Operazioni di inserimento e rimozione in liste collegate

Altre implementazioni per le liste

Operatori di inserimento/rimozione: ricordiamo la specifica …

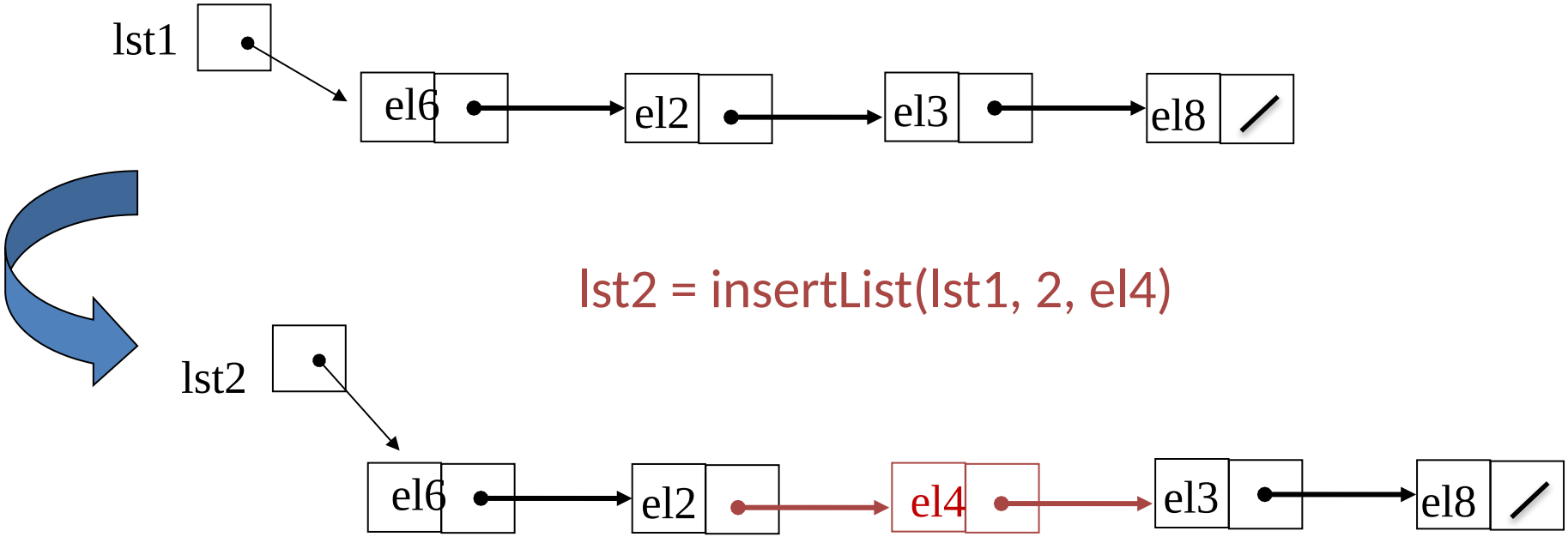
- **Specifica sintattica**
  - insertList(list, integer, item) → list
  - removeList(list, integer) → list

- **Specifica semantica**
  - insertList(l, p, e) → l'
    - Pre: sizeList(l) ≥ p AND p ≥ 0 // assumiamo 0 prima posizione
    - Post: l' si ottiene da l inserendo e in posizione p
  - removeList(l, p) → l'
    - Pre: sizeList(l) > p AND p ≥ 0 // assumiamo 0 prima posizione
    - Post: l' si ottiene da l eliminando l'elemento in posizione p

NB: insert(l, 0, val) = consList(val, l); remove(l, 0) = tailList(l)

Realizzazione di operatori di inserimento/rimozione

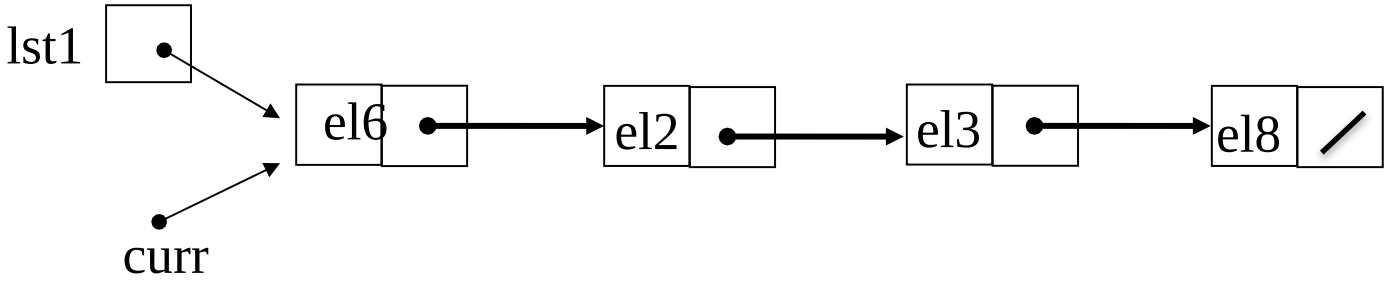
- Realizzare operatori di inserimento/rimozione utilizzando gli operatori di base è un po' complicato
- Ad esempio per l'inserimento:

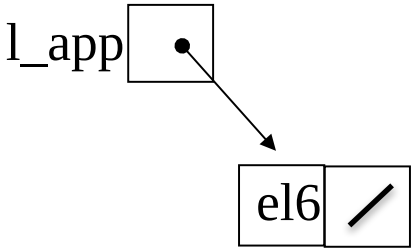


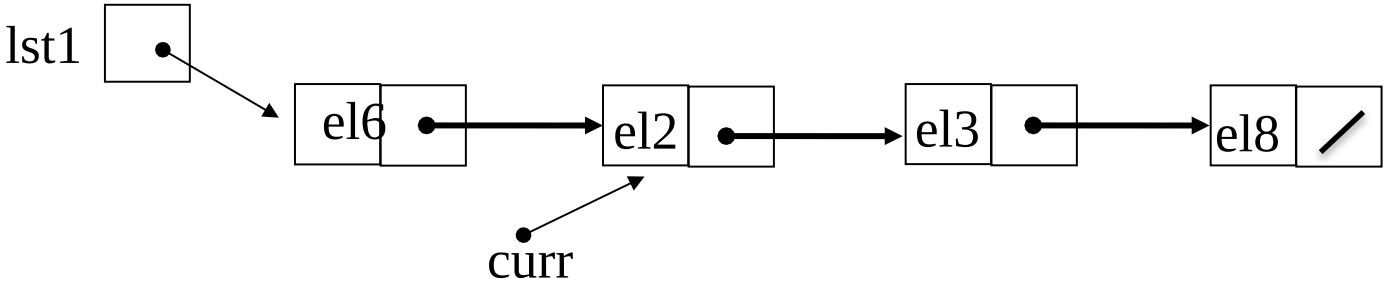
Realizzazione di operatori di inserimento/rimozione

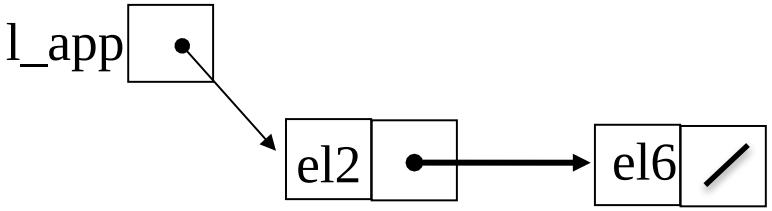
- Una prima idea per implementare insertList(l, pos, val) potrebbe essere la seguente:
  - Scorriamo la lista l e inseriamo (usando consList) in una lista di appoggio tutti gli elementi della lista l di input che precedono la posizione pos in cui inserire l'elemento val
  - Inseriamo l'elemento val nella lista di appoggio
  - Inseriamo i restanti elementi di l nella lista di appoggio
  - Restituiamo la reverse della lista di appoggio
- Farlo come esercizio ..

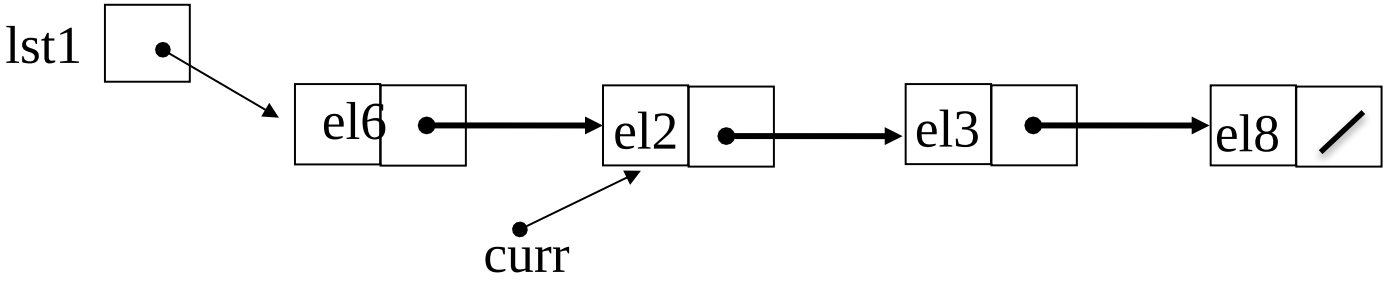
Si può fare di meglio ?


lst1
el6
el2
el3
el8
curr

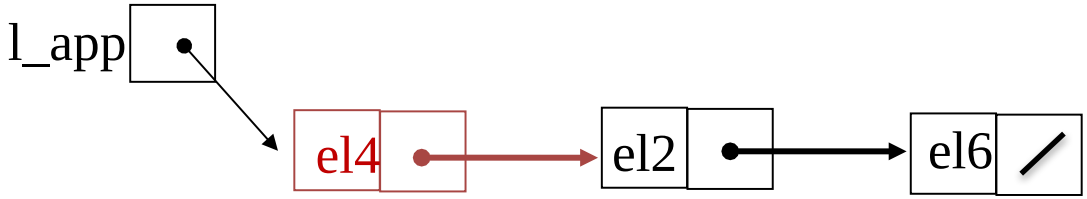

l_app
el6

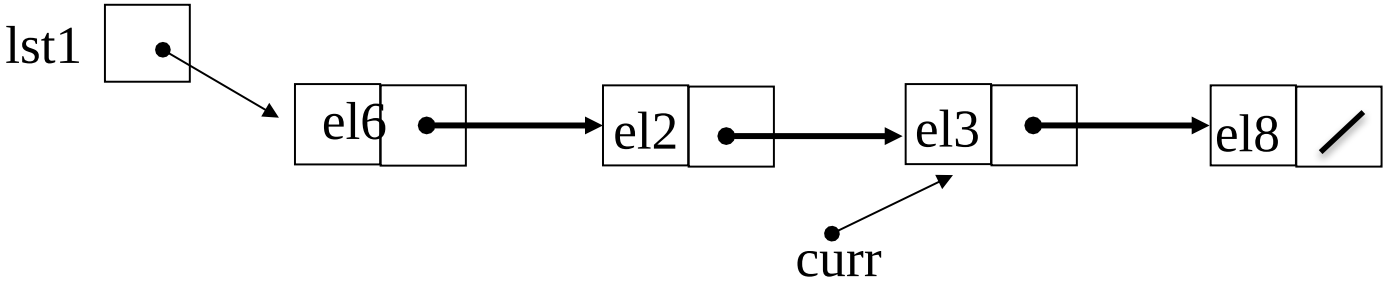

lst1
el6
el2
el3
el8
curr

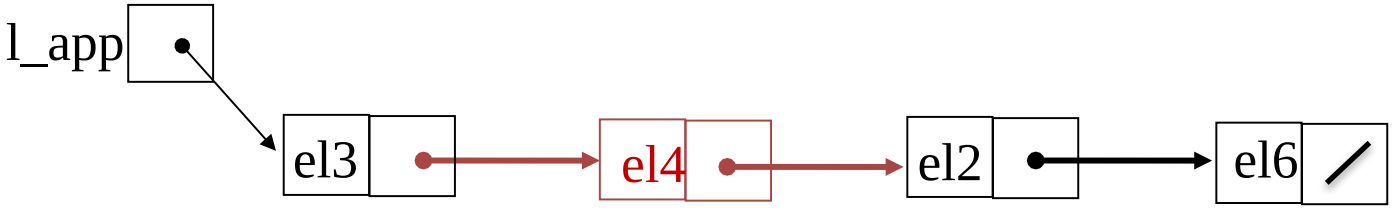

l_app
el2
el6

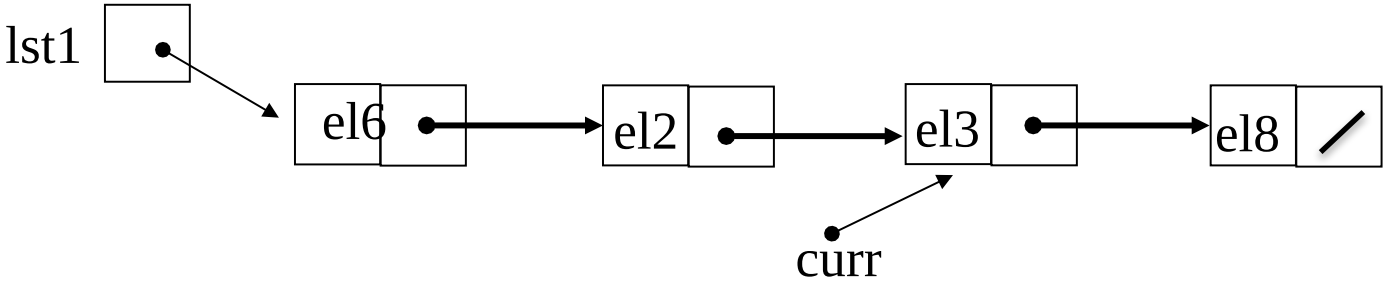


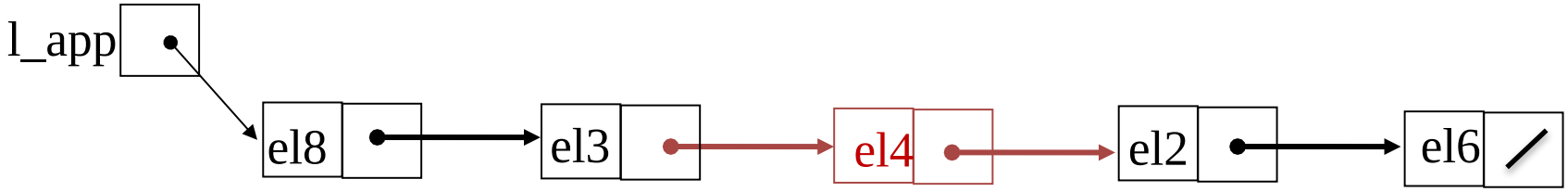
lst2 = insert(lst1, 2, el4)

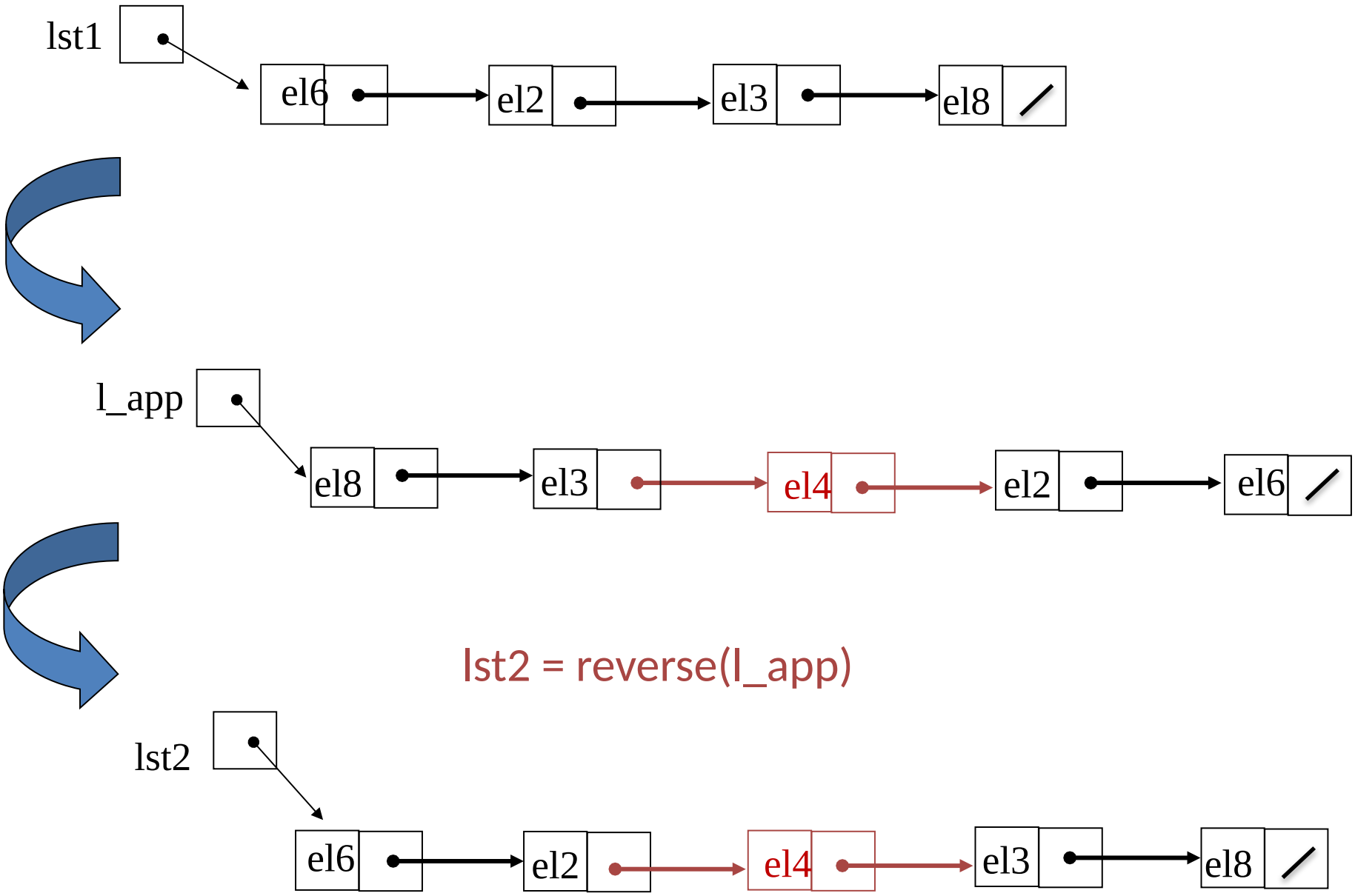



lst1
el6
el2
el3
el8
curr


l_app
el3
el4
el2
el6


lst1
el6
el2
el3
el8
curr

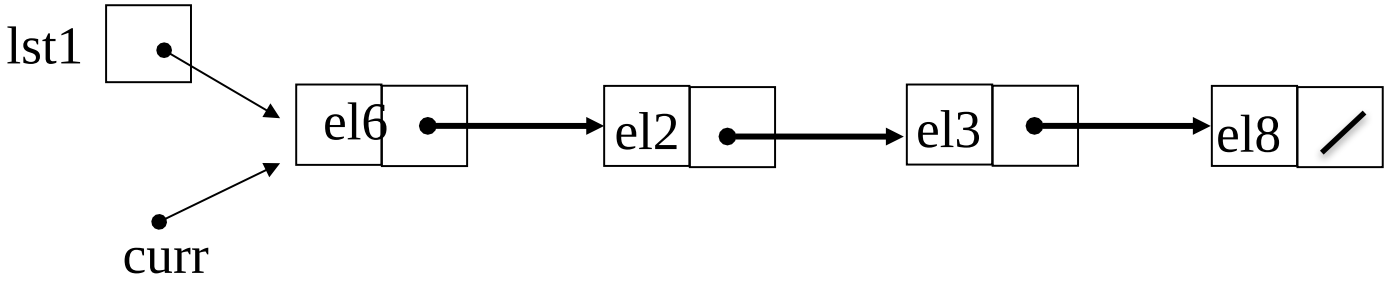

l_app
el8
el3
el4
el2
el6

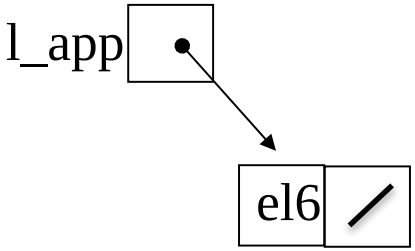

lst1
el6
el2
el3
el8
l_app
el8
el3
el4
el2
el6
lst2 = reverse(l_app)
lst2
el6
el2
el4
el3
el8

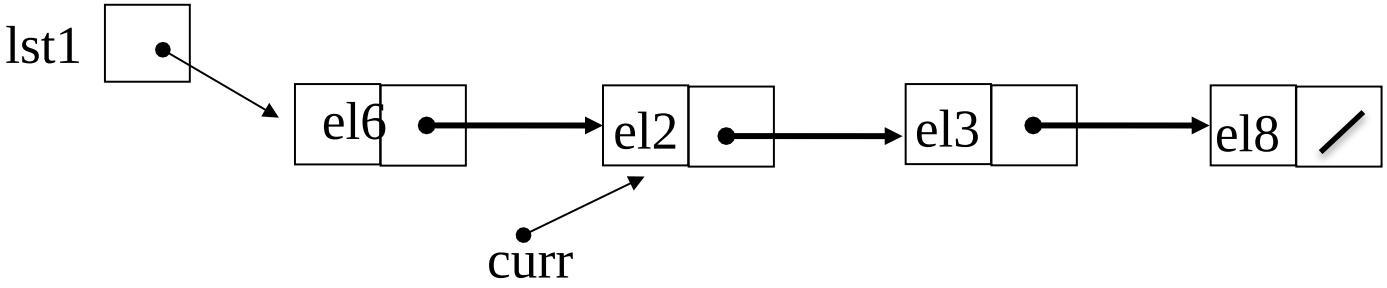
Realizzazione di operatori di inserimento/rimozione

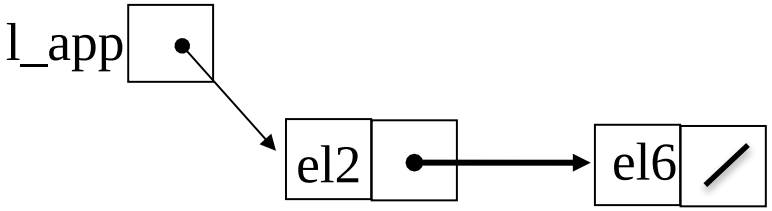
- Un'idea migliore è la seguente:
  - Scorriamo la lista l e inseriamo (usando consList) in una lista di appoggio tutti gli elementi della lista l di input che precedono la posizione pos in cui inserire l'elemento val (come prima)
  - Inseriamo val in testa a ciò che rimane della lista di input l
  - La lista di appoggio avrà gli elementi di testa della lista l in ordine inverso … scorriamo quindi la lista di appoggio e inseriamo gli elementi nella lista di output
  - Restituiamo la lista di output

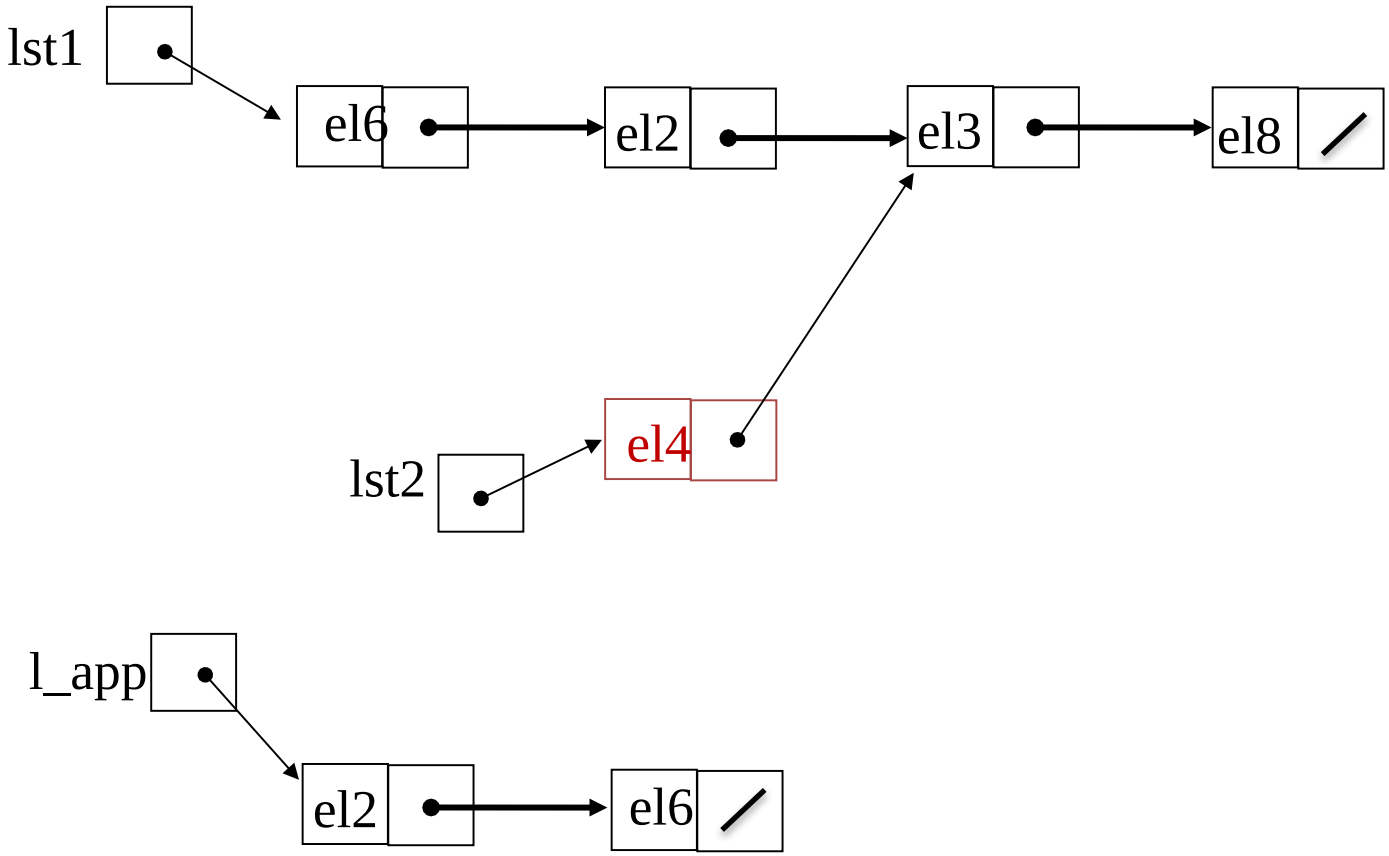
- Vediamo un esempio e poi il codice …

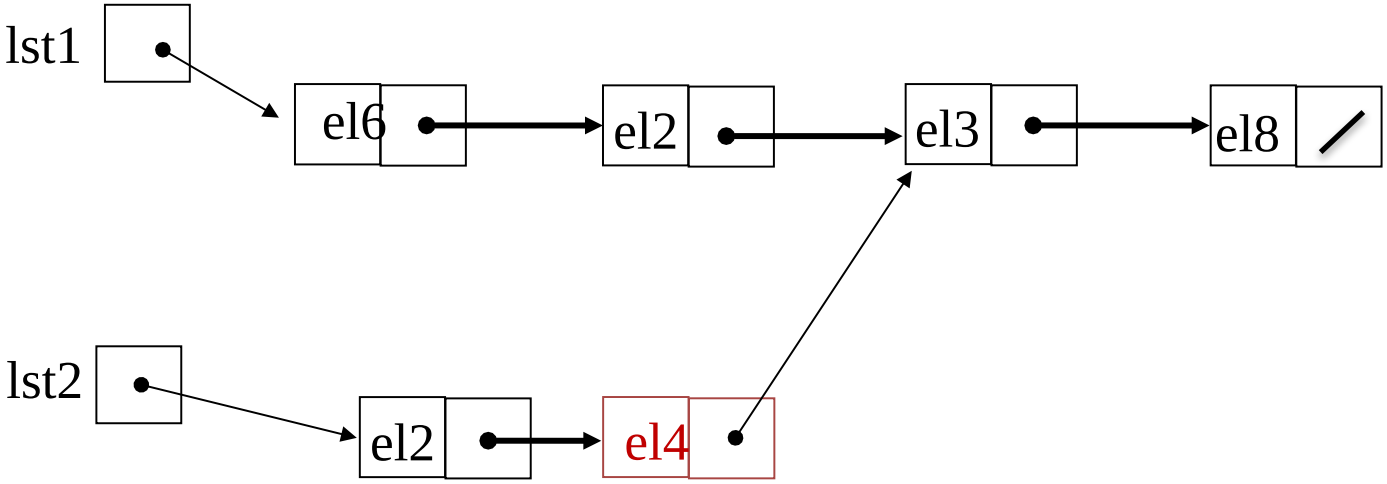

lst1
el6
el2
el3
el8
curr

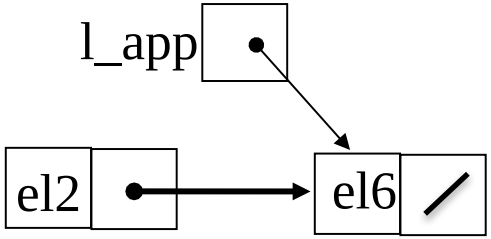

l_app
el6

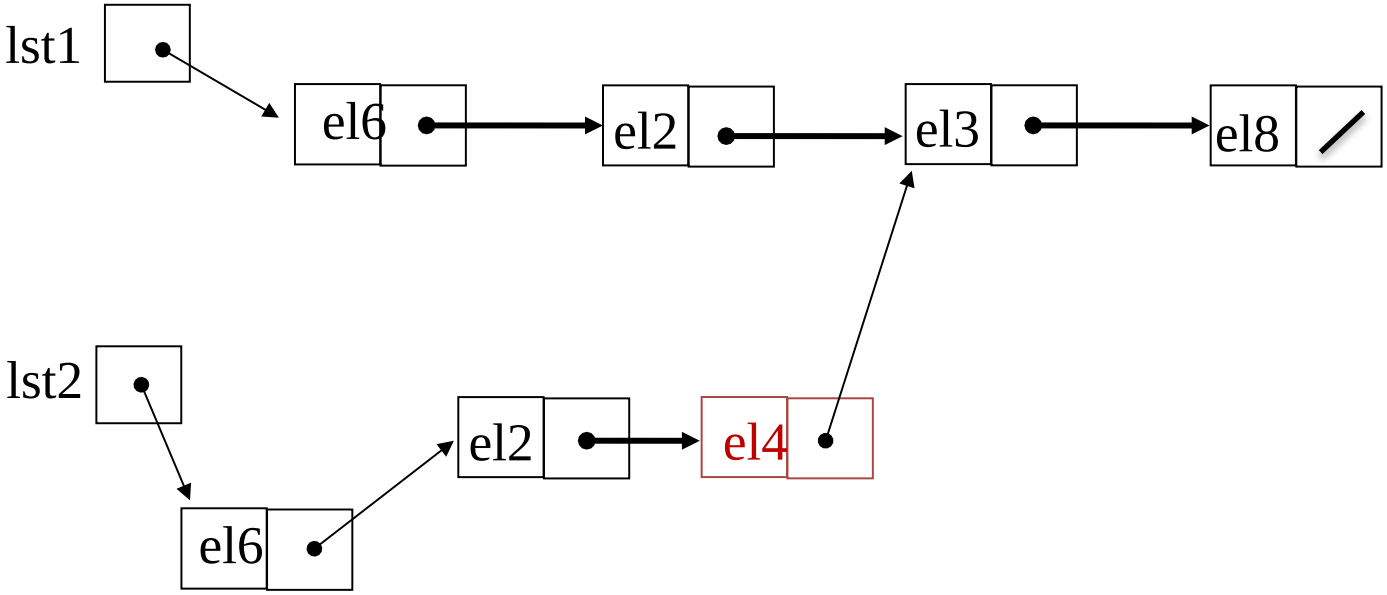

lst1
el6
el2
el3
el8
curr


l_app
el2
el6

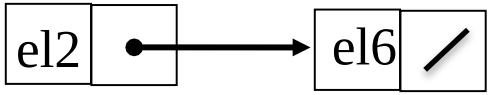

lst1
el6
el2
el3
el8
el4
lst2
l_app
el2
el6


lst1
el6
el2
el3
el8
lst2
el2
el4


l_app
el2
el6

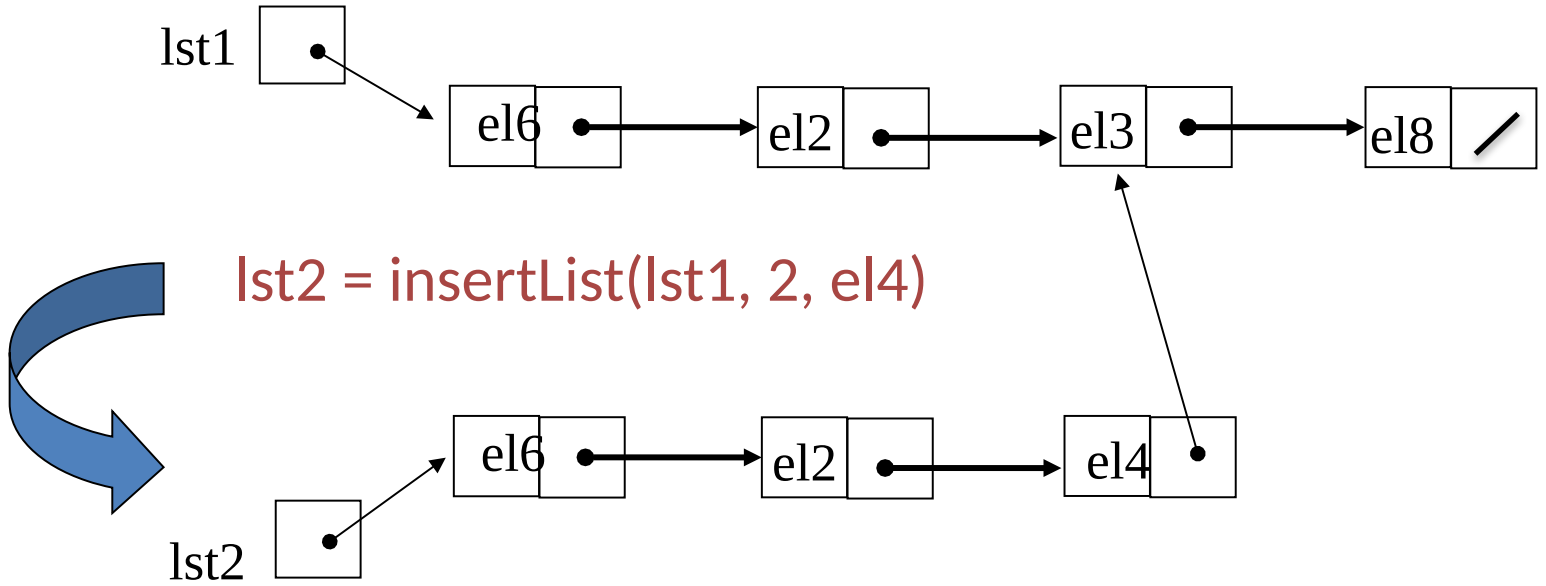

lst1
el6
el2
el3
el8
lst2
el2
el4
el6


l_app


el2
el6

Effetto dell'operatore insertList

- Come risultato della insertList, viene duplicata la parte della lista di input tra le posizioni 0 e pos-1, prima di inserire il nuovo elemento nella lista di output
- La lista di input e la lista di output condivideranno la parte di lista dopo l'elemento inserito …



Implementazione di insertList …

Realizziamo la funzione **insertList(l, pos, val)** che, dati una lista **l**, una posizione **pos** e un elemento **val**, restituisce una nuova lista ottenuta dalla lista di input inserendo l'elemento nella posizione specificata

```
list insertList (list l, int pos, item val)
{
      item x;
      int i =0;
      list ltmp = newList();      // lista di appoggio
      list lo = newList();        // lista di output

/* scandiamo la lista di input fino alla posizione pos e
memorizziamo i primi pos-1 elementi in una lista di appoggio ltmp */

      while (i < pos && !emptyList(l)) {
          x = getFirst(l);
          ltmp = consList(x, ltmp);
          l = tailList(l);
          i++;
        }
…     // continua su prossima slide
```

… Implementazione di insertList

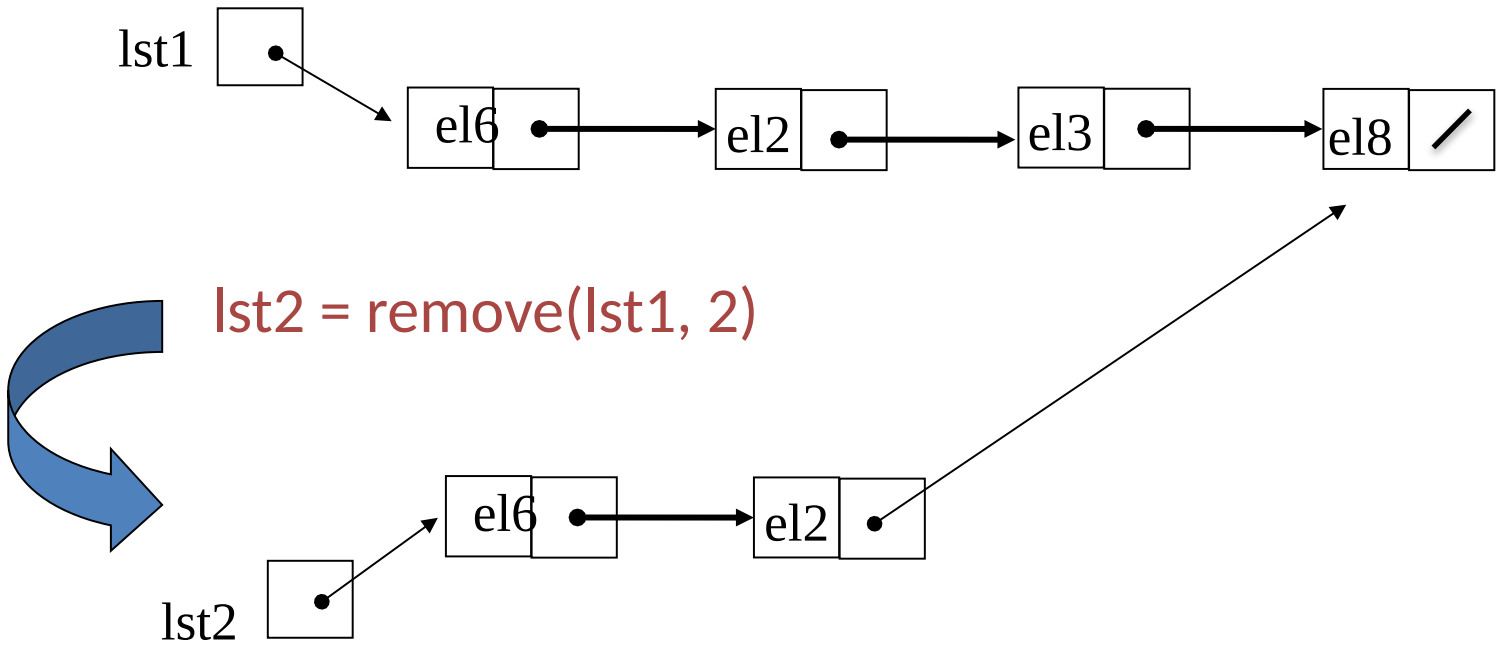
```
…
// controllo della precondizione: la lista di input ha almeno pos elementi
      if(i==pos) {
         lo = consList(val, l);

// inseriamo in lo gli elementi contenuti nella lista d'appoggio ltemp

         while(!emptyList(ltmp)) {
              x = getFirst(ltmp);
              lo = consList(x, lo);
              ltmp = tailList(ltmp);
           }
         }
       return(lo);
}
```

NB: nella lista ltmp gli elementi sono stati inseriti in ordine inverso rispetto a come erano inseriti nella lista l

Di conseguenza, quando vengono inseriti nella lista lo si troveranno nello stesso ordine in cui si trovavano nella lista l

Operatore removeList

- Stesso discorso per l'operatore removeList(list, pos) che rimuove l'elemento in posizione pos …

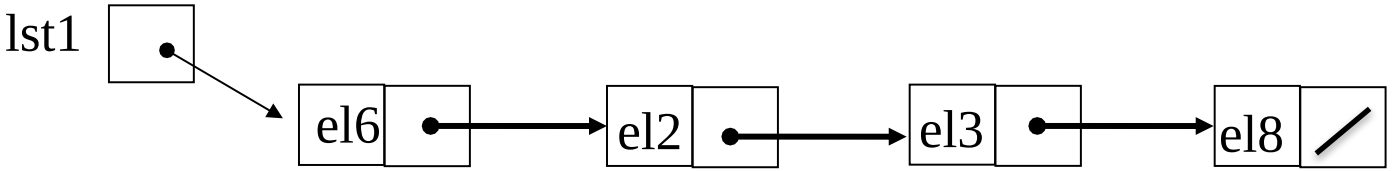


Farlo come esercizio …

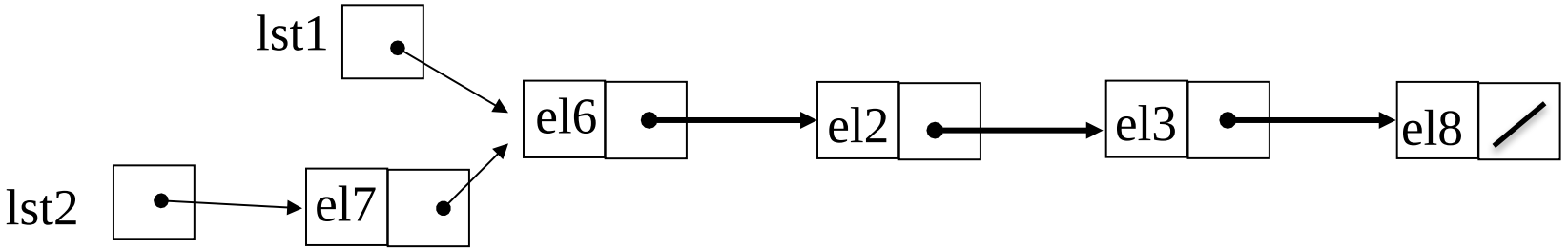
Alcune note sull'implementazione

- Se per realizzare nuovi operatori usiamo solo gli operatori dell'insieme di base, l'implementazione del tipo lista così definita consente operazioni di assegnamento del tipo:
  - lst2 = lst1;
  - dove lst1 e lst2 sono di tipo list
- Più in generale consente a liste diverse di condividere parte della struttura, perché le operazioni su una lista non producono modifiche (interferenze) sull'altra lista
- Ad esempio, vediamo cosa succede quando in un programma scriviamo le seguenti istruzioni:
  - lst2 = consList(e, lst1);
  - lst3 = tailList(lst1);

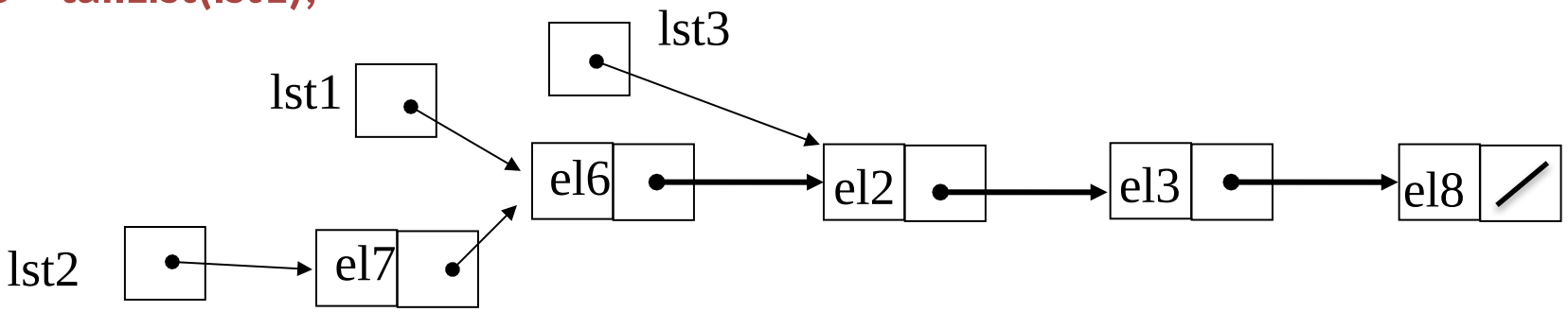
Implementazione libera da interferenze



lst2 = consList(el7, lst1);



lst3 = tailList(lst1);

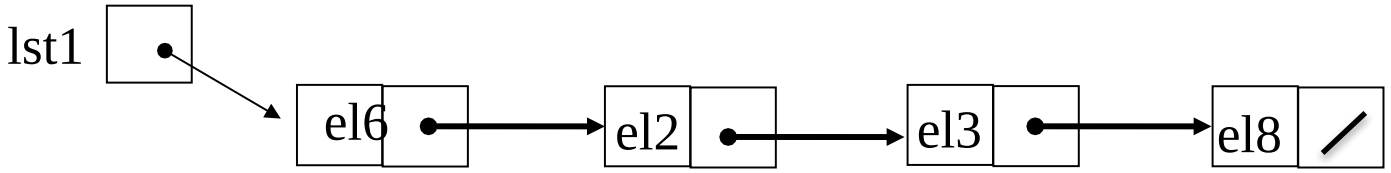


Problemi dell'implementazione

- Sebbene sia una soluzione elegante, questo tipo di implementazione e l'utilizzo degli operatori di base per estendere l'ADT lista con altri operatori non è pratica e in C presenta dei problemi
  - Non deallocando mai la memoria dinamica, i nodi che non vengono raggiunti da nessuna variabile automatica di tipo puntatore, costituiscono "garbage"
  - Si pensi ad esempio alla memoria allocata per la lista di appoggio negli operatori di inserzione e rimozione …
  - Non esistendo in C (a differenza di Java) un meccanismo di "garbage collection", capace di individuare e liberare le aree di memoria dinamica non utilizzate, in programmi reali tale implementazione potrebbe provocare un esaurimento della memoria dinamica disponibile ("heap overflow")

Esecuzione di insertList

lst2 = insert(lst1, 2, el4)

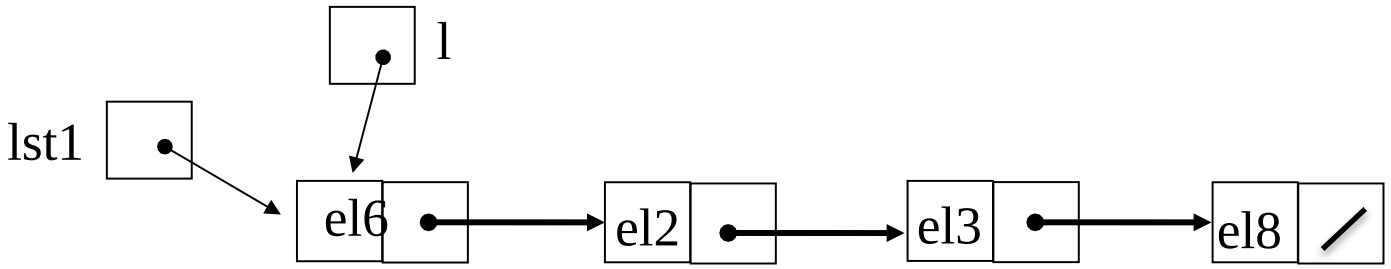


Questo è lo stato prima dell'esecuzione della funzione …

Vediamo cosa succede durante l'esecuzione …

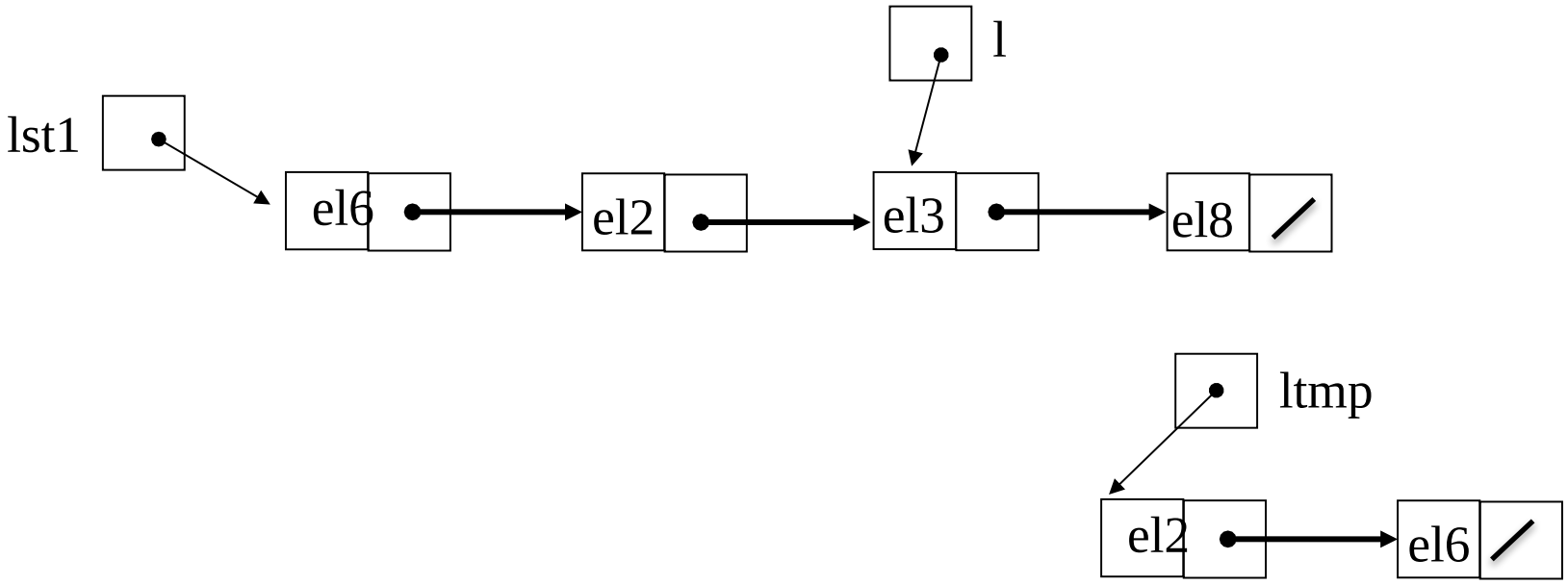
Esecuzione di insertList

lst2 = insert(lst1, 2, el4)



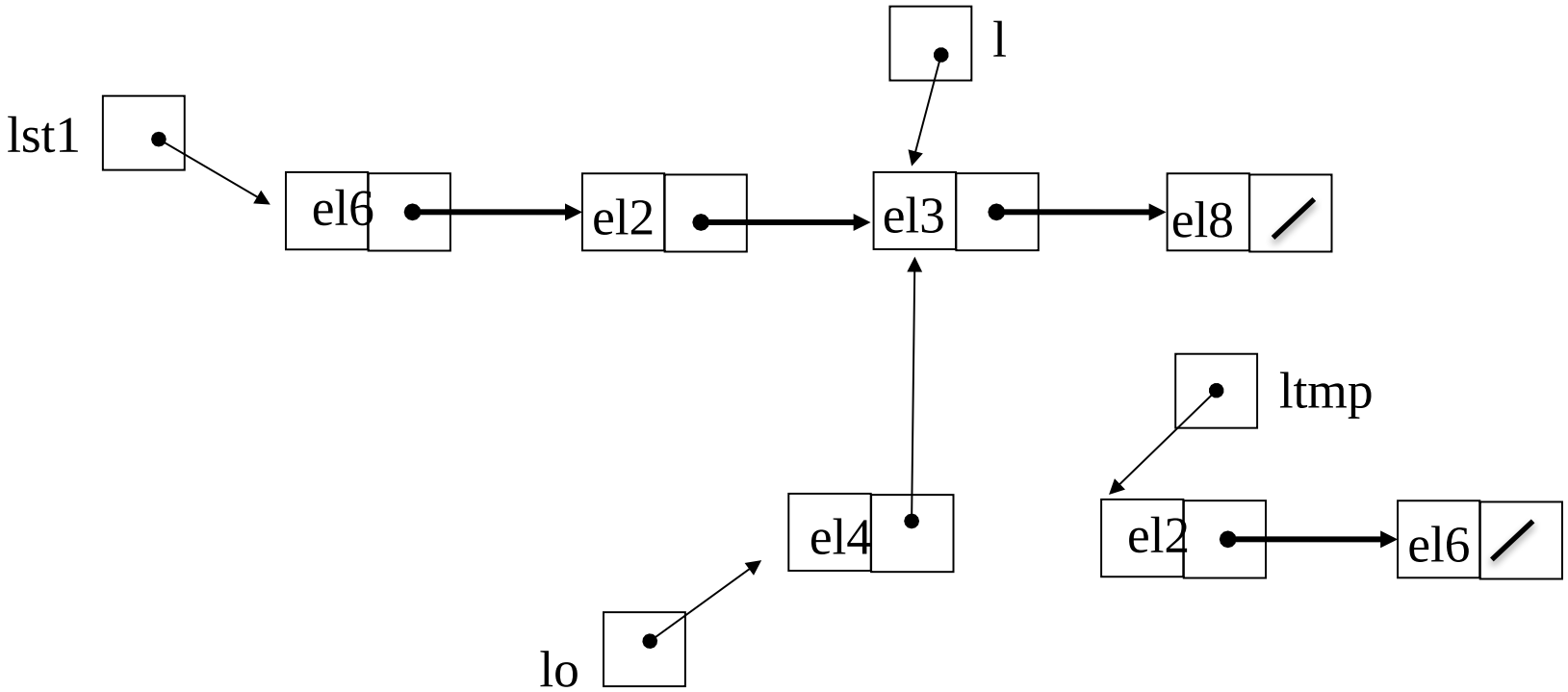
Esecuzione di insertList

lst2 = insert(lst1, 2, el4)



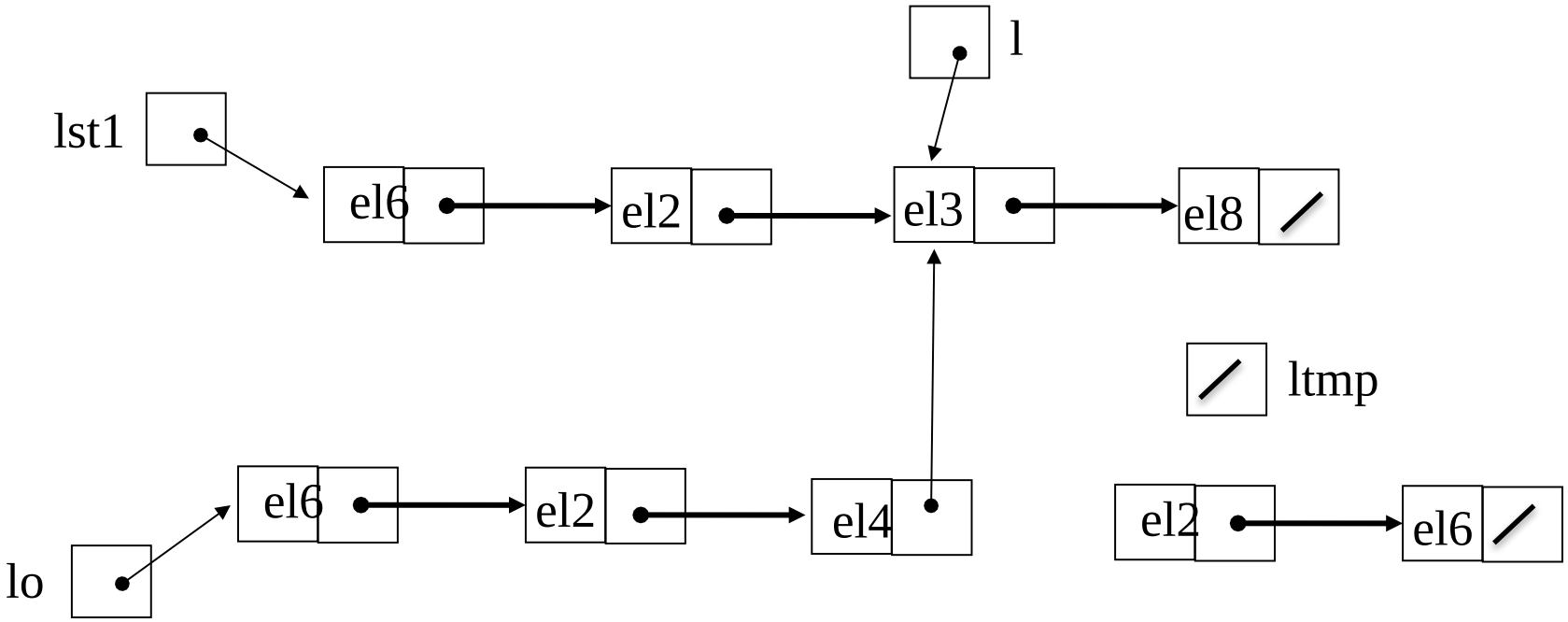
Esecuzione di insertList

lst2 = insert(lst1, 2, el4)



Esecuzione di insertList

lst2 = insert(lst1, 2, el4)

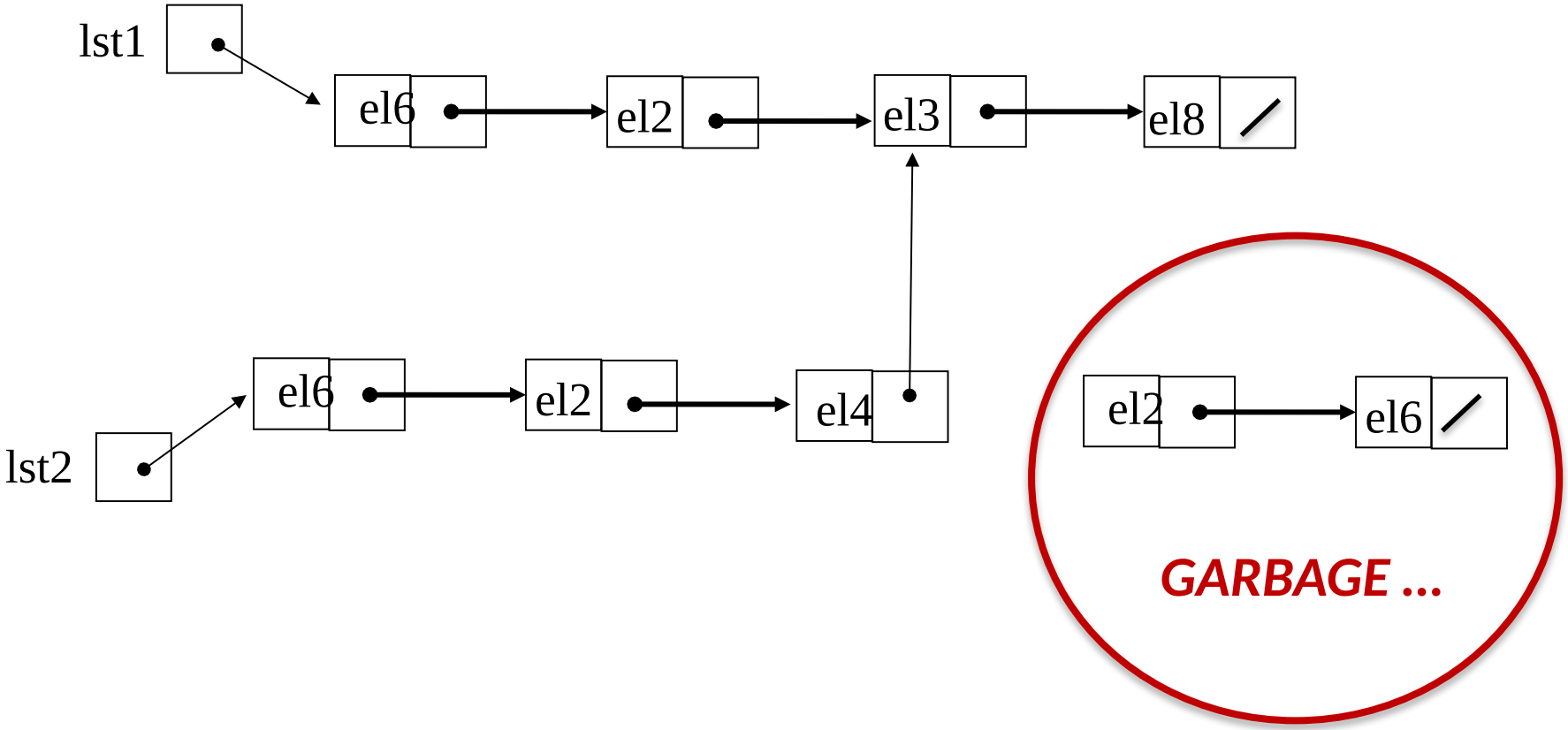


Questo è lo stato alla fine dell'esecuzione della funzione …

Vediamo cosa succede all'uscita della funzione …

Esecuzione di insertList

lst2 = insert(lst1, 2, el4)



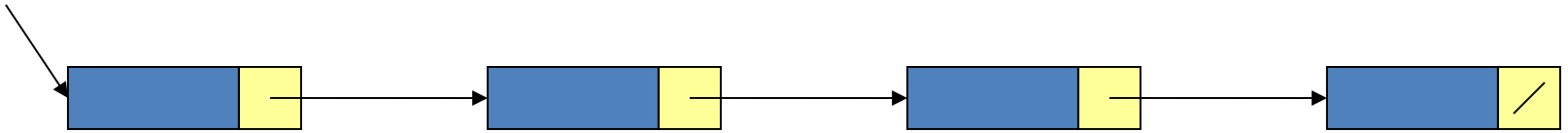
Una diversa soluzione

- Se nell'implementazione dei nuovi operatori di inserimento, invece di usare gli operatori di base si scorre direttamente la struttura a puntatori, si può inserire un elemento in una lista concatenata come successore di uno qualunque dei record già presenti …

- Stesso discorso vale per la rimozione …

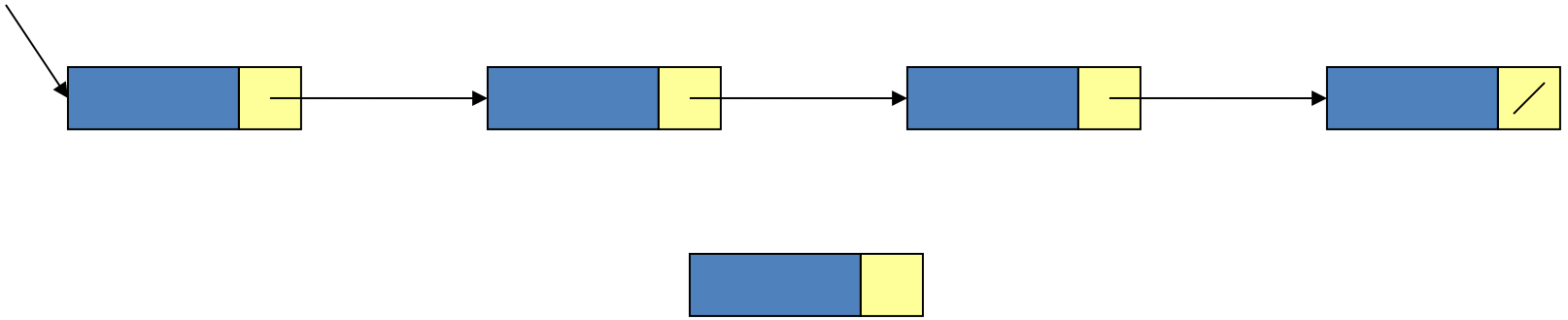
Inserire un elemento in una lista concatenata

- Si può inserire un elemento in una lista concatenata in qualunque posizione oltre alla prima, come successore di uno dei record già presenti.



Inserire un elemento in una lista concatenata

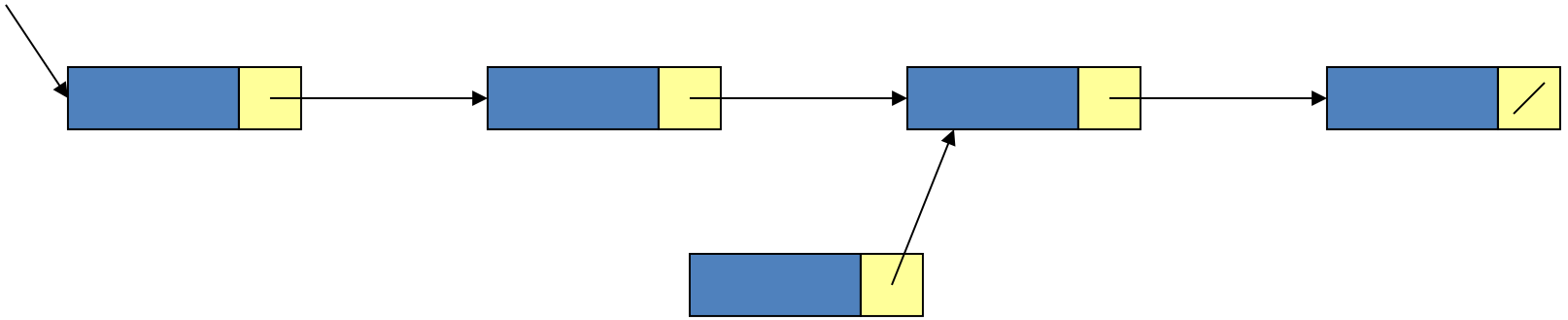
- Si può inserire un elemento in una lista concatenata in qualunque posizione oltre alla prima, come successore di uno dei record già presenti.



1. per prima cosa si crea il collegamento con il record successivo

Inserire un elemento in una lista concatenata

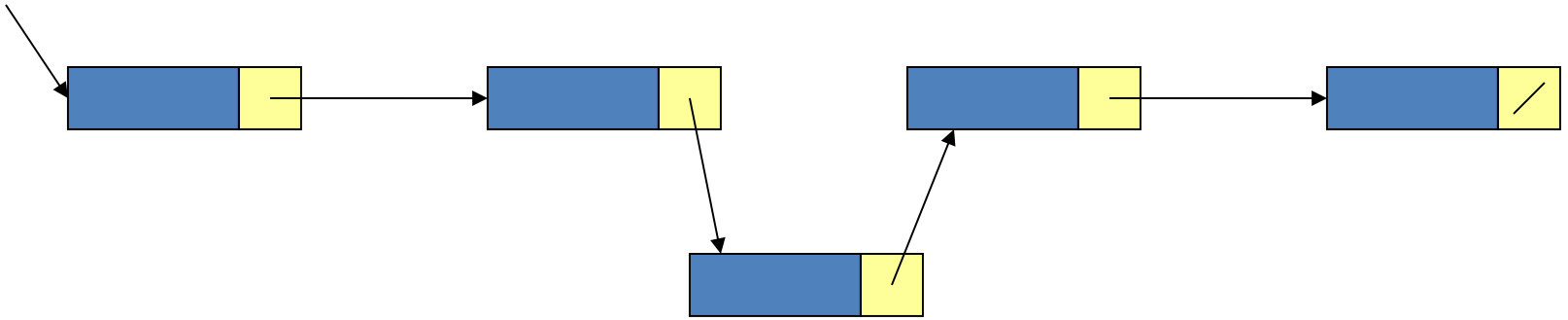
- Si può inserire un elemento in una lista concatenata in qualunque posizione oltre alla prima, come successore di uno dei record già presenti.



2. poi si collega il record precedente

Inserire un elemento in una lista concatenata

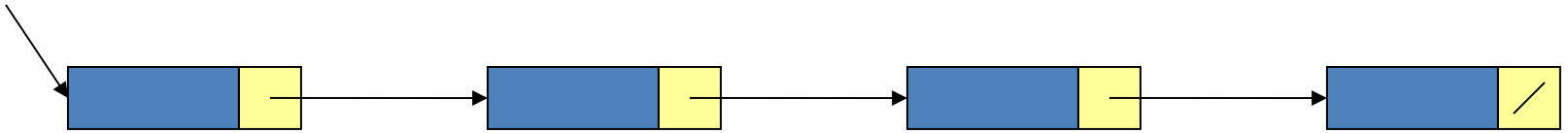
- Si può inserire un elemento in una lista concatenata in qualunque posizione oltre alla prima, come successore di uno dei record già presenti.



3. fatto!

Eliminazione di un elemento in una lista concatenata

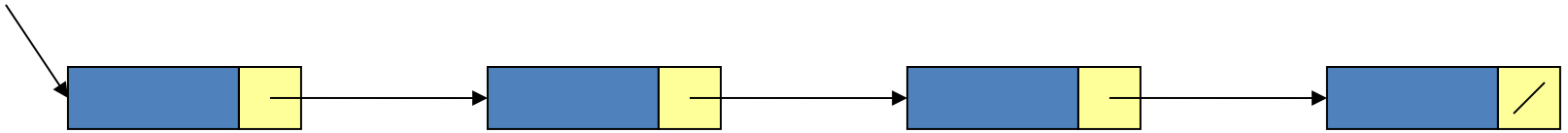
- Con il procedimento inverso, si può eliminare da una lista concatenata l'elemento successore di uno qualunque dei record già presenti.



- Supponiamo si voglia eliminare il terzo elemento

Eliminazione di un elemento in una lista concatenata

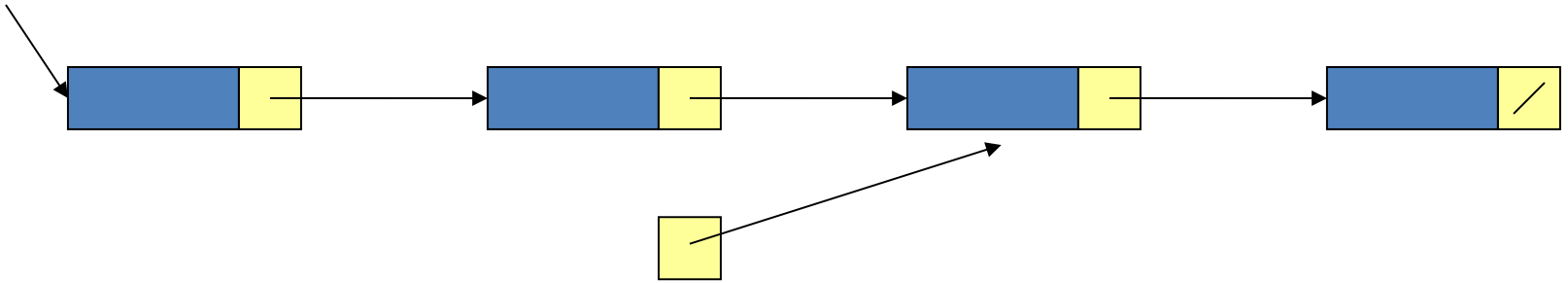
- Con il procedimento inverso, si può eliminare da una lista concatenata l'elemento successore di uno qualunque dei record già presenti.



1. per prima cosa si salva il collegamento al record successivo in una variabile temporanea

Eliminazione di un elemento in una lista concatenata

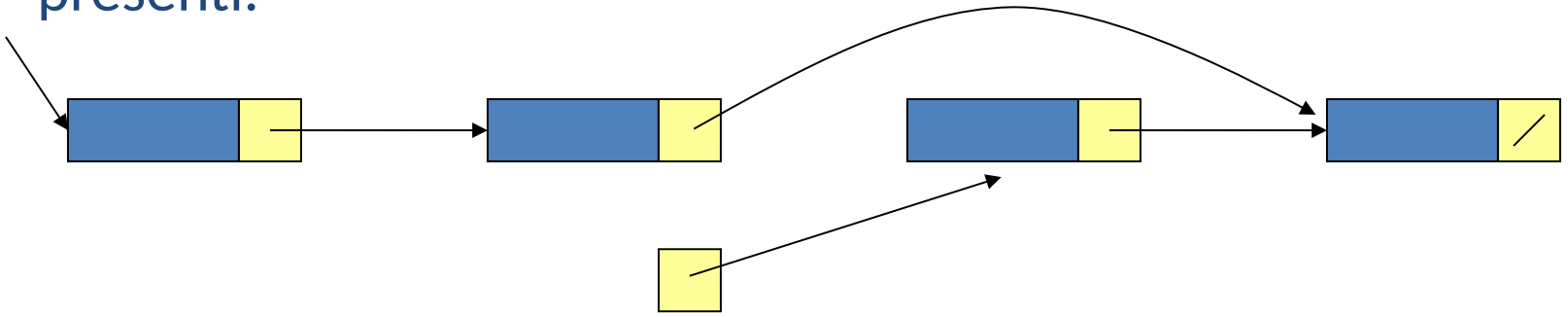
- Con il procedimento inverso, si può eliminare da una lista concatenata l'elemento successore di uno qualunque dei record già presenti.



2. Poi si collega il record precedente al record successivo

Eliminazione di un elemento in una lista concatenata

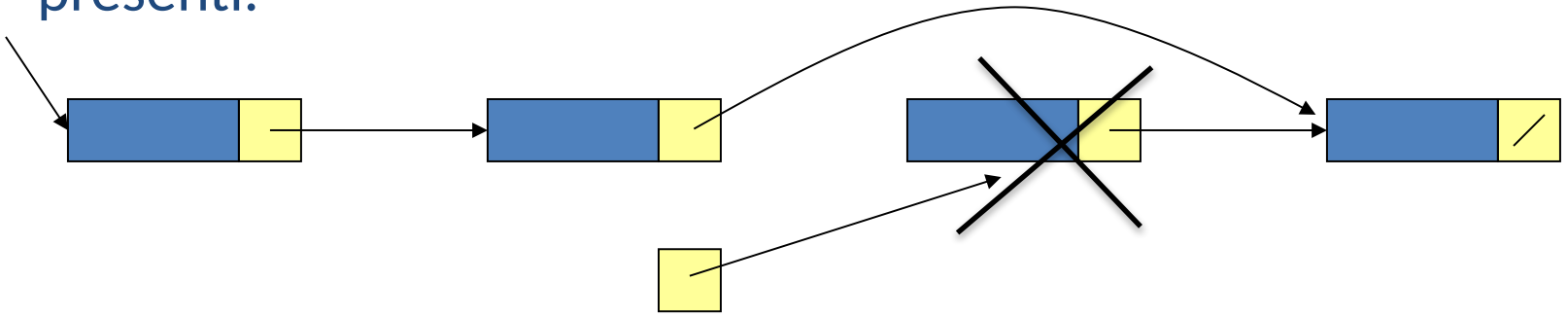
- Con il procedimento inverso, si può eliminare da una lista concatenata l'elemento successore di uno qualunque dei record già presenti.



3. Infine si elimina il record

Eliminazione di un elemento in una lista concatenata

- Con il procedimento inverso, si può eliminare da una lista concatenata l'elemento successore di uno qualunque dei record già presenti.



3. Fatto

Con questa soluzione deallochiamo la memoria …

Rivediamo l'operazione insertList …

```
list insertList (list l, int pos, item val)
{
    int i = 0;
    list l1, prec = l;

    if(pos == 0)          // inserimento in posizione 0
        return consList(val, l);

/* se non dobbiamo inserire in posizione 0 scorriamo la lista
fino alla posizione precedente a quella in cui inserire il nuovo nodo */

    while (i < pos-1 && prec!= NULL) {
            prec = prec->next;
            i++;
    }

      if(prec == NULL)  // la lista di input ha meno di pos elementi
            return l;
…
```

… Implementazione di insertList

```
…

/* se prec != NULL prec punta all'elemento di posizione pos-1 ed
è possibile inserire il nuovo elemento in posizione pos */

        l1 = consList(val, prec->next);
        prec->next = l1;

          return l;  /* se abbiamo inserito in posizione > 0 l punta ancora
                        al primo elemento */
}
```

Rivediamo l'operazione removeList …

```
list removeList (list l, int pos)
{
     list l1, prec;    // puntatore al nodo da eliminare
     int i;

     if(pos == 0 && l != NULL) {      // eliminazione in posizione 0
          l1 = l;
          l = tailList(l);
          free(l1);
       }
      else {       // se non dobbiamo cancellare in posizione 0, scorriamo
                   // la lista fino alla posizione precedente a quella del
                   // nodo da eliminare
          i = 0;
          prec = l;
          while (i < pos-1 && prec!= NULL) {
               prec = prec->next;
               i++;
          }
…
```

... Implementazione di removeList

```
...
/* alla fine del ciclo, se prec != NULL allora prec->next punta al nodo da
   eliminare. Se prec->next != NULL allora il nodo si può eliminare */

        if(prec != NULL && prec->next != NULL) { // short-circuit evaluation
            l1 = prec->next;
            prec->next = l1->next;
            free(l1);
          }
      }

    return l;
}
```

Attenti alle interferenze

- Questa implementazione è più efficiente, ma non è libera da interferenza …
- Operazioni del tipo

  lst1 = lst2;

  lst2 = insertList(lst1, pos, val);

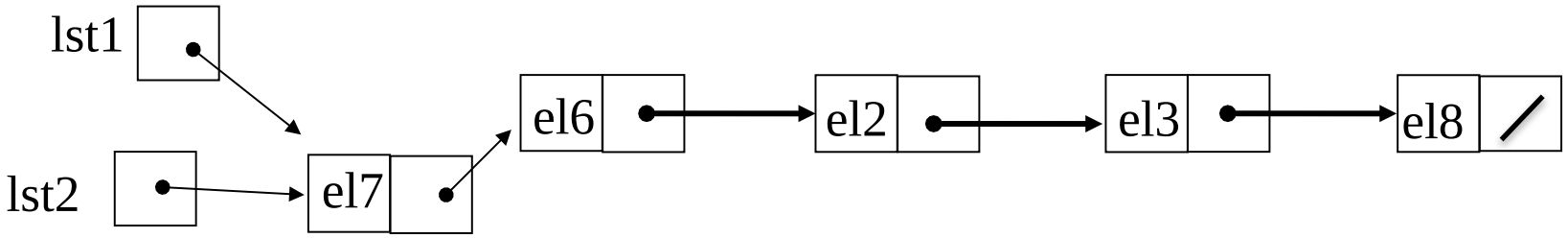
  lst2 = removeList(lst1, pos);

  non sono più ammissibili …

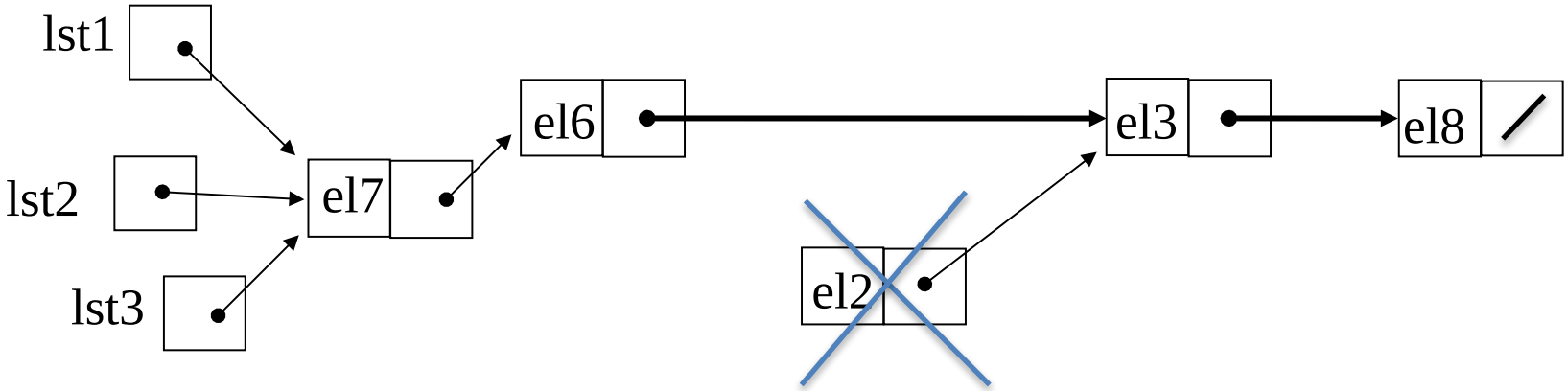
- Vediamo perché con un esempio …

Attenti alle interferenze!

Partiamo dalla seguente situazione, ottenuta mediante l'operazione lst1 = lst2.



Ed eseguiamo l'operazione lst3 = remove(lst2, 2) …



L'operazione è stata eseguita su tutte e tre le liste provocando un *effetto collaterale* (*side effect*) …

Evitare interferenze e side effect

1. Evitare operazioni del tipo: lst1 = lst2;
   - Se vogliamo che due liste abbiano gli stessi elementi introduciamo un operatore che duplica la struttura e il contenuto di una lista (implementarla come esercizio)
     - list cloneList(list);
   - ed usiamolo nel modo seguente
     - lst2 = cloneList(lst1);
2. Riguardo agli operatori insertList e removeList
   - Usarli solo nel modo seguente, assegnando il risultato dell'operazione alla variabile di input:
     - lst = insertList(lst, pos, val);
     - lst = removeList(lst, pos);

Un bel problema …

- Se il programmatore del modulo client non rispettasse queste convenzioni, si troverebbe in un bel pasticcio

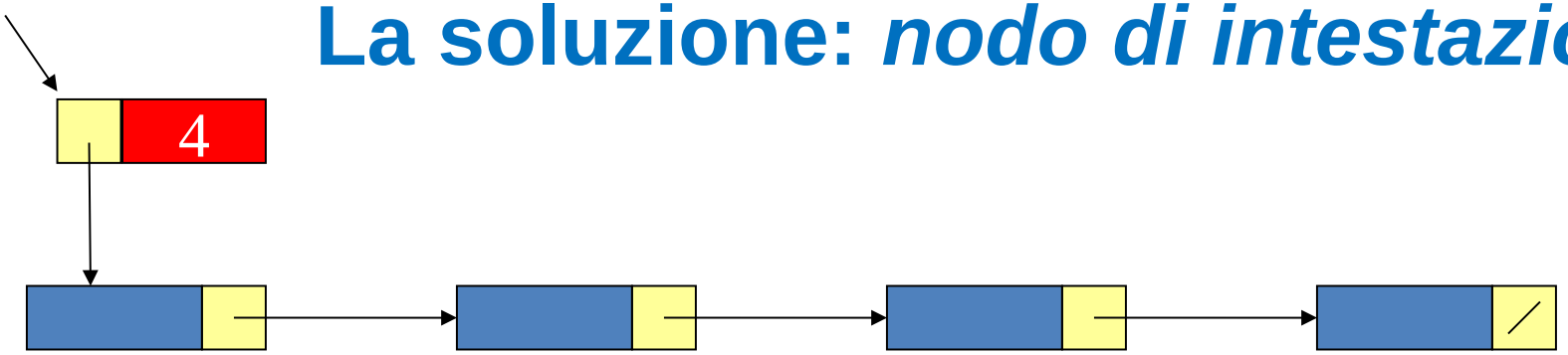
- Sarebbe meglio progettare gli operatori in modo che il parametro list sia al contempo sia di ingresso che di uscita, ad esempio:

  - void insertList(list *l, int pos, item val)
  - void removeList(list *l, int pos)

- Ma questa soluzione non è elegante, e modifica l'interfaccia

- Per evitare di passare un puntatore a list abbiamo bisogno di progettare in maniera diversa anche la struttura dati
  - Abbiamo bisogno di un ulteriore contenitore

La soluzione: *nodo di intestazione*

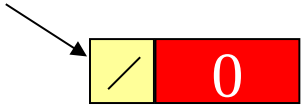



```
// in list.h

typedef struct c_list *list;
```

- La nostra lista sarà un puntatore ad una sorta di ***contenitore*** struct c_list che contiene due elementi
  - Un puntatore a struct node
  - Il numero di elementi contenuti nella lista
    - Questo semplifica operazioni come sizeList()

```
// in list.c

struct c_list {
    struct node *first;
    int size;
};

struct node {
  item value;
  struct node *next;
};
```

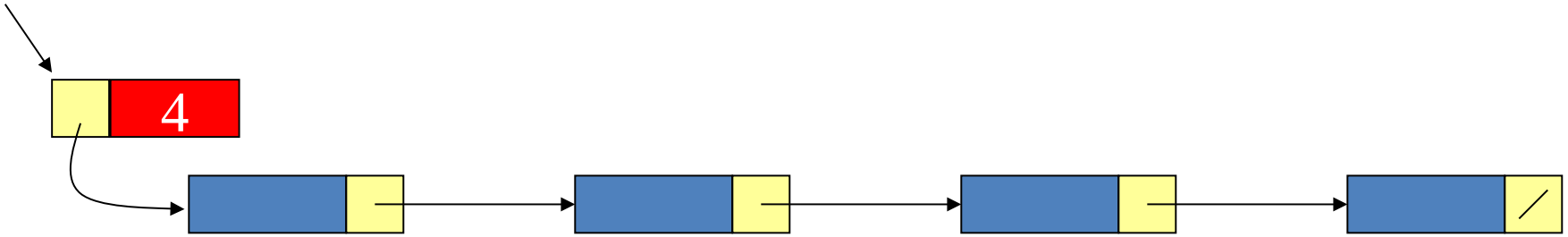
La lista vuota



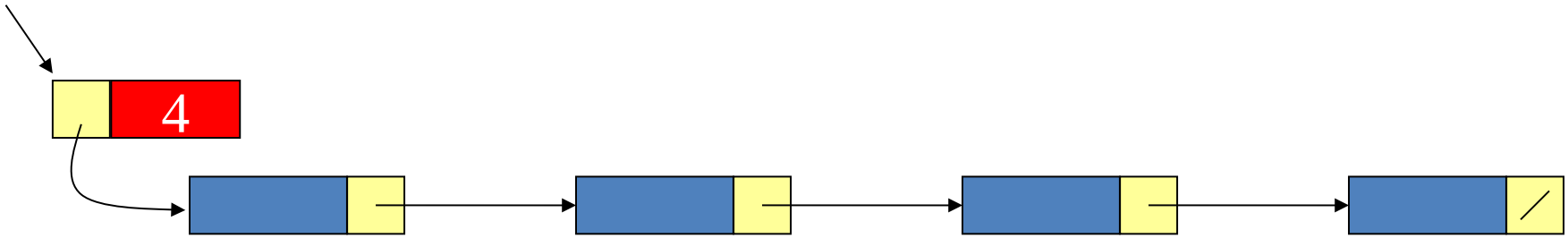

```
list newList(void)
{
    struct c_list *l;
    l = malloc (sizeof(struct c_list));
    if (l != NULL) {
            l->first = NULL;
            l->size = 0;
        }
     return l;
// il programma client dovrà controllare che il risultato non sia NULL
}
```

La soluzione

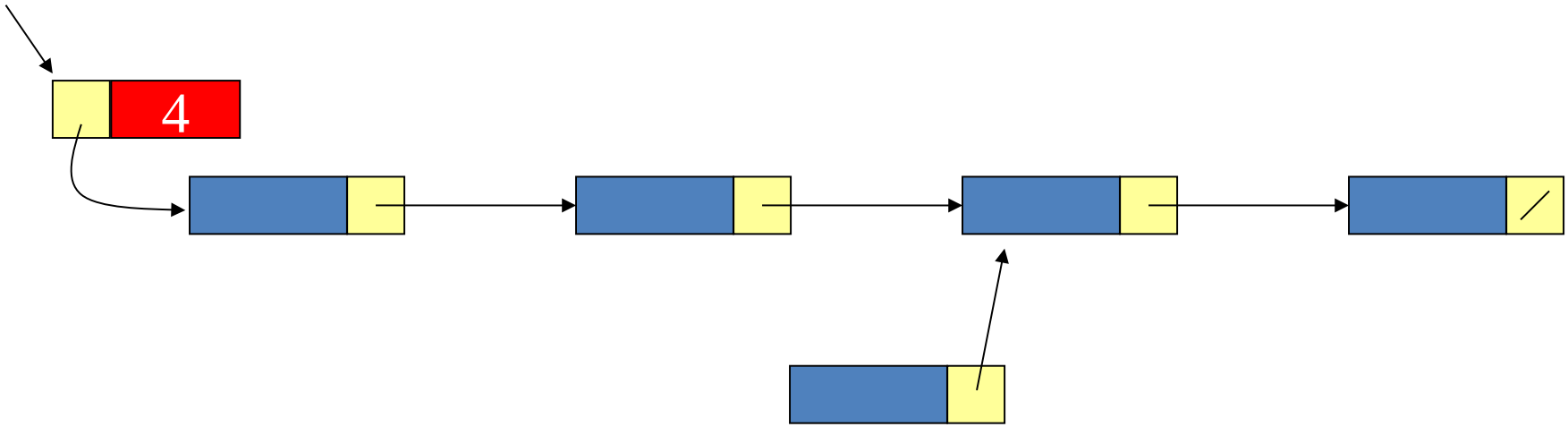
- In questo modo i nostri operatori di inserimento e rimozione saranno:
  - int insertList(list l, int pos, item val)
  - int removeList(list l, int pos)
  - Progettiamo i due operatori in modo che restituiscano un intero: 1 se l'operazione è andata a buon fine e 0 altrimenti
- Si va a modificare la lista collegata puntata da l->first
  - Esempio insert …



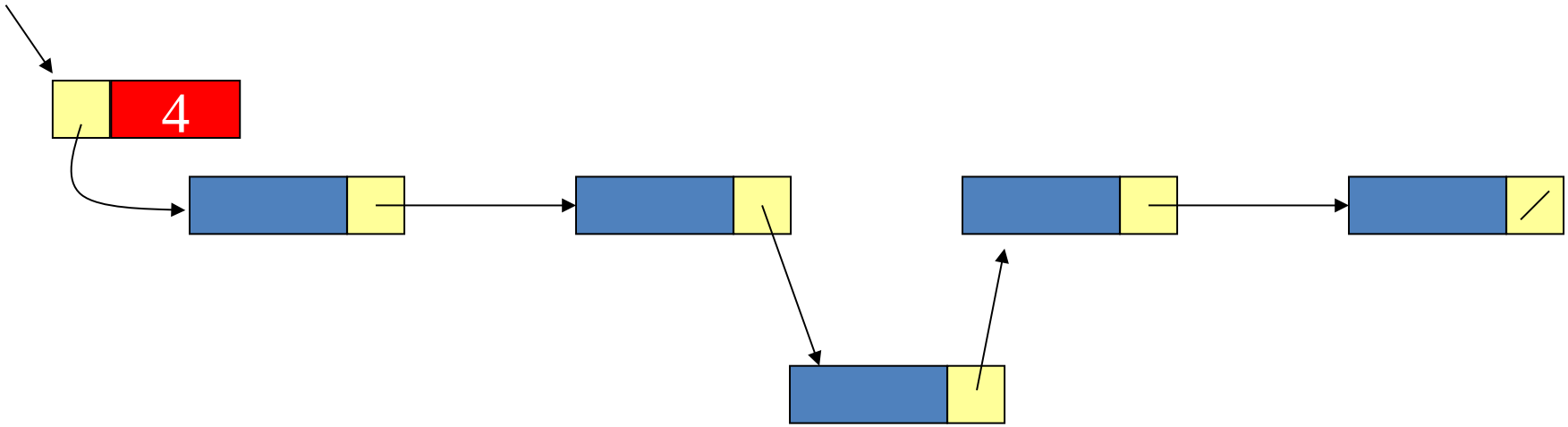
Esempio Insert



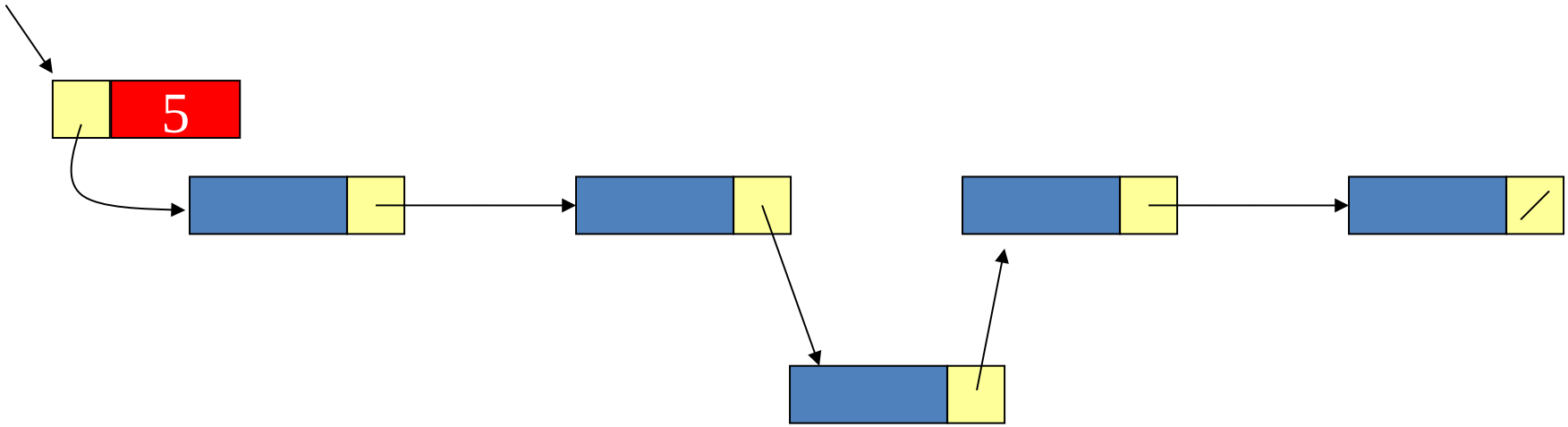
Esempio Insert



Esempio Insert



Esempio Insert



InsertList ...

```
int insertList (list l, int pos, item val)
{
  struct node* tmp= insertNode(l->first, pos, val);
  if(tmp==NULL) return 0;
  l->first = tmp;       l->size++;
  return 1;
}

static struct node* insertNode (struct node* l, int pos, item val)
{
    struct node *new, *prec = l;              int i;
    new = malloc(sizeof(struct node));
    if (!new)   return NULL;
    new->value = val;
    if(pos == 0)  {              // inserimento in testa
          new->next = l;
          return new;
    }
    while (i < pos-1 && prec!= NULL) {   // scorro la lista fino ad arrivare a pos
       prec = prec->next;        i++;
    }
    if (prec == NULL)  {                    // la lista di input ha meno di pos elementi
         free(new);     return NULL;
    }
    new->next = prec->next;       prec->next = new;      // aggiungo in posizione pos
    return l;
}
```

… Implementazione di removeList

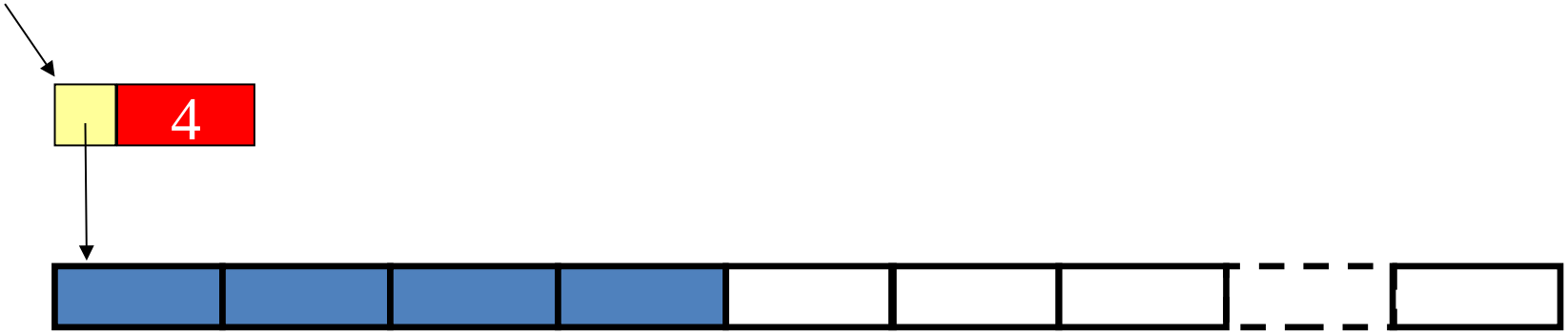
```
int removeList (list l, int pos)
{
   if (!l || l->first==NULL || l->size==0)   return 0;   // non ci sono elementi
   l->first = removeNode(l->first,pos);
   l->size--;
   return 1;
}

static struct node* removeNode(struct node* l, int pos)
{
   struct node* l1;    // puntatore al nodo da eliminare

   if(pos == 0 && l != NULL) {      // eliminazione in posizione 0
      l1 = l;
      l = l->next;
      free(l1);
    }
  // … segue al prossimo lucido
```

... Implementazione di removeList

```
else {
        // scorriamo la lista fino alla posizione precedente a quella
        // del nodo da eliminare
        int i = 0;
        struct node* prec = l;
        while (i < pos-1 && prec!= NULL) {
            prec = prec->next;
            i++;
        }
/* alla fine del ciclo, se prec != NULL allora prec->next punta al nodo da
   eliminare. Se prec->next != NULL allora il nodo si può eliminare */
        if(prec != NULL && prec->next != NULL) { // short-circuit evaluation
            l1 = prec->next;
            prec->next = l1->next;
            free(l1);
        }
    }
    return l;
}
```

Altre implementazioni

- La soluzione appena vista apre la strada ad altre implementazioni di strutture dati lineari, come ad esempio quelle basate su array …



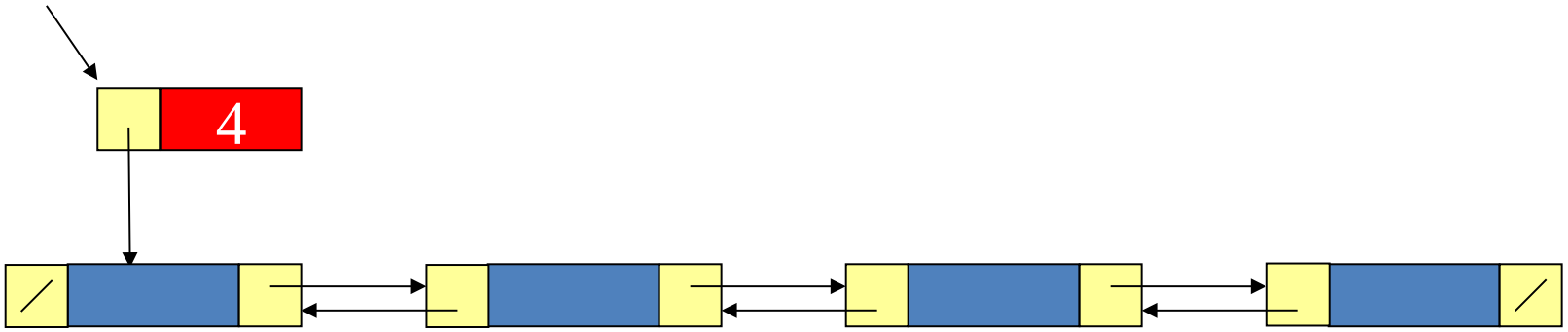
- Ovviamente in questo caso bisogna definire anche la dimensione massima della lista …
- Vedremo questo tipo di soluzione progettuale nell'implementazione di altre strutture dati lineari, come pile e code

Array o liste collegate ?

- Quale tipo di implementazione è migliore per una lista?
- Dipende da come verrà usata la lista …
- Se il numero di inserimenti/rimozioni è preponderante rispetto al numero di accessi, allora conviene usare liste collegate
  - Con un array ogni inserimento/rimozione richiede uno shift degli elementi
- Se il numero di inserimenti/rimozioni è marginale rispetto al numero di accessi agli elementi, ed è possibile stabilire una taglia massima, allora conviene usare un array
  - Con un array si ha accesso diretto all'elemento di posizione pos

Altre implementazioni collegate

- Se opto per una struttura collegata e ho necessità di scorrere la lista in tutte e due le direzioni, posso usare una lista doppiamente collegata …



Esercizi

- Implementare il tipo di dato astratto lista utilizzando la struttura a puntatori con il contenitore iniziale c_list
- Implementare tutti gli operatori della lista specificati nella lezione L06

Esercizi

- Si implementi, mediante l'uso di opportune strutture dati, un programma per la gestione dei libretti universitari degli studenti.
- Specificare e implementare l'ADT libretto: ogni libretto tiene traccia dei dati dello studente (cognome, nome, matricola) e degli esami sostenuti. Questi ultimi sono caratterizzati da nome, voto e data dell'esame. L'ADT libretto dovrà consentire di aggiungere esami al libretto e di ricercare un esame in base al nome.
- Realizzare il programma di test