguaggio C sono:
 - ***Direttive del preprocessore***
 - ***Commenti***
- Le direttive del preprocessore guidano alla compilazione del codice. L'uso più comune riguarda l'inclusione di codice esterno al file sorgente (librerie), composto da file che terminano con l'estensione ".h". Tali istruzioni iniziano con il simbolo "#".
- I commenti vengono indicati tra i simboli /* e */.



Introduzione al C

Esempio di Programma C

- Il seguente è un semplice esempio di programma scritto in linguaggio C

```
#include <stdio.h>
main() /* esempio di programma C */
{
    printf("Primo programma C\n");
}
```

- Questo programma ha una sola funzione “**main()**”.
- In un programma C esiste un solo **main** e l'esecuzione del programma corrisponde alla chiamata di tale funzione.
- Si notino inoltre la parentesi graffa aperta “{” per l'inizio del corpo della funzione e la parentesi graffa chiusa “}” per la fine del corpo della funzione



Introduzione al C

Standard C library

- Esaminiamo l'istruzione

```
printf ("Primo programma C\n");
```
- **printf** è il nome di una funzione il cui codice è già scritto ed inserito nella **standard C library "stdio.h"**.
- Questa libreria viene inclusa durante la fase di compilazione del programma attraverso la direttiva del preprocessore **#include <stdio.h>**.
- La libreria standard è necessaria alla gestione dei flussi di standard input, standard output e standard error.
- Altre funzioni appartenenti a questa libreria sono:
 - **printf()**, **fprintf()**, **sprintf()**, **scanf()**, **fscanf()**, **sscanf()**,
getc(), **gets()**, **getchar()**, **putc()**, **puts()**, **putchar()**, **fgetc()**,
fgets(), **fputs()**, **fwrite()**, **fread()**, etc...



Introduzione al C

Le variabili

- **Una *variabile* è un'astrazione di una o più celle di memoria.**
- Un **identificatore** è il nome associato ad un variabile e consiste di un numero qualsiasi di caratteri alfanumerici minuscoli o maiuscoli incluso il carattere "_" (underscore).
- Il primo carattere deve essere una lettera oppure underscore.
- Il compilatore fa differenza tra lettere minuscole e maiuscole.
- **Esempi validi:** sp_addr sp2_addr F_lock_user_found
- **Esempi non validi:** 20_secolo -pippo
- Ogni identificatore possiede due attributi che lo caratterizzano:
 - **classe di memoria:** determina la durata della memoria associata alla variable
 - **Tipo:** determina il significato dei valori assunti dalla variabile



Introduzione al C

Parole chiave

- Alcuni identificatori sono parole riservate (**keywords**) del linguaggio e pertanto non possono essere usate come nomi di variabili.
- Le **keywords** del linguaggio C standard sono 32:

| | | | |
|----------|--------|----------|----------|
| auto | double | int | struct |
| break | else | long | switch |
| case | enum | register | typedef |
| char | extern | return | union |
| const | float | short | unsigned |
| continue | for | signed | void |
| default | goto | sizeof | volatile |
| do | if | static | while |



Classi di Memoria

Classi di Memoria

- Esistono due classi di memoria: **automatica** e **statica**
- La classe di memoria automatica è relativa a quegli oggetti locali ad un blocco (funzione o programma) che viene liberata non appena si raggiunge la fine di quel blocco.
- La classe di memoria statica è relativa a quegli oggetti locali ad un blocco od esterni a qualsiasi blocco che non viene liberata tra uscite ed entrate successive tra diversi blocchi.
- Se non esistono altre specificazioni ogni oggetto dichiarato in un blocco ha classe di memoria automatica, se la loro definizione è accompagnata dalla parola chiave “**static**”, allora la loro classe di memoria è statica.
- Gli oggetti dichiarati all'esterno di qualsiasi blocco (variabili globali) hanno sempre classe di memoria statica.



Tipi di dati

Tipi di dati

- Tutte le variabili devono essere dichiarate con il loro tipo prima di essere utilizzate
- Ci sono 4 tipi base in C:
 - **char** (carattere): 8 bit
 - **int** (intero): 16 bit
 - **float** (numero in virgola mobile): 32 bit
 - **double** (float a doppia precisione): 64 bit



Interi

Interi

- La dichiarazione di variabili intere deve essere posta all'inizio di un blocco di codice.
- **Esempio:** `int a,b,c;`
- Dopo che una variabile è stata dichiarata è possibile assgnarle un valore intero tramite l'operatore di assegnamento
- **Esempio:** `a = 100;`
- Avendo a disposizione un calcolatore che memorizza interi a 16 bit, i due valori estremi che possono essere assegnati ad un intero sono: -32768 e +32767



Introduzione al C

Caratteri

- Le variabili di tipo carattere vengono dichiarate nel modo eseguente:
char anno, mese;
- Per assegnare un valore carattere **A** ad una variabile **c** di tipo **char** usiamo la seguente sintassi
c='A'
- In effetti, nel linguaggio C i caratteri sono valori numerici ai quali per convenzione sono associate lettere dell'alfabeto, segni di interruzione ed altri simboli alfanumerici.
- La codifica più comunemente usata è la codifica **ASCII** (American Standard Code for Information Interchange)
- Ad esempio la lettera 'A' viene codificata come 65 mentre la minuscola 'a' corrisponde a '97'



Introduzione al C

Dati in virgola mobile

- I dati floating point sono una approssimazione dei reali, espressi come frazioni decimali.
- Per esempio: 3.14159, 2.71828
- Solitamente vengono utilizzati 32 bit per la rappresentazione interna di questi dati, di cui almeno i primi 6 sono i più significativi.
- Il range di variazione dei floating point è: 1...e-39 fino a 1...e+38



Introduzione al C

Costanti

- In C esistono diversi tipi di costanti, in cui possiamo distinguere 5 tipi fondamentali:
 - costanti **intero**
 - costanti esplicitamente **long**
 - costanti **carattere**
 - costanti in **virgola mobile**
 - **stringhe di caratteri** costanti

Costanti particolari:

- New line (lf) '\n'
- Carriage return (cr) '\r'
- Backspace (bs) '\b'
- Horizontal tab (tab) '\t'
- Backslash (\) '\\'



Introduzione al C

Operatori

- Gli operatori sono classificati secondo le tre seguenti categorie

Aritmetici:

- "+" somma,
- "-" sottrazione,
- "*" moltiplicazione,
- "/" divisione,
- "%" modulo

Relazionali:

- "<" minore di,
- ">" maggiore di,
- "<=" minore o uguale,
- ">=" maggiore o uguale,
- "==" uguale a,
- "!=" diverso da

Logici:

- "&&" AND logico "&" AND bit a bit
- "||" OR logico "!" OR bit a bit
- "!" NOT logico "~~" NOT bit a bit



Introduzione al C

Operatori aritmetici e di assegnamento

- Si consideri l'espressione:

anno = anno + 1;

che significa: " prendere il valore corrente della variabile anno, sommargli 1 e memorizzare il risultato nuovamente in anno".

- Nell'espressione sono presenti due operatori “+” e “=” che implicano l'esecuzione dell'operazione in due passi.
- L'ordine di esecuzione è: somma e poi assegnamento.



Introduzione al C

Operatori di incremento e decremento

- Questi operatori sommano o sottraggono 1 all'operando cui sono applicati.
- Esempi: **a++** incrementa **a** di 1; **a--** decrementa **a** di 1
- La posizione degli operatori di incremento e decremento può essere *prefissa* o *suffissa*.
- Esempio: (per a=5):**
 - b =++a; → a=6 e b=6
 - b =a++; → a=6 e b=5

Operatori di assegnamento composti

- Sono operatori assegnamento opportunamente composti con quelli aritmetici
- L'elenco completo degli assegnamenti è il seguente:
- = += -= *= /= %=



Introduzione al C

Printf e Scanf con formattazione

- Si consideri il seguente codice:

```
#include <stdio.h>
main()
{
    int a,b,c;
    printf("\nIl primo numero e'");
    scanf("%d",&a);
    printf("Il secondo numero e'");
    scanf("%d",&b);
    c=a+b;
    printf("Il totale e' %d \n",c);
}
```

- Alcuni tipi di % che possono essere usati in ANSI C sono:

- %c char single character
- %d (%i) int signed integer
- %e float or double exponential format
- %f float or double signed decimal
- %s array of char sequence of characters



Introduzione al C

Strutture di controllo

■ SEQUENZIALI

- Statement semplice
- Statement composto

■ CONDIZIONALI

- if (< expr.>) { } else { }
- switch(<expr.>) { case < cost> : ... }

■ ITERATIVE

- while(<expr.>) { }
- for (...) { }
- do { } while(< expr.>)

Introduzione al C

Statement Semplici

- Uno statement semplice può essere :
 - assegnamento
 - espressione
 - chiamata a funzione
- Esempio:

```
main()
{
    int x;
    x = - 456; /* assegnamento */
    x = x + 1; /* espressione */
    printf (" X = %d \n ", x); /* funzione */
}
```



Introduzione al C

Istruzione "IF - ELSE"

- Lo statement "if - else" è usato per prendere delle decisioni all'interno di un programma

- Sintassi:**

```
if ( espressione )
    istruzione;
```

- Sintassi alternativa:**

```
if ( espressione ) {
    istruzione_1;
    istruzione_2;    }
else
    istruzione_3;
```

- Semantica**

- L'espressione viene valutata e, se risulta essere TRUE o NON ZERO, viene eseguito il set di istruzioni corrispondente.

Esempio:

```
if ( x > 0 )
{
    if ( k == m )
        y = m;
}
else
{
    y = 0;
    k = 100;
}
```



Introduzione al C

Operatore Ternario

- L'operatore ternario è una forma compressa dell'operatore **if - else**.
- **Sintassi:**
< expr > ? < expr1 > : < expr2 >;
- **Semantica:**
 - Se *< expr >* è vero viene valutato solo *< expr1 >* altrimenti viene valutato *< expr2 >*
- **Esempio:**

max = (alfa > beta) ? alfa : beta;

che corrisponde a...

```
if (alfa > beta)
    max = alfa;
else
    max = beta;
```



Introduzione al C

Istruzione else-if

- Quando occorre effettuare una scelta plurima è possibile, ma non consigliabile, utilizzare l'istruzione else-if.
- **Sintassi:**

```
if (espressione)
    <istruzione/i>
else if (espressione)
    <istruzione/i>
else if (espressione)
    <istruzione/i>
else
    <istruzione/i>
```



Introduzione al C

Istruzione Switch

- Trasferisce l'elaborazione a uno o più statement composti, in funzione della valutazione di una espressione.
- È usata quando occorre effettuare scelte plurime.
- L'esecuzione delle istruzioni avviene in modo SEQUENZIALE a partire dalla prima espressione **cost_x** che risulta vera e fino al termine dell'istruzione switch.
- Ciò permette di associare più possibilità ad un'unica etichetta.
- Per evitare l'esecuzione di istruzioni successive a quelle corrispondenti alla condizione soddisfatta si usa il comando **break**

Sintassi:

```
switch ( <expr> )
{
    case <cost_1>:
        statement;
    case <cost_2>:
        statement;
    case <cost_3>:
        statement;
    default
        statement;
}
```



Introduzione al C

Esempio di Switch

```
#include <stdio.h>
main()
{
    int i;
    printf("Enter a number between 1 and 3");
    scanf("%d",&i);
    switch (i)
    {
        case 1:
        printf("one"); break;
        case 2:
        printf("two"); break;
        case 3:
        printf("three"); break;
        default:
        printf("unrecognized number");
    } /* end of switch */
}
```



Introduzione al C

Istruzione "while"

- L'istruzione **while** permette di eseguire una serie di istruzioni fintanto che una condizione iniziale rimane **TRUE**.

- Sintassi:**

```
while ( <expr> ) {  
    statement;  
}
```

- Esempio**

```
a = 1;  
while ( a <= 100) {  
    total += a*a;  
    a += 1;  
}
```

Istruzione "do-while"

- L'istruzione **do while** permette di eseguire delle istruzioni e poi di rieseguirle fintanto che una condizione iniziale rimane **TRUE**.
- Dunque, un ciclo **do while** è eseguito almeno una volta e la condizione viene valutata dopo l'esecuzione delle istruzioni.

- Sintassi:**

```
do {  
    statement;  
} while ( <expr> );
```

Introduzione al C

Istruzione "for"

- L'istruzione for è molto simile al while.
- Sintassi:

```
for ( <inizializza var>; <expr>; <aggiorna var>)      {  
    statement;  
}
```
- Equivalente alla struttura seguente che utilizza while:

```
< inizializza variabili>;  
while ( <expr>)          {  
    < statement >;  
    < aggiorna variabili >;  }
```
- Esempio:

```
main()                                {  
    int sum , i;  
    sum = 0;  
    for ( i = 1 ; i <= 10; i++) {  
        sum += i;                  }  
}
```



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Linguaggio C – Seconda Parte

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 2

Parole chiave: **Funzioni, Array, Puntatori, Preprocessore**

next



Percorsi di Formazione a Distanza
e-Learning

Università degli Studi di Napoli Federico II



Centro di Ricerca e Sviluppo
C.R.S.



Progetto



Progetto



Progetto

Realizzata con il coinvolgimento del Istituto Universitario Lattesco. Misura 3.12. Maturazione anno I – P.D.R. Campagna 2000-2006

Indice della lezione

- **Funzioni**
- **Array**
- **Puntatori**
- **Preprocessore**

Funzioni

Nel C è possibile scomporre problemi complessi in moduli più semplici sfruttabili singolarmente.

Le funzioni sono blocchi di programmi indipendenti da altri moduli, ciascuno destinato ad una precisa operazione.

Un programma nel C non è altro che una grossa funzione *main()* che ingloba nel suo interno altre funzioni.

La comunicazione tra i diversi moduli avviene mediante gli argomenti, i valori di ritorno e le variabili esterne.

L'uso delle funzioni consente di nascondere l'implementazione di una certa operazione e concentrarsi solo sul "**cosa fa**" e non sul "**come lo fa**".



Funzioni (2)

Una funzione può:

- compiere un'azione: provoca il verificarsi di una certa azione
- effettuare un calcolo: il risultato del calcolo viene restituito dalla funzione stessa

Una funzione viene definita nel seguente modo:

- *tipo-ritornato nome_f (dichiarazione argomenti)*
- *{dichiarazioni ed istruzioni }*

Ogni funzione presenta un valore di ritorno:

- Può essere di qualsiasi tipo predefinito o definito dall'utente.
- Nel caso di funzione che compie un'azione, ovvero non deve ritornare nessun valore, si usa il tipo predefinito **void** come valore di ritorno.
- Una funzione può avere o meno una lista di argomenti



Controllo dell'esecuzione

All'atto della chiamata di una funzione il controllo nell'esecuzione viene passato alla prima istruzione del corpo della funzione stessa.

Esistono due modi per restituire il controllo al programma chiamante:

- attraverso l'istruzione: `return espressione;`
- termine dell'esecuzione della funzione `}"`

E' opportuno controllare sempre che la chiamata di una funzione ed il suo valore di ritorno siano consistenti.



Liste degli argomenti

La lista argomenti è usata per passare dati ad una funzione chiamata. La lista argomenti può essere anche vuota.

Le variabili da passare devono essere specificate tra parentesi dopo il nome della funzione.

Nel corpo della funzione le variabili devono essere dichiarate con il loro tipo corrispondente.

Esempio:

```
int lower(int c)
{
    int k;
    k = (c >= 'A' && c <= 'Z') ? (c + 'a' - 'A') : (c + 'A' - 'a');
    return k;
}
```



Argomenti di una funzione

Il metodo di passaggio delle variabili è **per valore**, ossia, nella funzione chiamata si farà una copia locale delle variabili passate e non si potrà modificarne il loro valore globale, a meno di non passare alla funzione il **"riferimento"** della variabile.

E' responsabilità del programmatore controllare che il numero ed il tipo degli argomenti passati ad una funzione corrisponda con quanto specificato nella dichiarazione della funzione stessa.

Con l'introduzione dei **prototipi** ad ogni funzione è assegnato un prototipo di chiamata, quindi il compilatore è in grado di controllare tipi e numero di argomenti passati.



Prototipi

Il prototipo di una funzione non rappresenta altro che una dichiarazione di funzione antecedente alla sua definizione:

```
int function(int valore, char carattere);
```

Nell'ANSI C l'uso dei prototipi è obbligatorio quando la definizione di una funzione avviene successivamente al suo utilizzo.

Lo scopo dei prototipi delle funzioni è quello di permettere al compilatore di compiere il controllo sui tipi degli argomenti che vengono passati ad una funzione.



Prototipi

Se i prototipi vengono inseriti prima della definizione di una funzione, il compilatore quando incontra la dichiarazione conosce il numero ed i tipi degli argomenti e può così controllarli.

La dichiarazione e la definizione di una funzione devono essere consistenti sia nel numero che nel tipo dei parametri.

Attenzione: se dichiarazione e definizione di funzione si trovano nello stesso file sorgente eventuali non corrispondenze nei tipi degli argomenti saranno rilevati dal compilatore, in caso contrario al più ci sarà un warning.



Array

Un array è una collezione di variabili dello stesso tipo che condividono un nome comune.

Un array viene dichiarato specificando il tipo, il nome dell'array e uno o più coppie di parentesi quadre contenenti le dimensioni.

Esempio:

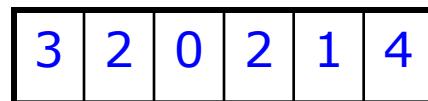
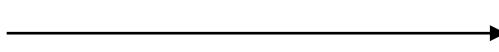
- `int nomi[4];` /* 1 dimensione di 4 interi */
- `float valori[3][4]` /* 2 dimensioni di 12 float */
- `char caratteri[4][3][5][7]` /* 4 dimensioni 420 elementi */



Rappresentazione di Array

Il C memorizza i valori degli elementi di un array in locazioni consecutive di memoria.

`int stanze[6]`



- *locazione base + 0: stanze[0] "3"*
- *locazione base + 1: stanze[1] "2"*
- *locazione base + 2: stanze[2] "0"*
- *locazione base + 3: stanze[3] "2"*
- *locazione base + 4: stanze[4] "1"*
- *locazione base + 5: stanze[5] "4"*

E' molto importante conoscere come sono stati memorizzati gli elementi poiché questo consente di capire come sia possibile puntare ad un preciso elemento.



Dichiarazione

In C è obbligatorio specificare in modo esplicito la dimensione di un array in fase di dichiarazione.

In una definizione di funzione, come vedremo, non occorre specificare la dimensione del vettore passato come parametro; sarà il preprocessore a risolvere l'ambiguità all'atto della chiamata.

Esempio:

```
int somma(int numeri[],dimensioni)
.....
main()
int num[3];
{
.....
totale = somma(num,3);
}
```



Array multidimensionali

- Gli array multidimensionali vengono dichiarati specificando il numero di elementi per ciascuna dimensione.
- Un array bidimensionale con 6 elementi per ciascuna dimensione viene dichiarato come: `int alfa[2][6];`
- Per referenziare un singolo elemento è necessario utilizzare due coppie di parentesi quadre: `alfa[1][2] = 1;`
- In pratica un array multidimensionale è una collezione di oggetti, ciascuno dei quali è un vettore.
- Esempio:

```
main()
{
    int tabelline[10][10]; int i, j;
    for (i = 0 ; i < 10 ; i++)
        for(j = 0 ; j < 10 ; j++)
            tabelline[i][j] = (i + 1) * (j + 1);
}
```



Indirizzamento

Per accedere ad un array si usano gli indici.

E' compito del programmatore fare in modo di non andare oltre i limiti di dimensione dell'array in questione.

Gli array partono dall'indice 0 fino alla lunghezza dichiarata meno 1;

Referenziare un array al di fuori dei suoi limiti può portare a errori di indirizzamento (**memory fault**) oppure può "sporcare" altre variabili in memoria diventando così molto difficile da localizzare.

Per copiare elementi da un array all'altro bisogna copiare singolarmente ogni elemento.



Array di stringhe

Una stringa è un array monodimensionale di caratteri ASCII terminati da un carattere *null* '\0'

Ad esempio "Questa è una stringa" è un array di 21 caratteri.

L'array è quindi il seguente:

elemento zero 'Q'

primo elemento 'u'

secondo elemento 'e'

.....

20^{mo} elemento 'a'

21^{mo} elemento '\0'

Esempio:

```
/* Stampa carat. e codifica ASCII */
char str[] = "Questa è una stringa";
main()
{
    int i = 0;
    while ( str[i] != '\0' )
    {
        printf("%c=%d\n", str[i], str[i] );
        ++i;
    }
}
```



Array come argomenti di funzioni

Il metodo di default di passaggio delle variabili è per valore, quindi si potrà modificare solo una copia locale della variabile.

Eccezione a questa regola globale sono gli array.

Gli array, infatti, vengono sempre passati per **reference** ossia è il loro indirizzo che viene passato invece del loro contenuto.

Questo consente, solo nel caso dei vettori, di poter agire direttamente sulla variabile e non sulla copia. Ovvero le azioni che si effettuano sull'argomento della funzione avranno effetto anche al termine dell'esecuzione della funzione.

Puntatori

Un **puntatore** è una variabile che contiene l'indirizzo di un'altra variabile.

I puntatori sono “**type bound**” cioè ad ogni puntatore è associato il tipo a cui il puntatore si riferisce.

Nella dichiarazione di un puntatore bisogna specificare un asterisco (*) di fronte al nome della variabile pointer.

Esempio:

`int *pointer;` puntatore a intero

`char *pun_car;` puntatore a carattere

`float *flt_pnt;` puntatore a float



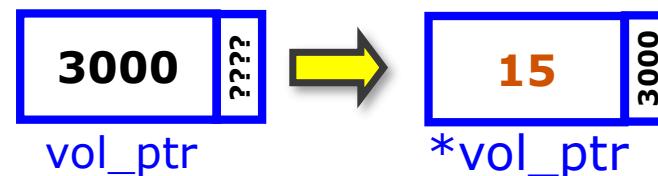
Inizializzazione di puntatori

Un pointer, prima del suo utilizzo, deve essere inizializzato, ovvero deve contenere l'indirizzo di un oggetto.

Per ottenere l'indirizzo di un oggetto si usa un operatore unario "**&**".

```
int volume, *vol_ptr;  
vol_ptr = &volume;
```

Esempio:



Il puntatore `vol_ptr` contiene ora l'indirizzo della var. `volume`.

Per assegnare un valore all'oggetto puntato da `vol_ptr` occorre utilizzare l'operatore di indirezione "*****".

```
*vol_ptr= 15;
```

Esempi

```
/* Dichiarazioni */  
int v1, v2, *v_ptr;  
/* moltiplica v2 per il valore puntato da v_ptr */  
v1 = v2* (*v_ptr);  
/* somma v1,v2 e il contenuto di v_ptr */  
v1 = v1+v2 +*v_ptr;  
/* assegna a v2 il valore che si trova tre interi dopo v_ptr */  
v2 = *(v_ptr + 3);  
/* incrementa di uno l'oggetto puntato da v_ptr */  
*v_ptr += 1;
```



Puntatori

Vantaggi nell'uso dei puntatori:

- Riduzione quantità di memoria statica usata dal sistema.
- Passaggio di parametri per “indirizzo” nelle chiamate a funzioni
 - In questo modo è possibile rendere globale ogni modifica sui parametri passati. Viceversa, se i parametri vengono passati per valore, ogni modifica della funzione sui parametri si perde all’uscita della funzione.

Aritmetica dei puntatori:

- Assegnamento tra puntatori dello stesso tipo,
 - `int *ptr1, *ptr2; *ptr1 = 1; ptr2 = ptr1;`
- Addizione e sottrazione tra puntatori ed interi,
 - `int *ptr, arr[10]; ptr = &arr[0];`
 - `ptr = ptr + 4 /* punta al quinto elemento dell’array arr[4] */`



Puntatori e stringhe di caratteri

- Molto spesso vengono usati i puntatori a caratteri in luogo degli array di caratteri (stringhe), questo perché il C non fornisce il tipo predefinito stringa.
- Esiste una differenza sostanziale tra array di caratteri e puntatori a carattere. Ad esempio in:
 - **char *ptr_chr = "Salve mondo";**
 - il compilatore non crea una copia della stringa "Salve mondo"
 - Il compilatore crea un puntatore che punta ad una locazione di memoria in cui risiede il primo carattere della stringa costante.
 - Quindi è possibile modificare il puntatore senza modificare il contenuto della stringa costante.
- Esempi di inizializzazione di array di caratteri:
 - **char caratteri[4] = { 'a' , 'A' , 'H' , 'k' };**
 - **char stringa_2 [] = "MMMM";**



Puntatori come argomenti di funzioni

- Se l'argomento di una funzione è una variabile puntatore il passaggio della variabile avviene per **reference** (per indirizzo) ossia la funzione chiamata sarà in grado di modificare il valore globale della variabile che riceve.
- Passaggio argomenti per valore:
`int numero; square(numero);`
- Passaggio per indirizzo:
 - `square(&numero);`

```
void swap(int *x_ptr, int *y_ptr)
{
    int temp;
    temp = *x_ptr;
    *x_ptr = *y_ptr;
    *y_ptr = temp;
}
```

```
main()
{
    int a = 3;
    int b = 5;
    swap(&a,&b);
    printf("%d %d\n",a,b);
}
```



Array di puntatori

- Un array di puntatori è un array i cui elementi sono dei puntatori a variabili: `int *arr_int[10]`
- `arr_int[0]` contiene l'indirizzo della locazione di memoria contenente un valore intero.
- Nel C i puntatori a caratteri vengono usati per rappresentare il tipo stringa che non risulta definito nel linguaggio, e gli array di puntatori per rappresentare stringhe di lunghezza variabile.
- Un insieme di stringhe potrebbe essere rappresentato come un array bidimensionale di caratteri, ma ciò comporta uno spreco di memoria.
- Ad esempio:
 - `char *term[100]` - Indica che gli elementi di `term` sono dei puntatori a carattere, cioè `term[i]` è l'indirizzo di un carattere.
 - `term[7] = "Ciao"` - Indica che il contenuto di `term[7]` è il puntatore alla stringa "Ciao";



Esempio

```
#include <stdio.h>
main()
{
    char *giorni[7] = {"Lunedì","Martedì","Mercoledì","Giovedì",
    "Venerdì","Sabato","Domenica"};
    int i;
    for( i=0; i< 7; i++)
    {
        printf("\n %d ",*giorni[i]);
        printf("%s",giorni[i]);
    }
}
```

76 Lunedì || 77 Martedì ||

- **Domanda:** quale istruzione devo scrivere per stampare la "b" di Sabato?



Preprocessore C

Il **preprocessore C** è una estensione al linguaggio che fornisce le seguenti possibilità:

- definizione delle costanti
- definizione di macro sostituzioni
- inclusione di file
- compilazione condizionale

I comandi del preprocessore iniziano con # nella prima colonna del file sorgente e non richiedono il ";" alla fine della linea.

Un compilatore C esegue la compilazione di un programma in due passi successivi. Nel primo passo ogni occorrenza testuale definita attraverso la direttiva # viene sostituita con il corrispondente testo da inserire (file, costanti, macro)



Preprocessore C : Costanti

Attraverso la direttiva `#define` del preprocessore è possibile definire delle costanti:

Sintassi:

| #define | nome | testo da sostituire : |
|----------------------|-------------|------------------------------|
| <code>#define</code> | MAXLEN | 100 |
| <code>#define</code> | YES | 1 |
| <code>#define</code> | NO | 0 |
| <code>#define</code> | | ERROR "File non trovat\n" |

E' uso comune usare lettere maiuscole per le costanti di `#define`

Perché usare queste costanti?

- favoriscono la leggibilità del programma
- consentono un facile riuso del codice



Preprocessore C : Macro

L'uso della direttiva **#define** consente anche di definire delle macro.

Una macro è una porzione di codice molto breve che è possibile rappresentare attraverso un nome; il preprocessore provvederà ad espandere il corrispondente codice in linea.

Una macro può accettare degli argomenti, nel senso che il testo da sostituire dipenderà dai parametri utilizzati all'atto della chiamata.

Il preprocessore espanderà il corrispondente codice in linea avendo cura di rimpiazzare ogni occorrenza del parametro formale con il corrispondente argomento reale.



Macro : Esempi

```
#define square(x) ((x)*(x))
#define MIN(a,b) ( (a<b)? (a) : (b) )
#define ASSERT(expr) if(!(expr)) printf("error")
```

Nel file sorgente le linee :

```
square(2);
MIN(2,3);
ASSERT (a > b);
```

saranno sostituite in compilazione con

```
((2) * (2));
( (2 < 3) ? (2) : (3) );
if (!(a > b)) printf("error");
```



Cosa non è una Macro

Anche se ciò può trarre in inganno, le macro NON sono funzioni.

Per esempio sugli argomenti delle macro non esiste controllo sui tipi.

Inoltre, una chiamata del tipo : MAX (i++, j++) verrà sostituita con:

((i++ > j++) ? (i++) : (j++);

E' importante stare attenti all'uso delle parentesi, ad esempio in:

#define square(x) x*x

Una chiamata del tipo: x = square(3+1); , genera:

x = 3 + 1 * 3 + 1; --> x = 7 ??????



Preprocessore C : Compilazione condizionale

Le **direttive** : #if #ifdef #ifndef #elif #else #endif consentono di associare la compilazione di alcune parti di codice alla valutazione di alcune costanti in fase di compilazione.

```
#if < espressione_costante>
< statement_1>
#else
<statement_2>
#endif
```

Se l'espressione costante specificata, valutata in compilazione ritorna TRUE allora verranno compilati gli statement_1 altrimenti verranno compilati gli statement_2



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Ordinamento, Ricorsione e Code di Priorità

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 3

Parole chiave: **Ordinamento, Insertion sort, Heapsort,**

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



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 sottoproblemi più piccoli fino ad arrivare ad un sottoproblema banale di cui si conosce immediatamente la soluzione.

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)**

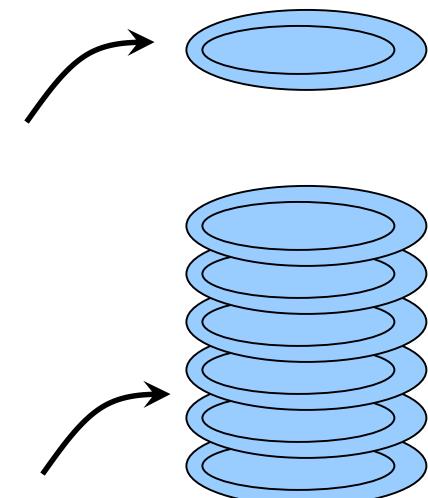
Algoritmo RICORSIVO: *un po' più semplice
del problema intero*
prendo un piatto,

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)
        riposati!
    else // passo ricorsivo
    {
        Prendi un piatto
        Lavallo
        lava_piatti(n-1)
    }
}
```

Condizione di terminazione

// *caso base*

Chiamata
Ricorsiva

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

Una funzione ricorsiva:

- Ha una o più condizioni di terminazione (caso base)
- Chiama se stessa ricorsivamente.
- 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)
stampo 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.

```
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)



Esempio di ricorsione

Calcolo del fattoriale di un numero:

$$n! = n * (n-1) * (n-2) * \dots * (n - (n-1))$$

```
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));
}
```

Esempio di ricorsione

Calcolo del numero di Fibonacci

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

RICORSIVO:

```
#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));
    }
}
```

Algoritmi di ordinamento

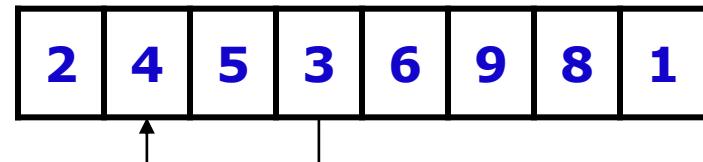


Insertion Sort

- 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.

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;
    }
}
```



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(interii,tot);
    printf("\nArray Ordinato:");
    for (i=0; i<tot; i++)
        printf(" %d",interi[i]);
}
```



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

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 $(1+2+\dots+N-1)$, 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 $O(n \log n)$ 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 suo nodo padre, figlio sx e dx possono essere calcolati nel modo seguente:

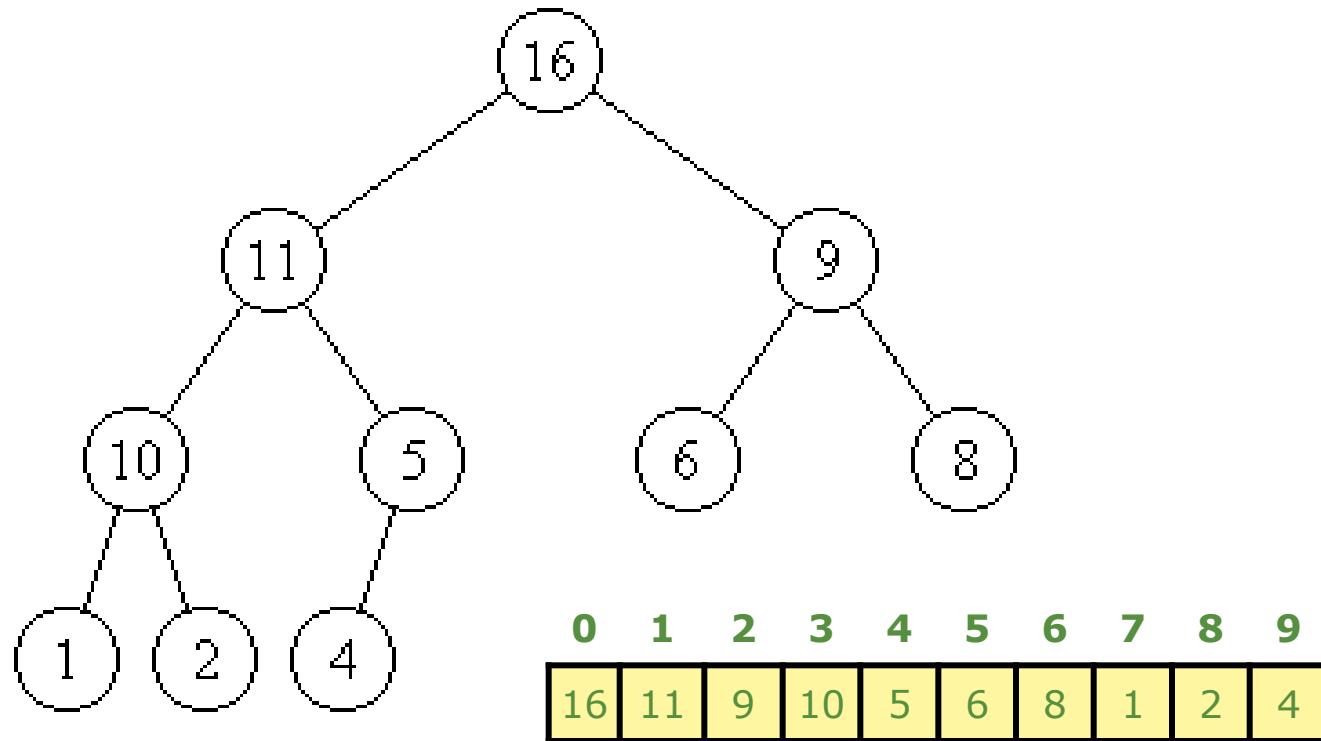
- int left(int i) { return 2*i+1; }
- int right(int i) { return 2*i+2; }
- int parent (int i) {return (i-1)/2; }

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[i/2]
- Left (i) return 2i
- Right(i) return 2i+1

Esempio di Heap

Heap con 10 vertici



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.

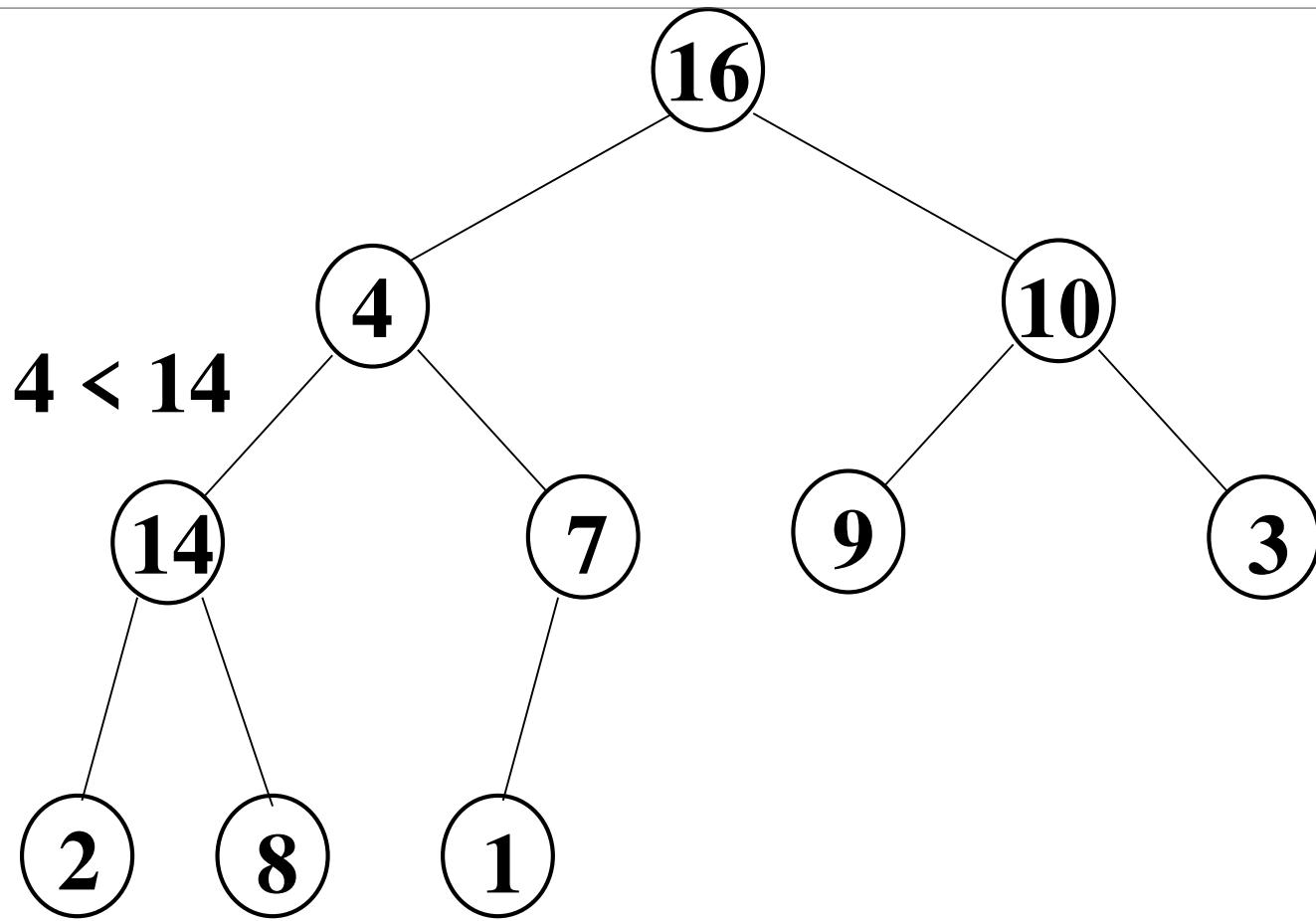
Di seguito mostriamo una funzione ricorsiva Heapify che ha il compito di far scendere il valore di un nodo che viola la proprietà di Heap lungo i suoi sottoalberi.

Implementazione di Heapify

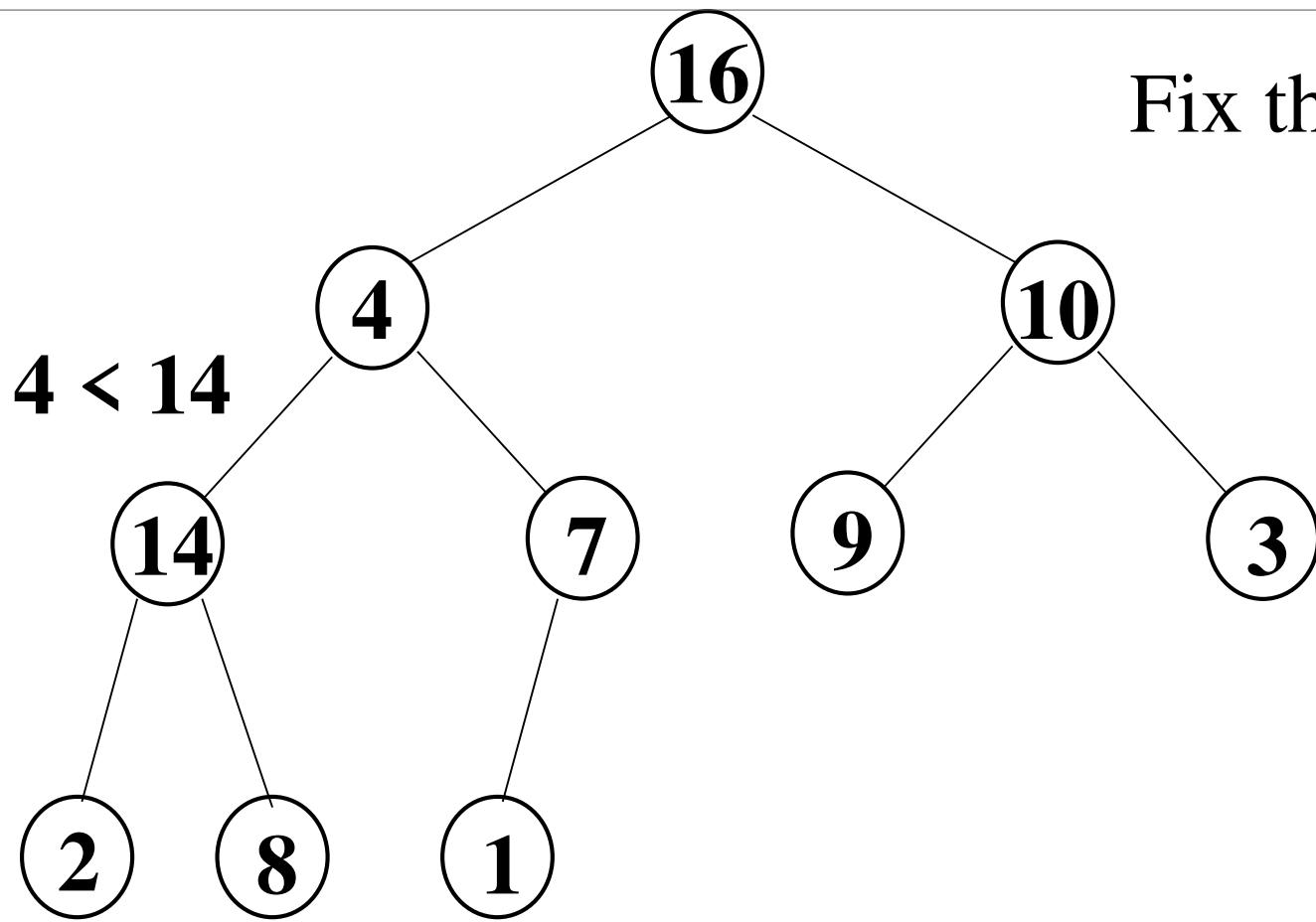
```
void Heapify(int A[MAX], int i)
{
    int l,r,largest;
    l = left(i);
    r = right(i);
    if (l < HeapSize && A[l] > A[i])
        largest = l;
    else largest = i;
    if (r < HeapSize && A[r] > A[largest])
        largest = r;

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

Example of heapify

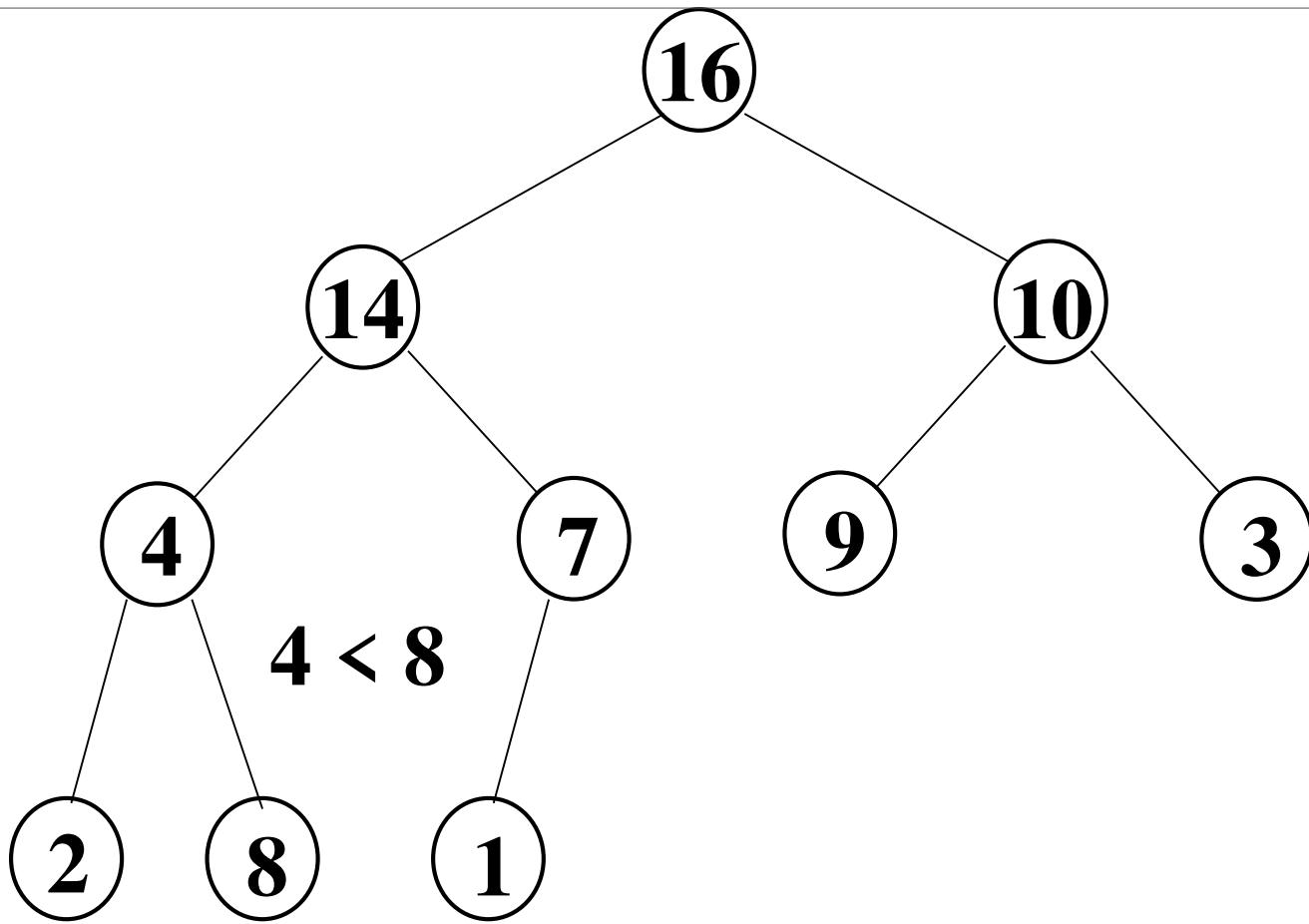


Example of heapify

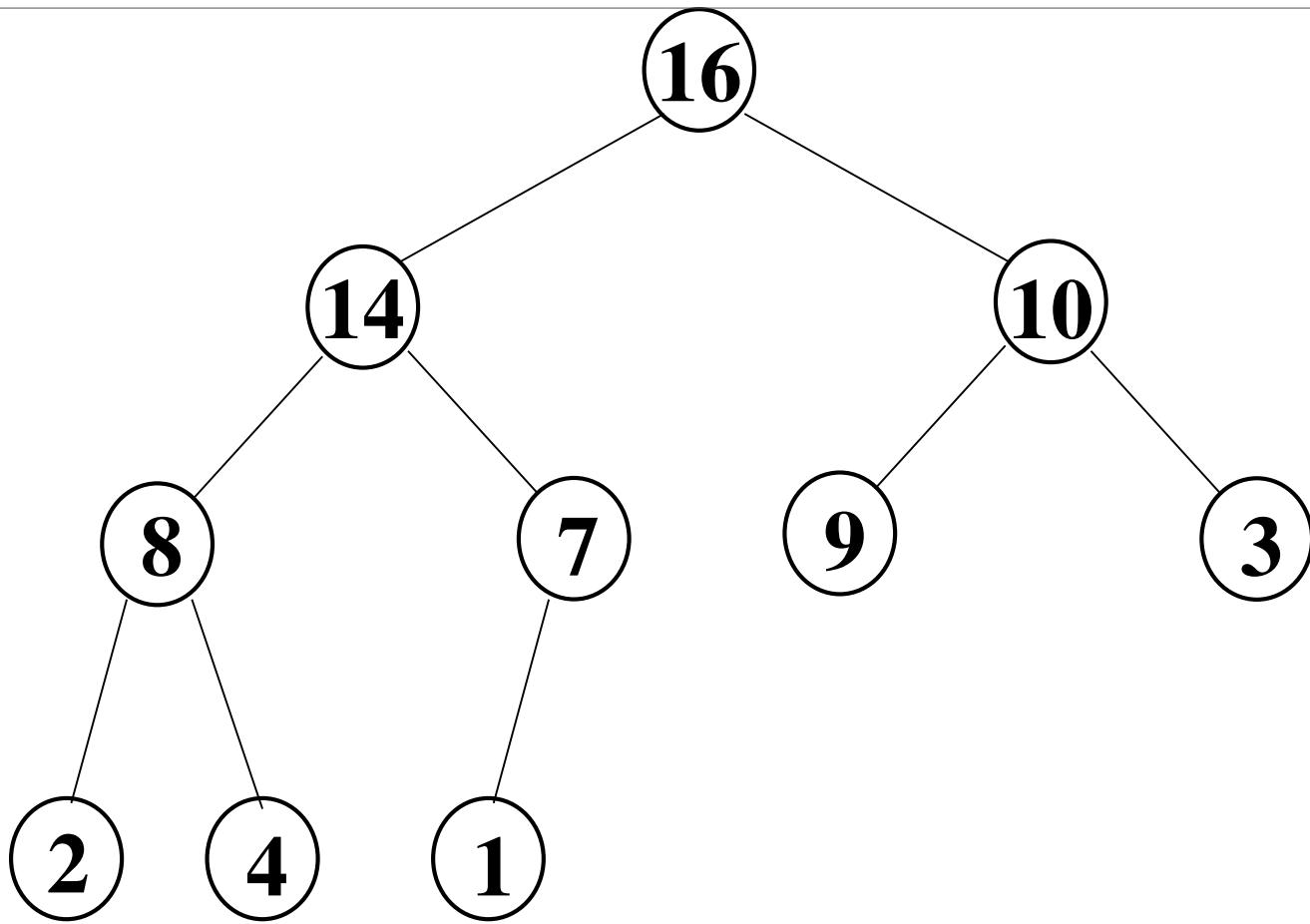


Fix the heap

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);
}
```

Risorse:

 [Esempio di Costruzione dell'heap](#)

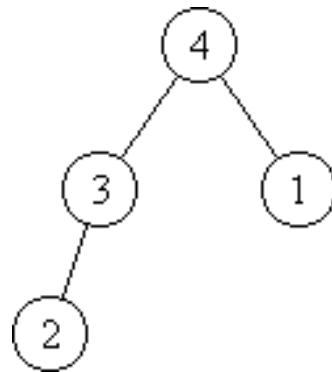
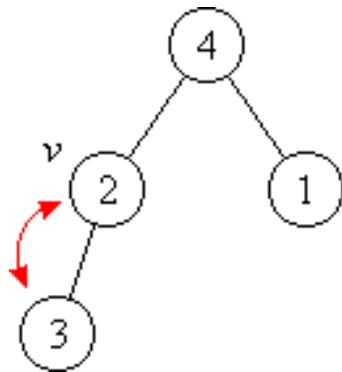
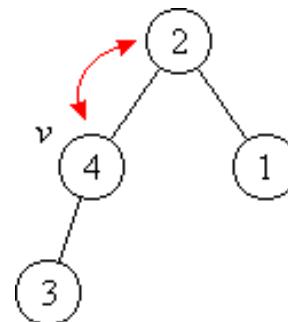
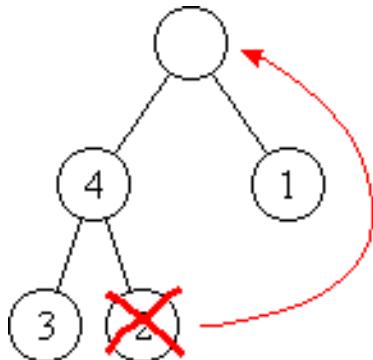
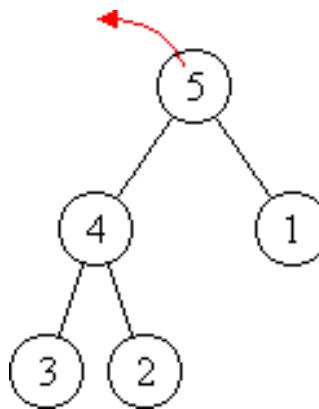
Funzione HeapSort

```
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](#)

Simulazione



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 $O(\log n)$. 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 $O(n\log n)$. 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 $O(n\log n)$
- 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.

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.

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.

Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Esercitazione su Ricorsione e Code di Piorità

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 4

Parole chiave: **Ricorsione, Code a priorita'**

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Realizzata con il coinvolgimento del Istituto Universitario Lattesca - Missa 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006



Missione 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006



Missione 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006



Missione 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006

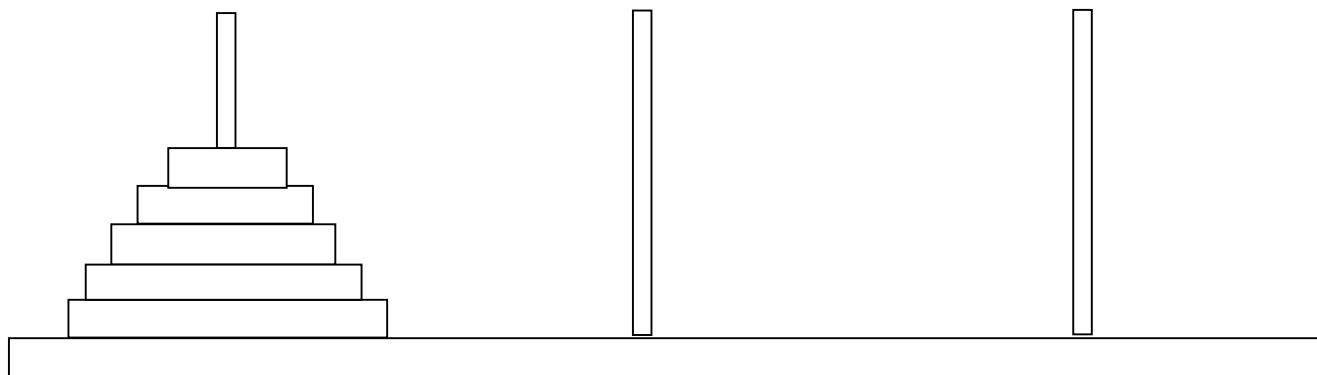


Missione 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006

Torri di Hanoi

Quello delle *Torri di Hanoi* è un gioco che si svolge con tre paletti e alcuni dischi di diametro differente con un foro al centro in modo da poter essere infilati nei paletti.

Inizialmente i dischi sono tutti impilati a piramide sul primo paletto. Il disco più grande è in basso, il più piccolo in alto.



Torri di Hanoi

Scopo del gioco:

- Lo scopo del gioco è quello di trasferire i dischi dal paletto di sinistra a quello di destra, senza mai mettere un disco su un altro di dimensione più piccola.

Regole del gioco:

- È possibile spostare un solo disco alla volta; tutti i dischi devono essere sempre infilati nei paletti.

Strategia:

- La strategia consiste nel considerare uno dei paletti come origine e un altro come destinazione. Il terzo paletto sarà utilizzato come deposito temporaneo.



Strategia

Supponiamo di avere n dischi, numerati dal più piccolo al più grande. Inizialmente sono tutti impilati nel paletto di sinistra. Il problema di spostare n dischi sul paletto di destra può essere descritto in modo ricorsivo così:

- Spostare i primi n-1 dischi dal paletto di sinistra a quello di centro.
- Spostare il disco n-esimo (il più grande) sul paletto di destra.
- Spostare i rimanenti n-1 dischi dal paletto di centro a quello di destra.

In questo modo il problema può essere risolto per qualsiasi valore di $n > 0$ ($n=0$ è la condizione di stop della ricorsione).



Programma

Vogliamo un programma che ci dia la strategia da seguire dato il numero di dischi

- il primo paletto (quello di sinistra) con Sorgente
- il secondo paletto (quello di centro) con Aux
- il terzo paletto (quello di destra) con Destinazione

Definiamo la procedura ricorsiva transfer, che trasferisce n dischi da un paletto all'altro.



Esercitazione su Heap (1)

Si consideri una coda di priorità per la gestione della coda di stampa di una rete implementata con una struttura dati heap H[MAX].

Si implementino le seguenti funzioni:

- void Heapify(int H[MAX], int el); \\ el è un indice di H
- void BuildHeap(int H[MAX]);
- void HeapSort(int H[MAX]);
- int ricerca (int H[MAX], int el); \\ restituisce l'indice del vettore in cui si trova l'elemento el; e -1 se l'elemento non è presente nel vettore



Esercitazione su Heap (2)

- Si consideri una coda di priorità per la gestione della coda di stampa di una rete realizzata con una struttura dati heap $H[MAX]$.
- Sia **heapsize** la variabile che memorizza la dimensione dell'heap
- Si implementi la funzioni

`void annulla_lavoro(int H[MAX], int el),`

che presi in input l'heap e un lavoro el (intero) da eliminare provveda ad eliminare el dall'heap.

- Descrivere la complessità della funzione implementata.

L'esercizio, completo di una breve documentazione (1 pagina), va consegnato via mail al tutor entro 3 giorni lavorativi



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Esercitazione di Laboratorio su
Stack e Code

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 5

Parole chiave: **LIFO, FIFO**

next



Percorsi di Formazione a Distanza
e-Learning

Università degli Studi di Napoli Federico II



Centro di Ricerca e Sviluppo
C.R.S.



European Project



Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II

Realizzata con il coinvolgimento del Istituto Universitario Lattesano - Missa 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006

Esercizio: Prima parte

Si implementino in Linguaggio C una libreria per la gestione di uno Stack che possa contenere al più MAX elementi

Si implementino in Linguaggio C una libreria per la gestione di una Coda che possa contenere al più MAX elementi



Esercizio: Seconda parte

Si implementi inoltre una libreria in linguaggio C per la gestione di una coda di MAX elementi utilizzando la libreria per Stack precedentemente implementata.

In particolare, questa libreria deve prescindere “il più possibile” dal modo in cui sono stati implementati gli stack

Discutere sulle eventuali cambiamenti di complessità di dequeue e enqueue in questa nuova implementazione rispetto alle implementazioni viste nella lezione precedente.



Una Possibile Soluzione al II esercizio

- Siano H e T sue Stack tali che la coda sia il risultato della concatenazione dello Stack H (partendo dal Top al Bottom) con lo Stack T (dal Bottom al Top).
- Nella situazione iniziale, tutti gli elementi sono posti nello Stack H dove l'elemento al Top è la testa (Head) della coda, mentre quello al Bottom rappresenta la fine della coda (Tail)
- Quando H è vuoto allora si svuota il rewerse di T in H. In dettaglio, per ogni elemento di T, si farà il Pop in T e il Push in H, fino a quando T non diventa vuoto.
- Per cancellare un elemento dalla coda, si farà un POP dallo stack H, il quale non sarà mai vuoto a meno che l'intera coda non diventi vuota.
- Per inserire un elemento nella coda si fa un Push nello Stack T.



Esercizio III parte

Scrivere un programma in linguaggio C per la gestione di code che possa funzionare indipendentemente da quale delle due librerie precedentemente definite venga utilizzata

Attenzione: L'implementazione deve avvenire modificando "al minimo" il main() già implementato per la gestione delle code visto nella precedente lezione.

Domanda: Quante volte ciascun elemento sarà oggetto di un Push e di un Pop prima di lasciare la coda?

L'esercizio completo va consegnato via mail al tutor entro 5 giorni lavorativi allegando un breve documentazione (1-2 pagine)



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Implementazioni di Liste Puntate Semplici

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 7

Parole chiave: **Liste dinamiche
singolarmente puntate**

next



Percorsi di Formazione a Distanza
e-Learning

Università degli Studi di Napoli Federico II



Centro di Ricerca e Sviluppo
C.R.S.



Progetto
Europa Plus



Università
degli Studi
di Napoli
Federico II



Università
degli Studi
di Napoli
Federico II

Realizzata con il coinvolgimento del Istituto Universitario Lattesco - Missa 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006

Indice

- **Liste puntate semplici:** Gli elementi sono organizzati in modo sequenziale e si possono scorrere in un unico verso. La lista ha un primo elemento (testa) e un ultimo elemento (coda)



- **Liste doppiamente puntate:** Simili alle liste puntate semplici, ma permettono di scorrere gli elementi in entrambi i versi



- **Liste puntate semplici circolari:** Sono liste puntate semplici senza testa ne coda.



- **Liste doppiamente puntate circolari:** Liste doppiamente puntate senza testa ne coda.



Torniamo al linguaggio C

**Per l'implementazione delle liste in linguaggio C,
possiamo utilizzare due importanti costrutti:**

- STRUTTURE
- ALLOCAZIONE DINAMICA DELLA MEMORIA





Strutture

Le strutture del C sono simili ai record del Pascal:

- sostanzialmente permettono un'aggregazione di variabili, molto simile a quella degli array, ma a differenza di questi non ordinata e non omogenea (una struttura può contenere variabili di tipo diverso).

Per denotare una struttura si usa la parola chiave struct seguita dal nome identificativo della struttura, che è opzionale.

- Nell'esempio sottostante si definisce una struttura "libro" e si crea un'istanza di essa chiamata "biblio":

```
struct libro{  
    char titolo[100];  
    char autore[50];  
    int anno_pubblicazione;  
    float prezzo;    };  
  
struct libro biblio;
```



Strutture (2)

- La variabile "biblio" può essere dichiarata anche mettendo il nome stesso dopo la parentesi graffa:

```
struct libro {  
    char titolo[100];  
    char autore[50];  
    int anno_pubblicazione;  
    float prezzo; } biblio;
```

- Inoltre, è possibile pre-inizializzare i valori, alla dichiarazione, mettendo i valori (giusti nel tipo) compresi tra parentesi graffe:
 - `struct libro biblio = {"Guida al C", "Fabrizio Ciacchi", 2003, 45.2};`
- Per accedere alle variabili interne della struttura si usa l'operatore ".":
 - Esempio: Per assegnare alla variabile interna prezzo il valore 50 usiamo `biblio.prezzo = 50;`



Nuovi tipi di dato

- **Per definire nuovi tipi di dato si utilizza la funzione `typedef`.**
- **Con `typedef` e l'uso di `struct` è possibile creare tipi di dato molto complessi, come mostrato nell'esempio seguente:**
 - `typedef struct libro {
 char titolo[100];
 char autore[50];
 int anno_pubblicazione;
 float prezzo; } t_libro;`
 - Per creare una variabile "guida" di tipo "t_libro", usiamo:
 - `t_libro guida={"Guida al C", "Fabrizio Ciacchi", 2003, 45.2};`



Nuovi tipi di dato

Come per ogni altro tipo di dato, anche con "t_libro" si possono creare degli array:

- t_libro raccolta[5000];

Nel caso di array, per accedere ad un variabile interna, si utilizza l'indice insieme all'operatore punto (.)

- Esempio: Per assegnare il prezzo 50 al libro con indice 10 usiamo raccolta[10].prezzo = 50;



Puntatori e Strutture

Consideriamo il seguente esempio di uso congiunto di strutture e puntatori:

```
struct PIPPO { int x, y; } elemento;  
struct PIPPO *puntatore;  
puntatore = &elemento;  
puntatore->x = 6;  
puntatore->y = 8;
```

Abbiamo dunque creato una struttura di tipo PIPPO e di nome "elemento", ed un puntatore ad una struttura di tipo PIPPO.

Per accedere ai membri interni della struttura "elemento" abbiamo usato l'operatore -> sul puntatore alla struttura.

- In pratica, puntatore->x = 6 semplifica (*puntatore).x=6;



Allocazione dinamica della memoria

A differenza di altri linguaggi, all'occorrenza il C permette di assegnare la giusta quantità di memoria alle variabili del programma.

Le funzioni utilizzate per gestire dinamicamente la memoria delle variabili sono principalmente:

- **malloc()** e **calloc()**, adibite all'allocazione della memoria;
- **free()** che serve per liberare la memoria allocata,
- **realloc()**, che permette la modifica di uno spazio di memoria precedentemente allocato.
- Infine, un comando particolarmente utile è **sizeof**, che restituisce la dimensione del tipo di dato da allocare.

Queste funzioni sono incluse nella libreria malloc.h,



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
main()
{
    int numero=100, allocati, *array, i; char buffer[15];
    printf("Numero di elementi dell'array: %d", numero);
    array = (int *)malloc(sizeof(int) * numero);
    if(array == NULL) { printf("Memoria esaurita\n"); exit(1); }
    allocati = sizeof(int) * numero;
    for(i=0; i<numero; i++) array[i] = i;
    printf("\n Valori degli elementi \n");
    for(i=0; i< numero; i++) printf("%d", array[i]);
    printf("\n\n Numero elementi %d \n", numero);
    printf("Dimensione elemento %d \n", sizeof(int));
    printf("Bytes allocati %d \n", allocati);
    free(array);
    printf("\n Memoria Liberata \n");
}
```



Uso di realloc()

La sintassi della funzione realloc() ha due argomenti, il primo riguarda l'indirizzo di memoria, il secondo specifica la nuova dimensione del blocco;

Esempio di frammento di codice:

```
while(scanf("%d", &x))
{
    allocati += sizeof(int)
    array = (int *)realloc(array, allocati);
    if(array == NULL)
    {
        printf("Memoria insufficiente\n");
        exit(1);
    }
    i++;
    array[i] = x;
}
```



Rischi della gestione dinamica della memoria

Produzione di “garbage”:

- quando la memoria allocata dinamicamente resta logicamente inaccessibile, perché si sono persi i riferimenti:
- Esempio (P e Q sono puntatori):
 - **P=malloc(sizeof(TipoDato));**
 - **P=Q;**

Riferimenti “dangling”(fluttuanti):

- quando si creano riferimenti a zone di memoria logicamente inesistenti
- Esempio (P e Q sono puntatori):
 - **P=Q; free(Q);**
- l'istruzione **free** libera l'area di memoria ma non provoca un assegnamento automatico di **NULL** al puntatore Q, per cui P e Q si riferiscono perciò a celle di memoria non più esistenti



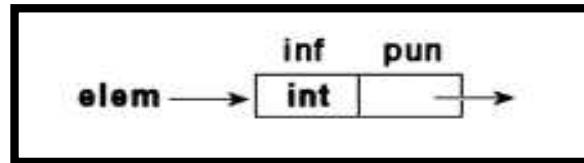
Liste puntate

Una lista è una collezione di elementi omogenei

A differenza dell'array, la dimensione di una lista non è nota a priori e può variare nel tempo. Inoltre un elemento nella lista occupa una posizione qualsiasi, che tra l'altro può cambiare dinamicamente durante l'utilizzo della lista stessa.

Ogni elemento nella lista ha uno o più campi contenenti informazioni, e, necessariamente, deve contenere un puntatore per mezzo del quale è legato all'elemento successivo

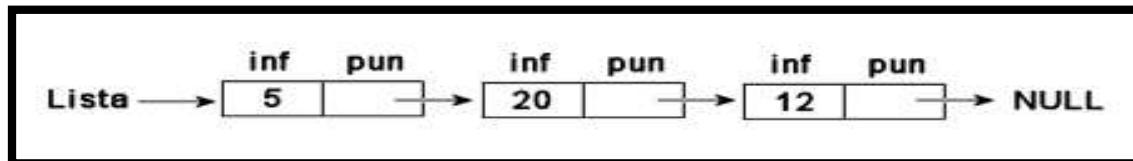
Esempio:



Liste puntate

Una lista puntata (semplice) “Lista” ha una gestione sequenziale, in cui è sempre possibile individuare la testa e la coda della lista.

Esempio:



Per definire un singolo elemento, usiamo la seguente sintassi

```
struct elemento
{
    int inf;
    struct elemento *next;
};
```

Lista può allora essere definita come un puntatore al primo elemento della lista, utilizzando il seguente codice

```
struct elemento *Lista;
```



Operazioni sulle liste

Le operazioni che agiscono su una lista rappresentano gli operatori elementari che agiscono sulle variabili di tipo lista (struct elemento)

Corrispondono a dei sottoprogrammi (funzioni)

Alcune operazioni modificano la lista

Operazioni tipiche:

- **Inizializzazione** - modifica la lista
- **Inserimento in testa** - modifica la lista
- **Inserimento in coda** - modifica la lista
- **Inserimento all'interno** - modifica la lista
- **Verifica lista vuota** - non modifica la lista
- **Ricerca elemento** - non modifica la lista
- **Stampa lista** - non modifica la lista



Verifica Lista vuota

Data una lista “Lista”, per verificare se essa è vuota è sufficiente controllare se essa punta a NULL.

La seguente verifica se Lista è vuota

```
void controlla_lista_vuota(struct elemento *Lista)
{
    return (Lista==NULL);
}
```



Inizializzazione

Consideriamo un semplice programma per l'inizializzazione di una lista di interi.

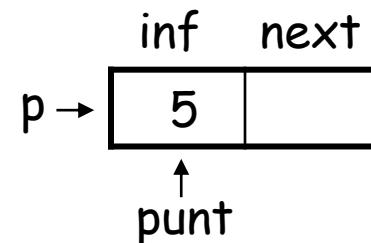
```
#include <stdio.h>
#include <malloc.h>
struct elemento {
    int inf;
    struct elemento *next;};
int main()
{
    struct elemento *lista; /*puntatore della lista */
    lista = crea_lista(); /* crea la lista */
    visualizza_lista(lista); /* stampa la lista */
}
```



Funzione crea_lista() 1/2

La funzione crea_lista() crea due puntatori ad elemento, uno di nome p (punta al primo elemento) e l'altro di nome punt (permette di scorrere la lista);

```
struct elemento *crea_lista()
{
    struct elemento *p, *punt;
    int i, n;
    printf("\n Specificare il numero di elementi... ");
    scanf("%d", &n);
    if(n==0)
        p = NULL;
    else {
        /* creazione primo elemento */
        p = (struct elemento *)malloc(sizeof(struct elemento));
        printf("\nInserisci il primo valore: ");
        scanf("%d", &p->inf);
        punt = p;
        .....
    }
}
```



Funzione crea_lista() 2/2

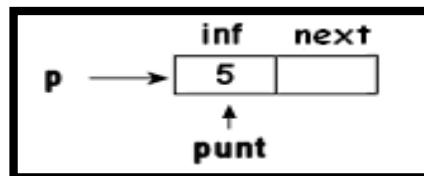
```
for(i=2; i<=n; i++)
{
    punt->next = (struct elemento *)malloc(sizeof(struct elemento));
    punt = punt->next;
    printf("\nInserisci il %d elemento: ", i);
    scanf("%d", &punt->inf);
} /* chiudo il for */
punt->next = NULL; /* marcatore fine lista */
} /* chiudo l'if-else */
return(p);
} /* chiudo la funzione */
```

Questo tipo di inserimento viene chiamata "inserimento in coda"

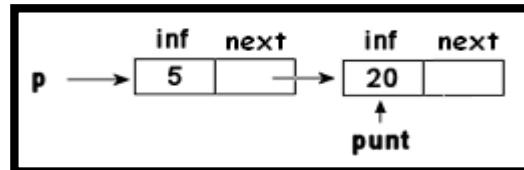


Esempio di funzionamento di crea_lista

- Assumiamo che si voglia creare una lista di 3 elementi (5,20,12); Alla prima iterazione abbiamo la seguente situazione:



- Supponiamo adesso di aver inserito i primi due elementi e stiamo per inserire il terzo. La lista avrà la seguente forma:



Esempio di funzionamento di crea_list

- A questo punto inserendo il valore 12 per prima cosa viene creato un altro oggetto della lista, identificato con punt -> next,
- poi "punt", il puntatore ausiliario, viene fatto puntare, non più al secondo elemento, bensì al terzo, all'atto pratico "punt" diventa il puntatore dell'oggetto da lui puntato (cioè, punt = punt -> next;).
- Quindi viene inserito il campo informazione dell'elemento tramite l'input da tastiera dell'utente; in questo caso viene inserito il valore 12;
- Alla fine, punt punta al valore NULL che identifica la fine della lista.



Funzione visualizza_list()

La seguente funzione iterativa permette di stampare tutti gli elementi interi presenti in una lista, nell'ordine in cui sono memorizzati

```
void visualizza_lista(struct elemento *p)
{
    printf("\n lista ---> ");
    while(p != NULL)
    {
        printf("%d", p->inf); /* visualizza l'informazione */
        printf(" ---> ");
        p = p->next; /* scorre la lista di un elemento */
    }
    printf("NULL\n\n");
}
```



Ricorsione su liste (1)

La ricorsione risulta particolarmente utile sulle liste collegate. Questo è dovuto al fatto che le liste si possono definire in modo ricorsivo:

Una lista è la lista vuota, oppure un elemento seguito da un'altra lista.

In altre parole, una variabile di tipo lista L può valere NULL (che rappresenta la lista vuota), oppure può essere un puntatore a una struttura che contiene un dato più un altro puntatore. Possiamo quindi dire che la struttura è composta da un elemento e da un puntatore, che rappresenta un'altra lista.



Ricorsione su liste (2)

La lista si ottiene guardando la struttura puntata e poi seguendo i puntatori fino a NULL.

Sia L una lista definita da

struct elemento {int inf; struct elemento *next;} L;

Una funzione ricorsiva su L avrà come argomento L, e al suo interno una chiamata ricorsiva a cui si passa $L \rightarrow next$.

Queste funzioni normalmente operano su $L \rightarrow inf$ (il primo elemento della lista), e poi agiscono sul resto della lista solo attraverso la chiamata ricorsiva.



Funzione visualizza_list() ricorsiva

```
void visualizza_lista(struct elemento *Lista)
{
    if(Lista==NULL) return;
    printf("%d ", Lista->inf);
    visualizza_lista(Lista->next);
}
```

Se Lista rappresenta la lista vuota, non si stampa niente; si esce semplicemente dalla funzione senza fare nulla.

Al passo i-esimo, si stampa la testa della lista e si richiama la funzione visualizza_list() sulla lista meno la testa.



Esercizio

Sia L una lista definita da

struct elemento {int inf; struct elemento *next;} L;

Scrivere in linguaggio C una funzione ricorsiva che preso in input L, raddoppi tutti gli elementi dispari della lista



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Esercitazione di laboratorio su
Liste Puntate Semplici

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 8

Parole chiave: **Liste dinamiche
singolarmente puntate**

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Centro di
Formazione
a Distanza



European Project



Università
degli Studi
di Napoli
Federico II

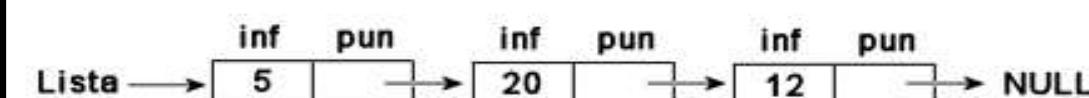


UNN

Realizzata con il coinvolgimento del Istituto Universitario Lattesano - Missa 3.12. Maturazione anno I - P.D.R. - Campagna 2007-2008

Liste puntate (remind)

Una lista puntata (semplice) ha una gestione sequenziale: gli elementi si possono scorrere in un unico verso. Inoltre, nella lista è sempre possibile individuare un primo elemento (testa) e un ultimo elemento (coda)



Operazioni sulle liste

**Nella precedente lezione abbiamo visto
l'implementazione delle seguenti operazioni su liste**

- **Inizializzazione**
- **Inserimento in coda**
- **Verifica lista vuota**
- **Stampa lista**

Oggi implementeremo inoltre le seguenti operazioni

- **Ricerca elemento**
- **Inserimento elemento**
- **Cancellazione di un elemento**

Esercizio I: Ricerca di un elemento

Sia L una lista definita da

struct lista {int val; struct lista *next;} L;

Scrivere in linguaggio C una funzione ricorsiva che preso in input L e un intero el, verifichi se esiste una occorrenza di el nella lista

Idea per la soluzione dell'esercizio:

- Scorrere in avanti la lista in chiamate ricorsive e
 1. Se L=NULL, ritornare 0
 2. se L->val=el, ritornare 1
 3. altrimenti, ritornare il risultato della chiamata ricorsiva su L->next

Esercizio II: Rimozione di un elemento

Sia L una lista definita da

struct lista {int val; struct lista *next;} L;

Scrivere in linguaggio C una funzione ricorsiva che preso in input L e un intero el, rimuova una ricorrenza di el dalla lista (se ne esiste una).

Idea per la soluzione dell'esercizio:

- Scorrere in avanti la lista in chiamate ricorsive e
 1. se L=NULL, ritornare L
 2. se L→val=el, settare L=L→next e ritornare L
 3. altrimenti, associare a L→next il risultato della chiamata ricorsiva su L→next e al ritorno dalla chiamata ricorsiva restituire L.

Esercizio III: Aggiunta di un elemento

Sia L una lista definita da

```
struct lista {int val; struct lista *next;} L;
```

Scrivere in linguaggio C una funzione ricorsiva che preso in input L, la modifichi in modo tale che ogni numero dispari sia seguito dal suo successore pari.

Idea per la soluzione dell'esercizio:

1. Dapprima scorrere la lista fino alla coda con chiamate ricorsive
2. Condizione di uscita: Se L=NULL, ritornare L
3. Al ritorno dalle chiamate ricorsive:
 1. Se L->val è dispari aggiungere un nuovo nodo tra L e L->next.
Attenzione qui a sistemare tutti i collegamenti!!!
 2. Ritornare la lista L.

Esercizio completo (da consegnare)

Si implementi in linguaggio C un menù di scelta multipla per la gestione “con funzioni ricorsive” di tutte le operazioni viste in precedenza. Dunque il menù permettere le seguenti operazioni:

- Creazione di una lista a puntatori (con il comando **struct**) e successivo riempimento della lista con n elementi dati in input
- Stampa della lista
- Ricerca di un elemento,
- inserimento (in coda o dopo un elemento specifico)
- rimozione della prima occorrenza di un elemento dato in input
- Ricerca e stampa del massimo e del minimo della lista

L'esercizio completo va consegnato via mail al tutor entro 5 giorni lavorativi

Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Implementazioni di Liste Doppiamente Puntate e Circolari

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 9

Parole chiave: **Liste dinamiche, liste circolari, liste doppiamente puntate**

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Università
degli Studi
di Napoli
Federico II



Università
degli Studi
di Napoli
Federico II



Università
degli Studi
di Napoli
Federico II



Università
degli Studi
di Napoli
Federico II



Università
degli Studi
di Napoli
Federico II

Indice

- **Liste puntate semplici:** Gli elementi sono organizzati in modo sequenziale e si possono scorrere in un unico verso. La lista ha un primo elemento (testa) e un ultimo elemento (coda)



- **Liste doppiamente puntate:** Sono simili alle liste puntate semplici, ma permettono di scorrere gli elementi in entrambi i versi



- **Liste puntate semplici circolari:** Sono liste puntate semplici senza testa ne coda.

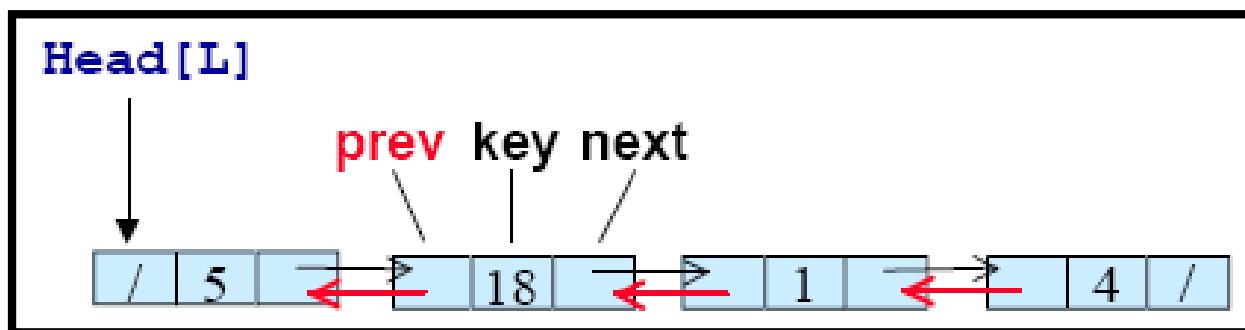


- **Liste doppiamente puntate circolari:** Liste doppiamente puntate senza testa ne coda.



Liste puntate doppie

Una Lista Doppia Puntata è un insieme dinamico in cui ogni elemento ha uno o più campi contenenti informazioni e due riferimenti, uno all'elemento successivo (next) della lista ed uno all'elemento precedente (prev) della lista.





Implementazione in C

- Per definire la struttura di un elemento di una lista doppiamente puntata bisogna utilizzare due puntatori alla stessa struttura. Il primo punterà all'elemento precedente mentre il secondo punterà a quello successivo:

```
struct el
{
    struct el *prev;
    int inf;
    struct el *next;
}
```

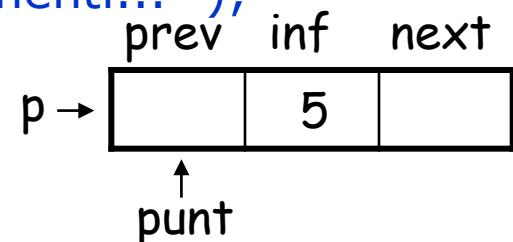
- Per l'inizializzazione di una lista doppiamente puntata si può considerare il codice visto per l'inizializzazione delle liste singolarmente puntate, introducendo opportunamente il codice per la gestione del puntatore all'elemento precedente.



Funzione crea_lista() 1/2

- La funzione **crea_lista()** crea due puntatori ad elemento, uno di nome **p** (puntatore al primo elemento della lista) e l'altro di nome **punt** (puntatore che permette di scorrere la lista);

```
struct el *crea_lista() {
    struct el *p, *punt;
    int i, n;
    printf("\n Specificare il numero di elementi... ");
    scanf("%d", &n);
    if(n==0)
        p = NULL;
    else {
        /* creazione primo elemento */
        p = (struct el *)malloc(sizeof(struct el));
        printf("\nInserisci il primo valore: ");
        scanf("%d", &p->inf);
        punt = p; p->prev=NULL;
```



Funzione crea_lista() 2/2

```
for(i=2; i<=n; i++)
{
    punt->next = (struct el *)malloc(sizeof(struct el));
    punt->next->prev=punt;
    punt = punt->next;
    printf("\nInserisci il %d elemento: ", i);
    scanf("%d", &punt->inf);
} // chiudo il for
punt->next = NULL; // marcatore fine lista
} // chiudo l'if-else
return(p);
} // chiudo la funzione
```



Inserimento in coda: main()

```
#include <stdio.h>
#include <malloc.h>
struct el {struct el *prev; int inf; struct el *next;};
struct el *crealista();
int inserisci_in_coda(struct el*,int);
int main() {
    struct el *lista;
    int valore;
    lista=crealista();  stampalista();
    printf("\nInserisci elemento da inserire: ");
    scanf("%d", &valore);
    inserisci_in_coda(lista,valore);
}
```



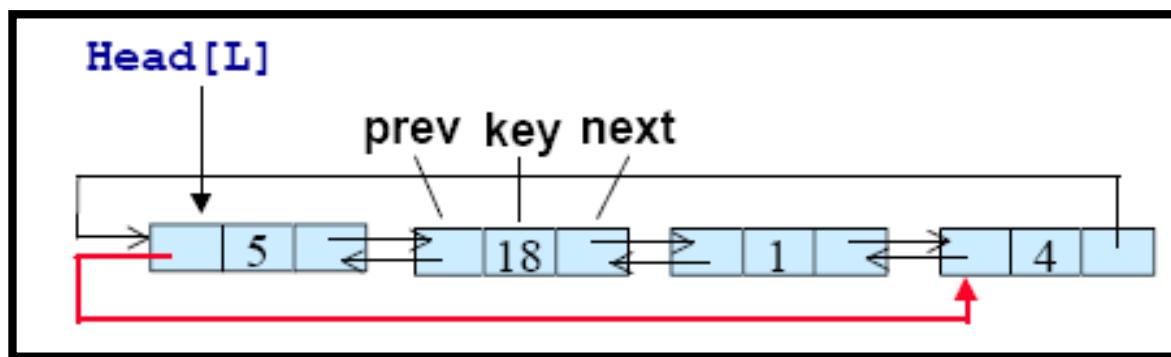
Inserimento in coda: funzione

```
struct el *inserisci(struct el *p, int valore)
{
    struct el *nuovo=NULL,
    struct el *testa;
    if (p==NULL) {
        p=(struct el *)malloc(sizeof(struct el));
        p->inf=valore; p->prev=NULL; testa=p; }
    else {
        testa=p;
        while (p->next!= NULL)
            p=p->next;
        nuovo=(struct el *)malloc(sizeof(struct el));
        nuovo->prev=p;
        nuovo->inf=valore;
        nuovo->next = NULL; }
    p->next=nuovo;
    return testa;
}
```



Liste puntate circolari

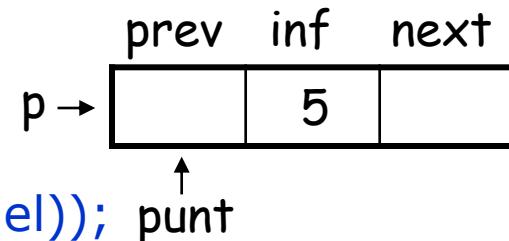
- Una Lista Circolare puntata è un una lista puntata in cui il puntatore next dell'ultimo elemento della lista punta all'elemento in testa alla lista.
- Infine, se la lista è doppiamente puntata, il puntatore prev della testa della lista punta all'elemento in coda alla lista



Funzione crea_lista() 1/2

- La funzione **crea_lista()** crea due puntatori ad elemento, uno di nome **p** (al primo elemento della lista) e l'altro di nome **punt** (che permette di scorrere la lista);

```
struct el *crea_lista() {  
    struct el *p, *punt;  
    int i, n;  
    printf("\n Specificare il numero di elementi... ");  
    scanf("%d", &n);  
    if(n==0) p = NULL;  
    else {  
        /* creazione primo elemento */  
        p = (struct el *)malloc(sizeof(struct el)); punt  
        printf("\nInserisci il primo valore: ");  
        scanf("%d", &p->inf);  
        punt = p; p->prev=NULL;  
    }  
}
```



Funzione crea_lista() 2/2

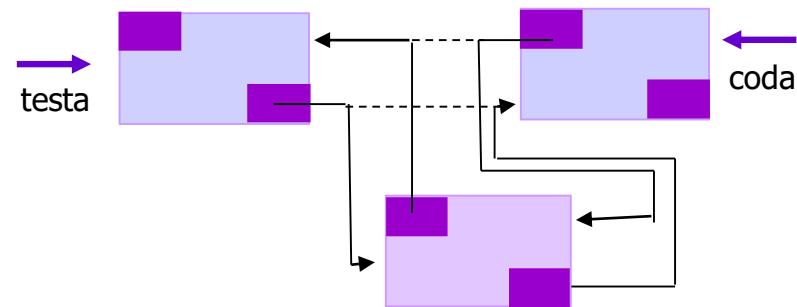
```
for(i=2; i<=n; i++)
{
    punt->next = (struct el *)malloc(sizeof(struct el));
    punt->next->prev=punt;
    punt = punt->next;
    printf("\nInserisci il %d elemento: ", i);
    scanf("%d", &punt->inf);
} // chiudo il for
→ punt->next = p; p->prev = punt; // circolarità della lista
} // chiudo l'if-else
return(p);
} // chiudo la funzione
```



Inserimento all'interno di liste doppiamente puntate

- Supponiamo di voler inserire un elemento **valore** in una lista prima dell'elemento puntato da pos

```
struct el *ins_in_testa(struct el *pos, int val)
{
    LISTA *p;
    p = (struct el *)malloc(sizeof(struct el));
    p->inf = val;
    /* aggiornamento dei puntatori */
    p->next = pos;
    p->prev=NULL;
    if !(pos==NULL) {
        p->prev = pos->prev;
        pos->prev=p;
        if !(p->prev== NULL)
            p->prev->next=p;
    }
    return (p);
}
```



Eliminazione elemento da una lista circolare doppiamente puntata

```

struct el *togli(struct el *p, int valore)
{
    struct el *inizio=NULL; int i=1;
    if (p!=NULL)
    {
        inizio=p;
        do
        {
            if (p->inf==valore)
            {
                if (p->next==p) inizio=NULL;
                else {
                    p->prev->next=p->next;
                    p->next->prev=p->prev;
                    if p==inizio inizio=p->next; }
                    free(p);
                    p=NULL;
                }
                else p=p->next;
            }
            while (p!=inizio && p!=NULL);
        }
        return inizio;
    }
}

```

Punterà alla testa della lista

Nel caso in cui il nodo rimosso è la testa della lista

Vero se il valore cercato è nella lista

Controlla che tutta la lista è stata letta



Esercizio 1

Siano L1 e L2 due liste non circolari doppiamente puntate definite da

```
struct el
{
    struct el *prev;
    int inf;
    struct el *next;
}L1, L2;
```

Si supponga che L1 e L2 siano ordinate in senso crescente.

Scrivere una funzione ricorsiva in linguaggio C che prese in input le due liste L1 e L2 generi una nuova lista L3 ordinata in senso crescente che contenga tutti gli elementi di L1 e L2



FEDERICA Federica

S_{M_{F_N}} Facoltà di Scienze
Matematiche
Fisiche Naturali

Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Esercitazione di laboratorio su
Liste Doppiaamente puntate

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 10
Parole chiave: [Liste dinamiche bidirezionali](#)

next

e-Learning

FEDERICA Federica

13/11/2008

S_{M_{F_N}} Facoltà di Scienze
Matematiche
Fisiche Naturali

Liste puntate doppie (remind)

Una Lista doppiaamente puntata è un insieme dinamico in cui ogni elemento ha uno o più campi contenenti informazioni e due riferimenti, uno all'elemento successivo (next) della lista ed uno all'elemento precedente (prev) della lista.

Head[L]

back next

FEDERICA Federica 13/11/2008 3 S_{M_N} Facoltà di Scienze Matematiche Fisiche Naturali

Esercizio

Si implementi in linguaggio C un menù di scelta multipla per la simulazione di un gioco che funzioni nel modo seguente:

1. Il gioco si svolge tra tre giocatori
2. Ogni giocatore ha una lista di 5 numeri con valore compreso tra 1 e 10
3. Esiste inoltre una lista "monte" di 5 numeri piu' un "numero speciale" di seguito spiegato
4. Ad ogni turno si gioca come segue:
 1. Si confrontano i valori in testa alle liste dei giocatori.
 2. Vince il giocatore che ha il numero maggiore
 3. Perde il giocatore che il numero minore
 4. Si pesca un num dalla testa del monte e si aggiunge in fondo alla lista perdente
 5. Si sposta il num del giocatore vincente nel monte in posizione random.
 5. Vince il giocatore che svuota per primo la lista
 6. Se il giocatore i pesca il num speciale, i rimuove dai suoi numeri il minimo (1 valore) che si inserisce nel monte. Il num speciale puo' essere usato una sola volta e non va reinserito nel monte
 7. Casi speciali: vincono/perdono contemporaneamente piu' giocatori: si sorteggia chi vince/perde il turno.

back next

FEDERICA Federica 13/11/2008 4 S_{M_N} Facoltà di Scienze Matematiche Fisiche Naturali

Esercizio

Si implementi un menu' che implementi

1. Il caricamento delle liste dei tre giocatori piu' la lista mone in modo casuale.
2. Permette di evidenziare la situazione iniziale, e descriva cio' che succede ad ogni turno
3. Mostri il giocatore vincente

back next

The screenshot shows a software window titled "Esercizio". At the top left is the "Federica" logo. In the center top is the date "13/11/2008". At the top right is the logo for "Facoltà di Scienze Matematiche Fisiche Naturali" with the letter "S" and "MNF". Below the title, there is a large empty rectangular area. In the center of this area is a red-bordered box containing the text: "L'esercizio completo va consegnato via mail al tutor e al docente entro lunedì sera". At the bottom of the window are three buttons: "back" on the left, a central button with a red "X", and "next" on the right.

This document was created with Win2PDF available at <http://www.win2pdf.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.
This page will not be added after purchasing Win2PDF.

Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Alberi Binari di Ricerca

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 11

Parole chiave: **Alberi Binari, Ricerca Binaria,
Visite di Alberi**

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Realizzata con il coinvolgimento del Istituto Universitario Lattesca - Mappa 3.0.0 Mbaunivm anno I - P.D.R. - Campagna 2000-2006



Centro di Ricerca e Sviluppo



Progetto



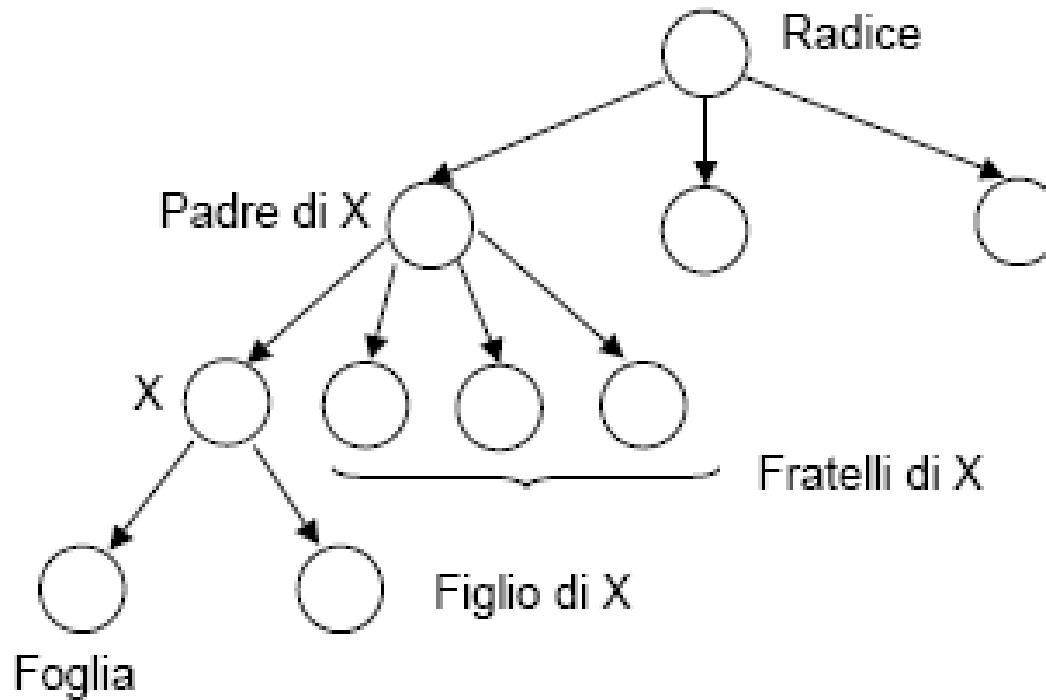
Progetto



Progetto

Alberi

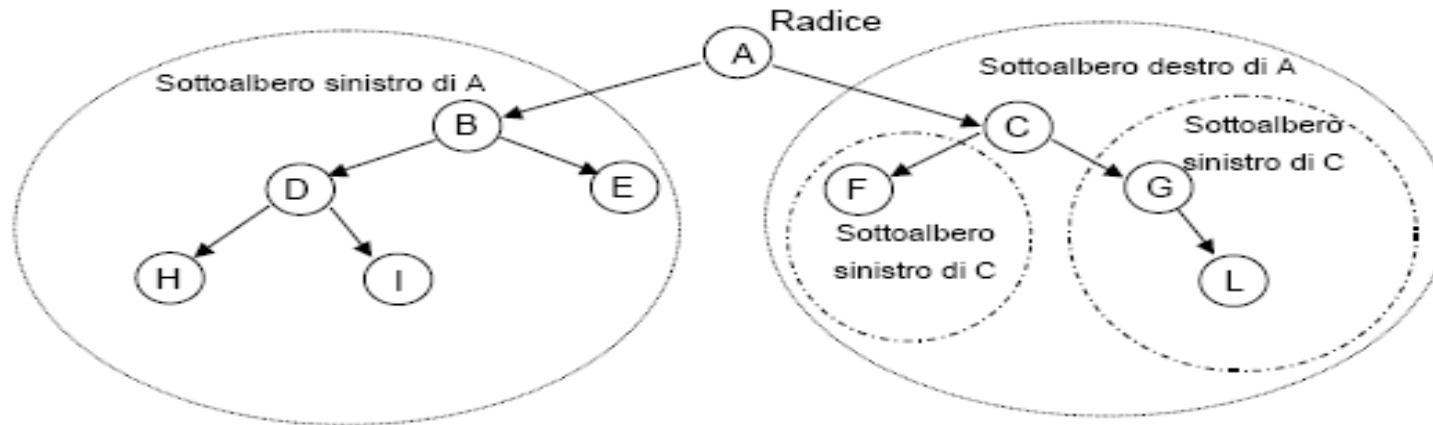
L'albero è un tipo astratto di dato utilizzato per rappresentare relazioni gerarchiche tra oggetti.



Alberi binari

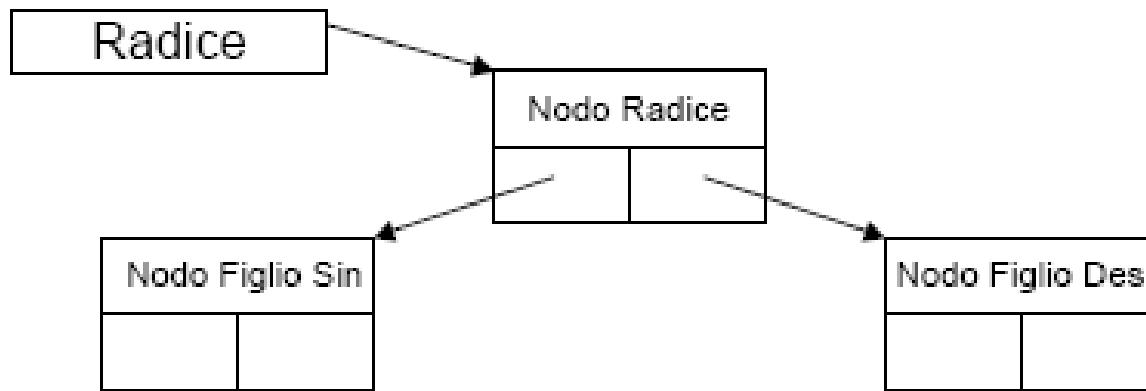
Un albero binario può essere facilmente rappresentato in modo ricorsivo. Infatti, un albero

- è un oggetto vuoto (cioè è un insieme vuoto di nodi); oppure
- è formato da un nodo A (chiamato radice) e da due sottoalberi, a loro volta alberi binari, chiamati rispettivamente sottoalbero sinistro e sottoalbero destro.



Rappresentazione di un albero binario

Per rappresentare un albero binario si può usare la seguente struttura ricorsiva:



```
struct nodo { int inforadice;
                struct nodo *sinistro, *destro;
            };
struct nodo *radice;
```

Primitive sugli alberi binari

- Per controllare se un nodo è vuoto possiamo usare la seguente funzione:

```
int vuoto (struct nodo *rad)
{
    if(rad) return 0;
    else return 1;
}
```

- Per sapere il valore di un nodo possiamo usare la seguente funzione che ritorna 0 se l'albero è vuoto, altrimenti memorizza nella variabile **val** il valore del nodo

```
int radice(struct nodo *rad, int *val)
{
    int ok=0;
    if !(vuoto(rad))
    {
        *val=rad->inforadice;
        ok=1;
    }
    return ok;
}
```

Altre Primitive

Per avere il punt. al figlio sinistro (destro) di un nodo:

```
struct nodo *sinistro (struct nodo *rad)
{
    struct nodo *risultato=NULL;
    if !(vuoto(rad)) risultato=rad->sinistro;
    return NULL;
}
```

Per costruire un nodo (o un albero a partire da due sottoalberi):

```
struct nodo * costruisci(struct nodo *s, int r, struct nodo *d)
{
    struct nodo *aux;
    aux=(struct nodo*)malloc(sizeof(struct nodo));
    if (aux) {
        aux->inforadice=r;
        aux->sinistro=s; aux->destro=d; }
    return aux;
}
```



Visita di un albero binario

Oltre alle operazioni primitive, si definiscono delle operazioni di visita ovvero di analisi dei nodi di un albero in determinato ordine.

Di seguito analizziamo le seguenti visite di un albero:

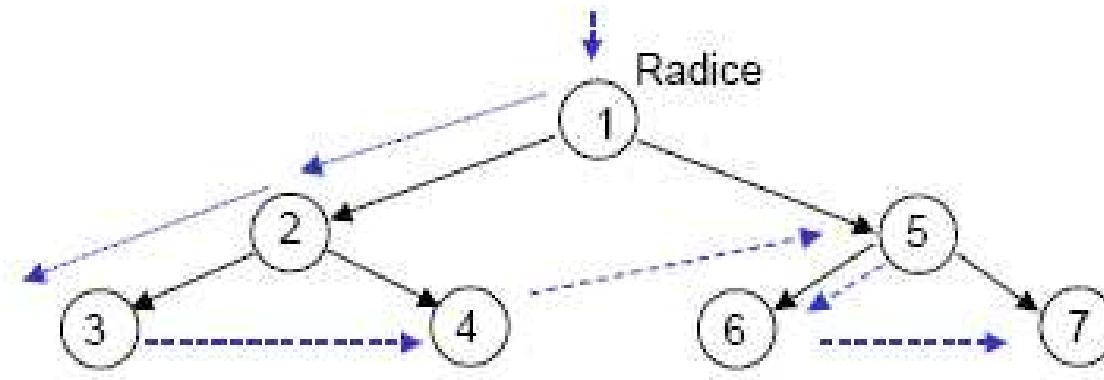
- Visita in Preordine
- Visita in Ordine
- Visita in Postordine



Visita in Preordine

Nella visita in Preordine, se l'albero non è vuoto:

- Si analizza la radice dell'albero;
- Si visita in preordine il sottoalbero sinistro;
- Si visita in preordine il sottoalbero destro.

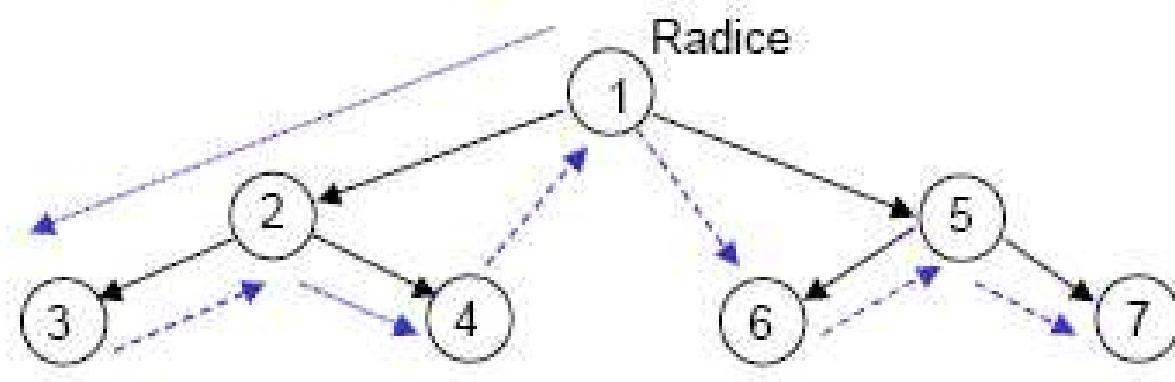


Nella visita in preordine del precedente albero i nodi verrebbero visitati nel seguente ordine: 1, 2, 3, 4, 5, 6, 7

Visita in Ordine

Nella visita in ordine, se l'albero non è vuoto:

- Si visita in ordine il sottoalbero sinistro;
- Si analizza la radice dell'albero;
- Si visita in ordine il sottoalbero destro.

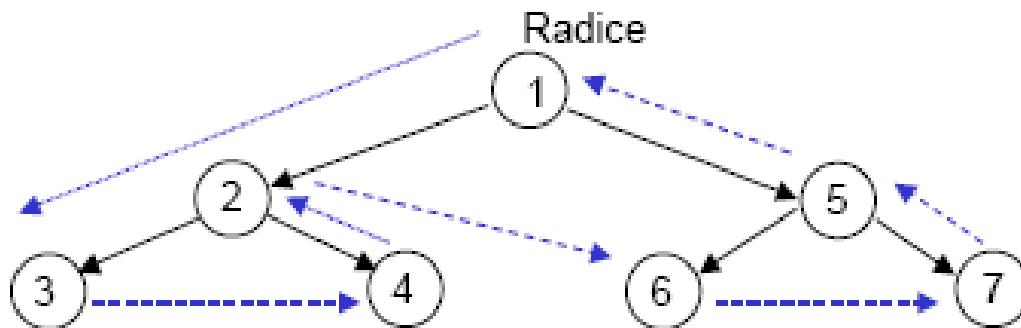


Nella visita in ordine del precedente albero i nodi verrebbero visitati nel seguente ordine: 3, 2, 4, 1, 6, 5, 7

Visita Postordine

Nella visita in postordine, se l'albero non è vuoto:

- Si visita in postordine il sottoalbero sinistro;
- Si visita in postordine il sottoalbero destro;
- Si analizza la radice dell'albero.



Nella visita in ordine del precedente albero i nodi verrebbero visitati nel seguente ordine: 3, 4, 2, 6, 7, 5, 1

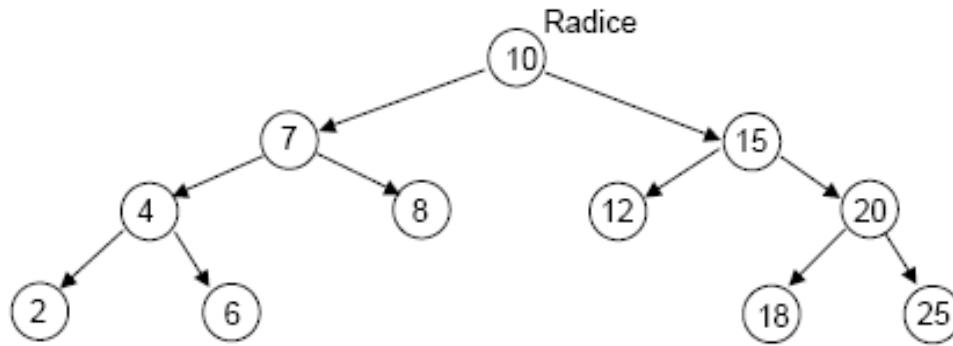
Codice per la visita di un albero

```
void visita_in_preordine(struct nodo *radice) {  
    if(radice) {    printf("%d ",radice->inforadice);  
                    visita_in_preordine(radice->sinistro);  
                    visita_in_preordine(radice->destro); } }  
  
void visita_in_ordine(struct nodo *radice){  
    if(radice) {    visita_in_ordine(radice->sinistro);  
                    printf("%d ",radice->inforadice);  
                    visita_in_ordine(radice->destro); } }  
  
void visita_in_postordine(struct nodo *radice) {  
    if(radice) {    visita_in_postordine(radice->sinistro);  
                    visita_in_postordine(radice->destro);  
                    printf("%d ",radice->inforadice); } }
```



Alberi binari di ricerca (ABR)

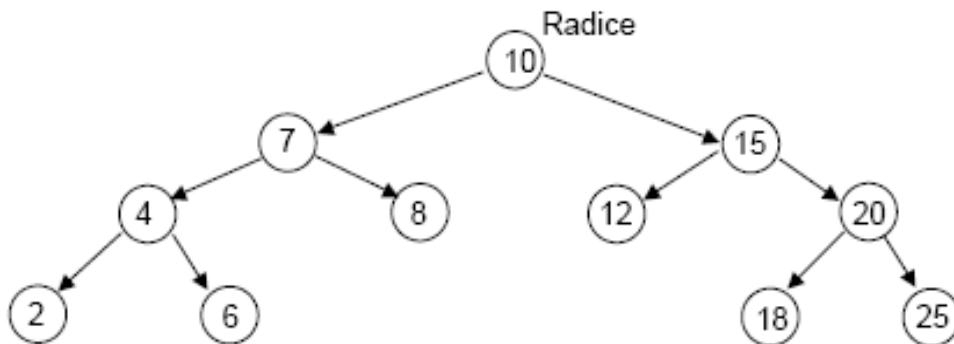
- Un albero binario di ricerca (ABR) è un albero binario in cui per ogni nodo dell'albero N tutti i nodi del sottoalbero sinistro di N hanno un valore minore o uguale di quello di N e tutti i nodi del sottoalbero destro hanno un valore maggiore di quello del nodo N.



- Il vantaggio principale di tale organizzazione è nella **ricerca**.
- Ogni volta che bisogna ricercare un elemento, il confronto del valore di un nodo dell'albero permette di eliminare dalla fase di ricerca o il sottoalbero corrente di destra o quello di sinistra.

Osservazione

Dato un ABR, la visita in “ordine” restituisce una lista ordinata crescente dei valori contenuti nell’albero



Nella visita in ordine del precedente albero, i nodi sono infatti visitati nell’ordine: 2,4,6,7,8,10,12,15,18,20,25



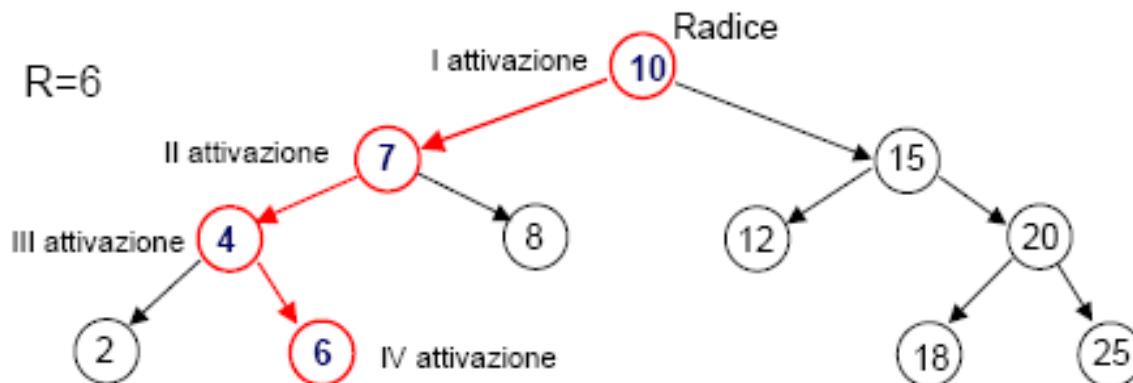
Esercizio

Scrivere una funzione in linguaggio C che preso in input un albero binario con n elementi valuti in tempo O(n) se l'albero è un ABR.



Ricerca in un albero binario di ricerca

- Per trovare un numero R si procede nel seguente modo:
 - Se l'albero è vuoto l'elemento non è presente;
 - Se la radice dell'albero == R l'elemento è stato trovato;
 - Se la radice dell'albero > R la ricerca viene condotta nel sottoalbero sinistro;
 - Altrimenti la ricerca viene condotta nel sottoalbero destro;



- La ricerca può essere realizzata mediante una funzione ricorsiva che nei casi 3 e 4 invoca se stessa.

Ricerca in un albero binario di ricerca

Versione iterativa della ricerca

```
int ricerca (struct nodo *radice, int r)
{
    int trovato=0;
    while(radice && trovato==0)
    {
        if(radice->inforadice==r)
            trovato=1; /* Trovato */
        else if(radice->inforadice > r)
            /* Cerca nel sottoalbero sinistro */
            radice=radice->sinistro;
        else /* Cerca nel sottoalbero destro */
            radice=radice->destro;
    }
    return trovato;
}
```



Ricerca in un albero binario di ricerca

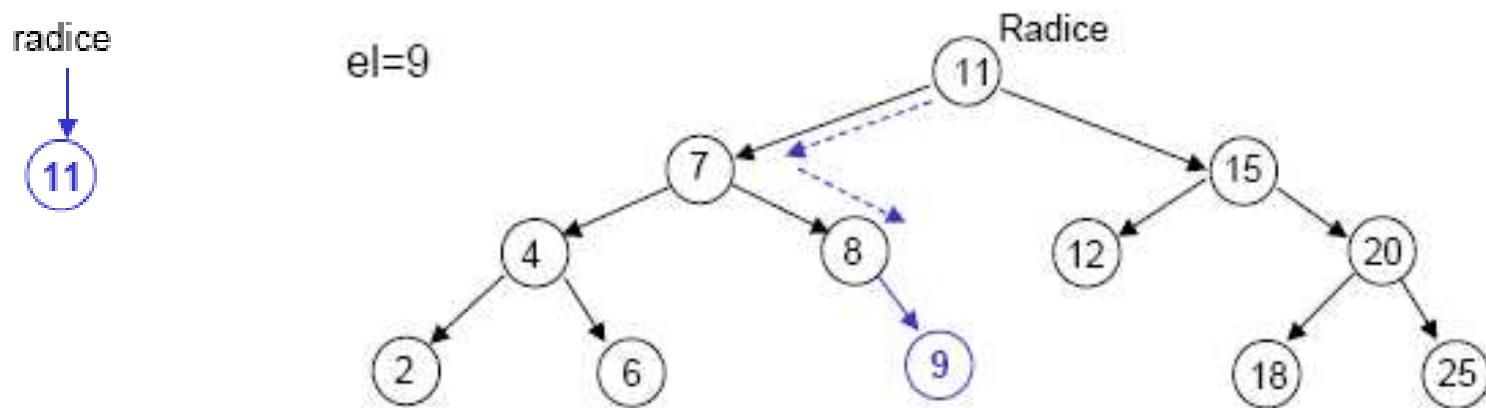
Versione ricorsiva della ricerca

```
int ricerca (struct nodo *radice, int r)
{
    int trovato=0;
    if !(vuoto (radice)) /*else non trovato poiché ABR vuoto */
    {
        if(radice->inforadice==r) return 1; /* Trovato */
        else if(radice->inforadice > r) /* Cerca nel sottoalbero sx */
            trovato=ricerca(radice->sinistro,r);
        else /* Cerca nel sottoalbero destro */
            trovato=ricerca(radice->destro,r);
    }
    return trovato;
}
```



Inserimento di un nuovo nodo in un ABR

- Se l'albero è vuoto, viene creato un nuovo nodo;
- Se l'elemento è minore o uguale alla radice dell'albero, l'inserimento va fatto nel sottoalbero sinistro;
- Se l'elemento è maggiore o uguale alla radice dell'albero, l'inserimento va fatto nel sottoalbero destro;



Inserimento di un nuovo nodo in un ABR

```
struct nodo *inserisci (struct nodo *radice, int e)
{struct nodo *aux;
if (vuoto(radice)) /* Creazione di un nuovo nodo */
{
    aux=(struct nodo*)malloc(sizeof(struct nodo));
    if(aux)
    {
        aux->info=e;
        aux->sx=aux->dx=NULL;
        radice=aux;
    }
    else printf("Memoria non allocata");
}
else if(e<radice->info) radice->sx = inserisci(radice->sx, e);
else if(e>radice->info) radice->dx = inserisci(radice->dx, e);
/* altrimenti il valore è già nell'ABR e non si fa niente */
return radice;
}
```

struttura

| | |
|------|----|
| info | |
| sx | dx |



Inserimento di un nuovo nodo tramite l'uso di puntatori a puntatori

```
void inserisci (struct nodo **radice, int e)
{ struct nodo *aux;
if(*radice==NULL)
{ /* Creazione di un nuovo nodo */
    aux=(struct nodo*)malloc(sizeof(struct nodo));
    if(aux)
    {
        aux->info=e;
        aux->sx=aux->dx=NULL;
        *radice=aux;
    }
    else printf("Memoria non allocata");
}
else if((*radice)->info>e) inserisci(&(*radice)->sx,e);
else if((*radice)->info<e) inserisci(&(*radice)->dx,e);
}
```

struttura

| | |
|------|----|
| info | |
| sx | dx |



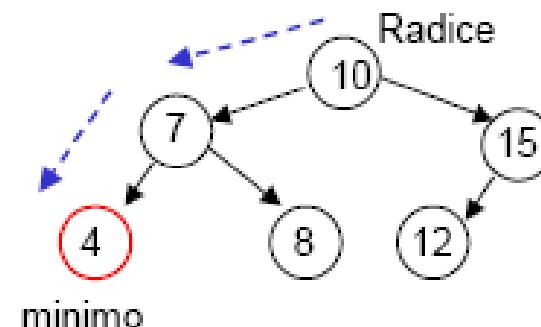
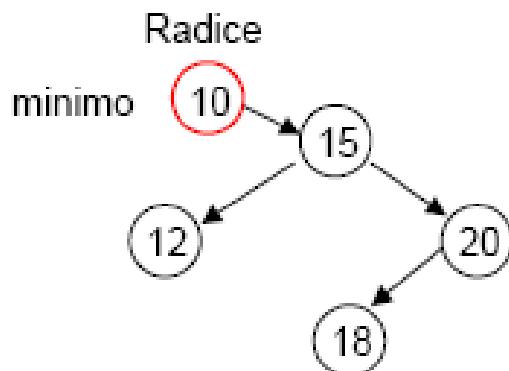
Osservazioni sulla slide precedente

- L'invocazione della funzione **inserisci** dipende da come è stato definito l'ABR nella funzione chiamante(che può anche essere main). Di seguito mostriamo due possibili casi:
 - L'ABR è definito con singolo puntatore:
struct nodo *radice =NULL;
allora la funzione inserisci sarà invocata con
inserisci(&(radice),valore);
 - L'ABR è definito con doppio puntatore:
struct nodo **radice;
radice=(struct nodo)malloc(sizeof(struct nodo));**
***radice=NULL;**
allora saraprima la funzione inserisci sarà invocata con
inserisci(radice,valore);



Ricerca minimo in un ABR

- Se il sottoalbero sinistro è vuoto, il minimo è la radice.
- Altrimenti il minimo è da cercare nel sottoalbero sinistro



```

int ricerca_minimo (struct nodo *radice)
{ /* per semplicità assumiamo tutti i valori dell'ABR positivi*/
  int min=0;
  if !(vuoto(radice)) {
    if(radice->sx==NULL) minimo=radice->info;
    else min= ricerca_minimo(radice->sx);}
  return min;
}
  
```



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Esercitazione di laboratorio su
Alberi Binari di Ricerca

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 12

Parole chiave: **Alberi Binari, Ricerca Binaria,
Visite di Alberi**

next



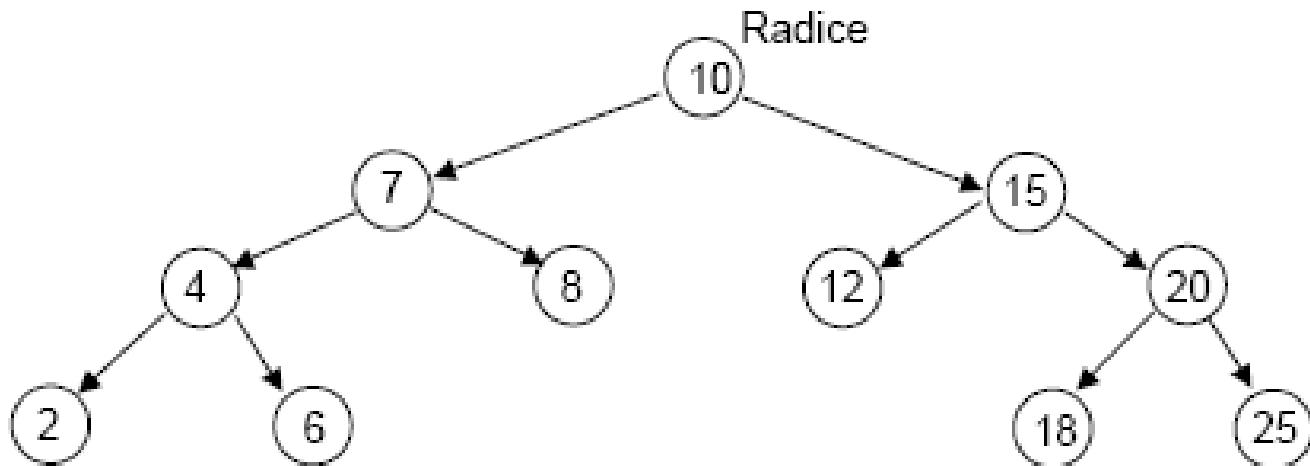
Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Realizzata con il coinvolgimento del Istituto Universitario Lattesano - Missa 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006

Alberi binari di ricerca (ABR)

- Si ricordi che un albero binario di ricerca (ABR) è un albero binario in cui per ogni nodo dell'albero tutti i nodi del suo sottoalbero sinistro hanno un valore minore (o uguale) mentre tutti quelli del suo sottoalbero destro hanno un valore maggiore.



Esercizio

Realizzare un menù a scelta multipla che permetta:

1. Creare un ABR di interi (senza valori uguali e tutti positivi);
2. Aggiungere un nodo ad un ABR;
3. Cercare un valore nell'ABR;
4. Cercare il minimo nell'ABR;
5. Stampare gli elementi dell'ABR con una visita in ordine.



Facoltativo

Aggiungere al menù precedente una scelta che, data una lista di interi non circolare e singolarmente puntata permetta,

6.1 di rimuovere dalla lista i numeri dispari e inserirli opportunamente nell'ABR

6.2 di stampare la lista e l'albero modificati.



Complessità

Si valutino le complessità delle funzioni precedenti.



Consegna

**Non è prevista la consegna di questa esercitazione.
Chi volesse, può comunque discutere la soluzione
dell'esercitazione con il docente o il tutor sia durante
le ore di laboratorio che durante l'orario di
ricevimento.**



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Alberi Binari di Ricerca
Cancellazione di un nodo

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 13

Parole chiave: **Alberi Binari, Ricerca Binaria,
Visite di Alberi**

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Realizzata con il coinvolgimento del Istituto Universitario Lattesca - Mappa 3.0.0 Maturazione anno I - P.D.R. - Campagna 2000-2006



Progetto
e-Scienze



Progetto
e-Scienze



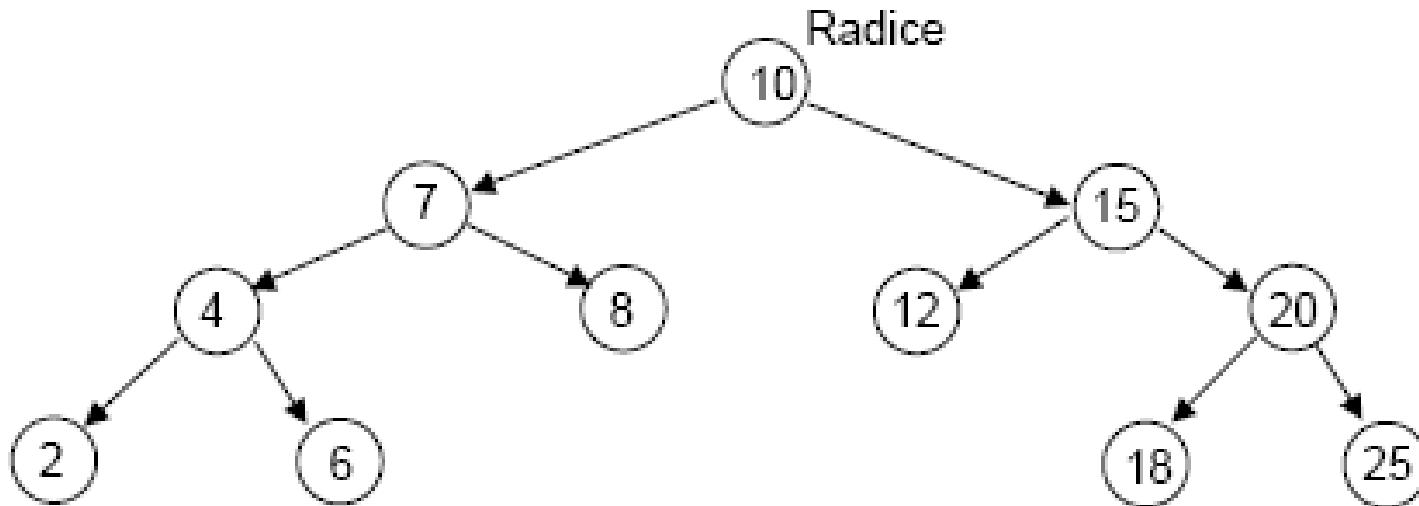
Progetto
e-Scienze



Progetto
e-Scienze

Alberi binari di ricerca (ABR)

- Un albero binario di ricerca (ABR) è un albero binario in cui per ogni nodo dell'albero N tutti i nodi del sottoalbero sinistro di N hanno un valore minore o uguale di quello di N e tutti i nodi del sottoalbero destro hanno un valore maggiore di quello del nodo N.



Riepilogo della precedente lezione in aula

Nella lezione precedente abbiamo studiato come:

- Definire un ABR tramite struct
- Creare un ABR con un unico nodo o come unione tramite un nuovo nodo radice di due ABR preesistenti;
- Controllare se un ABR è vuoto;
- Controllare che un albero è un ABR;
- Ottenere il valore della radice di un albero;
- Avere il puntatore al figlio sx/dx di un ABR.
- Stampare il contenuto di un albero tramite una visita
- Ricercare un dato elemento o il minimo di un ABR



Sommario

In questa lezione valuteremo come cancellare un nodo da un ABR.



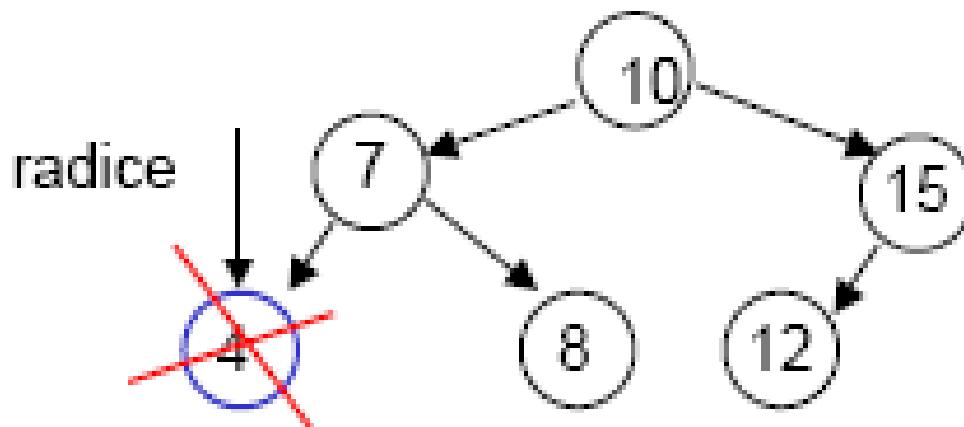
Cancellazione

Nella cancellazione di un elemento el da un albero bisogna distinguere i seguenti casi:

1. Albero vuoto: non viene realizzata alcuna cancellazione;
2. L'elemento el < radice albero: la cancellazione va effettuata nel sottoalbero sinistro;
`elimina(radice->sinistro,el);`
3. L'elemento el > radice albero: la cancellazione va effettuata nel sottoalbero destro;
`elimina(radice->destro,el);`
4. L'elemento el==radice albero. Si considerano i seguenti casi:
 1. La radice è una foglia:
 2. Il nodo ha un sottoalbero non vuoto e l'altro vuoto.
 3. Il nodo ha entrambi i sottoalberi non vuoti.

Primo caso

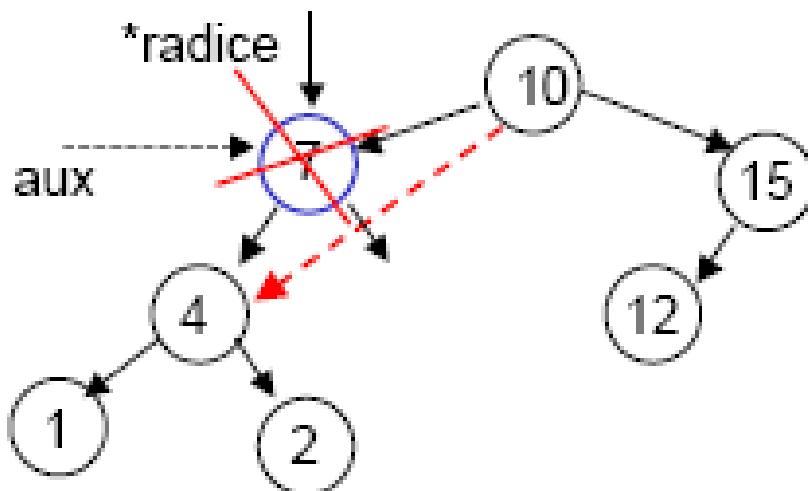
La radice è una foglia: viene eliminata liberando la memoria del nodo radice



Secondo Caso

Se il sottoalbero destro e' vuoto:

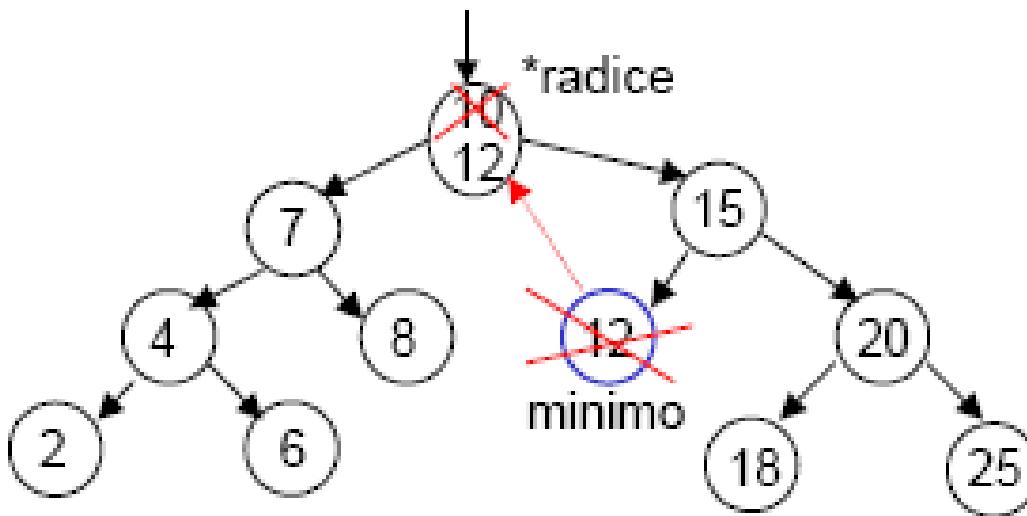
- il nodo figlio del nodo padre (10) di el (7) diventa il nodo figlio di el (4);
- Viene eliminato il nodo el (7).



Terzo Caso

Il sottoalbero destro e sinistro del nodo el sono non vuoti:

- viene ricavato il minimo (12) del sottoalbero destro;
- il minimo (12) viene copiato nel nodo el (10);
- viene eliminato il minimo (12) del sottoalbero destro.



Codice per la cancellazione (NO)

```
void *elimina (struct nodo *radice, int el)
{ struct nodo *aux;
  if(vuoto(radice)) { printf("elemento non trovato"); return; }
  if(radice->inforadice > el) /* el va cercato nel sottoalbero sinistro */
    elimina(radice->sinistro,el);
  else if(radice->inforadice< el) /* el va cercato nel sottoalb. destro */
    elimina(radice->destro,el);
  else /* Trovato l'elemento da cancellare */
    { if(radice->sinistro && radice->destro) /* Il nodo ha due figli*/
        {
          aux=ricerca_minimo(radice->destro);
          radice->inforadice=aux->inforadice;
          elimina(radice->destro,aux->inforadice); }
      else { /* Il nodo ha 0 oppure un figlio*/
          aux=radice;
          if (radice->sinistro = NULL) radice=radice->destro;
          else if (radice->destro = NULL) radice=radice->sinistro;
          free(aux); }
    }
  return;
}
```

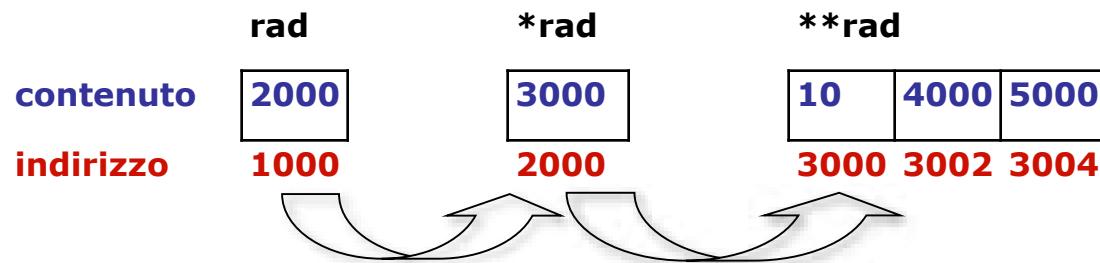
Osservazioni

- Il codice precedente, non cancella efficacemente un nodo perchè all'atto della cancellazione "free(radice); radice=NULL;" deallochiamo effettivamente la memoria per il nodo da cancellare, ma il nodo padre di quest'ultimo continuerà a puntare ad una locazione di memoria reale, che non è NULL.
- La funzione è scorretta perchè porta troppo in avanti il puntatore da non permette la modifica il campo dx o sx del nodo padre.
- Una soluzione è quella di utilizzare una funzione che ritorni un puntatore a una struttura ad albero (come proposto per le liste). Il valore ritornato, opportunamente gestito nelle chiamate ricorsive, permetterà di modificare l'ABR nel modo voluto.
- Un'altra soluzione è quella di utilizzare il metodo del puntatore a puntatore. Questo è da preferirsi quando si ha a che fare con funzioni che devono modificare più oggetti.
- Di seguito mostriamo la seconda soluzione.

Come usare il puntatore a puntatore

- Un ABR con doppio puntatore sarà definito nel seguente modo:


```
struct nodo {int info; struct nodo *sx; struct nodo *dx; }
struct nodo **rad;
rad=(struct nodo**)malloc(sizeof(struct nodo));
```
- L'ABR sarà referenziato con una cella che contiene un indirizzo (rad) di una cella contenente un indirizzo (*rad) di una cella in cui è memorizzato la radice dell'albero. Si veda per es. la figura sotto



- Una chiamata ad una funzione “**elimina(**rad, el)**”, con **el** intero da eliminare, passa alla funzione il riferimento alla cella con indirizzo 2000. Una modifica alla radice dell'ABR, modificherà anche il contenuto di questa cella.

Uso di puntatori a puntatori

```

void elimina(struct nodo **rad, int el)
{ struct nodo *aux;
aux= *rad;
if (!(*rad)) {
    if ((*rad)->info>el)
        /* cerca el a sx */
        elimina(&((*rad)->sx), el);
    else if ((*rad)->info<el)
        /* cerca el a dx */
        elimina(&((*rad)->dx), el);
    else { /* Elemento trovato */
        if (!((*rad)->sx) && !((*rad)->dx))
            { /* el e' una foglia */
                free(*rad);
                *rad=NULL;
            }
    }
}

```

****rad**

| | | |
|------------------------------|------------------------------|---------------------------|
| 10 (*rad)->info | 4000 (*rad)->sx | 5000 (*rad)->dx |
| 3000 &((*rad)->sx) | 3002 &((*rad)->dx) | 3004 |

In riferimento all'esempio di struttura precedente

Uso di puntatori a puntatori

```
/*continua else "Elemento trovato" */
if ((((*rad)->sx) && (!((*rad)->dx))) /* Se dx è vuoto*/
    *rad=aux->sx;
if ((((*rad)->sx) && (!((*rad)->dx))) /* Se sx è vuoto*/
    *rad=aux->dx;
if ((aux->destro==NULL) || (aux->sinistro==NULL))
{
    /* Se un sottoabbr è vuoto */
    free(aux);
    return;
}
if ((((*rad)->sx) && ((*rad)->dx)) /* Se sx e dx non vuoti*/
{
    (*rad)->info=ricerca_minimo((*rad)->dx);
    elimina(&(*rad)->destro, (*rad)->info);
}
```

}



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Esercizio di Laboratorio Gioco su alberi

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 13

Parole chiave: **Alberi Binari, Ricerca Binaria,
Visite di Alberi**

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Realizzata con il coinvolgimento del Istituto Universitario Lattesca - Mappa 3.0.0 Maturazione anno I - P.D.R. - Campagna 2000-2006



GIOCO TOM & JERRY

- Si implementi un gioco tra due giocatori (Tom e Jerry) realizzato su alberi binari di ricerca di interi nel modo seguente:
- Tom e Jerry hanno a disposizione un albero binario di ricerca di interi a testa.
- I due muovo a turni. Inizia Jerry a muovere.
- Scopo del gioco per TOM: acciuffare JERRY. Il che si concretizza per TOM nel trovarsi in un nodo il cui valore è identico a quello in cui si trova Jerry.

Strategia del Gioco

- Ogni giocatore può soltanto muovere da padre a figlio e non viceversa.
- Se un giocatore raggiunge una cella che ha un solo nodo figlio può soltanto muovere verso quel nodo figlio.
- Se un giocatore raggiunge un nodo che ha due figli allora si comporta come segue:
 - Se Jerry si trova in un nodo con valore “maggiore” a quello di Tom, Jerry cerca di scappare da Tom muovendo a destra. Se il nodo ha invece valore minore di quello di Tom, allora Jerry muove a sinistra
 - Tom farà l’opposto cercando di avvicinarsi a Jerry. In pratica, se Tom si trova in un nodo con valore “maggiore” a quello di Jerry, Tom muove a sinistra. Se invece è “minore”, allora muove a destra.
- Se un giocatore raggiunge una foglia non può più muovere.

Condizioni di terminazione

- Se Tom vince (entrambi i giocatori sono sullo stesso numero), si provvede a rimuovere il nodo dall'albero di Jerry e il gioco termina.
- Se entrambi i giocatori raggiungono una foglia e Tom non ha vinto, si trasferisce la foglia raggiunta da Jerry nell'albero di Tom e si riparte con il gioco.
- Quest'ultima operazione è ammessa per al più un numero prefissato di volte.
- Si consideri per semplicità che tutti i nodi dell'albero abbiano valore differente. Per cui, l'operazione di inserimento nell'albero di Tom non ha "effetto" se il nodo è già presente.
-

IMPLEMENTAZIONE

Scrivere in linguaggio C un menù a scelta multipla che permetta le seguenti funzioni:

- Riempimento (casuale o manuale) dei due alberi Tom e Jerry. Per semplicità si definiscano due costanti RANGE e TOT, e di riempire l'albero con TOT nodi presi nel range 1... RANGE
- Simulazione del gioco. Tale funzione prende in input i due alberi e restituisce i due alberi modificati al termine del gioco e riporta quanti turni sono stati giocati (si supponga che al più possano essere giocati NUM turni)
- Stampa in ordine del contenuto degli alberi prima e dopo il gioco.

Si discuta infine la complessità di tutte le funzioni implementate



Consegna

**L'esercizio completo va consegnato via mail
al tutor entro 4 giorni lavorativi.**



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Grafi: Implementazione ed operazioni di base

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Laboratorio di Algoritmi e Strutture Dati
13917
murano@na.infn.it
2007/2008

Lezione numero: 15

Parole chiave: Grafi, definizione e rappresentazioni

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Realizzata con il coinvolgimento del Istituto Universitario Lattesano - Missa 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006

Sommario delle lezioni sui grafi

I grafi sono un potente strumento per la rappresentazione di problemi complessi

La soluzione di moltissimi problemi può essere ricondotta alla soluzione di opportuni problemi su grafi.

Nel contesto dei grafi saranno approfonditi i seguenti argomenti:

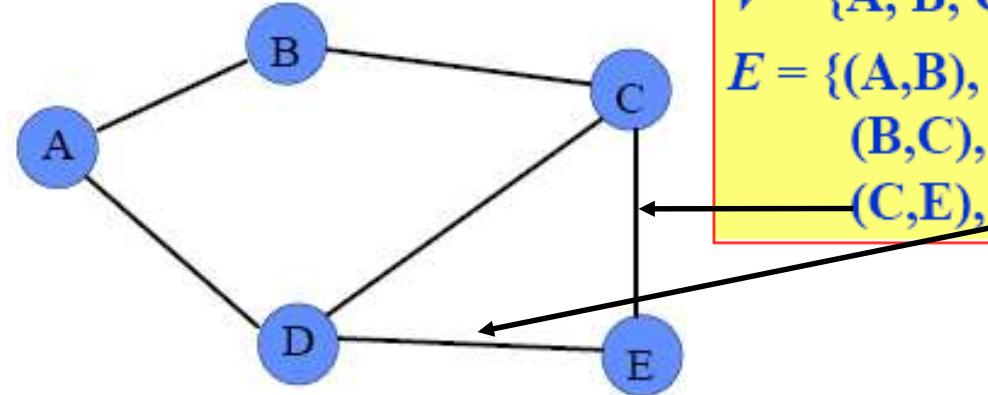
- Definizioni e rappresentazione di grafi:
 - Lista di adiacenza
 - Matrice di adiacenza
- Algoritmi di base su grafi
 - Ricerca in ampiezza (BFS)
 - Ricerca in profondità (DFS)
- Algoritmi avanzati sui grafi
 - Albero minimo di copertura (Minimum Spanning Tree)
 - Percorso minimo tra due vertici

Definizione di Grafo

Un grafo è una coppia (V, E) , dove

- V è un insieme di nodi, chiamati vertici
- E è un insieme di coppie di nodi, chiamati archi
- Un **arco** è una coppia (v,w) di vertici in V

Esempio:

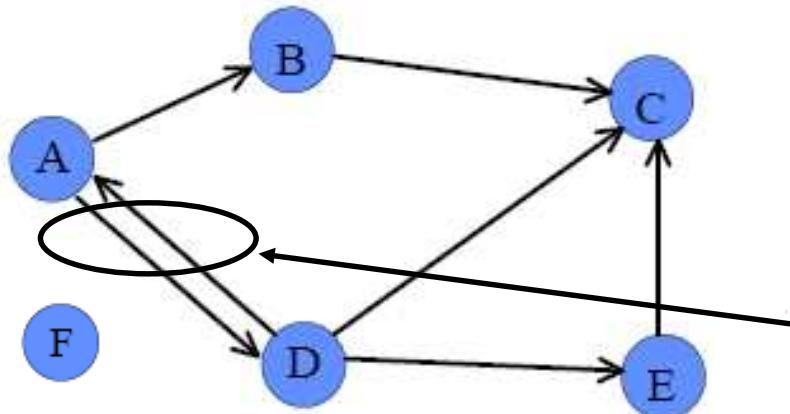


$$\begin{aligned} V &= \{A, B, C, D, E, F\} \\ E &= \{(A,B), (A,D), \\ &\quad (B,C), (C,D), \\ &\quad (C,E), (D,E)\} \end{aligned}$$

Grafi orientati e non orientati

Un grafo (V, E) è non orientato se l'insieme degli archi E è un insieme di coppie non ordinate

Un grafo (V, E) è orientato se l'insieme degli archi E è una relazione binaria tra vertici.



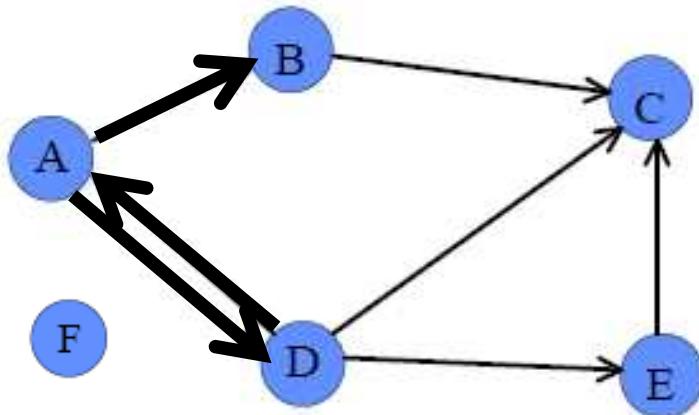
$$\begin{aligned} V &= \{A, B, C, D, E, F\} \\ E &= \{(A,B), (A,D), \\ &\quad (B,C), (C,D), \\ &\quad (C,E), (D,E)\} \end{aligned}$$

(A,D) e (D,A) denotano
due archi diversi

Proprietà di un grafo

In un grafo orientato, (w,v) si dice incidente da w a v , e v adiacente a w .

In un grafo non orientato, incidenze e adiacenze sono simmetriche.



- (A,B) è *incidente* da A a B
- (A,D) è *incidente* da A a D
- (D,A) è *incidente* da D a A

In un grafo non orientato il grado di un vertice è il numero di archi che da esso si dipartono. Per es., A ha grado 2, mentre F ha grado 0.

In un grafo orientato il grado entrante (uscente) di un vertice è il numero di archi incidenti in (da) esso. Per esempio A ha grado uscente 2 e grado entrante 1.

Percorsi sui Grafi

- Sia $G = (V, E)$ un grafo. Un percorso nel grafo è una sequenza di vertici $\langle w_1, w_2, \dots, w_n \rangle$ dove per ogni i , (w_i, w_{i+1}) è un arco di E
- La lunghezza del percorso è il numero totale di archi che connettono i vertici nell'ordine della sequenza.
- Un percorso si dice semplice se tutti i suoi vertici sono distinti (compaiono una sola volta nella sequenza), eccetto al più il primo e l'ultimo che possono coincidere.
- Se esiste un percorso p tra i vertici v e w , si dice che w è raggiungibile da v tramite p .
- Un ciclo in un grafo è un percorso $\langle w_1, \dots, w_n \rangle$ tale che $w_1 = w_n$.
- Un grafo senza cicli è detto aciclico.
- Un grafo è completo se ha un arco tra ogni coppia di vertici.
- Maggiori dettagli sulle slide di teoria del Prof. Benerecetti.....

Implementazione di grafi

Per rappresentare un grafo si può utilizzare:

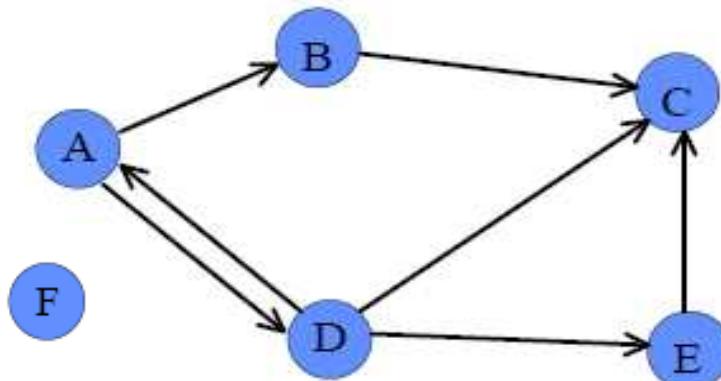
- Una Lista di Adiacenza
- Una matrice di Adiacenza.

Matrice di adiacenza

Supponiamo di avere un Grafo G di n nodi $0, 1, \dots, n$ e di volerlo rappresentare con una matrice di adiacenza. Si definisce allora una matrice $M[n,n]$ riempita utilizzando la seguente regola

$$M(v, w) = \begin{cases} 1 & \text{se } (v, w) \in E \\ 0 & \text{altrimenti} \end{cases}$$

Per esempio, il seguente grafo costituito da 6 nodi è rappresentato dalla seguente matrice $M[6,6]$



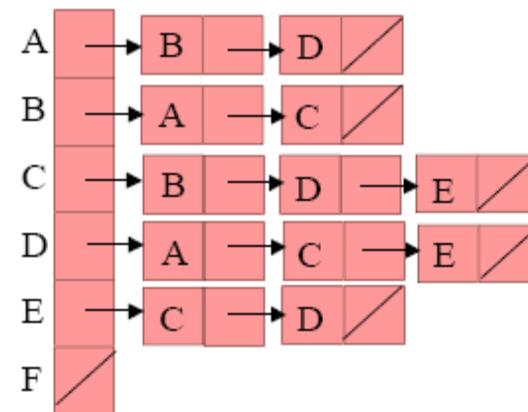
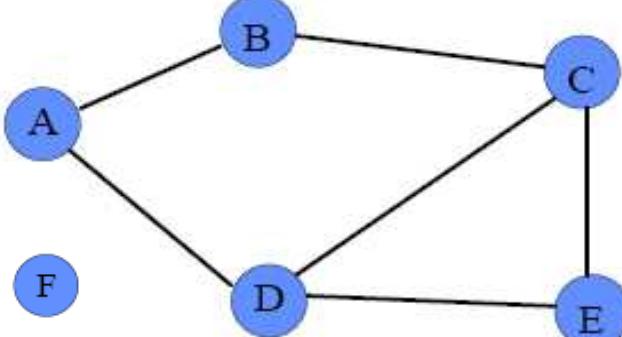
| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 1 | 0 | 1 | 0 | 0 |
| B | 0 | 0 | 1 | 0 | 0 | 0 |
| C | 0 | 0 | 0 | 0 | 1 | 0 |
| D | 1 | 0 | 1 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 1 | 0 | 0 |
| F | 0 | 0 | 0 | 0 | 0 | 0 |

Matrice di adiacenza

Dovendo rappresentare un grafo G con n nodi ordinati con una lista di adiacenza, si definiscono n liste (una per ogni vertice) riempite nel modo seguente. Per ogni nodo v del grafo,

$$L(v) = \text{lista di } w, \text{ tale che } (v, w) \in E,$$

Per esempio, il seguente grafo di 6 nodi è rappresentato nel modo seguente



Complessità

Matrice di adiacenza

- Spazio richiesto $O(|V|^2)$
- Verificare se i vertici u e v sono adiacenti richiede tempo $O(1)$
- Molti 0 nel caso di *grafi sparsi*

Liste di adiacenza

- Spazio richiesto $O(|E|+|V|)$
- Verificare se i vertici u e v sono adiacenti richiede tempo $O(|V|)$.

Interrogazione di un Grafo 1/2

Di seguito elenchiamo alcune delle operazioni più comuni di interrogazioni su un grafo G:

- `Isempty(G)`: restituisce TRUE se il grafo è vuoto.
- `numVertices(G)`: restituisce il numero di vertici.
- `numEdges(G)`: restituisce il numero di archi.
- `endVertices(G,e)`: Restituisce le due estremità dell'arco e.
- `Grado(G,v)`: Restituisce il grado di un nodo v.

Interrogazione di un Grafo 2/2

Di seguito elenchiamo alcune delle operazioni più comuni di interrogazioni su un grafo G:

- Adiacente(G,v): Restituisce i vertici adiacenti al vertice v.
- Incidente (G,v): Restituisce i vertici incidenti sul vertice v.
- SonoAdiacenti(G, v, w): Restituisce TRUE se i vertici v e w sono adiacenti.
- Completo(G): Restituisce TRUE se G è completo.
- Fortemente_connesso(G): valuta se G è fortemente connesso.
- Stampa(G): Stampa il grafo.

Aggiornamento di un Grafo

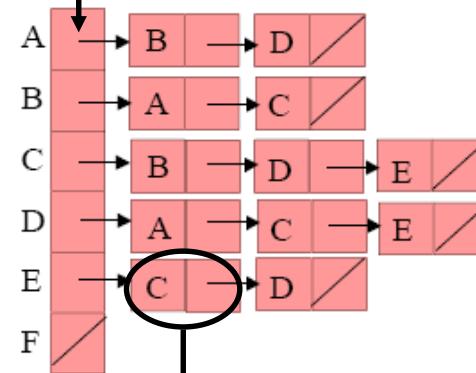
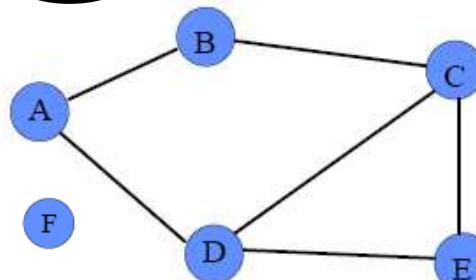
Di seguito elenchiamo alcune delle operazioni più comuni sulla modifica di un grafo G:

- Crea_grafo(n)
- Aggiungi_vertice(G, v)
- Aggiungi_arco(G, e)
- Rimuovi_vertice(G, v)
- Rimuovi_arco(G, e)
- Free(G)

Implementazione di un grafo

Implementazione della rappresentazione di un grafo orientato tramite lista di adiacenze

```
typedef struct graph {  
    int nv; /* numero di vertici del grafo */  
    edge **adj; /* vettore con le liste delle adiacenze */ } graph;
```

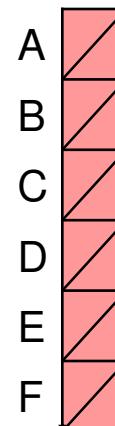


```
typedef struct edge {  
    int key;  
    struct edge *next; } edge;
```

Creazione di un grafo

Sia G un grafo non orientato di n elementi $0, 1, \dots, n-1$ senza archi, da rappresentare con liste di adiacenza. La seguente funzione crea e restituisce un puntatore ad una struttura dati grafo con n liste di adiacenza vuote. Si noti come la presenza di ogni nodo i è implicita nella allocazione di memoria per la lista (vuota) dei nodi adiacenti ad i .

```
graph *g_empty(int n)
{graph *G; int i;
 G = (graph*)malloc(sizeof(graph));
 if (G==NULL) printf("ERRORE: impossibile allocare memoria per il grafo\n");
 else {
    G->adj = (edge**)malloc(n*sizeof(edge*));
    if (G->adj==NULL) {
        printf("ERRORE: impossibile allocare memoria per la lista del grafo\n");
        free(G);
        G=NULL;}
    else {
        G->nv = n;
        for (i=0; i<n; i++)
            G->adj[i]=NULL;}}
return(G); }
```



Stampa di un grafo

La seguente funzione controlla se un grafo è vuoto:

```
int is_empty(graph *G) { return (G==NULL); }
```

La seguente funzione serve a stampare un grafo G non orientato. Si ricordi che il grafo ha n nodi ordinati 0,1,...n-1

```
void g_print(graph *G)
{
    int i, ne=0;
    edge *e;
    if (!is_empty(G))
    {
        printf("\n Il grafo ha %d vertici\n", G->nv);
        for (i=0; i<G->nv; i++)
        {
            printf("nodi adiacenti al nodo %d -> ", i);
            e=G->adj[i];
            while (e!=NULL)
            {
                printf("%d ", e->key); ne=ne+1; e=e->next; }
            printf("\n");
        }
        printf("\n Il grafo ha %d archi \n", ne); }
}
```

Aggiunta di un arco

Mostriamo una funzione che inserisce l'arco (u,v) in un grafo G .

Precondizioni (da controllare in fase di implementazione!!!):

- G diverso da NULL; u e v vertici del grafo (compresi tra 0 e $G->nv-1$); l'arco (u,v) non è già presente nel grafo

```
void g_add(graph *G, int u, int v)
{
    edge *new, *e;
    [... Aggiungere frammento di codice per i controlli vari]
    new = (edge*)malloc(sizeof(edge));
    if (new==NULL) printf("ERRORE: impossibile allocare memoria \n");
    else {
        new->key=v; new->next=NULL;
        if (G->adj[u] == NULL) //il nodo u non ha archi //
            G->adj[u] = new;
        else {
            e=G->adj[u];
            while (e->next!=NULL) e=e->next;
            e->next=new; }
    }
}
```

Si può anche aggiungere il nodo in testa.
Si deve comunque scorrere tutta la lista per controllare se il nodo è già presente.

Rimozione di un arco

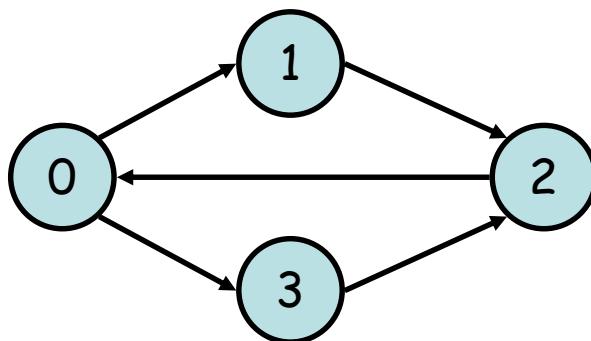
Di seguito mostriamo una funzione per la rimozione di un arco

Prerequisiti: G non NULL; u e v vertici del grafo (compresi tra 0 e $G->nv-1$);
l'arco (u,v) esiste

```
void g_remove_edge(graph *G, int u, int v)
{
    edge *prev; /* l'arco precedente a quello da togliere nella lista */
    edge *e; /* l'arco da togliere dalla lista */
    e=G->adj[u];
    if (e->key == v)
        G->adj[u] = e->next;
    else
    { prev=e;
        while (prev->next->key != v)  prev=prev->next;
        e=prev->next;
        prev->next=e->next;
    }
    free(e);
}
```

Grafo Trasposto

- Sia $G=(V,E)$ un grafo orientato. Il trasposto di G è un grafo $G'=(V,E')$, dove E' è l'insieme degli archi (v,u) tali che (u,v) è un arco di E .
- Dunque, il grafo trasposto G' è il grafo G con tutti i suoi archi invertiti.
- Per esempio, si consideri il seguente grafo, la sua rappresentazione e quella del suo trasposto con matrici di adiacenza:



| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 0 |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 1 | 0 | 0 | 0 |

- Nel caso di rappresentazione con matrice di adiacenza, il grafo G' è rappresentato dalla trasposta di G che è dunque calcolata in $O(V)$.
- Esercizio:** Implementare un algoritmo efficiente per la computazione del trasposto di un grafo rappresentato con liste di adiacenza e confrontare la complessità asintotica dell'algoritmo con quella dell'algoritmo precedente.

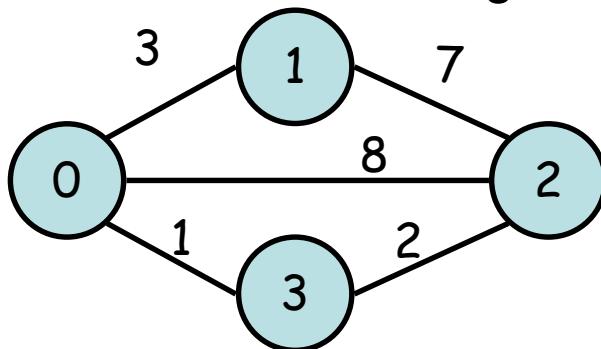
Grafi pesati

La struttura di un grafo può essere generalizzata associando un valore numerico, detto peso, ai suoi archi. Un grafo di tale genere si dice pesato. I grafi pesati vengono rappresentati mediante

- Matrici di adiacenza: se l'elemento di indici i, j della matrice di adiacenza è un valore diverso da 0 esso è il peso dell'arco (i,j) , altrimenti non esiste un arco fra i nodi i e j ;
- Liste di adiacenza: un elemento della lista di adiacenza al nodo i contiene un campo per memorizzare il nome del nodo adiacente (ad esempio, j), un campo per memorizzare il peso dell'arco (nell'esempio, il peso dell'arco (i,j)) ed un campo per memorizzare il puntatore all'elemento successivo nella lista.

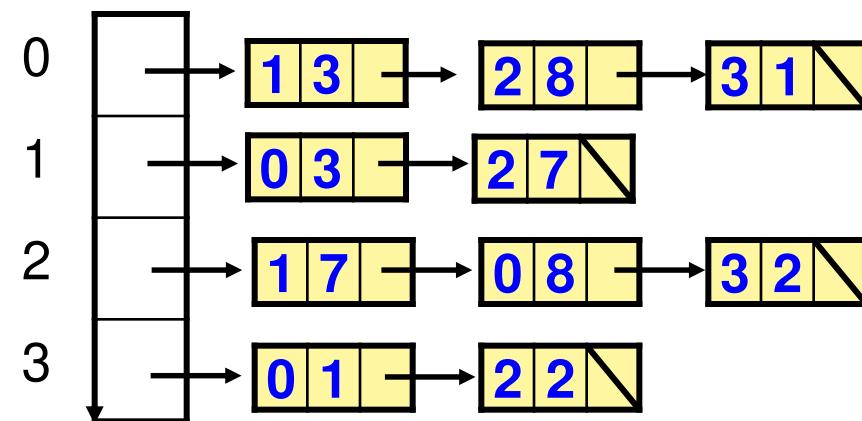
Rappresentazione con liste

Si consideri di nuovo il grafo dell'esempio precedente:



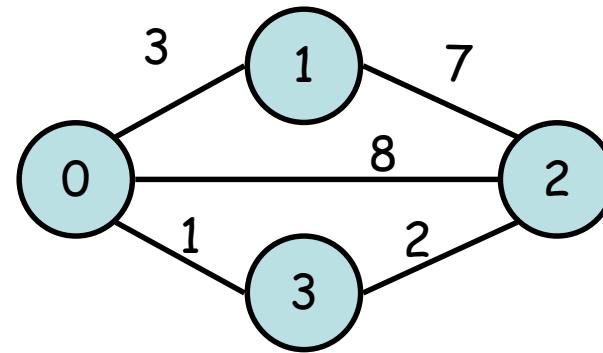
```
typedef struct edge
{
    int key;
    int peso;
    struct edge *next;} edge;
```

Vertice di collegamento
Peso dell'arco



Rappresentazione con matrice

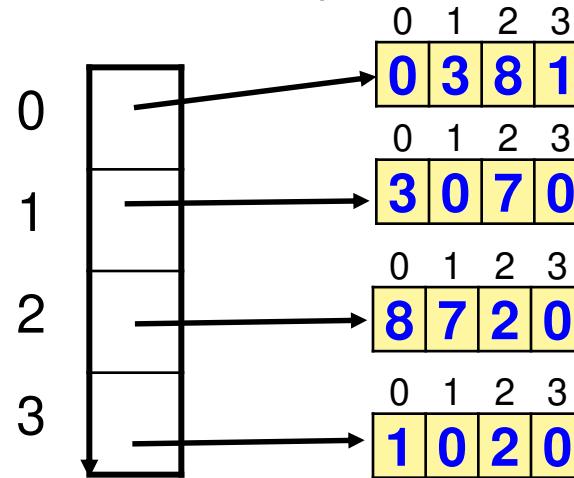
Per esempio, si consideri il seguente grafo:



Matrice di adiacenza

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 3 | 8 | 1 |
| 1 | 3 | 0 | 7 | 0 |
| 2 | 8 | 7 | 2 | 0 |
| 3 | 1 | 0 | 2 | 0 |

Matrice come vettore di puntatori



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

**Grafi:
Inserimento e Cancellazione di un nodo
Visite in ampiezza e profondità**

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 17

Parole chiave: **Modifica struttura, DFS, BFS**

next



Percorsi di Formazione a Distanza
e-Learning

Università degli Studi di Napoli Federico II



Centro di Ricerca e Sviluppo
CNR - IIT



Progetto
Europa 2000+



Università
degli Studi
di Napoli
Federico II



Università
degli Studi
di Napoli
Federico II

Realizzata con il coinvolgimento del Istituto Universitario Lattesco. Missa 3.0. Mhawehem anno I – P.D.R. Campagna 2000-2006

Possibili scenari

Bisogna distinguere i seguenti casi:

- Rappresentazione (matrice di adiacenza/lista di adiacenza)
- Grafo orientato/non orientato

Matrice di adiacenza

Per aggiungere o eliminare un vertice in un grafo rappresentato con matrici di adiacenza, abbiamo bisogno di definire la matrice di rappresentazione dinamicamente.

Una matrice definita dinamicamente può essere passata alle funzioni progettate per lavorare con matrici di dimensioni diverse.

Allocazione di memoria dinamica di una matrice di adiacenza

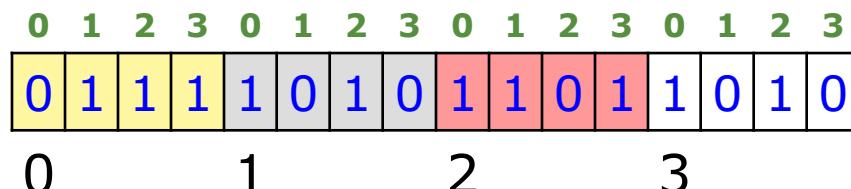
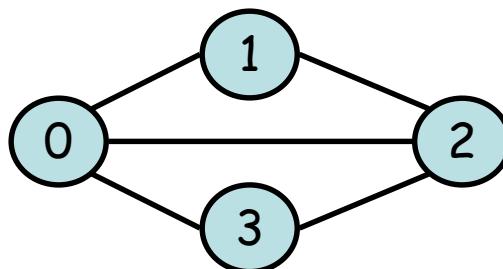
Sia G un grafo con n vertici e $M[n,n]$ la matrice di adiacenza corrispondente. Consideriamo le seguenti possibilità di allocazione di memoria dinamica per la matrice M :

- **Scelta 1.1:** M è un vettore di $n \times n$ elementi:
`int n,*M;`
`M = (int *) malloc(sizeof(int)*n*n);`

- **Scelta 1.2:** M è un vettore di puntatori:
`int n, i,**M;`
`M = (int **) malloc(n*sizeof(int*));`
`for (i=0; i<n; ++i)`
`M[i] = (int *) malloc(n*sizeof(int));`

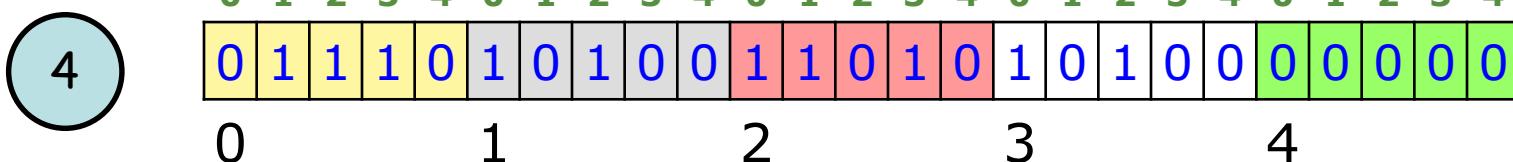
Scelta 1.1: Inserimento

La matrice è un array di n blocchi di n valori. Per esempio il secondo blocco descrive gli archi uscenti dal nodo 1:



Utilizzando questa tecnica, l'inserimento di un nodo consiste in una reallocazione di memoria (per aggiungere $n+1$ celle nel vettore) e in uno spostamento opportuno degli elementi:

- Il blocco relativo al nodo 0 resta al suo posto, il blocco relativo al nodo 1 viene spostato di una posizione verso destra... e così via, fino al blocco n .
- Infine si memorizzano i nuovi archi.

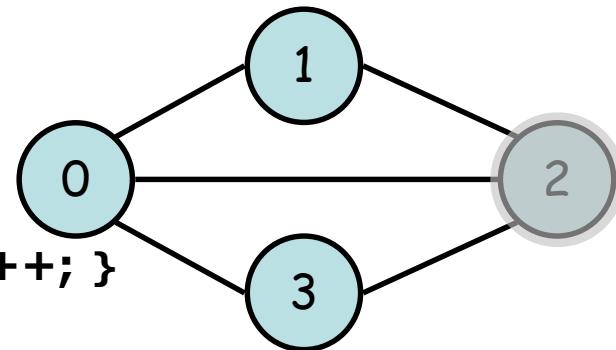


Scelta 1.1: Cancellazione

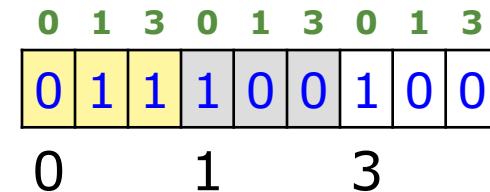
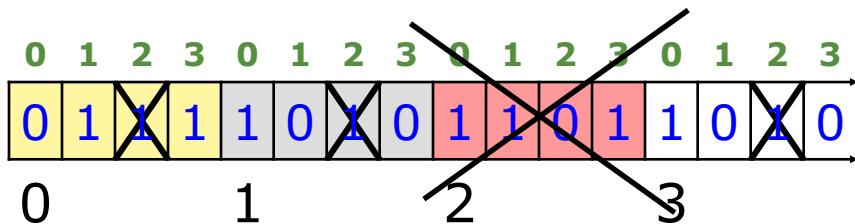
Utilizzando la scelta 1.1, la cancellazione di un nodo comporta lo spostamento opportuno degli elementi nel vettore e poi in una reallocazione del vettore nel seguente modo:

Supponendo di voler cancellare il nodo $(k+1)$ -esimo, in un vettore di n blocchi, il codice è il seguente:

```
for(i=0;i<n*n;i++) {  
    M[pos]=M[i];  
    if (( (i%n) !=k) || ( (i/n) !=k)) pos++; }
```

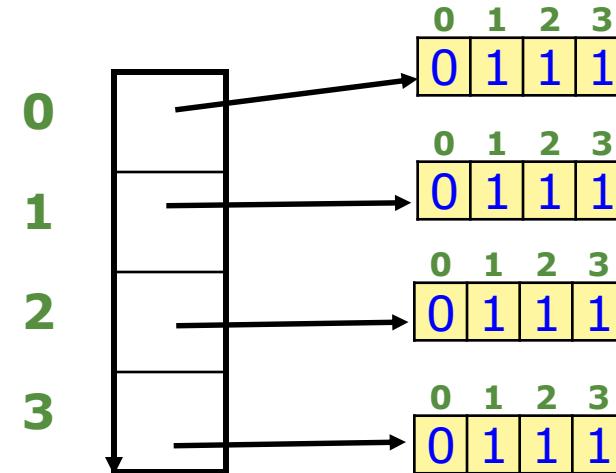
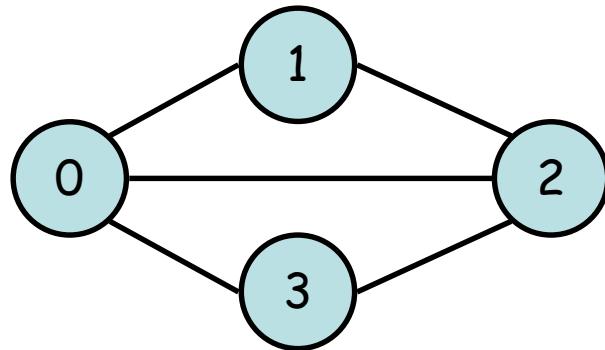


Per esempio, volendo cancellare il nodo 2 dall'esempio precedente, il risultato è il seguente:



Scelta 1.2

La matrice è un vettore di puntatori:

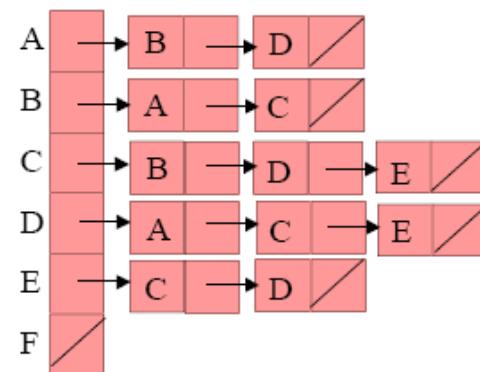
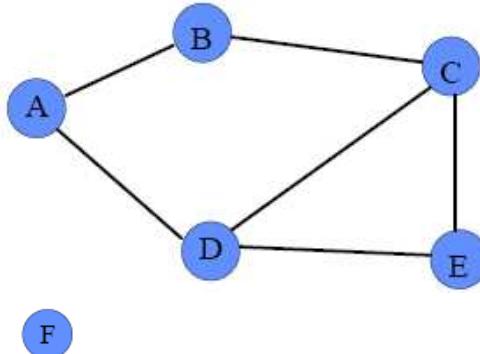


Utilizzando questa tecnica, l'inserimento di un nodo consiste in una reallocazione di memoria (aggiunta di una cella a puntatore), una allocazione di memoria per $M[n]$ e la memorizzazione dei nuovi archi.

La cancellazione invece di un nodo i consiste nella eliminazione del puntatore $M[i]$ shiftando i puntatori in M che seguono i , nella deallocazione di memoria per gli archi incidenti a i , e nella deallocazione della memoria per il vettore $*M[i]$.

Grafo con liste di adiacenza

```
typedef struct graph {
    int nv; /* numero di vertici del grafo */
    edge **adj; /* vettore con le liste delle adiacenze */
    graph;
}
```

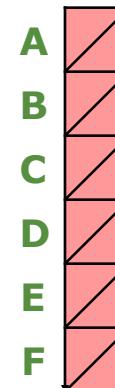


```
typedef struct edge {
    int key;
    struct edge *next; } edge;
```

Creazione di un grafo

La seguente funzione crea e restituisce un puntatore ad una struttura dati grafo che conserva il numero dei vertici e definisce n liste di adiacenza vuote.

```
graph *g_empty(int n)
{
    graph *G; int i;
    G = (graph*)malloc(sizeof(graph));
    if (G==NULL) printf("ERRORE: impossibile allocare memoria per il grafo\n");
    else
    {
        G->adj = (edge**)malloc(n*sizeof(edge*));
        if ((G->adj==NULL) && (n>0))
        {
            printf("ERRORE: impossibile allocare memoria per la lista del grafo\n");
            free(G); G=NULL;
        }
        else
        {
            G->nv = n;
            for (i=0; i<n; i++) G->adj[i]=NULL;
        }
    }
    return(G);
}
```



Inserimento di un nodo

La seguente funzione aggiunge un nodo ad un grafo G che contiene n nodi. Il nodo aggiunto avrà valore n+1.

```
graph *g_insert(G)
{
    edge **e;
    if (G==NULL) return graph *g_empty(1);
    e = realloc(G->adj, (G->nv+1) *sizeof(edge*));
    if ((e ==NULL) printf("ERRORE: impossibile reallocare memoria \n");
    else
    {
        G->adj=e;
        G->adj[G->nv]=NULL;
        G->nv = G->nv+1;
    }
    return(G);
}
```

Si ricordi che con realloc se lo spazio di memoria non può essere reallocated, il puntatore oggetto della realloc (in questo caso G->adj) rimane invariato e viene restituito Null.

Cancellazione di un nodo

Esistono due possibilità:

- cancellazione fisica del nodo.
- cancellazione logica del nodo.

Nel primo caso, occorre scalare il valore dei nodi!!!

Nel secondo caso, basta ricordarsi che il nodo è logicamente assente e riutilizzabile al prossimo inserimento di nodi.

In entrambi i casi, si richiede un free della lista dei nodi adiacenti al nodo da cancellare e degli archi incidenti e adiacenti al nodo.

Esercitazione: implementare il codice corrispondente.

Cancellazione di un grafo

La seguente funzione libera la memoria occupata da un grafo.

```
void g_free(graph *G)
{
    int i; edge *e, *enext;
    if (G!=NULL)
    {  if (G->nv > 0)
        {  for (i=0; i<G->nv; i++)
            {  e=G->adj[i];
                while (e!=NULL)
                {
                    enext=e->next;
                    free(e);
                    e=enext;
                }
            }
            free(G->adj);
        }
        free(G);
    }
}
```

Attraversamento in profondità (DFS)

L'attraversamento in profondità (deep-first search, DFS) di un grafo non orientato consiste nel visitare ogni nodo del grafo utilizzando il seguente ordine:

- "Il prossimo nodo da visitare è connesso con un arco al nodo più recentemente visitato che abbia archi che lo connettano a nodi non ancora visitati".

L'attraversamento DFS, fra le altre cose, permette l'individuazione delle componenti connesse di un grafo.

Intuitivamente, la visita in profondità si può schematizzare nel modo seguente:

visita_profondità(G)

1. Passo base

se $G == \text{NULL}$ esci;

2. Passo di induzione

visita il nodo G se non è stato visitato
per ogni nodo adiacente $G->\text{adj}$

visita_profondità($G->\text{adj}$);

Idea di implementazione

Implementiamo la visita DFS tramite una semplice funzione ricorsiva:

La procedura implementata usa un array di appoggio aux per memorizzare quando un vertice è stato già incontrato. Inoltre, la funzione principale chiama al suo interno un'altra procedura: dfs1

Quando dfs1 è richiamata si entra in una nuova componente connessa.

dfs1 richiamerà se stessa ricorsivamente fino a quando tutta la componente è stata visitata.

Codice per DFS

Di seguito mostriamo il codice per dfs e dfs1:

```
void dfs(struct graph *g) {
    int i, *aux = calloc(g->nv,sizeof(int));
    if(!aux) {printf("Errore di Allocazione\n");}
    else {
        for(i = 0; i < g->nv; i++)
            if(!aux[i]) {printf("\n%d,",i); dfs1(g,i,aux);}
        free(aux);}

void dfs1(struct graph *g, int i, int *aux) {
    edge *e;
    aux[i] = 1;
    for(e = g->adj[i]; e; e = e->next)
        if(!aux[e->key ]) { printf("%d,",e->v); dfs1(g,e->key,aux);}
}
```

Poiché **dfs1** contiene un ciclo sulla lista di adiacenza del nodo con cui è richiamata, ogni arco viene esaminato in totale due volte, mentre la lista di adiacenza di ogni vertice è scandita una volta sola.

La visita DFS con liste di adiacenza richiede $O(|V| + |E|)$.

Attraversamento in Ampiezza(BFS)

L'attraversamento in ampiezza (breadth-first search, BFS) è un modo alternativo al DFS per visitare ogni nodo di un grafo non orientato.

Il prossimo nodo da visitare lo si sceglie fra quelli che sono connessi al nodo visitato da più tempo e che abbia archi che lo connettano a nodi non ancora visitati.

Vediamone un'implementazione non ricorsiva che memorizza in una coda i nodi connessi al nodo appena visitato.

Sostituendo la coda con uno stack si ottiene una (leggera variante della) visita DFS.

Il codice per BFS

```
void bfs(struct graph *g) {
    int i, *aux = calloc(g->V,sizeof(int));
    if(!aux) { printf("Errore di Allocazione\n"); }
    else {
        for(i = 0; i < g->nv; i++)
            if(!aux[i]) { printf("\n%d,",i+1); bfs1(g,i,aux); }
        free(aux); }

void bfs1(struct graph *g, int i, int *aux) {
    edge *e;
    intqueue *q = createqueue();
    enqueue(q,i);
    while(!emptyq(q)) {
        i = dequeue(q);
        aux[i] = 1;
        for(e = g->adj[i]; e; e = e->next)
            if(!aux[e->key]) {
                enqueue(q,e->key); printf("%d,",e->key); aux[e->key] = 1; }
    destroyqueue(q);}
```

Idea di implementazione di BFS

La procedura implementata per la visita BFS usa un array di appoggio aux per memorizzare quando un vertice è già stato incontrato.

Quando bfs1 è richiamata da bfs si entra in una nuova componente connessa.

bfs1 usa una coda per memorizzare da quali vertici riprendere la visita quando la lista di adiacenza del vertice corrente è stata tutta esplorata.

Ogni lista è visitata una volta, e ogni arco due volte.

La visita BFS con liste di adiacenza richiede $O(|V| + |E|)$.

Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Esercitazione di laboratorio: Problema del venditore Prima parte

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Laboratorio di Algoritmi e Strutture Dati
13917
murano@na.infn.it
2007/2008

Lezione numero: 18

Parole chiave: Progetto

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Realizzata con il coinvolgimento del Istituto Universitario Lattesano - Missa 3.12. Murastrada anima I - P.O.R. Campania 2000-2006

Esercizio 1/2

Si consideri il seguente problema:

Un venditore ha clienti sparsi in n diverse città. Per ognuna di queste città, il venditore conosce esattamente se e in che modo è collegata alle altre città (collegamento mono o bidirezionale) e, per ogni collegamento, la sua lunghezza.

Si supponga per semplicità che se due città hanno un collegamento in entrambe le direzioni, la lunghezza dei due collegamenti deve essere la stessa.

Si supponga inoltre che il venditore conosca il fatturato per ogni singola città (compreso quello della sua città che è inclusa in n) e il tempo necessario da trascorrere in ogni città per ottenere il corrispondente fatturato.

Esercizio 2/2

Si implementino in linguaggio C le seguenti operazioni utilizzando come struttura dati di appoggio un grafo, indipendentemente dal fatto che il grafo sia rappresentato con liste di adiacenza o con matrice di adiacenza:

1. Creazione della struttura dati grafo contenente tutte le città con le relative informazioni.
2. Aggiunta di un collegamento.
3. Rimozione/modifica-lunghezza di un collegamento.
4. Aggiunta di una città
5. Cancellazione/modifica-dati di una città
6. Stampa di tutte le città e delle relative informazioni

Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Componenti fortemente connesse e Alberi minimi di copertura

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 21

Parole chiave: **Bellman-Ford, Dijkstra, Floyd-Warshall**

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II

Riepilogo delle lezioni precedenti

Definizione di Grafo orientato e non orientato

Rappresentazione di Grafi tramite l'suo di liste di adiacenza e di matrici di adiacenza

Implementazione di algoritmi di base per la gestione di grafi (orientati e non orientati):

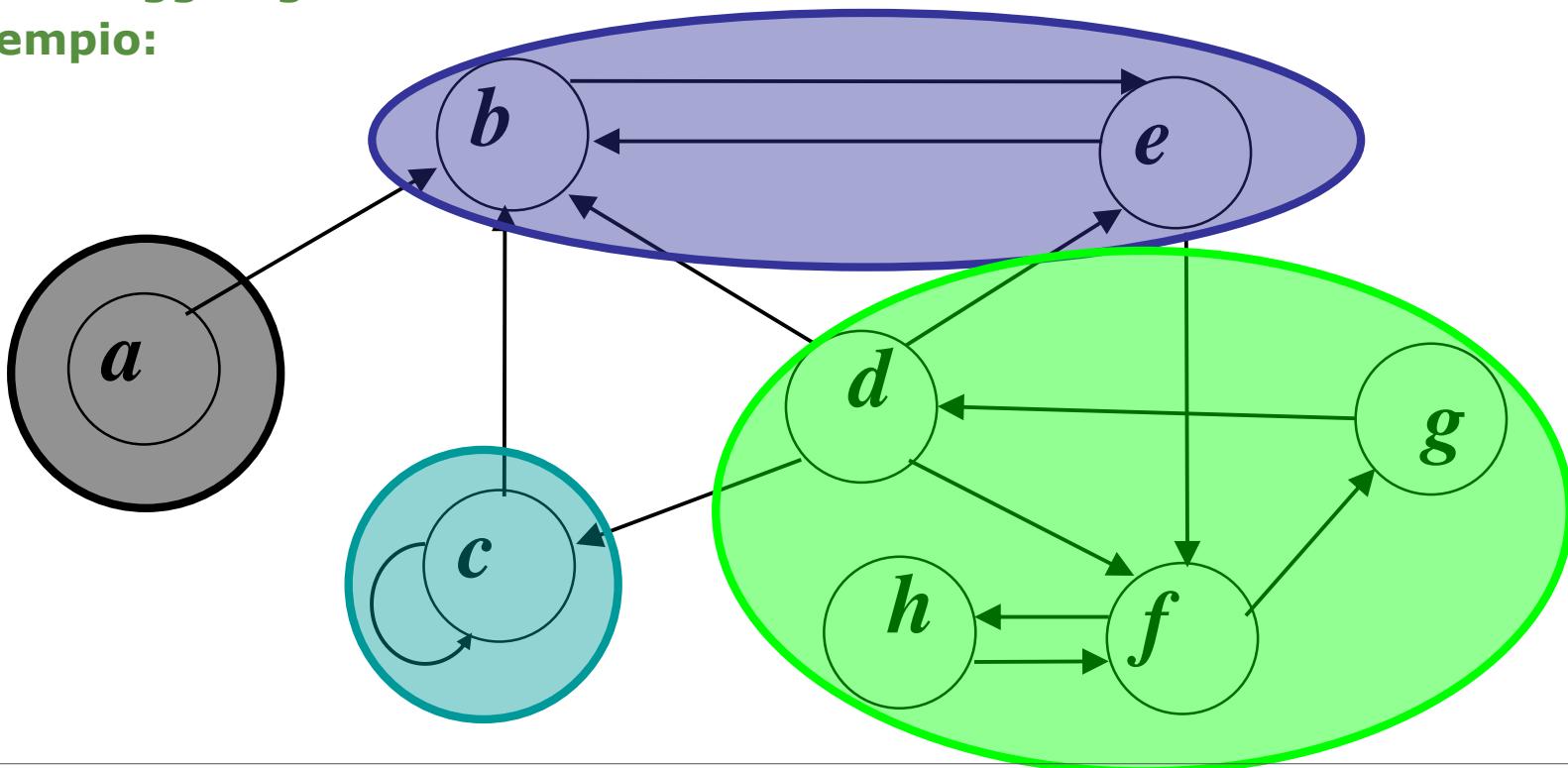
- Creazione di un grafo (semplice o pesato)
- Interrogazione di un grafo: calcola grado, stampa grafo, ecc.)
- Modifica di un grafo: Aggiungi/rimuovi vertice, aggiungi/rimuovi arco, grafo trasposto, ecc.
- Visita di tutti i nodi di un grafo (BFS e DFS) e raggiungibilità di un nodo.

Oggi presenteremo un algoritmo per il calcolo di componenti fortemente connesse. Inoltre, verranno introdotti due algoritmi per il calcolo di un albero minimo di copertura.

Componente fortemente connessa

Dato un grafo diretto $G=(V,E)$, una componente fortemente connessa (SCC, Strongly Connected Component) è un insieme massimale di vertici U sottoinsieme di V tale che per ogni coppia di vertici u e v in U , u è raggiungibile da v e viceversa.

Esempio:



Applicazioni

Dato un grafo, l'operazione di decomposizione nelle sue componenti fortemente connesse ha molte applicazioni pratiche

Per esempio:

- la disposizione di sensi unici in una città,
- I vincoli tra variabili all'interno di un programma, che devono dunque essere presi in considerazione contemporaneamente,
- la correlazione di moduli di un programma e dunque la capacità per un compilatore di raggruppare moduli in modo efficiente

Algoritmo per SCC

L'algoritmo che proponiamo per la decomposizione di un grafo nelle sue componenti fortemente connesse usa due visite in profondità, una sul grafo G e l'altra sul grafo trasposto G' :

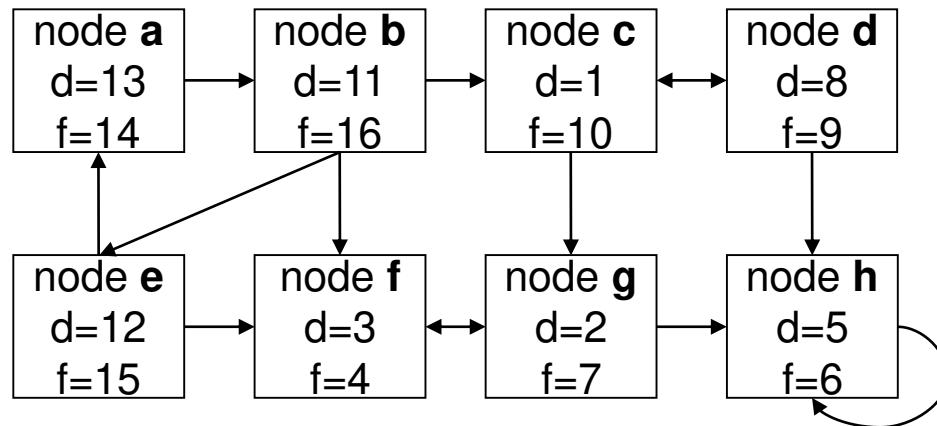
Algoritmo per SCC:

- Chiamare $\text{DFS}(G)$ per computare $f[v]$ (l'ordine finale di visita) per ogni vertice v
- Calcolare il grafo trasposto G' di G
- Chiamare $\text{DFS}(G')$, ma nel ciclo principale della DFS e si considerino i vertici in ordine decrescente di $f[v]$
- Restituire i vertici di ciascun albero calcolato con la $\text{DFS}(G')$ come una SCC

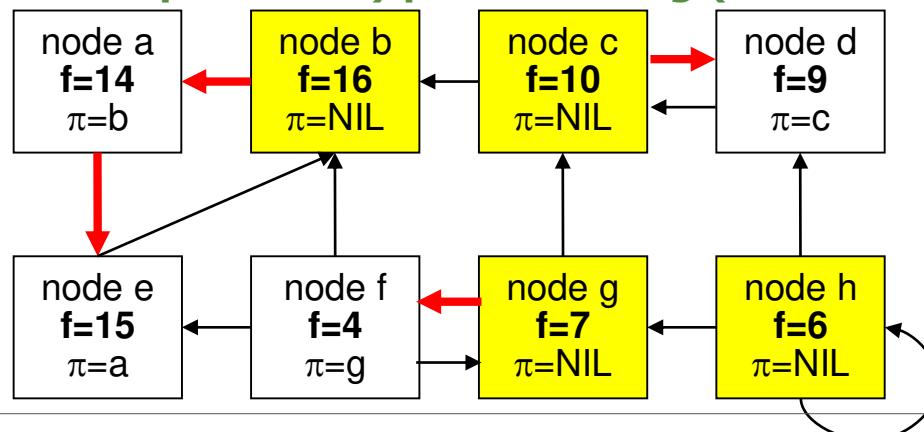
Per calcolare $f[v]$, si noti che nella vista in profondità, a ciascun vertice si possono associare due tempi: il primo corrisponde a quando il nodo è scoperto, il secondo corrisponde a quando la ricerca finisce di esaminare i suoi adiacenti.

Esempio

DFS su G, partendo da c.

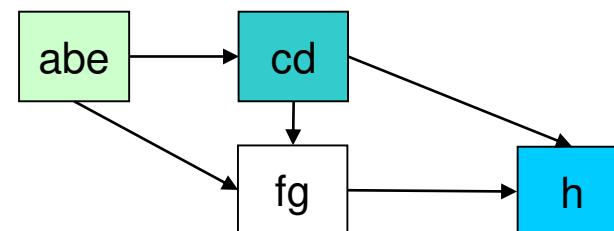
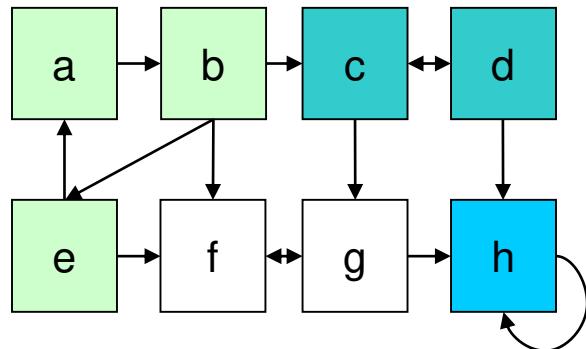
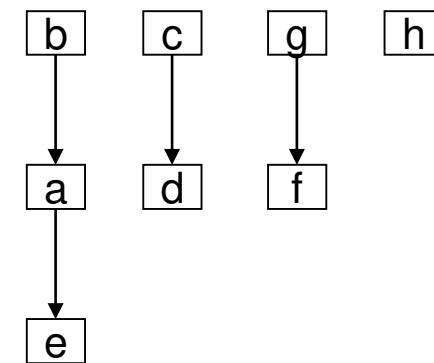
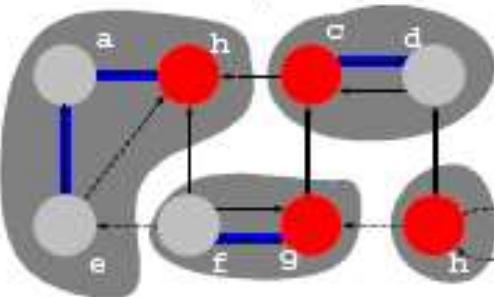
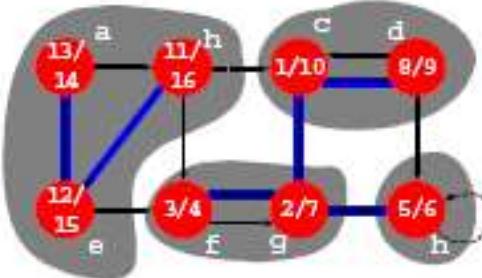


DFS su G' (il grafo trasposto di G) partendo da g (con f massima).



Strongly-Connected Components

Di seguito riportiamo i 4 alberi risultanti che formano i quattro SCC del grafo dato.



Albero ricoprente

Un albero ricoprente di un grafo $G=(V,E)$ è un albero $T=(V',E')$ tale che $V' = V$ e E' è un sottoinsieme di E .

Si ricorda che un albero è un grafo (non orientato) connesso e senza cicli. Ricordiamo anche che un ciclo in un grafo è un cammino in cui il nodo di partenza e di destinazione coincidono e che non passa due volte per nessun nodo intermedio.

Minimum Spanning Tree

Il problema del *Minimo Albero Ricoprente* (in breve, *MST*, *Minimum Spanning Tree*) è definito come segue:

- dato un grafo G (non orientato e) pesato, trovare un albero ricoprente di G tale che la somma dei pesi dei suoi archi sia minima.

L'importanza di tale problema è enorme. A titolo di esempio, si supponga di dover realizzare l'impianto di distribuzione dell'energia elettrica di una zona, nella quale esistono un certo numero di abitazioni che devono essere servite. Si supponga che di ogni abitazione si conosca la posizione, nonché la distanza da tutte le altre. L'albero di copertura minimo rappresenta la soluzione che consente la minimizzazione della lunghezza dei collegamenti da realizzare.

Nota: L'albero di copertura minimo per un grafo in generale non è unico.

Tecnica greedy per calcolare un MST

Gli algoritmi per il calcolo di un albero di copertura minimo che vedremo, si basano su un approccio chiamato greedy.

Gli algoritmi greedy rappresentano una particolare categoria di algoritmi di ricerca o ottimizzazione.

In molti problemi ad ogni passo l'algoritmo deve fare una scelta: seguendo la strategia greedy (ingordo), l'algoritmo fa la scelta più conveniente in quel momento.

Esempio: Supponiamo di avere un sacchetto di monete di euro e di voler comporre una precisa somma di denaro x , utilizzando il minor numero di monete. La soluzione consiste nel prendere dal sacchetto sempre la moneta più grande tale che sommata alle monete scelte in precedenza il totale non sia superiore a x .

In molti casi (ma non sempre), la tecnica greedy fornisce una soluzione globalmente ottima al problema.

Algoritmo generico per un MST

L'idea generale per calcolare un MST T su un grafo G è la seguente:

T=NULL;

while T non forma un albero di copertura;

do trova un arco (u,v) che sia **sicuro** per T;

T=T unito a $\{(u,v)\}$;

return T

Si definisce sicuro un arco che aggiunto ad un sottoinsieme di un albero di copertura minimo produce ancora un sottoinsieme di un albero di copertura minimo.

In seguito vedremo due tecniche per calcolare un arco sicuro. Entrambe queste tecniche utilizzano il concetto di taglio di un grafo.

Tagli di un grafo

Dato un grafo non orientato $G=(V,E)$, un taglio è una partizione di V in due sottoinsiemi.

Un arco attraversa il taglio se uno dei suoi estremi è in una partizione, e l'altro nell'altra.

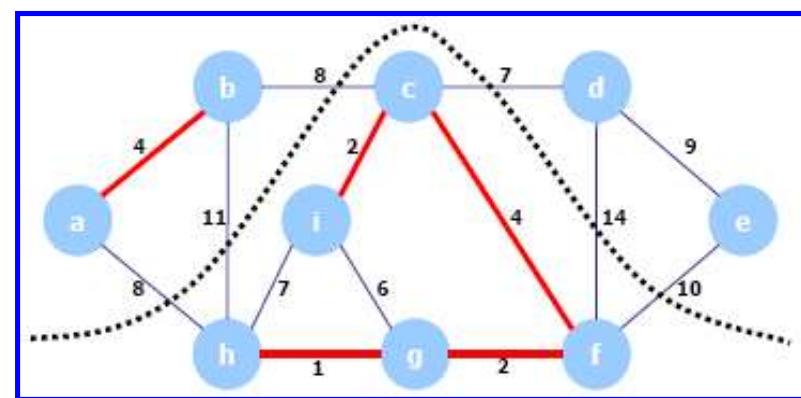
Un taglio rispetta un insieme A di archi se nessun arco di A attraversa il taglio.

Un arco si dice leggero se ha peso minimo tra gli archi che attraversano il taglio.

Esempio:

Siano gli archi in rosso un sottoinsieme A di E . Il taglio rispetta chiaramente A .

Se invece A è l'insieme degli archi adiacenti ai nodi d ed e , allora l'arco (d,c) è leggero per il taglio dato



Arco sicuro per G

Teorema:

- Sia $G=(V,E)$ un grafo non orientato e connesso con una funzione peso w a valori reali definita su E .
- Sia A un sottoinsieme di E contenuto in un qualche albero di copertura minimo per G .
- Sia $(S, V-S)$ un qualunque taglio che rispetta A .
- Sia (u,v) un arco leggero che attraversa $(S, V-S)$.
- Allora l'arco (u,v) è sicuro per A

In seguito analizziamo in dettaglio due algoritmi che calcolano un BST di un grafo formato da tutti archi sicuri calcolati utilizzando una tecnica greedy:

- **Algoritmo di Kruskal**
- **Algoritmo di Prim**

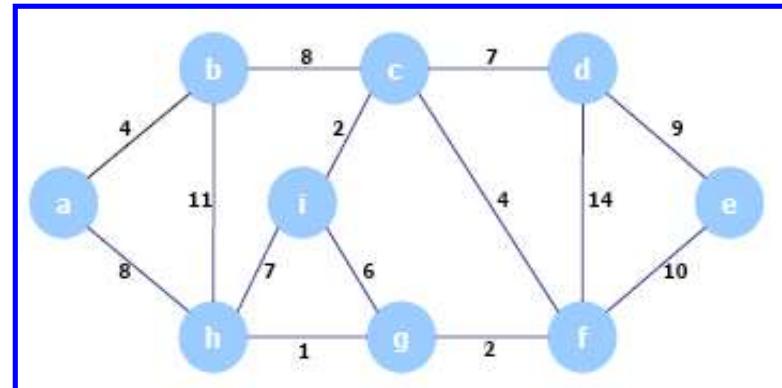
Algoritmo di Kruskal

MST-Kruskal(G,w)

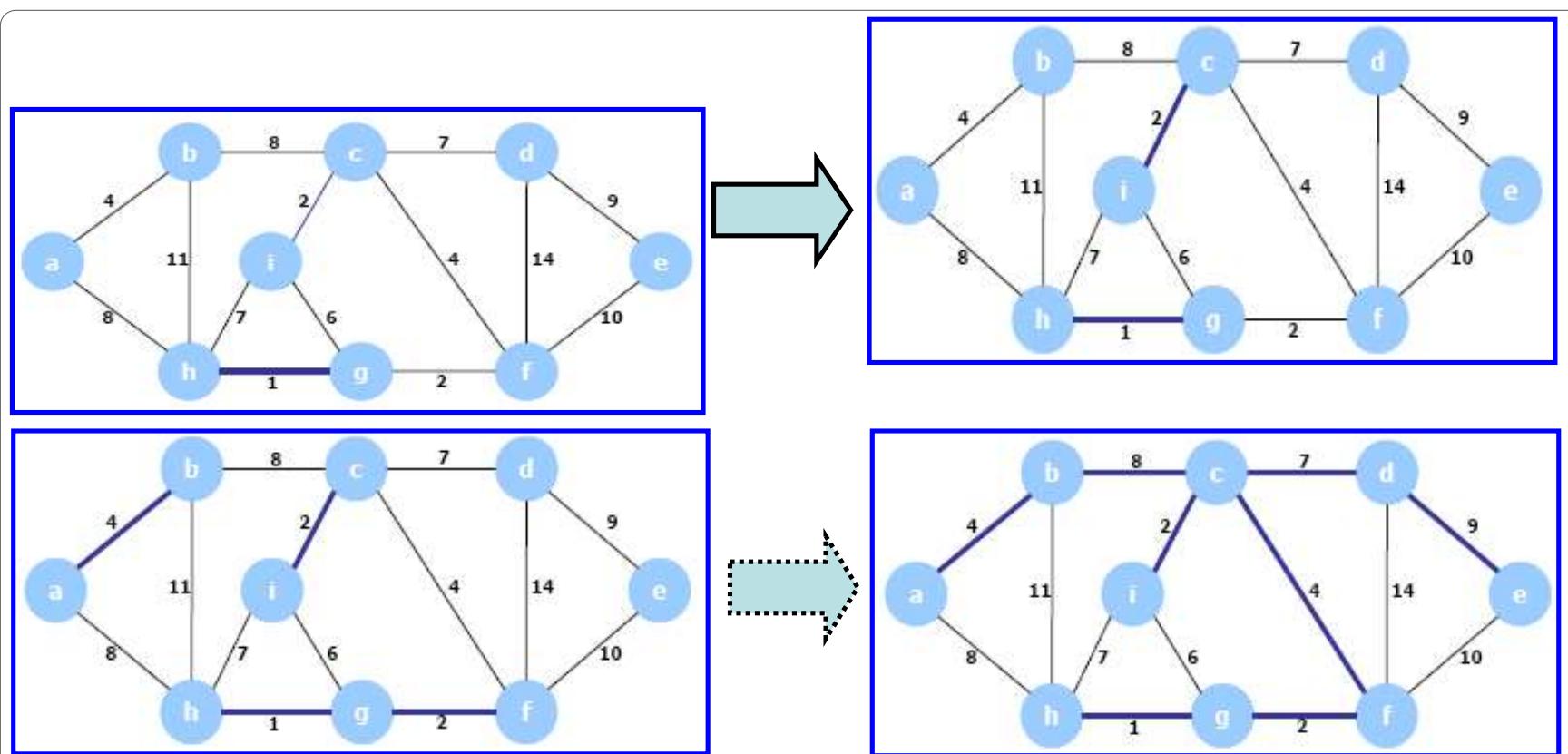
1. $A \leftarrow \emptyset$
2. **for** ogni vertice v di V
3. **do** Make-Set(V) $\backslash\backslash$ crea un insieme per ogni vertice V $\backslash\backslash$
4. ordina gli archi di E per peso w non decrescente
5. **for** ogni arco (u,v) di E , in ordine di peso non decrescente
6. **do if** Find-Set(u) \neq Find-Set(v)
7. $A \leftarrow A$ unito a $\{(u,v)\}$
8. Union(u,v) $\backslash\backslash$ fusione in un unico insieme degli insiemi di u e v
9. **return** A

Esempio:

Si consideri il seguente grafo. Inizialmente ogni insieme costituisce un insieme a se.



MST sull'esempio dato



La complessità dell'algoritmo di Kruskal dipende fondamentalmente dall'ordinamento degli archi pesati che prende tempo $O(E \log E)$.

Algoritmo di Prim

L'algoritmo di Prim parte da un nodo radice r ed espande ad ogni passo l'albero di copertura minimo A , sino a che questo non copre tutti i vertici.

Ad ogni passo viene aggiunto all'albero un arco leggero che collega un vertice in A ad un vertice in $V-A$.

Per la scelta dell'arco leggero si usa una coda di priorità.

Ad ogni istante la coda di priorità contiene tutti i vertici non appartenenti all'albero A .

La posizione di ciascun vertice v nella coda dipende dal valore di un campo chiave $key[v]$, corrispondente al minimo tra i pesi degli archi che collegano v ad un qualunque vertice in A .

Se v non è collegato a nessun vertice in A , allora $key[v]=\infty$.

Esempio:

MST-Prim(G, w, r)

1. $Q \leftarrow V[G]$
2. **for** ogni u di Q
3. **do** $key[u] \leftarrow \infty$ \\ chiave massima a tutti i nodi \\
4. $key[r] \leftarrow 0$ \\ chiave minima alla radice \\
5. $\pi[r] \leftarrow \text{NIL}$ \\ r non ha padre
6. **while** $Q \neq \emptyset$
7. **do** $u \leftarrow \text{Extract-Min}(Q)$
8. **for** ogni v in $\text{Adj}[u]$
9. **do if** v è in Q e $w(u,v) < key[v]$
10. $\pi[v] \leftarrow u$
11. $key[v] \leftarrow w(u,v)$

$O(V)$
Usando
buildheap

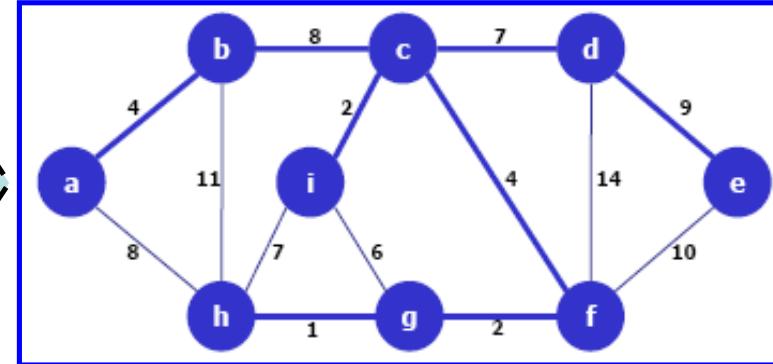
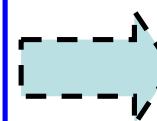
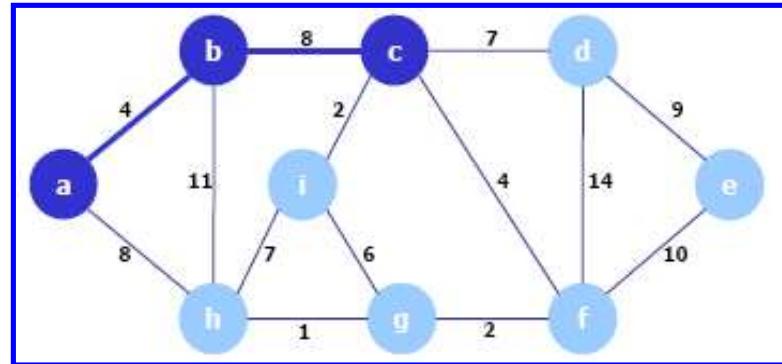
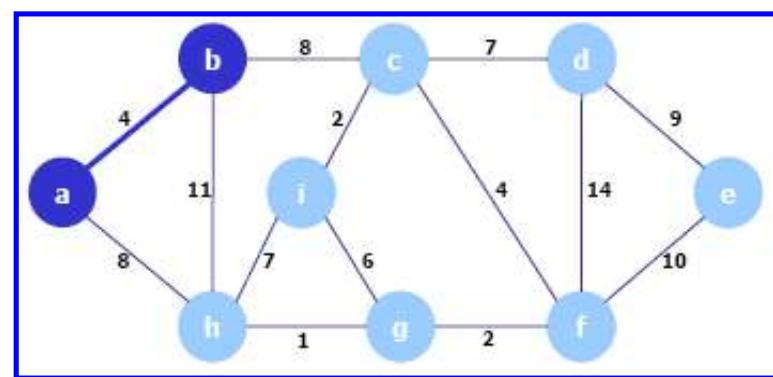
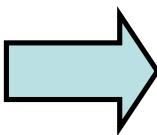
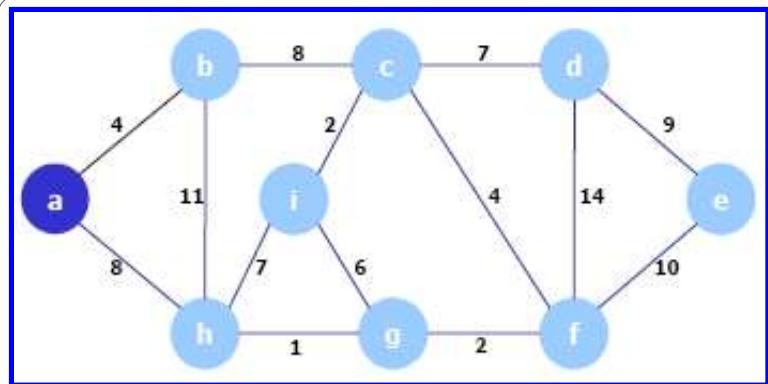
$O(\log V)$

$O(E)$

$O(\log V)$

Studi approfonditi dimostrano che la complessità asintotica dell'algoritmo di Prim è migliore dell'algoritmo di Kruskal

Esempio



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Esercitazione di laboratorio: Problema del venditore Seconda parte

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Laboratorio di Algoritmi e Strutture Dati
13917
murano@na.infn.it
2007/2008

Lezione numero: 20

Parole chiave: Progetto

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Realizzata con il coinvolgimento del Istituto Universitario Lattesano - Missa 3.12. Maturazione anno I - P.D.R. - Campagna 2000-2006

Esercizio 1/3

- Si consideri nuovamente il problema del venditore introdotto nella precedente lezione di laboratorio. Si implementino in modo efficiente, descrivendone le scelte opportune e le complessità asintotiche, le seguenti due operazioni:
 - Il venditore vuole vendere una connessione ad internet via cavo ai suoi clienti. Implementare in linguaggio C una funzione efficiente che permetta di definire la lunghezza minima di cavo necessaria per collegare tutte le città, partendo dalla città del venditore e sfruttando soltanto i collegamenti esistenti tra le città (senza tener necessariamente conto delle loro direzioni).



Esercizio 2/3

- Si supponga inoltre che una volta completato il lavoro precedente, il venditore voglia visitare tutte le città per riscuotere il pagamento del servizio dato. Si implementi dunque una funzione efficiente in linguaggio C che permetta di visitare tutte le città, rispettando i collegamenti e le direzioni esistenti tra le varie città. Tale funzione deve indicare:
 - l'ordine di visita
 - la distanza totale percorsa
 - la somma riscossa
 - le eventuali città che non sono raggiungibili.



Esercizio 3/3

- Le funzioni precedenti devono gestire anche la possibilità di modifica del numero di città e di collegamenti. In pratica, se un collegamento tra due città salta, bisogna ristabilire il collegamento internet tra per tutte le città utilizzando la parte di rete rimanente.
- Si implementino in linguaggio C le operazioni precedenti utilizzando come struttura dati di appoggio un grafo, indipendentemente dal fatto che esso sia rappresentato con liste di adiacenza o con matrice di adiacenza



Laboratorio di Algoritmi e Strutture Dati

Prof. Aniello Murano

Componenti fortemente connesse e Alberi minimi di copertura

Corso di Laurea
Codice insegnamento
Email docente
Anno accademico

Informatica
13917
murano@na.infn.it
2007/2008

Lezione numero: 21

Parole chiave: Bellman-Ford, Dijkstra, Floyd-Warshall

next



Percorsi di Formazione a Distanza
e-Learning
Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II



Università degli Studi di Napoli Federico II

Riepilogo delle lezioni precedenti

Definizione di Grafo orientato e non orientato

Rappresentazione di Grafi tramite l'suo di liste di adiacenza e di matrici di adiacenza

Implementazione di algoritmi di base per la gestione di grafi (orientati e non orientati):

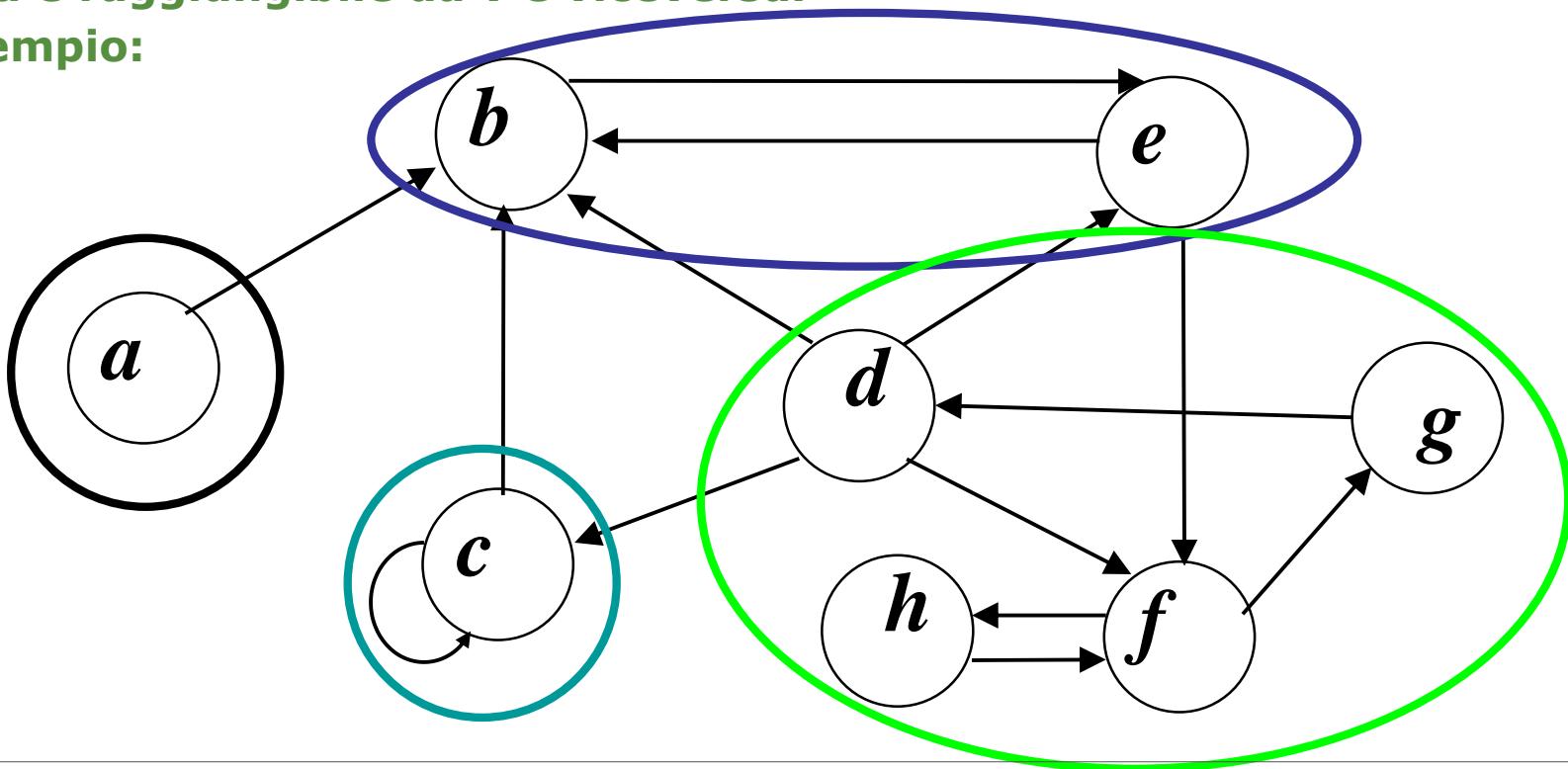
- Creazione di un grafo (semplice o pesato)
- Interrogazione di un grafo: calcola grado, stampa grafo, ecc.)
- Modifica di un grafo: Aggiungi/rimuovi vertice, aggiungi/rimuovi arco, grafo trasposto, ecc.
- Visita di tutti i nodi di un grafo (BFS e DFS) e raggiungibilità di un nodo.
- Calcolo del percorso minimo in un grafo

Oggi vedremo un algoritmo per il calcolo di componenti fortemente connesse, e due algoritmi per il calcolo di un albero minimo di copertura.

Componente fortemente connessa

Dato un grafo diretto $G=(V,E)$, una componente fortemente connessa (SCC, Strongly Connected Component) è un insieme massimale di vertici U sottoinsieme di V tale che per ogni coppia di vertici u e v in U , u è raggiungibile da v e viceversa.

Esempio:



Applicazioni

Dato un grafo, l'operazione di decomposizione nelle sue componenti fortemente connesse ha molte applicazioni pratiche

Per esempio:

- la disposizione di sensi unici in una città,
- I vincoli tra variabili all'interno di un programma, che devono dunque essere presi in considerazione contemporaneamente,
- la correlazione di moduli di un programma e dunque la capacità per un compilatore di raggruppare moduli in modo efficiente

Algoritmo (di Kosaraju) per SCC

L'algoritmo usa il teorema che se si fa una DFS di G e si calcola l'ordine finale di visita dei nodi ($f[v]$), e poi si rifà una DFS sul grafo trasposto iniziando sempre dai nodi con f massimo, il risultato è una decomposizione di G nelle sue componenti fortemente connesse:

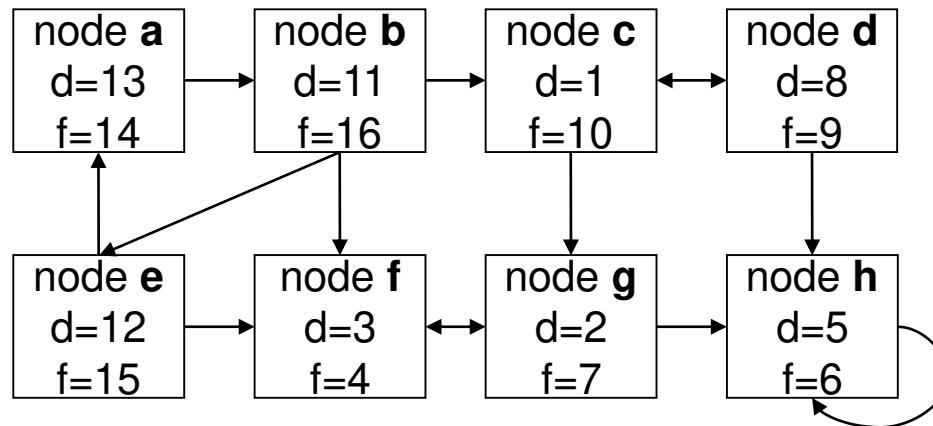
In sintesi, l'algoritmo di Kosaraju per SCC è il seguente:

- Chiamare $\text{DFS}(G)$ per computare $f[v]$ (l'ordine finale di visita) per ogni v
- Calcolare il grafo trasposto G' di G
- Chiamare $\text{DFS}(G')$, ma nel ciclo principale della DFS si considerino i vertici in ordine decrescente di $f[v]$
- Restituire i vertici di ogni albero calcolato con la $\text{DFS}(G')$ come una SCC

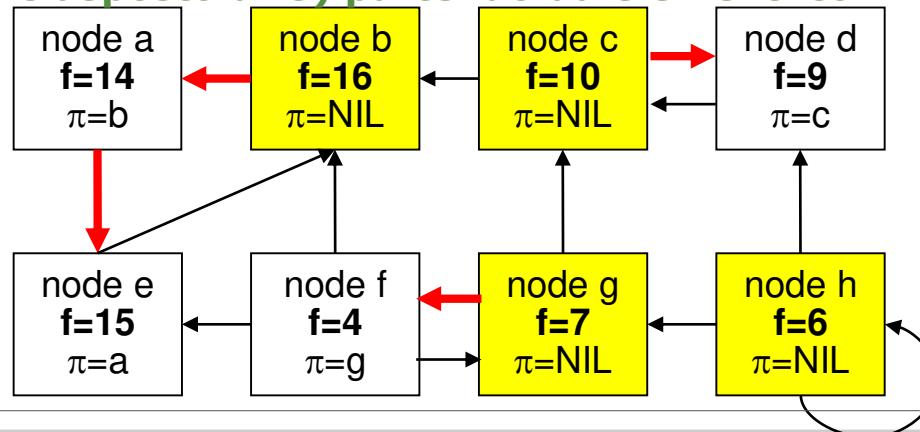
Per calcolare $f[v]$, nella DFS, può essere utile calcolare anche un tempo **d** relativo a quando v è scoperto (prima volta)

Esempio

DFS su G, partendo da c.

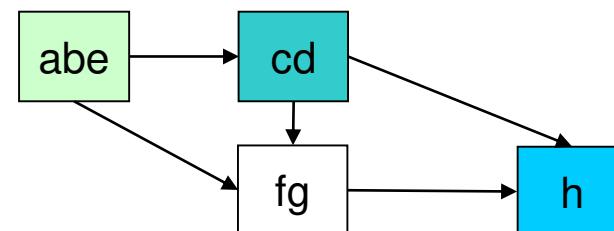
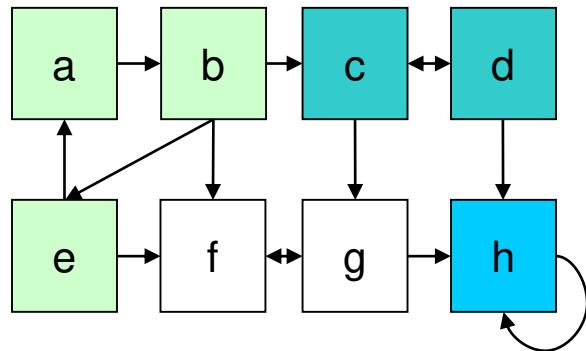
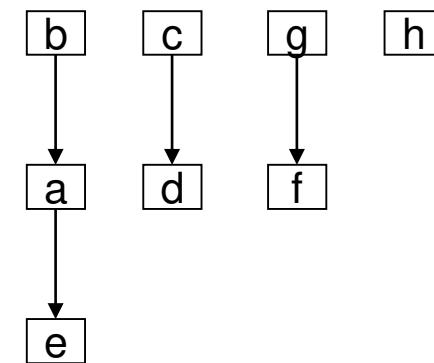
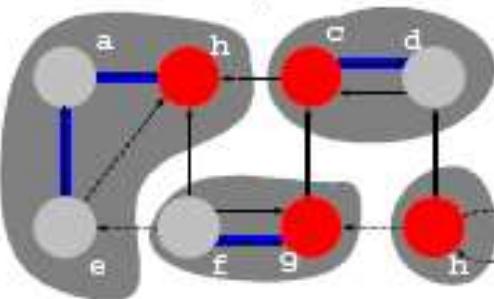
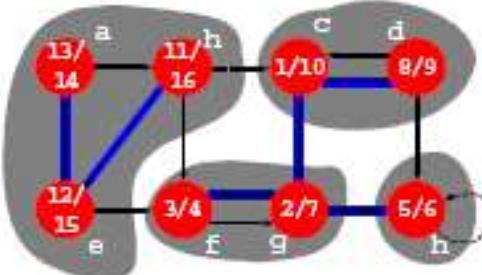


DFS su G' (il grafo trasposto di G) partendo da elementi con f massima.



Strongly-Connected Components

Di seguito riportiamo i 4 alberi risultanti che formano i quattro SCC del grafo dato.





Esercizio

**Implementare in linguaggio C
l'algoritmo di Kosaraju**

Albero ricoprente

Un albero ricoprente di un grafo $G=(V,E)$ è un albero $T=(V',E')$ tale che $V' = V$ e E' è un sottoinsieme di E .

Si ricorda che un albero è un grafo (non orientato) connesso e senza cicli. Ricordiamo anche che un ciclo in un grafo è un cammino in cui il nodo di partenza e di destinazione coincidono e che non passa due volte per nessun nodo intermedio.

Minimum Spanning Tree

Il problema del *Minimo Albero Ricoprente* (in breve, *MST*, *Minimum Spanning Tree*) è definito come segue:

- dato un grafo G (non orientato e) pesato, trovare un albero ricoprente di G tale che la somma dei pesi dei suoi archi sia minima.

L'importanza di tale problema è enorme. A titolo di esempio, si supponga di dover realizzare l'impianto di distribuzione dell'energia elettrica di una zona, nella quale esistono un certo numero di abitazioni che devono essere servite. Si supponga che di ogni abitazione si conosca la posizione, nonché la distanza da tutte le altre. L'albero di copertura minimo rappresenta la soluzione che consente la minimizzazione della lunghezza dei collegamenti da realizzare.

Nota: L'albero di copertura minimo per un grafo in generale non è unico.

Tecnica greedy per calcolare un MST

Gli algoritmi per il calcolo di un albero di copertura minimo che vedremo, si basano su un approccio chiamato greedy.

Gli algoritmi greedy rappresentano una particolare categoria di algoritmi di ricerca o ottimizzazione.

In molti problemi ad ogni passo l'algoritmo deve fare una scelta: seguendo la strategia greedy (ingordo), l'algoritmo fa la scelta più conveniente in quel momento.

Esempio: Supponiamo di avere un sacchetto di monete di euro e di voler comporre una precisa somma di denaro x , utilizzando il minor numero di monete. La soluzione consiste nel prendere dal sacchetto sempre la moneta più grande tale che sommata alle monete scelte in precedenza il totale non sia superiore a x .

In molti casi (ma non sempre), la tecnica greedy fornisce una soluzione globalmente ottima al problema.

Algoritmo generico per un MST

L'idea generale per calcolare un MST T su un grafo G è la seguente:

T=NULL;

while T non forma un albero di copertura;

do trova un arco (u,v) che sia **sicuro** per T;

T=T unito a $\{(u,v)\}$;

return T

Si definisce sicuro un arco che aggiunto ad un sottoinsieme di un albero di copertura minimo produce ancora un sottoinsieme di un albero di copertura minimo.

In seguito vedremo due tecniche per calcolare un arco sicuro. Entrambe queste tecniche utilizzano il concetto di taglio di un grafo.

Tagli di un grafo

Dato un grafo non orientato $G=(V,E)$, un taglio è una partizione di V in due sottoinsiemi.

Un arco attraversa il taglio se uno dei suoi estremi è in una partizione, e l'altro nell'altra.

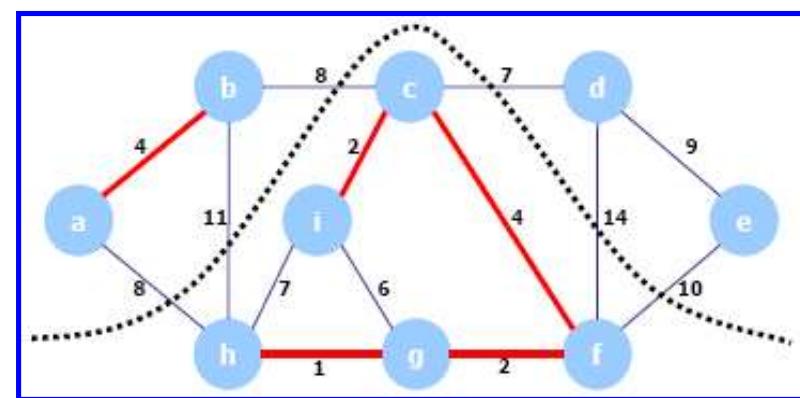
Un taglio rispetta un insieme A di archi se nessun arco di A attraversa il taglio.

Un arco si dice leggero se ha peso minimo tra gli archi che attraversano il taglio.

Esempio:

Siano gli archi in rosso un sottoinsieme A di E . Il taglio rispetta chiaramente A .

Se invece A è l'insieme degli archi adiacenti ai nodi d ed e , allora l'arco (d,c) è leggero per il taglio dato



Arco sicuro per G

Teorema:

- Sia $G=(V,E)$ un grafo non orientato e connesso con una funzione peso w a valori reali definita su E .
- Sia A un sottoinsieme di E contenuto in un qualche albero di copertura minimo per G .
- Sia $(S, V-S)$ un qualunque taglio che rispetta A .
- Sia (u,v) un arco leggero che attraversa $(S, V-S)$.
- Allora l'arco (u,v) è sicuro per A

In seguito analizziamo in dettaglio due algoritmi che calcolano un BST di un grafo formato da tutti archi sicuri calcolati utilizzando una tecnica greedy:

- **Algoritmo di Kruskal**
- **Algoritmo di Prim**

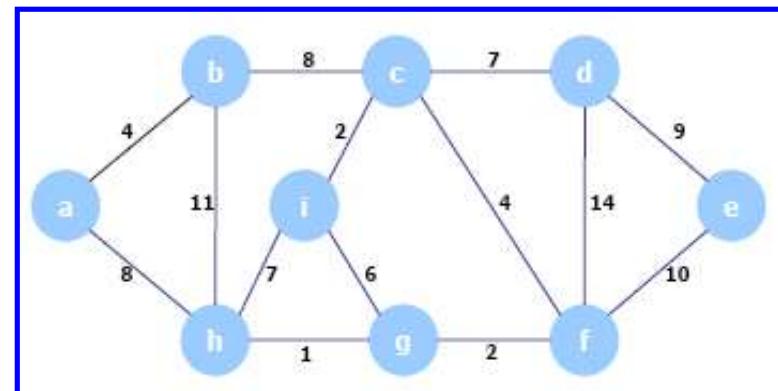
Algoritmo di Kruskal

MST-Kruskal(G,w)

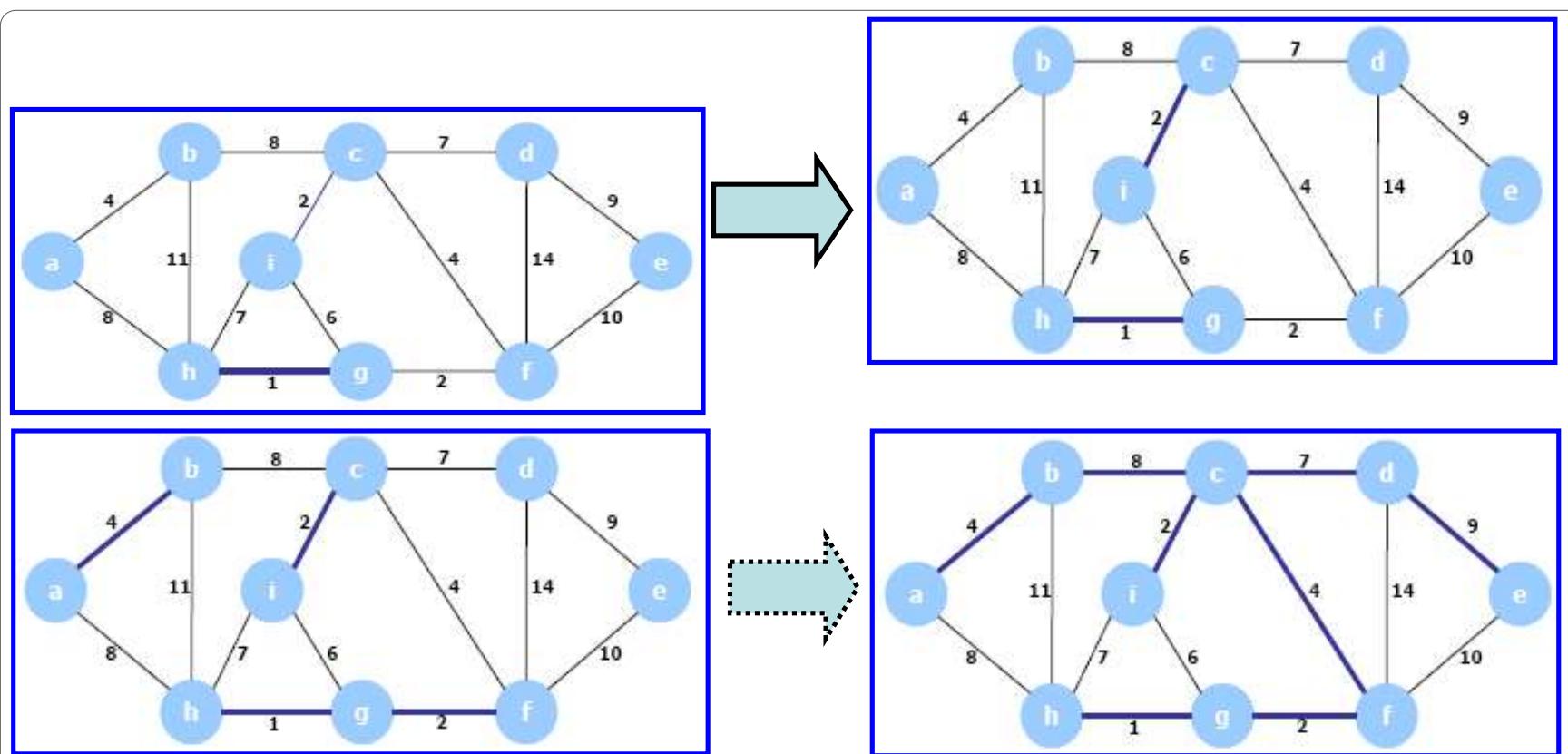
1. $A \leftarrow \emptyset$
2. **for** ogni vertice v di V
3. **do** Make-Set(V) $\backslash\backslash$ crea un insieme per ogni vertice V $\backslash\backslash$
4. ordina gli archi di E per peso w non decrescente
5. **for** ogni arco (u,v) di E , in ordine di peso non decrescente
6. **do if** Find-Set(u) \neq Find-Set(v)
7. $A \leftarrow A$ unito a $\{(u,v)\}$
8. Union(u,v) $\backslash\backslash$ fusione in un unico insieme degli insiemi di u e v
9. **return** A

Esempio:

Si consideri il seguente grafo. Inizialmente ogni insieme costituisce un insieme a se.



MST sull'esempio dato



La complessità dell'algoritmo di Kruskal dipende fondamentalmente dall'ordinamento degli archi pesati che prende tempo $O(E \log E)$.

Algoritmo di Prim

L'algoritmo di Prim parte da un nodo radice r ed espande ad ogni passo l'albero di copertura minimo A , sino a che questo non copre tutti i vertici.

Ad ogni passo viene aggiunto all'albero un arco leggero che collega un vertice in A ad un vertice in $V-A$.

Per la scelta dell'arco leggero si usa una coda di priorità.

Ad ogni istante la coda di priorità contiene tutti i vertici non appartenenti all'albero A .

La posizione di ciascun vertice v nella coda dipende dal valore di un campo chiave $\text{key}[v]$, corrispondente al minimo tra i pesi degli archi che collegano v ad un qualunque vertice in A .

Se v non è collegato a nessun vertice in A , allora $\text{key}[v]=\infty$.

Esempio:

MST-Prim(G,w,r)

1. $Q \leftarrow V[G]$
 2. **for** ogni u di Q
 3. **do** $\text{key}[u] \leftarrow \infty$ \\ chiave massima a tutti i nodi \\
 4. $\text{key}[r] \leftarrow 0$ \\ chiave minima alla radice \nwarrow
 5. $\pi[r] \leftarrow \text{NIL}$ \\ r non ha padre
 6. **while** $Q \neq \emptyset$
 7. **do** $u \leftarrow \text{Extract-Min}(Q)$
 8. **for** ogni v in $\text{Adj}[u]$
 9. **do if** v è in Q e $w(u,v) < \text{key}[v]$
 10. $\pi[v] \leftarrow u$
 11. $\text{key}[v] \leftarrow w(u,v)$

$O(\log V)$

$O(V)$
Usando
buildheap

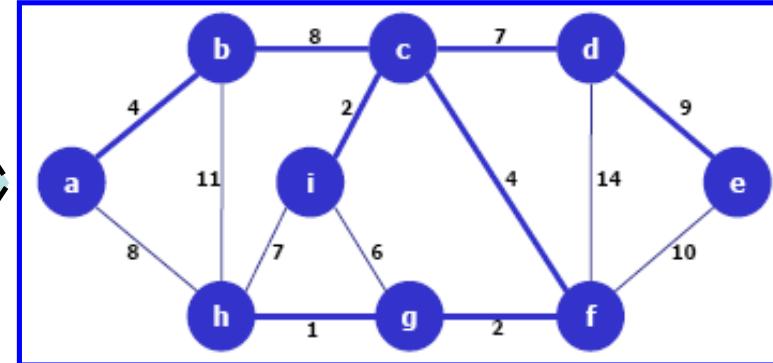
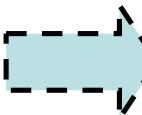
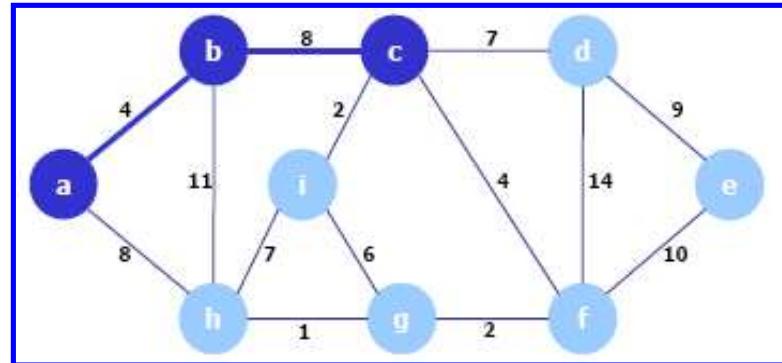
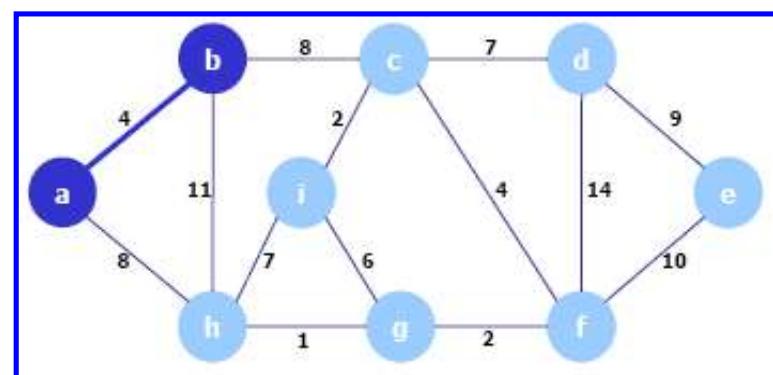
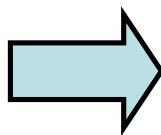
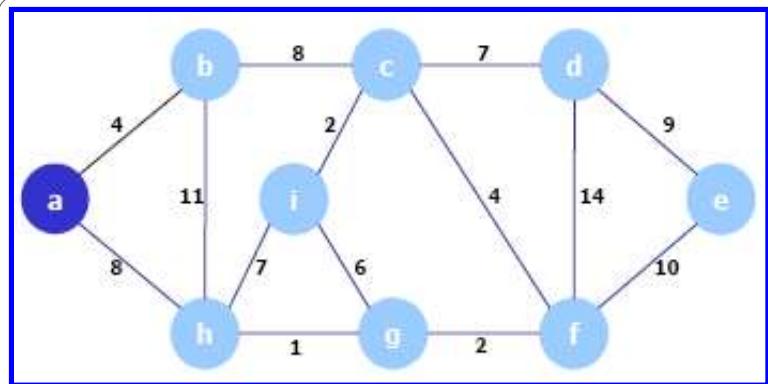
$O(\log V)$

$O(E)$

$O(\log V)$

Studi approfonditi dimostrano che la complessità asintotica dell'algoritmo di Prim è migliore dell'algoritmo di Kruskal

Esempio





Esercizio

**Implementare in linguaggio C
l'algoritmo di Prim**