STRUTTURE

Sommario

- Introduzione
- Definizione di strutture
- Inizializzazione di strutture
- Accesso ai membri di una struttura
- Uso delle strutture con le funzioni
- Typedef

INTRODUZIONE

L'array è un esempio di struttura dati. Utilizza dei tipi di dati semplici, come int, char o double e li organizza in un array lineare di elementi.

L'array costituisce la soluzione in numerosi casi ma non tutti, in quanto c'è la restrizione che tutti i suoi elementi siano dello stesso tipo.

In alcuni casi è infatti necessario poter gestire all'interno della propria struttura un misto di dati di tipo diverso.

Si consideri a titolo d'esempio l'informazione relativa ai dipendenti di una ditta, per ognuno si ha: *nominativo*, *età* e *salario*. Il nominativo richiede una stringa, ossia un array di caratteri terminati dal carattere '\0', l'età ed il salario richiedono interi.

Con le conoscenze acquisite fino ad ora è possibile solamente dichiarare delle variabili separate, soluzione non altrettanto efficiente dell'utilizzo di una *unica struttura dati* individuata da un unico nome: il linguaggio C a tale scopo dispone della struct.

Le strutture

- Strutture
  - Collezione di variabili collegate fra loro (aggregati) sotto un unico nome
    - Possono contenere variabili di tipo diverso
  - Combinate con i puntatori, consentono di creare liste concatenate, pile, code ed alberi.

Definizione di una struct

La dichiarazione di una struct è un processo a **due fasi**.

La **prima** è la definizione di una struttura con i campi dei tipi desiderati, utilizzata poi per definire tutte le variabili necessarie con la suddetta struttura.

Si consideri il seguente esempio: si desidera gestire i dati di un insieme di persone relativamente al loro nominativo, all'età e al loro salario.

Per prima cosa si definisce la struct che consente di memorizzare questo tipo di informazioni, nel seguente modo:

```
struct s_dipendente
{
char nominativo[40];
int anni;
int salario;
};
```


La **seconda fase** consente di definire una variabile con la struttura appena introdotta:

struct s_dipendente dip;

La variabile si chiama dip ed è del tipo struct s_dipendente definito precedentemente.

La **sintassi** per la definizione di una struct è la seguente:

struct *nome_struttura*
{
lista dei campi (tipo-nome)
};

In seguito è possibile definire **variabili** come segue:

struct *nome_struttura nome_variabile*;

Definizione di strutture

Esempio

```
struct card {
  char *face;  /*asso,due,tre,…,regina,re*/
  char *suit;  /*cuori,quadri,fiori,picche*/
 };
```

- **struct** introduce la definizione della struttura **card**
- **card** è il *nome della struttura* e serve per dichiarare variabili di *tipo struttura*
- **card** contiene due membri di tipo stringa, cioè **char *** , che sono **face** e **suit**

Definizione di strutture

- Struct
  - Una struct non può contenere un riferimento a se stessa
  - Può però contenere un membro che è un **puntatore allo stesso tipo di struttura** (**strutture ricorsive**)
  - La definizione di una struttura non riserva spazio in memoria
  - Crea un nuovo tipo di dati che serve per dichiarare variabili di tipo struct.
- Dichiarazione
  - E' come per le altre variabili:

```
card oneCard, deck[ 52 ], *cPtr;
```

  - Può essere una lista di elementi separati da virgole:

```
struct card {
   char *face;
   char *suit;
} oneCard, deck[ 52 ], *cPtr;
```


Operazioni sulle strutture

- Operazioni valide
  - Assegnare una struttura ad una struttura dello stesso tipo
  - Acquisire l'indirizzo (`&`) di una struttura
  - Accedere ai membri di una struttura
  - Usare l'oeratore **`sizeof`** per determinare la dimensione di una struttura

Accesso ai campi della struttura

Per accedere ai campi della struttura, per scrivere un valore o per leggerlo, è necessario indicare il nome della variabile seguito da quello del campo di interesse, separati da un punto.

Ad esempio, per la struttura s_dipendente precedentemente dichiarata:

```
struct s_dipendente
{
char nominativo[40];
int anni;
int salario;
 };


...
dip.anni = 30;
 ...
printf("%d\n", dip.anni);
 ...
```

Una volta individuato il campo d'interesse mediante la sequenza nome_variabile.nome_campo si ha a che fare con una variabile normale, e nel caso di dip.anni con una variabile di tipo intero.

Inizializzazione di strutture

- Lista degli inizializzatori
  - Esempio:

```
card oneCard = { "Three", "Hearts" };
```

- Istruzioni di assegnamento
  - Esempio:

```
card threeHearts = oneCard;
```

  - Oppure:

```
card threeHearts;
threeHearts.face = "Three";
threeHearts.suit = "Hearts";
```


Esempio di definizione e accesso ai campi di una struttura

```
#include <stdio.h>
/*concessionario auto di varie marche e
modelli e n. auto vendute*/
struct automobile {
  char *marca;
  char *modello;
  int venduto;
};

main()
{
struct automobile a1, a2;

a1.marca = "FERRARI";
a1.modello = "F40";
a1.venduto = 200;

a2.marca = "OPEL";
a2.modello = "ASTRA";
a2.venduto = 1200;

printf("marca auto = %s\n", a1.marca);

printf("modello auto = %s\n", a1.modello);

printf("vendute = %d\n", a1.venduto);

printf("marca auto = %s\n", a2.marca);

printf("modello auto = %s\n", a2.modello);

printf("vendute = %d\n", a2.venduto);

}
```


Per **riassumere**:

l'aspetto significativo delle struct è determinato dalla possibilità di memorizzare informazioni di natura diversa al'interno di un'unica variabile.

Una struct può essere utilizzata per integrare un gruppo di variabili che formano un'unità coerente di informazione.

Ad esempio il linguaggio C non possiede un tipo fondamentale per rappresentare i numeri complessi: una soluzione semplice consiste nell'utilizzare una struct e nel definire un insieme di funzioni per la manipolazione di variabili.

Si consideri a tale scopo il seguente esempio:

```
struct s_complesso
{
float reale;
float immaginaria;
};
```

A questo punto è possibile definire tre variabili a, b, c di tipo struct s_complesso:

```
struct s_complesso a, b, c;
```


È possibile effettuare operazioni di assegnamento tra i vari campi della struct come ad esempio:

a.reale = b.reale;

D'altra parte non si può scrivere un'espressione del tipo
c = a + b;
per la quale è necessario invece scrivere:

c.reale = a.reale + b.reale;
c.immaginaria = a.immaginaria + b.immaginaria;

A questo punto potrebbe quindi essere conveniente scriversi un insieme di funzioni che effettuino le operazioni elementari sui numeri complessi da richiamare ogni volta.

Puntatori a strutture

Come per tutti i tipi fondamentali è possibile definire un puntatore ad una struct.

struct s_dipendente * ptr

definisce un puntatore ad una struttura s_dipendente. Il funzionamento è pressoché inalterato.

(*ptr).anni

è il campo anni della struttura s_dipendente a cui punta ptr, ed è un numero intero.

È necessario utilizzare le parentesi in quanto *il punto '.' ha una priorità superiore all'asterisco '*'.*

Di fatto l'utilizzo di puntatori a struct è estremamente comune e la combinazione della notazione '*' e '.' è particolarmente soggetta ad errori;

esiste quindi una forma alternativa più diffusa che equivale a (*ptr).anni, ed è la seguente:

prt->anni

Questa notazione dà un'idea più chiara di ciò che succede:

prt punta (cioé ->) alla struttura e .anni "preleva" il campo di interesse.

L'utilizzo di puntatori consente di riscrivere la funzione di somma di numeri complessi passando come parametri non le struct quanto i puntatori a queste.

```
void s_complesso somma(struct s_complesso *a, struct s_complesso *b , struct s_complesso *c)
{
c->reale = a->reale + b->reale;
 c->immaginaria = a->immaginaria + b->immaginaria;
}
```

In questo caso c è un puntatore e la chiamata deve essere fatta così:

```
somma(&x, &y, &z);
```

In questo caso si risparmia spazio nella chiamata alla funzione, in quanto si passano i tre indirizzi invece delle strutture intere

Accesso ai membri di una struttura

- Accesso ai membri di una struttura
  - Si usa **l'operatore punto (.)** – con il nome della struttura:

```
card myCard;
printf( "%s", myCard.suit );
```

  - Si usa **l'operatore freccia (->)** – con puntatori a variabili di tipo struttura:

```
card *myCardPtr = &myCard;
printf( "%s", myCardPtr->suit );
```

`myCardPtr->suit` equivale a `( *myCardPtr ).suit`

Strutture e funzioni

- Passaggio di strutture a funzioni
  - Passaggio dell'intera struttura
    - Oppure, passaggio dei singoli elementi (membri)
  - In tutti e due i casi il passaggio è per valore
- Per passare una struttura per indirizzo:
  - Si passa il suo indirizzo
  - Si passa un riferimento ad esso
- Per passare un array per valore
  - Si crea una struttura che ha l'array come membro
  - Si passa la struttura

Strutture e funzioni

La maggior parte dei compilatori C consente di passare a funzioni e farsi restituire come parametri intere strutture. Ad es.:

```
/*somma è una funzione di tipo struct s_complesso */
struct s_complesso somma(struct s_complesso a , struct s_complesso b)
{
struct s_complesso c;
c.reale = a.reale + b.reale;
c.immaginaria = a.immaginaria + b.immaginaria;
return (c);
}
```

Definita la funzione somma è possibile chiamarla, come nel seguente esempio:

```
struct s_complesso x, y, z;
 ...
x = somma(y, z);
```

Si tenga presente che il passaggio di una struct per valore può richiedere un elevato quantitativo di memoria.

Typedef

La dichiarazione della struct per poter gestire insiemi di dati non omogenei viene spesso completata introducendo un **nuovo tipo**, definito appunto dall'utente, che si va ad affiancare ai tipi fondamentali del C.
Si consideri il caso della struct per la gestione dei numeri complessi. Al fine di evitare di dover scrivere ogni volta che si dichiara una variabile struct s_complesso è possibile definire un nuovo tipo apposito:

```
typedef struct s_complesso complesso;
```

In questo modo abbiamo introdotto un nuovo tipo che si affianca ad int, char, ... che si chiama complesso ed è possibile utilizzarlo nella dichiarazione di variabili, come mostrato di seguito:

```
struct s_complesso {
float reale;
float immaginaria;
};
typedef struct s_complesso complesso;
...
void main()
{
...int a, b;
complesso x, y;
}
```


Frequentemente la dichiarazione del tipo mediante la typedef viene fatta contemporaneamente alla dichiarazione della struct, secondo la seguente sintassi:

typedef struct *nome_struttura* {
lista dei campi (tipo-nome)
} *nome_tipo_struttura*;

La dichiarazione è più compatta.
Riportata all'esempio precedente, il codice che si ottiene è il seguente:

typedef struct s_complesso {
float reale;
float immaginaria;
} **complesso;**

STRUTTURE DATI

Sommario

Introduzione
Strutture ricorsive
Allocazione dinamica della memoria
Strutture concatenate
Pile
Code

Introduzione

Strutture dati dinamiche – Il numero dei loro elementi crescono e decrescono durante l'esecuzione del programma

Liste concatenate – Si possono inserire ed eliminare elementi in qualunque posizione all'interno della lista

Pile – Inserimenti ed eliminazioni possono essere fatti solo sulla "testa" della pila

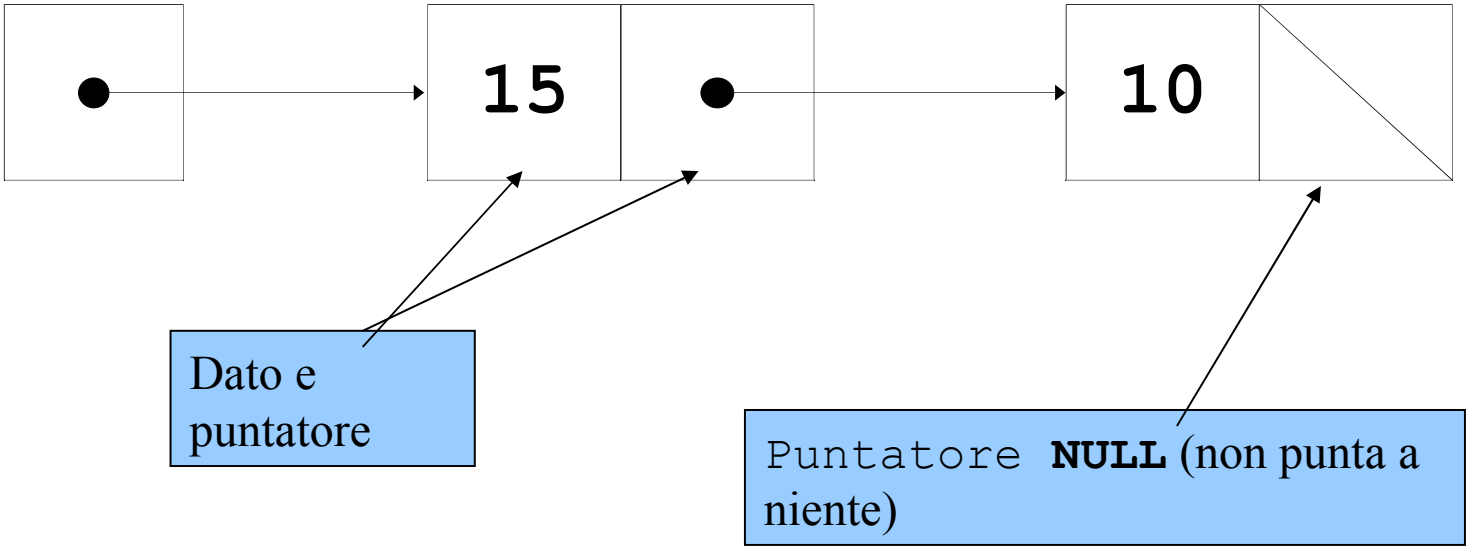
Code – Gli inserimenti possono essere fatti solo in "fondo" alla coda e le eliminazioni solo in "testa"

Alberi binari – Consentono la ricerca e l'ordinamento di dati in maniera veloce ed efficiente, come pure l'eliminazione e l'inserimento

Strutture ricorsive

- Strutture ricorsive
  - Sono strutture che contengono un puntatore a strutture dello stesso tipo
  - Possono essere collegate fra loro per formare strutture dati utili come: liste, code, pile ed alberi
  - Terminano con un puntatore **`NULL`** (`0`)

- Esempio di due strutture ricorsive collegate fra loro:



Strutture ricorsive (cont.)

```
struct node {
   int data;
   struct node *nextPtr;
}
```

- **nextPtr** – punta ad un oggetto di tipo **node**
  - Questa struttura è detta concatenata o ricorsiva – collega un **node** ad un altro **node**

La memoria dinamica

Tutte le variabili dichiarate ed utilizzate come array venivano allocate in **modo statico**, riservando loro spazio nella porzione di memoria denominata **stack,** che è destinata alla memoria statica.

Il compilatore vede dal tipo della variabile, al momento della dichiarazione, quanti byte devono essere allocati.

Per staticità si intende che i dati non cambieranno di dimensione nella durata del programma (si ricordi il vincolo di dimensionare opportunamente un array, eventualmente sovradimensionandolo).

Esiste una porzione di memoria denominata **Heap** ('mucchio' in italiano) dove è possibile allocare porzioni di memoria in **modo dinamico** durante l'esecuzione del programma, a fronte di richieste di spazio per variabili.

Allocazione dinamica

Con questo metodo di allocazione è possibile allocare n byte di memoria per un tipo di dato (n sta per la grandezza di byte che devono essere riservati per quel tipo di dato).

A questo scopo esistono specifiche funzioni della libreria standard **(malloc** e **free**) per **l'allocazione e il rilascio della memoria.**

Per identificare la dimensione della memoria da allocare dinamicamente, si utilizza l'operatore **sizeof** che prende come parametro il tipo di dato (int, float, ...) e restituisce il **numero di byte necessari per memorizzare un dato di quel tipo.**

Si ricordi che il numero di byte necessari per memorizzare un numero intero dipende dall'architettura del calcolatore e dal compilatore stesso. In generale questo valore è pari a 4 byte, ma potrebbe essere differente su architetture diverse. Per cui, onde evitare di riservare una quantità di memoria sbagliata, è opportuno far uso di tale funzione.

malloc

La funzione C per allocare dinamicamente uno spazio di memoria per una variabile è la seguente:

void * malloc(size_t);

La funzione, appartenente alla libreria standard stdlib.h riserva uno blocco di memoria di m byte dalla memoria heap e restituisce il puntatore a tale blocco.

Nel caso lo spazio sia esaurito, restituisce il puntatore nullo (NULL).

Quindi, per poter sfruttare la possibilità di allocare dinamicamente della memoria è necessario dichiarare delle variabili che siano dei puntatori, a cui verrà assegnato il valore dell'indirizzo del blocco richiesto quando si allocherà della memoria.

Ad esempio, per poter allocare dello spazio per una variabile intera è necessario aver dichiarato una variabile puntatore ad intero e poi nel corpo del programma aver chiesto lo spazio in memoria mediante la funzione malloc, come segue:

...

int *numx; **/*-1-*/**

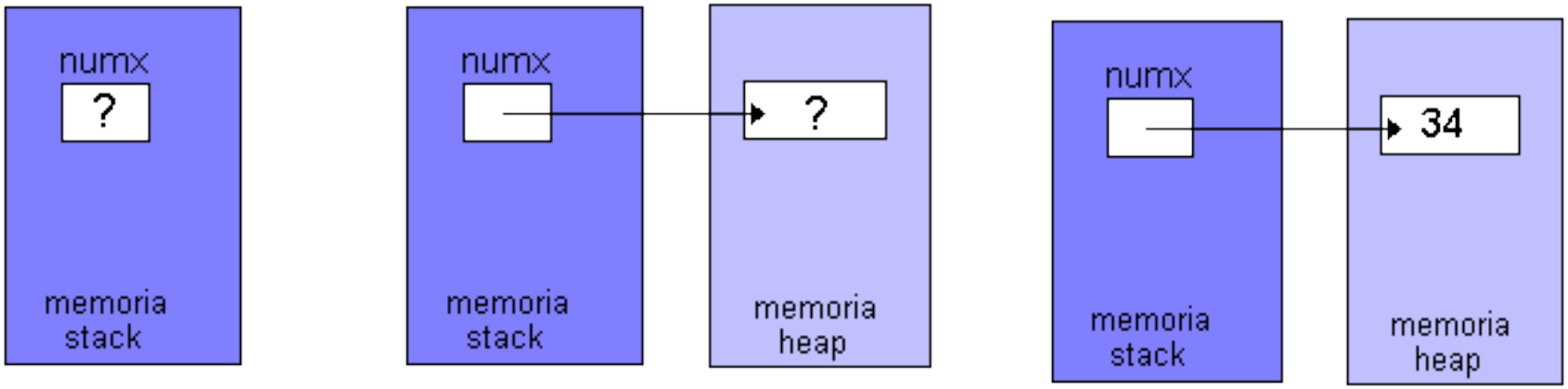
...

numx = (int *) malloc(sizeof(int)); **/*-2-*/**

...

***numx = 34;** **/*-3-*/**

Poichè la malloc restituisce un puntatore generico void *, viene fatto un riferimento esplicito al tipo intero, (int *). La situazione della memoria, corrispondente ai punti -1-, -2- e -3- è mostrata nella figura seguente:



-1- **-2-** **-3-**

Rilascio della memoria allocata dinamicamente

Quando la memoria allocata dinamicamente non serve più è opportuno liberarla, in modo tale che possa essere riutilizzata, riducendo i rischi di esaurire la memoria.

La funzione **free** effettua questa operazione; il prototipo è il seguente:

void free(void *);

La memoria riceve in ingresso un parametro: il puntatore alla memoria che deve essere liberata.

Una volta eseguita l'istruzione fare accesso al puntatore senza prima riassegnarlo, se non per verificare che punti a NULL, causa un errore durante l'esecuzione.

È anche possibile allocare dinamicamente un numero di byte sufficienti a contenere più dati dello stesso tipo, ossia un ***array* allocato dinamicamente.**

Si consideri il seguente stralcio di codice, che dopo aver chiesto all'utente quanti dati intende inserire, alloca dinamicamente la memoria per poi procedere nell'elaborazione:

```
...
int *Numeri, n;
 ...
printf("Quanti dati si desidera inserire?");
scanf("%d", &n);              /* numero di dati - omesso controllo di validità*/
Numeri = (int *)malloc(n * sizeof(int));      /*vengono allocati
                                               n*numero_byte_per_un_intero */
for(i = 0; i < n; i++)
{
      printf("Inserisci il dato %d: " i+1);
      scanf("%d", &Numeri[i]);
}
...
```


Le liste concatenate

Una lista concatenata è una sequenza di nodi in cui ogni nodo è *collegato* al nodo successivo: è possibile aggiungere collegare nella lista un numero qualsivoglia di nodi, eliminarli, ordinarli in base ad un qualche criterio.

Si accede alla lista concatenata mediante un puntatore al primo elemento della lista, da lì in poi ogni elemento punterà a quello successivo.

Per convenzione, l'ultimo nodo punterà a NULL ad indicare il termine della lista.

Ogni nodo della lista, oltre a mantenere il collegamento all'elemento successivo **memorizza anche i dati veri e propri che devono essere gestiti**: l'infrastruttura della lista è un *accessorio* per poter richiedere la memoria di volta in volta in base alle esigenze.

Infatti i nodi vengono creati solo quando c'è un nuovo dato da memorizzare.

E' una struttura utile per operazioni di ricerca e ordinamento di dati

Strutture per la realizzazione dei nodi delle liste

Per costruire una lista concatenata solitamente si dichiara un tipo di dato per i nodi della lista, i cui campi sono i dati veri e propri ed un campo di tipo puntatore. Ad es.:

```
typedef struct nodo_s {
 char carattere;
int frequenza;
struct nodo_s * prox;
} nodo_t;
```

Di seguito quindi struct nodo_s costituisce un nome alternativo per il tipo nodo_t.

Nella parte di dichiarazione del tipo di utilizza poi struct nodo_s * per il puntatore (prox) ad un altro elemento dello stesso tipo.

La testa della lista

La testa della lista ha come obiettivo quello di indicare sempre il primo nodo della lista.

Si tratta di un puntatore ad un elemento di tipo nodo_t, non serve un intero nodo in quanto non memorizza un dato, ma solo un puntatore ad un nodo.

Inizialmente, la lista è vuota, per cui all'inizio del programma si inizializza la testa a NULL.

```
void main()
{
nodo_t * testa;
 ...
testa = NULL;
 ...
}
```


Creare nuovi nodi

Per creare un nuovo nodo è necessario allocare la memoria, verificare che non sia stata esaurita e quindi memorizzare il valore dei dati negli appositi campi della struttura del nodo.

```
...
nodo_t *n1;
n1 = (nodo_t *)malloc(sizeof(nodo_t)); /* alloca memoria per un elemento
                                          di tipo nodo_t */
if (n1)
{
/* verifica che non ci siano stati problemi di memoria */
n1->carattere = val_c;
n1->frequenza = 1;
n1->prox = NULL;                    /* inizializzato a puntare a NULL */
                                    /* verrà modificato quando inserito nella lista
*/
... inserimento nella lista ...
}
```

Una volta inserito il nodo nella lista il puntatore n1 potrà essere utilizzato per creare un nuovo nodo.

Attraversamento/scorrimento della lista

In numerosi situazioni è necessario processare tutti i nodi della lista o comunque si deve esaminarli alla ricerca di uno specifico.

Per attraversare la lista è necessario posizionarsi sul primo nodo, mediante la testa, e poi seguire la catena di puntatori all'elemento successivo.

Se si desidera processare tutti i nodi ci si fermerà nell'attraversamento quando la lista termina, in altri casi (come nell'esempio successivo dell'inserimento in coda) è desiderabile fermarsi sull'ultimo elemento della lista.

Lo stralcio di codice proposto nel seguito effettua la stampa di tutti i nodi della lista:

```
...
nodo_t *testa, *temp;
testa = NULL;
 ...
/* stampa della lista */
temp = testa;
while(temp){       /*ci si ferma quando temp = NULL*/
printf("%c: %d\n", temp->carattere, temp->frequenza);
temp = temp->prox;
}
```


Operazioni sui nodi

Le operazioni fondamentali per la gestione della lista concatenata sono le seguenti:

- inserire un nodo all'inizio della lista (*in testa*)
- inserire un nodo alla fine della lista (*in coda*)
- eliminare un nodo dalla lista

Ci sono altre operazioni che è possibile svolgere, come per esempio l'inserimento di un nodo in un punto ben preciso della struttura per mantenere o realizzare un ordinamento dei nodi, che si basano su quelle fondamentali citate.

Le operazioni di inserimento ed eliminazione consistono nell'aggiornare opportunamente i puntatori di alcuni nodi in modo tale che il nuovo nodo venga incluso nella catena oppure in modo che ne venga escluso.

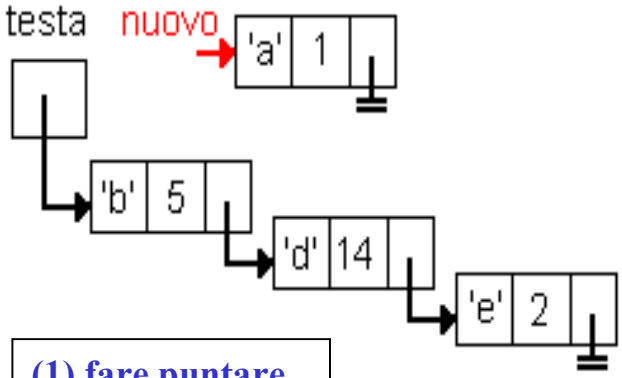
Nell'affrontare ognuna di queste operazioni è necessario considerare i casi particolari in cui la lista è vuota e in cui c'è un unico elemento (in alcune situazioni anche il caso con solo due nodi può risultare speciale).

Si ricordi, infatti, che accedere ad un puntatore nullo crea un errore durante l'esecuzione.

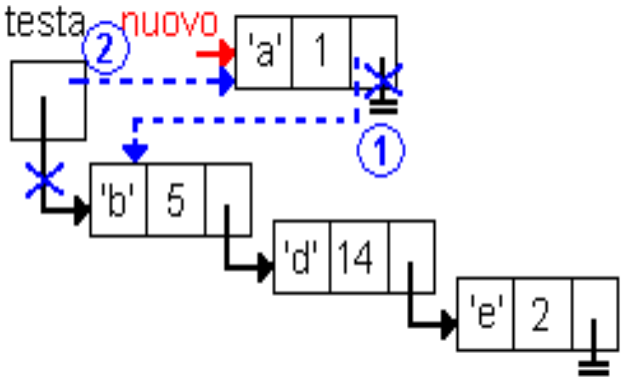
Inserimento in testa

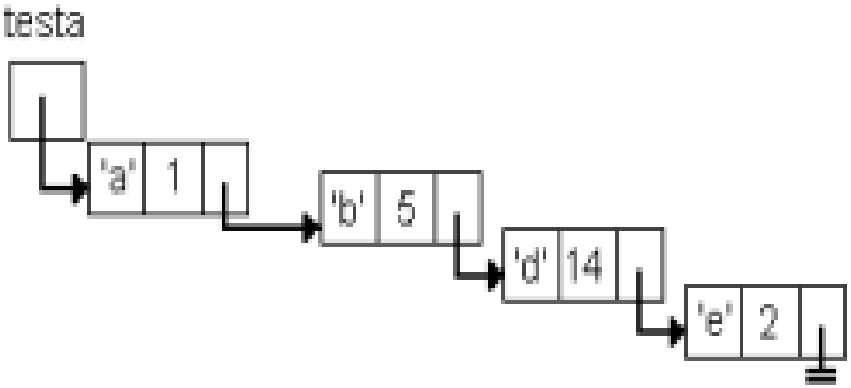
Viene trattato in primo luogo il caso generale in cui la lista ha almeno un elemento.

La figura seguente mostra quali sono i passi da svolgere per inserire un nodo all'inizio della lista, come nuovo primo nodo.



(1) fare puntare il nuovo nodo (accessibile tramite il puntatore nuovo) all'attuale primo nodo (quello puntato dalla testa)





(2) fare puntare la testa al nuovo (primo) nodo. Automaticamente vengono eliminati i vecchi collegamenti.

Ci si ricordi che aggiornando un puntatore per indirizzarlo ad un nodo diverso da quello attualmente puntato, elimina il precedente collegamento.

Lo stralcio di codice che effettua l'inserimento in testa è illustrato qui di seguito, in cui si suppone che nuovo punti ad un nuovo nodo, con i campi già impostati al valore corretto, e testa che punta correttamente al primo nodo della lista:

```
...
nuovo->prox = testa;
testa = nuovo;
 ...
```

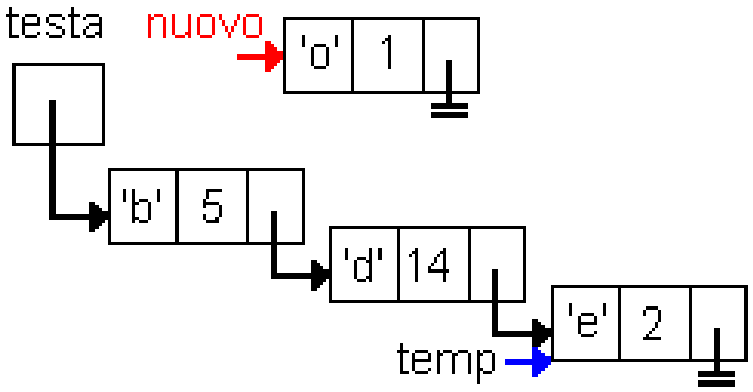
Nel caso in cui la lista sia vuota, testa punterà a NULL ad indicare questa situazione.

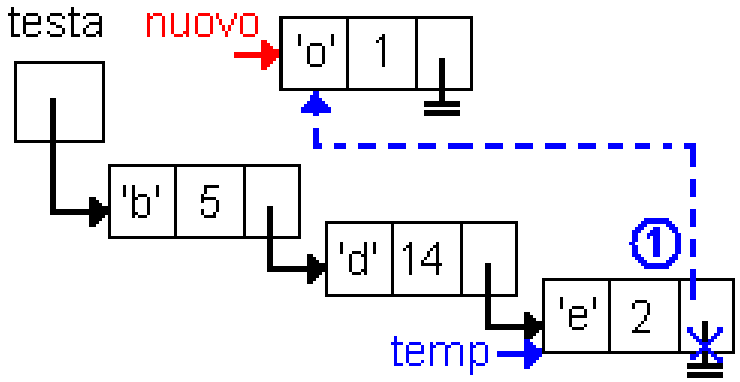
Il codice prima indicato mantiene la propria validità: infatti dopo la prima istruzione nuovo->prox punterà a NULL, il che indica che è l'ultimo elemento (essendo l'unico è così). L'effetto della seconda istruzione non cambia.

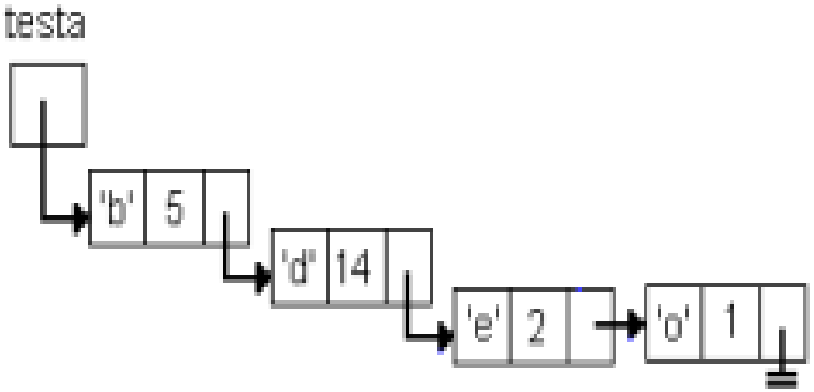
Dall'analisi si deduce che nell'inserimento di un nuovo elemento in testa alla lista non è necessario trattare esplicitamente il caso di lista vuota.

Inserimento in coda

Per effettuare l'inserimento in coda è necessario scorrere tutta la lista e portarsi sull'ultimo elemento, facendo in modo che quando si sta per effettuare l'operazione di aggiornamento, il puntatore indirizzi l'ultimo nodo. Le figure seguenti mostrano la sequenza di operazioni.







Lo stralcio di codice che effettua lo scorrimento della lista fino all'ultimo elemento e l'operazione di inserimento in coda è riportato qui di seguito.

```
...
nodo_t *testa, *temp, *nuovo;
temp = testa;
while(temp->prox){              /* scansione della lista */
temp = temp->prox;
}                   /*quando si arriva qui temp punta all'ultimo
                    elemento*/
temp->prox = nuovo;             /* inserimento */
```

L'istruzione nuovo->prox = NULL; non è stata eseguita in quanto viene fatta nel momento in cui si crea un nuovo nodo.

In questo caso è necessario gestire il caso specifico della lista vuota in quanto se la lista è vuota (temp = NULL) l'accesso successivo al puntatore temp con l'istruzione temp->prox causerebbe un errore durante l'esecuzione.

Lo stralcio di codice che gestisce anche il caso lista vuota è il seguente:

```
...
temp = testa;
if (temp){
    while(temp->prox) /*si arriva solo se temp non è NULL*/
        temp=temp->prox;
                        /* temp punta all'ultimo */
    temp->prox = nuovo;
    }  else             /*caso lista vuota*/
        testa = nuovo;
```

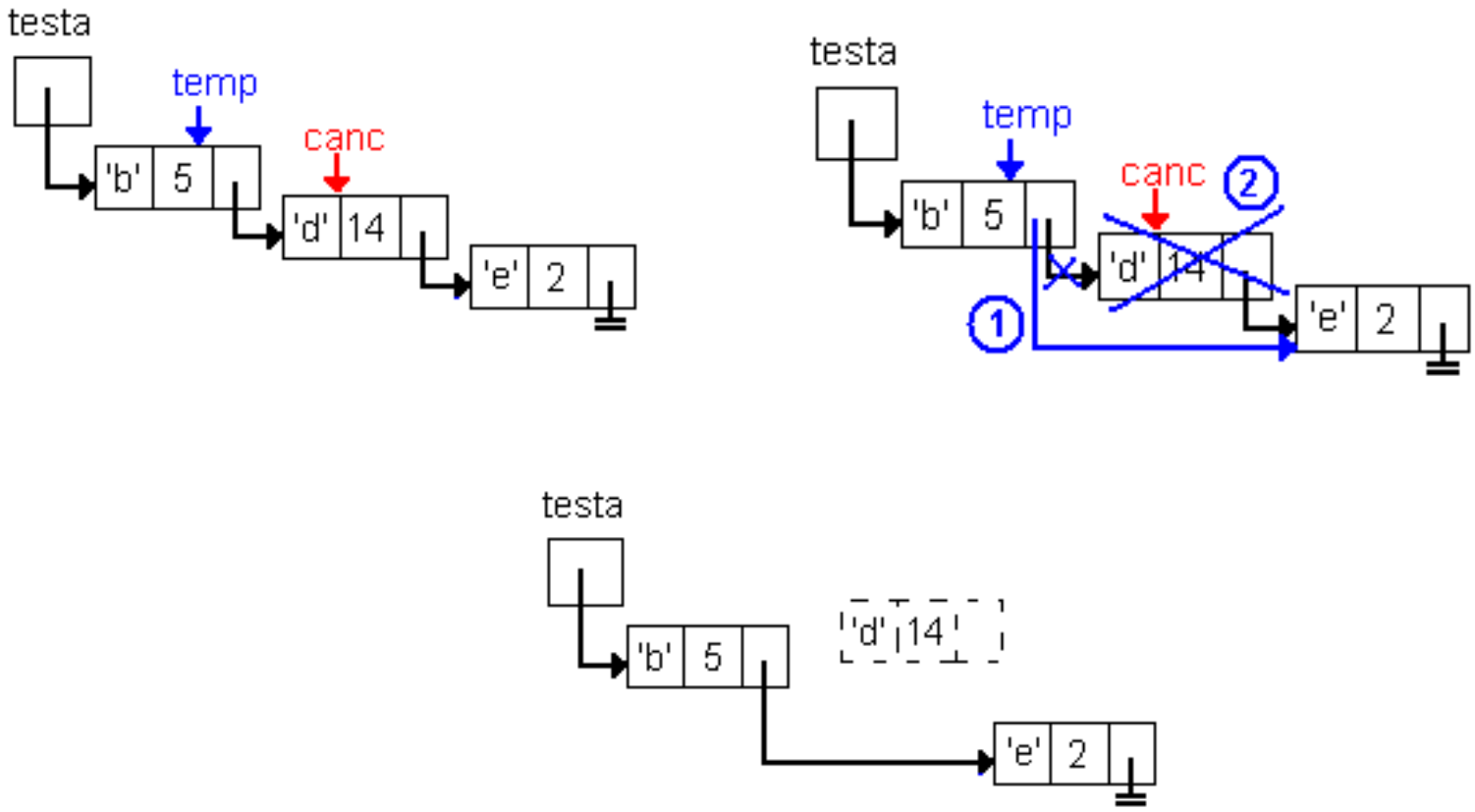

Eliminazione di un elemento della lista

Questa è l'operazione che richiede maggior attenzione in quanto sono necessari due puntatori d'appoggio per poter svolgere l'eliminazione senza perdere una parte della lista concatenata.

Per eliminare un nodo, oltre a sistemare opportunamente i puntatori dei nodi che rimangono nella lista, è anche necessario liberare la memoria del nodo mediante l'istruzione free (per questo motivo serve un secondo puntatore).

Verrà mostrato il caso più generale in cui si desidera eliminare un nodo che si trova in mezzo alla lista facendo poi alcune considerazioni sugli altri casi.

Le figure seguenti mostrano la sequenza delle operazioni da svolgere.



Il puntatore canc punta al nodo da eliminare, il puntatore di supporto temp punta al nodo precedente a quello da cancellare: si scorrerà quindi la lista fino a trovare il punto in cui fermare temp e di conseguenza si definirà canc, quindi si provvederà a spostare i puntatori e a chiamare la free.

Il codice è riportato qui di seguito, in cui non si considera il caso generale.

```
...
nodo_t *canc, *temp;
char valore_cercato;
 ...
 temp = testa;                                   /* i casi speciali non sono trattati */
while(temp->prox && temp->prox->carattere != valore_cercato)
temp = temp->prox;          /* quando si e' qui, o non si è trovato l'elemento o si è
                               su quello che precede quello da eliminare */
if (temp->prox->carattere == valore_cercato){          /* si può procedere
                                                          all'eliminazione */
canc = temp->prox;                          /* punta all'elemento da cancellare */
temp->prox = canc->prox;                    /* spostamento dei puntatori */
free(canc);                                 /* libera la memoria */
}                    /*non c'è un else perchè se l'elemento non c'è non si fa nulla */
```

Casi particolari: lista vuota, l'elemento è il primo della lista, nella lista c'è un solo elemento

Sono le situazioni che provocherebbero un errore nell'esecuzione del precedente codice privo di controlli.

Il seguente sottoprogramma gestisce tutte le casistiche citate, ipotizzando che la lista non sia ordinata (altrimenti sarebbe possibile interrompere la scansione della lista non appena si è terminato di eliminare un elemento dalla lista ed il successivo non è da eliminare).

```
nodo_t * EliminaN(nodo_t * head, int val)
{
nodo_t * tmp, *canc;
tmp = head; /* se la lista e' vuota non c'e'
                        nulla da eliminare */

/* eliminazione del primo elemento della
lista */
while(tmp && tmp->frequenza == val){
tmp = head;
head = head->prox;
free(tmp);
 }
 /* eliminazione di un elemento qualsiasi */
/* tmp punta al primo elemento */
/* che non e' da eliminare (altrimenti
saremmo ancora nel ciclo */
while(tmp)
if(tmp->prox && tmp->prox->frequenza
== val){
canc = tmp->prox;
tmp->prox = tmp->prox->prox;
free(canc);
}
else
/* si va avanti solo se non si cancella un
nodo */
tmp = tmp->prox;
return head;
}
```


Crea e visualizza una lista di interi

```
/* 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 <stdlib.h>

/* Struttura degli elementi della lista */
struct elemento {
  int inf;
  struct elemento *pun;
};

struct elemento *creaLista();
void visualizzaLista(struct elemento *);

main()
{
struct elemento *puntLista;                  /* puntatore alla testa della lista */
puntLista = creaLista();                     /* chiamata funzione per creare la lista */
visualizzaLista(puntLista);     /* chiamata funzione per visualizzare la lista */
}
```


```
/* Funzione per l'accettazione dei valori
immessi e la creazione della lista. Restituisce
il puntatore alla testa  */
struct elemento *creaLista()
{
struct elemento *p, *paus;
int i, n;

printf("\n Da quanti elementi e' 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 1a 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 %da
informazione: ", i);
   scanf("%d", &paus->inf);
  }
  paus->pun = NULL;          /*
marca di fine lista */
}
return(p);
}
```

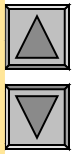

```
/* Funzione per la visualizzazione della lista.
   Il parametro in ingresso è il puntatore alla testa */
void visualizzaLista(struct elemento *p)
{
printf("\npuntLista---> ");

/* 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");
}
```


Creazione lista e ricerca elemento maggiore

```
/* Accetta in ingresso una sequenza di interi e li memorizza in una lista. La sequenza termina
quando viene immesso il valore zero. La lista creata viene visualizzata. Determina il maggiore
della lista */
#include <stdio.h>
#include <stdlib.h>

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

struct elemento *creaLista2();
void visualizzaLista(struct elemento *);
int maggioreLista(struct elemento *);

main()
{
struct elemento *puntLista;                    /* puntatore alla testa  della lista */
puntLista = creaLista2();                      /* chiamata funzione per creare la lista  */
visualizzaLista(puntLista);                /* chiamata funzione per  visualizzare la lista */
/* Stampa il valore di ritorno della funzione maggioreLista() */
printf("\nIl maggiore e': %d\n\n", maggioreLista(puntLista));
}
```


```
/* Accetta in ingresso una sequenza  di interi e li memorizza in una lista. Il numero di interi che compongono  la sequenza termina con il valore zero  */
struct elemento *creaLista2()
{
struct elemento *p, *paus;
struct elemento x;

printf("\nInserisci un'informazione (0 per fine lista): ");
scanf("%d", &x.inf);

if(x.inf==0)  p = NULL;          /* lista vuota */
else {
/* Creazione del primo elemento */
 p = (struct elemento *)malloc(sizeof(struct elemento));

 p->inf = x.inf;
 paus=p;
while(x.inf!=0) {

  printf("\nInserisci un'informazione (0 per fine lista): ");

  scanf("%d", &x.inf);

  if(x.inf!=0) {

   /* Creazione dell'elemento successivo */

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

   paus = paus->pun;      /* attualizzazione di paus */

   paus->inf = x.inf;     /* inserimento dell'informazione nell'elemento  */

  }

  else

   paus->pun = NULL;      /* Marca di fine lista*/

 }

}

return(p);

}
```


```
/* Determina il maggiore della lista. Il parametro in ingresso è il puntatore alla testa */
maggioreLista(struct elemento *p)
{
int max = INT_MIN;
/* Ciclo di scansione della lista */
while(p != NULL) {
  if(p->inf > max)
    max = p->inf;
  p = p->pun;            /* scorre di un elemento in avanti */
}
return(max);
}

/* Visualizza la lista */
void visualizzaLista(struct elemento *p)
{
printf("\npuntLista---> ");

/* Ciclo di scansione della lista */
while(p!=NULL) {
  printf("%d", p->inf);    /* visualizza il campo informazione */
  printf("---> ");
  p = p->pun;            /* scorre di un elemento in avanti */
}
printf("NULL\n\n");
}
```

Outline

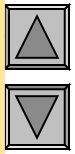
```
1
2     /* Operazioni su una lista concatenata */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  struct listNode {    /* struttura ricorsiva */
7     char data;
8     struct listNode *nextPtr;
9  };
10
11 typedef struct listNode ListNode;
12 typedef ListNode *ListNodePtr;
13
14 void insert( ListNodePtr *, char );
15 char delete( ListNodePtr *, char );
16 int isEmpty( ListNodePtr );
17 void printList( ListNodePtr );
18 void instructions( void );
19
20 int main()
21 {
22    ListNodePtr startPtr = NULL;
23    int choice;
24    char item;
25
26    instructions();  /* Mostra il menu */
27    printf( "? " );
28    scanf( "%d", &choice );
```

Definizione di struct

Prototipi delle funzioni
Insert = inserimento
Delete = cancellazione
isEmpty = lista vuota
printList = stama della lista
Instructions = menu

Scelta da menu (choice)

```
29
30    while ( choice != 3 ) {
31
32       switch ( choice ) {
33          case 1:
34             printf( "Enter a character: " );
35             scanf( "\n%c", &item );
36             insert( &startPtr, item );
37             printList( startPtr );
38             break;
39          case 2:
40             if ( !isEmpty( startPtr ) ) {
41                printf( "Enter character to be deleted: " );
42                scanf( "\n%c", &item );
43
44                if ( delete( &startPtr, item ) ) {
45                   printf( "%c deleted.\n", item );
46                   printList( startPtr );
47                }
48                else
49                   printf( "%c not found.\n\n", item );
50             }
51             else
52                printf( "List is empty.\n\n" );
53
54             break;
55          default:
56             printf( "Invalid choice.\n\n" );
57             instructions();
58             break;
59       }
```

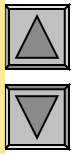


Outline

Switch:

1-Iniserimento e stampa lista

2-Cancellazione con controllo su lista vuota e stampa lista

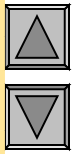


Outline

```
60
61       printf( "? " );
62       scanf( "%d", &choice );
63    }
64
65    printf( "End of run.\n" );
66    return 0;
67 }
68
69 /* Print the instructions */
70 void instructions( void )
71 {
72    printf( "Enter your choice:\n"
73            "   1 to insert an element into the list.\n"
74            "   2 to delete an element from the list.\n"
75            "   3 to end.\n" );
76 }
77
78 /* Insert a new value into the list in sorted order */
79 void insert( ListNodePtr *sPtr, char value )
80 {
81    ListNodePtr newPtr, previousPtr, currentPtr;
82
83    newPtr = malloc( sizeof( ListNode ) );
84
85    if ( newPtr != NULL ) {     /* is space available */
86       newPtr->data = value;
87       newPtr->nextPtr = NULL;
88
89       previousPtr = NULL;
90       currentPtr = *sPtr;
```

Stampa del menu

Inserimento



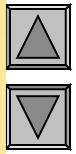
Outline

Inserimento (cont.) con controllo su lista vuota e sulla memoria disponibile

Cancellazione con liberazione della memoria (free)

```
91
92       while ( currentPtr != NULL && value > currentPtr->data ) {
93          previousPtr = currentPtr;          /* walk to ...   */
94          currentPtr = currentPtr->nextPtr;  /* ... next node */
95       }
96
97       if ( previousPtr == NULL ) {
98          newPtr->nextPtr = *sPtr;
99          *sPtr = newPtr;
100      }
101      else {
102         previousPtr->nextPtr = newPtr;
103         newPtr->nextPtr = currentPtr;
104      }
105   }
106   else
107      printf( "%c not inserted. No memory available.\n", value );
108}
109
110/* Delete a list element */
111char delete( ListNodePtr *sPtr, char value )
112{
113   ListNodePtr previousPtr, currentPtr, tempPtr;
114
115   if ( value == ( *sPtr )->data ) {
116      tempPtr = *sPtr;
117      *sPtr = ( *sPtr )->nextPtr;  /* de-thread the node */
118      free( tempPtr );             /* free the de-threaded node */
119      return value;
120   }
```

```
121   else {
122      previousPtr = *sPtr;
123      currentPtr = ( *sPtr )->nextPtr;
124
125      while ( currentPtr != NULL && currentPtr->data != value ) {
126         previousPtr = currentPtr;          /* walk to ...   */
127         currentPtr = currentPtr->nextPtr;  /* ... next node */
128      }
129
130      if ( currentPtr != NULL ) {
131         tempPtr = currentPtr;
132         previousPtr->nextPtr = currentPtr->nextPtr;
133         free( tempPtr );
134         return value;
135      }
136   }
137
138   return '\0';
139}
140
141/* Return 1 if the list is empty, 0 otherwise */
142int isEmpty( ListNodePtr sPtr )
143{
144   return sPtr == NULL;
145}
146
147/* Print the list */
148void printList( ListNodePtr currentPtr )
149{
150   if ( currentPtr == NULL )
151      printf( "List is empty.\n\n" );
152   else {
153      printf( "The list is:\n" );
```



Outline

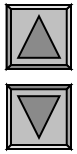
Cancellazione (cont.) con controllo su lista vuota

Lista vuota: Se la lista è vuota restituisce il valore 1, altrimenti 0

Stampa della lista

```
154
155       while ( currentPtr != NULL ) {
156          printf( "%c --> ", currentPtr->data );
157          currentPtr = currentPtr->nextPtr;
158       }
159
160       printf( "NULL\n\n" );
161    }
```

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A
The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL
```



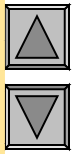
Outline

Stampa della lista (cont.)

Esecuzione e output

Pila o stack

- Stack
  - I nodi possono essere inseriti o eliminati solo dalla cima (testa)
  - Simile ad una pila di piatti
  - Struttura: Last-In, First-Out (LIFO)
  - La fine della pila è indicata da un puntatore **null**
  - E' una versione con vincoli della lista concatenata
- Push (inserimento)
  - Aggiunge un nuovo nodo alla testa della pila
- Pop (cancellazione)
  - Elimina un nodo dalla testa della pila
  - Può effettuare il salvataggio dell'elemento eliminato



Outline

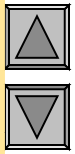
```
1  /* Fig. 12.8: fig12_08.c
2     dynamic stack program */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  struct stackNode {   /* self-referential structure */
7     int data;
8     struct stackNode *nextPtr;
9  };
10
11 typedef struct stackNode StackNode;
12 typedef StackNode *StackNodePtr;
13
14 void push( StackNodePtr *, int );
15 int pop( StackNodePtr * );
16 int isEmpty( StackNodePtr );
17 void printStack( StackNodePtr );
18 void instructions( void );
19
20 int main()
21 {
22    StackNodePtr stackPtr = NULL;  /* punta alla testa della pila*/
23    int choice, value;
24
25    instructions();
26    printf( "? " );
27    scanf( "%d", &choice );
28
```

Definizione della struct

Definizione delle funzioni

Menu per scelta dell'input

```
29    while ( choice != 3 ) {
30
31       switch ( choice ) {
32          case 1:      /* push value onto stack */
33             printf( "Enter an integer: " );
34             scanf( "%d", &value );
35             push( &stackPtr, value );
36             printStack( stackPtr );
37             break;
38          case 2:      /* pop value off stack */
39             if ( !isEmpty( stackPtr ) )
40                printf( "The popped value is %d.\n",
41                        pop( &stackPtr ) );
42
43             printStack( stackPtr );
44             break;
45          default:
46             printf( "Invalid choice.\n\n" );
47             instructions();
48             break;
49       }
50
51       printf( "? " );
52       scanf( "%d", &choice );
53    }
54
55    printf( "End of run.\n" );
56    return 0;
57 }
58
```

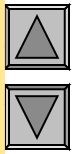


Outline

Switch:

1-Inserimento (push) in testa alla pila

2-Estrazione (pop) di un elemento dalla testa della pila

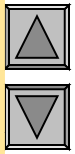


Outline

Funzione menu

Funzione inserimento

```
59 /* Print the instructions */
60 void instructions( void )
61 {
62    printf( "Enter choice:\n"
63            "1 to push a value on the stack\n"
64            "2 to pop a value off the stack\n"
65            "3 to end program\n" );
66 }
67
68 /* Insert a node at the stack top */
69 void push( StackNodePtr *topPtr, int info )
70 {
71    StackNodePtr newPtr;
72
73    newPtr = malloc( sizeof( StackNode ) );
74    if ( newPtr != NULL ) {
75       newPtr->data = info;
76       newPtr->nextPtr = *topPtr;
77       *topPtr = newPtr;
78    }
79    else
80       printf( "%d not inserted. No memory available.\n",
81               info );
82 }
83
```



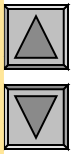
Outline

Funzione eliminazione

Funzione stampa

```
84 /* Remove a node from the stack top */
85 int pop( StackNodePtr *topPtr )
86 {
87    StackNodePtr tempPtr;
88    int popValue;
89
90    tempPtr = *topPtr;
91    popValue = ( *topPtr )->data;
92    *topPtr = ( *topPtr )->nextPtr;
93    free( tempPtr );
94    return popValue;
95 }
96
97 /* Print the stack */
98 void printStack( StackNodePtr currentPtr )
99 {
100   if ( currentPtr == NULL )
101      printf( "The stack is empty.\n\n" );
102   else {
103      printf( "The stack is:\n" );
104
105      while ( currentPtr != NULL ) {
106         printf( "%d --> ", currentPtr->data );
107         currentPtr = currentPtr->nextPtr;
108      }
109
110      printf( "NULL\n\n" );
111   }
112}
113
```

```
114 /* Is the stack empty? */
115 int isEmpty( StackNodePtr topPtr )
116 {
117    return topPtr == NULL;
118 }
```

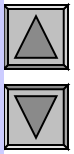


Outline

Funzione controllo pila vuota

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL
```

Output



Outline

Output

```
? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

/* GESTIONE DI UNA PILA

Operazioni di inserimento, eliminazione e visualizzazione. Utilizza una lista lineare per implementare la pila */

```
#include <stdio.h>
#include <malloc.h>

struct elemento {
  int inf;
  struct elemento *pun;
};
/* dichiarazione di funzioni per inserimento,cancellazione, pila vuota*/
void gestionePila(void);
struct elemento *inserimento(struct elemento *, int ele);
struct elemento *eliminazione(struct elemento *, int *);
int pilaVuota(struct elemento *);
void visualizzazionePila(struct elemento *);

main()
{
  gestionePila();
}
```


```
void gestionePila(void)                    /* Menu di scelta per inserim., cancell., visualizz., fine*/
{
struct elemento *puntTesta = NULL;
int scelta = -1, ele;
char pausa;
while(scelta!=0) {
 printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
 printf("\t\tESEMPIO UTILIZZO STRUTTURA ASTRATTA: PILA");
 printf("\n\n\n\t\t\t 1. Per inserire un elemento");
 printf("\n\n\t\t\t 2. Per eliminare un elemento");
 printf("\n\n\t\t\t 3. Per visualizzare la pila");
 printf("\n\n\t\t\t 0. Per finire");
 printf("\n\n\n\t\t\t\t Scegliere una opzione: ");
 scanf("%d", &scelta);
 printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

 switch(scelta) {
  case 1:
    printf("Inserire un elemento: ");
    scanf("%d", &ele);
    puntTesta = inserimento(puntTesta, ele);
    if(puntTesta==NULL) {
      printf("Inserimento impossibile: ");
      printf("memoria disponibile terminata");
      printf("\n\n Premere un tasto per continuare...");
      scanf("%c", &pausa);
    }
    break;
```


```
case 2:
      if(pilaVuota(puntTesta)) {
        printf("Eliminazione impossibile: pila vuota");
        printf("\n\n Premere un tasto per continuare...");
        scanf("%c", &pausa);
      }
      else {
        puntTesta = eliminazione(puntTesta, &ele);
        printf("Eliminato: %d", ele );
        printf("\n\n Premere un tasto per continuare...");
        scanf("%c", &pausa);
      }
      break;
   case 3:
      visualizzazionePila(puntTesta);
      printf("\n\n Premere un tasto per continuare...");
      scanf("%c", &pausa);
      break;
  }
}
}
```

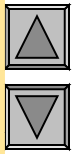

```
void visualizzazionePila(struct elemento *p)
{
struct elemento *paus = p;

printf("\n<------ Testa della pila ");
while(paus!=NULL) {
  printf("\n%d", paus->inf);
  paus = paus->pun;
}
}


struct elemento *inserimento(struct elemento *p, int ele)
{
struct elemento *paus;

paus = (struct elemento *)malloc(sizeof(struct elemento));

if(paus==NULL) return(NULL);

paus->pun = p;
p = paus;
p->inf = ele;
return(p);
}
```


```
struct elemento *eliminazione(struct elemento *p, int *ele)
{
struct elemento *paus;

*ele = p->inf;
paus = p;
p = p->pun;
free(paus);
return( p );
}


int pilaVuota(struct elemento *p)
{
if(p==NULL)
  return(1);
else
  return(0);
}
```


Coda

- Coda
  - Simile alla coda ad uno sportello o alla cassa del supermercato
  - ***First-In, First-Out (FIFO)***
  - I nodi sono eliminati dalla testa - *head*
  - I nodi sono inseriti in coda - *tail*

- Operazioni di inserimento ed eliminazione
  - Enqueue (inserimento)e dequeue (eliminazione)



Outline

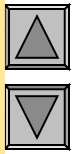
Definizione della struct

Prototipi delle funzioni:
Stampa
Coda vuota
Estrazione
Inserimento
Menu

Menu di scelta

```
1  /* Fig. 12.13: fig12 13.c
2     Operating and maintaining a queue */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  struct queueNode {    /* self-referential structure */
8     char data;
9     struct queueNode *nextPtr;
10 };
11
12 typedef struct queueNode QueueNode;
13 typedef QueueNode *QueueNodePtr;
14
15 /* function prototypes */
16 void printQueue( QueueNodePtr );
17 int isEmpty( QueueNodePtr );
18 char dequeue( QueueNodePtr *, QueueNodePtr * );
19 void enqueue( QueueNodePtr *, QueueNodePtr *, char );
20 void instructions( void );
21
22 int main()
23 {
24    QueueNodePtr headPtr = NULL, tailPtr = NULL;
25    int choice;
26    char item;
27
28    instructions();
29    printf( "? " );
30    scanf( "%d", &choice );
```

```
31
32    while ( choice != 3 ) {
33
34       switch( choice ) {
35
36          case 1:
37             printf( "Enter a character: " );
38             scanf( "\n%c", &item );
39             enqueue( &headPtr, &tailPtr, item );
40             printQueue( headPtr );
41             break;
42          case 2:
43             if ( !isEmpty( headPtr ) ) {
44                item = dequeue( &headPtr, &tailPtr );
45                printf( "%c has been dequeued.\n", item );
46             }
47
48             printQueue( headPtr );
49             break;
50
51          default:
52             printf( "Invalid choice.\n\n" );
53             instructions();
54             break;
55       }
56
57       printf( "? " );
58       scanf( "%d", &choice );
59    }
60
61    printf( "End of run.\n" );
62    return 0;
63 }
64
```



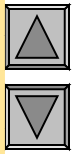
Outline

Switch:

1-Inserimento

2-Estrazione

3-Fine



Outline

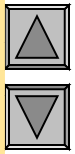
```
65 void instructions( void )
66 {
67    printf ( "Enter your choice:\n"
68             "   1 to add an item to the queue\n"
69             "   2 to remove an item from the queue\n"
70             "   3 to end\n" );
71 }
72
73 void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
74               char value )
75 {
76    QueueNodePtr newPtr;
77
78    newPtr = malloc( sizeof( QueueNode ) );
79
80    if ( newPtr != NULL ) {
81       newPtr->data = value;
82       newPtr->nextPtr = NULL;
83
84       if ( isEmpty( *headPtr ) )
85          *headPtr = newPtr;
86       else
87          ( *tailPtr )->nextPtr = newPtr;
88
89       *tailPtr = newPtr;
90    }
91    else
92       printf( "%c not inserted. No memory available.\n",
93               value );
94 }
95
```

Menu

Funzione di inserimento

Controlla che ci sia memoria disponibile (puntatore al nuovo elemento diverso da NULL)

isEmpty è uguale a 1 se la coda è vuota, altrimenti vale 0



Outline

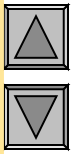
```
96 char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
97 {
98    char value;
99    QueueNodePtr tempPtr;
100
101   value = ( *headPtr )->data;
102   tempPtr = *headPtr;
103   *headPtr = ( *headPtr )->nextPtr;
104
105   if ( *headPtr == NULL )
106      *tailPtr = NULL;
107
108   free( tempPtr );
109   return value;
110}
111
112int isEmpty( QueueNodePtr headPtr )
113{
114   return headPtr == NULL;
115}
116
117void printQueue( QueueNodePtr currentPtr )
118{
119   if ( currentPtr == NULL )
120      printf( "Queue is empty.\n\n" );
121   else {
122      printf( "The queue is:\n" );
```

Funzione di eliminazione

Funzione di controllo coda vuota: se la condizione == è verificata, resituisce il valore 1,altrimenti 0

Funzione di stampa coda

```
123
124       while ( currentPtr != NULL ) {
125          printf( "%c --> ", currentPtr->data );
126          currentPtr = currentPtr->nextPtr;
127       }
128
129       printf( "NULL\n\n" );
130    }
131 }
```

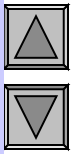


Outline

Funzione di stampa coda (cont.)

Output

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C --> NULL
```



Outline

Output (cont.)

```
? 2
A has been dequeued.
The queue is:
B --> C --> NULL

? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.

Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```