Capitolo 12 (Deitel)
Le strutture dati dinamiche

Sommario

© 2000 Prentice Hall, Inc. All rights reserved.

12.1 - Introduzione

- Strutture dati dinamiche
  - In molte situazioni, è importante avere a disposizione delle strutture dati dinamiche, ovvero che abbiano dimensione **variabile**
  - Possono **crescere e/o decrescere durante l'esecuzione del programma**
  - La loro **occupazione di memoria varia** in modo corrispondente a runtime
  - Il contesto tipico è quello delle collezioni di oggetti che devono poter crescere in dimensione (es. un'agenda contenente degli indirizzi)
- Tramite le strutture dati dinamiche è possibile creare:
  - ***Liste concatenate*:** sequenze dinamiche di dati in cui l'inserimento e/o l'eliminazione di elementi può avvenire in qualsiasi posizione della lista
  - ***Pile (o stack):*** sequenze dinamiche di dati in cui sia l'inserimento che l'eliminazione di elementi può avvenire solo in cima alla pila (LIFO)
  - ***Code:*** sequenze dinamiche di dati in cui l'inserimento di elementi può avvenire solo alla fine (in coda) e l'eliminazione solo in testa (FIFO)
  - ***Alberi binari:*** sequenze dinamiche di dati organizzate in modo gerarchico, tale da rendere molto veloce la ricerca e l'ordinamento dei dati, permettendo inoltre di eliminare in modo efficiente eventuali elementi duplicati

© 2000 Prentice Hall, Inc. All rights reserved.

12.2 - Allocazione dinamica della memoria (1/2)

- I tipi di dato visti fino ad ora sono tutti **statici**
  - Non variano la propria dimensione nel corso dell'esecuzione
  - Vettori e stringhe non fanno eccezione alla regola
  - La loro dimensione è infatti prestabilita e va dichiarata esplicitamente al momento della definizione delle variabili
  - Se invece si volesse "allungare" di un elemento un vettore, si dovrebbe poter aumentare in corrispondenza la memoria allocata per il vettore
  - L'unico modo per farlo è utilizzare delle strutture dati dinamiche

- Allocazione dinamica della memoria
  - Per far uso delle strutture dinamiche è **necessario quindi poter allocare dinamicamente (a runtime) lo spazio in memoria** ad esse dedicato
    - Ovvero ottenere e rilasciare memoria durante l'esecuzione del programma
  - Il C offre due funzioni dedicate per allocare dinamicamente la memoria
  - **malloc(..)** permette di allocare a runtime una certa quantità di memoria
  - **free(..)** consente il rilascio della memoria allocata dalla funzione **malloc**
  - L'uso di queste funzioni coinvolge anche l'ormai noto operatore **sizeof(..)**
  - Per usare le funzioni **malloc / free** serve includere la libreria **<stdlib.h>**

© 2000 Prentice Hall, Inc. All rights reserved.

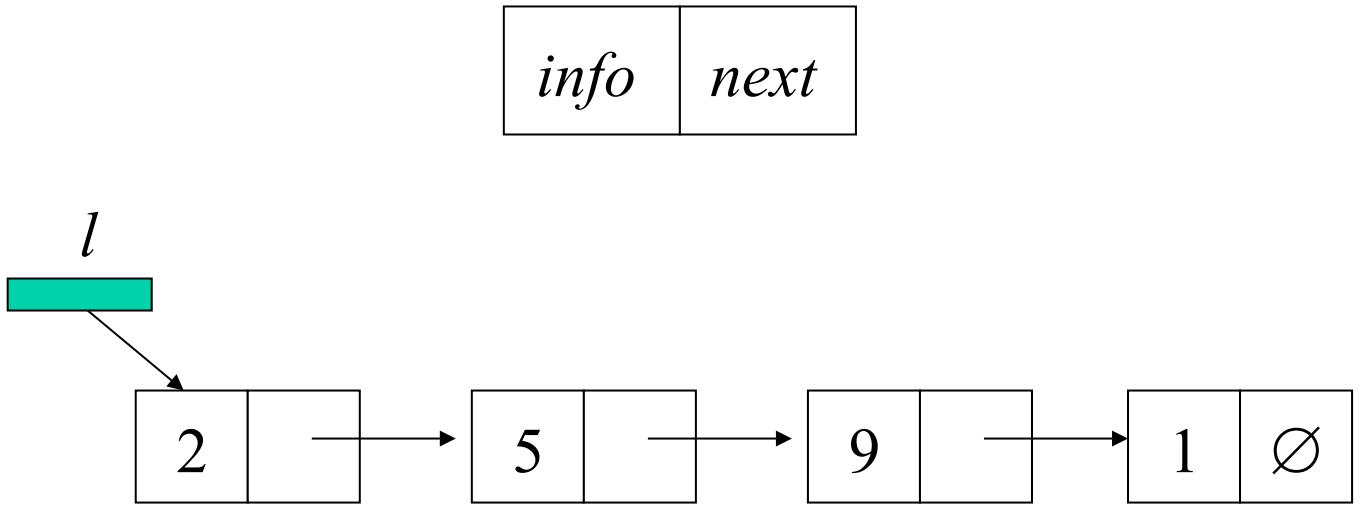
12.2 - Allocazione dinamica della memoria (2/2)

- **`void *malloc(unsigned int size)`**
  - Richiede come parametro il numero di byte di memoria da allocare
    - Essendo questo pari alla dimensione del nuovo dato da creare dinamicamente, di cui si sa a priori il tipo, lo si può ottenere usando **`sizeof(tipo_dato)`**
  - **Restituisce il puntatore alla memoria allocata**
    - Questo puntatore è simile a qualsiasi altro a cui si assegni l'indirizzo di una variabile tramite l'operatore **&**, solo che qui **non se ne conosce il tipo a priori**
    - Quindi deve necessariamente essere di tipo **`void *`**, proprio perché solo così esso può far riferimento (puntare) ad una variabile di qualsiasi tipo
    - Infatti **il puntatore ritornato dalla malloc va assegnato ad un puntatore avente tipo uguale a quello del dato da creare dinamicamente**
    - Dopo l'esecuzione della malloc, tale **puntatore diventa l'unico riferimento nel programma da usare per accedere al nuovo dato**
    - Se non c'è memoria disponibile al momento, restituisce il valore **`NULL`**
    - Esempio: **`struct Nodo *ptr=malloc(sizeof(struct Nodo));`**
- **`void free(void *memPtr)`**
  - Richiede come argomento il puntatore alla zona di memoria allocata
    - Esempio: **`free(ptr);`**

© 2000 Prentice Hall, Inc. All rights reserved.

Le liste
astrazione di vettori
potenzialmente infiniti

Come struttura dati una lista è una sequenza di record,
ciscuno dei quali contiene un campo che punta al successivo:

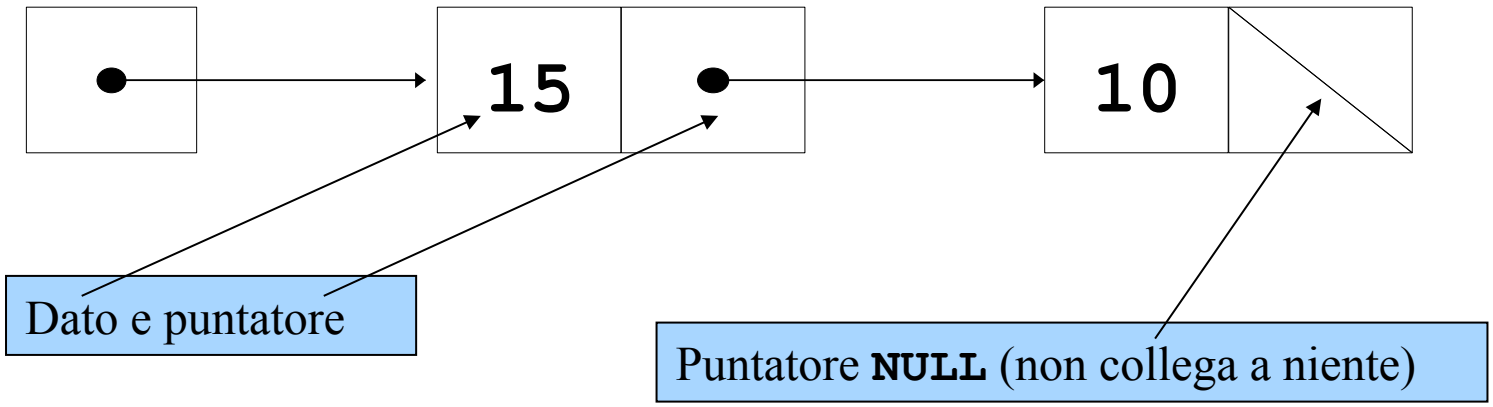


© 2000 Prentice Hall, Inc. All rights reserved.

12.3 - Strutture ricorsive e liste (1/2)

- **Una struttura (struct) ricorsiva**
  - Contiene tra i suoi membri un puntatore che fa riferimento ad una (o più) variabili strutturate dello stesso tipo di quella in cui esso è contenuto
  - Molteplici variabili strutturate ricorsive possono essere collegate in vari modi per formare utili organizzazioni di dati (liste, code, pile ed alberi)
  - Tali sequenze vengono terminate con un puntatore finale **`NULL`** (`0`)
  - Grazie all'allocazione dinamica della memoria e alle strutture ricorsive, è possibile in questo modo definire tipi di dati che crescono all'occorrenza

- Esempio: due strutture ricorsive concatenate



© 2000 Prentice Hall, Inc. All rights reserved.

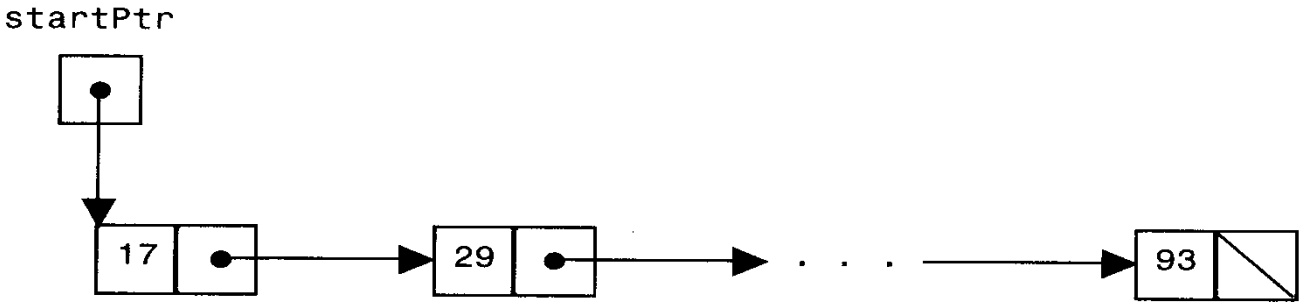
12.3 - Strutture ricorsive e liste (1/2)

- Esempio di struttura ricorsiva

```
struct Nodo{
   int dato;
   struct Nodo *nextPtr;
};
```

  - **nextPtr** punta ad un'altra variabile strutturata di tipo **Nodo**
    - Fa riferimento all'altra variabile strutturata come fosse un LINK
    - Collega una struttura di tipo **Nodo** ad un'altra di pari tipo

- Una sequenza di variabili strutturate ricorsive come la precedente permette di generare una lista concatenata



© 2000 Prentice Hall, Inc. All rights reserved.

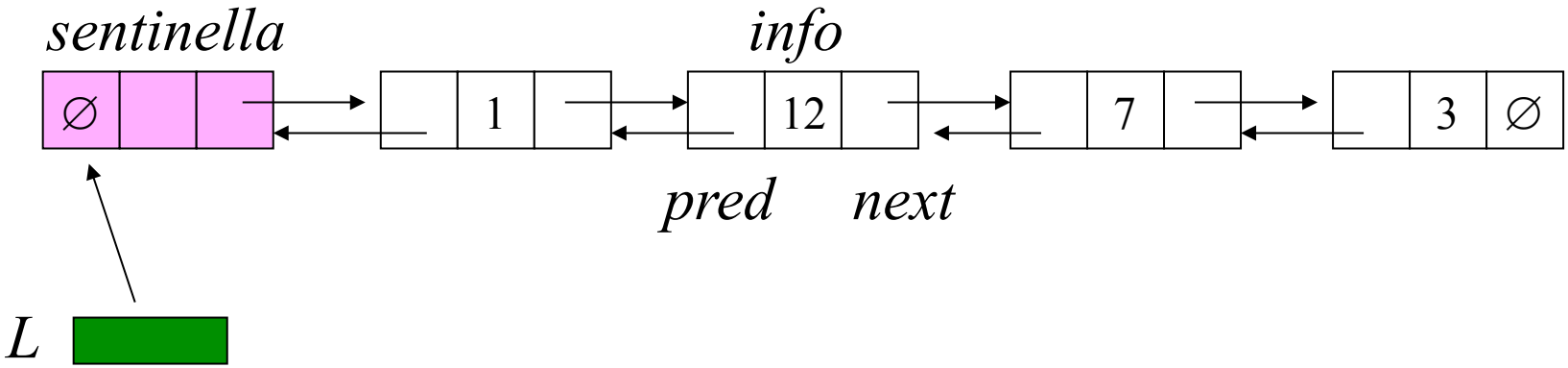
12.4 - Le liste concatenate (1/2)

- Una lista concatenata
  - E' una **collezione lineare di strutture ricorsive**, ognuna delle quali è chiamata ***nodo***, connesse da puntatori, detti ***link***
  - Si accede alla lista tramite un puntatore esterno collegato al suo primo nodo
  - **Si accede ai nodi successivi al primo tramite il puntatore di concatenamento immagazzinato come membro in ogni nodo intermedio**
  - Il puntatore di concatenamento dell'ultimo nodo è impostato a NULL per marcare/evidenziare la fine della lista
  - **Si usa una lista concatenata al posto di un vettore quando**
    - **Il numero di dati/elementi da contenere non è determinabile a priori**
    - **Si vuole inserire/togliere/ordinare gli elementi velocemente**
- Operazioni tipiche sulle liste
  - Inserimento di un nuovo elemento (in testa/fondo o in ordine se ordinate)
  - Ricerca di un elemento e ordinamento della lista
  - Cancellazione di un elemento o della lista
  - Inversione della lista (tramite inversione di tutti i puntatori)

© 2000 Prentice Hall, Inc. All rights reserved.

Liste bidirezionali

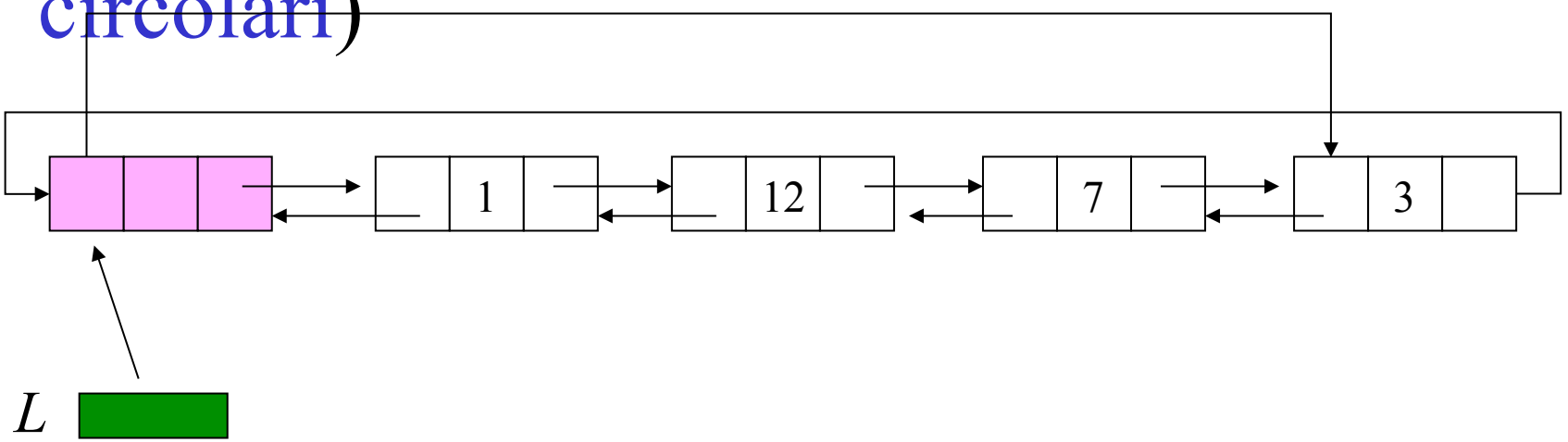
Realizzazione di liste bidirezionali con sentinella (liste doppie)



© 2000 Prentice Hall, Inc. All rights reserved.

Liste bidirezionali

Realizzazione di liste bidirezionali circolari con sentinella (liste doppie circolari)



© 2000 Prentice Hall, Inc. All rights reserved.

12.4 - Le liste concatenate (2/2)

- Tipi di liste concatenate:
  - ***Lista concatenata singolarmente***
    - Vi si accede con un puntatore esterno collegato al primo nodo
    - Termina con un puntatore nullo contenuto nell'ultimo nodo
    - E' attraversata/scandita in un sola direzione (quella data dai puntatori)
  - ***Lista circolare concatenata singolarmente***
    - Come la precedente, ma in aggiunta il puntatore contenuto nell'ultimo nodo si ricollega indietro al primo nodo della lista
  - ***Lista doppiamente concatenata***
    - Vi si può accedere tramite due distinti puntatori esterni, uno collegato al primo nodo e l'altro all'ultimo nodo
    - Ogni nodo ha due puntatori, uno che va avanti verso il nodo successivo ed uno che ritorna indietro al precedente
    - Il puntatore che va avanti dell'ultimo nodo è nullo così come quello che va indietro del primo nodo
    - Può essere attraversata/scandita sia in avanti che indietro (fino ai suoi estremi)
  - ***Lista circolare doppiamente concatenata***
    - Come la precedente, ma il puntatore che va avanti dell'ultimo nodo si collega al primo nodo e il puntatore che va indietro del primo all'ultimo nodo

© 2000 Prentice Hall, Inc. All rights reserved.

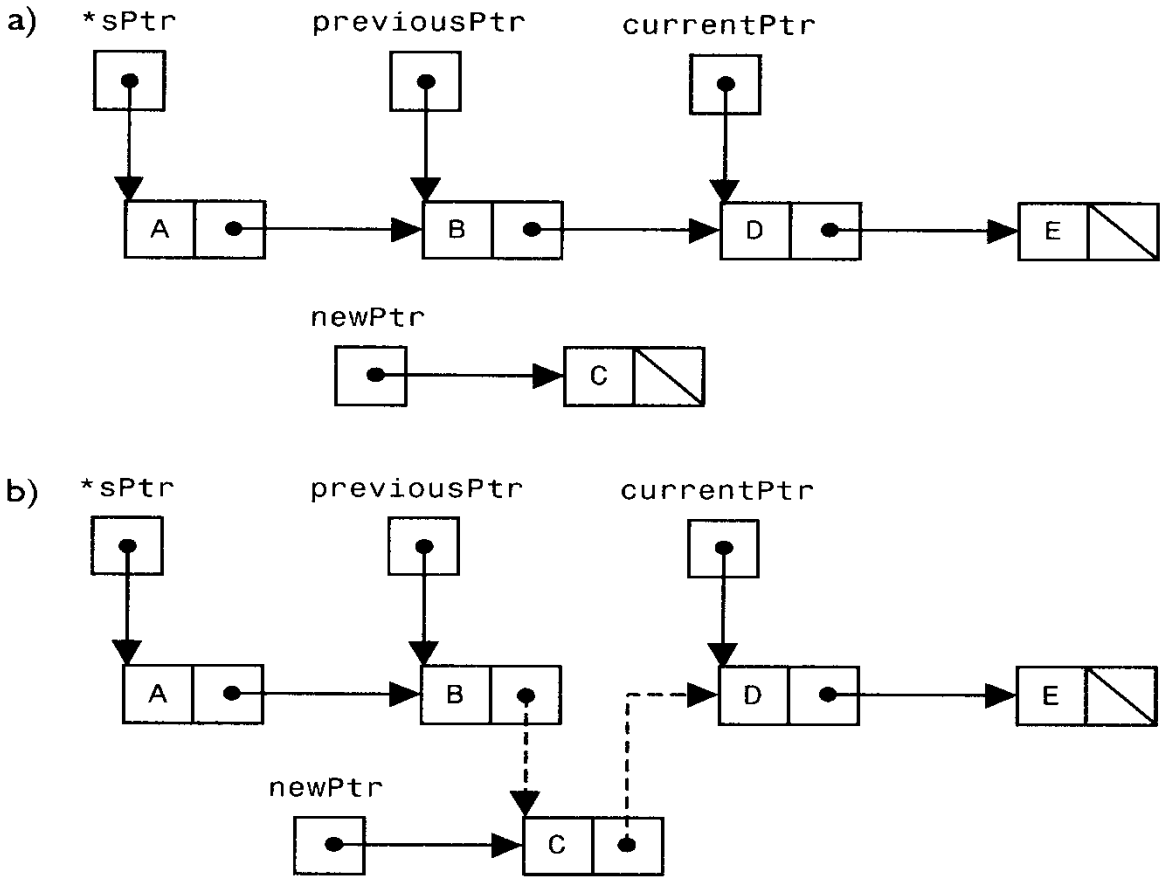
12.5 - Inserimento ordinato di un elemento in una lista ordinata (1/2)

- L'inserimento ordinato di un elemento (nodo) nella lista presuppone che la lista sia appunto ordinata
- Se l'elemento da inserire ha un valore minore del valore del primo elemento della lista allora va inserito in testa
- Se l'elemento da inserire ha un valore maggiore del valore dell'ultimo elemento della lista allora va inserito in coda
- Altrimenti:
  - Si accede alla lista partendo dalla testa
  - Si scorre la lista confrontando il valore dell'elemento da inserire con quello dei nodi intermedi presi in esame volta per volta
  - Per scandire la lista si fa scorrere insieme, parallelamente, un puntatore a "nodo corrente" e uno a "nodo precedente"
  - Quando si trova il primo nodo che ha valore maggiore di quello del dato da inserire, allora si è trovata la posizione in cui inserirlo
  - Va inserito tra il nodi indicati dal puntatore a "nodo precedente" e quello a "nodo corrente" (che punta al dato appena maggiore, l'ultimo scandito)

© 2000 Prentice Hall, Inc. All rights reserved.

12.5 - Inserimento ordinato di un elemento in una lista ordinata (2/2)

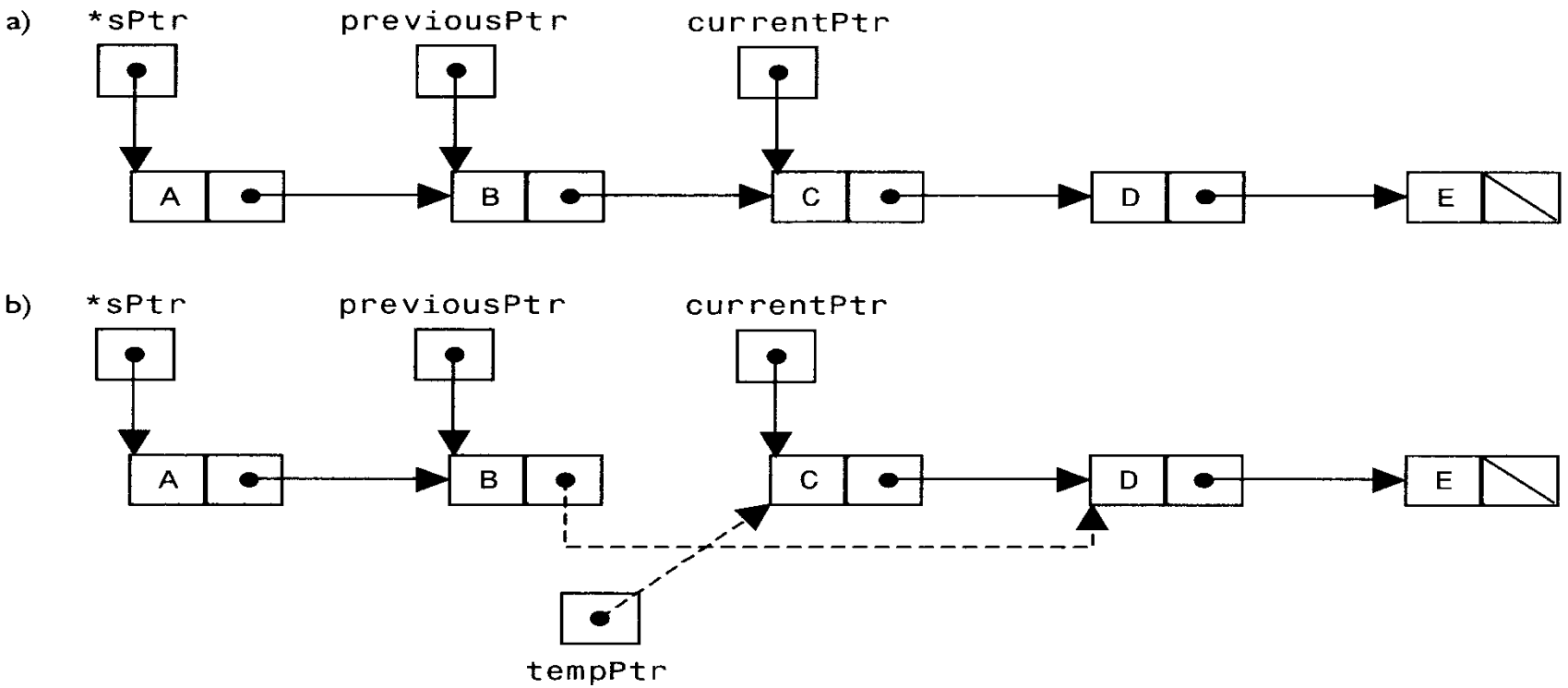
- Inserimento ordinato di un elemento (nodo) nella lista



© 2000 Prentice Hall, Inc. All rights reserved.

12.6 - Eliminazione di un elemento dalla lista

- Anche per questa operazione si usa la coppia di puntatori al “nodo corrente” e al “nodo successivo”
  - Servono a scandire la lista e trovare la posizione del nodo da rimuovere
  - Una volta trovato, permettono di aggiornare i link della lista in modo da sfilare il nodo preservando comunque la struttura ordinata della lista

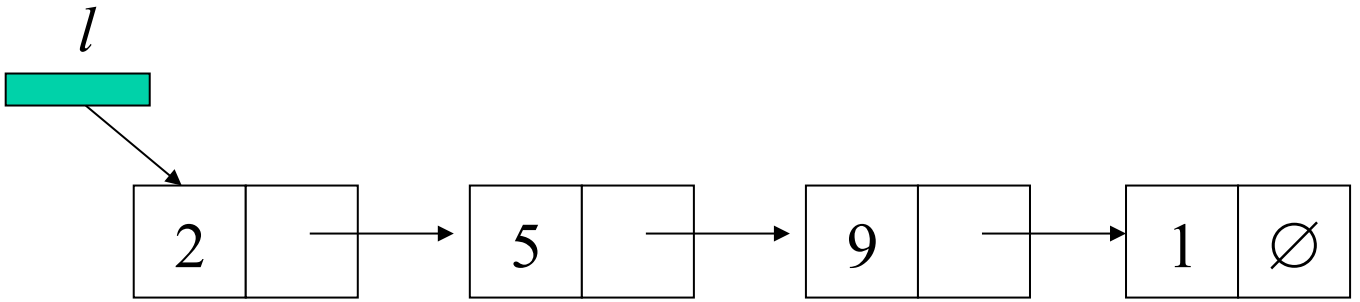


© 2000 Prentice Hall, Inc. All rights reserved.

Le liste

```
typedef struct nodo* Lista;
typedf struct nodo
{    T info;
     Lista next;
} Nodo;
```

Lista è ricorsivo

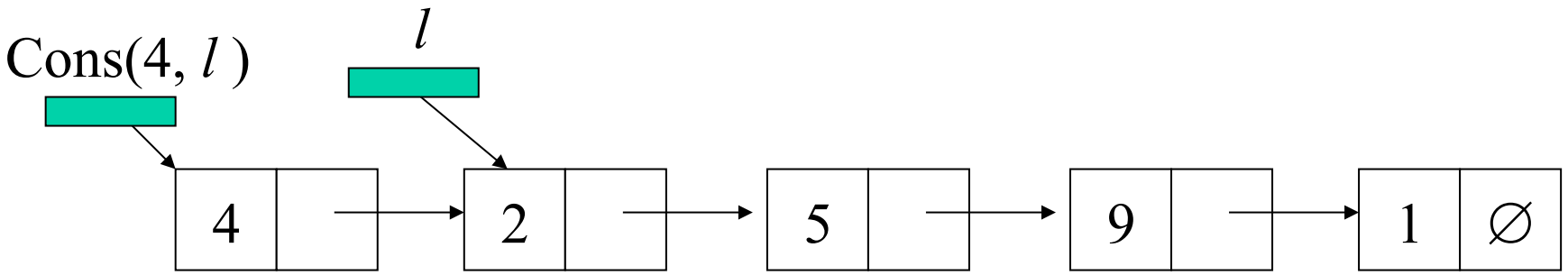


© 2000 Prentice Hall, Inc. All rights reserved.

La funzione Cons

```
Lista Cons (T x, Lista l)
{    Lista nl = new Nodo;
     nl->info = x;
     nl->next = l;
     return nl;
}
```

Cons è un allocatore



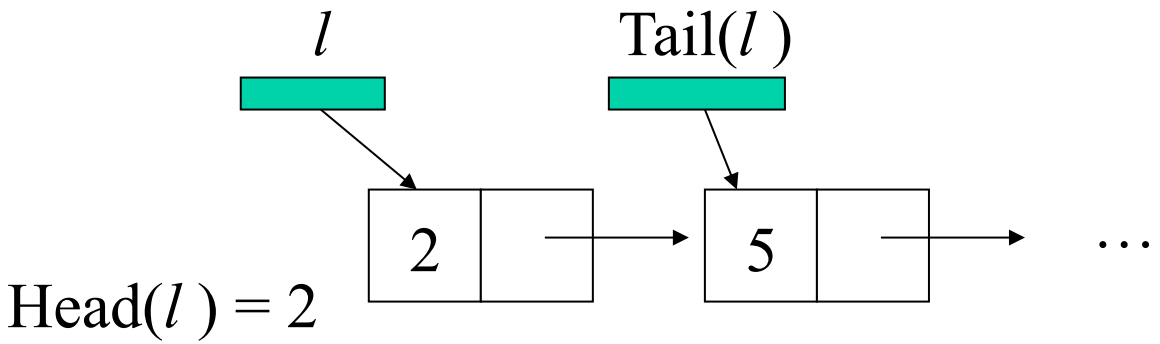
© 2000 Prentice Hall, Inc. All rights reserved.

Le funzioni Head e Tail

```
T Head (Lista l)
// Pre: l non è vuota
{    return l->info;  }
```

Può essere un L-value

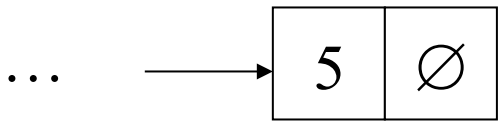
```
Lista& Tail (Lista l)
// Pre: l non è vuota
{    return l->next;  }
```



© 2000 Prentice Hall, Inc. All rights reserved.

La lista vuota

```
bool ListaVuota (Lista l)
{    return l == NULL;  }
```

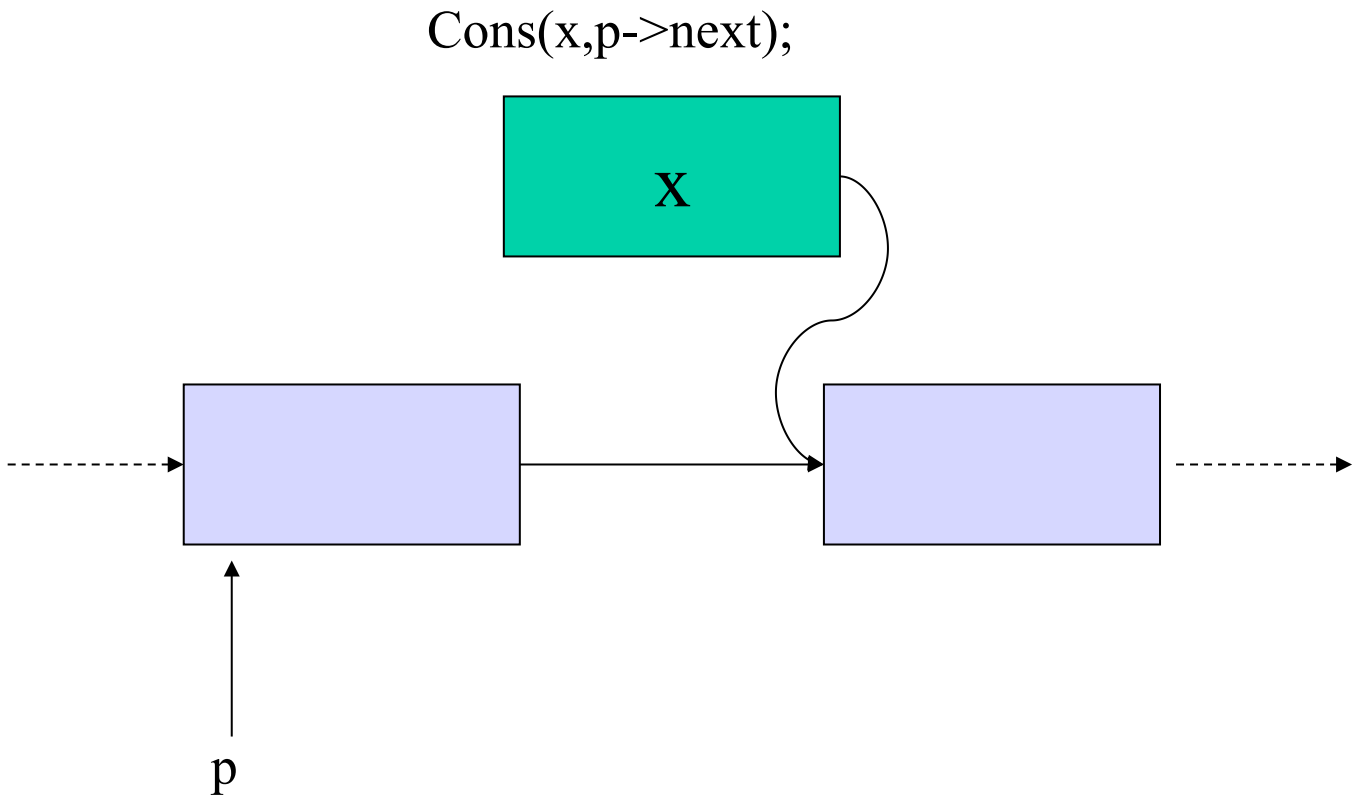


Al fondo di ogni lista c'è la lista vuota, ossia un puntatore a NULL

```
void StampaLista (Lista l)
{    while (!ListaVuota(l))
     {    cout <<  Head(l) << " ";
          l = Tail(l);
     }
}
```

© 2000 Prentice Hall, Inc. All rights reserved.

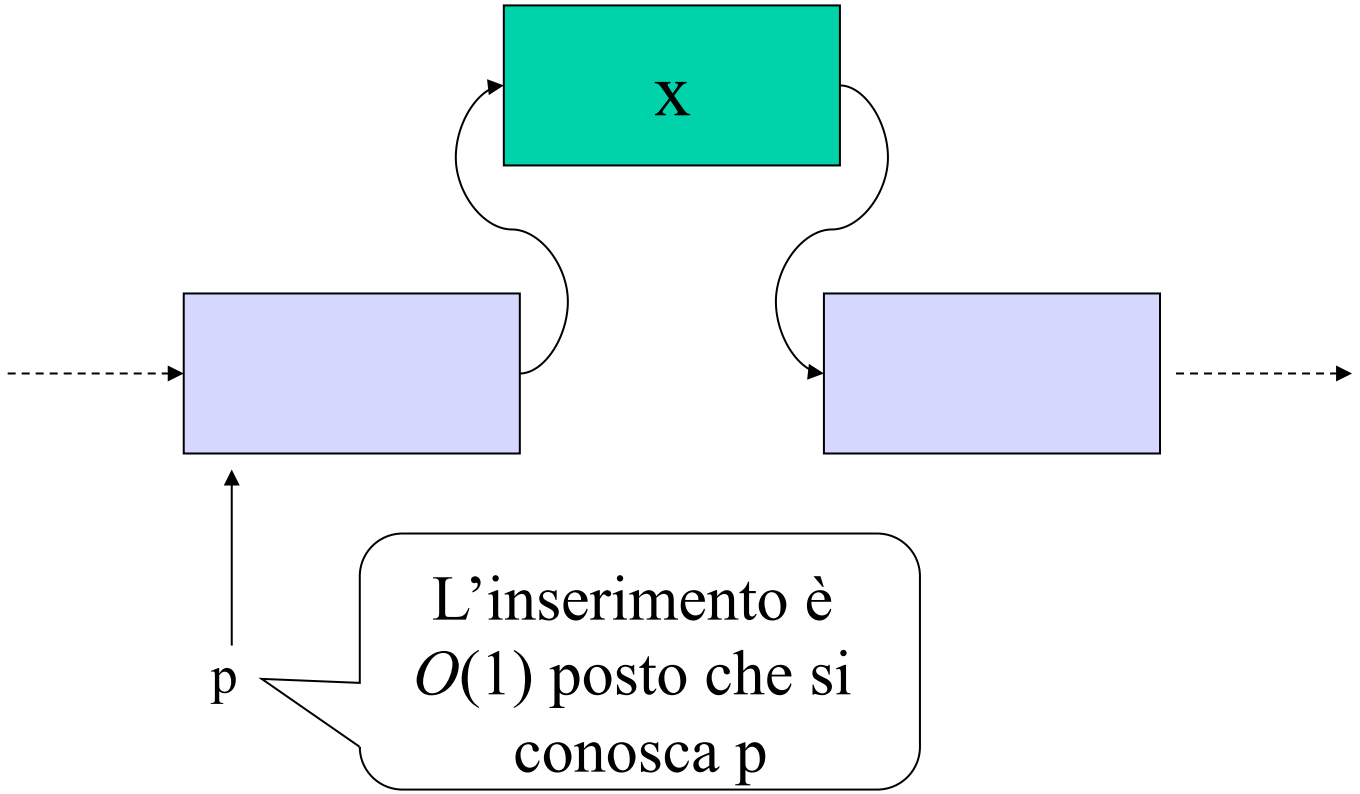
Inserimento in una lista



© 2000 Prentice Hall, Inc. All rights reserved.

Inserimento in una lista

p->next = Cons(x,p->next);



© 2000 Prentice Hall, Inc. All rights reserved.

Inserimento iterativo

```
void Inserimento (T x, int i, Lista& l)
// Pre: 1 <= i <= lunghezza(l) + 1
// Post: inserisce x in l come i-esimo el.
{     if (i == 1) l = Cons(x, l);
      else // 1 < i <= lunghezza(l) + 1
      {   int j = 1; Lista p = l;
          ile (j < i-1 && p != NULL)
                p punta al j-esimo el. di l
              ++;  p = Tail(p);}
              != NULL)
          lora j == i-1,
                  Tail(p) = Cons(x, Tail(p));
      }
}
```

Se p è da cercare allora l'inserimento è $O(n)$

effetto collaterale

© 2000 Prentice Hall, Inc. All rights reserved.

Un metodo ricorsivo

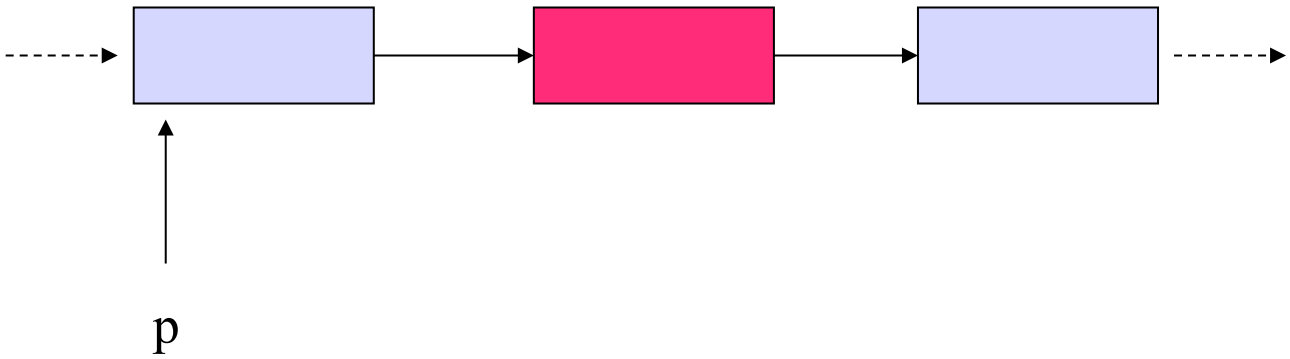
```
Lista InsRic (T x, int i, Lista l)
// Pre: 1 <= i <= lunghezza(l) + 1
// Post: inserisce x in l come i-esimo el.
{  if (i == 1) return Cons(x, l);
   else {  Tail(l) = InsRic (x, i-1, Tail(l));
           return l;
        }
}
```

© 2000 Prentice Hall, Inc. All rights reserved.

Cancellazione da una lista

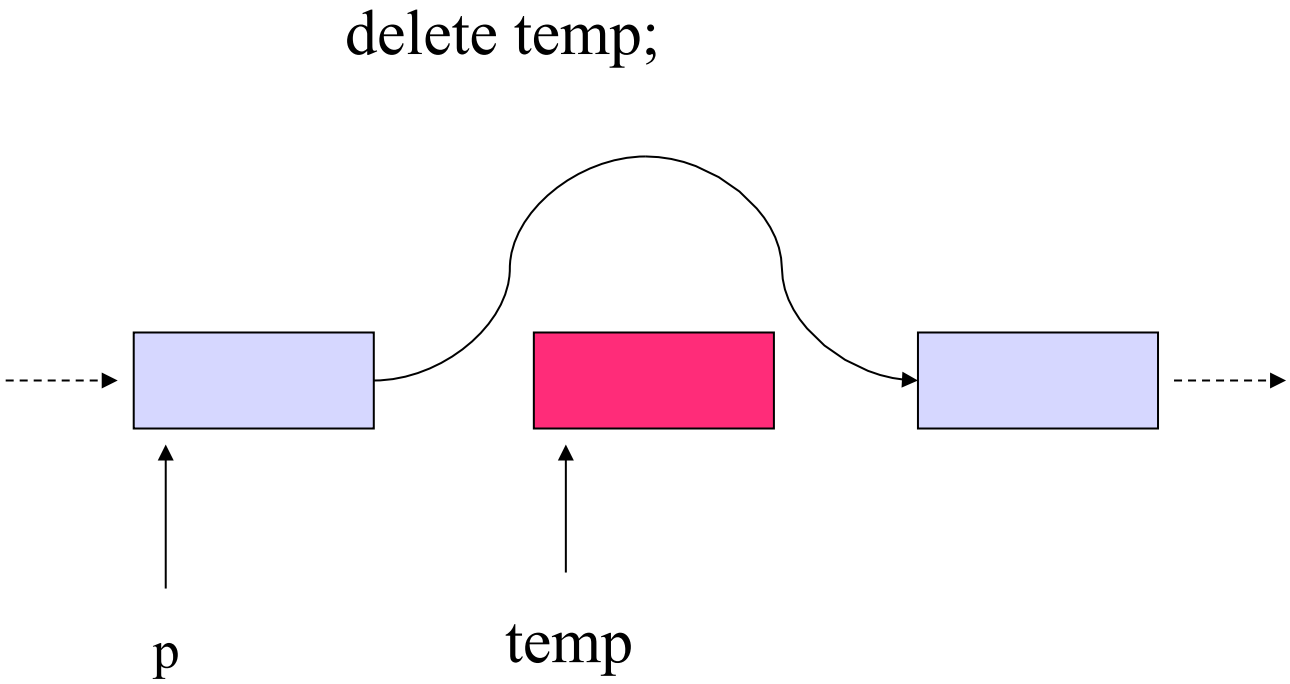
temp = p->next;

p->next = p->next->next



© 2000 Prentice Hall, Inc. All rights reserved.

Cancellazione da una lista



© 2000 Prentice Hall, Inc. All rights reserved.

12.7 - Un esempio completo sulle liste (1/7)

```
1  /* Fig. 12.3: fig12_03.c - Operazioni di gestione di una lista */
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  struct Nodo{    /* struttura ricorsiva */
6    char dato;
7    struct Nodo *proxPtr;
8  };
9
10 typedef struct Nodo NodoLista;
11 typedef NodoLista *NodoListaPtr;
12
13 void inserisciNodo( NodoListaPtr *, char );
14 char cancellaNodo( NodoListaPtr *, char );
15 int listaVuota( NodoListaPtr );
16 void stampaLista( NodoListaPtr );
17 void stampaMenu( void );
18
19 int main(){
20   NodoListaPtr testaPtr = NULL;
21   int scelta;
22   char elemento;
23
```

1. Definisce la struttura che rappresenta ogni nodo della lista

2. Crea un alias per un riferimento diretto al tipo “puntatore a nodo”

3. Prototipi delle funzioni

4. Inizializza le variabili e il puntatore al puntatore (doppio puntatore) al primo nodo della lista, che per ora non punta a nulla

© 2000 Prentice Hall, Inc. All rights reserved.

12.7 - Un esempio completo sulle liste (2/7)

```
24    stampaMenu();  /* visualizza il menù */
25    printf( "? " );
26    scanf( "%d", &scelta );
27    while( scelta != 3 ){
28      switch( scelta ){
29        case 1:
30          printf( "Inserire un carattere: " );
31          scanf( "\n%c", &elemento);
32          inserisciNodo( &testaPtr, elemento );
33          stampaLista( testaPtr );
34          break;
35        case 2:
36          if( !listaVuota( testaPtr ) ){
37            printf("Inserire il carattere da eliminare:");
38            scanf( "\n%c", &elemento );
39            if( cancellaNodo( &testaPtr, elemento ) ){
40              printf("%c è stato eliminato.\n", elemento);
41              stampaLista( testaPtr );
42            }
43            else printf( "%c non esiste.\n\n", elemento );
44          }
45          else printf( "La lista è vuota.\n\n" );
46          break;
```

5. Invoca la funzione che stampa il menu delle operazioni e acquisisce in input la scelta

6. Se l'input è 1, richiede in input il dato (carattere) da inserire, lo inserisce in modo ordinato nella lista e stampa la lista aggiornata
   Alla funzione di inserimento si passa il puntatore al puntatore al primo nodo della lista

7. Se l'input è 2, richiede in input il dato (carattere) da eliminare
   7.1 Se la lista è vuota, non fa la ricerca e lo comunica
   7.2 Se lo trova lo rimuove e stampa la lista aggiornata
   7.3 Se non lo trova, lo comunica

© 2000 Prentice Hall, Inc. All rights reserved.

12.7 - Un esempio completo sulle liste (3/7)

```
47          default:
48             printf( "Scelta non valida.\n\n" );
49             stampaMenu();
50             break;
51       }
52       printf( "? " );
53       scanf( "%d", &scelta );
54    }
55    printf( "Fine del programma.\n" );
56    return 0;
57 }
58
59 /* Visualizza il menu dei comandi */
60 void stampaMenu( void ){
61    printf( "Inserire la scelta:\n"
62            "   1 per inserire un elemento in lista.\n"
63            "   2 per cancellare un elemento dalla lista.\n"
64            "   3 per chiudere il programma.\n" );
65 }
66
67 /* Inserisce ordinatamente un nuovo valore nella lista */
68 void inserisciNodo( NodoListaPtr *tPtr, char valore ){
69    NodoListaPtr nuovoPtr, precPtr, corrPtr;
```

8. Se l'input è diverso dai valori precedenti, comunica che la scelta non è valida e mostra il menu di nuovo

9. Definizione della funzione che stampa il menu delle azioni il dato (carattere) da eliminare

10. Definizione della funzione che inserisce in modo ordinato il dato passato nella lista

© 2000 Prentice Hall, Inc. All rights reserved.

12.7 - Un esempio completo sulle liste (4/7)

```
70
71    nuovoPtr = malloc( sizeof( NodoLista ) );
72    if( nuovoPtr != NULL ){     /* La memoria è disponibile? */
73      nuovoPtr->dato = valore;
74      nuovoPtr->proxPtr = NULL;
75      precPtr = NULL;
76      corrPtr = *tPtr;
77      while( corrPtr != NULL && valore > corrPtr->dato ){
78        precPtr = corrPtr;          /* passa al prossimo nodo */
79        corrPtr = corrPtr->proxPtr;
80      }
81      if( precPtr == NULL ){
82        nuovoPtr->proxPtr = *tPtr;
83        *tPtr = nuovoPtr;
84      }
85      else{
86        precPtr->proxPtr = nuovoPtr;
87        nuovoPtr->proxPtr = corrPtr;
88      }
89    }
90    else printf( "%c non inserito. Manca la memoria.\n", valore);
91  }
92
```

11. Assegna ad un nuovo puntatore l'area di memoria allocata

12. Assegna il nuovo dato al campo dati del nodo allocato e inizialmente NULL al campo che punta al successivo nodo della lista

13. Assegna inizialmente il puntatore al 1° nodo della lista al puntatore al nodo corrente

14. Ciclo che cerca dove inserire il nuovo dato facendo scorrere sia il puntatore all'elemento corrente che quello al nodo precedente

15. Inserisce nel posto opportuno il nuovo nodo, separando il caso in cui vada messo in cima da tutti gli altri casi

© 2000 Prentice Hall, Inc. All rights reserved.

12.7 - Un esempio completo sulle liste (5/7)

```
93  /* Elimina un elemento della lista */
94  char cancellaNodo( NodoListaPtr *tPtr, char valore ){
95    NodoListaPtr precPtr, corrPtr, tempPtr;
96
97    if( valore == ( *tPtr )->dato ){
98      tempPtr = *tPtr;
99      *tPtr = ( *tPtr )->proxPtr;  /* "sfila" il nodo */
100     free( tempPtr );  /*libera la memoria usata dal nodo sfilato*/
101     return valore;
102   }
103   else{
104     precPtr = *tPtr;
105     corrPtr = ( *tPtr )->proxPtr;
106     while( corrPtr != NULL && corrPtr->dato != valore ){
107       precPtr = corrPtr;          /* passa al ...   */
108       corrPtr = corrPtr->proxPtr;  /* ... Prossimo nodo */
109     }
110     if( corrPtr != NULL ){
111       tempPtr = corrPtr;
112       precPtr->proxPtr = corrPtr->proxPtr;
113       free( tempPtr );
114       return valore;
115     }
```

16. Definizione della funzione che elimina il dato passato dalla lista

17. Se il valore che si vuole eliminare sta in cima si salva a parte il puntatore al 1° nodo (per poterne liberare la memoria) e pone il puntatore al 1° nodo pari al quello al secondo nodo

18. Scandisce la lista (fa scorrere la solita coppia di puntatori) e se trova la posizione del nodo da togliere salva a parte il puntatore ad esso; poi pone il puntatore al nodo precedente a quello da eliminare pari al puntatore al nodo successivo di quello corrente

© 2000 Prentice Hall, Inc. All rights reserved.

12.7 - Un esempio completo sulle liste (6/7)

```
116    }
117    return '\0';
118 }
119
120 /* Se la lista è vuota, restituisce 1, altrimenti 0 */
121 int listaVuota( NodoListaPtr tPtr ){
122    return tPtr == NULL;
123 }
124
125 /* Visualizza la lista */
126 void stampaLista( ListNodePtr corrPtr ){
127    if( corrPtr == NULL ) printf( "Lista vuota.\n\n" );
128    else{
129       printf( "La lista è:\n" );
130       while( corrPtr != NULL ){
131          printf( "%c --> ", corrPtr->dato );
132          corrPtr = corrPtr->proxPtr;
133       }
134       printf( "NULL\n\n" );
135    }
136 }
```

19. In ogni caso alla fine libera la memoria

20. Definizione della funzione che verifica se la lista è al momento vuota; per farlo controlla se il puntatore al primo nodo è NULL

21. Definizione della funzione che stampa la sequenza di dati nella lista

22. Finchè il puntatore al nodo successivo a quello attuale non è NULL significa che esiste un altro elemento di cui si deve stampare il dato Per ogni giro del ciclo si assegna al puntatore al nodo attuale quello al nodo successivo

© 2000 Prentice Hall, Inc. All rights reserved.

12.7 - Un esempio completo sulle liste (7/7)

```
Enter your choice: 1 per inserire un elemento in lista.
                   2 per cancellare un elemento dalla lista.
                   3 per chiudere il programma.
? 1
Inserire un carattere: B
La lista è:
B --> NULL
? 1
Inserire un carattere: A
La lista è:
A --> B --> NULL
? 2
Inserire il carattere da eliminare : D
D non esiste.
? 2
Inserire il carattere da eliminare : B
B è stato eliminato.
La lista è:
A --> NULL
Inserire il carattere da eliminare : A
A è stato eliminato.
Lista vuota.
? 2
Lista vuota.
? 4
Scelta non valida.
? 3
Fine del programma.
```

Visualizzazione del programma

© 2000 Prentice Hall, Inc. All rights reserved.