Strutture dati dinamiche in C

Emilio Di Giacomo

Strutture dati

- Una struttura dati è un “contenitore” in cui i dati sono organizzati in maniera che possano essere recuperati e manipolati efficientemente
- Un esempio di struttura dati che abbiamo utilizzato ampiamente è l’array
- In un array vengono memorizzati *n* elementi di uno stesso tipo
  - ogni elemento viene messo in corrispondenza con un indice da 0 a *n*-1
  - è possibile recuperare un elemento o modificarlo specificando il suo indice

Strutture dati statiche e dinamiche

- Si distingue tra strutture dati statiche e strutture dati dinamiche
- In una struttura dati statica la capacità, cioè il numero di elementi che è possibile memorizzare nella struttura, è fissato all'atto della creazione
  - Un esempio di struttura dati statica è l'array
- In una struttura dati dinamica invece la capacità non è fissata a priori

Strutture dati statiche e dinamiche

- Una struttura dati statica ci obbliga a sapere il numero massimo di elementi che dovremo memorizzare all'interno della struttura nel momento in cui questa viene creata
- Ci sono casi in cui questa informazione non è nota e si vorrebbe poter disporre di una struttura dati in grado di crescere in base alle esigenze

Strutture dati statiche e dinamiche

- Esistono diverse strutture dati dinamiche che si differenziano per le operazioni di manipolazione e interrogazione messe a disposizione e per l'efficienza di queste operazioni
- Non parleremo in dettaglio di strutture dati dinamiche
- Vedremo un solo esempio con l'obiettivo di capire come si realizza una struttura dati dinamica mediate strutture autoreferenziali
  - l'esempio sarà relativo alle liste collegate

Liste

- Una lista è una struttura dati che permette di rappresentare una sequenza finita di elementi.
- Per ciascun elemento della lista tranne il primo e l'ultimo è definito il successore e il predecessore
  - Il primo elemento ha soltanto il successore
  - l'ultimo elemento ha soltanto il predecessore
- Il primo elemento viene detto testa della lista,
- l'ultimo elemento è detto coda della lista
- Il numero di elementi è la lunghezza della lista.

Liste

- Nel seguito descriveremo le liste usando quella che viene chiamata la notazione parentetica
- In base a tale notazione una lista viene rappresentata elencando i suoi elementi nell'ordine in cui appaiono nella sequenza, separati da virgole e racchiusi tra parentesi tonde

Liste

- Ad esempio, (12,2,54,7) rappresenta una lista con quattro elementi
  - 12 è la testa della lista mentre 7 è la coda della lista; 2 è il successore di 12 e 54 è il predecessore di 7
- La lista ("alfa") contiene la sola stringa "alfa" che è al contempo testa e coda della lista
- La lista () è la lista vuota, cioè la lista che non contiene alcun elemento
  - in quest'ultimo caso la testa e la coda della lista sono nulli

Liste collegate

- Un modo di realizzare una lista (e anche altre strutture dati dinamiche) in C è tramite l'uso di strutture dati autoreferenziali
- L'informazione sulla sequenza di elementi non è centralizzata (come avviene ad esempio con gli array) ma è distribuita sugli elementi stessi
- Ogni elemento presente nella lista è "collegato" a quello che lo segue
- Per questo si parla di liste collegate
  - più in generale di rappresentazioni collegate di una struttura dati

Liste collegate

- La gestione distribuita dei riferimenti avviene mediante strutture autoreferenziali, che chiameremo anche nodi della lista
- Ciascun nodo ha due campi:
  - un campo che memorizza l'elemento da memorizzare nella lista
  - un puntatore al nodo successivo
- Per gestire la lista è sufficiente mantenere un riferimento al primo nodo della lista

Liste collegate

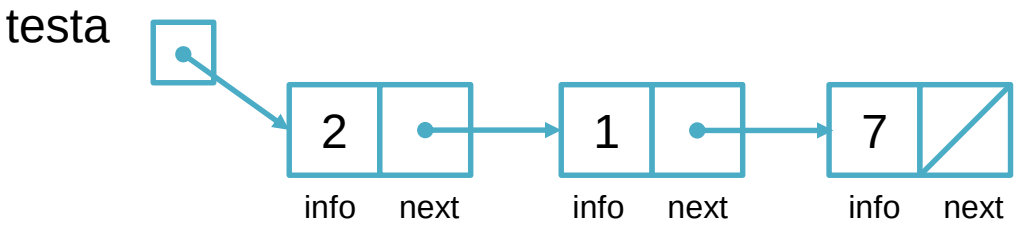
- La seguente *struct* definisce nodi in cui è possibile memorizzare elementi di tipo *int*

```
typedef struct node {
    int info;
    struct node *next;
} node;
```

- In questo modo ogni nodo memorizza un elemento ed è collegato (tramite il puntatore) al nodo successivo

Liste collegate

- Rappresentazione grafica di una lista collegata



- *testa* è un puntatore al primo nodo della lista
- Ogni nodo punta al successivo
- L'ultimo nodo della lista non punta a nulla
  - il puntatore *next* sarà pari a *NULL*

Allocaz./Deallocaz. della memoria

- Per gestire strutture dati dinamiche è necessaria l'allocazione dinamica della memoria
- Cioè la possibilità di
  - richiedere nuova memoria quando necessario
  - rilasciare memoria precedentemente allocata quando non più necessaria
- Le funzioni per l'allocazione/deallocazione della memoria sono la *malloc* e la *free* entrambe definite nella libreria standard *stdlib.h*

malloc(…)

- La funzione *malloc* alloca una certa quantità di memoria
- Riceve come parametro il numero di byte da allocare
  - tipicamente si usa la funzione *sizeof*
- Restituisce un puntatore alla prima cella di memoria allocata
  - tale puntatore è di tipo *void **
- Se la memoria non è stata allocata viene restituito *NULL*

malloc(…)

- Esempio:

*struct node * nPtr=malloc(sizeof(struct node));*

- La funzione *malloc* alloca memoria per un numero di byte pari a *sizeof(struct node)*
  - cioè la quantità necessaria a rappresentare una struttura di tipo *struct node*
- Il puntatore restituito viene memorizzato nella variabile *nPtr* di tipo *struct node **
  - si ricordi che un puntatore di tipo *void ** può essere assegnato a qualunque variabile puntatore

free(…)

- La funzione *free* rilascia una certa quantità di memoria precedentemente allocata
- Riceve come parametro un puntatore alla memoria da liberare
  - il parametro è di tipo *void **
- Esempio:

  free(nPtr);

Ancora sull'allocazione di memoria

- È buona norma rilasciare la memoria allocata quando essa non serve più
- Se non lo facessimo la memoria resterebbe allocata anche se non la stiamo utilizzando più
  - ciò potrebbe portare a non avere memoria sufficiente anche se in realtà c'è memoria potenzialmente utilizzabile

Liste collegate

- Scriveremo adesso un programma che permette di gestire una lista collegata di interi
- Il programma creerà una lista e ci permetterà di:
  - aggiungere un elemento alla lista
  - rimuovere un elemento dalla lista
  - visualizzare l'intera lista

Struct per i nodi

- Struct per rappresentare i nodi

```
struct nodo{
     int info;
     struct nodo * next;
};
```

- Alias

```
typedef struct nodo node;
typedef node * nodePtr;
```

Funzione print()

- La funzione *print* stampa l'intera lista secondo la notazione parentetica
- Riceve un parametro di tipo *nodePtr* (cioè un puntatore a *node*)
  - tale puntatore dovrà puntare al primo elemento della lista

Funzione print()

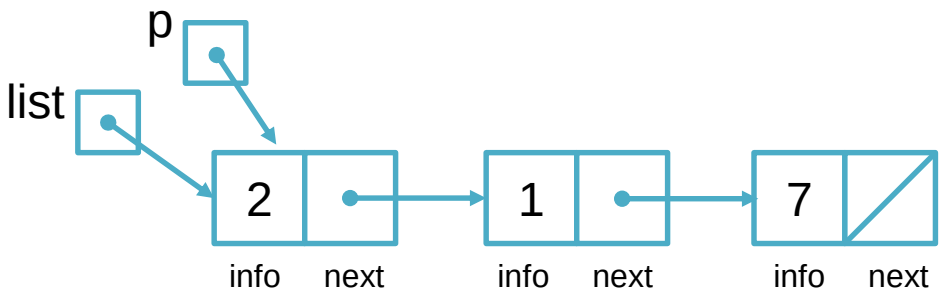
```
void print(nodePtr list){
     printf("(");
     nodePtr p=list;

     while(p!=NULL){
          printf("%d",p->info);
          p=p->next;
          if(p!=NULL)
               printf(", ");
     }

     printf(")");
}
```

Funzione print()

- Il ciclo *while* scorre l'intera lista tramite il puntatore *p*
  - esso inizialmente punta al primo nodo della lista
  - ad ogni iterazione avanza da un nodo al successivo (dopo aver stampato il contenuto del nodo corrente)
  - quando *p* diviene uguale a *NULL* il ciclo termina

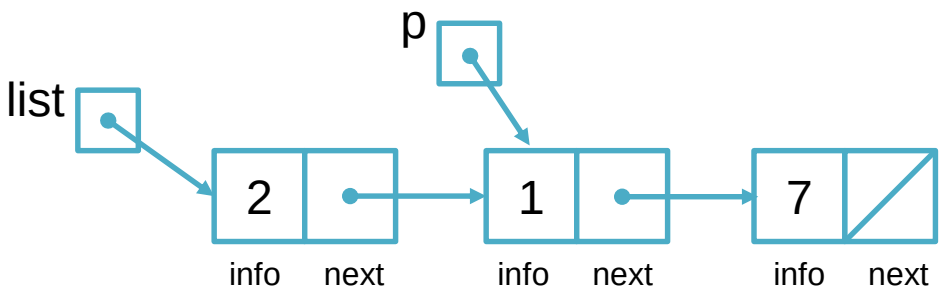
Funzione print()

- Il ciclo *while* scorre l'intera lista tramite il puntatore *p*
  - esso inizialmente punta al primo nodo della lista
  - ad ogni iterazione avanza da un nodo al successivo (dopo aver stampato il contenuto del nodo corrente)
  - quando *p* diviene uguale a *NULL* il ciclo termina



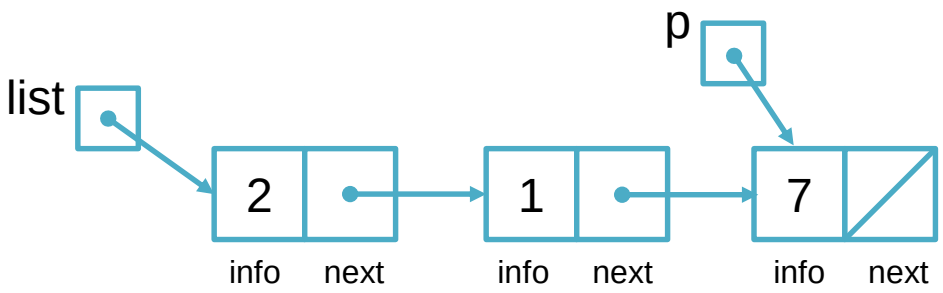
Funzione print()

- Il ciclo *while* scorre l'intera lista tramite il puntatore *p*
  - esso inizialmente punta al primo nodo della lista
  - ad ogni iterazione avanza da un nodo al successivo (dopo aver stampato il contenuto del nodo corrente)
  - quando *p* diviene uguale a *NULL* il ciclo termina



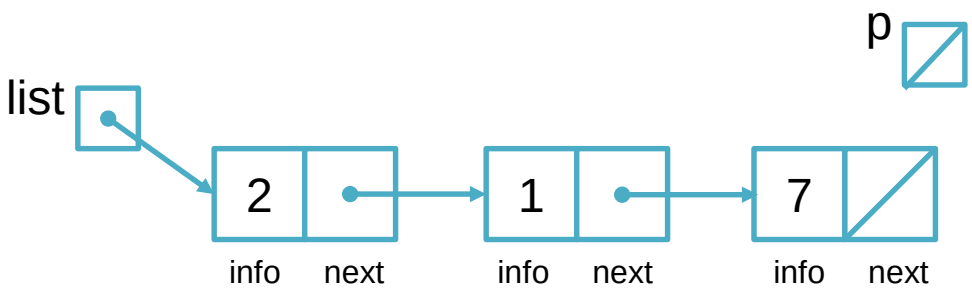
Funzione print()

- Il ciclo *while* scorre l'intera lista tramite il puntatore *p*
  - esso inizialmente punta al primo nodo della lista
  - ad ogni iterazione avanza da un nodo al successivo (dopo aver stampato il contenuto del nodo corrente)
  - quando *p* diviene uguale a *NULL* il ciclo termina



Funzione print()

- Il ciclo *while* scorre l'intera lista tramite il puntatore *p*
  - esso inizialmente punta al primo nodo della lista
  - ad ogni iterazione avanza da un nodo al successivo (dopo aver stampato il contenuto del nodo corrente)
  - quando *p* diviene uguale a *NULL* il ciclo termina



Funzione insert()

- La funzione *insert* inserisce un intero dato in coda alla lista
  - cioè come ultimo elemento
- la funzione deve ricevere in input la lista (cioè il puntatore al primo elemento) e il valore da inserire
- Poiché il puntatore al primo elemento della lista deve poter essere modificato esso deve essere passato per riferimento
  - sarà quindi di tipo *nodePtr **
  - cioè puntatore a puntatore a *node*

Funzione insert()

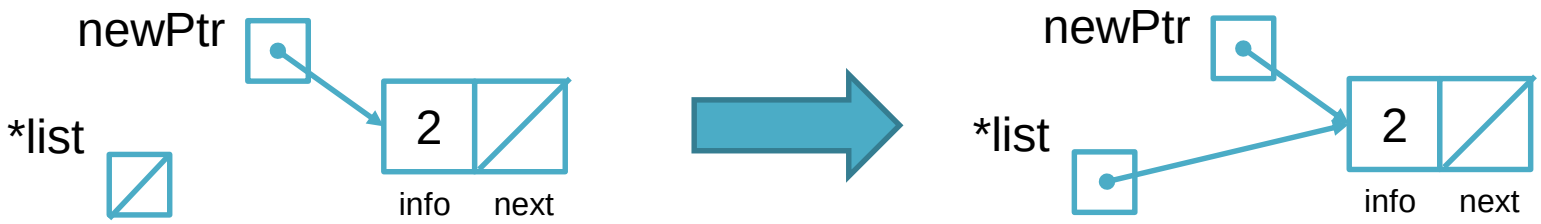
```
void insert(nodePtr *list, int value){
        nodePtr newPtr=malloc(sizeof(node));
        if(newPtr!=NULL){
                newPtr->info=value;
                newPtr->next=NULL;

                if(*list==NULL){
                        *list=newPtr;
                }else{
                        nodePtr p=*list;
                        while(p->next!=NULL)
                                p=p->next;
                        p->next=newPtr;
                }
        }
}
```

Funzione insert()

- Viene innanzi tutto allocata memoria per il nuovo nodo
- Qualora l'allocazione abbia avuto successo si assegnano i valori ai campi del nodo creato:
  - al campo *info* viene assegnato il valore *value*
  - al campo *next* viene assegnato il valore *NULL*
    - il nuovo nodo sarà l'ultimo della lista
- Si hanno poi due possibili casi:
  - la lista era vuota prima dell'inserimento
  - la lista non era vuota prima dell'inserimento

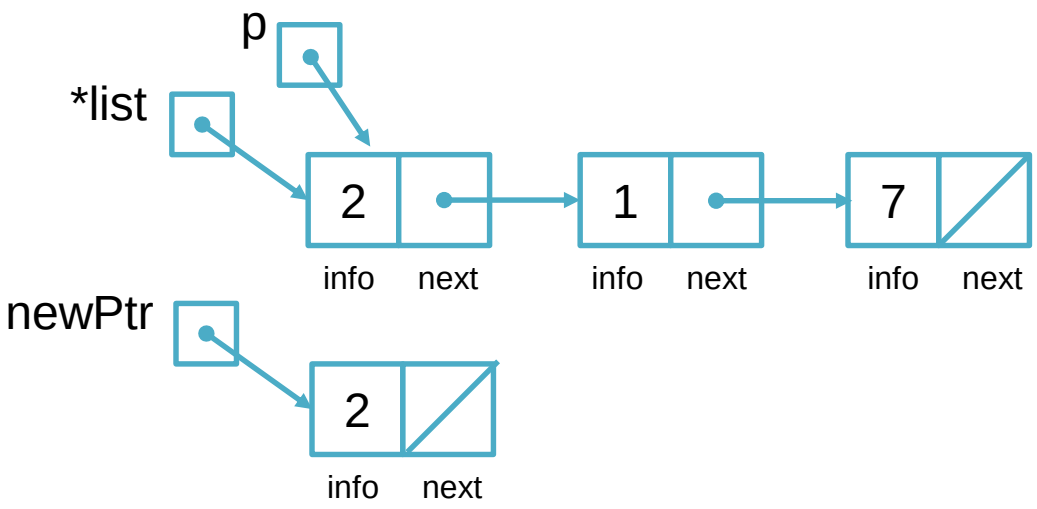
Funzione insert()

- Se la lista era vuota il nuovo nodo deve essere puntato da **list*



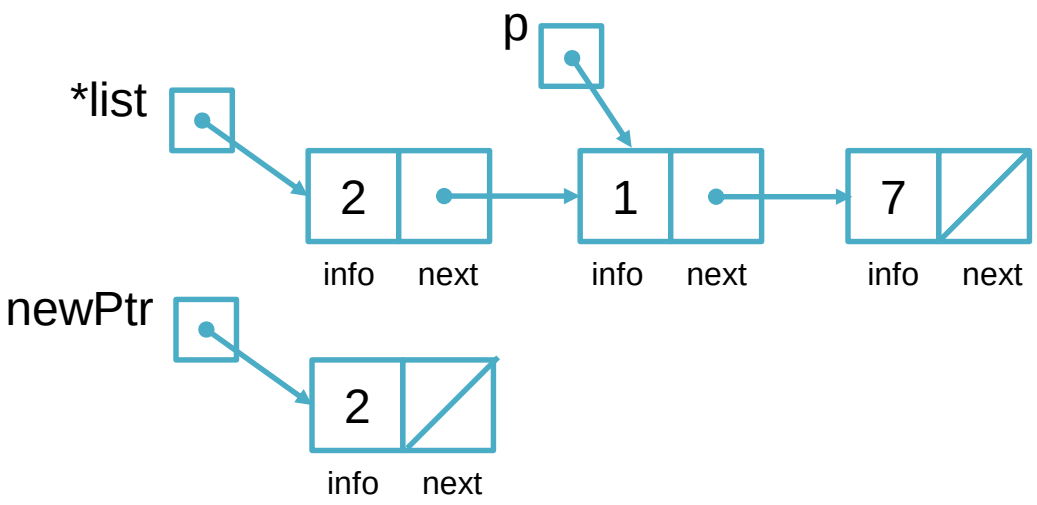
Funzione insert()

- Se la lista non era vuota bisogna scorrerla tutta per inserire il nuovo nodo in coda
  - il puntatore *p* viene fatto scorrere fino a che punta l'ultimo nodo



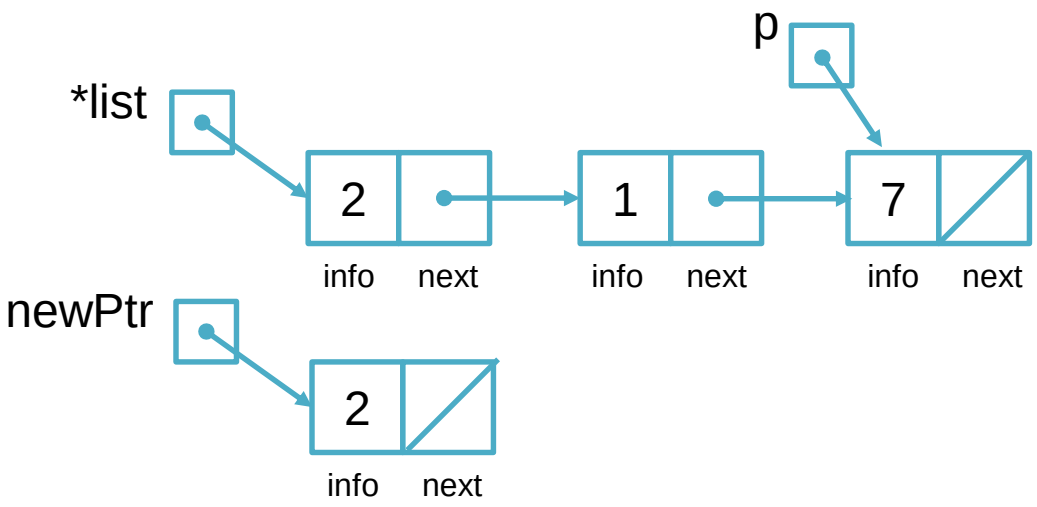
Funzione insert()

- Se la lista non era vuota bisogna scorrerla tutta per inserire il nuovo nodo in coda
  - il puntatore *p* viene fatto scorrere fino a che punta l'ultimo nodo



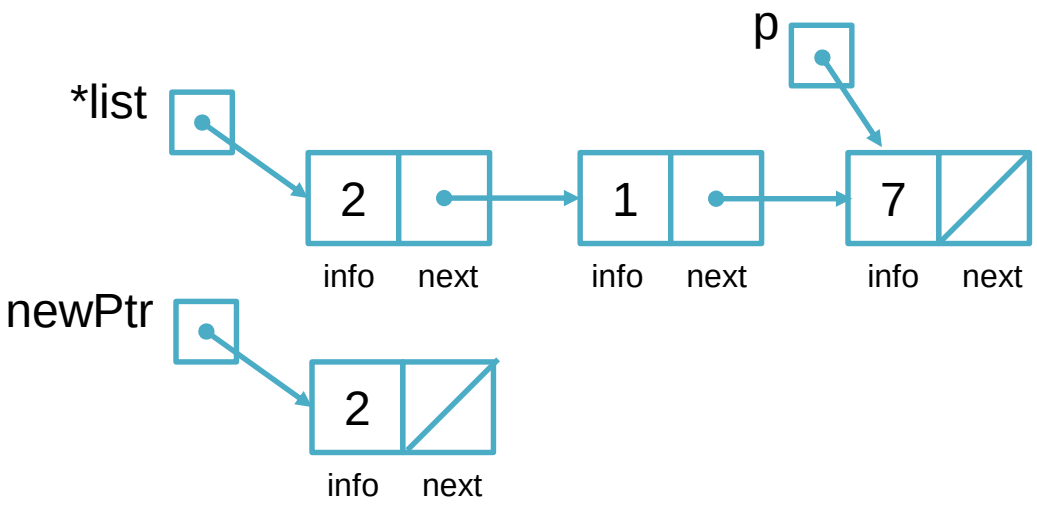
Funzione insert()

- Se la lista non era vuota bisogna scorrerla tutta per inserire il nuovo nodo in coda
  - il puntatore *p* viene fatto scorrere fino a che punta l'ultimo nodo



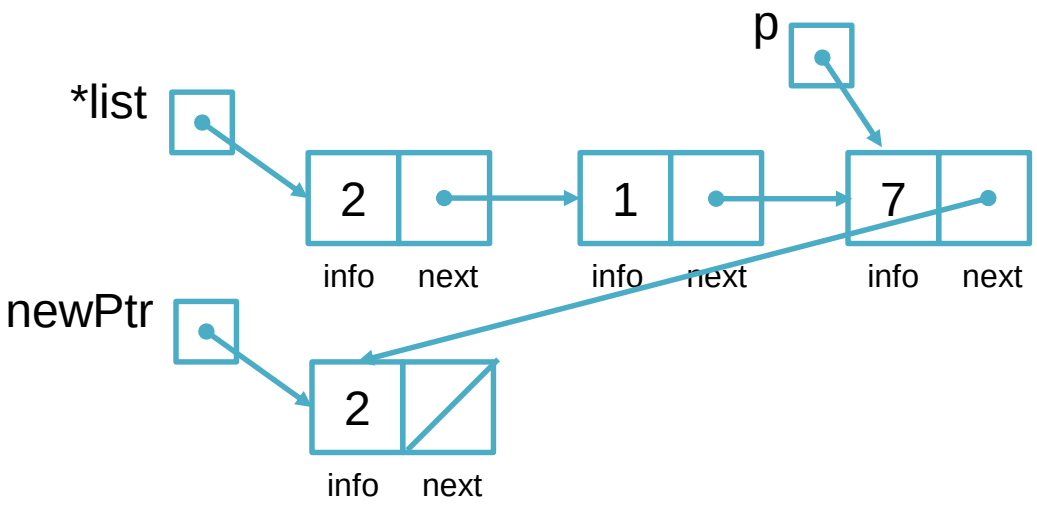
Funzione insert()

- Se la lista non era vuota bisogna scorrerla tutta per inserire il nuovo nodo in coda
  - il puntatore *p* viene fatto scorrere fino a che punta l'ultimo nodo
  - il campo *next* dell'ultimo nodo viene fatto puntare al nuovo nodo



Funzione insert()

- Se la lista non era vuota bisogna scorrerla tutta per inserire il nuovo nodo in coda
  - il puntatore *p* viene fatto scorrere fino a che punta l'ultimo nodo
  - il campo *next* dell'ultimo nodo viene fatto puntare al nuovo nodo



Funzione delete()

- La funzione *delete* riceve un valore intero e rimuove la prima occorrenza di quel valore dalla lista
  - se il valore non è presente non fa nulla
- la funzione deve ricevere in input la lista (cioè il puntatore al primo elemento) e il valore da rimuovere
- Poiché il puntatore al primo elemento della lista deve poter essere modificato esso deve essere passato per riferimento
  - sarà quindi di tipo *nodePtr **
  - cioè puntatore a puntatore a *node*

Funzione delete()

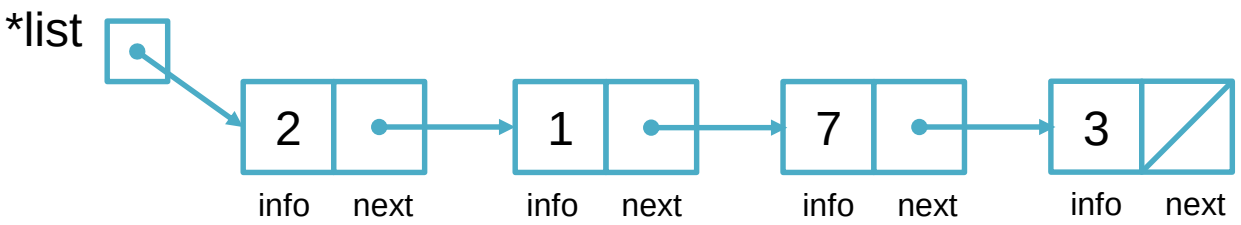
```
void delete(nodePtr *list, int value){
        if(*list!=NULL){
                if((*list)->info==value){
                        nodePtr p =(*list);
                        *list=(*list)->next;
                        free(p);
                }else{
                        nodePtr p=*list;
                        while(p->next!=NULL && p->next->info!=value)
                                p=p->next;
                        if(p->next!=NULL){
                                nodePtr q=p->next;
                                p->next=p->next->next;
                                free(q);
                        }
                }
        }
}
```

Funzione delete()

- Viene innanzi tutto controllato se la lista è vuota
  - in tal caso non succede nulla
- Si hanno poi due possibili casi:
  - il nodo da rimuovere è il primo
  - il nodo da rimuovere non è il primo

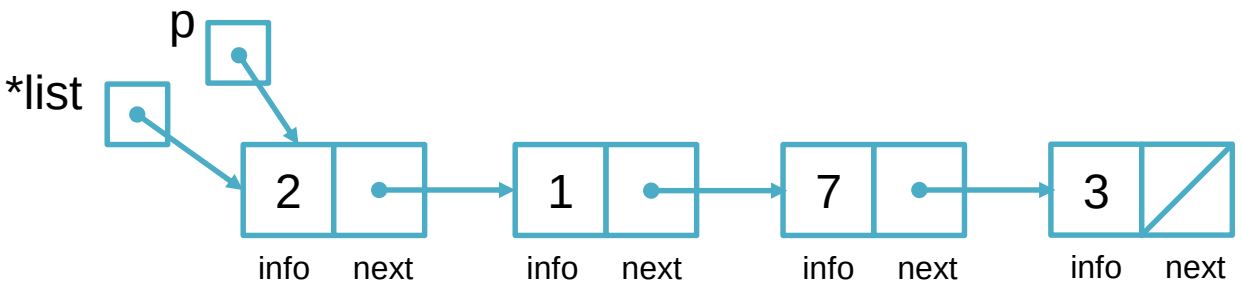
Funzione delete()

- Se il nodo da rimuovere è il primo è necessario modificare **list*



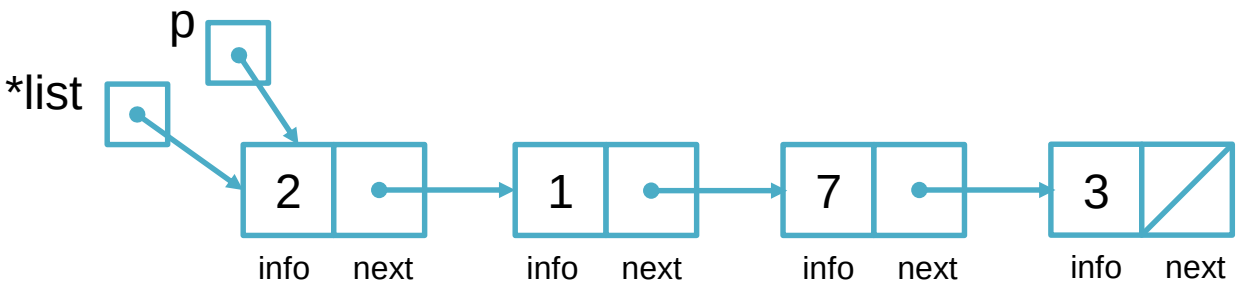
Funzione delete()

- Se il nodo da rimuovere è il primo è necessario modificare **list*
  - il puntatore *p* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)



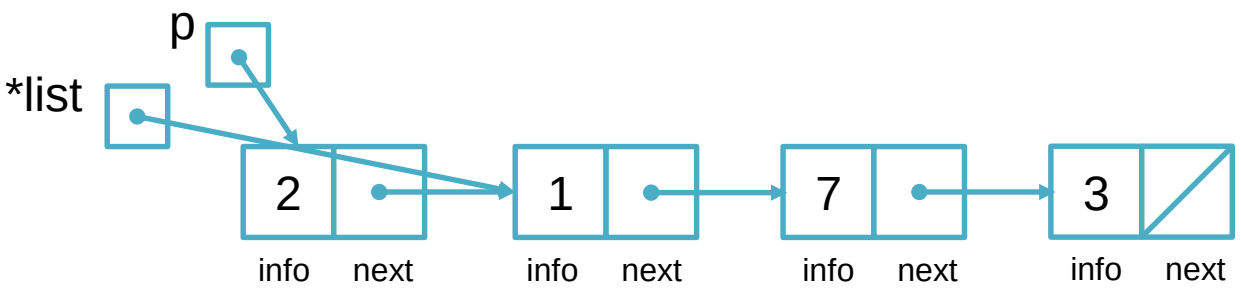
Funzione delete()

- Se il nodo da rimuovere è il primo è necessario modificare **list*
  - il puntatore *p* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)
  - **list* viene posto uguale al campo *next* del primo nodo
    - quindi punta al secondo nodo (o a *NULL* se la lista aveva un solo elemento)



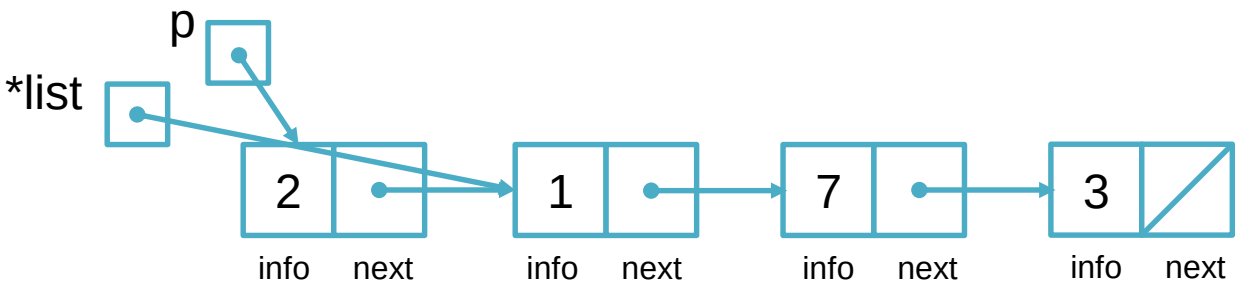
Funzione delete()

- Se il nodo da rimuovere è il primo è necessario modificare **list*
  - il puntatore *p* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)
  - **list* viene posto uguale al campo *next* del primo nodo
    - quindi punta al secondo nodo (o a *NULL* se la lista aveva un solo elemento)



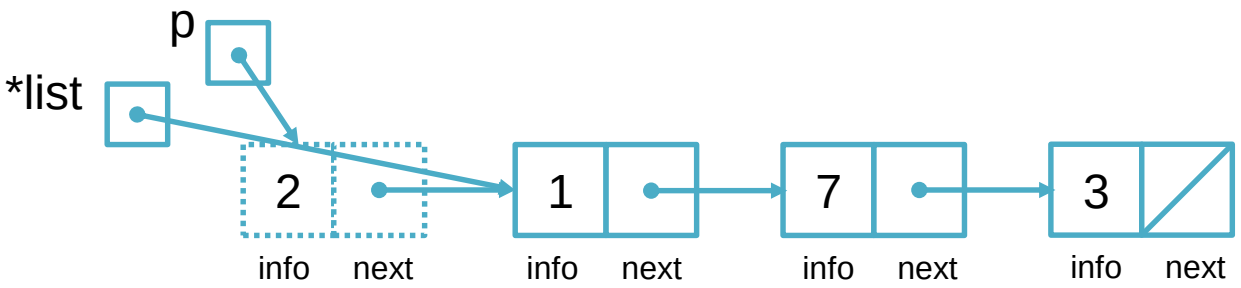
Funzione delete()

- Se il nodo da rimuovere è il primo è necessario modificare **list*
  - il puntatore *p* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)
  - **list* viene posto uguale al campo *next* del primo nodo
    - quindi punta al secondo nodo (o a *NULL* se la lista aveva un solo elemento)
  - il nodo puntato da *p* viene deallocato



Funzione delete()

- Se il nodo da rimuovere è il primo è necessario modificare **list*
  - il puntatore *p* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)
  - **list* viene posto uguale al campo *next* del primo nodo
    - quindi punta al secondo nodo (o a *NULL* se la lista aveva un solo elemento)
  - il nodo puntato da *p* viene deallocato

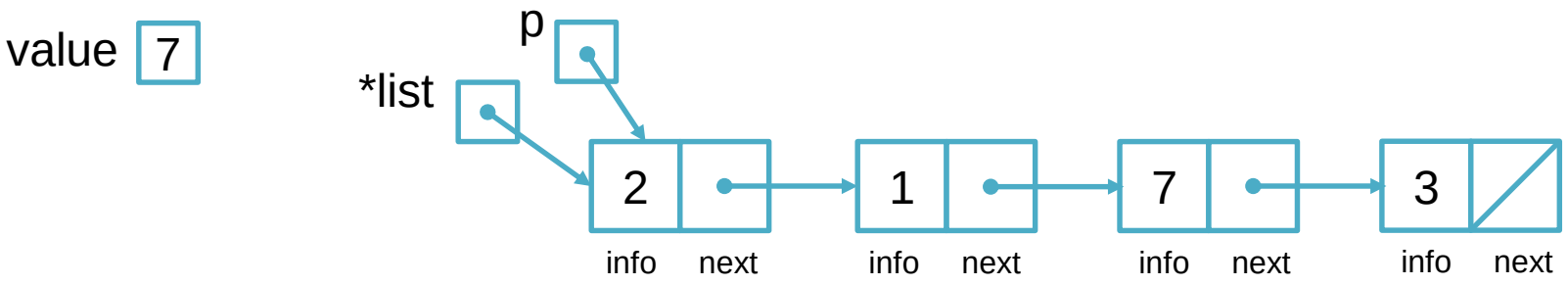


Funzione delete()

- Se il nodo da rimuovere non è il primo bisogna scorrere la lista per cercare il nodo da rimuovere
  - il puntatore *p* viene fatto scorrere fino a che punta al nodo precedente quello da rimuovere
  - oppure punta all'ultimo
    - in questo caso nessun elemento deve essere rimosso

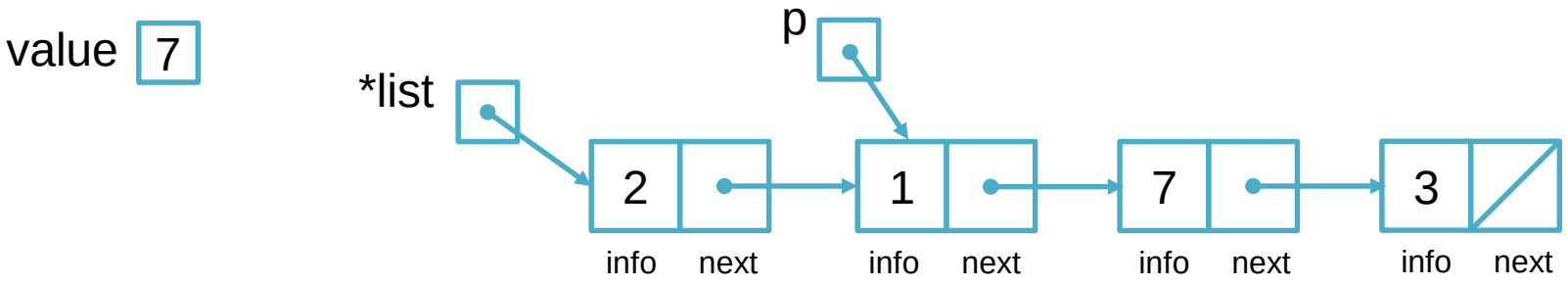
Funzione delete()

- Se il nodo da rimuovere non è il primo bisogna scorrere la lista per cercare il nodo da rimuovere
  - il puntatore *p* viene fatto scorrere fino a che punta al nodo precedente quello da rimuovere
  - oppure punta all'ultimo
    - in questo caso nessun elemento deve essere rimosso



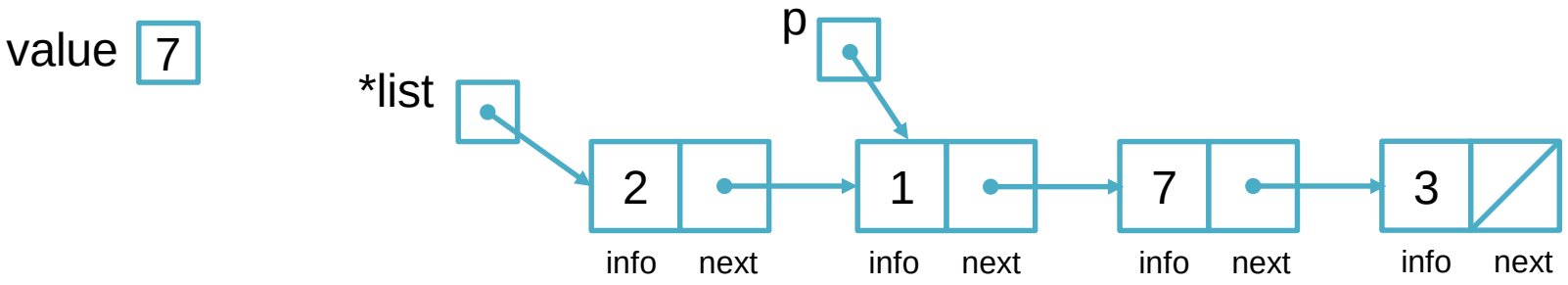
Funzione delete()

- Se il nodo da rimuovere non è il primo bisogna scorrere la lista per cercare il nodo da rimuovere
  - il puntatore *p* viene fatto scorrere fino a che punta al nodo precedente quello da rimuovere
  - oppure punta all'ultimo
    - in questo caso nessun elemento deve essere rimosso



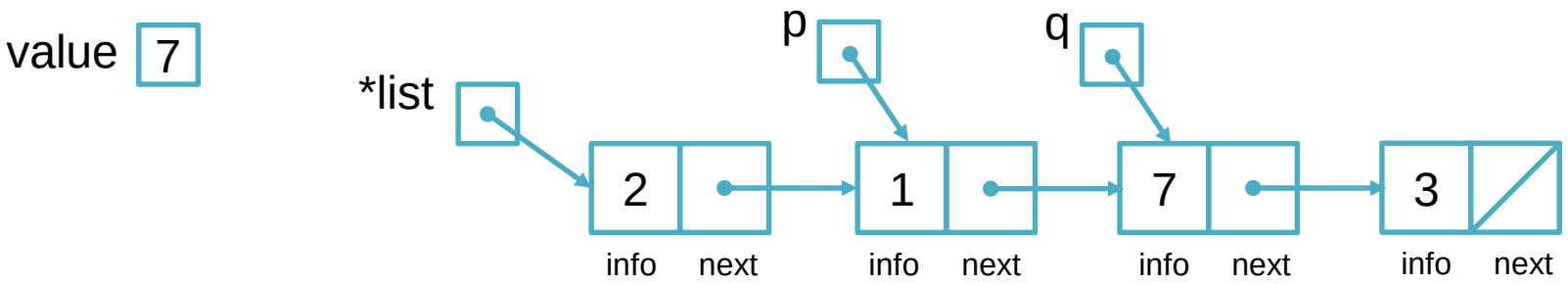
Funzione delete()

- Se è stato trovato un nodo da rimuovere
  - il puntatore *q* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)



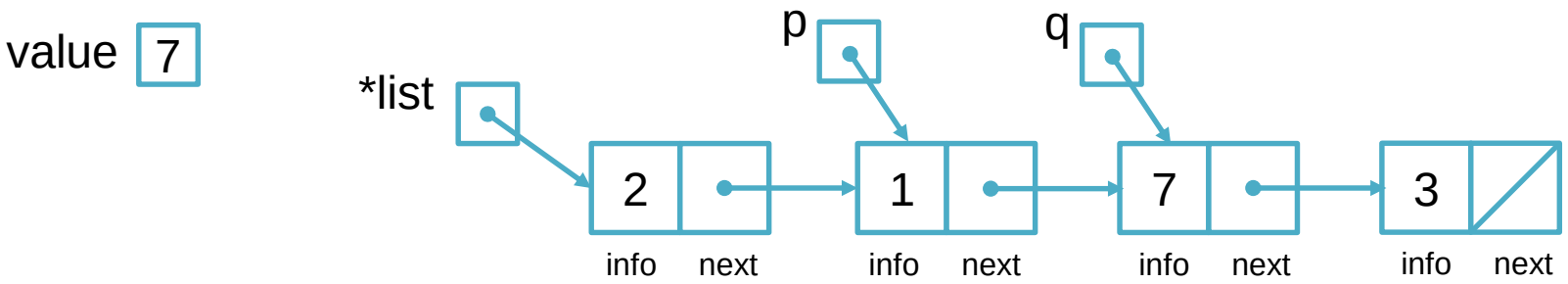
Funzione delete()

- Se è stato trovato un nodo da rimuovere
  - il puntatore *q* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)



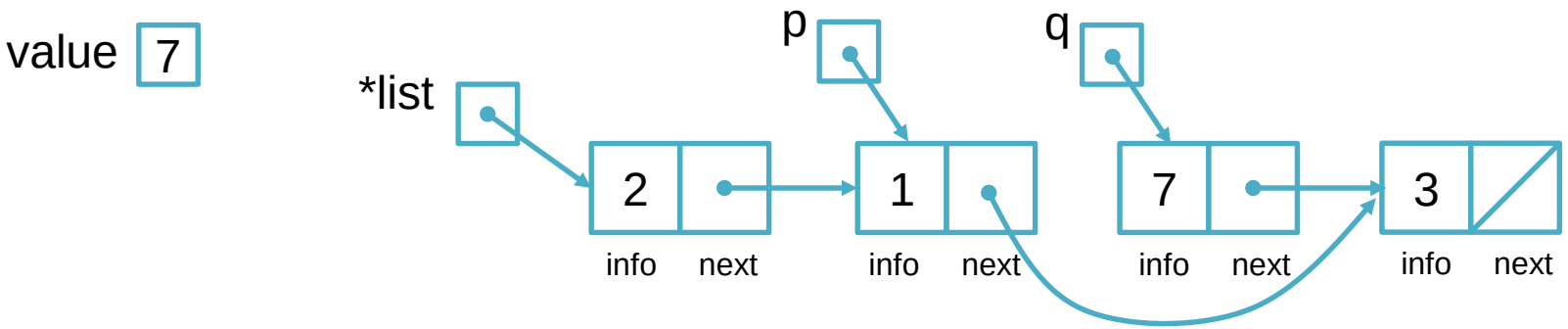
Funzione delete()

- Se è stato trovato un nodo da rimuovere
  - il puntatore *q* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)
  - il campo *next* del nodo puntato da *p* viene posto uguale al campo *next* del nodo da rimuovere
    - in questo modo punta al successore del nodo da rimuovere o a *NULL* (se il nodo da rimuovere è l'ultimo)



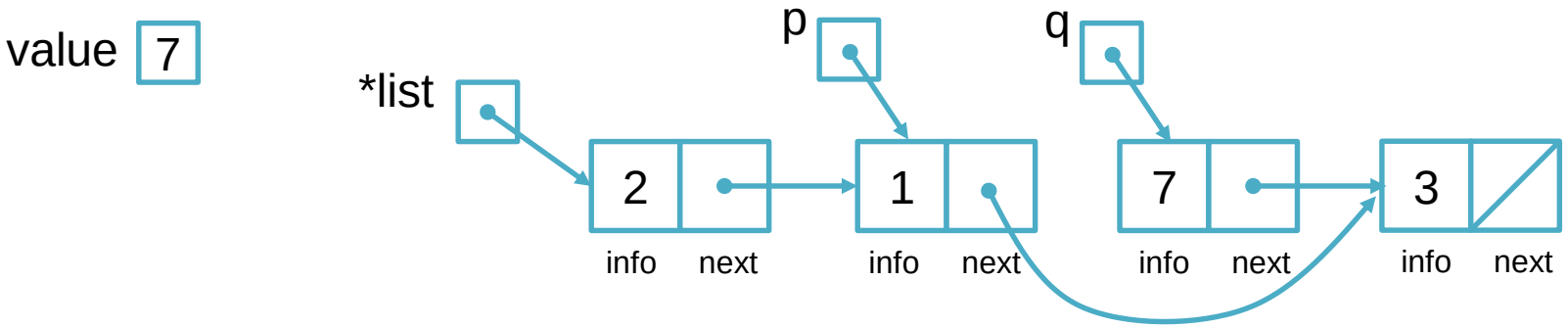
Funzione delete()

- Se è stato trovato un nodo da rimuovere
  - il puntatore *q* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)
  - il campo *next* del nodo puntato da *p* viene posto uguale al campo *next* del nodo da rimuovere
    - in questo modo punta al successore del nodo da rimuovere o a *NULL* (se il nodo da rimuovere è l'ultimo)



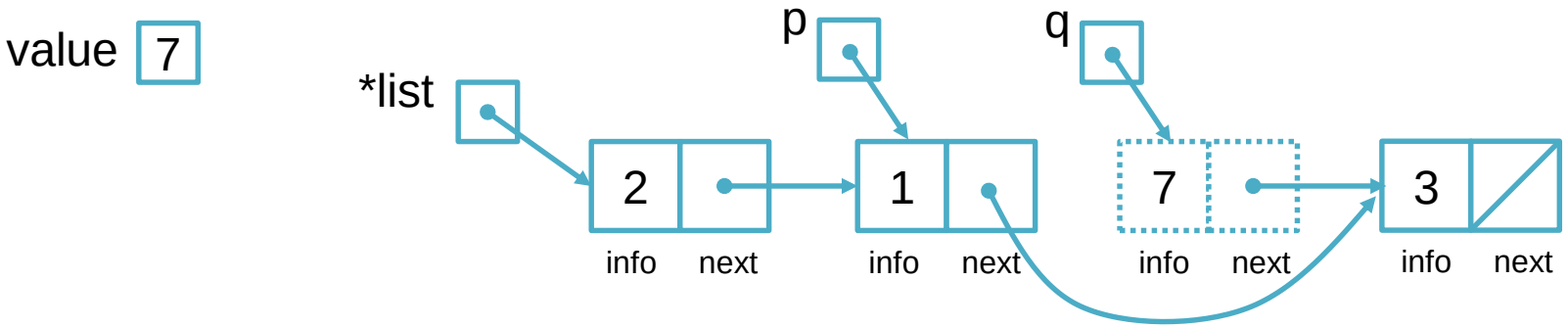
Funzione delete()

- Se è stato trovato un nodo da rimuovere
  - il puntatore *q* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)
  - il campo *next* del nodo puntato da *p* viene posto uguale al campo *next* del nodo da rimuovere
    - in questo modo punta al successore del nodo da rimuovere o a *NULL* (se il nodo da rimuovere è l'ultimo)
  - il nodo puntato da *q* viene deallocato



Funzione delete()

- Se è stato trovato un nodo da rimuovere
  - il puntatore *q* viene fatto puntare al nodo da rimuovere (per poter poi deallocare la memoria)
  - il campo *next* del nodo puntato da *p* viene posto uguale al campo *next* del nodo da rimuovere
    - in questo modo punta al successore del nodo da rimuovere o a *NULL* (se il nodo da rimuovere è l'ultimo)
  - il nodo puntato da *q* viene deallocato



Il programma di gestione delle liste

```
int main(){
	nodePtr head=NULL; // lista vuota

	printf("Opzioni:\n");
	printf("1 - Inserisci un elemento\n");
	printf("2 - Rimuovi un elemento\n");
	printf("3 - Esci\n");

	int opzione;
	printf("Scegli un'opzione\n");
	scanf("%d", &opzione);
	while(opzione!=3){
		...
	}
}
```

Il programma di gestione delle liste

```
...
while(opzione!=3){
        if(opzione==1){
                int value;
                printf("Dammi l'elemento da inserire\n");
                scanf("%d", &value);
                insert(&head, value);
                print(head);
                printf("\n");
        }else if(opzione==2){
                int value;
                printf("Dammi l'elemento da rimuovere\n");
                scanf("%d", &value);
                delete(&head, value);
                print(head);
                printf("\n");
        }else if(opzione!=3){
                printf("Opzione non valida");
        }
        printf("Scegli un'opzione\n");
        scanf("%d", &opzione);
}
...
```

Commenti

- Quello visto è solo un esempio di uso delle liste collegate
- Esistono altri modi di rappresentare le liste
  - ad esempio nelle liste doppiamente collegate ogni nodo punta al suo predecessore e al suo successore
- Esistono inoltre altri tipi di strutture dati dinamiche
  - pile, code, alberi, ecc.
  - anche tali strutture dati possono essere realizzate usando opportunamente delle strutture autoreferenziali