

Algoritmi e Strutture Dati

Schifano S. Fabio `schifano@fe.infn.it`

Strutture dati Dinamiche: Le liste

Una **lista** é una sequenza di elementi di un certo tipo in cui é possibile aggiungere e/o togliere elementi.

La lista a differenza del vettore é:

- ▷ una struttura **dinamica**, cioè il numero degli elementi varia durante l'esecuzione del programma
- ▷ l'accesso agli elementi della lista non avviene specificando l'indice ma si può accedere solamente ad un numero ristretto di elementi: generalmente il primo e l'ultimo.
- ▷ per accedere ad un generico elemento della lista, occorre scandire la lista sino a posizionarsi in corrispondenza dell'elemento in questione.

Esempio: uno schedario a cassetto contenente cartelle (folder) é un buon esempio pratico di lista.

Notazione:

- ▷ indichiamo con $L = a_1, \dots, a_n$ una generica lista di n elementi
- ▷ ogni elemento é caratterizzato da una posizione **pos_i** e da un valore **a_i**. La posizione **pos_i** é considerata un tipo di dato che varia a seconda della realizzazione della lista.
- ▷ per comodità denotiamo con **pos₀** la posizione che precede l'elemento a_1 , primo elemento della lista
- ▷ per comodità denotiamo con **pos_{n+1}** la posizione che segue l'elemento a_{n+1} , ultimo elemento della lista
- ▷ la lista L **vuota** é denotata dal simbolo Λ , in questo caso $n = 0$.

Le liste: Operazioni

Un tipico insieme di operazione che possono essere effettuate sulle liste comprende:

- ▷ **CREALISTA**: crea una lista vuota
- ▷ **PRIMOLISTA**: accede direttamente al primo elemento della lista
- ▷ **ULTIMOLISTA**: accede direttamente all'ultimo elemento della lista
- ▷ **SUCCLISTA**: dato un elemento della lista permette di accedere all'elemento successivo
- ▷ **PREDLISTA**: dato un elemento della lista permette di accedere all'elemento precedente
- ▷ **LISTAVUOTA**: predicato che ci dice quando una lista é vuota
- ▷ **FINELISTA**: predicato che ci dice se siamo posizionati al di fuori della lista
- ▷ **LEGGILISTA**: permette di leggere il valore di un elemento della lista
- ▷ **SCRIVILISTA**: funzione che permette di scrivere il valore di un elemento della lista
- ▷ **INSLISTA**: funzione che permette di aggiungere un nuovo elemento
- ▷ **CANCLIST**: funzione che permette di cancellare un elemento

Le liste: Specifica Sintattica

La **specifica sintattica** definisce i nomi dei tipi di dato, i nomi delle operazioni con i relativi domini e codomini, i nomi delle costanti.

Nel caso delle liste si definiscono i seguenti nomi di tipo di dato:

- ▷ il tipo di dato **lista** che stiamo definendo
- ▷ il tipo di dato **posizione**
- ▷ il tipo di dato **tipoelem**

I nomi degli operatori e dei relativi domini e codomini sono:

CREALISTA:	$() \longrightarrow \text{lista}$
PRIMOLISTA:	$\text{lista} \longrightarrow \text{posizione}$
ULTIMOLISTA:	$\text{lista} \longrightarrow \text{posizione}$
SUCCLISTA:	$(\text{posizione}, \text{lista}) \longrightarrow \text{posizione}$
PREDLISTA:	$(\text{posizione}, \text{lista}) \longrightarrow \text{posizione}$
LISTAVUOTA:	$\text{lista} \longrightarrow \text{booleano}$
FINELISTA:	$(\text{posizione}, \text{lista}) \longrightarrow \text{booleano}$
LEGGILISTA:	$(\text{posizione}, \text{lista}) \longrightarrow \text{tipoelem}$
SCRIVILISTA:	$(\text{tipoelem}, \text{posizione}, \text{lista}) \longrightarrow \text{lista}$
INSLISTA:	$(\text{tipoelem}, \text{posizione}, \text{lista}) \longrightarrow \text{lista}$
CANCLIST:	$(\text{posizione}, \text{lista}) \longrightarrow \text{lista}$

Le liste: Specifica Semantica

La **specifica semantica** associa un insieme matematico ad ogni tipo di dato, un valore ad ogni costante, una funzione ad ogni operatore.

Indichiamo con L una generica lista di lunghezza arbitraria i cui elementi sono di tipo **tipoelem**. Indichiamo con L' una lista ottenuta come risultato dell'applicazione di un qualche operatore.

▷ **CREALISTA**() = L'

Post: $L' = \Lambda$

▷ **PRIMOLISTA**(L) = p

Post: $p = \text{pos}_1$ (se $L = \Lambda$ $\text{pos}_1 = \text{pos}_{n+1}$)

▷ **ULTIMOLISTA**(L) = p

Post: $p = \text{pos}_n$ (se $L = \Lambda$ $\text{pos}_n = \text{pos}_0$)

▷ **SUCCLISTA**(p, L) = q

Pre: $L = a_1, \dots, a_n$, $p = \text{pos}_i$, $1 \leq i \leq n$

Post: $q = \text{pos}_i + 1$, **$q = \text{pos}_{n+1}$ se $p = \text{pos}_n$**

▷ **PREDLISTA**(p, L) = q

Pre: $L = a_1, \dots, a_n$, $p = \text{pos}_i$, $1 \leq i \leq n$

Post: $q = \text{pos}_i - 1$, **$q = \text{pos}_0$ se $p = \text{pos}_1$**

▷ **LISTAVUOTA**(L) = b

Post: $b = \text{True}$ se $L = \Lambda$, $b = \text{False}$ altrimenti

▷ **FINELISTA**(p, L) = b

Pre: $L = a_1, \dots, a_n, p = \text{pos}_i, 0 \leq i \leq n + 1$

Post: b = True se $p = \text{pos}_0$ or $p = \text{pos}_{n+1}$, b = False altrimenti

▷ **LEGGILISTA**(p,L) = a

Pre: $L = a_1, \dots, a_n, p = \text{pos}_i, 1 \leq i \leq n$

Post: $a = a_i$

▷ **SCRIVILISTA**(a,p,L) = L'

Pre: $L = a_1, \dots, a_n, p = \text{pos}_i, 1 \leq i \leq n$

Post: $L' = a_1, \dots, a_{i-1}, \mathbf{a}, a_{i+1}, \dots, a_n$

▷ **INSLISTA**(a,p,L) = L'

Pre: $L = a_1, \dots, a_n, p = \text{pos}_i, 1 \leq i \leq n + 1$

Post:

○ $L' = a_1, \dots, a_{i-1}, \mathbf{a}, a_i, a_{i+1}, \dots, a_n, \text{ se } 1 \leq i \leq n$

○ $L' = a_1, \dots, a_n, \mathbf{a}, \text{ se } i = n + 1$

○ $L' = \mathbf{a}, \text{ se } i = 1 \text{ e } L = \Lambda$

▷ **CANCLIST**(p, L): L'

Pre: $L = a_1, \dots, a_n, p = \text{pos}_i, 1 \leq i \leq n$

Post:

○ $L' = a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n, \text{ se } 1 \leq i \leq n$

○ $L' = \Lambda, \text{ se } i = n = 1$

Considerazioni

Dalle specifiche date ne segue che:

- ▷ per accedere ad ogni elemento della lista bisogna conoscere la sua posizione
- ▷ le uniche funzioni che restituiscono una posizione sono **PRIMOLISTA** ed **ULTIMOLISTA**
- ▷ quindi per accedere ad un elemento della lista bisogna partire dal primo elemento o dall'ultimo, e tramite le funzioni **SUCCLISTA** e **PREDLISTA** scandire la sequenza sino all'elemento desiderato
- ▷ la funzione **INSLISTA**
 - ▷ inserisce un elemento in una specifica posizione, **allungando** la lista e facendo slittare in avanti l'elemento in posizione **p** e tutti quelli successivi.
 - ▷ posiziona il nuovo elemento al posto di quello in posizione **p** oppure, se **p** denota la posizione dell'ultimo elemento, aggiunge un elemento in fondo alla sequenza.
Dopo l'applicazione della funzione, il valore di **p** denoterà la posizione del nuovo elemento appena inserito nella lista.
 - ▷ adotta una politica **pre-insertion**
- ▷ **CANCLISTA** cancella un elemento in una specifica posizione causando la **contrazione** della sequenza con conseguenze slittamento all'indietro di tutti gli elementi successivi a **p**.
Dopo l'applicazione della funzione, il valore di **p** denoterà la posizione dell'elemento successivo a quello appena cancellato.

Esempio

Data una lista di numeri interi calcolare la somma di tutti gli elementi.

```
sumlist ( lista L ) {  
  
    posizione p = PRIMOLISTA(L);  
    int sum      = 0;  
  
    while ( not FINELISTA(p, L) ) {  
        sum += LEGGILISTA(p,L);  
        p = SUCCLISTA(p, L);  
    }  
  
    return sum;  
}
```

La funzione `sumlist` utilizza una variabile `posizione p` per scandire la lista `L`.

Per ogni elemento puntato dalla `posizione p` ne legge il valore ed aggiorna il valore della variabile `sum`.

Per valutare la complessità della funzione `sumlist` ho bisogno di conoscere la complessità delle operazioni sulle liste. Supposto che la complessità di tali operazioni è $\mathcal{O}(1)$ la complessità della funzione `sumlist` è $\mathcal{O}(n)$ ove n è il numero degli elementi della lista.

Esempio

Data una lista L di numeri naturali costruire due liste contenenti rispettivamente i numeri pari e i numeri dispari.

```
pardis ( lista L ) {
    lista Pari, Dispari;

    lista Pari      = CREALISTA();           // crea lista numeri pari
    lista Dispari   = CREALISTA();           // crea lista numeri dispari
    posizione pL    = PRIMOLISTA(L);
    posizione pP    = PRIMOLISTA(Pari);
    posizione pD    = PRIMOLISTA(Dispari);

    while ( not FINELISTA(pL, L) ) {
        temp = LEGGILISTA(pL, L);
        if ( temp % 2 == 0 ) {
            INSLISTA(temp, pP, Pari);
        } else {
            INSLISTA(temp, pD, Dispari);
        }
        pL = SUCCLISTA(pL, L);
    }
}
```

N.B.: secondo la specifica data, dopo l'applicazione dell'operatore INSLISTA, le variabili posizione pP o pD , denotano la posizione del nuovo elemento inserito..

La definizione dell'algoritmo è indipendente da una specifica realizzazione della struttura dati lista.

Le Liste: Realizzazione tramite vettore

Una possibile implementazione delle liste é quella di memorizzare gli elementi della lista in un vettore. In questo caso gli elementi della sequenza sono memorizzati in un area di memoria contigua.

Questa realizzazione permette in modo **efficiente**, ovvero $\mathcal{O}(1)$, di:

- ▷ di passare da un elemento in posizione i al successivo o al precedente, basta incrementare o decrementare l'indice i
- ▷ di controllare se si stanno oltrepassando gli estremi della lista, basta controllare se l'indice oltrepassa la lunghezza del vettore
- ▷ di leggere o scrivere un elemento

Questa implementazione presenta però i seguenti **svantaggi**:

- ▷ occorre dimensionare a priori il vettore, quindi stabilire un numero massimo di elementi che la lista deve contenere:
- ▷ non implementa e non realizza l'idea di una struttura dati dinamica
- ▷ se il numero di elementi della sequenza supera la dimensione del vettore bisogna ricopiare la sequenza su un nuovo vettore più grande
- ▷ l'inserzione di un elemento ha costo $\mathcal{O}(n)$ poiché occorre scalare di una posizione in avanti tutti gli elementi dalla posizione pos_i alla posizione pos_n

- ▷ la cancellazione di un elemento ha costo $\mathcal{O}(n)$ poiché occorre scalare di una posizione indietro tutti gli elementi dalla posizione pos_{i+1} alla posizione pos_n

N.B.: nel caso di realizzazione tramite le liste, il tipo di dato **posizione** è realizzato tramite il tipo di dato indice di un array ovvero tramite una variabile di tipo **integer**.

Una realizzazione più **efficiente** delle liste consiste nel l'uso dei puntatori.

In questo caso alla contiguità degli elementi della lista non corrisponde una contiguità degli elementi in memoria.

La realizzazione tramite puntatori ci permetterà di ottenere sia una implementazione efficiente, ovvero $\mathcal{O}(1)$, delle operazioni, sia di non limitare il massimo numero di elementi della lista.

Le Liste: Realizzazione tramite puntatori

Ci sono vari modi di realizzare una lista tramite puntatori, l'idea di base é la seguente:

memorizzare una lista di n elementi in n record detti **celle**, tali che l' i -esima cella contenga il valore dell' i -esimo elemento della lista e l'indirizzo o **puntatore** della cella contenente l'elemento successivo

Se ogni cella contiene solamente il puntatore alla cella successiva l'implementazione si dice **monodirezionale**, se contiene anche il puntatore alla cella precedente l'implementazione si dice **bidirezionale**.

- ▷ la prima cella é indirizzata da una variabile di tipo puntatore
- ▷ l'ultima cella contiene il valore NULL nel campo indirizzo della cella successiva
- ▷ nel caso di implementazione bidirezionale, la prima cella contiene il valore NULL nel campo indirizzo della cella precedente

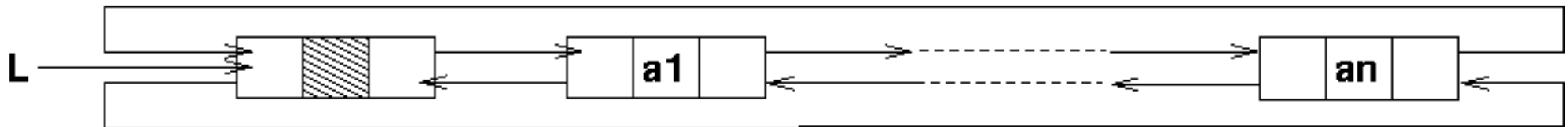
Distinguiamo 4 possibili implementazioni:

- ▷ **monodirezionale**: ogni cella indirizza solamente la cella successiva
- ▷ **bidirezionali**: ogni cella indirizza la successiva e la precedente
- ▷ **monodirezionale circolare**: l'ultima cella indirizza la prima come successiva
- ▷ **bidirezionale circolare**: la prima cella indirizza l'ultima come precedente

Le Liste: Realizzazione con sentinella

Per rendere più leggibile il codice delle funzioni **INSLISTA** e **CANCLIST** quando operano sugli elementi estremi della lista, è conveniente introdurre una ulteriore cella, che non contiene alcuna informazione valida, detta **sentinella**.

La sentinella è posizionata prima della prima cella della lista, e nel caso bidirezionale, dopo l'ultima.



Con l'introduzione della **sentinella**, le possibili differenti implementazioni sono 8 e tutte quante richiedono uno spazio di memoria $\Theta(n)$ con n uguale al numero degli elementi della lista.

Analizziamo in dettaglio la realizzazione di una lista bidirezionale con sentinella.

Realizzazione di una lista bidirezionale con sentinella

Nella lista bidirezionale con sentinella pos_i indicherà la posizione dell'elemento a_i , mentre pos_0 e pos_{n+1} denoteranno la posizione della cella sentinella.

Utilizzando le variabili puntatore del C si ottiene la seguente realizzazione.

```
typedef int tipoelem;  
  
typedef struct cella * lista;  
typedef struct cella * posizione;  
  
struct cella {  
    posizione precedente;  
    tipoelem elemento;  
    posizione successivo;  
};
```

- ▷ il tipo di dato **lista** è stato definito come un puntatore ad una variabile di tipo **cella**
- ▷ il tipo di dato **posizione** è stato realizzato come un puntatore ad una variabile di tipo **cella**
- ▷ una variabile di tipo **cella** contiene tre campi (o membri in C):
 1. il puntatore alla cella precedente,
 2. l'informazione vera e propria
 3. il puntatore alla cella successiva

```
lista CREALISTA () {  
    lista L;  
    L = malloc(sizeof (struct cella) );  
    L->successivo = L;  
    L->precedente = L;  
    return L;  
}
```

La funzione CREALISTA

- ▷ crea la cella sentinella
- ▷ per definizione, definisce se stessa come successore e predecessore
- ▷ restituisce il puntatore alla cella sentinella

Quindi, dopo la chiamata alla funzione **CREALISTA** il valore restituito che secondo la specifica identifica la lista, corrisponde nella realizzazione al puntatore alla sentinella.

```
int LISTAVUOTA (lista L) {  
    int listavuota;  
    listavuota = ((L->successivo == L) && (L->precedente == L)) ? 1 : 0;  
    return listavuota;  
}
```

Una lista é vuota se contiene solamente la lista sentinella.

Definizione di Operatori sulle Liste

```
posizione PRIMOLISTA (lista L) {  
    return L->successivo;  
}
```

```
posizione ULTIMOLISTA (lista L) {  
    return L->precedente;  
}
```

```
posizione SUCCLISTA (posizione p) {  
    return p->successivo;  
}
```

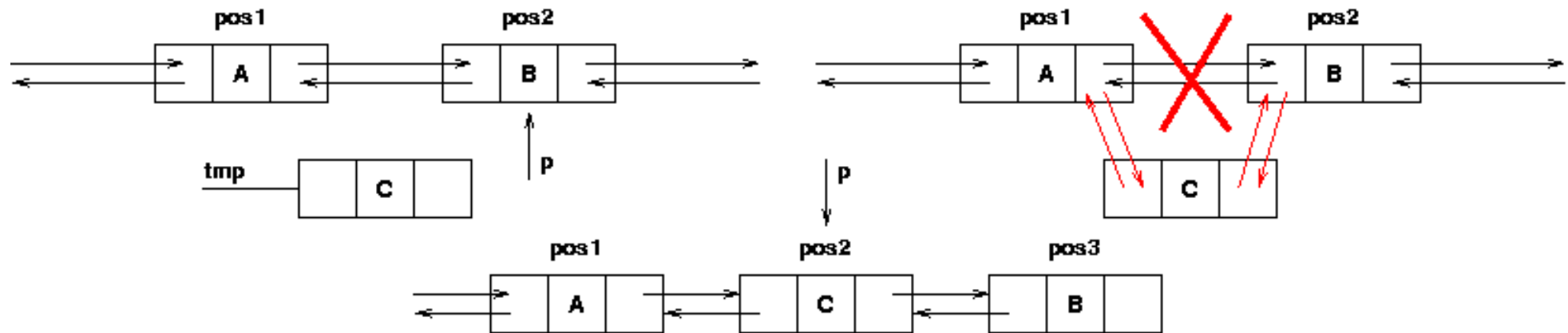
```
posizione PREDLISTA (posizione p) {  
    return p->precedente;  
}
```

```
int FINELISTA (posizione p, lista L) {  
    return ((p == L) ? 1 : 0); // True if p is the position of the sentinel  
}
```

```
int LEGGILISTA (posizione p) {  
    return p->elemento;  
}
```

```
void SCRIVILISTA (int a, posizione p) {  
    p->elemento = a;  
}
```


Operazione Inslista



```
// p passato per variabile: secondo la specifica deve essere aggiornato
// con il valore del puntatore alla nuova cella
void INSLISTA (int a, posizione * p) {
    posizione tmp;

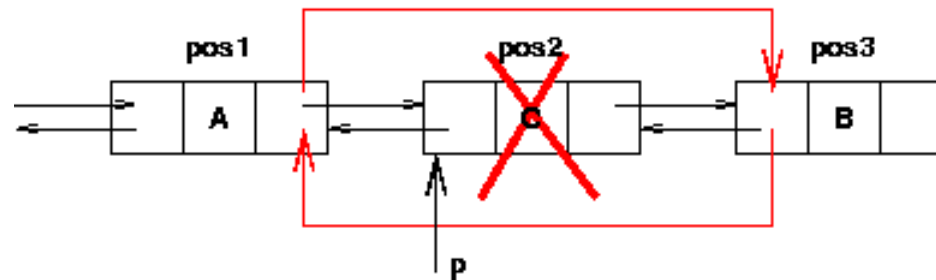
    tmp = malloc(sizeof(struct cella)); // crea una nuova cella
    tmp->elemento = a;                  // che contiene l'elemento a

    tmp->precedente = (*p)->precedente; // collega la cella tmp alla cella
    tmp->successivo = (*p);              // precedente a quella puntata da p

    ((*p)->precedente)->successivo = tmp; // collega la cella tmp alla cella
    (*p)->precedente = tmp;              // puntata da p

    (*p) = tmp; // aggiorna p
}
```

Operazione Canclista



```
void CANCLISTA (posizione * p) {  
    posizione tmp;  
  
    tmp = *p;  
  
    // collega la cella precedente a p a quella successiva a p  
    ((*p)->precedente)->successivo = (*p)->successivo;  
  
    // collega la cella successiva a p a quella precedente a p  
    ((*p)->successivo)->precedente = (*p)->precedente;  
  
    *p = (*p)->successivo;  
  
    free(tmp); // dealloca l'area di memoria puntata da tmp  
}
```

Osservazione

Nella specifica semantica di **INSLISTA** la preconditione asserisce che p sia una posizione (un **puntatore** nel caso di implementazione con i puntatori, un **indice di un array** nel caso di implementazione con i vettori) ad un elemento della lista L .

Per questo motivo la specifica prevede di passare L come parametro di input alla procedura **INSLISTA**.

Per verificare tale condizione bisognerebbe scandire tutta la lista e verificare che p corrisponda ad una posizione di un qualche elemento della lista L .

Poiché, verificare tale condizione ad ogni chiamata di **INSLISTA** é molto oneroso, per ragioni di efficienza di **realizzazione** delle liste ometteremo tale controllo, e di conseguenza non passeremo la lista come parametro di input alla procedura **INSLISTA**.

N.B.: dopo l'esecuzione della procedura **INSLISTA** il valore di p è uguale, conformemente alla specifica semantica, alla posizione dell'elemento i -esimo, ovvero dell'elemento appena inserito.

Per questo motivo nella realizzazione, il parametro p è passato per **variabile** (ovvero tramite **puntatore**), poiché deve essere modificato dalla procedura stessa con il valore della posizione dell'elemento appena inserito.

Osservazione

Le stesse considerazione appena fatte per la procedura **INSLISTA** valgono anche per la procedura **CANCLISTA**.

N.B.: dopo l'esecuzione della procedura **CANCLISTA** il valore di p è uguale alla posizione dell'elemento i -esimo, ovvero dell'elemento successivo a quello cancellato.

A causa dell'omissione di tali controlli:

è quindi responsabilità del programmatore assicurarsi un corretto uso delle specifiche delle procedure e delle funzioni operanti sulle liste.

Uso **ERRATO** delle liste

Questo é un esempio di uso **sbagliato** del tipo di dato **lista**, cioè non conforme alle specifiche che abbiamo definito:

```
int main () {
    lista      L;
    posizione p;

    L = CREALISTA();

    INSLISTA (1, &L); // ERRORE: il valore di L viene aggiornato !!!!
    INSLISTA (2, &L);
    INSLISTA (3, &L);

    p = PRIMOLISTA(L);

    while ( ! FINELISTA(p, L) ) {
        printf ("a: %d ", LEGGILISTA(p) );
        p = SUCCLISTA(p);
    }
}
```

Infatti, la definizione di INSLISTA prevede di **modificare** il valore della posizione passata come parametro di input con il valore della posizione dell'elemento inserito.

Quindi dopo la prima esecuzione di INSLISTA(1, &L) il valore di L non sarà piú quello restituito dalla funzione CREALISTA, ovvero la posizione della sentinella, ma sarà il valore della posizione del nuovo elemento inserito. Coí facendo, abbiamo perso il puntatore alla **sentinella**, compromettendo l'integritá della struttura dati.

Esempio Corretto

```
int main () {  
  
    lista      L;  
    posizione p;  
  
    L = CREALISTA();  
    p = PRIMOLISTA(L);  
  
    INSLISTA (1, &p); // il valore di p viene aggiornato  
    INSLISTA (2, &p); // con la posizione del nuovo elemento  
    INSLISTA (3, &p);  
  
    p = PRIMOLISTA(L);  
  
    while ( ! FINELISTA(p, L) ) {  
        printf ("a: %d\n", LEGGILISTA(p) );  
        p = SUCCLISTA(p);  
    }  
}
```

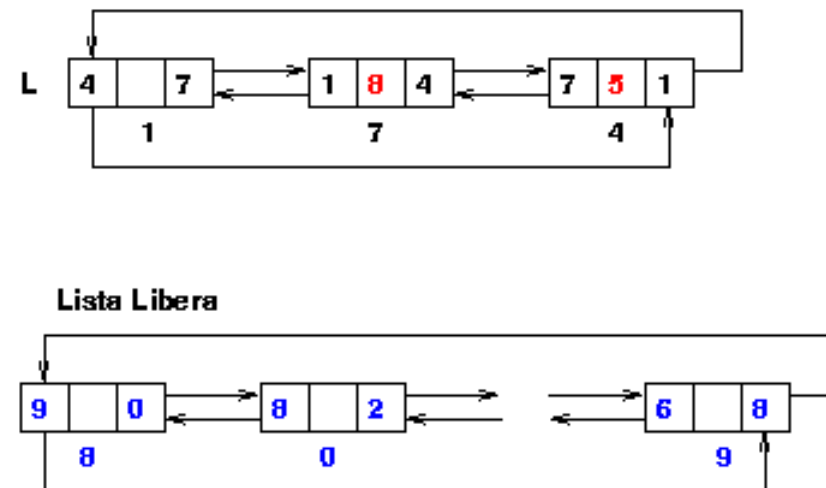
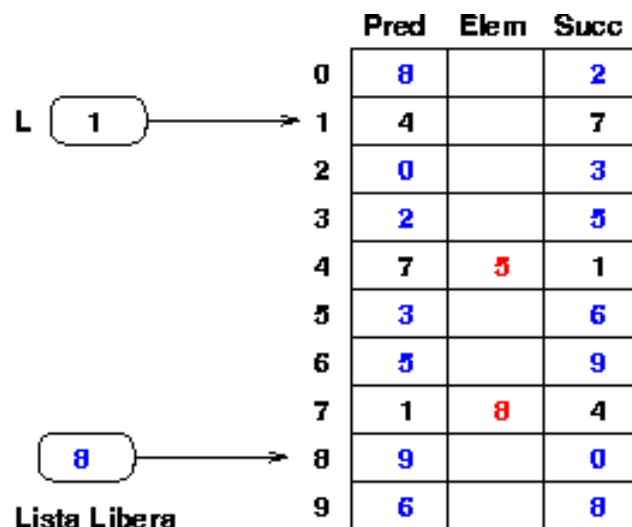
Realizzazione tramite cursori

- ▷ come si realizzano le liste in linguaggi che non hanno i puntatori ?
- ▷ come viene gestita la memoria riservata alle liste dal sistema operativo ?

É possibile simulare i puntatori con i **cursori**, cioè variabili intere il cui valore é interpretato come un indice di un vettore **SPAZIO** che simula la memoria disponibile per i puntatori.

Il vettore **SPAZIO**:

- ▷ contiene tutte le liste, ognuna individuata da un **cursore**
- ▷ contiene tutte le celle libere, organizzate anch'esse come una lista detta **lista libera**



Nei linguaggi che non hanno come dato primitivo il puntatore le routine di sistema di allocazione e deallocazione della memoria operano su un vettore SPAZIO che rappresenta la memoria disponibile per i puntatori:

- ▷ l'operazione di **malloc** stacca una cella dalla lista libera e restituisce il suo indirizzo
- ▷ l'operazione di **free** riaggancia una cella alla lista libera

La complessità delle operazioni usando la realizzazione con i cursori è la stessa di quella usando i puntatori, e l'occupazione di memoria è $\mathcal{O}(\text{MAXLUNG})$ per tutte le liste.

Esercizio: scrivere gli operatori delle liste, CREALISTA, INSLISTA, READLISTA, ..., per il caso di realizzazione tramite cursore.

Esercizio

Il **rango** di un elemento di una lista L é definito come la somma del suo valore e dei valori degli elementi che lo seguono.

Esempio:

$$L = (1, 3, 2, 4, 6, 5), \quad \text{rango}(2) = 2 + 4 + 6 + 5 = 17$$

Scrivere una procedura ricorsiva di complessità **ottima** che data una lista L modifica ogni elemento di L in modo che ogni elemento contenga il proprio rango.

Esempio:

$$L = (1, 3, 2, 4, 6, 5) \implies L = (21, 20, 17, 15, 11, 5)$$

Una limitazione **inferiore** del problema é determinata dalla lunghezza L della lista, infatti bisogna sostituire ogni elemento con il suo rango. Quindi la complessità del problema é almeno $\Omega(n)$.

La soluzione al problema può essere suggerita dalla seguente proprietà della funzione rango():

$$\begin{aligned} \text{rango}(a_i) &= a_i + \text{rango}(a_{i+1}) & \text{se } i < n \\ \text{rango}(a_i) &= a_i & \text{se } i = n \end{aligned}$$

La precedente proprietà può quindi essere utilizzata come specifica per il seguente algoritmo (pseudo-codice):

```
procedure rango ( posizione p, lista L )  
    int r;  
    begin  
        if ( p = ULTIMOLISTA(L) ) {  
            r = LEGGILISTA(p);  
        } else  
            r = LEGGILISTA(p) + rango(SUCCLISTA(p), L);  
            SCRIVILISTA(p, r);  
        }  
    end  
    return r;
```

Il calcolo del rango di tutti gli elementi della lista é innescato dalla seguente chiamata di procedura:

```
rango ( PRIMOLISTA(L), L )
```

Analisi di Complessità Temporale

Osserviamo innanzitutto che un limite inferiore alla complessità del problema è dato dal numero degli elementi della lista, quindi la complessità temporale deve essere almeno $\Omega(n)$.

Da ciò possiamo osservare che una procedura sarà **ottima** se la sua complessità temporale $T(n) \in \mathcal{O}(n)$.

Il **tempo di calcolo** $T(n)$ impiegato dall'algoritmo **rango** è espresso dalla seguente relazione di ricorrenza:

$$\begin{cases} T(n) = \alpha & \text{se } n = 1 \\ T(n) = T(n-1) + \beta & \text{se } n > 1 \end{cases}$$

Sviluppando manualmente tale relazione si ha:

$$\begin{aligned} T(n) &= T(n-1) + \beta \\ &= T(n-2) + 2\beta \\ &= T(n-3) + 3\beta \\ &\vdots \\ &= T(n - (n-1)) + (n-1)\beta \\ &= T(1) + (n-1)\beta \\ &= \alpha + (n-1)\beta \end{aligned}$$

Quindi $T(n) \in \mathcal{O}(n)$, e poiché $T(n) \in \Omega(n)$, possiamo concludere che la complessità temporale dell'algoritmo rango è **ottima**.

Esercizio TANGENTOPOLI (semiserio e un pó macabro!)

Un gruppo di politici dopo essere stati inquisiti decidono di suicidarsi in modo **buffo**: si dispongono in cerchio e si uccidono uno ogni k .

Scrivere una procedura C che data la lista L di politici, aspiranti suicidi, e dato un k stampa l'ordine di suicidio.

Esempio:

$L = A B C D E F G$

$k = 2$

$B D F A E C G$

Traccia di soluzione:

1. finché la lista non é vuota
2. sposto un cursore posizione di k celle
3. stampo il contenuto della cella puntata dal cursore
4. effettuo la rimozione della cella in questione

Esercizio TANGENTOPOLI

```
tangentopoli ( lista L, int k ) {  
    posizione p;  
    int i, j;  
  
    p = PRIMOLISTA(L);  
    j = 1;  
  
    while ( ! LISTAVUOTA(L) ) {  
  
        for ( i=0 ; i < k; i++ ) {  
  
            p = SUCCLISTA(p);  
  
            if ( FINELISTA(p, L) ) { // se il cursore finisce fuori lista (sentinella)  
                p = PRIMOLISTA(L); // lo re-inizializzo al primo elemento  
            }  
  
        }  
  
        printf("%d %03d \n", j, LEGGILISTA(p));  
  
        CANCLISTA(&p);  
  
        j++;  
    }  
}
```

Esercizio APPARTIENELISTA

Definire un operatore **APPARTIENELISTA** che prende in input un valore a ed una lista L e restituisce `True` se l'elemento a appartiene alla lista L e `False` altrimenti.

Sia a un elemento di tipo `tipoelem`, L una lista, b un elemento di tipo `boolean`, allora la specifica semantica della funzione **APPARTIENELISTA**(a, L) = b :

- ▷ Pre: $L = a_1, \dots, a_n$
- ▷ Post: $b = \text{True}$ se $\exists 1 \leq i \leq n \mid a_i = a$,
 $b = \text{False}$ altrimenti

Traccia di soluzione:

Scandisco tutta la lista con un solo cursore e confronto ogni elemento con quello di input.

Esercizio APPARTIENELISTA Codifica

```
int APPARTIENELISTA (int a, lista L) {  
    posizione p;  
    int found;    // realizzazione in C del tipo boolean  
  
    p = PRIMOLISTA(L);  
  
    found = 0;  
  
    while ( ! found && ! FINELISTA(p, L) ) {  
  
        if ( p->elemento == a ) {  
            found = 1;  
        } else {  
            p = SUCCLISTA(p);  
        }  
    }  
  
    return found;  
}
```

Il tempo di esecuzione della funzione **APPARTIENELISTA** é $\mathcal{O}(n)$. Poichè una limitazione inferiore é $\Omega(n)$ e poichè gli operatori sulle liste utilizzati hanno ordine di complessità $\mathcal{O}(1)$, si conclude che la realizzazione data è **ottima**.