Liste a puntatori

Prof. Orazio Mirabella

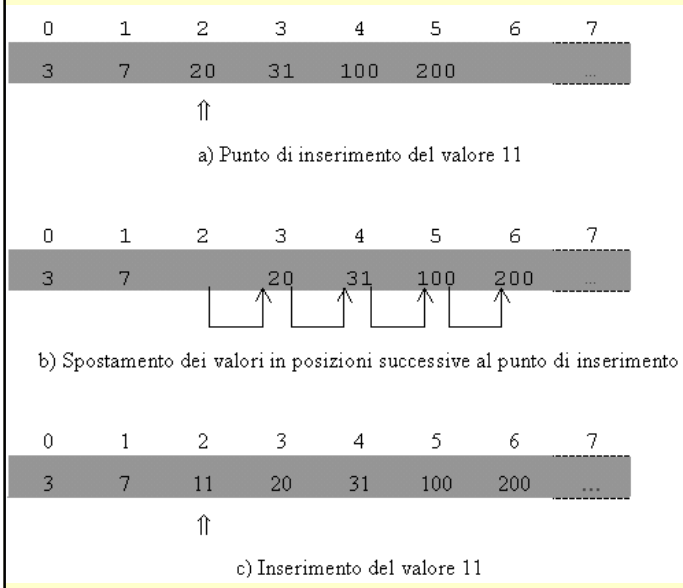
Organizzazione in Insiemi

- Le strutture dati individuate per la risoluzione dei problemi sono genericamente dette "astratte".
- Una *struttura dati astratta* definisce l'organizzazione delle informazioni indipendentemente dalla traduzione in uno specifico linguaggio di programmazione;
- la rappresentazione che una struttura astratta prende in un linguaggio di programmazione viene detta *implementazione*.
- Per facilitare il lavoro del programmatore, il C mette a disposizione una serie di strumenti, tra i quali gli array, le struct e i puntatori.

Organizzazione in Insiemi

- Molto spesso bisogna gestire insiemi di oggetti **dello stesso tipo** **su cui effettuare delle operazioni**.
- Un modo **molto semplice** per tenere in memoria tali insiemi consiste nel creare un **array** per contenerli.
- Gli array presentano però alcuni **svantaggi**:
  - ➔ La **dimensione** deve essere **prefissata**;
  - ➔ **L'occupazione effettiva** di memoria non coincide con la **necessità** del processo in fase di esecuzione;
  - ➔ Non è possibile **allungare** un array (se non creandone uno **nuovo**);
  - ➔ Non è possibile inserire elementi **in mezzo** (se non spostandone alcuni **in avanti**): ➔ bassa velocità di esecuzione

Organizzazione in Insiemi



Se si memorizza la lista in un array l'operazione di inserimento di un valore deve prevedere le fasi:
1. ricerca del punto d'inserimento;
2. spostamento di un posto di tutti gli elementi successivi;
3. inserimento dell'informazione nell'array

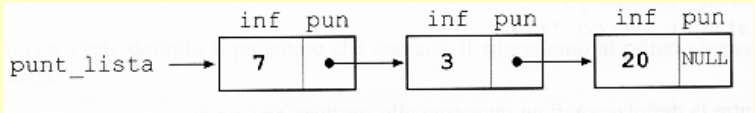
Organizzazione in Insiemi

- Noi vorremmo:
  - ➔ Gli elementi devono essere organizzati in **un'unica struttura**;
  - ➔ La dimensione del nostro insieme deve poter **crescere e descrescere** in funzione delle **reali necessità** del processo in esecuzione;
  - ➔ Deve essere possibile mantenere un **ordine** nell'insieme;
  - ➔ Deve essere possibile inserire elementi **nel mezzo**

Soluzione: Lista **lineare a puntatori** !

Liste lineari

- Una lista lineare è una successione di elementi omogenei che occupano in memoria una posizione qualsiasi.
- Ciascun elemento contiene un'informazione e un puntatore per mezzo del quale è legato al successivo.
- L'accesso alla lista avviene con il puntatore al primo elemento. :
  Elemento = informazione + puntatore
- Il puntatore è il riferimento a un elemento, il suo valore è l'indirizzo dell'elemento nella memoria del sistema.



- Il campo puntatore dell'ultimo elemento della lista non fa riferimento a nessun altro elemento; il suo contenuto corrisponde a un segnale di fine lista che in C è il valore NULL.
- Una *lista vuota* non ha elementi ed è rappresentata da punt_lista che punta a NULL.

Elementi della lista

- La parte informazione dell'elemento dipende dal tipo di dati che stiamo trattando.
- In C può essere costituita da uno qualsiasi dei tipi semplici che conosciamo: int, float ecc.
- Nel caso il cui il campo informazione sia di tipo intero, la dichiarazione della struttura di ogni elemento può essere la seguente:

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

- La precedente dichiarazione descrive la struttura di elemento ma non alloca spazio in memoria.

  La definizione: struct elemento *punt_lista;

  stabilisce che punt_lista è un puntatore che può riferirsi a variabili di tipo elemento.

Operazioni sulle Liste

- Per **l'ADT** Lista definiamo le seguenti **tipologie** di operazioni:
  - ➔ *Inserimento*
  - ➔ *Ricerca*
  - ➔ *Cancellazione*
    - ✓ Ognuna di queste operazioni è implementata in modo **diverso** a seconda che la lista sia mantenuta **ordinata** (rispetto al campo info) oppure no.
- È da sottolineare la posizione in memoria non sequenziale: quando si aggiunge un ulteriore elemento alla lista si deve allocare uno spazio di memoria, connetterlo all'ultimo elemento della lista e inserirvi l'informazione relativa.
- Nella lista lineare a puntatori, non esiste alcun modo di risalire da un elemento al suo antecedente. La lista si può *scandire* solo in ordine, da un elemento al successivo, per mezzo dei puntatori.

Gestione di una lista

- Si voglia memorizzare e successivamente visualizzare una sequenza di *n* interi. Il valore di *n* non è conosciuto a priori, ma è determinato in fase di esecuzione.
- Si propende per una soluzione che utilizzi la memoria in modo dinamico.
- Il tipo elemento è una struttura composta da due campi, un campo inf di tipo int e un campo pun puntatore alla struttura stessa:

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

- Il problema presentato è divisibile nei due sottoproblemi:
  - · memorizzare la sequenza;
  - · visualizzare la sequenza.

Gestione di una lista

- La soluzione dei due sottoproblemi è delegata a due funzioni la cui dichiarazione è:

```
struct elemento *crea_lista();
void visualizza_lista(struct elemento *);
```

- Nel main viene definito il puntatore che conterrà il riferimento al primo elemento della lista:

```
struct elemento *punt_lista;
```

- Le due funzioni vengono chiamate in sequenza dallo stesso main;
- punt_lista = crea_lista();
- visualizza_lista(punt_lista);

- La procedura crea_lista restituisce al main il puntatore alla lista che è assegnato a punt_lista e che viene successivamente passato a visualizza_lista.

Funzione crea_lista 1/4

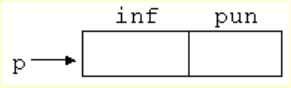
- La funzione crea_lista è di tipo puntatore a strutture elemento. Non prevede il passaggio di valori dal chiamante:

  struct elemento *crea_lista();

- La funzione crea_lista comprende la dichiarazione di **p**, puntatore alla testa della lista, e di **paus**, puntatore ausiliario, che permette la creazione degli elementi successivi al primo, senza perdere il puntatore iniziale alla lista.
- In primo luogo si deve richiedere all'utente di inserire il numero n di elementi da cui è composta la lista.
- se n è uguale a zero, si assegna a p il valore NULL, che corrisponde a lista vuota. In questo caso il sottoprogramma termina.
- Se n è maggiore di zero crea_lista deve creare il *primo elemento*:

  p = (struct elemento *)malloc(sizeof(struct elemento));
- L'operatore sizeof restituisce la quantità di memoria occupata da un elemento e malloc alloca uno spazio corrispondente di memoria libera. Successivamente viene effettuato un *cast* del valore ritornato da malloc, in modo da trasformarlo in un puntatore allo spazio allocato che viene assegnato a p.
- Per richiamare malloc si deve includere nel programma il riferimento alla libreria malloc.h e/o stdlib.h;

Funzione crea_lista 2/4

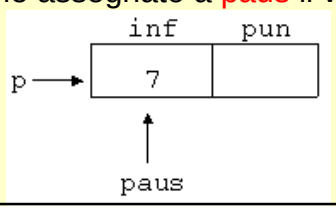


- l'utente inserisce la prima informazione che viene assegnata p->inf, campo inf del primo elemento della lista:

  scanf("%d", &p->inf);

- dopo di che viene assegnato a paus il valore di p:

  paus = p;



Funzione crea_lista 3/4

- Finita la gestione del primo elemento, inizia un ciclo per la creazione degli *elementi successivi al primo, che* si ripete *n*–1 volte
- for(i = 2; i<=n; i++) {

  a) *crea il nuovo elemento concatenato al precedente*

  b) *sposta di una posizione avanti* paus

  c) *chiedi all'utente la nuova informazione della sequenza*; *inserisci l'informazione nel campo* inf *del nuovo elemento*

  }
- Nel caso dell'esempio proposto, il ciclo si ripete due volte (da i=2 a i=n=3); a ogni iterazione le operazioni descritte corrispondono a:

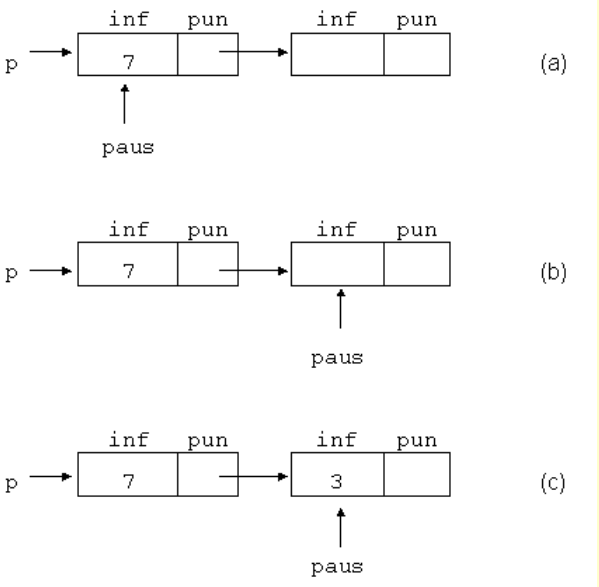
  a) paus->pun = (struct elemento *)malloc(sizeof(struct elemento));

  b) paus = paus->pun;

  c) printf("\nInserisci la %d informazione: ", i);

  scanf("%d", &paus->inf);

Funzione crea_lista 4/4



Infine la funzione assegna al campo puntatore dell'ultimo elemento il valore NULL e termina passando al chiamante il valore di p, cioè il puntatore alla lista: return(p);

Funzione visualizza_lista

visualizza_lista(punt_lista);

Il parametro attuale punt_lista corrisponde al parametro formale p. Per effettuare la scansione della lista utilizziamo il seguente ciclo:

```
while(p!=NULL) {
printf("%d", p->inf); /* Visualizza il campo informazione */
printf("---> ");
p = p->pun; /* Scorri di un elemento in avanti */
}
```

Il ciclo di scansione è controllato dal test sopra il puntatore p: se p!=NULL continua l'iterazione. Questo controllo ci è permesso perché abbiamo avuto cura di porre il segnale di fine lista nella funzione crea_lista. La scansione degli elementi è consentita dall'operazione di assegnamento:

```
p = p->pun;
```

Programma esempio

- Vogliamo ora sintetizzare tutti i concetti che abbiamo imparato sulle Liste a puntatori, in un semplice programma.
- Questo programma permette all'utente di costruire una lista a puntatori, di inserire i valori e poi visualizzare la lista.
- Il programma successivamente verrà arricchito da un menù che permetterà di scegliere anche altre operazioni sulle liste e in particolare l'inserimento di un nuovo elemento (in testa, in fondo ed in ordine) e la sua cancellazione

```
/* Accetta in ingresso una sequenza di interi e li memorizza in una lista. Il
    numero di interi che compongono la sequenza è richiesto all'utente. La
    lista creata viene visualizzata */
#include <stdio.h>
#include <malloc.h>
/* struttura degli elementi della lista */
struct elemento {
int inf;
struct elemento *pun;
};
struct elemento *crea_lista(); /*dichiarazione della funzione crea_lista*/
void visualizza_lista(struct elemento *); /*dich.della fun. visualizza_lista*/
main()
{
struct elemento *punt_lista; /* Puntatore alla testa della lista */
punt_lista = crea_lista(); /* Chiamata funzione per creare la lista */
visualizza_lista(punt_lista); /* Chiamata funzione per visualizzare la lista */
} /*fine main*/
```

Funzione crea_lista

```
/* accetta i  valori immessi e crea la lista. Restituisce il puntatore alla testa */
struct elemento *crea_lista()
{
struct elemento *p, *paus;
int i, n;
printf("\n Di quanti elementi è composta la sequenza? ");
scanf("%d", &n);
if(n==0) p = NULL; /* lista vuota */
else
{
/* Creazione del primo elemento */
p = (struct elemento *)malloc(sizeof(struct elemento));
printf("\nInserisci la 1 informazione: ");
scanf("%d", &p->inf);
paus = p;
```

```
/* creazione degli elementi successivi */
for(i=2; i<=n; i++) {
paus->pun = (struct elemento *)malloc(sizeof(struct
    elemento));
paus = paus->pun;
printf("\nInserisci la %d informazione: ", i);
scanf("%d", &paus->inf);
}
paus->pun = NULL; /* Marca di fine lista */
}
return(p);
}
```

Funzione per la visualizzazione della lista.

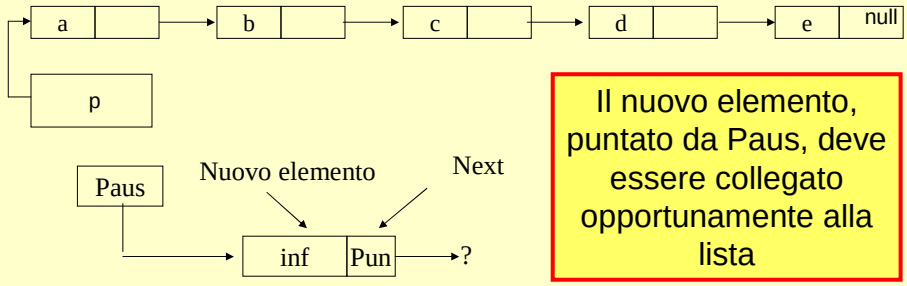
```
/* Il parametro in ingresso è il puntatore alla testa */
void visualizza_lista(struct elemento *p)
{
printf("\npunt_lista---> ");
/* Ciclo di scansione della lista */
while(p!=NULL) {
printf("%d", p->inf); /* Visualizza il campo informazione */
printf("---> ");
p = p->pun; /* Scorri di un elemento in avanti */
}
printf("NULL\n\n");
}
```

Inserimento in testa

- Viene effettuato dalla funzione inserisci_in_testa:

struct elemento *inserisci_in_testa(struct elemento *p)

- Il puntatore P che viene passato come parametro di ingresso è quello che punta al primo elemento della lista.
- Il puntatore che viene restituito, punta al primo elemento della lista dopo che l'inserimento è stato effettuato



Il nuovo elemento, puntato da Paus, deve essere collegato opportunamente alla lista

Funzione inserisci in testa

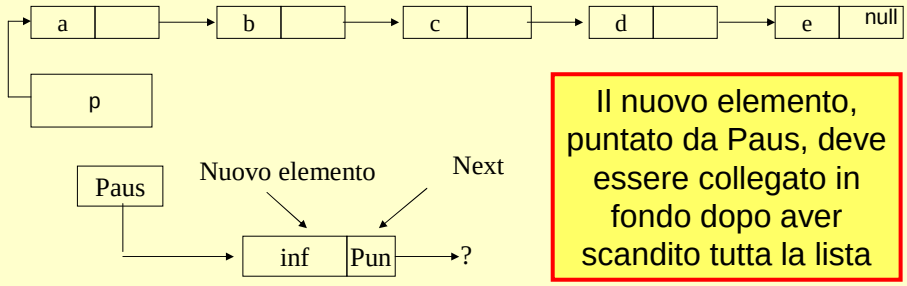
```
struct elemento *inserisci_in_testa(struct elemento *p)
{
struct elemento  *paus;
int i, n;

/* Creazione del primo elemento */
paus = (struct elemento *)malloc(sizeof(struct elemento));
printf("\nInserisci la 1 informazione: ");
scanf("%d", &paus->inf);
paus->pun = p;
p=paus;
return (p);}
```

Inserimento in fondo

- Viene effettuato dalla funzione inserisci_in_fondo:

struct elemento *inserisci_in_fondo(struct elemento *p)

- Il puntatore P che viene passato come parametro di ingresso è quello che punta al primo elemento della lista.
- Il puntatore P viene modificato solo nel caso di lista vuota (l'elemento da inserire in fondo è il primo elemento)



Il nuovo elemento, puntato da Paus, deve essere collegato in fondo dopo aver scandito tutta la lista

Funzione inserisci in fondo

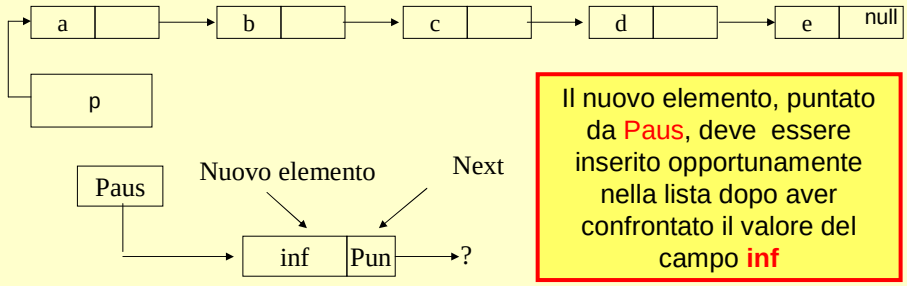
```
struct elemento *inserisci_in_fondo(struct elemento *p)
{
struct elemento  *paus, *paux;
int i, n;

/* Creazione del nuovo elemento */
paus = (struct elemento *)malloc(sizeof(struct elemento));
printf("\nInserisci la 1 informazione: ");
scanf("%d", &paus->inf);
paus->pun = (NULL);
paux = p; //ora paus punta al primo elemento
while (paux->pun !=NULL)
paux=paux->pun;
paux->pun=paus;
return (p);}
```

Inserimento in ordine

- Viene effettuato dalla funzione inserisci_in_ordine:

struct elemento *inserisci_in_ordine(struct elemento *p)

- Il puntatore P che viene passato come parametro di ingresso è quello che punta al primo elemento della lista.
- Il puntatore che viene restituito, punta al primo elemento della lista dopo che l'inserimento è stato effettuato.



Il nuovo elemento, puntato da Paus, deve essere inserito opportunamente nella lista dopo aver confrontato il valore del campo **inf**

Funzione inserisci in ordine

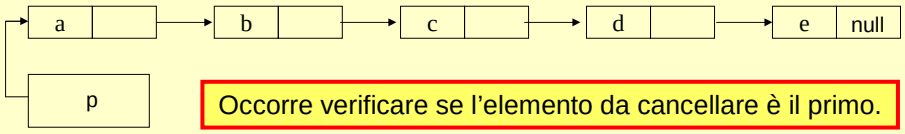
```
struct elemento *inserisci_in_ordine(struct elemento *p)
{
struct elemento  *paus, *paux, *paux2;
int i, n;
/* Creazione del nuovo elemento */
paus = (struct elemento *)malloc(sizeof(struct elemento));
printf("\nInserisci la 1 informazione: ");
scanf("%d", &paus->inf);
paus->pun = (NULL);
paux = p; //ora paux punta al primo elemento
while (paux->inf<paus->inf)
{
paux2=paux;  /*punta all'elemento attuale*/
paux=paux->pun;}  /*punta all'elemento successivo*/
paux2->pun=paus;  /* colleghiamo il nuovo elemento*/
paus->pun=paux;  /*completiamo il collegamento*/
return (p);}
```

Cancellazione di un elemento

- Viene effettuato dalla funzione elimina_elemento:

struct elemento *elimina_elemento(struct elemento *p)

- Il puntatore P che viene passato come parametro di ingresso è quello che punta al primo elemento della lista.
- Occorre scorrere la lista per trovare l'elemento da cancellare.
- La funzione free permette di recuperare la memoria dell'elemento cancellato



Occorre verificare se l'elemento da cancellare è il primo.

Funzione Elimina_elemento

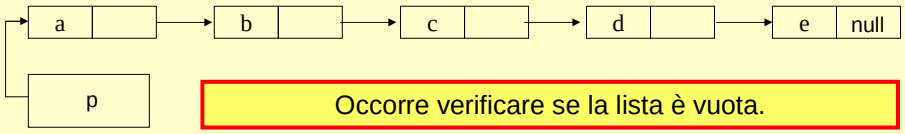
```
struct elemento *elimina_elemento(struct elemento *p)
{
struct elemento  *paus, *paux, *paux2;
int i, n;
printf("\nInserisci il campo val dell'elemento da cancellare\n");
scanf("%d", &n);
paux = p; //ora paux punta al primo elemento
if (paux->inf==n) //occorre cancellare il primo elemento
    {p=p->pun;
    free(paux);}
else
    {
while (paux->inf!=n)
    {
    paux2=paux;
    paux=paux->pun;}
    paux2->pun=paux->pun;
    free(paux);
    }
return (p);}
```

Ricerca dell'elemento maggiore

- Viene effettuato dalla funzione cerca_maggiore:

void *cerca_maggiore(struct elemento *p)

- Il puntatore P che viene passato come parametro di ingresso è quello che punta al primo elemento della lista.
- Si prende come maggiore il primo elemento della lista
- Occorre scorrere la lista confrontando i campi inf e sostituire il valore maggiore , se necessario..



Occorre verificare se la lista è vuota.

Funzione "cerca il maggiore"

```
void cerca_maggiore(struct elemento *p)
{
struct elemento  *paux;
int n;
paux = p;  //ora paux punta al primo elemento
if (paux==NULL)
printf("la lista e' vuota");
else
    {
    n=paux->inf;
          while (paux->pun!=NULL)
          {
          paux=paux->pun;
          if (n< paux->inf)
          n=paux->inf;};
printf("\ncome si vede il maggiore e': %d\n",n);
    }
}
```

Creazione menù

```
do
{
printf("\n\nvuoi effettuare una
    operazione?\n");
printf("Digita una cifra:\n");
printf("\n0: inserisci in testa\n");
printf("1: inserimento in fondo\n");
printf("2: inserimento ordinato (si
    ipotizza che i primi elementi siano
    stati inseriti in ordine\n");
Printf("3");
printf("4: elimina elemento\n");
printf("5: esci\n");
scanf("%d", &x);
switch(x) {
case 0:
punt_lista=inserisci_in_testa
    (punt_lista);
break;
case 1:
punt_lista =inserisci_in_fondo
    (punt_lista);
break;
case 2:
punt_lista=inserisci_in_ordine
    (punt_lista);
break;
case 3:
Cerca_maggiore(punt_lista);
break;
case 4:
punt_lista=elimina_elemento
    (punt_lista)break;
case 5:
printf("cinque\n");
break;
default:
printf("non compreso\n");
break;
}
visualizza_lista(punt_lista);
}
while (x<5);
```