PROGRAMMAZIONE 2: SPERIMENTAZIONI

Lezione 3 – Liste in C

Agenda

- Introduzione
- Prerequisiti
- Strutture autoreferenziali
- Allocazione dinamica della memoria
- Liste collegate
- Operazioni sulle liste
  - creazione di una lista;
  - visualizzazione di una lista;
  - conteggio elementi (nodi) in una lista;
  - ricerca di un elemento (nodo);
  - concatenazione di due liste;
  - inserimento di un elemento (nodo);
  - cancellazione di un elemento (nodo).

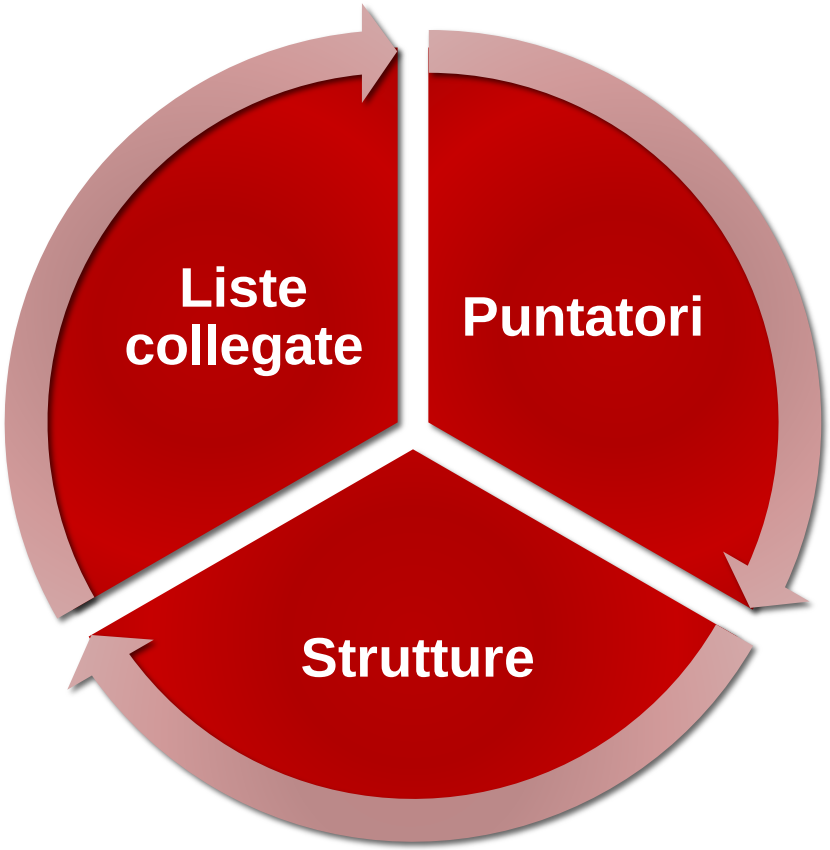
Video lezioni disponibili su YouTube

Introduzione

- Nel corso di Programmazione 1 sono state studiate strutture dati di dimensioni fisse come gli array, le matrici (array bidimensionali) e le strutture.
- Questa lezione introduce le **strutture dinamiche dei dati che possono crescere e ridursi al momento dell'esecuzione**.
- Nello specifico, si studieranno le **liste collegate**.

Prerequisiti

- I puntatori
- Le strutture in C



Strutture autoreferenziali

- Nella lezione sulle **strutture** sono già state nominate le **strutture autoreferenziali** ovvero quelle **strutture che contengono un membro di tipo puntatore che punta a una struttura dello stesso tipo**.

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

Strutture autoreferenziali

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

- Come si può notare, il membro `nextPtr` punta a una struttura dello stesso tipo, da ciò il termine **struttura autoreferenziale**.
- Il membro `nextPtr` è anche detto **link** perché può essere usato per **collegare** una struttura a un'altra; a volte è anche detto campo **next**.
- Questo schema va a formare strutture dati collegate come **liste**, code, pile e alberi.

Strutture autoreferenziali

Se il membro puntatore (link) **non** punta ad alcuna struttura, deve essere impostato a `NULL`.

La mancata impostazione del link a una struttura o a `NULL` può portare a errori in fase di esecuzione.

Allocazione dinamica della memoria

- Creare e mantenere strutture dinamiche di dati che possono crescere e ridursi durante l'esecuzione del programma richiede l'**allocazione dinamica della memoria** ossia **la capacità del programma di ottenere un maggiore spazio di memoria al momento dell'esecuzione**.
- Le funzioni fondamentali del C per la gestione dinamica della memoria sono la **`malloc`**, la **`sizeof`** e la **`free`**.

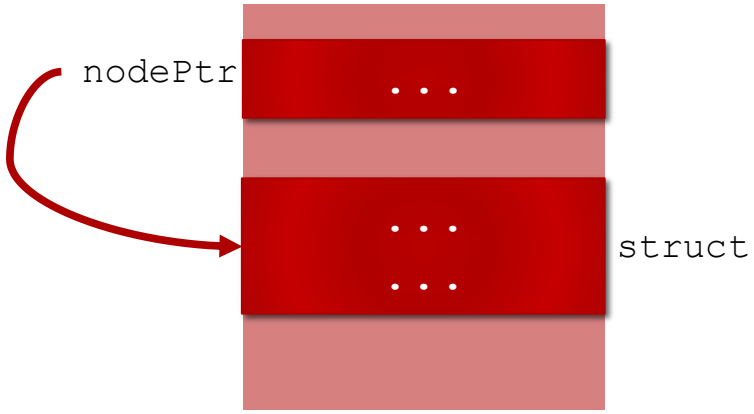
Allocazione dinamica della memoria

- La funzione **`malloc`** (definita in `<stdlib.h>`) riceve come argomento il numero di byte da allocare in memoria e restituisce un puntatore alla memoria allocata (ovvero restituisce l'indirizzo di memoria in cui è stato allocato lo spazio) .
- Il tipo di puntatore restituito dalla **`malloc`** è di tipo `void *` (detto anche *puntatore a void*); un puntatore a void può essere assegnato a una variabile di qualsiasi tipo di puntatore.
- Se non è disponibile spazio in memoria, la **`malloc`** restituisce **`NULL`**.

Allocazione dinamica della memoria

- La **`malloc`** viene solitamente usata con la funzione **`sizeof`** (definita in `<stdlib.h>`).
- La **`sizeof`** serve a determinare la dimensione in byte di un oggetto.

```
struct node *nodePtr;
nodePtr = malloc(sizeof(struct node));
```



Allocazione dinamica della memoria

- La funzione **`free`** (definita in `<stdlib.h>`) riceve come argomento l'indirizzo di memoria da rimuovere quindi libera la memoria restituendola al sistema, in modo che possa essere riallocata in futuro.

```
struct node *nodePtr;
nodePtr = malloc(sizeof(struct node));
free(nodePtr);
```

Allocazione dinamica della memoria

Quando si utilizza la `malloc`, verificare **sempre** se il valore di ritorno è un puntatore a `NULL`. Se il puntatore è nullo, la memoria non è stata allocata.

La dimensione di una struttura in memoria ricavata con la `sizeof` non è necessariamente la somma della dimensione dei suoi membri.

È sempre bene *restituire* la memoria allocata tramite la `free` quando non serve più per evitare che il sistema esaurisca la memoria a disposizione (questo problema è indicato con il termine di *memory leak*).

Far riferimento alla memoria che è stata liberata è un errore che può provocare l'arresto del programma.

Allocazione dinamica della memoria

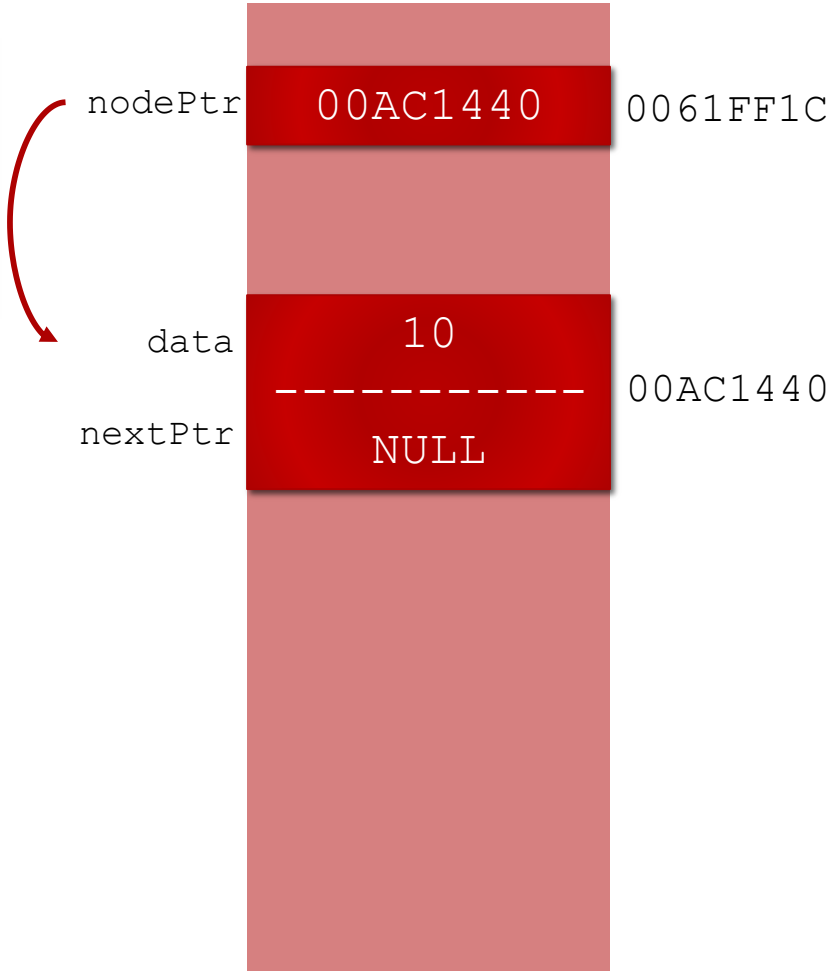
- **Esempio 1**: creare una struttura puntata da un puntatore.

es1.c

```c
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *nextPtr;
};

int main(void)
{
    // nodePtr è un puntatore a struct node
    // ovvero può solo contenere indirizzi di memoria
    struct node *nodePtr;

    // malloc crea uno spazio in memoria
    // la dimensione dello spazio necessario è data da sizeof
    // in nodePtr viene salvato l'indirizzo di memoria creato dalla malloc
    // quindi:
    // nodePtr punta all'indirizzo di memoria creato dinamicamente
    // *nodePtr punta il contenuto dell'indirizzo di memoria
    nodePtr = malloc(sizeof(struct node));

    if (nodePtr!=NULL){
        printf("Indirizzo di nodePtr: %p\n", &nodePtr);
        printf("Memoria puntata da nodePtr: %p\n", nodePtr);
        nodePtr->data = 10; // oppure (*nodePtr).data = 10
        nodePtr->nextPtr = NULL; // oppure (*nodePtr).nextPtr = NULL
        printf("Contenuto puntato da nodePtr: %d\n", nodePtr->data);
    }
    else { // se NULL la malloc non ha creato lo spazio in memoria
        printf("Spazio non allocato!");
    }

     // elimina dalla memoria il contenuto puntato da nodePtr
     // l'indirizzo puntato è così di nuovo utilizzabile
    free(nodePtr);

    return 1;
}
```

Allocazione dinamica della memoria

```
C:\programmazione_2\liste>gcc es1.c

C:\programmazione_2\liste>a.exe
Indirizzo di nodePtr: 0061FF1C
Memoria puntata da nodePtr: 00AC1440
Contenuto puntato da nodePtr: 10
```

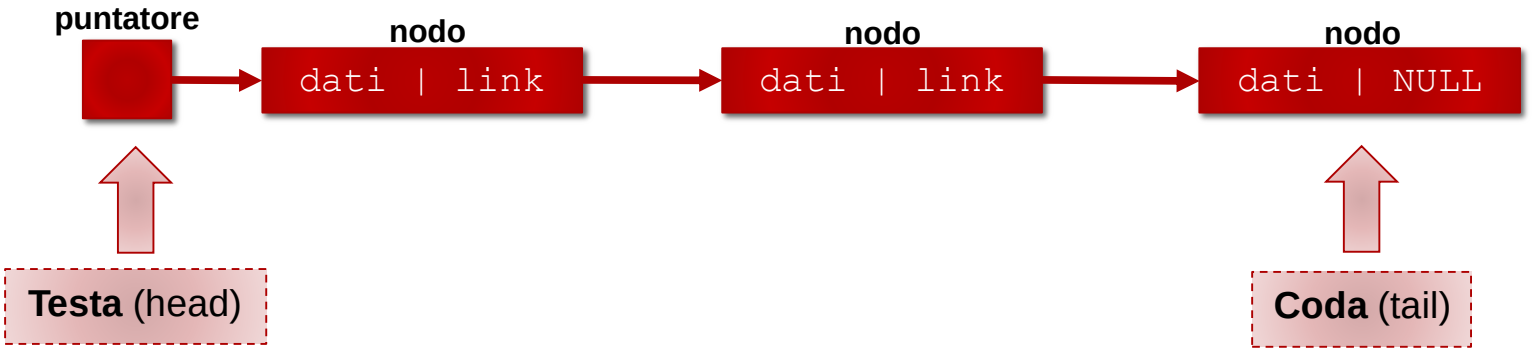


Liste collegate

- Una **lista collegata** è una collezione di strutture autoreferenziali, chiamate **nodi**, connesse da puntatori di collegamento (**link**).
- A una lista collegata si accede mediante un puntatore al **primo nodo** della lista. Il primo nodo è anche detto **testa** (**head**) della lista.
- Ai nodi successivi al primo si accede tramite **il link memorizzato in ogni nodo**.
- Il link dell'ultimo nodo è posto a `NULL`, per indicare la **fine** della lista. L'ultimo nodo è anche detto **coda** (**tail**) della lista.

Liste collegate

- I dati della lista sono memorizzati dinamicamente: **ogni nodo è creato quando necessario** (tramite la `malloc`).
- **Un nodo può contenere dati di ogni tipo** comprese altre `struct`.



Liste collegate

- Le liste collegate **sono utilizzate quando il numero di elementi da rappresentare non è prevedibile**.
  - Esempio: si riceve un file di dati da manipolare e non si conosce a priori il numero di elementi contenuti al suo interno da estrarre e importare nella lista.
- Le liste collegate sono dinamiche, pertanto **la lunghezza di una lista può aumentare o diminuire in fase di esecuzione del programma** in base alla necessità.
- Le liste collegate **si *riempiono* solo quando il sistema non ha più memoria sufficiente per allocare dinamicamente la memoria.**

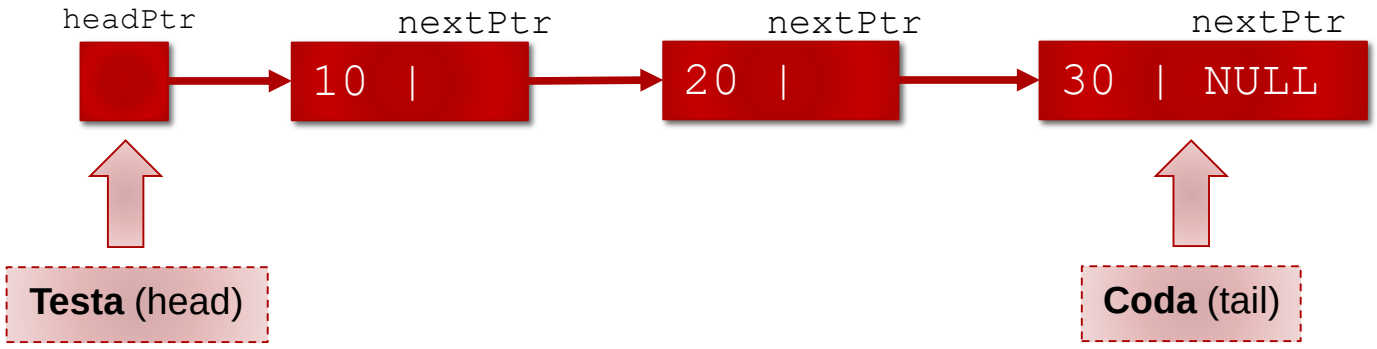
Liste collegate

- Le principali operazioni di base sulle liste collegate sono:

1) creazione di una lista;
2) visualizzazione di una lista;
3) conteggio del numero di elementi;
4) ricerca di un elemento;
5) concatenazione di due liste;
6) inserimento di un elemento (in testa, in coda, nel mezzo);
7) cancellazione di un elemento.

Creazione di una lista

- **Esempio 2**: creare una lista contenente tre nodi i cui membri includeranno i valori 10, 20 e 30.
- Come già indicato in precedenza, si ricorda che:
  - ogni lista che si desidera creare deve avere un puntatore detto **testa** (che fornisce quindi l'indirizzo di memoria del primo elemento della lista);
  - l'ultimo elemento della lista viene detto **coda** e il suo campo `nextPtr` **deve** essere impostato a `NULL`.

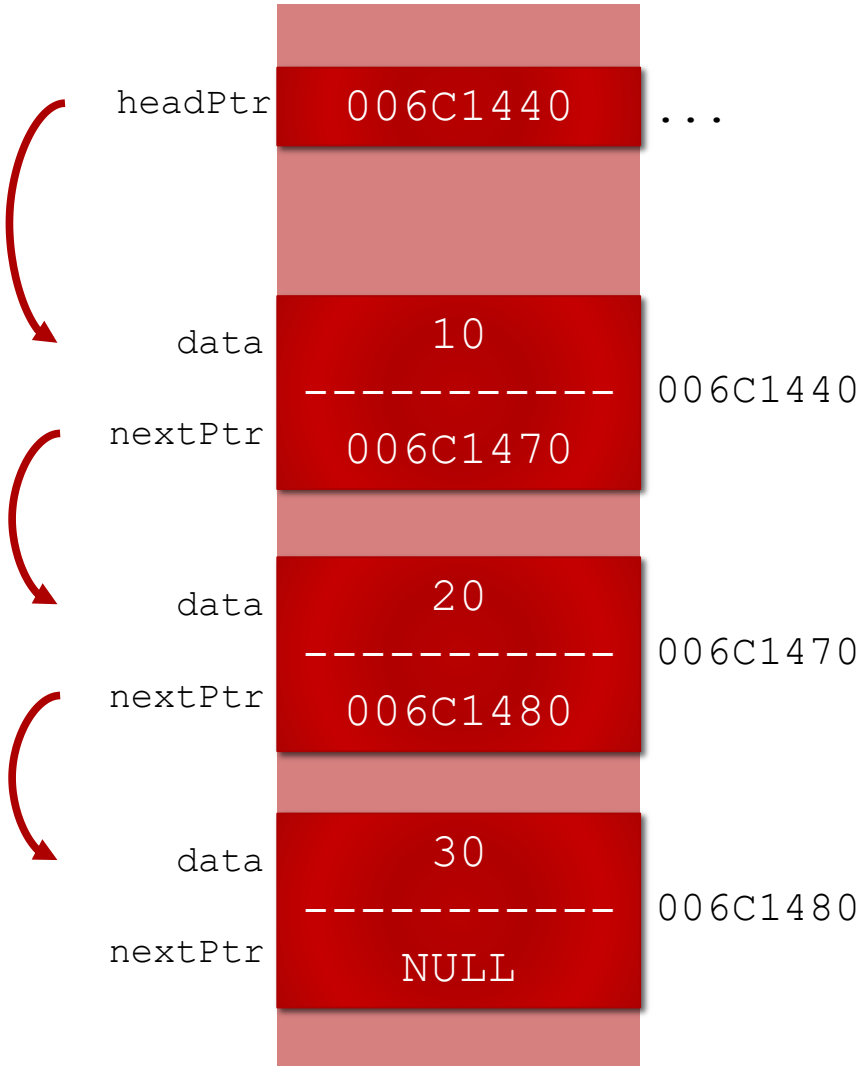


Creazione di una lista

es2.c

```c
int main(void)
{
    // puntatore alla testa della lista
    struct node *headPtr;

    // TESTA/HEAD (PRIMO NODO)
    // la malloc crea lo spazio in memoria per il nodo e restituisce l'indirizzo di memoria dello spazio creato
    // headPtr "punta" l'indirizzo di memoria creato
    headPtr = malloc(sizeof(struct node));

    headPtr->data = 10;         // imposta il dato
    headPtr->nextPtr = NULL;    // inizializza il link

    printf("Nodo 1 (testa) creato all'indirizzo: %p\n", headPtr);

    // NODI SUCCESSIVI AL PRIMO
    // puntatore a un qualsiasi nodo della lista
    struct node *newPtr1;

    newPtr1 = malloc(sizeof(struct node));

    printf("Nodo 2 creato all'indirizzo: %p\n", newPtr1);

    // l'indirizzo di memoria di newPtr1 è inserito nel nextPtr del nodo precedente
    // in modo tale da collegare i due nodi
    headPtr->nextPtr = newPtr1;
    printf("Nodo 1 -> nextPtr: %p\n", headPtr->nextPtr);

    newPtr1->data = 20;         // imposta il dato
    newPtr1->nextPtr = NULL;    // inizializza il link

    struct node *newPtr2;

    newPtr2 = malloc(sizeof(struct node));

    printf("Nodo 3 (coda) creato all'indirizzo: %p\n", newPtr2);

    // l'indirizzo di memoria di newPtr2 è inserito nel nextPtr del nodo precedente
    // in modo tale da collegare i due nodi
    newPtr1->nextPtr = newPtr2;
    printf("Nodo 2 -> nextPtr: %p\n", newPtr1->nextPtr);

    newPtr2->data = 30;         // imposta il dato
    newPtr2->nextPtr = NULL;    // CODA/TAIL (NULL o 0)

    printf("Nodo 3 -> nextPtr: %p\n", newPtr2->nextPtr);

    return 1;
}
```

Creazione di una lista




```
Prompt dei comandi

C:\programmazione_2\liste>a.exe
Nodo 1 (testa) creato all'indirizzo: 006C1440
Nodo 2 creato all'indirizzo: 006C1470
Nodo 1 -> nextPtr: 006C1470
Nodo 3 (coda) creato all'indirizzo: 006C1480
Nodo 2 -> nextPtr: 006C1480
Nodo 3 -> nextPtr: 00000000
```

I nodi delle liste collegate solitamente **non** sono registrati in modo contiguo in memoria. Solamente da un punto di vista **logico** le liste vengono rappresentate come contigue.

Creazione di una lista

La visualizzazione degli indirizzi di memoria utilizzati dal programma è utile solo a fini didattici o di debug.

In fase di rilascio del programma, gli indirizzi di memoria **non** devono essere stampati.

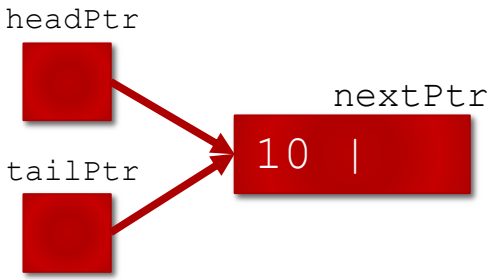
Anche se non presente nel codice, verificare **sempre** che la `malloc` non abbia restituito un valore `NULL` (a indicare l'impossibilità di creare un nuovo nodo in memoria).

Creazione di una lista

- Come si potrà facilmente intuire, è possibile creare liste collegate in maniera **iterativa**, utilizzando i cicli `for`, `while` e `do-while` già studiati nel corso di Programmazione 1.
- A prescindere dal ciclo utilizzato, la creazione di una lista in modo iterativo si può riassumere in **due** fasi:
  1) creazione del primo nodo della lista (testa);
  2) creazione dei nodi successivi al primo sino alla coda.

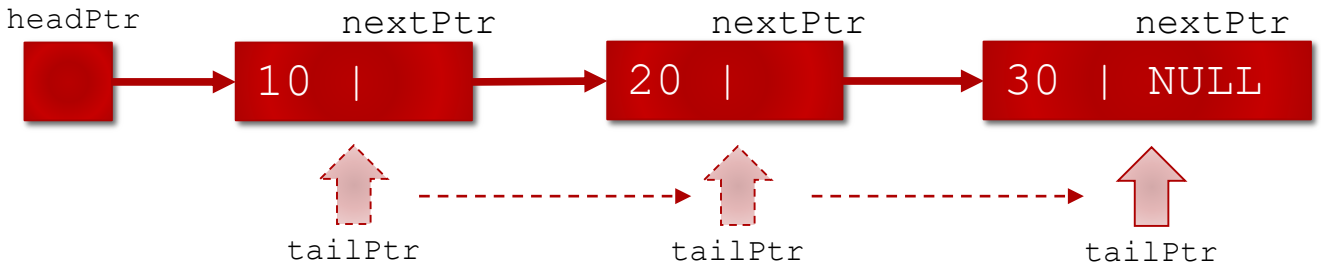
Creazione di una lista

- Creazione del primo nodo della lista (testa):
  1) viene creato un nodo in memoria;
  2) si fa puntare il puntatore di testa a tale nodo;
  3) si fa puntare anche il puntatore di coda al nodo creato (di fatto testa e coda coincidono essendoci un solo elemento nella lista);
  4) viene inserito il dato nel nodo.



Creazione di una lista

- Creazione dei nodi successivi al primo sino alla coda:
  1) viene creato un nuovo nodo;
  2) si fa puntare il campo next del nodo precedente a quello appena creato;
  3) viene spostato il puntatore di coda sul nuovo nodo (in modo tale che diventi il nodo attuale ovvero l'ultimo, la coda);
  4) viene inserito il dato nel nodo;
  5) si torna al punto 1);
  6) al termine del ciclo, il campo next dell'ultimo nodo rimane impostato a `NULL` (quindi è la coda).



Creazione di una lista

es3.c

```c
typedef struct node Lista;

int main(void)
{
    // puntatori ai nodi
    Lista *headPtr;     // testa
    Lista *tailPtr;     // coda (nodi successivi sino all'ultimo)
    Lista *tempPtr;     // nodo temporaneo

    int i;
    int n = 3;

    for (i=0; i<n; i++) // i varrà 0, 1, 2
    {
        if (i==0) // primo nodo (testa)
        {
            headPtr = malloc(sizeof(Lista));
            printf("Nodo testa creato all'indirizzo: %p \n", headPtr);
            // inizialmente, testa e coda puntano lo stesso nodo
            tailPtr = headPtr;
            headPtr->data = (i+1)*10;   // dati
            headPtr->nextPtr = NULL;    // link
        }
        else
        {
            // crea lo spazio e ottiene l'indirizzo di memoria per il nuovo nodo
            tempPtr = malloc(sizeof(Lista));
            printf("Nodo creato all'indirizzo: %p \n", tempPtr);

            // salva l'indirizzo nel campo next del nodo precedente
            // in questo modo il nuovo nodo temp è collegato al nodo precedente
            tailPtr->nextPtr = tempPtr;

            tailPtr = tempPtr;
            // sposta il puntatore al nuovo nodo
            // in questo modo al prossimo passaggio il nodo "corrente" è l'ultimo creato

            tailPtr->data = i*10;
            tailPtr->nextPtr = NULL; // se è l'ultimo nodo il next rimarrà NULL
        }
    }
    return 1;
}
```

Creazione di una lista

- Se si desidera spostare la creazione in una funzione, è possibile ovviamente farlo.
- La funzione di creazione di una lista solitamente **restituisce il puntatore alla testa della lista** in modo tale da utilizzare tale puntatore nelle altre funzioni.

Creazione di una lista

La funzione restituisce il puntatore alla testa della lista.

es4.c

```c
int main(void)
{
    Lista * headPtr;
    headPtr = lista_crea();
    return 1;
}

Lista * lista_crea()
{
    // puntatori ai nodi
    Lista *headPtr;     // testa
    Lista *tailPtr;     // coda (nodi successivi sino all'ultimo)
    Lista *tempPtr;     // nodo temporaneo

    int i;
    int n = 3;

    for (i=0; i<n; i++) // i varrà 0, 1, 2
    {
        if (i==0) // primo nodo (testa)
        {
            headPtr = malloc(sizeof(Lista));
            printf("Nodo testa creato all'indirizzo: %p \n", headPtr);
            // inizialmente, testa e coda puntano lo stesso nodo
            tailPtr = headPtr;
            headPtr->data = (i+1)*10;    // dati
            headPtr->nextPtr = NULL;     // link
        }
        else
        {
            // crea lo spazio e ottiene l'indirizzo di memoria per il nuovo nodo
            tempPtr = malloc(sizeof(Lista));
            printf("Nodo creato all'indirizzo: %p \n", tempPtr);

            // salva l'indirizzo nel campo next del nodo precedente
            // in questo modo il nuovo nodo temp è collegato al nodo precedente
            tailPtr->nextPtr = tempPtr;

            tailPtr = tempPtr;
            // sposta il puntatore al nuovo nodo
            // in questo modo al prossimo passaggio il nodo "corrente" è l'ultimo creato

            tailPtr->data = i*10;
            tailPtr->nextPtr = NULL; // se è l'ultimo nodo il next rimarrà NULL
        }
    }
    return headPtr;
}
```

Visualizzazione di una lista

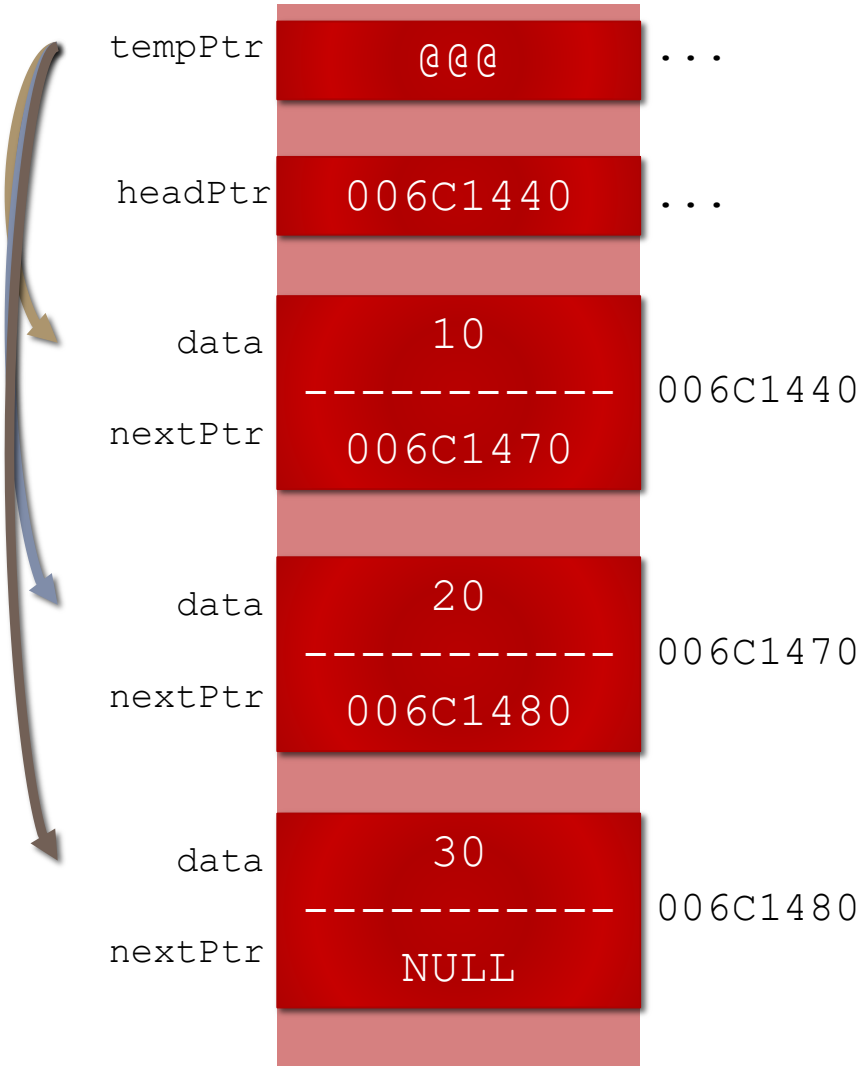
- Se si desidera **visualizzare una lista** procedere come segue:
  1) si crea un puntatore temporaneo;
  2) si fa puntare il puntatore temporaneo alla testa della lista;
  3) si fa spostare il puntatore temporaneo attraverso la lista utilizzando i membri next dei nodi.

Visualizzazione di una lista

es5.c

```
15  Lista * lista_crea();
16  void lista_visualizza(Lista *headPtr);
17
18  int main(void)
19  {
20      Lista *headPtr;
21      headPtr = lista_crea();
22      lista_visualizza(headPtr);
23      return 1;
24  }
25
26  Lista * lista_crea()
27  {
67
68  void lista_visualizza(Lista *headPtr)
69  {
70      // puntatore a una lista temporaneo
71      Lista *tempPtr;
72      // salva la testa nel puntatore temporaneo
73      tempPtr = headPtr;
74
75      // scorre i nodi della lista
76      while(tempPtr!=NULL)
77      {
78          printf("Dato nel nodo: %d\n", tempPtr->data);
79          tempPtr = tempPtr->nextPtr;
80      }
81  }
```

Visualizzazione di una lista



Visualizzazione di una lista:

- inizialmente `tempPtr` contiene lo stesso valore puntato dalla testa della lista (`headPtr`);
- a ogni passaggio `tempPtr` assume il valore del membro `nextPtr` puntato;
- il ciclo termina quando `tempPtr` vale `NULL` perché l'ultimo valore del campo `nextPtr` che ha assunto era nullo (coda).
- Seguendo come esempio lo schema della memoria a fianco, `tempPtr` assumerà i seguenti valori:
  - `006C1440`
  - `006C1470`
  - `006C1480`
  - `NULL`

Visualizzazione di una lista

Per visualizzare una lista è stato utilizzato un puntatore temporaneo in quanto, dopo la creazione di una lista, **il puntatore alla testa della lista non deve mai essere modificato**.

La modifica della testa della lista comporta la perdita del riferimento al primo nodo e/o a quelli successivi.

Conteggio elementi (nodi) in una lista

- Se si desidera **contare i nodi una lista** è sufficiente creare una funzione che ha in input la testa della lista e scorrerla in maniera iterativa in maniera simile alla funzione per la visualizzazione.
- La funzione restituirà 0 se la lista è vuota.
- Si procede quindi come segue:
  1) si crea un puntatore temporaneo;
  2) si fa puntare il puntatore temporaneo alla testa della lista;
  3) si fa spostare il puntatore temporaneo attraverso la lista utilizzando i membri next dei nodi.

Conteggio elementi (nodi) in una lista

es6.c

```
88  int lista_conta_elementi(Lista * headPtr)
89  {
90    // puntatore (temporaneo) a una lista
91    Lista *tempPtr;
92
93    // salva la testa della lista nel puntatore temporaneo
94    tempPtr = headPtr;
95
96    int c = 0;
97
98    // scorre i nodi della lista tramite il puntatore temporaneo
99    while(tempPtr!=NULL)
100   {
101     c++;
102     tempPtr = tempPtr->nextPtr;
103   }
104
105   return c;
106 }
```

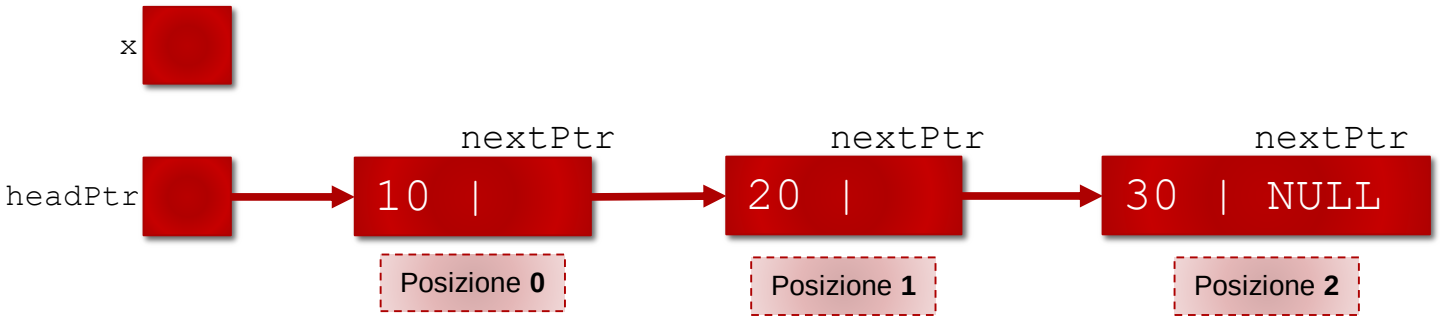
Conteggio elementi (nodi) in una lista

es6.c

```
19  int main(void)
20  {
21    Lista *headPtr;
22    int n;
23    headPtr = lista_crea();
24    lista_visualizza(headPtr);
25    n = lista_conta_elementi(headPtr);
26    printf("La lista contiene %d elementi.\n",n);
27    return 1;
28  }
```

Ricerca di un elemento (nodo)

- L'operazione di **ricerca di un elemento (nodo) all'interno di una lista** si realizza con la scansione della lista stessa, a partire dalla testa e verificando, per ciascun nodo della struttura, se il campo `data` del nodo corrente corrisponde al valore cercato.
- La funzione restituirà la **posizione** del nodo in cui è stata trovata **la prima occorrenza del dato cercato** (**il nodo 0 è la testa**) oppure -1 in caso di dato non trovato.



Ricerca di un elemento (nodo)

es7.c

```
122  int lista_cerca_elemento(Lista * headPtr,int x)
123  {
124    // puntatore (temporaneo) a una lista
125    Lista *tempPtr;
126    // salva la testa della lista nel puntatore temporaneo
127    tempPtr = headPtr;
128    int trovato = -1;
129    int c = -1;
130    // scorre i nodi della lista tramite il puntatore temporaneo
131    while(tempPtr!=NULL){
132      c++;
133      // dato trovato
134      if (tempPtr->data == x){
135        trovato = c;
136        break; // interrompe il ciclo
137      }
138      tempPtr = tempPtr->nextPtr;
139    }
140    return trovato;
141  }
```

Ricerca di un elemento (nodo)

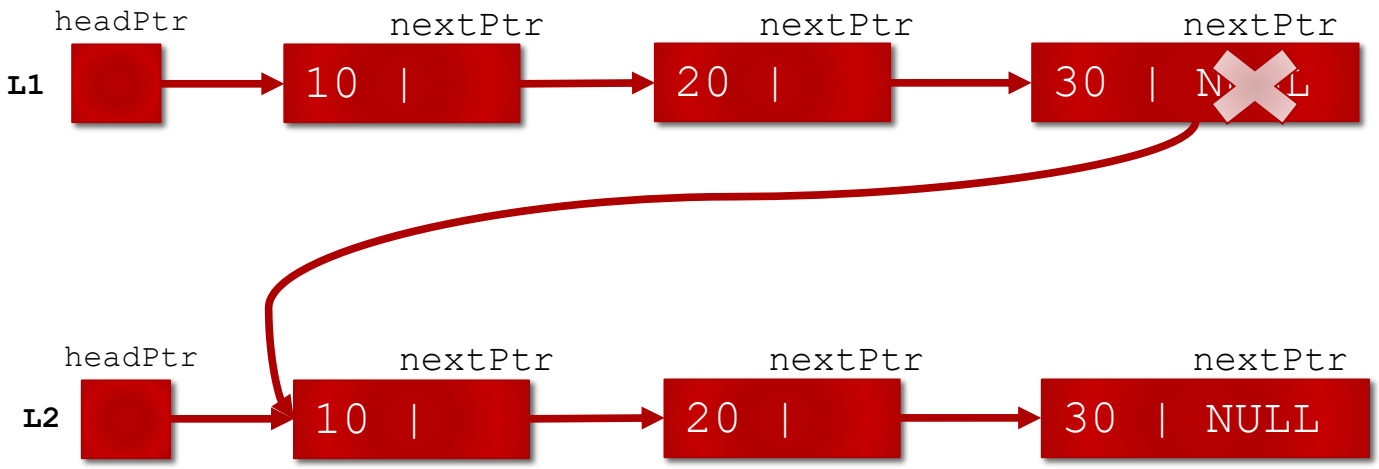
es7.c

```
20  int main(void)
21  {
22    Lista *headPtr;
23    int n;
24    int x; // dato da cercare
25    int trovato; // posizione del nodo
26    headPtr = lista_crea();
27    lista_visualizza(headPtr);
28    n = lista_conta_elementi(headPtr);
29    printf("La lista contiene %d elementi.\n",n);
30    printf("Inserisci un dato da cercare: ");
31    scanf("%d",&x);
32    trovato = lista_cerca_elemento(headPtr,x);
33    if (trovato < 0)
34    {
35      printf("Elemento non trovato\n");
36    }
37    else
38    {
39      printf("Elemento trovato nel nodo %d\n",trovato);
40    }
41    return 1;
42  }
```

Concatenazione di due liste

- Spesso può tornare utile ottenere un'unica lista a partire da due liste distinte.
- La concatenazione di due liste `l1` e `l2` (nell'ipotesi che `l1` non sia vuota) determina l'ottenimento di una lista unica collegando la lista `l2` alla fine della lista `l1`.
- La funzione di concatenazione attraversa la lista `l1` sino alla coda quindi modifica il membro next dell'ultimo nodo rimuovendo `NULL` e inserendo l'indirizzo della testa (il primo nodo) della seconda lista `l2`.

Concatenazione di due liste



Il membro next dell'ultimo nodo di L1 verrà fatto puntare all'indirizzo del primo nodo di L2.

Concatenazione di due liste

es8.c

```
95  void lista_concatena(Lista *l1, Lista *l2)
96  {
97      // puntatore (temporaneo) a una lista
98      Lista *tempPtr;
99      // salva la testa della lista di l1 nel puntatore temporaneo
100     tempPtr = l1;
101
102     while(tempPtr!=NULL){
103         // verifica se è l'ultimo nodo della lista
104         if (tempPtr->nextPtr == NULL){
105             tempPtr->nextPtr = l2;
106             break;
107         }
108         tempPtr = tempPtr->nextPtr;
109     }
110 }
```

Concatenazione di due liste

es8.c

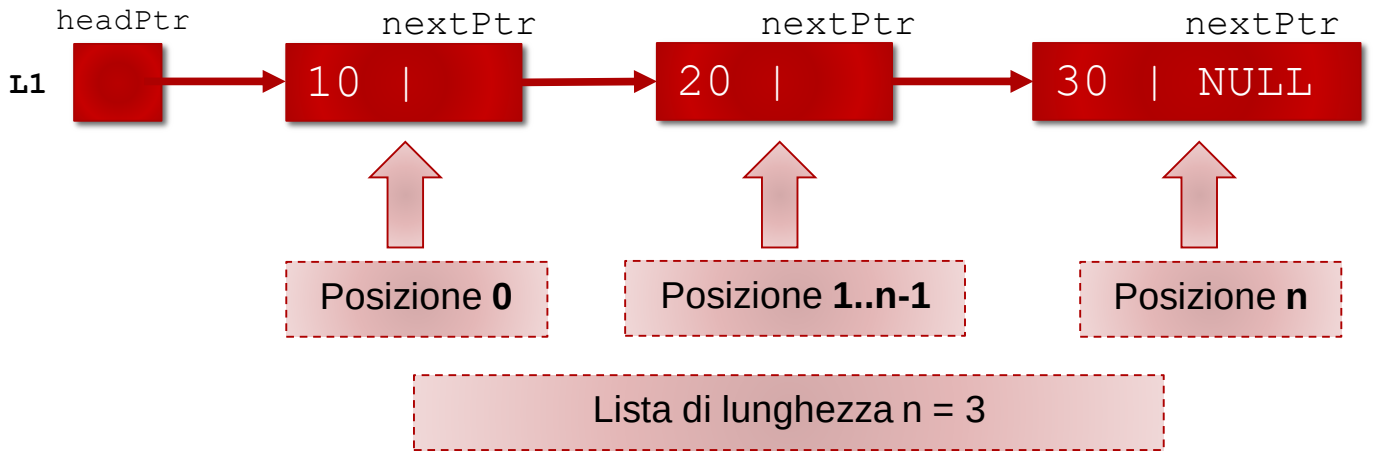
```
19  int main(void)
20  {
21      // l1 è il puntatore alla testa della prima lista
22      // l2 è il puntatore alla testa della seconda lista
23      Lista *l1, *l2;
24      l1 = lista_crea();
25      printf("Lista 1: \n");
26      lista_visualizza(l1);
27      l2 = lista_crea();
28      printf("Lista 2: \n");
29      lista_visualizza(l2);
30      // aggiunge i dati di l2 in l1
31      lista_concatena(l1,l2);
32      printf("Lista 1 (concatenata a Lista 2)\n");
33      lista_visualizza(l1);
34      return 1;
35  }
```

Inserimento di un elemento (nodo)

- Una delle proprietà più utili delle liste è rappresentata dal fatto che l'inserimento di un nodo richiede una quantità di tempo costante, una volta determinata la posizione in cui inserirlo.
- L'idea alla base è la seguente: individuata la posizione in cui aggiungere il nodo, si modificano i puntatori per "inserirlo" nella posizione corretta.

Inserimento di un elemento (nodo)

- È possibile individuare **tre** casi d'inserimento del nuovo nodo:
  1) posizione 0 → inserimento in testa alla lista;
  2) posizione n → inserimento in coda alla lista;
  3) posizione 1 .. n-1 → inserimento al *centro* della lista.



Inserimento di un elemento (nodo)

es9.c

```
127  Lista * lista_inserisci(Lista *headPtr, int x, int p)
128  {
129    // crea un nuovo nodo e lo fa puntare da newPtr
130    Lista *newPtr;
131    newPtr = malloc(sizeof(Lista));
132
133    // puntatore temporaneo alla testa
134    Lista *tempPtr;
135    tempPtr = headPtr;
136
137    Lista *prevPtr = NULL;
138
139    // contatore dei nodi attraversati (posizione)
140    int c = 0;
141
142    // se c'è spazio in memoria
143    if (newPtr!=NULL) {
144      newPtr->data = x;
145      newPtr->nextPtr =  NULL;
```

Inserimento di un elemento (nodo)

es9.c

```
146
147       // finché non arriva alla fine (NULL) o alla posizione desiderata
148       while(tempPtr!=NULL & c!=p){
149         c++;
150         prevPtr = tempPtr;
151         tempPtr = tempPtr->nextPtr;
152       }
153
154       // inserimento in testa
155       // il nodo "precedente" non è stato modificato
156       // quindi la posizione in cui inserire è in testa
157       if (prevPtr==NULL){
158         // il membro next del nuovo nodo diventa la vecchia testa
159         newPtr->nextPtr = headPtr;
160         // la testa diventa il nuovo nodo
161         headPtr = newPtr;
162       }
163       else{
164         prevPtr->nextPtr = newPtr;
165         newPtr->nextPtr = tempPtr;
166       }
167     }
168     // non ha potuto creare il nuovo nodo
169     else {
170       printf("Spazio in memoria esaurito, nuovo nodo non inserito.\n");
171     }
172     return headPtr;
173   }
```

Inserimento di un elemento (nodo)

es9.c

```c
int main(void)
{
  // l1 è il puntatore alla testa della prima lista
  Lista *l1;
  int x; // dato da inserire nel nuovo nodo
  int n; // dimensione della lista
  int p; // posizione in cui inserire il nuovo nodo
  l1 = lista_crea();
  printf("Lista: \n");
  lista_visualizza(l1);
  n = lista_conta_elementi(l1);
  do {
    printf("Indicare la posizione in cui inserire il nodo (0...%d): ", n);
    scanf("%d",&p);
  } while(p<0 || p>n);
  printf("Inserire il dato del nodo: ");
  scanf("%d",&x);
  l1 = lista_inserisci(l1, x, p);
  printf("Lista aggiornata: \n");
  lista_visualizza(l1);
  return 1;
}
```

Inserimento di un elemento (nodo)

```
Lista 1:
Dato nel nodo 0: 10
Dato nel nodo 1: 20
Dato nel nodo 2: 30
Dato nel nodo 3: 40
Indicare la posizione in cui inserire il nodo (0...4): 0
Inserire il dato del nodo: -1
Lista 1 aggiornata
Dato nel nodo 0: -1
Dato nel nodo 1: 10
Dato nel nodo 2: 20
Dato nel nodo 3: 30
Dato nel nodo 4: 40
```

Posizione **0** (inserimento in testa)

Posizione **n** (inserimento in coda)

```
Lista 1:
Dato nel nodo 0: 10
Dato nel nodo 1: 20
Dato nel nodo 2: 30
Dato nel nodo 3: 40
Indicare la posizione in cui inserire il nodo (0...4): 4
Inserire il dato del nodo: -1
Lista 1 aggiornata
Dato nel nodo 0: 10
Dato nel nodo 1: 20
Dato nel nodo 2: 30
Dato nel nodo 3: 40
Dato nel nodo 4: -1
```

Inserimento di un elemento (nodo)

```
Lista 1:
Dato nel nodo 0: 10
Dato nel nodo 1: 20
Dato nel nodo 2: 30
Dato nel nodo 3: 40
Indicare la posizione in cui inserire il nodo (0...4): 1
Inserire il dato del nodo: -1
Lista 1 aggiornata
Dato nel nodo 0: 10
Dato nel nodo 1: -1
Dato nel nodo 2: 20
Dato nel nodo 3: 30
Dato nel nodo 4: 40
```

```
Lista 1:
Dato nel nodo 0: 10
Dato nel nodo 1: 20
Dato nel nodo 2: 30
Dato nel nodo 3: 40
Indicare la posizione in cui inserire il nodo (0...4): 2
Inserire il dato del nodo: -1
Lista 1 aggiornata
Dato nel nodo 0: 10
Dato nel nodo 1: 20
Dato nel nodo 2: -1
Dato nel nodo 3: 30
Dato nel nodo 4: 40
```

Posizioni **1..n-1** (inserimento in *centro*)

```
Lista 1:
Dato nel nodo 0: 10
Dato nel nodo 1: 20
Dato nel nodo 2: 30
Dato nel nodo 3: 40
Indicare la posizione in cui inserire il nodo (0...4): 3
Inserire il dato del nodo: -1
Lista 1 aggiornata
Dato nel nodo 0: 10
Dato nel nodo 1: 20
Dato nel nodo 2: 30
Dato nel nodo 3: -1
Dato nel nodo 4: 40
```

Inserimento di un elemento (nodo)

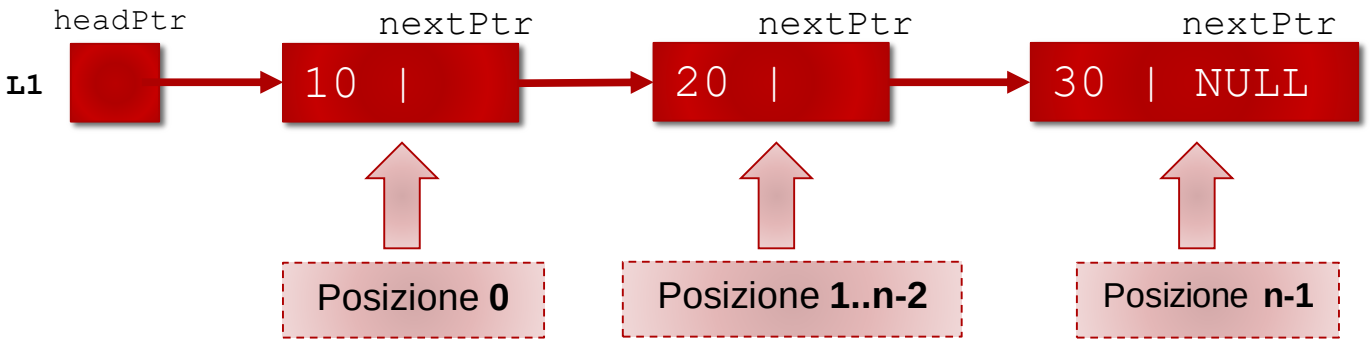
Quando si utilizza la `malloc` per creare il nuovo nodo, verificare sempre se il valore di ritorno è un puntatore a `NULL`. Se il puntatore è nullo, la memoria non è stata allocata.

Cancellazione di un elemento (nodo)

- Come per l'inserimento di un nodo, un'altra proprietà delle liste è rappresentata dal fatto che la **cancellazione di un nodo** richiede una quantità di tempo costante, una volta determinata la posizione del nodo da cancellare.
- L’idea alla base è la seguente: individuata la posizione in cui cancellare il nodo, si modificano i puntatori per "saltarlo".

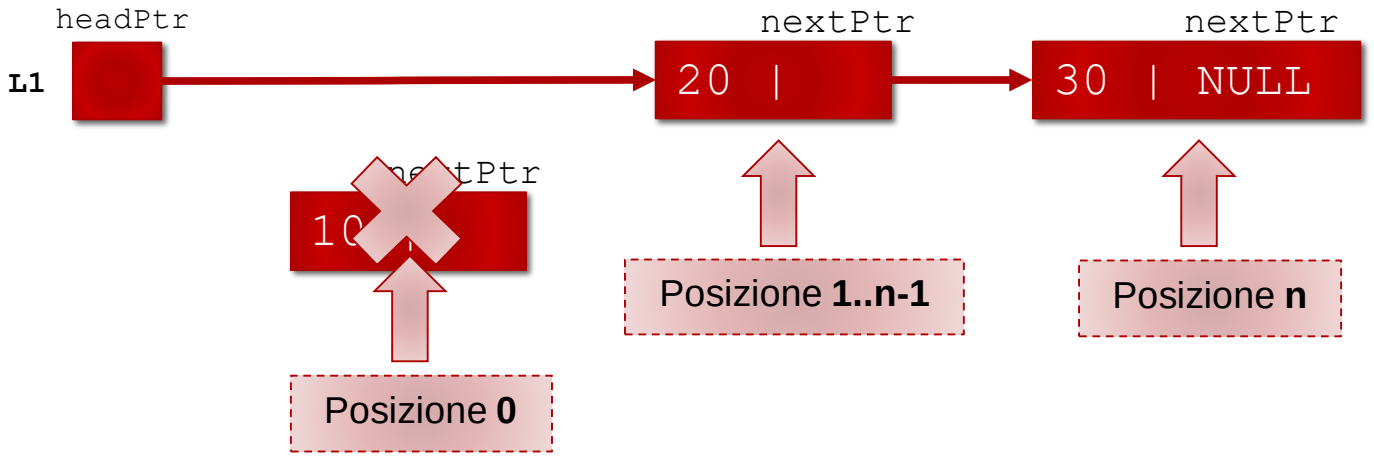
Cancellazione di un elemento (nodo)

- È possibile individuare **due** casi di cancellazione del nodo:
  1) posizione 0 → cancellazione della testa della lista;
  2) posizione 1 .. n-1 → cancellazione di un nodo al *centro o in coda* della lista.



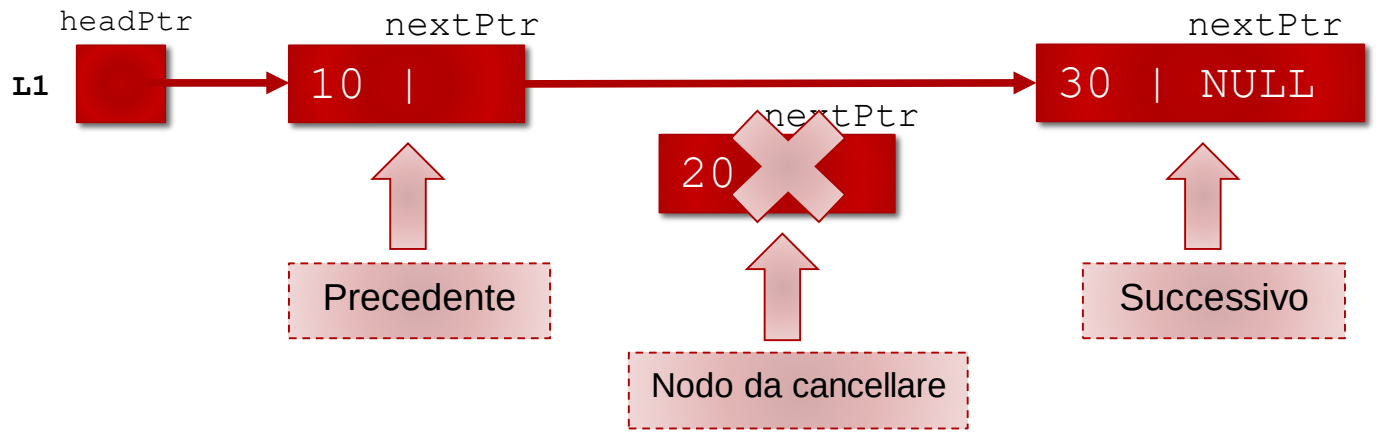
Cancellazione di un elemento (nodo)

- Nel caso si debba cancellare il nodo iniziale (*testa*), procedere come segue:
  - creare un secondo puntatore alla testa della lista;
  - impostare come nuova testa il secondo elemento della lista (ovvero la nuova testa è il membro next della testa attuale);
  - cancellare il nodo puntato tramite la funzione `free`.



Cancellazione di un elemento (nodo)

- Nel caso si debbano cancellare i nodi centrali o la coda, procedere come segue:
  - visitare la lista cercando la posizione **precedente** al nodo cercato;
  - utilizzare un puntatore temporaneo una volta individuato il nodo da cancellare;
  - impostare come campo next nel nodo precedente a quello da cancellare, quello subito successivo al nodo da cancellare.



Cancellazione di un elemento (nodo)

es10.c

```
121 Lista * lista_cancella(Lista *headPtr, int p)
122 {
123     // puntatore temporaneo alla testa
124     Lista *tempPtr;
125     tempPtr = headPtr;
126
127     // contatore dei nodi attraversati
128     int c = 0;
129
130     // cancellazione in testa
131     // la nuova testa è il membro next dell'attuale testa
132     if (p==0)
133     {
134         headPtr = tempPtr->nextPtr;
135         free(tempPtr);
136         return headPtr;
137     }
138
139     // si posizione al nodo PRECEDENTE (p-1) a quello da cancellare
140     while(tempPtr!=NULL && c!=p-1){
141             c++;
142             tempPtr = tempPtr->nextPtr;
143         }
144
145     // tempPtr contiene il nodo PRECEDENTE a quello da cancellare
146     // quindi tempPtr->nextPtr è il nodo da cancellare
147     // quindi collega a tempPtr il nodo presente in tempPtr->next->next
148     Lista *newPtr = tempPtr->nextPtr->nextPtr;
149
150     // cancella il nodo successivo al precedente
151     free(tempPtr->nextPtr);
152
153     tempPtr->nextPtr = newPtr;
154
155     return headPtr;
156 }
```

Cancellazione di un elemento (nodo)

es10.c

```
20  int main(void)
21  {
22      // l1 è il puntatore alla testa della prima lista
23      Lista *l1;
24      int x; // dato da inserire nel nuovo nodo
25      int n; // dimensione della lista
26      int p; // posizione in cui inserire il nuovo nodo
27      l1 = lista_crea();
28      printf("Lista: \n");
29      lista_visualizza(l1);
30      n = lista_conta_elementi(l1);
31      do {
32          printf("Indicare la posizione in cui cancellare il nodo (0...%d): ", n-1);
33          scanf("%d",&p);
34      } while(p<0 || p>n-1);
35      l1 = lista_cancella(l1, p);
36      printf("Lista aggiornata: \n");
37      lista_visualizza(l1);
38      return 1;
39  }
```

Cancellazione di un elemento (nodo)

Utilizzare **sempre** la funzione `free` per liberare la memoria non più utilizzata, onde evitare che il sistema esaurisca prematuramente la memoria (*memory leak*).

Porre sempre attenzione alla **testa** della lista: la cancellazione del nodo iniziale della lista senza un preventivo salvataggio del nodo successivo (la nuova testa), comporta la perdita completa dei dati della lista.

FINE PRESENTAZIONE