

# Liste con puntatori - 1

In una lista concatenata ciascun elemento, oltre a contenere dei dati, contiene un puntatore ad un elemento dello stesso tipo.



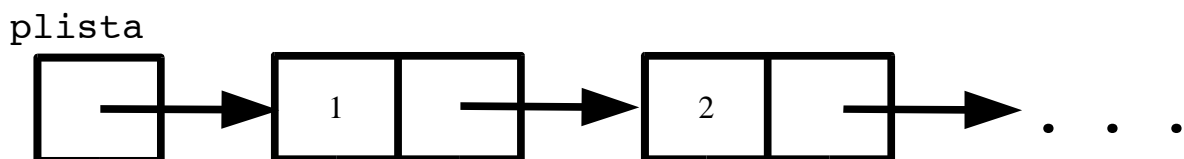
Un elemento della lista sarà quindi definito come:

```
struct element {  
    int dato;  
    struct element *next;  
};  
  
typedef struct element elem;
```

Per individuare la lista è sufficiente tenere un puntatore al suo primo elemento.

```
elem *plista = NULL;
```

La precedente riga di programma dichiara un puntatore `*plista` ad un record di tipo `element` che useremo come puntatore al primo componente della lista.



La creazione dello spazio in memoria per un componente (di tipo `element`) della lista avviene usando l'istruzione `malloc()`

```
elem *p;  
p = (elem *) malloc(elementsize);
```

Una volta creato il componente, occorre concatenarlo con il resto della lista.

---

## Liste con puntatori - 2

Notare che la dichiarazione :

```
elem *p;
```

crea solo un puntatore ad un componente della lista, cioè crea un contenitore per l'indirizzo.

Mentre la dichiarazione:

```
elem var_elem;
```

crea una variabile che conterrà il campo dato ed il campo puntatore.

Variabili di tipo `elem` possono essere create dinamicamente tramite `malloc(n)`:

```
p = (elem *) malloc(elementsize);
```

Con la precedente istruzione creiamo una variabile “anonima” che possiamo accedere solo attraverso il puntatore `p`. Il vantaggio nell'uso di `malloc` è che ci permette di creare a run-time tutte le variabili che ci occorrono.

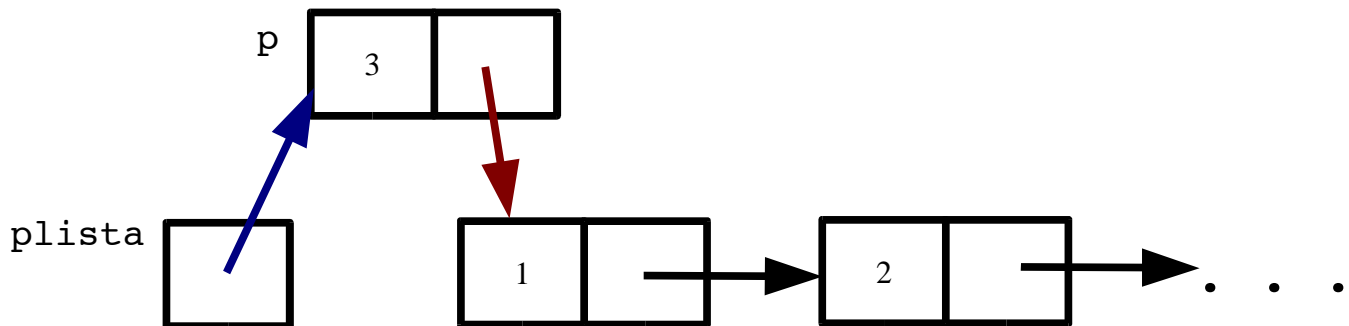
Se la parte di memoria puntata da `p`, non ci serve più dobbiamo avere cura di liberarla così da poterla riutilizzare per successive allocazioni. L'istruzione da usare è `free(p)`.

Notare che dopo l'esecuzione di `free(p)`, `p` non perde il suo valore, ma dobbiamo stare attenti a non usarlo più.

# Liste con puntatori - 3

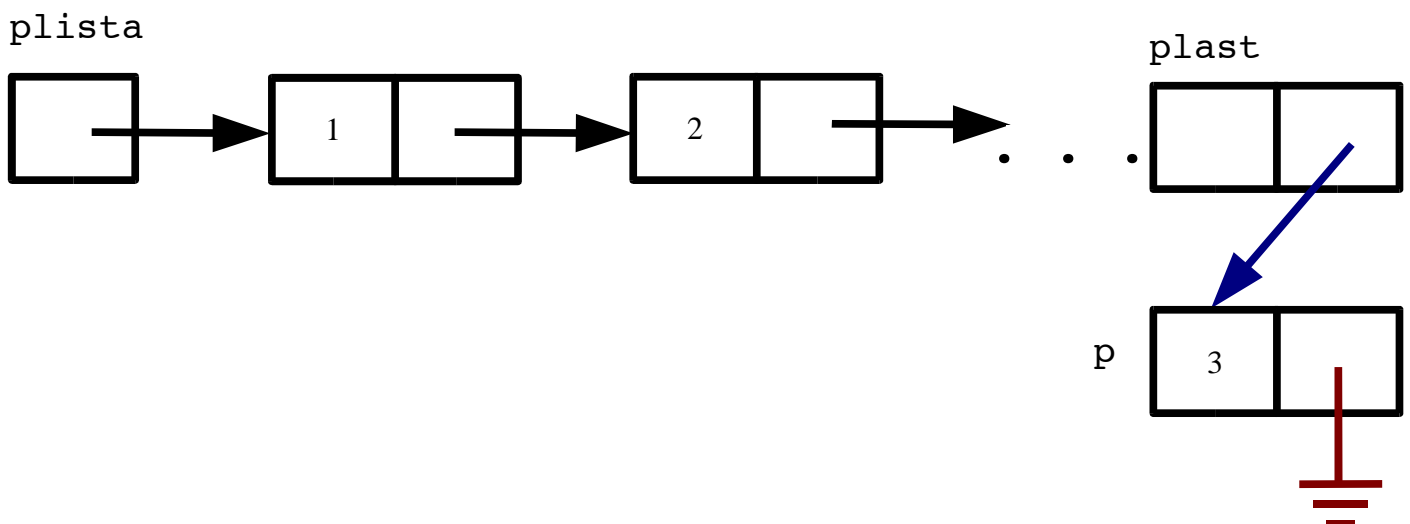
Se vogliamo fare un inserimento in testa alla lista, ovvero il componente creato recentemente sarà il primo componente della lista:

```
p->next = plista; // creiamo la freccia rossa
plista = p;      // creiamo la freccia blu
```



Mentre se vogliamo fare un inserimento in coda (ovvero il componente creato sarà l'ultimo della lista), occorre avere il puntatore all'ultimo componente già inserito (plast):

```
p->next = NULL; // creiamo la freccia rossa
if (plast == NULL) plista = p;
else plast->next = p; // creiamo la freccia blu
```



---

# Liste con puntatori - 4

Nel caso di inserimento in coda, possiamo tenere un puntatore all'ultimo componente della coda, chiamato `plast` nel precedente esempio, oppure creare una funzione che fornisce questo puntatore.

La funzione `getLast()` restituisce il puntatore all'ultimo componente della lista.

```
elem* getLast() {
    if (plista == NULL) return NULL;
    elem *p = plista;
    while (p->next != NULL) p = p->next;
    return p;
}
```

Notare che per individuare la lista basta avere il puntatore al primo componente, poichè una volta noto il primo componente posso conoscere tutti gli altri utilizzando i puntatori presenti nei vari componenti.

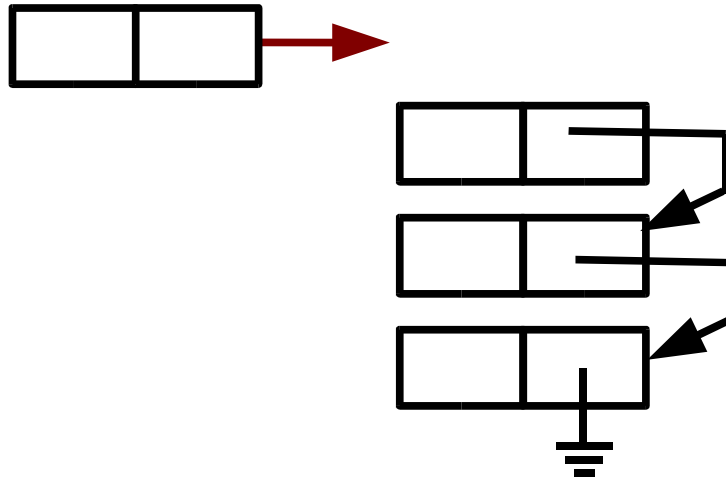
Funzioni utili per manipolare la lista: `delete()`, `visit()`, `find()`

Oltre agli inserimenti in testa ed in coda ad una lista dei nuovi componenti creati, si può fare l'inserimento in modo da mantenere l'ordinamento della lista. Dove una lista si dice ordinata se l'informazione contenuta in un nodo è minore o uguale all'informazione contenuta nel nodo successivo (se esiste).

Esercizio: implementare una funzione che realizza l'inserimento ordinato. [Suggerimento: partire dal primo nodo e scorrere fino a quando non si incontra un elemento maggiore di quello che si vuole inserire.]

# Stack - 1

E' un contenitore di elementi caratterizzato da una politica L.I.F.O. (Last In First Out). Ovvero, un nuovo elemento si può inserire solo in cima e l'estrazione di un elemento avviene solo dalla cima.



Le funzioni che lo gestiscono sono:

`push ( )` per l'inserimento di un elemento;

`pop ( )` per l'estrazione di un elemento;

`isEmpty ( )` per verificare se lo stack è vuoto;

`top()` per leggere l'elemento in cima allo stack.

Creiamo una funzione per l'inserimento di un elemento di un certo tipo in un qualsiasi stack, ovvero vogliamo che la funzione `push ( )` possa lavorare con qualsiasi stack. Per ottenere questo dobbiamo rendere la funzione parametrica ed ad essa passiamo: il puntatore ad uno stack e l'elemento che vogliamo inserire. La funzione ci deve restituire il puntatore allo stack ottenuto.

Definizione di un elemento dello stack:

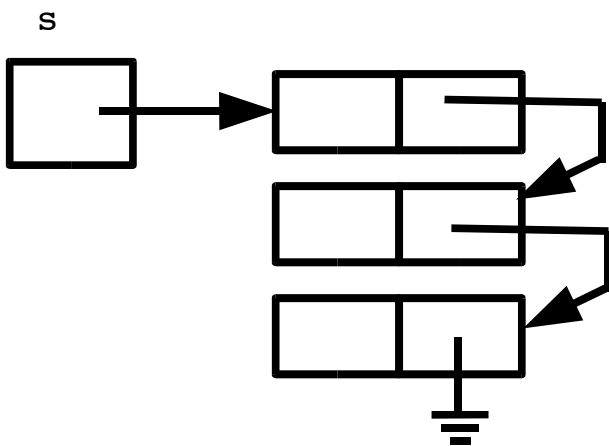
```
typedef struct element s_element; /*definiamo elemento stack*/
```

# Stack - 2

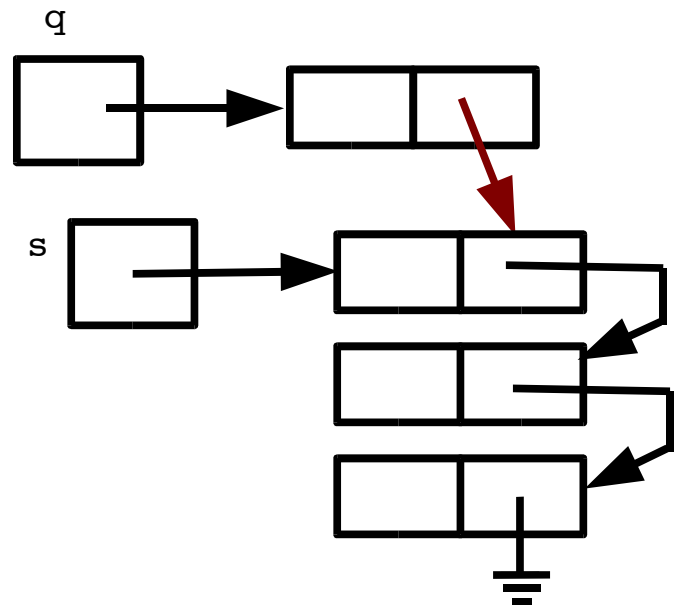
Implementazione della funzione per l'inserimento:

```
s_element *s_push(s_element *s, int n) {
    /* Prima */
    s_element *q = malloc(sizeof(s_element));
    if (q== NULL) { /* la malloc non ha avuto successo */
        printf("Errore allocazione della memoria\n");
        exit(-1);
    }
    q->dato = n;
    q->next = s;          // crea la freccia rossa
    /* Dopo */
    return q;
}
```

Prima



Dopo

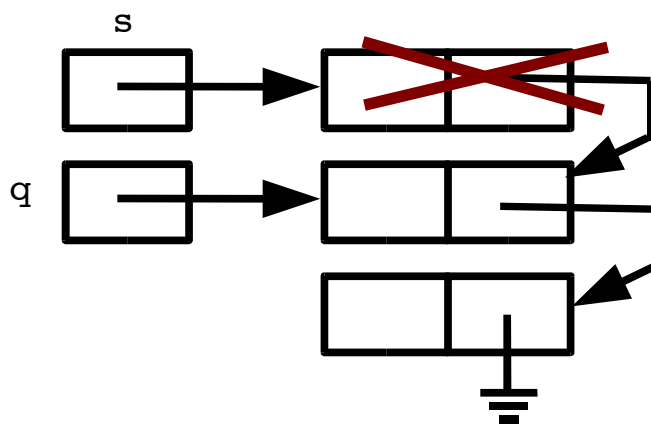


Il programma chiamante dovrà aggiornare il proprio puntatore allo stack con il valore di ritorno (q) della funzione s\_push

# Stack - 3

Analogamente, creiamo una funzione `pop()` a cui passiamo lo stack e che ci restituisce come valore di ritorno lo stack ottenuto togliendo l'elemento in cima.

```
s_element *s_pop(s_element *s) {  
    s_element *q;  
    q = s->next;  
    free(s);  
    return q;  
}
```



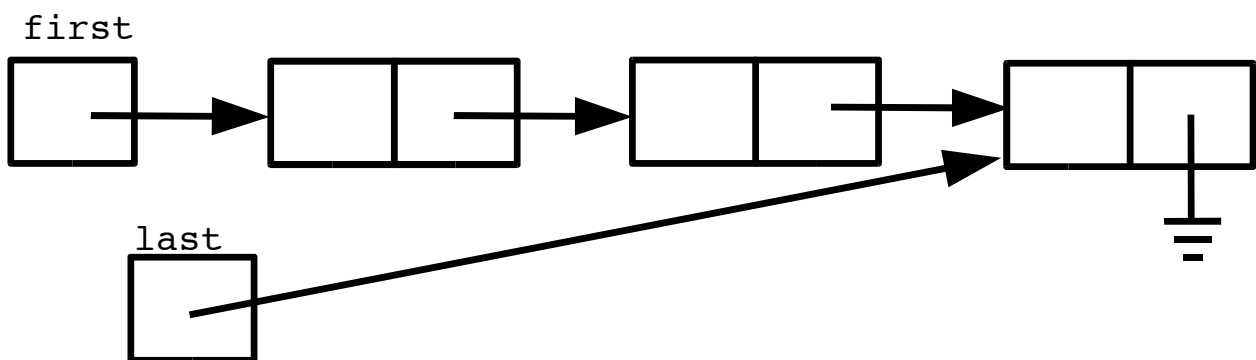
Grazie all'uso dei parametri, le funzioni così create possono manipolare tanti stack contemporaneamente presenti in memoria.

I vantaggi dell'uso dei puntatori, rispetto ad una implementazione fatta con un array: (a) posso allocare dinamicamente la memoria che mi serve in base ai dati che saranno disponibili a run-time; (b) non ho fissato la dimensione dello stack, quindi, a meno di esaurire la memoria, posso inserire tutti i dati che voglio; (c) a meno dei puntatori per la gestione della struttura, non spreco memoria, mentre con gli array devo riservare una gran numero di elementi per prevenire l'overflow.

# Coda (Queue) - 1

E' un contenitore di elementi caratterizzato da una politica F.I.F.O. (First In First Out). Ovvero, un nuovo elemento si può inserire solo in coda e l'estrazione di un elemento avviene solo dalla cima.

Occorre avere due puntatori, uno alla testa ed uno alla coda. Il primo è usato per estrarre elementi dalla struttura, il secondo consente di inserire degli altri elementi nella struttura.



Se vogliamo avere un'unica variabile per i due puntatori che individuano la coda definiamo una struttura opportuna:

```
struct queue /* struttura per rappresentare la coda */
{
    q_element *first; /* primo elemento della lista */
    q_element *last; /* ultimo elemento della lista */
};
```

Le funzioni che permettono di gestire la coda sono:

`q_in()` per l'inserimento di un elemento;

`q_out()` per l'estrazione di un elemento;

`q_isEmpty()` per verificare se è vuota;

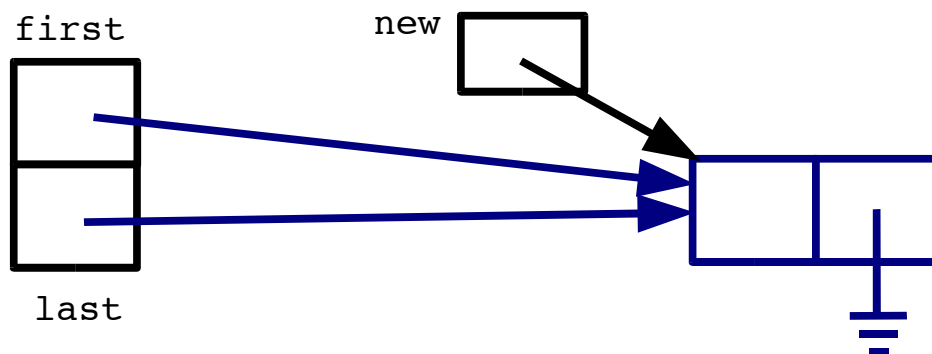
`q_first()` per leggere il primo elemento della coda.

# Coda (Queue) - 2

Creiamo la funzione `q_in()` che richiede in ingresso la coda e l'elemento da inserire:

```
void q_in(queue *q, int n) {
    q_element *new = malloc(sizeof(q_element));
    if (new == NULL) { /* la malloc non ha avuto successo */
        printf("Errore allocazione della memoria\n");
        exit(-1);
    }
    new->dato = n;
    new->next = NULL;
    if (q->first == NULL) { /* la coda e' vuota */
        q->first = q->last = new;
    }
    else { /* la coda non e' vuota */
        q->last->next = new;
        q->last = new;
    }
}
```

Primo inserimento:



Secondo inserimento

