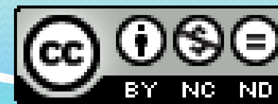


Fondamenti di Informatica

AA 2018/2019

Eng. Ph.D. Michela Paolucci

DISIT Lab http://www.disit.dinfo.unifi.it
Department of Information Engineering, DINFO
University of Florence
Via S. Marta 3, 50139, Firenze, Italy
tel: +39-055-2758515, fax: +39-055-2758570
michela.paolucci@unifi.it

UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

DISIT Lab
http://www.disit.dinfo.unifi.it
Department of Information Engineering, DINFO
University of Florence
Via S. Marta 3, 50139, Firenze, Italy

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Orario del Corso e Ricevimento

Insegnamento: FONDAMENTI DI INFORMATICA (A-L) (FonDiInf(A-L))

Ingegneria FIRENZE - A.A. 2018/2019 - 2° periodo

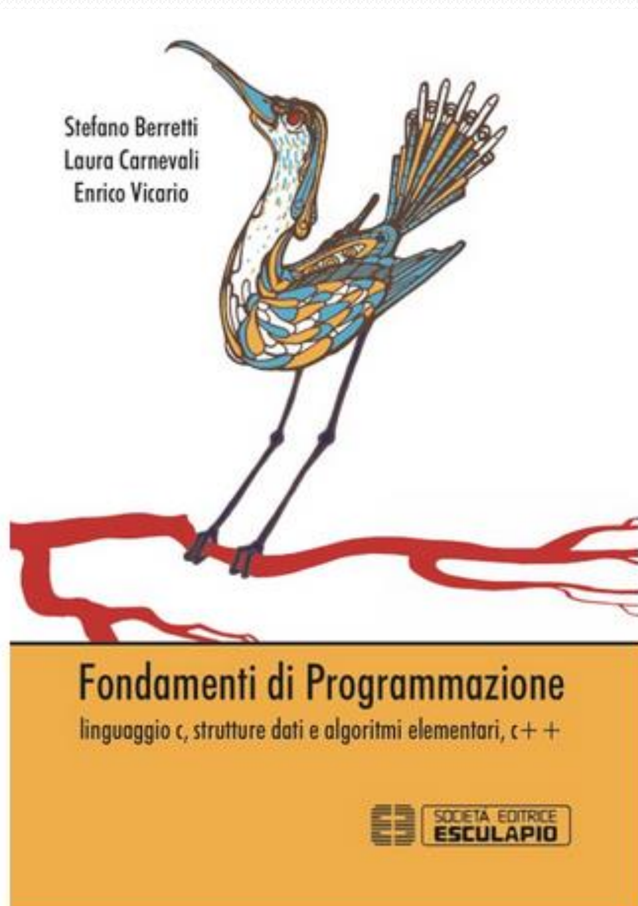
	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
08:15						
09:15						
10:15						
11:15	(Auditorium A - C.D.M.)					
12:15	(Auditorium A - C.D.M.)					
14:00					(Auditorium A - C.D.M.)	
15:00					(Auditorium A - C.D.M.)	
16:00					(Auditorium A - C.D.M.)	
17:00						
18:00						

Docenti: PAOLUCCI MICHELA

Legenda: C.D.M.: Centro Didattico Morgagni

- Il ricevimento si svolge su appuntamento contattando la docente via email:
  - michela.paolucci@unifi.it

Libro di testo



- Stefano Berretti, Laura Carnevali, Enrico Vicario
- Fondamenti di programmazione. Linguaggio C, strutture dati, algoritmi elementari, C++
- Editore: Esculapio
- ISBN: 9788893850513
- NOTA: ***Queste slide integrano e non sostituiscono quanto scritto sul libro di testo.***

Pagina del Corso

http://www.disit.org/drupal/?q=node/7020

Qui trovate:
- AVVISI
- Slide del corso
- Approfondimenti

Outline

- Strutture dati e algoritmi elementari
  - Liste
    - Rappresentazione in forma collegata con puntatori
    - Iterazione e ricorsione
  - Cenni sugli alberi
  - Algoritmi di ordinamento
    - Sequential sort

Strutture dati e algoritmi elementari

- La trattazione delle strutture dati permette di applicare i costrutti del C e di introdurre in modo concreto i seguenti concetti:
  - Separazione tra logica e implementazione di una struttura dati
  - Concetti di ricorsione e iterazione
  - Disciplina nella programmazione
  - Riuso di schemi e soluzioni

Liste (1)

- Una lista è una **successione finita di valori** di un tipo
- Come ogni tipo di dato, la lista è qualificata non solo dai **valori** che rappresenta, ma anche dalle **operazioni** che ci si eseguono sopra
- Queste operazioni sono:
  - Inizializzazione
  - Inserimento
  - Cancellazione
  - Visita
  - Ricerca

Liste (2)

- **Inserimento**: Aggiunta di un nuovo valore alla lista.
  - Esistono diverse varianti della operazione di inserimento:
    - in testa
    - in coda
    - in una posizione intermedia in base a qualche criterio di posizionamento (es: lista ordinata).
- **Cancellazione**: rimozione di un elemento dalla lista. Esistono diverse varianti della operazione di cancellazione:
  - in testa
  - in coda
  - su un elemento determinato in base qualche criterio (ad esempio il valore che contiene).

Liste (3)

- **Visita**: applicazione di un trattamento comune a tutti gli elementi della lista. Una visita può:
  - Stampare i valori della lista
  - Calcolare la somma dei valori
  - Identificare il massimo
  - Modificare il valore secondo una comune legge di trasformazione
  - Etc.
- **Ricerca:** determina se la lista contiene un elemento qualificato da una condizione sul valore. Tipicamente restituisce vero/falso, ma può anche restituire l'indirizzo (o comunque la posizione) dell'elemento cercato nel caso in cui questo sia presente nella struttura

Liste (4)

- **Inizializzazione:** non ha un significato proprio nell'elaborazione dell'informazione.
  - Non corrisponde a nessuna operazione sulla lista
  - Inizializza la particolare rappresentazione concreta della lista (si dice che 'fa da **costruttore** della lista')
  - Tipicamente serve ad asserire un qualche invariante che viene sfruttato e mantenuto nel corso dell'elaborazione
  - Spesso serve a identificare il primo e/o l'ultimo elemento della lista.

…ancora sul concetto di puntatore…

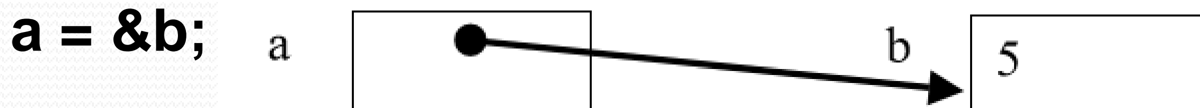
Concetto di puntatore

- I puntatori sono delle variabili che mantengono il valore dell'indirizzo di altre variabili o di **altri puntatori**
- Operatori: & e *
- Data la dichiarazione:

  int *a, b=5;
- Per convenzione il valore iniziale di un puntatore è NULL. NULL è una costante simbolica pari a 0

- Data una variabile è possibile risalire al suo indirizzo tramite l'operatore unario &. Per esempio data la variabile intera b SE si scrive &b, si intende fare riferimento all'indirizzo della variabile b. Quindi si può anche scrivere la seguente assegnazione:

  a = &b;



A cosa servono in pratica i puntatori

- Attraverso i puntatori una funzione può restituire valori multipli, così da superare il limite di un unico valore di ritorno ottenuto con l'istruzione **return**
- Affinché la funzione restituisca un valore di tipo intero (int) il parametro **formale** viene specificato del tipo puntatore a intero.

Esempio:

```
void SommaP(int A, int B, int *C) {//variabili formali
        *C = A + B;
        return;
}
main() {
        int a, b = 5, c = 2;
        SommaP(b,c,&a); //variabili attuali
        printf("A che valore punta a? %d", a); }
```

Puntatori e dati strutturati (1)

```
main() {//esempio di struttura persona
   struct persona pers, *p_pers;
   pers.nome = "Matteo";
   pers.cognome = "Fossi";
   pers.indirizzo = "Via S.Marta";
   pers.civico = 3;
   printf("La struttura persona pers contiene i seguenti dati:\n");
   printf("nome %s\ncognome %s\nindirizzo %s\ncivico %d\n", pers.nome,
          pers.cognome, pers.indirizzo, pers.civico);
   p_pers = &pers; //assegna un puntatore a pers
   printf("\n\nIl puntatore p_pers punta alla struttura persona pers che contiene
          i dati assegnati a pers:\n");
   printf("nome %s\ncognome %s\nindirizzo %s\ncivico %d\n", p_pers->
nome, p_pers->cognome, p_pers->indirizzo, p_pers->civico);
}
```

```
struct persona { //definizione
        char *nome;
        char *cognome;
        char *indirizzo;
        int civico;
};
```

- L’operatore ‘**.**’ permette l’accesso ad un membro di una struttura (**pers.nome**)
- L’operatore ‘**->**’ permette l’accesso a un membro della struttura puntata dal puntatore (**p_pers->nome**)

Puntatori e dati strutturati (2)

```
C:\WINDOWS\system32\cmd.exe
La struttura persona pers contiene i seguenti dati:
nome Matteo
cognome Fossi
indirizzo Via S.Marta
civico 3


Il puntatore p_pers punta alla struttura persona pers
che contiene i dati assegnati a pers:
nome Matteo
cognome Fossi
indirizzo Via S.Marta
civico 3
Premere un tasto per continuare . . .
```

Concetto di doppio puntatore (puntatore a puntatore) (1)

- I puntatori sono delle variabili che mantengono il valore dell'indirizzo di altre variabili o di **altri puntatori**
- Dichiarazione di un puntatore a un puntatore:

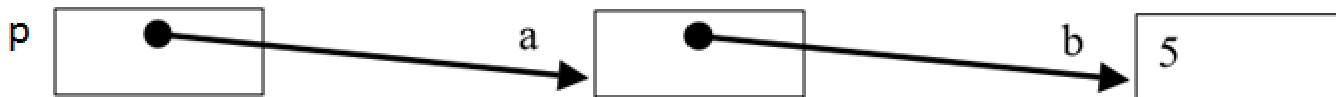
int **p;

Esempio:

int **p,*a, b=5;

a = &b;

p = &a;



Concetto di doppio puntatore (puntatore a puntatore) (2)

```
main(){
    int **p,*a, b=5;
    a = &b;
    p = &a;
    printf("Quanto vale il valore
    puntato dal puntatore a cui
    punta p? %d\n",**p);
}
```

Strutture1 (In debug) - Microsoft Visual Studio

File Modifica Visualizza Progetto Compilazione Debug Team St

Processo: [8408] Strutture1.exe Eventi ciclo di vita Thread:

Main.c* Main.c

Strutture1 (Ambito globale) main()

```
14  main() {
15
16      int **p, *a, b = 5;
17      a = &b;
18      p = &a;
19  }
```

75 %

Espressione di controllo 1

Nome	Valore	Tipo
b	5	int
a	0x005df9a0 {5}	int *
	5	int
p	0x005df9ac {0x005df9a0 {5}}	int * *
	0x005df9a0 {5}	int *
	5	int

Auto Variabili locali Espressione di controllo 1

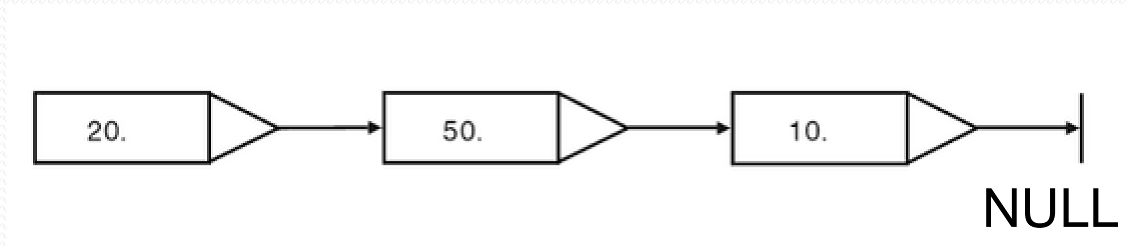
C:\Users\disit\Documents\Visual Studio 2015\Projects\Strutture1\Debug\Strutture1.exe

```
Quanto vale il valore puntato dal puntatore a cui punta p? 5
```

Liste: rappresentazione collegata con puntatori

Rappresentazione collegata con puntatori (1)

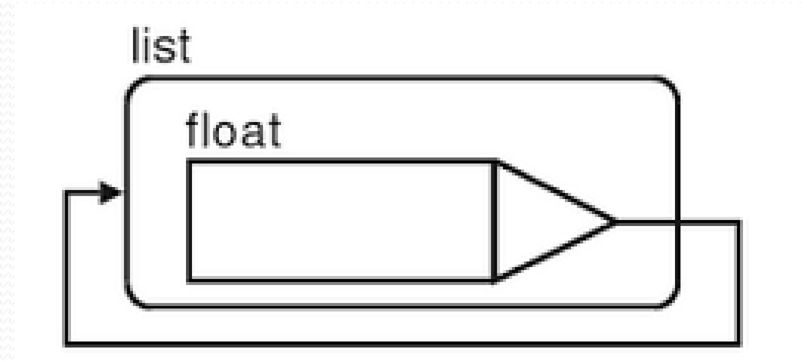
- Nella rappresentazione collegata con puntatori i valori sono memorizzati su elementi di memoria allocati separatamente in posizioni indipendenti
- Ciascun valore è associato a un indirizzo che identifica la locazione di memoria nella quale si trova il valore successivo
- Un indirizzo nullo codifica il termine della lista



Rappresentazione collegata con puntatori (2)

- Definizione del tipo:

```
struct list {
        float value;
        struct list * next_ptr;
};
```



Rappresentazione collegata con puntatori (3)

Struttura generale del programma – parte I

```
struct list {
        float value;
        struct list * next_ptr;
};
//definizioni per l'uso di booleani
typedef unsigned short int Boolean;
#define TRUE 1
#define FALSE 0

//dichiarazioni delle funzioni
void init(struct list ** ptrptr);   //inizializzazione lista
void pre_insert(struct list ** ptrptr, float value); //inserimento in testa
void ord_insert(struct list ** ptrptr, float value); //inserimento ordinato
void suf_insert(struct list ** ptrptr, float value); //inserimento in coda
void visit(struct list * ptr); //visita
Boolean search(struct list * ptr, float value, struct list ** ptrptr);
Boolean consume_first(struct list ** ptrptr, float * value_ptr); //cancellazione
```

Rappresentazione collegata con puntatori (4)

```
main() {
    struct list *lista;
    int size, res_search = FALSE, res_canc = FALSE;
    size = 5;
    float value, deleted_value;
    char selection[10];
    init(&lista);
    printf("Digita uno dei seguenti caratteri\nper fare le operazioni sulla
                         lista collegata con array e puntatori:\n");
    printf("-a per fare un inserimento in testa\n");
    printf("-b per fare un inserimento in coda\n");
    printf("-c per fare un inserimento ordinato\n");
    printf("-d per fare una ricerca\n");
    printf("-e per fare una vista\n");
    printf("-f per fare la cancellazione di un elemento\n");
    printf("-x per uscire dal programma\n");
    Boolean exit_required = FALSE;
    int inserimento = 0;
    do {…} while(exit_required == FALSE);
}
```

Dichiarazione inizializzazione:
void **init**(struct list ** ptrptr);

Struttura generale del programma – parte II

Rappresentazione collegata con puntatori (5)

```
…
do {//con struct list *lista;
    printf("\nChe operazione vuoi fare? ");
    scanf("%s", selection);
    switch (selection[0]) {
      case 'a':
          getvalue(&value);
          pre_insert(&lista, value); //inserimento in testa
          break;
      case 'b':
          getvalue(&value);
          suf_insert(&lista, value); //inserimento in coda
          break;
      case 'c':
          getvalue(&value);
          ord_insert(&lista, value); //inserimento ordinato
          break;
    …altri casi….
    }
} while(exit_required == FALSE);
…
```

Struttura generale del programma – parte III

Nota: dichiarazione inserimento in testa
void **pre_insert**(struct list ** ptrptr, float value);

Nota: dichiarazione inserimento in coda
void **suf_insert**(struct list ** ptrptr, float value);

Nota: dichiarazione inserimento in coda
void **ord_insert**(struct list ** ptrptr, float value);

Rappresentazione collegata con puntatori (6)

```
…
do {//con struct list *lista;
    printf("\nChe operazione vuoi fare? ");
    scanf("%s", selection);
    switch (selection[0]) {
        …altri casi….
        case 'd':
            getvalue(&value); //ricerca
            res_search = search(lista, value, &lista);
            if (res_search == TRUE)
                printf("Trovato!");
            else
                printf("Niente, riprova!");
            break;
        case 'e':
            visit(lista); //visita
            break;
        …altri casi….
    }
} while(exit_required == FALSE);
```

Struttura generale del programma – parte IV

Nota: dichiarazione inserimento in coda
Boolean **search**(struct list * ptr, float value, struct list ** ptrptr);

Nota: dichiarazione inserimento in coda
void **visit**(struct list * ptr);

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Rappresentazione collegata con puntatori (7)

```
…
do {//con struct list *lista;
    printf("\nChe operazione vuoi fare? ");
    scanf("%s", selection);
    switch (selection[0]) {
        …altri casi….
        case 'f':
            //cancellazione elemento in testa
            res_canc = consume_first(&lista, &deleted_value);
            if (res_canc == TRUE)
                printf("Cancellato elemento in testa: %f\n", deleted_value);
            else
                printf("Niente, riprova!");
            break;
        case 'x':
            exit_required = TRUE;
            break;
        default:
            notify_selection_failure(selection[0]);
    }
} while(exit_required == FALSE);
```

Struttura generale del programma – parte V

Nota: dichiarazione inserimento in coda
Boolean **consume_first**(struct list **ptrptr, float * value_ptr);

UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

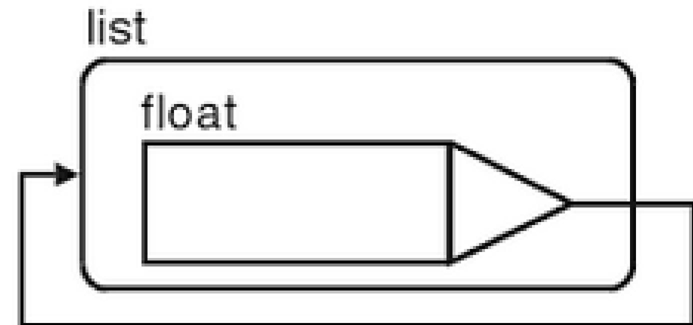
Rappresentazione collegata con puntatori (8)

- Una lista è una coppia composta di:
  - Un valore
  - Un puntatore ad una lista

```
//Con:
struct list {
        float value;
        struct list * next_ptr;
};
```

Definizione funzione
Di inizializzazione

```
//inizializzazione lista collegata con puntatori
void init(struct list ** ptrptr) {
        *ptrptr = NULL;
}
```



Rappresentazione collegata con puntatori (9)

Definizione
Inserimento in testa (1)

```
//inserimento in testa
void pre_insert(struct list ** ptrptr, float value) {
        struct list * tmp_ptr;
        tmp_ptr = *ptrptr;
        *ptrptr = (struct list *)malloc(sizeof(struct list));
        (*ptrptr)->value = value;
        (*ptrptr)->next_ptr = tmp_ptr;
}
```

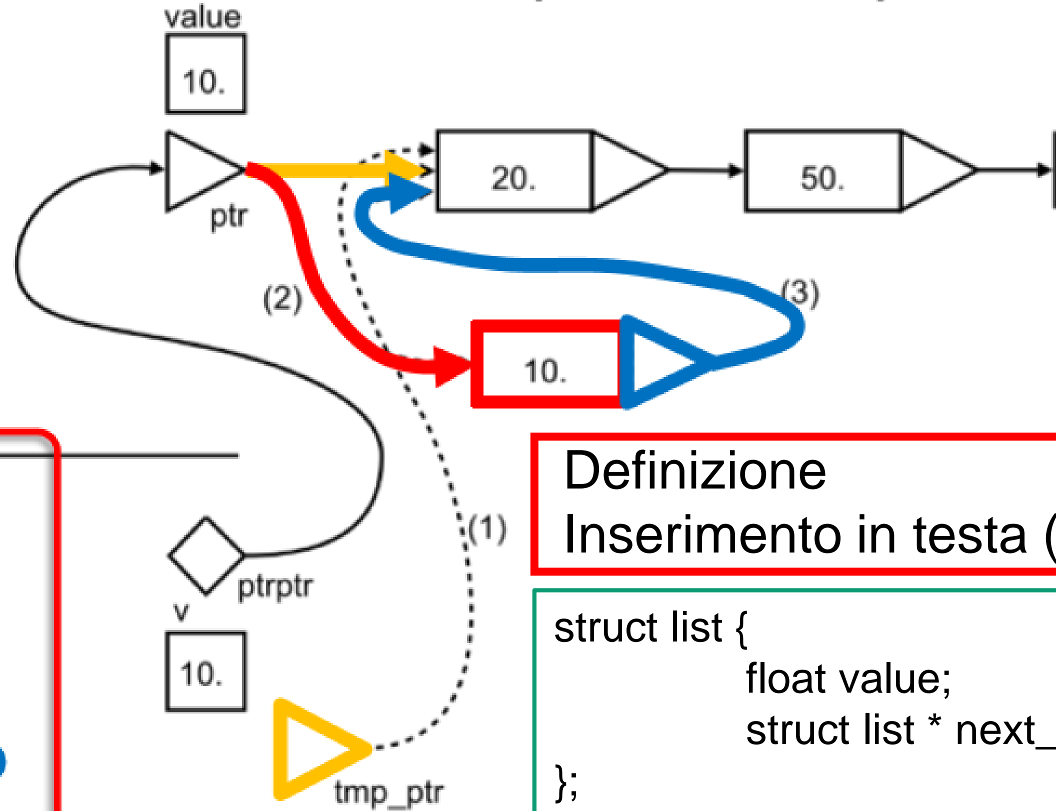
- L'inserimento prefisso deve modificare il puntatore di testa della lista. Per questo la funzione pre_insert() deve ricevere l'indirizzo del puntatore stesso
- E' significativo l'uso del doppio puntatore:
  - La funzione deve modificare l'indirizzo del primo elemento della lista e quindi deve conoscere l'indirizzo del puntatore che contiene l'indirizzo del primo elemento della lista

Rappresentazione collegata con puntatori (10)

Inserimento Prefisso (in Testa)

```
client( )
{
  float value;
  struct list * ptr;
  ...
  value=10.;
  pre_insert(&ptr,val);
  ...
}
```

```
pre_insert(struct list **ptrptr, float v)
{
   struct list * tmp_ptr;
   tmp_ptr=*ptrptr;      //(1)
   *ptrptr=malloc(...); //(2)
   (*ptrptr)->value=v;
   (*ptrptr)->next_ptr=tmp_ptr; //(3)
}
```



Definizione
Inserimento in testa (2)

```
struct list {
        float value;
        struct list * next_ptr;
};
```

- L'inserimento prefisso deve **modificare il puntatore di testa** della lista. Per questo la funzione `pre_insert()` deve ricevere l'indirizzo del puntatore stesso.

```
*ptrptr = (struct list *)malloc(sizeof(struct list));
```

Rappresentazione collegata con puntatori (11)

```
void suf_insert(struct list ** ptrptr, float value) {//inserimento in coda
/* Scorre il doppio puntatore fino a che questo punta un puntatore nullo che,
per costruzione è il campo next_ptr dell'ultimo elemento della lista, ovvero il
puntatore su cui applicare la malloc. Poi applica un inserimento in testa sulla
sotto-lista puntata dal doppio puntatore */
        while (*ptrptr != NULL) {
                ptrptr = &((*ptrptr)->next_ptr);
        }
        pre_insert(ptrptr, value);
}
```

Definizione
Inserimento in coda (1)

- La funzione riceve un doppio puntatore che contiene l'indirizzo del puntatore alla testa della lista
- **Se la lista è non vuota**, il doppio puntatore viene avanzato lungo la lista puntando il campo next_ptr dei successivi elementi della lista fino a puntare il primo puntatore che contiene il valore NULL
- Su questo puntatore viene richiamata la pre_insert(…) e:
  - Effettuata la malloc() e creato/inserito il nuovo elemento
- …

Rappresentazione collegata con puntatori (12)

```
void suf_insert(struct list ** ptrptr, float value) {//inserimento in coda
/* Scorre il doppio puntatore fino a che questo punta un puntatore nullo che,
per costruzione è il campo next_ptr dell'ultimo elemento della lista, ovvero il
puntatore su cui applicare la malloc. Poi applica un inserimento in testa sulla
sotto-lista puntata dal doppio puntatore */
        while (*ptrptr != NULL) {
                ptrptr = &((*ptrptr)->next_ptr);
        }
        pre_insert(ptrptr, value);
}
```

Definizione
Inserimento in coda (2)

- Se la lista non è vuota, il puntatore modificato è il campo next_ptr di un elemento della lista
- **Se la lista è vuota** o se l'inserimento deve avvenire in testa per rispettare l'ordine (caso di inserimento ordinato per il quale valgono le stesse osservazioni viste qui per l'inserimento in coda), il puntatore da modificare è ancora quello che contiene l'indirizzo del primo elemento, come nel caso di un inserimento in testa
- Per effettuare l'operazione, la funzione ha quindi bisogno di conoscere l'indirizzo del puntatore alla testa della lista

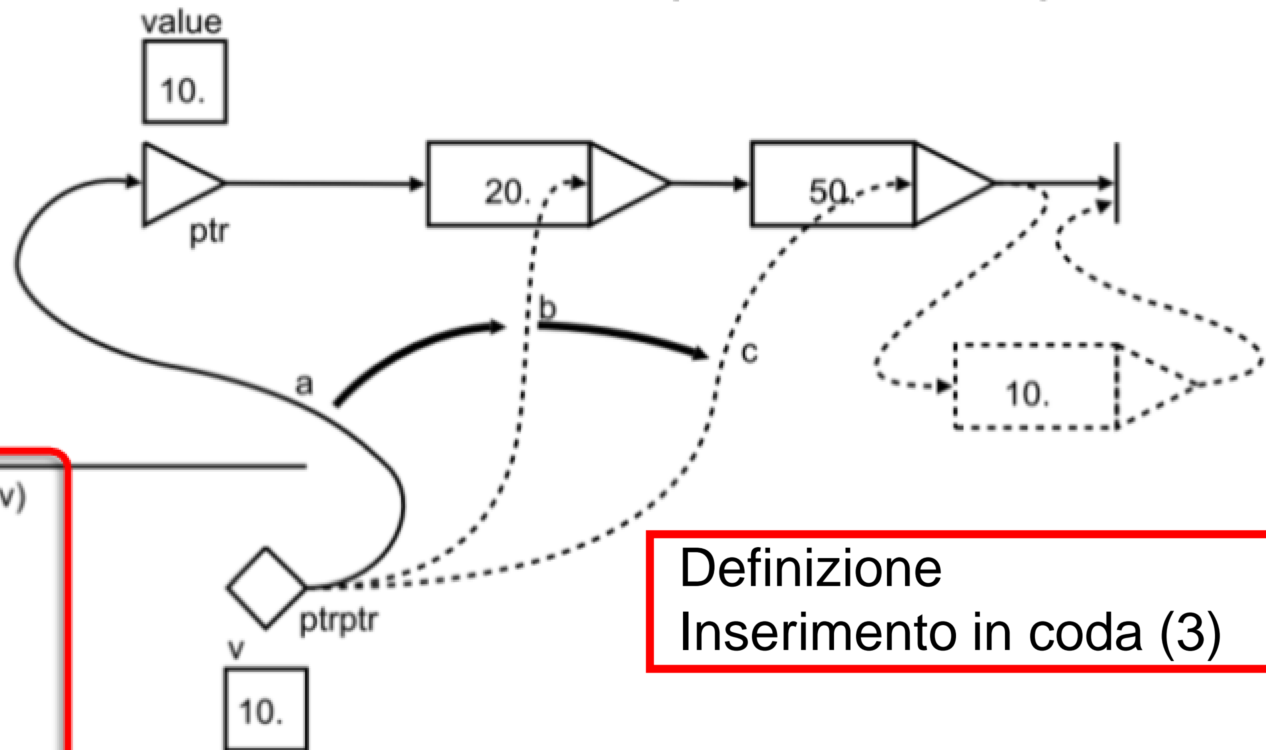
Rappresentazione collegata con puntatori (13)

Inserimento Suffisso (in Coda)

```
client( )
{
  float value;
  struct list * ptr;
  ...
  value=10.;
  suf_insert(&ptr,val);
  ...
}
```

```
suf_insert(struct list **ptrptr, float v)
{
   while(*ptrptr!=NULL)
   *ptrptr=&((*ptrptr)->next_ptr);

   pre_insert(ptrptr,v);
}
```



Rappresentazione collegata con puntatori (14)

```
//inserimento ordinato
void ord_insert(struct list ** ptrptr, float value) {
    /* inserimento in ordine. Simile all'inserimento in coda,
       salvo la condizione di arresto dell'avanzamento
       del doppio puntatore */
       while (*ptrptr != NULL && (*ptrptr)->value < value) {
              ptrptr = &((*ptrptr)->next_ptr);
       }
       pre_insert(ptrptr, value);
}
```

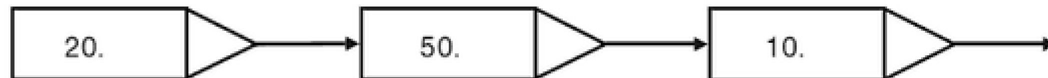
Definizione
Inserimento ordinato

- Valgono le stesse osservazioni fatte per l'inserimento in coda

Rappresentazione collegata con puntatori (15)

```
void visit(struct list * ptr) {
        while (ptr != NULL) {
                printf("%f\n", ptr->value);
                ptr = ptr->next_ptr;
        }
}
```

Definizione visita



Rappresentazione collegata con puntatori (15)

```
Boolean search(struct list * ptr, float value, struct list ** ptrptr) {
/* se esiste un elemento che contiene value, restituisce TRUE e
scrive in *ptrptr il puntatore all'elemento che contiene value.
Altrimenti restituisce FALSE e non modifica *ptrptr */
        Boolean found;
        found = FALSE; // x def. all'inizio non è trovato
        while (ptr != NULL && found == FALSE) {
                if (ptr->value == value) {
                        found = TRUE; // trovato?
                        *ptrptr = ptr;
                }
                else
                        ptr = ptr->next_ptr;
        }
        return found;
}
```

Definizione Ricerca

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Rappresentazione collegata con puntatori (16)

```
//cancellazione elemento in testa
Boolean consume_first(struct list ** ptrptr, float * value_ptr) {
        /* se la lista e' vuota restituisce FALSE;
        altrimenti restituisce TRUE, scrive in *value_ptr il valore
        nell'elemento in testa e lo ritorna prima di cancellarlo
        (deallocarlo) dalla lista */
        struct list *tmp_ptr = NULL;
        if (*ptrptr != NULL) { // se lista non vuota...
                *value_ptr = (*ptrptr)->value; //salva il valore della testa
                tmp_ptr=*ptrptr;               // salva il suo ptr
                *ptrptr = (*ptrptr)->next_ptr; // il ptrptr è il nuovo inizio
                free(tmp_ptr);                 // libera la memoria
                return TRUE;
        }
        else {
                return FALSE;
        }
}
```

Definizione Cancellazione

Rappresentazione collegata con puntatori (17)

```
void getvalue(float *value_ptr) {
        //acquisisce un float da tastiera (passaggio per
        puntatore)
        printf("Inserisci un valore: ");
        scanf("%f", value_ptr);
}

void notify_selection_failure(char selection) {
        //notifica il fallimento della selezione
        printf("\n%c Selezione non legale!!", selection);
}
```

Rappresentazione collegata con puntatori (18)

C:\WINDOWS\system32\cmd.exe

```
Digita uno dei seguenti caratteri
per fare le operazioni sulla lista collegata con array e puntatori:
-a per fare un inserimento in testa
-b per fare un inserimento in coda
-c per fare un inserimento ordinato
-d per fare una ricerca
-e per fare una vista
-f per fare la cancellazione di un elemento
-x per uscire dal programma

Che operazione vuoi fare? a
Inserisci un valore: 23

Che operazione vuoi fare? b
Inserisci un valore: 4

Che operazione vuoi fare? e
4.000000
23.000000

Che operazione vuoi fare? c
Inserisci un valore: 36

Che operazione vuoi fare? e
4.000000
23.000000
36.000000

Che operazione vuoi fare? f
Cancellato elemento in testa: 4.000000

Che operazione vuoi fare? e
23.000000
36.000000

Che operazione vuoi fare?
```

I File

I File

- I file sono dei contenitori di dati, in tali contenitori possono essere immagazzinate informazioni di varia natura e principalmente:
  - Lettura
  - Scrittura
- La natura del file può essere:
  - ASCII (testuale)
  - Binaria
- I file ASCII sono facilmente riconoscibili perché il loro contenuto può essere visionato ad esempio usando un editor e contiene Caratteri ‘comprensibili’:
  - lettere maiuscole e minuscole, segni di punteggiatura, numeri, simboli, caratteri ASCII che identificano i simboli di controllo, etc.

I nomi dei File (1)

- Ogni file deve avere un nome univoco, che permette cioè di distinguerlo da altri nello stesso contenitore di file
- Esempi:
  - test.txt
  - main.c
  - programma.cpp
  - relazione.doc
- La lunghezza del nome e i caratteri che possono essere usati per scrivere tale nome dipendono dal sistema operativo usato (Solitamente 8+3 per l'estensione)
- L'estensione del file è un modo immediato per dichiarare il tipo di file:
  - File testuali: txt
  - File di testo prodotti da editor: doc
  - File eseguibili: exe, com o bat

I nomi dei File (2)

- Nei sistemi operativi ad ogni file vengono associate diverse informazioni (**metadati**):
  - Data della creazione
  - Lunghezza
  - Tipo di file: archivio, lettura, scrittura, etc.
  - etc.
- I file possono essere contenuti nelle **directory**
- In ogni directory non possono esistere file con lo stesso nome
- Il percorso tra le directory per ‘raggiungere’ ogni file è detto **path**:
  - **assoluto**:
    - C:\Users\Documents\VisualStudio\Projects\Strutture1\Strutture1\prova.txt
  - **relativo:**
    - prova.txt

Le operazioni di base dei file (1)

- Le operazioni di base sui file sono:
  - **Apertura** del file:
    - Richiesta da parte del programma al sistema operativo di aprire un file su disco. Il file può essere già presente o meno:
      - Se il file NON è presente: può essere creato
  - **Inserimento** di **dati** nel file (operazione possibile solo se il file è stato aperto dal programma):
    - Inserimento di dati in posizioni specifiche
    - Inserimento di dati alla fine del file
  - **Lettura** dei dati dal file (possibile solo se il file è stato aperto dal programma):
    - Lettura in un punto specifico
    - Lettura a partire dall'inizio del file in modo sequenziale
  - **Chiusura** del file (possibile solo se il file è stato aperto):
    - Rilascio del file da parte del programma

Le operazioni di base dei file (2)

- In alcuni casi le operazioni di inserimento/scrittura e lettura/estrazione di dati dipendono dal tipo di file e da come questo è stato aperto
- Tramite degli strumenti del sistema operativo, è possibile impostare le **proprietà** di un file in modo che questo non possa essere manipolato in modo improprio
- Per esempio è possibile impedirne la cancellazione, la scrittura e anche la lettura

Apertura dei file - funzione fopen() (1)

- L'operazione di **apertura** dei file pregiudica le funzionalità del file stesso
- In C si possono aprire i file tramite la funzione di libreria **fopen()**
- FILE * **fopen** (char filename, char * modalita); //dichiarazione

```
main(){
        FILE *gt;
        gt = fopen("prova.txt", "w+");
        if (!gt)
                printf("Impossibile aprire il file");
        else {
                fprintf(gt,"Questo e' un file");
                printf("File aperto in scrittura!\n");
                fclose(gt);
                printf("File chiuso!");
        }
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Apertura dei file - funzione fopen() (2)

- La funzione di libreria **fopen**():
  - INPUT: prende in ingresso due parametri:
    - Nome del file
    - Modalità di apertura del file
  - OUTPUT: restituisce un identificativo del FILE da usare per le successive operazioni. Tale identificativo viene chiamato Handle (‘maniglia’)
  - Nell’esempio l’handle (gt) viene usato dalla funzione fprintf() per scrivere sul file
- La funzione **fprintf**():
  - INPUT: prende in ingresso due parametri:
    - Handle del file
    - Stringa da scrivere nel file

Apertura dei file - funzione fopen() (3)

```
    char *nome;
    char *cognome;
    char *indirizzo;
    int civico;
};*/

main() {
    FILE *gt;
    gt = fopen("prova.txt", "w+");
    if (!gt)
        printf("Impossibile aprire il file");
    else {
        fprintf(gt,"Questo e' un file");
        printf("File aperto in scrittura!\n");

    }
    fclose(gt);
    printf("File chiuso!");

/* //esempio semplice doppio puntatore
```

Output console:

```
File aperto in scrittura!
File chiuso!
```

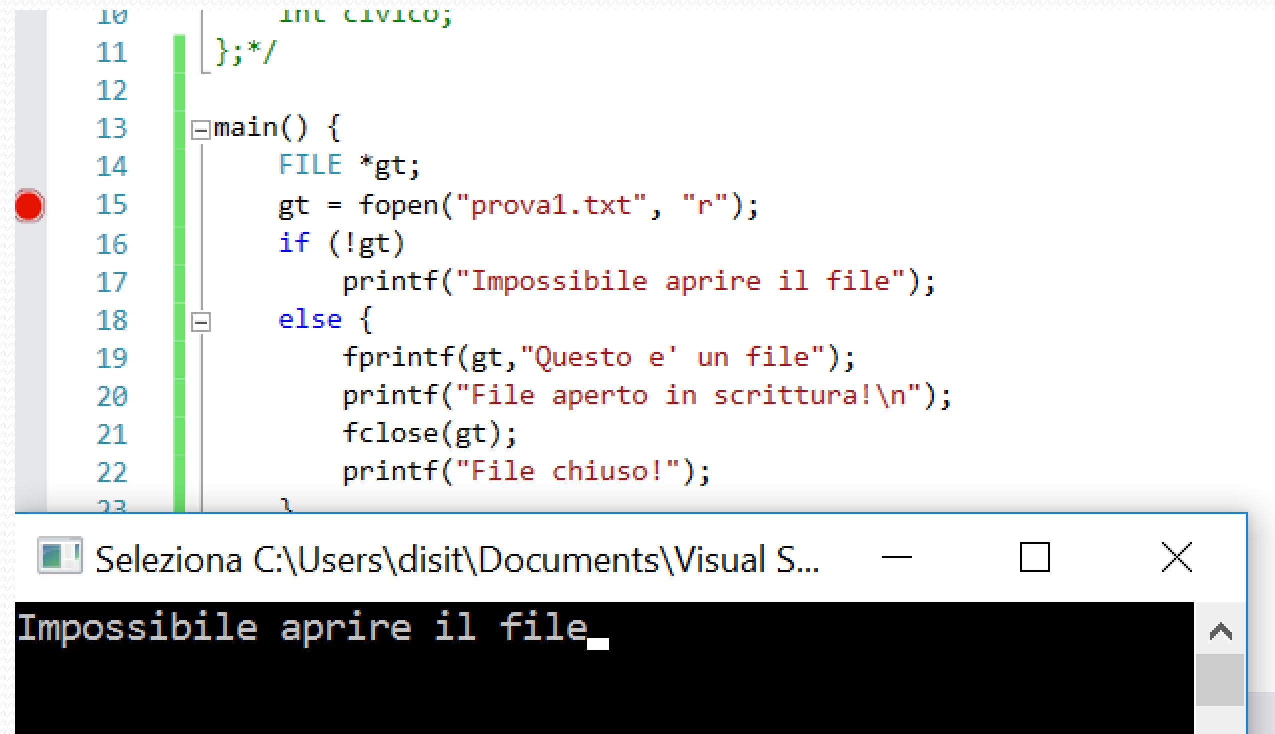
Nome	Ultima modifica	Tipo	Dimensione
Debug	24/04/2017 18:55	Cartella di file	
Main.c	24/04/2017 19:04	C Source	11 KB
prova.txt	24/04/2017 19:04	Documento di testo	1 KB
Strutture1.vcxproj	17/04/2017 11:03	VC++ Project	8 KB
Strutture1.vcxproj.filters	17/04/2017 10:33	VC++ Project Filters File	1 KB

prova.txt - Blocco note:

```
Questo e' un file
```

Apertura dei file - funzione fopen() (3)

- Quando l'operazione di apertura del file non ha successo, per esempio a causa della mancanza di spazio su disco (se voglio aprire un nuovo file) oppure per la mancanza di un file da leggere, allora la fopen() rende un NULL



Apertura dei file - funzione fopen() (4)

- La funzione fopen() presenta come secondo parametro una stringa, che indica la modalità di apertura del file in accordo con la seguente tabella

Modalita'	Descrizione
"r"	Apertura di un file testuale per lettura
"w"	Apertura di un file testuale per scrittura
"a"	Apertura di un file testuale per appendere alla sua fine
"r+"	Apertura di un file testuale per lettura e scrittura
"w+"	Apertura di un file testuale per scrittura, se il file esiste viene cancellato e ricreato da zero
"a+"	Apertura di un file testuale per scrittura alla fine, se il file non esiste viene creato

Salvataggio di dati in un file di testo (1)

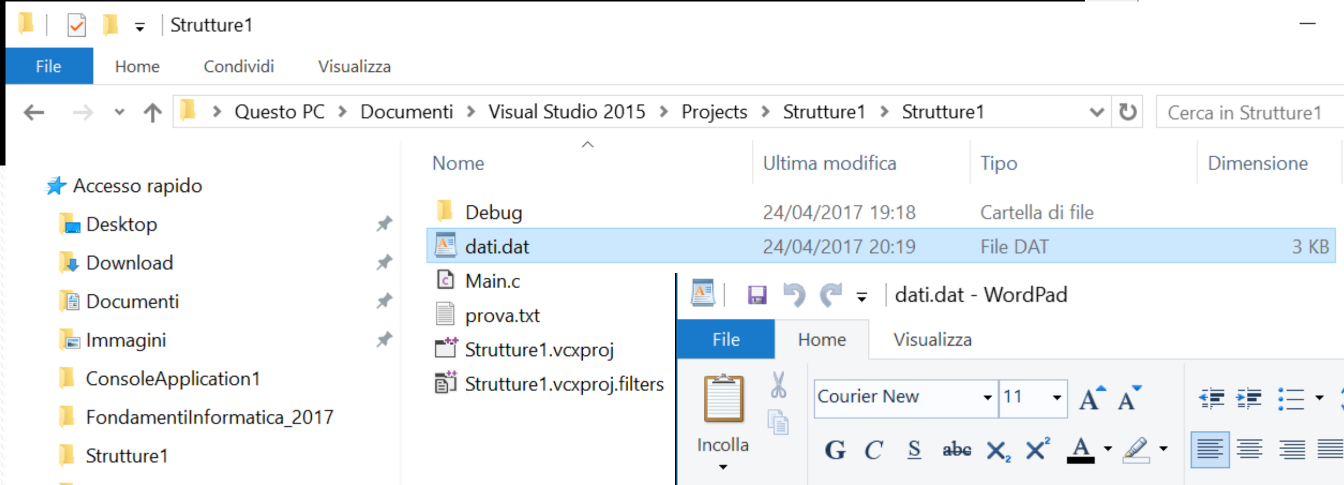
```
main() {
    int i, num=50;
    FILE *hand;
    float x, y;
    hand = fopen("dati.dat", "w+");//sovrascrivo il file se esiste
    if (!hand)
        printf("Impossibile aprire il file");
    else{
        fprintf(hand, "Numero di cicli: %d\n", num);
        for (i = 1; i<=num; i++) {
            x = i*0.3;
            y = sin(x)/x;
            fprintf(hand, "x pari a:%f | sin(x)/x pari a:%f\n", x, y);
        }
        fclose(hand);
        printf("File aperto in scrittura! ... e chiuso subito dopo!\n");
    }
}
```

FILE * **fopen** (char filename, char * modalita);

int fprintf(FILE *stream, char *formato, argomenti ...);

Si noti che:
char *formato è una stringa!!

Salvataggio di dati in un file di testo (2)



dati.dat - WordPad

```
Numero di cicli: 50
x pari a:0.300000 | sin(x)/x pari a:0.985067
x pari a:0.600000 | sin(x)/x pari a:0.941071
x pari a:0.900000 | sin(x)/x pari a:0.870363
x pari a:1.200000 | sin(x)/x pari a:0.776699
x pari a:1.500000 | sin(x)/x pari a:0.664997
x pari a:1.800000 | sin(x)/x pari a:0.541026
x pari a:2.100000 | sin(x)/x pari a:0.411052
x pari a:2.400000 | sin(x)/x pari a:0.281443
x pari a:2.700000 | sin(x)/x pari a:0.158289
x pari a:3.000000 | sin(x)/x pari a:0.047040
x pari a:3.300000 | sin(x)/x pari a:-0.047802
x pari a:3.600000 | sin(x)/x pari a:-0.122922
x pari a:3.900000 | sin(x)/x pari a:-0.176350
x pari a:4.200000 | sin(x)/x pari a:-0.207518
```

Lettura di file testuali (1)

- Per la lettura di dati da disco si usa la funzione fscanf() con modalità simili alla scanf()
- Il file deve essere aperto in modalità "**r+**" (il file viene aperto in lettura e deve esistere), altrimenti l'handle assumerà valore NULL

Lettura di file testuali (2)

```
#define NUM 10
main() {
    int i, numero;
    FILE *hand;
    float x, y, vx[NUM], vy[NUM];
    char nome[]= "dati.dat";
    //…scrittura del file (si veda esempio precedente)
    hand = fopen(nome, "r+");
    printf("\n\nFile aperto in lettura....\n");
    if (!hand)
        printf("Impossibile aprire il file");
    else{//lettura file testuali
        …
    }
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Lettura di file testuali (3)

```
…
if (!hand)
      printf("Impossibile aprire il file");
else {//lettura file testuali
   fscanf(hand, "Numero di cicli: %d\n", &numero);
   if (numero > NUM) {
      printf("Vettore dati troppo lungo");
      return(1);
   }
   printf("Il file %s contiene i seguenti %d elementi:\n", nome,
         numero);
   for (i = 0; i < numero; i++) {
         fscanf(hand, "x pari a:%f | sin(x)/x pari a:%f\n", &vx[i], &vy[i]);
         printf("x pari a:%f | sin(x)/x pari a:%f\n", vx[i], vy[i]);
    }
    fclose(hand);
    printf("File chiuso!\n");
}
…
```

*int fscanf(FILE *stream, char *formato,* argomenti *...);*

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Lettura di file testuali (3)

Con:

#define NUM 10

```
C:\WINDOWS\system32\cmd.exe
File aperto in scrittura!.... e chiuso subito dopo!


File aperto in lettura....
Il file dati.dat contiene i seguenti 10 elementi:
x pari a:0.300000 | sin(x)/x pari a:0.985067
x pari a:0.600000 | sin(x)/x pari a:0.941071
x pari a:0.900000 | sin(x)/x pari a:0.870363
x pari a:1.200000 | sin(x)/x pari a:0.776699
x pari a:1.500000 | sin(x)/x pari a:0.664997
x pari a:1.800000 | sin(x)/x pari a:0.541026
x pari a:2.100000 | sin(x)/x pari a:0.411052
x pari a:2.400000 | sin(x)/x pari a:0.281443
x pari a:2.700000 | sin(x)/x pari a:0.158289
x pari a:3.000000 | sin(x)/x pari a:0.047040
File chiuso!
Premere un tasto per continuare . . .
```

Outline

- Strutture dati e algoritmi elementari
  - Liste
    - Rappresentazione in forma sequenziale
    - Iterazione e ricorsione
  - Cenni sugli alberi
  - Algoritmi di ordinamento
    - Sequential sort

Concetto di Ricorsione

Concetto di Ricorsione

- Per funzione ricorsiva o algoritmo ricorsivo, si intende un procedimento/algoritmo definito in termini di se stesso
- Per esempio anche il prodotto di due numeri può essere definito in termini ricorsivi:
  - Prodotto Non ricorsivo:
    - P(q,r) = q*r
  - Prodotto scritto in modo ricorsivo:
    - Se (q>0), allora: P(q,r) = r + P(q-1, r)
    - Altrimenti P(q,r) = 0
- Il criterio di arresto permette di contenere il processo ricorsivo che altrimenti in teoria, andrebbe avanti all'infinito

Esempio di Ricorsione: il fattoriale (1)

- Un numero fattoriale viene rappresentato dal numero stesso con il punto esclamativo
- La definizione naturale è basata sul principio di induzione:
  - Se n<1, allora n! = 1 //passo base
  - Altrimenti, n! = n *(n-1)! //passo di induzione
- La possibilità di usare la ricorsione nei linguaggi di programmazione permette la traduzione diretta della definizione di un programma
- La funzione **fattoriale** traduce in modo diretto la definizione data, separando passo base e passo di induzione

Esempio di Ricorsione (2)

```
… //include…
long int fattoriale(int n); //dichiarazione funzione ricorsiva
main() {
    int numero;
    printf("Inserisci il numero di cui vuoi calcolare il fattoriale: ");
    scanf("%d", &numero);
    printf("\nEcco il fattoriale: %d\n", fattoriale(numero));
}
```

Se n<1, allora n! = 1 //passo base
*Altrimenti, n! = n *(n-1)! //passo di induzione*

```
long int fattoriale(int n) {//definizione funzione ricorsiva
        if (n < 1)
                return 1; //passo base
        else
                return n*fattoriale(n-1); //passo di induzione
}
```

Ricorsione nelle funzioni (1)

- Una funzione f() è ricorsiva quando essa delega parte della sua operazione ad una ulteriore **istanza di f() stessa.**
- Questo può avvenire in:
  - modo **diretto**, quando il corpo di f() **contiene un riferimento a f() stessa**
  - in modo **indiretto**, quando il corpo di f() contiene un riferimento a **una funzione che in modo diretto o indiretto fa riferimento a f()**

Ricorsione nelle funzioni (2)

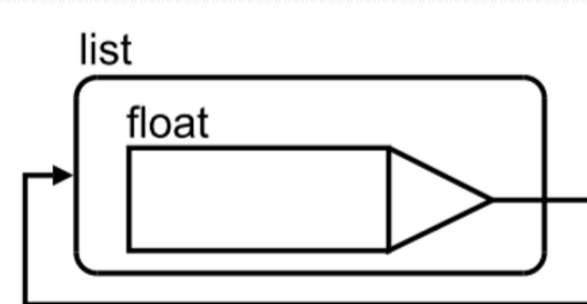
- La ricorsione si applica in modo naturale nell'esecuzione di una operazione applicata a un insieme di dati, come potrebbe essere quello di una operazione su una lista.

- In questo caso, una forma ricorsiva si ottiene in modo naturale attribuendo alla funzione la responsabilità di trattare un singolo dato e delegando ad una sua ulteriore istanza il trattamento di tutti gli altri.

…altro esempio di Ricorsione

- La definizione della struttura **struct list** che rappresenta un elemento di una lista è **ricorsiva** perché include al suo interno il campo **next_ptr** che a sua volta è un **puntatore ad una struttura di tipo struct list**:

```
struct list {
        float value;
        struct list * next_ptr;
};
```



Ricorsione nelle funzioni (3)

- Così facendo, le successive istanze nidificate ricevono la responsabilità di trattare un insieme di dati di dimensione strettamente decrescente fino a ridurre il problema al trattamento di un unico dato
- Una condizione di guardia arresta la nidificazione delle chiamate quando l'insieme dei dati da trattare è esaurito
- Il concetto è esemplificato in modo elementare nel caso della operazione di visita ricorsiva in una lista:

```
void visit_r(struct list * ptr){
    if(ptr!=NULL) {
        printf("%f \n",ptr->value);
        visit_r(ptr->next_ptr);
    }
}
```

```
//visita iterativa
void visit(struct list * ptr) {
    while (ptr != NULL) {
            printf("%f\n", ptr->value);
            ptr = ptr->next_ptr;
    }
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Ricorsione nelle funzioni (4)

- Dovendo stampare una sequenza di valori, **visit_r**() ne stampa uno e delega ad una ulteriore istanza di **visit_r**() stessa di stampare gli elementi a partire dal successivo di quello già stampato;
- La guardia dell'if termina la nidificazione quando l'operazione di visita viene invocata su una lista vuota.

```
void visit_r(struct list * ptr){
    if(ptr!=NULL) {
            printf("%f \n",ptr->value);
            visit_r(ptr->next_ptr);
    }
}
```

Esercizi

- Inserire gli elementi in una lista:
  - A) immettendoli da console
  - B) leggendoli da file di testo
- Stampare tramite visita ricorsiva gli elementi di una lista

Outline

- Strutture dati e algoritmi elementari
  - Liste
    - Rappresentazione in forma sequenziale
    - Iterazione e ricorsione
  - Cenni sugli alberi
  - Algoritmi di ordinamento
    - Sequential sort

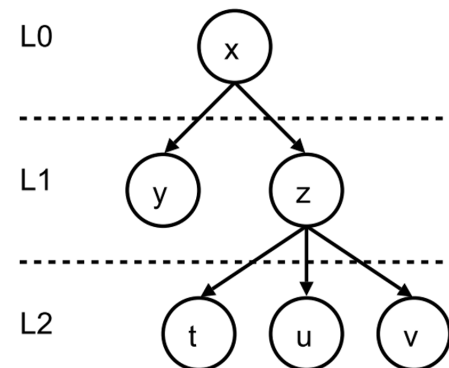
Cenni sugli alberi

Albero (1)

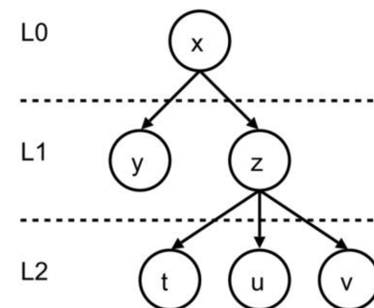
- L'albero è un insieme di elementi, detti **nodi**, sui quali è definita una relazione di discendenza
- L'albero gode delle seguenti caratteristiche:
  - Esiste un unico nodo, detto **radice**, che non ha predecessori
  - Qualsiasi altro nodo ha un unico predecessore
  - I nodi che non hanno successori sono detti **foglie**
  - Un nodo che non è né la radice né una foglia si dice **intermedio**

Albero (2)

- Nella caratterizzazione di un albero hanno rilevanza:
  - il **grado di uscita** e la **profondità** dei nodi
- Il grado di uscita di un nodo è il numero dei suoi successori diretti
- La profondità definisce la distanza di un nodo dalla radice ed è definita in modo ricorsivo:
  - la radice ha profondità 0
  - un generico nodo diverso dalla radice ha profondità pari a quella del suo predecessore aumentata di 1
- Per estensione, la **profondità di un albero** è il massimo delle profondità dei diversi nodi



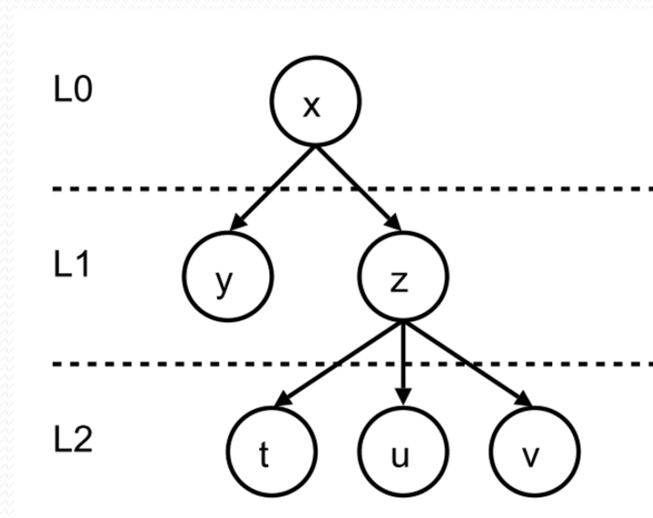
Albero (3)



- L'insieme dei nodi che si trovano ad una stessa profondità forma un livello
- Su un albero si eseguono sostanzialmente le stesse operazioni che si applicano anche ad una lista, con un paio di differenze:
  - su un albero non esiste un unico nodo terminale ma esiste invece una molteplicità di foglie, quindi l'operazione di inserimento in "coda" deve essere qualificata con un criterio specifico capace di selezionare quale tra le diverse foglie è il target dell'operazione di inserimento
  - l'inserimento sulla radice o su nodi intermedi tende a degenerare la struttura dell'albero verso una forma sequenziale, per cui gli inserimenti sono tipicamente eseguiti sulle foglie

Albero (4)

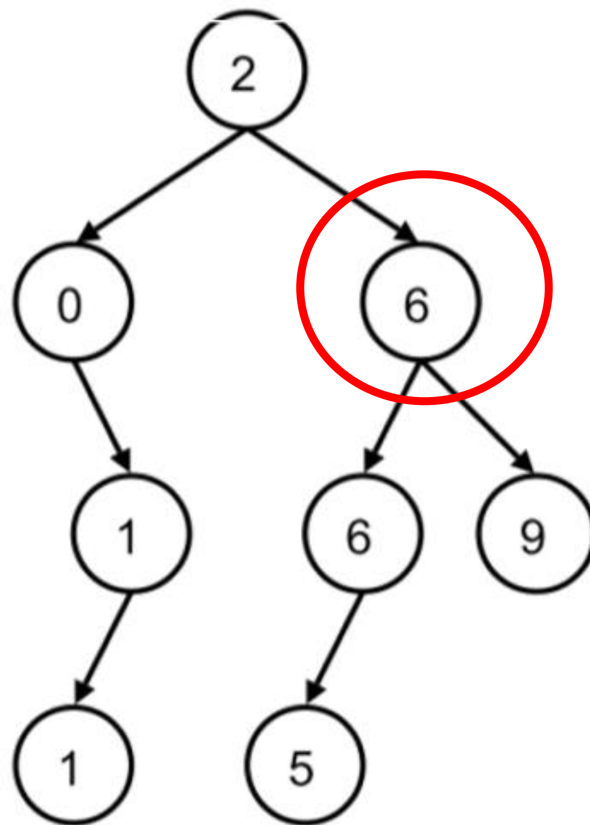
- Esempio:
  - l’elemento x è la radice e ha due successori y e z
  - z ha tre successori t, u e v
  - i nodi y, t, u e v sono foglie
  - z è un nodo intermedio e realizza il massimo grado di uscita (pari a 3)
  - l’albero è ripartito su tre livelli ed ha profondità 2



- NOTA: un albero in cui ciascun nodo ha al massimo un successore degenera nella forma di una lista

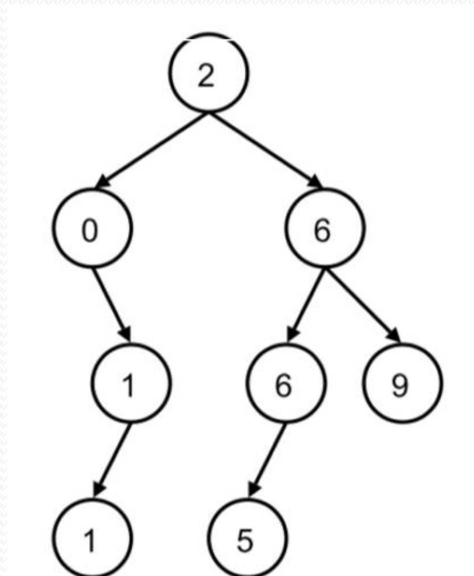
Alberi binari di ricerca

- Un albero binario è un albero sul quale:
  - ciascun nodo ha grado di uscita minore o uguale a 2
- Un albero binario si dice '**Albero binario di ricerca**' quando:
  - il valore codificato su ciascun nodo è maggiore o uguale del valore codificato sul figlio sinistro del nodo stesso e minore secco del valore codificato sul figlio destro

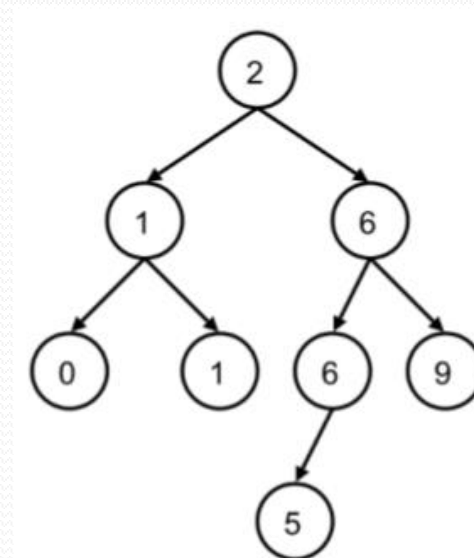


Albero binario di ricerca bilanciato (1)

- Un albero binario si dice anche bilanciato quando ogni nodo che si trova su un livello diverso dall'ultimo o dal penultimo ha due figli



Albero binario di ricerca NON Bilanciato



Albero binario di ricerca Bilanciato

Rappresentazione dell’Albero Binario

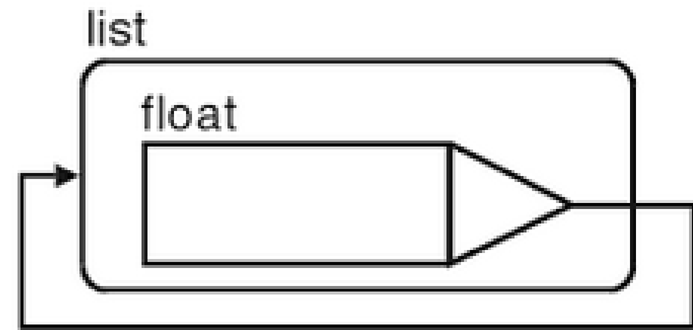
- L’albero binario può essere rappresentato:
  - in forma sequenziale
  - in forma collegata con puntatori

- Prima di passare alla rappresentazione in forma collegata con puntatori di un albero binario di ricerca, ricordiamo la rappresentazione in forma collegata con puntatori di una lista

Rappresentazione dell’Albero Binario in forma collegata con puntatori

Rappresentazione in forma collegata con puntatori di una LISTA

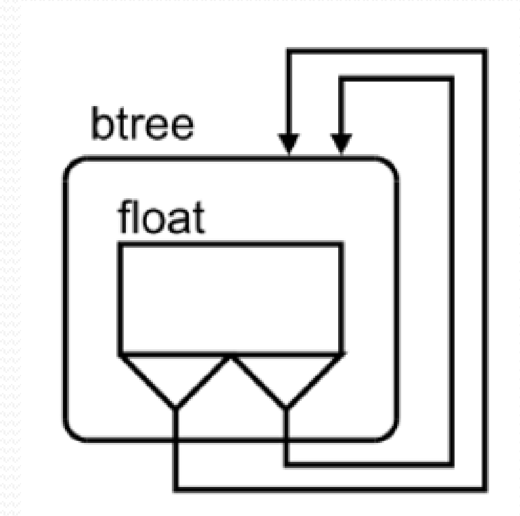
- Una lista è una coppia composta di:
  - Un valore
  - Un puntatore ad una lista




```
//implementazione in C:
struct list {
        float value;
        struct list * next_ptr;
};
```

Rappresentazione in forma collegata con puntatori di un Albero Binario (1)

- Un nodo di un Albero Binario è una TRIPLA composta di:
  - Un valore
  - **DUE** puntatore a due nodi

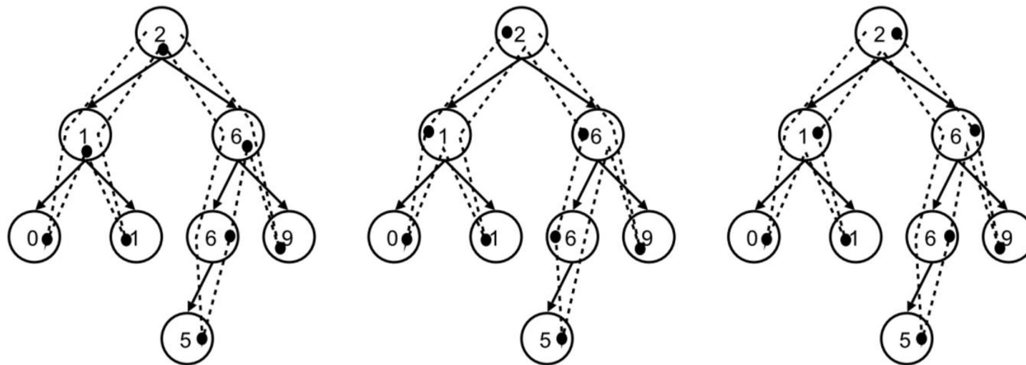



```
//implementazione in C:
struct btree {
        float value;
        struct btree * left_ptr;
        struct btree * right_ptr;
};
```

//NOTA: SE uno o entrambi i figli NON esistono, il puntatore corrispondente ha valore NULL

Ordine di visita di un Albero binario

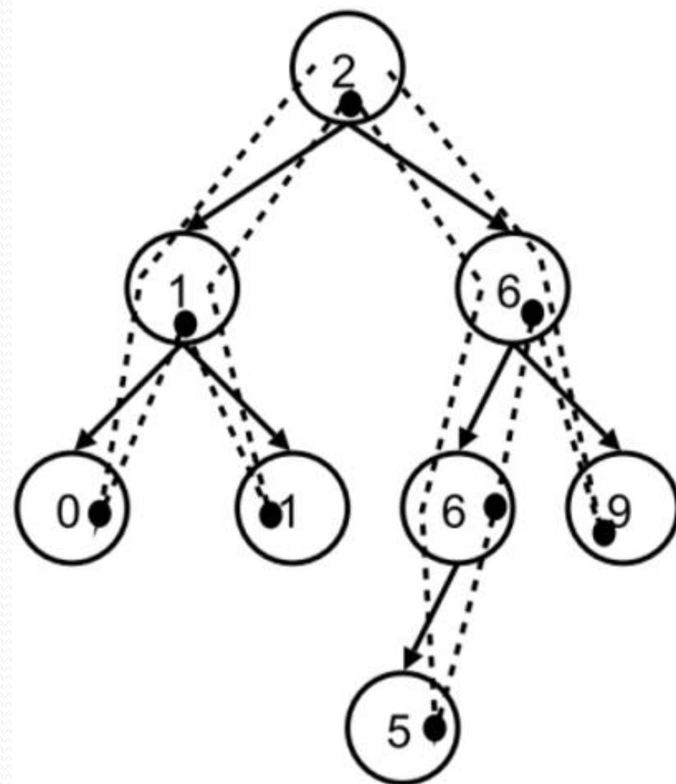
- L'ordine di visita dei nodi di un Albero Binario può essere in forma:
    - Simmetrica
    - Anticipata
    - Posticipata



- La linea tratteggiata rappresenta il flusso del controllo
- I punti marcati lungo la linea rappresentano il momento in cui si esegue l'operazione della visita del nodo (es: stampa del valore del nodo visitato) e il numero indica l'ordine di visita

Visita Simmetrica di un Albero binario

- Ordine:
  - Sottoalbero **sinistro**
  - **Radice** (es: operazione stampa)
  - Sottoalbero **destro**

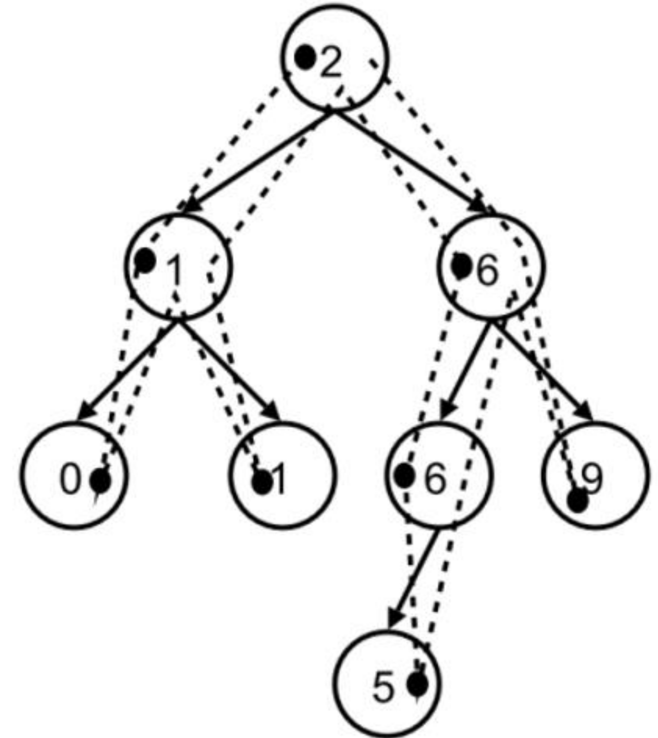


- La radice viene trattata:
  - **Solo DOPO** aver trattato tutti i nodi del sottoalbero **sinistro**, e **PRIMA** di trattare i nodi del sottoalbero **destro**
- Così facendo, SE l'albero è di ricerca, i valori sono stampati in ordine.

Un Albero Binario di Ricerca è t.c. il valore codificato su ciascun nodo è: i) maggiore o uguale del valore codificato sul figlio sinistro del nodo stesso; ii) minore secco del valore codificato sul figlio destro

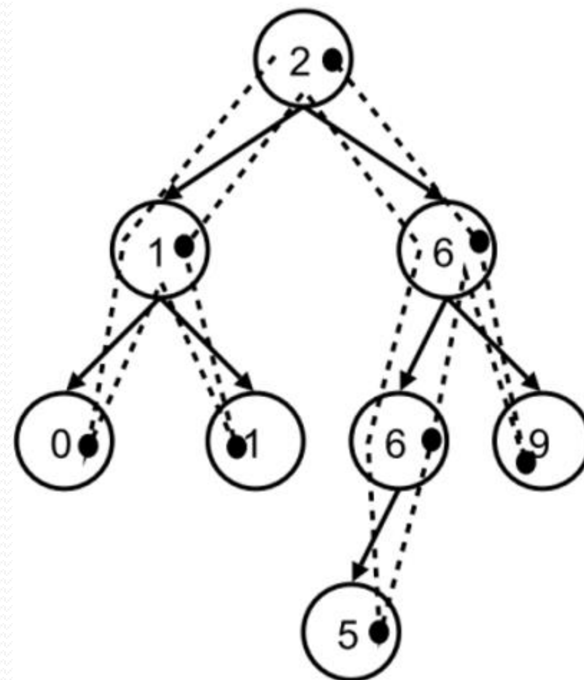
Visita anticipata di un Albero binario

- Ordine:
  - Radice (es: operazione stampa)
  - Sottoalbero sinistro
  - Sottoalbero destro



Visita Posticipata di un Albero binario

- Ordine:
  - Sottoalbero sinistro
  - Sottoalbero destro
  - Radice (es: operazione stampa)



Rappresentazione in forma collegata con puntatori di un Albero Binario (2)

```
typedef unsigned short int Boolean; //definizioni per l'uso di booleani
#define TRUE 1
#define FALSE 0

//struttura albero
struct btree {
            float value;
            struct btree * left_ptr;
            struct btree * right_ptr;
};

//dichiarazioni funzioni gestione albero binario
void init(struct btree ** ptrptr);
void insert_inorder(struct btree ** ptrptr, float value);
void visit_r(struct btree * ptr);
Boolean search(struct btree * ptr, float target);
//funzioni interazione console
void getvalue(float *value_ptr);
void notify_selection_failure(char selection);
```

Rappresentazione in forma collegata con puntatori di un Albero Binario (3)

```
main() {
    struct btree *bintree;
    int size, res_search = FALSE, res_canc = FALSE;
    size = 5;
    float value, deleted_value;
    char selection[10];
    init(&bintree);
    printf("Digita uno dei seguenti caratteri per operazioni su albero:\n");
    printf("-a per fare un inserimento (ordinato)\n");
    printf("-b per fare una visita\n");
    printf("-c per fare una ricerca\n");
    printf("-x per uscire dal programma\n");
    Boolean exit_required = FALSE;
    int inserimento = 0;
    …. do{…}
    while(exit_required == FALSE);
}
```

Nota: dichiarazione inizializzazione albero
void init(struct btree ** ptrptr);

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Rappresentazione in forma collegata con puntatori di un Albero Binario (4)

```
….. //struct btree *bintree;
do {
    printf("\nChe operazione vuoi fare? ");
    scanf("%s", selection);
    switch (selection[0]) {
      case 'a'://inserimento
          getvalue(&value);
          insert_inorder(&bintree, value);
          break;
      case 'b'://vista simmetrica
          visit_r(bintree);
          break;
      …..//altri casi
} while (exit_required == FALSE);

//altro…
```

Nota: dichiarazione inserimento ordinato
void **insert_inorder**(struct btree ** ptrptr, float value);

Nota: dichiarazione visita ricorsiva simmetrica
void **visit_r**(struct btree * ptr);

Rappresentazione in forma collegata con puntatori di un Albero Binario (5)

```
… do {//struct btree *bintree;
    … // altri casi
    case 'c'://ricerca
        getvalue(&value);
        res_search = search(bintree, value);
        if (res_search == TRUE)
                printf("Trovato!");
        else
                printf("Niente, riprova!");
        break;
        break;
    case 'x': //uscita
        exit_required = TRUE;
        break;
    default:
        notify_selection_failure(selection[0]);
    }
} while (exit_required == FALSE);   //….
```

Nota: dichiarazione inserimento in coda
Boolean **search**(struct btree * ptr, float target)

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

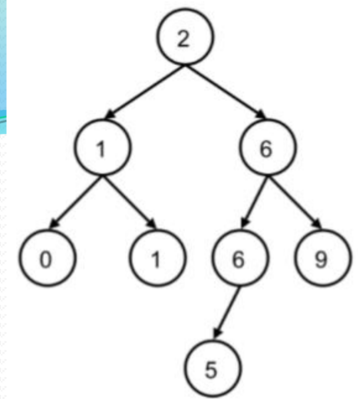
Inizializzazione

Con:

```
struct btree {
        float value;
        struct btree * left_ptr;
        struct btree * right_ptr;
};


void init(struct btree ** ptrptr) {
        *ptrptr = NULL;
}
```

Inserimento ordinato ricorsivo in un Albero Binario di ricerca



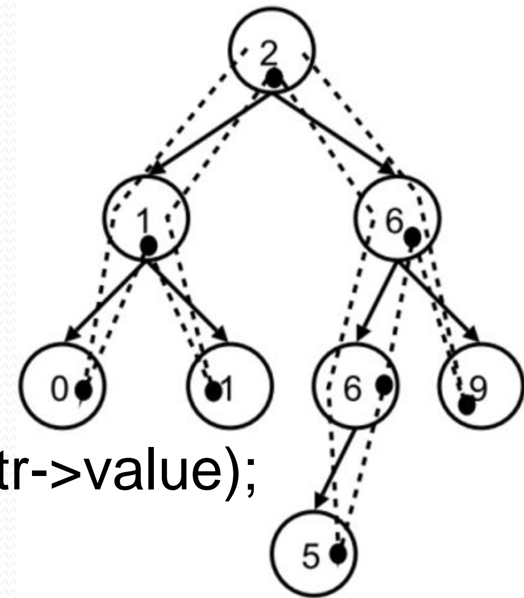

```
void insert_inorder(struct btree ** ptrptr, float value) {
        if (*ptrptr != NULL) {    // NON è una  foglia
                if (value <= (*ptrptr)->value) //si discende l'albero
                        insert_inorder(&((*ptrptr)->left_ptr), value);
                else
                        insert_inorder(&((*ptrptr)->right_ptr), value);
        }
        else { // foglia raggiunta… aggiungi elemento
                (*ptrptr) = (struct btree *)malloc(sizeof(struct btree));
                (*ptrptr)->value = value;     // metti valore
                (*ptrptr)->left_ptr = NULL;  // inizializza figli…
                (*ptrptr)->right_ptr = NULL; // a NULL.
        }
}
```

Visita Simmetrica di un Albero binario in forma ricorsiva

```
//visita simmetrica
void visit_r(struct btree * ptr) {
        if (ptr != NULL) {
                printf("Cerco il figlio sinistro... ");
                visit_r(ptr->left_ptr);

                printf("\nStampo il valore: %f \n", ptr->value);

                printf("Cerco il figlio destro... ");
                visit_r(ptr->right_ptr);
        }
        printf("Risalgo... ");
}
```



La radice viene trattata:
Solo DOPO** aver trattato tutti i nodi del sottoalbero **sinistro**, e **PRIMA** di trattare i nodi del sottoalbero **destro

Ricerca su un Albero Binario (1)

- Su un albero binario di ricerca è possibile eseguire una ricerca in modo efficiente, evitando di visitare esaustivamente tutti i nodi
- L'algoritmo di ricerca è descritto in forma ricorsiva:
  - SE la radice contiene il valore da cercare, la ricerca termina con successo
  - SE il valore della radice è maggiore (minore) del target allora questo può trovarsi solo nel sottoalbero sinistro (destro) per cui l'esito della ricerca è quello della ricerca sul sottoalbero sinistro (destro)
  - SE la ricerca fallisce appena viene raggiunto un sottoalbero vuoto

Ricerca in un Albero binario in forma ricorsiva

```
Boolean search(struct btree * ptr, float target) {
    if (ptr != NULL) { // NON è una foglia
        if (ptr->value == target) {    // trovato
                return TRUE;
        }
        else { // NON trovato
                if (target < ptr->value)  // vai a sx
                        return search(ptr->left_ptr, target);
                else // oppure vai a dx
                        return search(ptr->right_ptr, target);
        }
    }
    else // se NULL => foglia… quindi nn trovato
        return FALSE;
}
```

Funzioni interazione console

```
void getvalue(float *value_ptr) {
	//acquisisce un float da tastiera (passaggio per puntatore)
		printf("Inserisci un valore: ");
		scanf("%f", value_ptr);
}

void notify_selection_failure(char selection) {
		//notifica il fallimento della selezione
		printf("\n%c Selezione non legale!!", selection);
}
```

NOTE: Inserimento in coda in un Albero Binario

- L’inserimento in coda a sua volta pone il problema di stabilire quale sia la foglia in cui effettuare l’inserimento (ambiguità)
- Nel caso di un albero binario di ricerca il problema è risolto inserendo ogni nuovo valore **nell’unica foglia** che mantiene l’albero in **ordine**:
  - Si discende l’albero a partire dalla radice
  - Se l’albero è non vuoto l’inserimento è delegato ad una chiamata ricorsiva che opera sull’unico dei due sottoalberi abilitati a contenere il valore da inserire
  - Quando la ricorsione raggiunge una foglia viene effettuato l’inserimento

Esercizi

- Implementare le visite anticipata e posticipata forma ricorsiva

Algoritmi di ricerca

Algoritmi di ricerca su array (1)

- Problema: Data una sequenza di elementi, verificare se un elemento, detto ‘chiave’, fa o meno parte di tale sequenza
- Una sequenza di elementi può essere memorizzata in un array
- Gli elementi (supponiamo ad esempio che siano interi) possono essere inseriti in un array in modo:
  - Non Ordinato

    int A[5] = {122, 2, 30, 40, 12};
  - Ordinato

    int B[5] = {2, 12, 30, 40, 122};

Algoritmi di ricerca su array (2)

- In base alla modalità in cui gli elementi sono memorizzati sull’array, occorre eseguire un tipo di ricerca ad hoc:
  - Se l’array NON è ordinato: si effettua una **ricerca lineare (o sequenziale)**
  - Se l’array è ordinato: si effettua una **ricerca binaria** (o **dicotomica**)

Ricerca lineare (sequenziale) su vettori (1)

- Problema: Data una sequenza NON ORDINATA di elementi memorizzati in un array, verificare su un elemento, detto ‘chiave’, fa o meno parte di tale sequenza

Algoritmo ITERATIVO

- Gli elementi dell’array vengono analizzati in sequenza:
  - ogni elemento dell’array viene confrontato con l’elemento da ricercare (chiave)
  - Ovvero: gli elementi dell’array vengono scanditi uno dopo l’altro sequenzialmente
- Quando si trova un elemento uguale alla chiave la ricerca termina

NOTE:

- Se l’elemento viene trovato prima di raggiungere la fine della sequenza non sarà necessario proseguire la ricerca
- COSTI: l’algoritmo prevede che al più tutti gli elementi dell’array vengano confrontati con la chiave

Ricerca lineare (sequenziale) su vettori (2)

```
….inclusioni librerie… //definizioni per l'uso di booleani
typedef int Boolean;
#define TRUE 1
#define FALSE 0
//dichiarazione funzione ricerca su array di interi
Boolean sequential_search(int *V, int N, int key);
main() {
        int A[] = { 10, 35, 8, 9, 123 }, result = 0;
        int chiave = 11; //valore da cercare
        result = sequential_search(A, 5, chiave);
        if (result == TRUE)
                printf("Trovato: %d!", result);
        else
                printf("NON trovato!");
}
```

Ricerca lineare (sequenziale) su vettori (2)

```
Boolean sequential_search(int *V, int N, int key) {
    Boolean found;
    int count;
    found = FALSE;
    count = 0;
    while (found == FALSE && count < N) {
        if (V[count] == key) //caso di valori interi
                found = TRUE;
        else
                count++;
    }
    return found;
}
```

Algoritmo di ricerca binaria (1)

- L'algoritmo di ricerca lineare richiede che al più tutti gli elementi dell'array vengano confrontati con la chiave. Questo è necessario nel caso di sequenza NON ORDINATA
- Se la sequenza su cui si effettua la ricerca è ORDINATA si può usare un algoritmo di ricerca molto più efficiente che cerca la chiave sfruttando il fatto che gli elementi della sequenza sono già disposti in un dato ordine
  - Esempi di sequenze ordinate: elenco telefonico, agenda, etc.
- In questi casi si usa un **algoritmo di ricerca binaria** (o **dicotomica**) che è più efficiente perché riduce lo spazio di ricerca

Algoritmo di ricerca binaria (1)

- Problema: Data una sequenza **ORDINATA** di elementi memorizzati in un array, verificare su un elemento, detto ‘chiave’, fa o meno parte di tale sequenza

Algoritmo **RICORSIVO**

- La chiave viene confrontata con l’elemento in posizione mediana dell’array:
  - SE la chiave è uguale al valore contenuto in posizione mediana, allora la ricerca termina con successo
  - SE la chiave è maggiore del valore contenuto in posizione mediana, allora la ricerca prosegue nella metà destra dell’array
  - SE la chiave è minore del valore contenuto in posizione mediana, allora la ricerca prosegue nella metà destra dell’array

Ricerca Binaria (o dicotomica) (1)

```
….inclusioni librerie…
//definizioni per l'uso di booleani
typedef int Boolean;
#define TRUE 1
#define FALSE 0

Boolean b_search(int *V, int N, int key); //dichiarazione…

main() { //array NON ordinato
        int B[] = { 8, 9, 10, 35, 123 };
        result = 0;
        result = b_search(B, 5, 123);
        if (result == TRUE)
                printf("Trovato!");
        else
                printf("NON trovato!");
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Ricerca Binaria (o dicotomica) (2)

```
Boolean b_search(int *V, int N, int key) {
        if (N > 0) {
                if (V[N / 2] == key) //N/2 rende la parte intera
                        return TRUE;
                if (V[N / 2] > key)
                   return b_search(V, N/2, key);
                else
                   return b_search(&V[N/2+1], N-(N/2)-1, key);
        }
        else
                return FALSE;
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Esercizi

- Prendere da console gli elementi da inserire in un array (supporre che siano interi)
  - Supporre che l'array sia ORDINATO / NON ORDINATO
  - Chiedere da Console la chiave di ricerca
  - Stampare il risultato della ricerca
- Prendere da console gli elementi da inserire in un array
  - Determinare SE l'array è ordinato
  - Applicare uno dei due algoritmi in base al tipo di ordinamento degli elementi nell'array
- Gestire il caso di Array di float

Outline

- Strutture dati e algoritmi elementari
  - Liste
    - Rappresentazione in forma sequenziale
    - Iterazione e ricorsione
  - Cenni sugli alberi
  - Algoritmi di ordinamento
    - Sequential sort

Algoritmi di ordinamento

Ricerca Sequenziale

Sequential-Sort (1)

- Problema: Data una sequenza di elementi memorizzati in un array, ordinare i valori dal più piccolo al più grande
- Una sequenza di elementi può essere memorizzata in un array

- Algoritmo Sequential-Sort:
  - seleziona di volta in volta il numero minore nella sequenza di partenza e lo sposta nella sequenza ordinata
  - Di fatto la sequenza viene suddivisa in due parti:
    - la **sotto-sequenza ordinata,** che occupa le prime posizioni dell'array,
    - la **sotto-sequenza da ordinare**, che costituisce la parte restante dell'array.

Sequential-Sort (2)

Algoritmo

- Dovendo ordinare un array a di lunghezza N, si fa scorrere l'indice j=0..n-1 ripetendo i seguenti passi:
  - si cerca il più piccolo elemento della sotto-sequenza a[i] con i=j+1..n-1
  - si scambia questo elemento con l'elemento j-esimo

Concetto di swap (scambio)

```
#include <stdio.h>
main()
{
    int a, b, temp;
    printf("Inserisci due interi su cui eseguire lo swap:\n");
    scanf("%d %d", &a, &b);
    printf("Hai inserito a=%d e b=%d!Adesso li inverto...\n", a, b);

    temp = a; //salvo il vecchio valore di a in temp
    a = b; //assegno ad il valore di b
    b = temp; //assegno a b il vecchio valore di a, che adesso è in temp

    printf("Ecco i valori invertiti:\n  a = %d;  b = %d\n", a, b);
}
```

Sequential-Sort (3)

```
#include <stdio.h>
void Selection_Sort(int *V, int N);
void swap(int *A, int *B);
main() {
    int A[] = { 10, 35, 8, 9, 897, 0, 99, 123 }, i;  //array NON
    ordinato
    printf("\n\nStampo array inziale NON ordinato (array A): \n");
    for (i = 0; i < 8; i++) {
        printf("%d ", A[i]);
    }
    Selection_Sort(A, 8);
    printf("\n\nStampo gli elementi in Ordine dell'array A (dopo
        esecuzione algoritmo): \n");
    for (i = 0; i < 8; i++) {
        printf("%d ", A[i]);
    }
}
```

Sequential-Sort (3)

```
void Selection_Sort(int* a, int n) { //o sequential sort
     int i, j, iMin;
     for (j = 0; j < n - 1; j++) {//ciclo esterno scandisce array da ordinare
          iMin = j; //inizializzazione indice corrispondente al valore minimo
          for (i = j + 1; i < n; i++) { // ciclo interno per trovare indice effettivo del
                                                   //valore minimo
               if (a[i] < a[iMin]) {//entro se tra gli elementi NON ordinati, ne trovo uno
                   iMin = i;     //che ha valore minore rispetto all'attuale minimo a[iMin]
                                 //e aggiorno l'indice
               }
          }
          if (iMin != j) { //Se gli indici sono diversi, scambio i valori
                  swap(&a[j], &a[iMin]); //scambio i due valori: quello che esamino ora
                 //a[j] con il valore minimo tra gli elementi NON ancora ordinati a[iMin]
          }
     }
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Sequential-Sort (4)

```
void swap(int *A, int *B) {
        int temp = *B;
        *B = *A;
        *A = temp;
}
```

Esempio (1)

```
#include <stdio.h>
#include <stdbool.h>
//definizioni per l'uso di booleani
typedef int Boolean;
#define TRUE 1
#define FALSE 0
Boolean sequential_search(int *V, int N, int key);
Boolean b_search(int *V, int N, int key);
void Selection_Sort(int *V, int N);
void swap(int *A, int *B);
main(){ //A = array NON ordinato | B= array ordinato
   int A[] = { 10, 35, 8, 9, 897, 0, 99, 123 }, result = 0, i;
   int B[] = { 8, 9, 10, 35, 123 }, find_order, find_not_order;
   printf("\n\nStampo array iniziale NON ordinato (array A): \n");
   for (i = 0; i < 8; i++) {
       printf("%d ", A[i]);
   } ….
}
```

Esempio (2)

```
…
main(){ //A = array NON ordinato | B= array ordinato
   int A[] = { 10, 35, 8, 9, 897, 0, 99, 123 }, result = 0, i;
   int B[] = { 8, 9, 10, 35, 123 }, find_order, find_not_order;
   printf("\n\nStampo array iniziale NON ordinato (array A): \n");
   for (i = 0; i < 8; i++) {
       printf("%d ", A[i]);
   }
   printf("\n\nStampo array inziale Ordinato (array B): \n");
   for (i = 0; i < 5; i++) {
       printf("%d ", B[i]);
    }
printf("\n\nCerco %d nell'array NON Ordinato: \n", find_not_order);
   result = sequential_search(A, 5, find_not_order); //ricerca
                                   //sequenziale in un array NON ordinato
   if (result == TRUE)
        printf("Trovato valore!");
   else
        printf("NON trovato!");
…}
```

Nota: dichiarazione ricerca binaria
(array ordinato)
Boolean **sequential_search**(int *V, int N, int key);

Esempio (3)

```
…
main(){ //A = array NON ordinato | B= array ordinato
  int A[] = { 10, 35, 8, 9, 897, 0, 99, 123 }, result = 0, i;
  int B[] = { 8, 9, 10, 35, 123 }, find_order =11, find_not_order=123;
…. //ricerca in un array
printf("\nCerco l'elemento %d nell'array Ordinato: \n", find_order);
  result = b_search(B, 5, find_order); //ricerca binaria in un array
                                       //ORDINATO
  if (result == TRUE)
          printf("Trovato!");
  else
          printf("NON trovato!");
//ordino l'array non ordinato A
   Selection_Sort(A, 8);
   printf("\n\nStampo gli elementi in Ordine dell'array A (dopo
          esecuzione algoritmo): \n");
   for (i = 0; i < 8; i++) {
       printf("%d ", A[i]);
   }
}
```

Nota: dichiarazione ricerca binaria (array ordinato)
Boolean **b_search**(int *V, int N, int key);

Nota: dichiarazione ordinamento sequenziale
void **Selection_Sort**(int *V, int N);

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Esempio (4)

C:\WINDOWS\system32\cmd.exe

```
Stampo array inziale NON ordinato (array A):
10 35 8 9 897 0 99 123

Stampo array inziale Ordinato (array B):
8 9 10 35 123

Cerco l'elemento 11 nell'array NON Ordinato:
NON trovato!
Cerco l'elemento 123 nell'array Ordinato:
Trovato!

Stampo gli elementi in Ordine dell'array A (dopo esecuzione algoritmo):
0 8 9 10 35 99 123 897 Premere un tasto per continuare . . .
```

Ricorsione nelle funzioni (5)

- Fatta eccezione per l'inizializzazione e l'inserimento in testa (che trattano un singolo dato e non richiedono quindi né iterazione né ricorsione), tutte le restanti operazioni sfruttano lo stesso paradigma…:
  - search_r
  - suf_insert_r
  - ord_insert_r

Liste - Rappr. Collegata con puntatori: Ricerca ricorsiva

```
Boolean search_r(struct list * ptr, float value, struct list ** ptrptr){
/* ricerca in forma ricorsiva */
        if(ptr!=NULL){
                if(ptr->value==value){
                        *ptrptr=ptr;
                        return TRUE;
                }
                else return search_r(ptr->next_ptr, value, ptrptr);
        }
        else return FALSE;
}
```

Liste - Rappr. Collegata con puntatori: Inserimento Ordinato Ricorsivo

```
void ord_insert_r(struct list ** ptrptr, float value){
/* inserimento in ordine in forma ricorsiva */
        if(*ptrptr!=NULL && (*ptrptr)->value<value)
                ord_insert_r(&((*ptrptr)->next_ptr),value);
        else pre_insert(ptrptr,value);
}
```

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (1)

- Nella forma iterativa:
  - l'operazione elementare printf() viene applicata a una molteplicità di variabili collocandola nel corpo di un ciclo while
  - dopo l'esecuzione dell'operazione elementare, l'avanzamento del puntatore ptr = ptr->next_ptr aggiorna l'indirizzo della variabile che sarà trattata nella successiva iterazione

```
void visit(struct list * ptr) {
        while (ptr != NULL) {
                printf("%f\n", ptr->value); //operaz. elementare
                ptr = ptr->next_ptr; //aggiornamento guardia
        }
}
```

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (2)

- Nella forma ricorsiva:
  - l’istruzione che realizza l’operazione elementare (printf(…)) è esattamente la stessa ma in questo caso l’aggiornamento sul dato da trattare è ottenuto modificando il parametro con cui viene attivata la funzione visit_r() stessa

```
void visit_r(struct list * ptr) {
        if (ptr != NULL) {
                printf("%f \n", ptr->value);//operaz. elementare
                visit_r(ptr->next_ptr);
        }
}
```

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (3)

- Lo schema ricorsivo incorre in un maggiore costo in termini di spazio di memoria e tempo di esecuzione, dovuto al diverso carico sullo stack di sistema:
  - l'esecuzione della visita comporta la nidificazione di un numero di istanze della funzione vist_r() pari alla lunghezza della lista
  - per ogni istanza, sullo stack viene copiato l'indirizzo di ritorno della funzione e il valore dei parametri attuali
  - se ce ne fossero, sarebbero allocate anche le eventuali variabili locali dichiarate nel corpo della funzione

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (4)

- Questo fattore di costo aggiuntivo della forma ricorsiva rispetto allo schema iterativo è generale, ed è la buona ragione per cui soluzioni ricorsive non sono adatte a implementare operazioni che incidono significativamente sulla prestazione di un sistema
- Gli schemi ricorsivi hanno talvolta una maggiore capacità di adattarsi alla struttura dei dati:
  - Questa è la ragione per cui in molti casi, pur a costo di una minore efficienza di esecuzione, uno schema ricorsivo può risultare migliore

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (5)

- Un esempio che illustra bene il concetto di adattamento alla struttura dati, è il problema della visita di una lista in ordine inverso: In sostanza, assegnata una lista { 1, 2, 5 } si vuole vedere comparire a video la sequenza { 5, 2, 1 }
- In uno schema **ricorsivo**, l'effetto di invertire l'ordine di visita degli elementi si ottiene:
  - postponendo l'operazione elementare (printf()) sul dato corrente, alla distribuzione del controllo mediante la chiamata ricorsiva:

```
void visit_r_backward(struct list * ptr){//inverto l'ordine: prima richiamo
        if(ptr!=NULL){//la visita e POI effettuo l'op. elementare
                visit_r_backward(ptr->next_ptr);
                printf("%f",ptr->value); //operazione elementare
        }
}
```

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (6)

- Ottenere lo stesso effetto in uno schema **iterativo** è più complesso
- In questo caso, la soluzione migliore è quella di:
  - copiare i valori su una lista di appoggio che inverte la relazione di successione POI scandire la lista di appoggio
- La lista invertita si genera in modo naturale:
  - scandendo la lista originale (con un ciclo while…) E inserendo ogni elemento, in testa alla lista di appoggio

Dichiarazione funzione di inizializzazione
void init(struct list ** ptrptr);

Dichiarazione inserimento in testa
pre_insert(struct list ** ptrptr, float value);

```
void visit_backward(struct list *ptr) {
	struct list *tmp_ptr; //lista di appoggio
	init(&tmp_ptr);
	while (ptr!=NULL) {//inserimento
//di ogni elemento della lista originale in testa alla lista temporanea
		pre_insert(&tmp_ptr, ptr->value);
		ptr = ptr->next_ptr;
	}
	visit(tmp_ptr);
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (7)

- Soluzione iterativa:
  - Gli inserimenti sono effettuati in testa alla lista temporanea creata (o lista di appoggio) che viene creata in tempo lineare con un numero di operazioni proporzionale alla dimensione della lista
  - Non viene quindi modificato l'ordine di grandezza della complessità della visita che è comunque una operazione di complessità lineare
- Rispetto allo schema ricorsivo:
  - emerge il fatto che la lista di appoggio (nella soluzione iterativa) richiede uno spazio di memoria proporzionale alla dimensione della lista, il che (apparentemente) non avviene nell'implementazione ricorsiva

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (8)

- Ad una analisi attenta del carico sullo stack di sistema, si osserva però che anche la soluzione ricorsiva incorre nella stessa occupazione di memoria:
  - Gli elementi visitati e non ancora stampati a video sono memorizzati come parametri formali sullo stack di sistema
  - Quando la ricorsione raggiunge la sua massima nidificazione, sullo stack sono copiati gli indirizzi di tutti i nodi della lista

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (9)

- L'implementazione ricorsiva ha in ogni caso il pregio di:
  - rendere trasparente al programmatore la memorizzazione delle variabili intermedie
  - Gode di una maggiore efficienza nella copia di variabili sullo stack di sistema (piuttosto che in aree di memoria allocate dal programma mediante l'istruzione malloc() usate nella funzione pre_insert())

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (10)

- GUARDIA (condizione usata per scandire fine/proseguimento della visita in lista):
  - Forma **iterativa** della operazione di visita, il controllo sulla terminazione della ripetizione è effettuato in testa prima del corpo del while:
    - while (ptr!=NULL){…}
  - Forma **ricorsiva**, il controllo di terminazione è effettuato da un if in testa al corpo della funzione:
    - if(ptr!=NULL){…}
- Nell'approccio iterativo la terminazione della ripetizione può essere controllata in uscita, usando un do-while al posto del while

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (11)

- SE si usa il do{…}while, al posto del while, implica che:
  - All’inizio di una iterazione deve essere garantito che l’insieme dei dati da trattare sia NON vuoto
  - La struttura di controllo permette il rientro nel ciclo SOLO SE, dopo il trattamento di un dato (printf()), è garantito che ci siano ancora altri dati nella lista

- In questa ottica, è necessario aggiungere un controllo iniziale, eseguito una sola volta, per verificare che l’insieme iniziale da trattare sia NON vuoto:
  - if(ptr !=NULL)

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (12)

```
void visit(struct list * ptr) {
    while (ptr != NULL) {
        printf("%f\n", ptr->value);
        ptr = ptr->next_ptr;
    }
}
```

- Caso1:
  - Ciclo while()

```
void visit_p(struct list *ptr) {
    if (ptr != NULL) {//controllo
        do{
            printf("%f \n", ptr->value);
            ptr = ptr->next_ptr;
        } while (ptr!=NULL);
    }
}
```

- Caso 2:
  - Ciclo do{…}while();

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (13)

- Passando ad uno schema ricorsivo: il controllo in uscita si traduce nell'attribuire al chiamante la responsabilità del controllo sulla legalità del dato da trattare:
  - il corpo della funzione assume che il suo chiamante abbia già provveduto a verificare che l'insieme dei dati da trattare sia NON vuoto
  - Quindi, prima di propagare il controllo con una chiamata ricorsiva, la funzione controlla che l'insieme dei dati che rimangono da trattare sia non vuoto

```
void visit_pr(struct list *ptr) {
        // si assume che ptr!=NULL (controllo da mettere quando
        //viene richiamata la funzione)
        printf("%f \n", ptr->value);
        if (ptr->next_ptr != NULL)
                visit_pr(ptr->next_ptr);
}
```

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (14)

```
void visit(struct list * ptr) {
     while (ptr != NULL) {
          printf("%f\n", ptr->value);
          ptr = ptr->next_ptr;
     }
}
```

```
void visit_p(struct list *ptr) {
    if (ptr != NULL) {//controllo
        do{
            printf("%f \n", ptr->value);
            ptr = ptr->next_ptr;
        } while (ptr!=NULL);
    }
}
```

```
void visit_pr(struct list *ptr) {
// si assume che ptr!=NULL nel chiamante
    printf("%f \n", ptr->value);
    if (ptr->next_ptr != NULL)
        visit_pr(ptr->next_ptr);
}
```

- Caso1: Iterazione
  - Ciclo while()
- Caso 2: Iterazione
  - Ciclo do{…}while();
- Caso 3: Ricorsione

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (15)

Esempio di controllo

- La responsabilità di verificare che la prima attivazione della funzione sia legale è della funzione esterna che usa visit_rp(), ovvero la **funzione chiamante:**

```
[...]
struct list * ptr;
[...]
if(ptr!=NULL) // verifica di legalità
visit_rp(ptr);
[...]
```

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (18)

- Nelle implementazioni in forma ricorsiva, l'uso del valore di ritorno delle funzioni permette spesso di ottenere soluzioni di maggiore semplicità
- ESEMPIO: operazione di somma dei valori degli elementi di una lista (visita_r_backward()) usando un valore di ritorno:

```
int sum_r_return(struct list * ptr) {
        if(ptr!=NULL) {
                return ptr->value + sum_r_return(ptr->next_ptr);
        }
        else {
                return 0;
        }
}
```

Ricorsione e valore di ritorno

UNIVERSITÀ DEGLI STUDI FIRENZE | DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE | DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (19)

- L'uso del valore di ritorno risolve in modo naturale il problema della variabile di accumulazione della somma (sum)
- Diversa natura delle espressioni di return:

Ricorsione e valore di ritorno

```
…
if(ptr!=NULL)
        return ptr->value + sum_r_return(ptr->next_ptr); //caso 1
else
        return 0; //caso 2
…
```

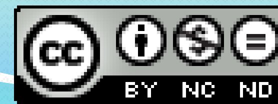
  - Caso1: propaga il valore restituito da altre istanze della funzione aggiungendoci qualcosa
  - Caso 2: determina un risultato finale

Liste - Rappr. Collegata con puntatori: confronto fra Visita Ricorsiva e Iterativa (20)

- Il concetto è illustrato in modo più elaborato nel caso di una operazione di ricerca su una **lista ordinata**
- In questo caso la terminazione può avvenire perché:
  - Viene identificato un elemento che contiene il valore cercato
  - La visita esaurisce la lista
  - La visita raggiunge un valore maggiore di quello cercato

```
Boolean search_r_return(struct list * ptr, float target){
        if( ptr!=NULL && ptr->value <= target ) {
                if ( ptr->value == value)
                        return true;
                else
                        return search_r_return( ptr->next_ptr, target);
        }
                else {
                        return false;
        }
}
```

Ricorsione e valore di ritorno – Operazione di ricerca



Fondamenti di Informatica

Eng. Ph.D. Michela Paolucci

DISIT Lab *http://www.disit.dinfo.unifi.it/*
Department of Information Engineering, DINFO
University of Florence
Via S. Marta 3, 50139, Firenze, Italy
tel: +39-055-2758515, fax: +39-055-2758570
michela.paolucci@unifi.it