1 Le liste

Cos'è una lista (linked list)?

È una possibile implementazione per la struttura dati astratta, **sequenza**. È costituita da una **sequenza di nodi**, contenenti dati arbitrari e $1/2$ puntatori all'elemento successivo e/o precedente.

Contiguità delle liste

È importante specificare che la contiguità degli elementi in una lista **non implica** una loro contiguità nella memoria (*contiguità lista* $\not\Rightarrow$ *contiguità in memoria*).

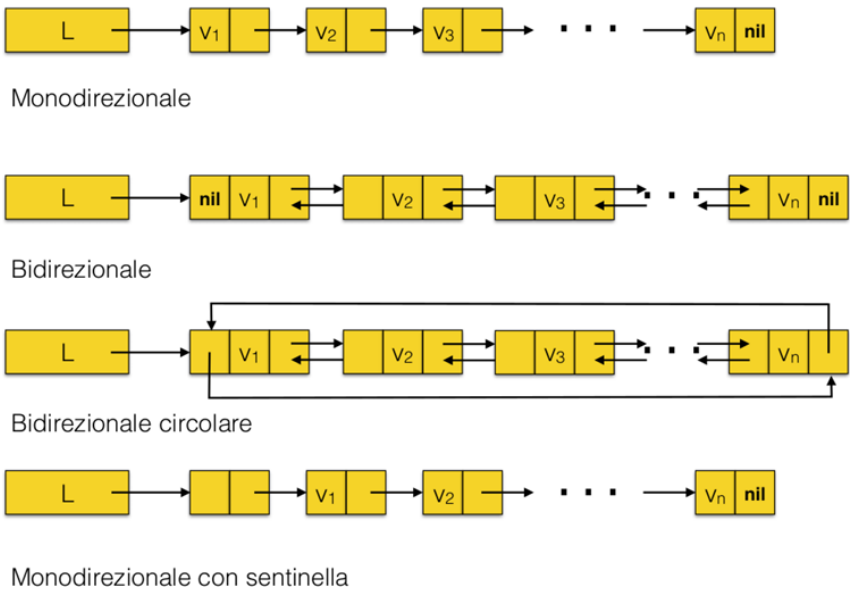
Nelle liste, infatti, i nodi **non sono necessariamente salvati in celle di memoria adiacenti**, come accade invece negli array. Ogni nodo può trovarsi in una posizione qualunque della memoria, e il **collegamento logico** tra di essi è **garantito dai puntatori**, non dalla loro vicinanza fisica.

Esempio: possiamo immaginare che la CPU debba accedere a posizioni di memoria anche molto distanti tra loro per poter percorrere l'intera lista.

Nel contesto delle **linked list**, le operazioni di **inserimento** o **cancellazione**, nota la posizione, hanno **costo costante**, cioè $O(1)$.
Questo perché vengono utilizzati dei **passaggi fissi**, aggiornando un numero limitato di puntatori tra i nodi, indipendentemente dalla lunghezza della lista.

Esempio: immaginiamo la seguente lista $[a] \to [b] \to [c]$.
Se volessimo inserire un nuovo nodo `x` dopo a, sarebbe sufficiente:
1. creare il nodo `x`;
2. impostare `x.next = b`;
3. impostare `a.next = x`.

Il numero di operazioni rimane invariato anche se la lista cresce, quindi il costo resta $O(1)$.
Al contrario, l'accesso a un elemento tramite il suo indice richiede di scorrere la lista dall'inizio fino al nodo desiderato, con un costo $O(n)$.



Le linked list presentano **diverse possibili implementazioni**:
- Liste **monodirezionali** o **bidirezionali**;
- Liste con **sentinella** e **senza sentinella**;
- Liste **circolari** e **non circolari**.

1.1 Liste concatenate

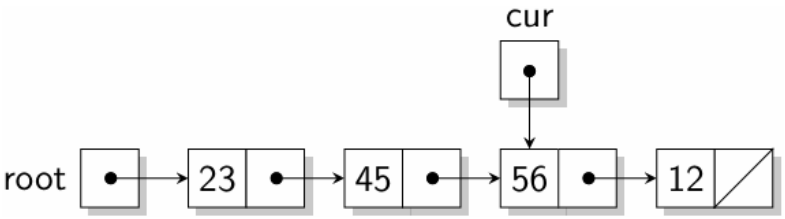
Come sono strutturate?

La **lista singolarmente concatenata** (monodirezionale) è la più semplice struttura dinamica basata su nodi. La lista è definita da un "*root (o head) pointer*" e sfrutta un **puntatore corrente** per la sua gestione.

Le liste concatenate vengono definite nel seguente modo:

```
struct slist {
int dato;
struct slist *next;
};
```

```
/* lista vuota */
struct slist *root = NULL;
```



Come si può vedere, quando viene inizializzato il puntatore root (la testa della lista) a ***null***, la lista è vuota.

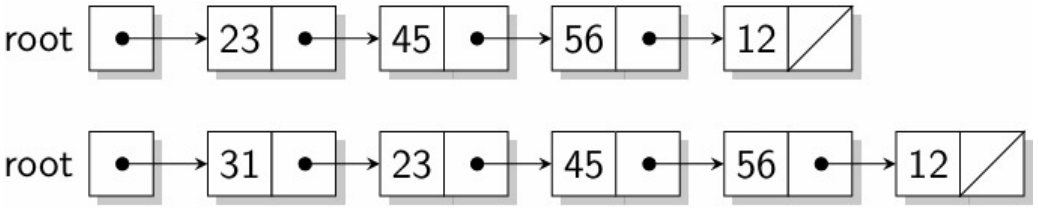
Per manipolare una lista semplicemente concatenata esistono **diverse funzioni**, tra cui:

- Inserimento di un nodo (prima del puntatore corrente);
- Cancellazione di un nodo (indicato dal puntatore corrente);
- Creazione nuova lista;
- Cancellazione lista;
- Attraversamento con esecuzione di funzione;
- Ricerca con esecuzione di funzione;
- Concatenazione;
- Conteggio;
- Inserimento ordinato.

1.1.1 Inserimento in testa

Tramite la seguente funzione è possibile inserire un nuovo elemento in testa alla lista.

```
1  slist *insert(slist *p, int elem) {
2  slist *q = malloc(sizeof(slist));
3
4  if(!q) {
5      fprintf(stderr,"Allocation error\n");
6      exit(-1);
7  }
8  q->dato = elem;
9  q->next = p;
10 return q;
11 }
```



- **Tipo di ritorno**: La funzione restituisce un puntatore a `slist`. Il puntatore restituito sarà la nuova testa della lista;
- **Parametri**: i parametri definiti nella funzione sono (`slist *p, int elem`), rispettivamente il **puntatore corrente** alla testa della lista (che può essere `null` se la lista è vuota) e il **valore da inserire** nel nuovo nodo;
- **Funzionamento del codice**: per poter inserire un nuovo nodo in testa alla lista, supponiamo di possedere una struttura dati come quella definita al capitolo 1.1.
  - *riga 2*: creo un puntatore `*q` al nuovo nodo di grandezza `slist`;
  - *riga 4-7*: eseguo un crontollo su `q` per assicurarmi che la `malloc` sia andata a buon fine;
  - *riga 8*: il campo `dato` del nuovo nodo prende il valore di `elem` passato tramite la funzione;
  - *riga 9*: Collega il nuovo nodo al resto della lista: il campo `next` del nuovo nodo punta all'elemento che era in testa (`p`).
    → **Se la lista è vuota** (`p == null`), la nuova lista contiene solo in nuovo nodo `q`, qundi: `q -> next = null;`
    → **Se la lista non è vuota**, `q` è inserito prima della vecchia testa: `q -> old_head -> ...`
  - *riga 10*: restituisce il puntatore al nuovo nodo: ora `q` è la testa della lista aggiornata

Esempio di utilizzo

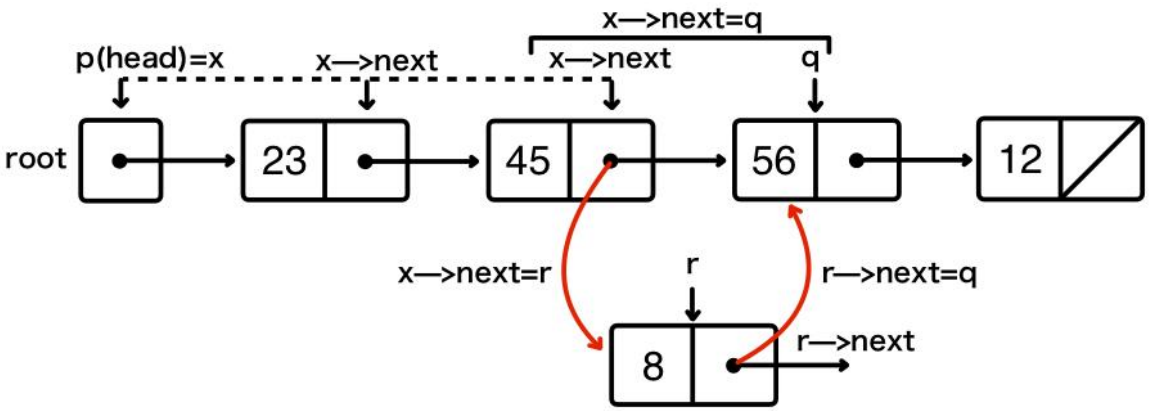
```
slist *head = NULL;       // lista vuota
head = insert(head, 10); // ora head punta al nodo con dato 10
head = insert(head, 5);  // ora head punta al nodo con dato 5; next punta
    al nodo 10
```

1.1.2 Inserimento generico

Questa funzione è un passo avanti rispetto alla `insert()`. `insert2()` permette un inserimento generico in qualunque posizione della lista (in testa, in mezzo o in coda).

```
1  slist *insert2(slist *p, slist *q, int elem) {
2  slist *x;
3  slist *r = malloc(sizeof(slist));
4  if(!r) {
5      fprintf(stderr,"Allocation error\n");
6      exit(-1);
7  }
8  r->dato = elem;
9  if(p == q){ // in testa
10     r->next=p;
11     return r;
12 }
13 else{ // in mezzo o in coda (se q==NULL)
14     for(x= p; x && x->next != q; x = x->next);
15
16     // prima di collegare mi assicuro che q sia valido
17     if(x && x->next == q){
18         r->next = q;
19         x->next = r;
20     }
```

```
21      return p;
22  }
23  }
```



- **Tipo di ritorno**: La funzione restituisce un puntatore a `slist`, ovvero il puntatore alla testa aggiornata della lista.
- **Parametri**: i parametri definiti nella funzione sono (`slist *p, slist *q, int elem`), rispettivamente:
  - il **puntatore corrente** alla testa della lista (che può essere `null` se la lista è vuota);
  - il **puntatore al nodo di riferimento** (davanti al quale inserire) oppure `null` se si vuole inserire in coda;
  - l **valore da inserire** nel nuovo nodo.
- **Funzionamento del codice**: per poter inserire un nuovo nodo in testa alla lista, supponiamo di possedere una struttura dati come quella definita al capitolo 1.1.
  - *riga 2*: `x` è un puntatore alla lista che serve come supporto per eseguire lo scorrimento della stessa;
  - *riga 3*: creo un puntatore `*r` al nuovo nodo di grandezza `slist`;
  - *riga 4-7*: eseguo un crontollo su `r` per assicurarmi che la `malloc` sia andata a buon fine;
  - *riga 8*: inizializzo il campo `dato` del nuovo nodo con il valore del campo `elem` passato tra i parametri;
  - *riga 9-12*: viene **gestito l'inseriemento in testa**. Infatti, se il nodo `q` coincide con `p`, mi basta semplicemente collegare il nuovo nodo alla vecchia testa (`r -> next = p`), e poi restituire il nuovo nodo (`return r`);
  - *riga 13-21*: viene **gestito l'inserimento in mezzo o in coda**.
    Tramite il ciclo `for` si fa scorrere il puntatore di appoggio (`x`) partendo dalla testa (`p`) della lista. Lo scorrimento continua fintanto che è verificata la condizione `x->next!=q`, e **presenta due possibilità di terminazione**:
    1. La condizione diventa `x->next==q` poiché `q` è un nodo interno alla lista. In altre parole quando `x` precede il nodo `q` specificato nei parametri, si esce dal ciclo;
    2. La condizione non si avvera perché il `q` passato nei parametri è `null`, e il ciclo termina le sue iterazioni.

    Dato che lo scorrimento ha due possibili terminazioni, bisognerà effettuare un ulteriore controllo per determinare se si tratta di un inserimento **in mezzo o in coda**.
    La condizione dell'`if` indica che `x != null` e che `x->next==q`, dunque:
    1. Si ha un **inserimento in mezzo** quando si verifica il caso 1.
       In questo caso (`r->next=q`) collega il nuovo nodo al nodo di riferimento, mentre

(`x->next=r`) collega il nodo precedente al nuovo nodo.

2. Si ha un **inserimento in coda** quando si verifica il caso 2. In questo caso `q==null`, per cui `r->next=null` diventando così l'ultimo nodo della lista, mentre il precedente ultimo nodo viene aggiornato con `x->next = r`..

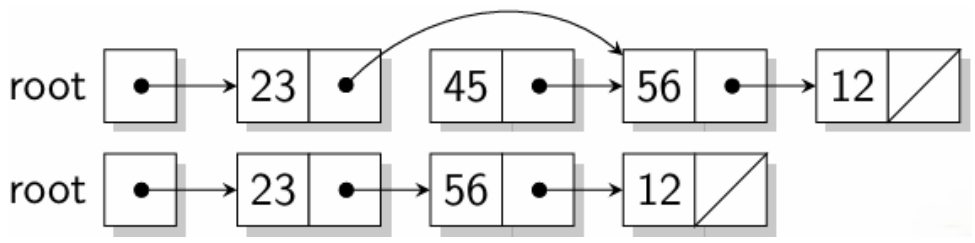
Esempio di utilizzo

```
slist *head = NULL;    // lista vuota

// Inserisco alcuni elementi in testa (uso insert2 con q == p)
head = insert2(head, head, 56);    // lista: 56
head = insert2(head, head, 45);    // lista: 45 -> 56
head = insert2(head, head, 23);    // lista: 23 -> 45 -> 56
printList(head);

// Inserimento in mezzo (prima di 56)
slist *q = head->next->next;       // q punta al nodo con dato 56
head = insert2(head, q, 8);        // inserisco 8 prima di 56
printList(head);                   // lista: 23 -> 45 -> 8 -> 56

// Inserimento in coda (q == NULL)
head = insert2(head, NULL, 12);    // inserisco 12 alla fine
printList(head);                   // lista: 23 -> 45 -> 8 -> 56 -> 12

return 0;
```

1.1.3 Cancellazione elemento corrente

La funzione `delete()` consente di eliminare un nodo specifico (`q`) da una lista semplicemente concatenata la cui testa è puntata da `p`, assumendo che `q != null`.

```
1  slist *delete(slist *p, slist *q) {
2  slist *r;
3  if(p == q) p = p->next;
4  else {
5      for(r = p; r && r->next != q; r = r->next);
6      if(r && r->next == q)
7      r->next = r->next->next;
8  }
9  free(q);
10 return p;
11 }
```



- **Tipo di ritorno**: la funzione restituisce un puntatore a `slist`, ovvero il puntatore alla testa aggiornata della lista dopo l'eliminazione;
- **Parametri**: i parametri definiti nella funzione sono (`slist *p, slist *q`), rispettivamente il **puntatore corrente** alla testa della lista e il **puntatore al nodo di riferimento** (davanti al quale rimuovere);
- **Funzionamento del codice**: per poter eliminare un nodo all'interno della lista, supponiamo di possedere una struttura dati come quella definita al capitolo 1.1.
  - *riga 2*: dichiarazione di un puntatore ausiliario `r`, che servirà per scorrere la lista.
  - *riga 3*: se `p == q`, significa che il nodo da eliminare è la testa della lista. In questo caso basta avanzare la testa al nodo successivo: `p = p->next`;
  - *riga 4-8*: se il nodo da eliminare non è quello in testa alla lista (`p!=q`), utilizzando il puntatore ausiliario, utilizzo un ciclo per scorrere tutti i nodi, **ricordando che il nodo `q` specificato nei parametri non è `null`**.
    Proprio **per questo motivo**, a differenza di quanto detto nel capitolo 1.1.2, ho **un solo caso di terminazione**, e cioé quando `r->next=q`, ovvero il nodo precedente a quello da eliminare.
    Trovato il predecessore di `q`, quest'ultimo si salta collegando `r->next` direttamente al nodo successivo a `q`, ovvero `r->next=r->next->next`;
  - *riga 9*: si libera la memoria allocata dal nodo eliminato `q`;
  - *riga 10*: restituzione del puntatore alla testa aggiornato.

Esempio di utilizzo

```
slist *head = NULL;

// Creo una lista: 23 -> 45 -> 56 -> 12
head = insert2(head, head, 56);
head = insert2(head, head, 45);
head = insert2(head, head, 23);
head = insert2(head, NULL, 12);

printList(head);  // Output: 23 -> 45 -> 56 -> 12 -> NULL

// Elimino il nodo in testa
head = delete(head, head);
printList(head);  // Output: 45 -> 56 -> 12 -> NULL

// Elimino un nodo in mezzo (quello con dato 56)
slist *q = head->next;
head = delete(head, q);
printList(head);  // Output: 45 -> 12 -> NULL

// Elimino il nodo in coda
q = head->next;
head = delete(head, q);
printList(head);  // Output: 45 -> NULL

return 0;
```

1.1.4 Creazione lista

La funzione `createlist()` crea una lista vuota e ne restituisce il puntatore radice.

```
slist *createlist(void) {
return NULL;
}
```

- **Tipo di ritorno**: la funzione restituisce un puntatore a `slist`, cioè al tipo di dato che rappresenta un nodo della lista;
- **Parametri**: il parametro è vuoto, perché non serve alcuna informazione esterna per creare una lista vuota;
- **Funzionamento del codice**: per poter creare una lista vuota, supponiamo di possedere una struttura dati come quella definita al capitolo 1.1. Il codice si occupa semplicemente di restituire `null`, che rappresenta una lista vuota (assenza di nodi).

Esempio di utilizzo

```
// Creazione di una lista vuota
slist *head = createlist();

// Inserimento di elementi
head = insert2(head, head, 10);
head = insert2(head, head, 20);
head = insert2(head, head, 30);

printList(head);  // Output: 10 -> 20 -> 30 -> NULL

return 0;
```

1.1.5 Cancellazione intera lista

La funzione `destroylist()` distrugge una lista, assumendo che `p!=null`.

```
void destroylist(slist *p){
while(p = delete(p,p)); // N.B: cancello in testa
}
```

- **Tipo di ritorno**: la funzione non restituisce alcun valore, poiché il suo scopo è esclusivamente quello di deallocare la memoria della lista passata come argomento;
- **Parametri**: Il parametro `p` rappresenta il puntatore alla testa della lista che si vuole distruggere;
- **Funzionamento del codice**: all'interno del ciclo `while`, viene eseguita ripetutamente l'istruzione `p = delete(p, p)`.
  Ad **ogni iterazione viene eliminato il primo nodo della lista**, poiché la funzione `delete()` viene chiamata passando due volte lo stesso puntatore `(p, p)`, cioè indicando che il nodo da cancellare è proprio la testa.
  La funzione `delete()` restituisce il nuovo puntatore alla testa della lista (che è ora il nodo successivo): il ciclo `while` continua fino a quando `delete()` restituisce `null`, cioè finché non rimangono più nodi da cancellare.

Esempio di utilizzo

```
// Creazione di una lista
slist *head = createlist();

// Inserimento di elementi
head = insert2(head, head, 10);
head = insert2(head, head, 20);
head = insert2(head, head, 30);

printList(head);  // Output: 10 -> 20 -> 30 -> NULL

destroylist(head); // Distruzione della lista

return 0;
```

1.1.6 Attraversamento con funzione

La funzione `traverse()` ha lo scopo di scorrere (visitare) tutti i nodi di una lista concatenata ed eseguire, su ciascuno di essi, un'operazione specificata dall'utente.

```
1 void traverse(slist *p, void (*op)(slist *)){
2 slist *q;
3 for(q = p; q; q = q->next) (*op)(q);
4 }
```

- **Tipo di ritorno**: la funzione non restituisce alcun valore. Il suo scopo è **puramente esecutivo**, applicando una funzione a ogni elemento della lista;
- **Parametri**: i parametri definiti nella funzione sono `(slist *p, void (*op)(slist *))`, rispettivamente il **puntatore alla testa** della lista che si vuole attraversare e un **puntatore a funzione**, ovvero un parametro che rappresenta una funzione da richiamare per ogni nodo visitato. Tale funzione deve avere `la stessa firma`, cioè ricevere come parametro un puntatore a `slist` e non restituire nulla (tipo `void`);
- **Funzionamento del codice**: per poter scorrere una lista, supponiamo di possedere una struttura dati come quella definita al capitolo 1.1.
  - *riga 2*: viene dichiarato un puntatore `*q`, che servirà come puntatore di scorrimento.
  - *riga 3*: inizia il ciclo che scorre l'intera lista partendo dalla testa. Ad ogni iterazione viene chiamata la **funzione passata come parametro** `(op)`, applicandola al nodo corrente: `(*op)(q)`. Il ciclo termina quando `q` diventa `null`, cioè quando è stato raggiunto l'ultimo nodo della lista.

    ```
    /* Esempio: funzione op che stampa il contenuto del nodo */
    void printelem(slist *q) {
    printf("\t-------\n\t|%5d|\n\t-------\n\t| %c |\n\t---%c---\n\t",
            q->dato,
            q->next ? '.' : 'X',
            q->next ? '|' : '-');
    if(q->next) printf(" | \n\t V\n");
    }
    ```

Esempio di utilizzo

```
// Creazione della lista
slist *head = createlist();
head = insert2(head, head, 10);
head = insert2(head, NULL, 20);
head = insert2(head, NULL, 30);

// Attraversamento della lista e stampa di ciascun elemento
printf("Contenuto della lista:\n");
traverse(head, printelem);

return 0;
```

1.1.7 Ricerca elemento con funzione