Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

La struttura dati LISTA

Andrea Piroddi - andrea.piroddi@unibo.it
Credit: Pietro Di Lena

Introduzione

- Una **lista** è una **struttura dati astratta** che rappresenta una sequenza di *oggetti*.
  1. Ogni oggetto nella lista è associato ad una posizione nella lista: possiamo accedere ad un oggetto tramite riferimento alla sua posizione nella lista.
  2. Lo stesso oggetto può apparire più volte nella lista: ogni occorrenza è considerata un'entità distinta.
  3. La lista può contenere un numero illimitato di oggetti: l'unica limitazione è data dai limiti di memoria del calcolatore, non possiamo fissare una capienza massima a priori per la lista.

Introduzione

- Una **lista** è una **struttura dati astratta** che rappresenta una sequenza di *oggetti*.
  1. Ogni oggetto nella lista è associato ad una posizione nella lista: possiamo accedere ad un oggetto tramite riferimento alla sua posizione nella lista.
  2. Lo stesso oggetto può apparire più volte nella lista: ogni occorrenza è considerata un'entità distinta.
  3. La lista può contenere un numero illimitato di oggetti: l'unica limitazione è data dai limiti di memoria del calcolatore, non possiamo fissare una capienza massima a priori per la lista.
- Numerosi linguaggi di programmazione supportano la struttura dati lista e forniscono specifici costrutti sintattici per dichiarare ed eseguire operazioni su liste.
- Nel linguaggio C, gli array ci permettono di gestire una struttura dati di tipo lista.
- La sintassi del linguaggio C per la manipolazione degli array non ci permette di gestire una struttura dati di tipo lista che rispetti tutte le proprietà elencate sopra (nello specifico, la proprietà 3).

Implementazione del tipo di dato astratto lista in C

- Vediamo diversi approcci per implementare una struttura dati di tipo lista in C.
  - Per semplicità, ci concentriamo unicamente su liste di interi (tipo di dato `int`).
  - Le implementazioni possono essere facilmente modificate per poter gestire tipi di dato generici (ad esempio, liste di liste).

Implementazione del tipo di dato astratto lista in C

- Vediamo diversi approcci per implementare una struttura dati di tipo lista in C.
  - Per semplicità, ci concentriamo unicamente su liste di interi (tipo di dato `int`).
  - Le implementazioni possono essere facilmente modificate per poter gestire tipi di dato generici (ad esempio, liste di liste).
- Come procediamo.
  - Partiamo dai prototipi di alcune operazioni che vogliamo implementare su una struttura dati di tipo lista.
  - Dobbiamo definire il tipo di dato lista e implementare le operazioni richieste su tale tipo di dato.

Implementazione del tipo di dato astratto lista in C

- Vediamo diversi approcci per implementare una struttura dati di tipo lista in C.
  - Per semplicità, ci concentriamo unicamente su liste di interi (tipo di dato `int`).
  - Le implementazioni possono essere facilmente modificate per poter gestire tipi di dato generici (ad esempio, liste di liste).
- Come procediamo.
  - Partiamo dai prototipi di alcune operazioni che vogliamo implementare su una struttura dati di tipo lista.
  - Dobbiamo definire il tipo di dato lista e implementare le operazioni richieste su tale tipo di dato.
- Scelte progettuali generali.
  - Vogliamo che il nostro codice sia **efficiente**: dobbiamo fornire l'implementazione più efficiente in termini di **utilizzo di memoria** e **tempo di calcolo** (queste caratteristiche dipendono da come abbiamo definito il tipo di dato lista).
  - Vogliamo che il nostro codice sia **modulare**: le funzioni devono essere **semplici** (poche righe di codice), **specifiche** (un solo compito) e **riutilizzabili** (vogliamo poter riutilizzare una funzione in tutti i contesti in cui sia possibile).

Operazioni sulla struttura dati astratta lista di interi

- Consideriamo una serie di operazioni *astratte* su una lista che contenga interi.
- Ci concentriamo unicamente su alcune operazioni basilari.
  - Creazione di una lista
  - Distruzione di una lista
  - Test di verifica su lista vuota
  - Test di verifica su presenza di un elemento
  - Inserimento in testa alla lista
  - Inserimento in coda alla lista
  - Inserimento in una posizione della lista
  - Rimozione della testa della lista
  - Rimozione della coda della lista
  - Rimozione dell'elemento in una posizione della lista
  - Selezione dell'elemento in testa alla lista
  - Selezione dell'elemento in coda alla lista
  - Selezione dell'elemento in una posizione della lista

Operazioni sulla struttura dati astratta lista di interi

- Consideriamo una serie di operazioni *astratte* su una lista che contenga interi.
- Ci concentriamo unicamente su alcune operazioni basilari.
  - Creazione di una lista
  - Distruzione di una lista
  - Test di verifica su lista vuota
  - Test di verifica su presenza di un elemento
  - Inserimento in testa alla lista
  - Inserimento in coda alla lista
  - Inserimento in una posizione della lista
  - Rimozione della testa della lista
  - Rimozione della coda della lista
  - Rimozione dell'elemento in una posizione della lista
  - Selezione dell'elemento in testa alla lista
  - Selezione dell'elemento in coda alla lista
  - Selezione dell'elemento in una posizione della lista
- Per capire come sono definite a livello semantico queste operazioni:
  - rappresentiamo in modo astratto una lista di interi come una sequenza di interi, sperati da una virgola e racchiusi tra parentesi graffe.

    $\{$`-10, 2, 3, 0, 4, -1`$\}$
  - la lista vuota viene rappresentata con una coppia di parentesi graffe vuote:

    $\{\}$

Operazioni sulla struttura dati astratta lista di interi: semantica 1/4

- **Creazione di una lista**. L'operazione genera una lista vuota $\{\}$.

Operazioni sulla struttura dati astratta lista di interi: semantica 1/4

- **Creazione di una lista**. L'operazione genera una lista vuota $\{\}$.
- **Distruzione di una lista**. L'operazione rimuove tutti gli elementi nella lista. Esempio, l'operazione di distruzione applicata alla seguente lista

$$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

produce la lista vuota

$$\{\}$$

Operazioni sulla struttura dati astratta lista di interi: semantica 1/4

- **Creazione di una lista**. L'operazione genera una lista vuota $\{\}$.
- **Distruzione di una lista**. L'operazione rimuove tutti gli elementi nella lista. Esempio, l'operazione di distruzione applicata alla seguente lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

  produce la lista vuota

  $$\{\}$$

- **Test booleno su lista vuota**. Verifica se la lista è vuota. Esempio, l'operazione ritorna `true` sulla lista

  $$\{\}$$

  e `false` sulla lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

Operazioni sulla struttura dati astratta lista di interi: semantica 1/4

- **Creazione di una lista**. L'operazione genera una lista vuota $\{\}$.
- **Distruzione di una lista**. L'operazione rimuove tutti gli elementi nella lista. Esempio, l'operazione di distruzione applicata alla seguente lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

  produce la lista vuota

  $$\{\}$$

- **Test booleno su lista vuota**. Verifica se la lista è vuota. Esempio, l'operazione ritorna `true` sulla lista

  $$\{\}$$

  e `false` sulla lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

- **Test booleno sulla presenza di un elemento**. Verifica se un elemento è presente nella lista. Esempio, l'operazione ritorna `true` se verifichiamo la presenza del numero 3 e `false` se verifichiamo la presenza del numero 10 nella lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

Operazioni sulla struttura dati astratta lista di interi: semantica 2/4

- **Inserimento in testa**. L'operazione inserisce un elemento in testa alla lista. Esempio, l'inserimento in testa del numero `0` nella lista

$$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

genera la lista

$$\{\texttt{0, -10, 2, 3, 0, 4, -1}\}$$

Operazioni sulla struttura dati astratta lista di interi: semantica 2/4

- **Inserimento in testa**. L'operazione inserisce un elemento in testa alla lista. Esempio, l'inserimento in testa del numero `0` nella lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

  genera la lista

  $$\{\texttt{0, -10, 2, 3, 0, 4, -1}\}$$

- **Inserimento in coda**. L'operazione inserisce un elemento in coda alla lista. Esempio, l'inserimento in coda del numero `0` nella lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

  genera la lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1, 0}\}$$

Operazioni sulla struttura dati astratta lista di interi: semantica 2/4

- **Inserimento in testa**. L'operazione inserisce un elemento in testa alla lista. Esempio, l'inserimento in testa del numero `0` nella lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4,\ -1\}$$

  genera la lista

  $$\{0,\ -10,\ 2,\ 3,\ 0,\ 4,\ -1\}$$

- **Inserimento in coda**. L'operazione inserisce un elemento in coda alla lista. Esempio, l'inserimento in coda del numero `0` nella lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4,\ -1\}$$

  genera la lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4,\ -1,\ 0\}$$

- **Inserimento indicizzato**. L'operazione inserisce un elemento nella posizione `i`-esima della lista. Esempio, l'inserimento in posizione `i=2` del numero `0` nella lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4,\ -1\}$$

  genera la lista

  $$\{-10,\ 2,\ 0,\ 3,\ 0,\ 4,\ -1\}$$

  Nota. Assumiamo che gli indici delle posizioni nella lista partano da 0 (come di default nel linguaggio C).

Operazioni sulla struttura dati astratta lista di interi: semantica 3/4

- **Rimozione in testa**. L'operazione rimuove l'elemento in testa alla lista. Esempio, la rimozione in in testa nella lista

$$\{-10,\ 2,\ 3,\ 0,\ 4,\ -1\}$$

genera la lista

$$\{2,\ 3,\ 0,\ 4,\ -1\}$$

Operazioni sulla struttura dati astratta lista di interi: semantica 3/4

- **Rimozione in testa**. L'operazione rimuove l'elemento in testa alla lista. Esempio, la rimozione in in testa nella lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4,\ -1\}$$

  genera la lista

  $$\{2,\ 3,\ 0,\ 4,\ -1\}$$

- **Rimozione in coda**. L'operazione rimuove l'elemento in coda alla lista. Esempio, la rimozione in coda nella lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4,\ -1\}$$

  genera la lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4\}$$

Operazioni sulla struttura dati astratta lista di interi: semantica 3/4

- **Rimozione in testa**. L'operazione rimuove l'elemento in testa alla lista. Esempio, la rimozione in in testa nella lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4,\ -1\}$$

  genera la lista

  $$\{2,\ 3,\ 0,\ 4,\ -1\}$$

- **Rimozione in coda**. L'operazione rimuove l'elemento in coda alla lista. Esempio, la rimozione in coda nella lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4,\ -1\}$$

  genera la lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4\}$$

- **Rimozione indicizzata**. L'operazione rimuove l'elemento nella posizione `i`-esima della lista. Esempio, la rimozione dell'elemento in posizione `i`=2 nella lista

  $$\{-10,\ 2,\ 3,\ 0,\ 4,\ -1\}$$

  genera la lista

  $$\{-10,\ 2,\ 0,\ 4,\ -1\}$$

Operazioni sulla struttura dati astratta lista di interi: semantica 4/4

- **Selezione della testa**. L'operazione ritorna il valore dell'elemento in testa alla lista. Esempio, la selezione in testa nella lista

$$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

ritorna il numero `-10`. La lista non viene modificata.

Operazioni sulla struttura dati astratta lista di interi: semantica 4/4

- **Selezione della testa**. L'operazione ritorna il valore dell'elemento in testa alla lista. Esempio, la selezione in testa nella lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

  ritorna il numero `-10`. La lista non viene modificata.
- **Selezione della coda**. L'operazione ritorna il valore dell'elemento in coda alla lista. Esempio, la selezione in coda nella lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

  ritorna il numero `-1`. La lista non viene modificata.

Operazioni sulla struttura dati astratta lista di interi: semantica 4/4

- **Selezione della testa**. L'operazione ritorna il valore dell'elemento in testa alla lista. Esempio, la selezione in testa nella lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

  ritorna il numero `-10`. La lista non viene modificata.
- **Selezione della coda**. L'operazione ritorna il valore dell'elemento in coda alla lista. Esempio, la selezione in coda nella lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

  ritorna il numero `-1`. La lista non viene modificata.
- **Selezione indicizzata**. L'operazione ritorna il valore dell'elemento nella posizione `i`-esima della lista. Esempio, la selezione dell'elemento in posizione `i=2` nella lista

  $$\{\texttt{-10, 2, 3, 0, 4, -1}\}$$

  ritorna il numero `3`. La lista non viene modificata.

Implementazioni delle operazioni sulla struttura dati lista di interi

- Consideriamo i seguenti prototipi, che definiscono le operazioni richieste su una lista.
- I prototipi prendono in input un puntatore al tipo di dato `list` (da definire).

```
1  list list_create();
2
3  void list_delete(list *);
4
5  int is_empty(list *);
6
7  int is_inlist(list *, int);
8
9  int head_insert(list *, int);
10
11 int tail_insert(list *, int);
12
13 int indx_insert(list *, int, unsigned int);
14
15 int head_delete(list *);
16
17 int tail_delete(list *);
18
19 int indx_delete(list *, unsigned int);
20
21 int head_select(list *);
22
23 int tail_select(list *);
24
25 int indx_select(list *, unsigned int);
```

Operazioni sulla struttura dati lista: creazione e distruzione

- Le seguenti funzioni ci permettono di:
  - *creare* un oggetto di tipo lista. Inizialmente la lista non contiene elementi,
  - *distruggere* l'intero contenuto di un oggetto di tipo lista.

```
1  /*
2   * Crea e ritorna una lista vuota.
3   */
4  list list_create();
5
6  /*
7   * Elimina tutti gli elementi contenuti nella lista
8   * puntata da L.
9   *
10  * Al termine della chiamata, lo stato della lista puntata
11  * da L Ã¨ equivalente allo stato di una lista creata
12  * con list_create().
13  */
14 void list_delete(list *L);
```

Operazioni sulla struttura dati lista: test di proprietà della lista

- Le seguenti funzioni (booleane) ci permettono di:
  - verificare se la lista è vuota o meno,
  - verificare se la lista contenga o meno un certo elemento.

```
1  /*
2   * Ritorna 0 se la lista non e' vuota,
3   * un intero diverso da zero altrimenti.
4   */
5  int is_empty(list *L);
6
7  /*
8   * Ritorna 0 se elem non e' contenuto nella lista,
9   * un intero diverso da zero altrimenti.
10  */
11 int is_inlist(list *L, int elem);
```

Operazioni sulla struttura dati lista: inserimento

- Le seguenti funzioni permettono di inserire un elemento all'interno della lista.

```c
/*
 * Inserisce elem in testa alla lista puntata da L.
 *
 * Ritorna 0 se l'inserimento ha successo,
 * un intero diverso da zero altrimenti.
 */
int head_insert(list *L, int elem);
/*
 * Inserisce elem in coda alla lista puntata da L.
 *
 * Ritorna 0 se l'inserimento ha successo,
 * un intero diverso da zero altrimenti.
 */
int tail_insert(list *L, int elem);
/*
 * Inserisce elem nella posizione i-esima della
 * lista puntata da L.
 * In una lista contenente esattamente n interi,
 * le posizioni i valide vanno da 0 a n:
 * - i=0 equivale ad inserimento in testa,
 * - i=n equivale ad inserimento in coda.
 *
 * Ritorna 0 se l'inserimento ha successo,
 * un intero diverso da zero altrimenti.
 */
int indx_insert(list *L, int elem, unsigned int i);
```

Operazioni sulla struttura dati lista: rimozione

- Le seguenti funzioni permettono di rimuovere un elemento all'interno della lista.

```
1  /*
2   * Rimuove l'elemento in testa alla lista puntata da L.
3   *
4   * Ritorna 0 se la rimozione ha successo,
5   * un intero diverso da zero altrimenti.
6   */
7  int head_delete(list *L);
8  /*
9   * Rimuove l'elemento in coda alla lista puntata da L.
10  *
11  * Ritorna 0 se la rimozione ha successo,
12  * un intero diverso da zero altrimenti.
13  */
14 int tail_delete(list *L);
15 /*
16  * Rimuove l'elemento nella posizione i-esima della
17  * lista puntata da L.
18  * In una lista contenente esattamente n interi,
19  * le posizioni i valide vanno da 0 a n-1:
20  * - i=0 equivale a rimozione  in testa,
21  * - i=n-1 equivale a rimozione in coda.
22  *
23  * Ritorna 0 se la rimozione ha successo,
24  * un intero diverso da zero altrimenti.
25  */
26 int indx_delete(list *L, unsigned int elem);
```

Operazioni sulla struttura dati lista: selezione

- Le seguenti funzioni permettono di recuperare un valore all'interno della lista.

```
1  /*
2   * Ritorna il valore in testa alla lista puntata da L.
3   *
4   * Se la lista Ã¨ vuota, il comportamento Ã¨ non definito.
5   */
6  int head_select(list *L);
7  /*
8   * Ritorna il valore in coda alla lista puntata da L.
9   *
10  * Se la lista Ã¨ vuota, il comportamento Ã¨ non definito.
11  */
12 int tail_select(list *L);
13 /*
14  * Ritorna il valore nella posizione i-esima
15  * della lista puntata da L.
16  *
17  * In una lista contenente esattamente n interi,
18  * le posizioni i valide vanno da 0 a n-1.
19  * - i=0 equivale a rimozione  in testa,
20  * - i=n-1 equivale a rimozione in coda.
21  *
22  * Se la lista Ã¨ vuota, oppure i non Ã¨ un indice valido,
23  * il comportamento Ã¨ non definito.
24  */
25 int indx_select(list *L, unsigned int i);
```

Liste con array

- La rappresentazione più semplice per la struttura dati lista in C è tramite array.
- L'array deve essere ridimensionato (all'occorrenza) per poter rispettare uno dei vincoli richiesti dalla struttura dati lista: non possiamo fissare a priori il numero massimo di oggetti memorizzabili.

Liste con array

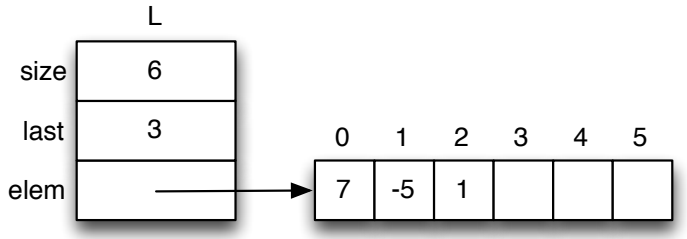
- La rappresentazione più semplice per la struttura dati lista in C è tramite array.
- L'array deve essere ridimensionato (all'occorrenza) per poter rispettare uno dei vincoli richiesti dalla struttura dati lista: non possiamo fissare a priori il numero massimo di oggetti memorizzabili.
- Per poter gestire efficientemente l'uso di memoria (una delle nostre scelte progettuali) siamo costretti a prevedere nell'implementazione alcune strategie per:
  - aumentare la dimensione dell'array quando siamo al limite della capienza,
  - diminuire la dimensione dell'array quando è sotto-utilizzato.

Liste con array

- La rappresentazione più semplice per la struttura dati lista in C è tramite array.
- L'array deve essere ridimensionato (all'occorrenza) per poter rispettare uno dei vincoli richiesti dalla struttura dati lista: non possiamo fissare a priori il numero massimo di oggetti memorizzabili.
- Per poter gestire efficientemente l'uso di memoria (una delle nostre scelte progettuali) siamo costretti a prevedere nell'implementazione alcune strategie per:
  - aumentare la dimensione dell'array quando siamo al limite della capienza,
  - diminuire la dimensione dell'array quando è sotto-utilizzato.
- Ulteriori problemi implementativi legati alla rappresentazione tramite array:
  - Come facciamo a capire qual è la prima posizione libera nell'array per un inserimento in coda?
  - Come facciamo per gestire un inserimento nella posizione `i`-esima (interna alla lista), senza perdere il valore dell'`i`-esimo elemento attualmente memorizzato?

Il tipo di dato lista con rappresentazione array

- Possiamo rappresentare il tipo di dato `list` come una **struttura** contente:
  - Un campo puntatore (`elem`) ad un array di interi (allocato dinamicamente).
  - Un campo lunghezza (`size`), contenente la lunghezza dell'array.
  - Un campo indice (`last`) che indica la prima posizione libera nell'array.




```
1  struct list {
2    unsigned int size;
3    unsigned int last;
4    int *elem;
5  };
6
7  typedef struct list list;
```

- La struttura `list` contiene tutte le informazioni necessarie per gestire l'inserimento, rimozione, ricerca di elementi e riallocazione dinamica dell'array.
- Le funzioni di libreria hanno come argomento formale un puntatore alla struttura `list`.

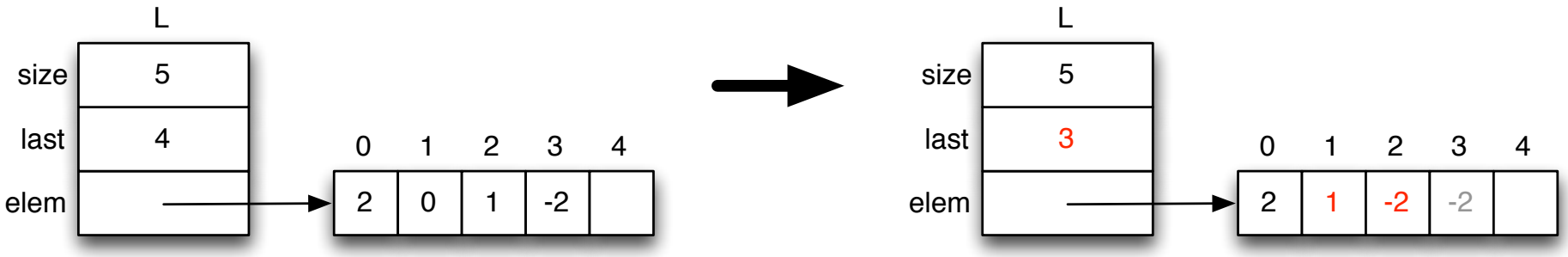
Untility di libreria per liste con array

- Le implementazioni delle funzioni di libreria possono essere semplificate facendo uso di alcune utility (nascoste all'esterno).
- In particolare, le funzioni `extend()` e `reduce()` implementano strategie per gestire il ridimensionamento dell'array dinamico: se richiamate dopo un inserimento o rimozione decidono se è il caso o meno di ridimensionare l'array.

```
1  /* Lunghezza iniziale */
2  #define N 1
3  /*
4   * Sposta tutti gli elementi nella lista di una posizione
5   * a sinistra a partire dalla posizione i
6   */
7  static void list_lshift(list *L, unsigned i);
8  /*
9   * Sposta tutti gli elementi nella lista di una posizione
10  * a destra a partire dalla posizione i
11  */
12 static void list_rshift(list *L, unsigned i);
13 /*
14  * Se necessario, estende la dimensione della lista.
15  */
16 static void list_extend(list *L);
17 /*
18  * Se necessario, riduce la dimensione della lista.
19  */
20 static void list_reduce(list *L);
```

Utility per lo shift di elementi a sinistra: `list_lshift()`

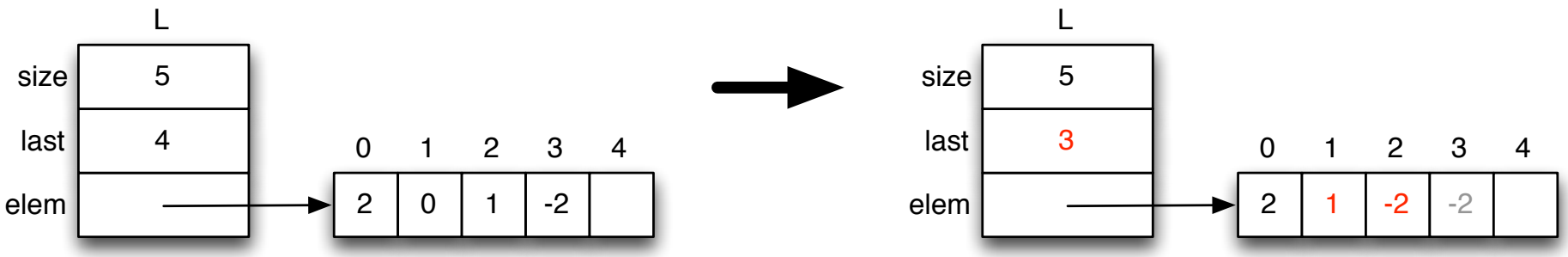
- Sposta a sinistra di un passo tutti gli elementi nell'array a partire da una posizione `i`.
- Esempio: shift a sinistra a partire dalla posizione `i` = 2.



- E' necessario aggiornare il campo `last`: nuova prima posizione libera.
- Non è necessario sovrascrivere il contenuto attuale nella posizione `last`.

Utility per lo shift di elementi a sinistra: `list_lshift()`

- Sposta a sinistra di un passo tutti gli elementi nell'array a partire da una posizione `i`.
- Esempio: shift a sinistra a partire dalla posizione `i` = 2.



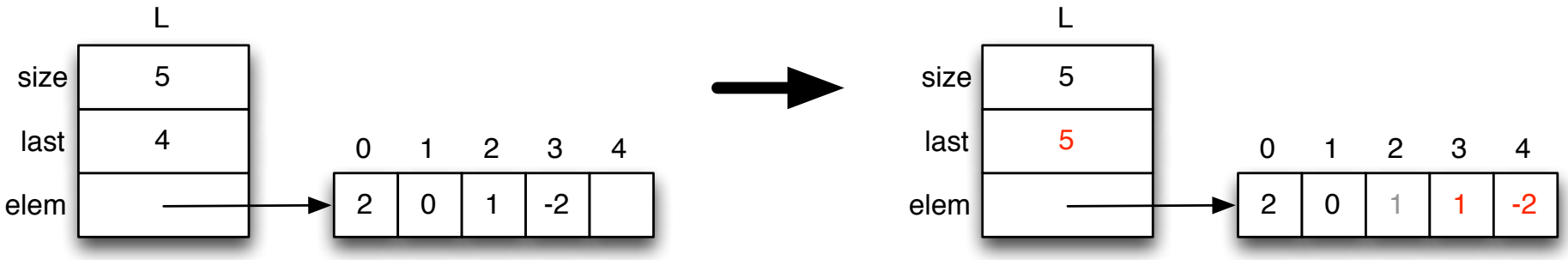
- E' necessario aggiornare il campo `last`: nuova prima posizione libera.
- Non è necessario sovrascrivere il contenuto attuale nella posizione `last`.

```
1 static void list_lshift(list *L, unsigned int i) {
2   while(i < L->last) {
3     L->elem[i-1] = L->elem[i];
4     i++;
5   }
6 }
```

- La funzione non controlla che gli argomenti passati siano *consistenti* (ex `i` $\neq 0$).
- Non può essere chiamata all'esterno della libreria, quindi è nostro compito invocarla correttamente quando necessario.

Utility per lo shift di elementi a destra: `list_rshift()`

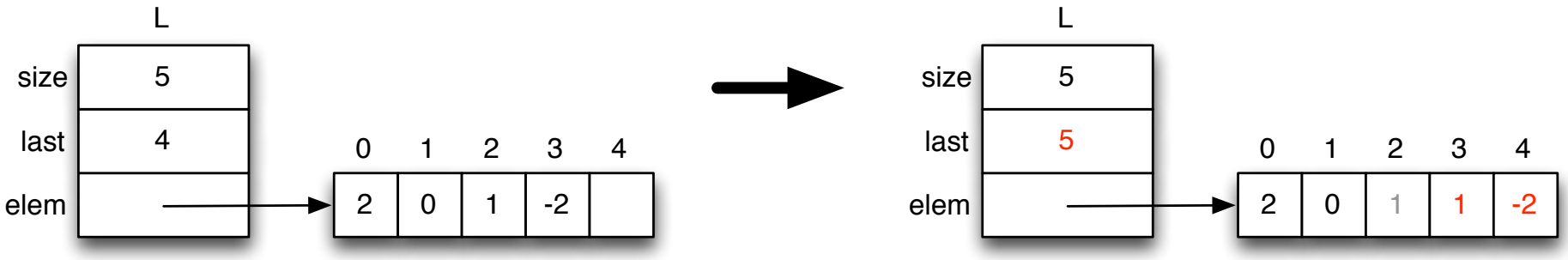
- Sposta a destra di un passo tutti gli elementi nell'array a partire da una posizione `i`.
- Esempio: shift a destra a partire dalla posizione `i` = 2.



- E' necessario aggiornare il campo `last`: nuova prima posizione libera.
- Non è necessario sovrascrivere il contenuto attuale nella posizione `i`, adesso *libera*.

Utility per lo shift di elementi a destra: `list_rshift()`

- Sposta a destra di un passo tutti gli elementi nell'array a partire da una posizione `i`.
- Esempio: shift a destra a partire dalla posizione `i` = 2.



- E' necessario aggiornare il campo `last`: nuova prima posizione libera.
- Non è necessario sovrascrivere il contenuto attuale nella posizione `i`, adesso *libera*.

```
1 static void list_rshift(list *L, unsigned int i) {
2   unsigned int j = L->last;
3   while(i < j) {
4     L->elem[j] = L->elem[j-1];
5     j--;
6   }
7 }
```

- Come per `list_lshift()`, è nostro compito invocarla correttamente.

Utility per espandere la lista: `list_extend()`

- E' richiesta una struttura dati in grado di gestire un numero illimitato di elementi.
- La scelta implementativa con array impone la necessità di avere qualche procedura per espandere la dimensione della lista quando necessario.
- Vogliamo che la procedura di espansione sia indipendente il più possibile dalle funzioni di inserimento: in questo modo possiamo successivamente modificare unicamente la funzione che gestisce l'espansione senza dover modificare tutto il resto.

Utility per espandere la lista: `list_extend()`

- E' richiesta una struttura dati in grado di gestire un numero illimitato di elementi.
- La scelta implementativa con array impone la necessità di avere qualche procedura per espandere la dimensione della lista quando necessario.
- Vogliamo che la procedura di espansione sia indipendente il più possibile dalle funzioni di inserimento: in questo modo possiamo successivamente modificare unicamente la funzione che gestisce l'espansione senza dover modificare tutto il resto.
- In quest'ottica, facciamo la seguente scelta progettuale:
  - la procedura di espansione viene richiamata sempre dopo ogni inserimento: decide lei se espandere o meno l'array.

```
1  static void list_extend(list *L) {
2    if(L->size == L->last) {
3      int *tmp = realloc(L->elem,2*L->size*sizeof(int));
4
5      if(tmp != NULL) {
6        L->size = 2*L->size;
7        L->elem = tmp;
8      }
9    }
10 }
```

- La strategia implementata è quella di raddoppiare la dimensione dell'array solo quando abbiamo raggiunto il limite massimo di memorizzazione.
- Se la riallocazione fallisce, la funzione non modifica nulla.

Utility per ridurre la lista: `list_reduce()`

- Per ottimizzare l'uso della memoria, vogliamo evitare di avere un array troppo capiente quando ci sono pochi elementi memorizzati.
- Nello spirito della procedura di espansione, vogliamo che la procedura di riduzione sia indipendente il più possibile dalle funzioni di cancellazione.

Utility per ridurre la lista: `list_reduce()`

- Per ottimizzare l'uso della memoria, vogliamo evitare di avere un array troppo capiente quando ci sono pochi elementi memorizzati.
- Nello spirito della procedura di espansione, vogliamo che la procedura di riduzione sia indipendente il più possibile dalle funzioni di cancellazione.
- Adottiamo la seguente strategia progettuale:
  - la procedura di riduzione viene richiamata sempre dopo ogni cancellazione: decide lei se ridurre o meno l'array.

```
1  static void list_reduce(list *L) {
2    if(L->size/3 > L->last) {
3      int *tmp = realloc(L->elem,L->size/2*sizeof(int));
4
5      if(tmp != NULL) {
6        L->elem = tmp;
7        L->size = L->size/2;
8      }
9    }
10 }
```

- La strategia implementata è quella di dimezzare la dimensione dell'array solo quando siamo ad un terzo della sua capienza.
- Nota: il controllo a riga 5 sul valore di `tmp` e l'assegnamento a riga 6 sono superflui dato che si tratta di un ridimensionamento in difetto.

Creazione e distruzione di una lista: `list_create()` e `list_delete()`

- La funzione di creazione alloca un array dinamico di lunghezza `N` (costante macro).
- Se l'allocazione fallisce, il campo `elem` punta a `NULL` e il campo `size` è uguale a zero.

```
1 list list_create() {
2   struct list tmp = {0, 0, NULL};
3
4   tmp.elem = (int *)calloc(N,sizeof(int));
5   if(tmp.elem != NULL)
6     tmp.size = N;
7   return tmp;
8 }
```

Creazione e distruzione di una lista: `list_create()` e `list_delete()`

- La funzione di creazione alloca un array dinamico di lunghezza `N` (costante macro).
- Se l'allocazione fallisce, il campo `elem` punta a `NULL` e il campo `size` è uguale a zero.

```
1 list list_create() {
2   struct list tmp = {0, 0, NULL};
3
4   tmp.elem = (int *)calloc(N,sizeof(int));
5   if(tmp.elem != NULL)
6     tmp.size = N;
7   return tmp;
8 }
```

- La funzione di cancellazione degli elementi libera la memoria puntata da `elem` e ricrea una lista vuota richiamando la funzione `list_create()`.

```
1 void list_delete(list *L) {
2   if(L != NULL) {
3     free(L->elem);
4     *L = list_create();
5   }
6 }
```

Funzioni di test: `is_empty()` e `is_inlist()`

- Per determinare se una lista è vuota è sufficiente controllare il valore del campo `last`.
- Per limitare possibili usi impropri della funzione, eseguiamo anche un controllo per verificare che il puntatore `L` passato in input non sia uguale a `NULL`.

```
1 int is_empty(list *L) {
2   return L != NULL && L->last == 0;
3 }
```

Funzioni di test: `is_empty()` e `is_inlist()`

- Per determinare se una lista è vuota è sufficiente controllare il valore del campo `last`.
- Per limitare possibili usi impropri della funzione, eseguiamo anche un controllo per verificare che il puntatore `L` passato in input non sia uguale a `NULL`.

```
1 int is_empty(list *L) {
2   return L != NULL && L->last == 0;
3 }
```

- Per verificare se un elemento è presente o meno nella lista è necessario *accedere* a tutti gli elementi memorizzati nell'array.
- Anche in questo caso eseguiamo un controllo preliminare sul puntatore `L`.

```
1 int is_inlist(list *L, int elem) {
2   if(L != NULL) {
3     unsigned int i;
4     for(i = 0; i < L->last; i++)
5       if(L->elem[i] == elem)
6         return 1;
7   }
8   return 0;
9 }
```

Inserimento in testa: `head_insert()`

- I passi della procedura di inserimento in testa sono sintetizzati come segue.
  - Se i parametri passati non sono consistenti (riga 2), ritorna 1 (inserimento fallito).
  - Altrimenti, se la lista non è vuota (riga 5) crea spazio in testa eseguendo uno shift a destra a partire dalla posizione 0 (riga 6).
  - Salva il valore passato in input in testa alla lista (riga 7).
  - Aggiorna la posizione della prima cella libera (riga 8).
  - Richiama la procedura di estensione (riga 9).
  - Ritorna 0 (inserimento riuscito).

```
1  int head_insert(list *L, int elem) {
2    if(L == NULL || L->elem == NULL || L->size == L->last) {
3      return 1;
4    } else {
5      if(!is_empty(L))
6        list_rshift(L,0);
7      L->elem[0] = elem;
8      L->last++;
9      list_extend(L);
10     return 0;
11   }
12 }
```

- Nota. A riga 2, verifichiamo anche che la lista non sia piena (`L->size == L->last`). Per come abbiamo impostato il tutto, questo potrebbe capitare se precedentemente una operazione di ridimensionamento (`realloc()`) non è andata a buon fine.

Inserimento in coda: `tail_insert()`

- I passi della procedura di inserimento in coda sono sintetizzati come segue.
  - Se i parametri passati non sono consistenti (riga 2), ritorna 1 (inserimento fallito).
  - Altrimenti, salva il valore passato in input in coda alla lista (riga 5).
  - Aggiorna la posizione della prima cella libera (riga 6).
  - Richiama la procedura di estensione (riga 7).
  - Ritorna 0 (inserimento riuscito).

```
1  int tail_insert(list *L, int elem) {
2    if(L == NULL || L->elem == NULL || L->size == L->last) {
3      return 1;
4    } else {
5      L->elem[L->last] = elem;
6      L->last++;
7      list_extend(L);
8      return 0;
9    }
10 }
```

- Nota. Come per la `head_insert()`, a riga 2 verifichiamo anche che la lista non sia già piena (`L->size == L->last`).

Inserimento indicizzato: `indx_insert()`

- I passi della procedura di inserimento indicizzato sono sintetizzati come segue.
  - Se i parametri passati non sono consistenti (riga 2), ritorna 1 (inserimento fallito). In particolare, verifichiamo che l'indice sia compreso tra 0 e n, dove n è il numero di elementi nella lista.
  - Altrimenti, se la lista non è vuota (riga 5) crea spazio nella posizione `i` eseguendo uno shift a destra a partire dalla posizione `i` (riga 6).
  - Salva il valore passato in input nella posizione `i` (riga 7).
  - Aggiorna la posizione della prima cella libera (riga 8).
  - Richiama la procedura di estensione (riga 9).
  - Ritorna 0 (inserimento riuscito).

```
1  int indx_insert(list *L, int elem, unsigned int i) {
2    if(L == NULL || L->elem == NULL || L->size == L->last || i > L->last) {
3      return 1;
4    } else {
5      if(!is_empty(L))
6        list_rshift(L,i);
7      L->elem[i] = elem;
8      L->last++;
9      list_extend(L);
10     return 0;
11   }
12 }
```

- Nota. Anche in questo caso, a riga 2 verifichiamo anche che la lista non sia già piena.

Rimozione della testa: `head_delete()`

- I passi della procedura di rimozione in testa sono sintetizzati come segue.
  - Se i parametri passati non sono consistenti (riga 2), ritorna 1 (rimozione fallita).
  - Altrimenti, sposta con uno shift a sinistra gli elementi della lista, a partire dalla posizione 1 (riga 5).
  - Aggiorna la posizione della prima cella libera (riga 6).
  - Richiama la procedura di riduzione (riga 7).
  - Ritorna 0 (rimozione riuscita).

```
1  int head_delete(list *L) {
2    if(L == NULL || L->elem == NULL || is_empty(L)) {
3      return 1;
4    } else {
5      list_lshift(L,1);
6      L->last--;
7      list_reduce(L);
8      return 0;
9    }
10 }
```

Rimozione della coda: `tail_delete()`

- I passi della procedura di rimozione in coda sono sintetizzati come segue.
  - Se i parametri passati non sono consistenti (riga 2), ritorna 1 (rimozione fallita).
  - Altrimenti, aggiorna la posizione della prima cella libera (riga 5).
  - Richiama la procedura di riduzione (riga 6).
  - Ritorna 0 (rimozione riuscita).

```
1 int tail_delete(list *L) {
2   if(L == NULL || L->elem == NULL || is_empty(L)) {
3     return 1;
4   } else {
5     L->last--;
6     list_reduce(L);
7     return 0;
8   }
9 }
```

Rimozione indicizzata: `indx_delete()`

- I passi della procedura di rimozione indicizzato sono sintetizzati come segue.
  - Se i parametri passati non sono consistenti (riga 2), ritorna 1 (rimozione fallita). In particolare, verifichiamo che l'indice sia compreso tra 0 e $n-1$, dove n è il numero di elementi nella lista.
  - Altrimenti, sposta con uno shift a sinistra gli elementi della lista, a partire dalla posizione `i+1` (riga 5).
  - Aggiorna la posizione della prima cella libera (riga 6).
  - Richiama la procedura di riduzione (riga 7).
  - Ritorna 0 (inserimento riuscito).

```
1  int indx_delete(list *L, unsigned int i) {
2    if(L == NULL || L->elem == NULL || i >= L->last) {
3      return 1;
4    } else {
5      list_lshift(L,i+1);
6      L->last--;
7      list_reduce(L);
8      return 0;
9    }
10 }
```

Funzioni di selezione: `head_select()`, `tail_select()`, `indx_select()`

- Le funzioni di selezione sono molto semplici, dato che lavoriamo su un array.
- Il comportamento è non definito se i parametri in input non sono consistenti (riga 2): passaggio di lista vuota, indice oltre i limiti della lista, ecc.
- Nell'implementazione mostrata, le funzioni ritornano 0 (riga 3) se ci sono errori.

```
1 int head_select(list *L) {
2   if(L == NULL || L->elem == NULL || is_empty(L))
3     return 0;
4   else
5     return L->elem[0];
6 }
```

```
1 int tail_select(list *L) {
2   if(L == NULL || L->elem == NULL || is_empty(L))
3     return 0;
4   else
5     return L->elem[L->last-1];
6 }
```

```
1 int indx_select(list *L, unsigned int i) {
2   if(L == NULL || L->elem == NULL || i >= L->last)
3     return 0;
4   else
5     return L->elem[i];
6 }
```

Pro e contro della rappresentazione con array

- **Pro**.
  - Implementazione estremamente semplice grazie al fatto che il linguaggio C ci mette a disposizione diversi costrutti sintattici per poter lavorare con array.
  - L'accesso indicizzato ad un elemento della lista è molto efficiente: riusciamo ad accedere all'`i`-esimo elemento con una sola istruzione.

Pro e contro della rappresentazione con array

- **Pro**.
  - Implementazione estremamente semplice grazie al fatto che il linguaggio C ci mette a disposizione diversi costrutti sintattici per poter lavorare con array.
  - L'accesso indicizzato ad un elemento della lista è molto efficiente: riusciamo ad accedere all'`i`-esimo elemento con una sola istruzione.
- **Contro**.
  - Ad eccezione delle operazioni di inserimento e rimozione in coda, le altre operazioni di inserimento e rimozione richiedono lo spostamento di un certo numero di elementi nella lista: gestione poco efficiente dal punto di vista computazionale.
  - Il ridimensionamento tramite la funzione di libreria `realloc()` non permette di gestire in modo ottimale la memoria sul calcolatore: la `realloc()` potrebbe fallire nel trovare sull'heap un blocco di memoria contiguo sufficientemente grande, anche quando è disponibile molta memoria (frammentata).
  - Il ridimensionamento tramite la `realloc()` non è efficiente dal punto di vista computazionale: la ri-allocazione potrebbe comportare l'operazione di copia dell'intera lista in una nuova locazione di memoria.

Liste concatenate

- Una lista concatenata è una struttura dinamica: ogni elemento della lista (nodo) contiene, oltre alle informazioni da memorizzare, un puntatore al nodo successivo.
- La rappresentazione con liste concatenate è uno degli approcci più comuni per l'implementazione del tipo di dato astratto lista.

Liste concatenate

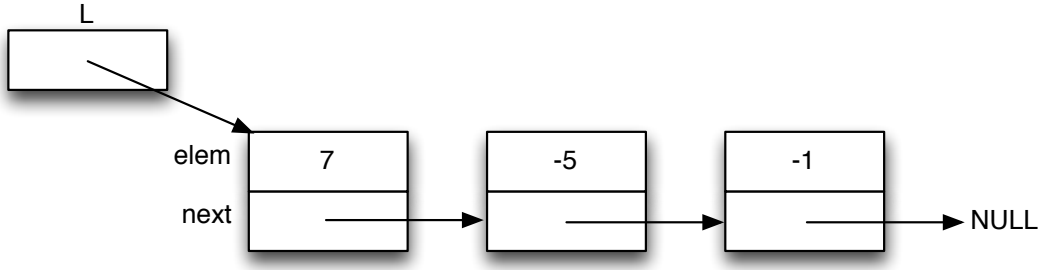
- Una lista concatenata è una struttura dinamica: ogni elemento della lista (nodo) contiene, oltre alle informazioni da memorizzare, un puntatore al nodo successivo.
- La rappresentazione con liste concatenate è uno degli approcci più comuni per l'implementazione del tipo di dato astratto lista.
- A differenza della rappresentazione tramite array, l'inserimento o rimozione di elementi non comporta la riallocazione o riorganizzazione dell'intera struttura:
  - i nodi della lista non sono necessariamente contigui in memoria,
  - la rimozione o inserimento di un nodo non comporta lo spostamento di altri elementi ma semplicemente l'aggiornamento di puntatori.

Liste concatenate

- Una lista concatenata è una struttura dinamica: ogni elemento della lista (nodo) contiene, oltre alle informazioni da memorizzare, un puntatore al nodo successivo.
- La rappresentazione con liste concatenate è uno degli approcci più comuni per l'implementazione del tipo di dato astratto lista.
- A differenza della rappresentazione tramite array, l'inserimento o rimozione di elementi non comporta la riallocazione o riorganizzazione dell'intera struttura:
  - i nodi della lista non sono necessariamente contigui in memoria,
  - la rimozione o inserimento di un nodo non comporta lo spostamento di altri elementi ma semplicemente l'aggiornamento di puntatori.
- D'altra parte, a differenza degli array, l'accesso ad un elemento della lista non può essere diretto ma richiede l'attraversamento sequenziale di tutti o di un certo numero di nodi all'interno della lista.

Il tipo di dato lista concatenata

- Possiamo rappresentare il tipo di dato `list` come **puntatore a struttura** contente:
  - Un campo (`elem`) di tipo in `int`
  - Un campo puntatore (`next`) ad una struttura dello stesso tipo.




```
1 struct node {
2   int elem;
3   struct node *next;
4 };
5
6 typedef struct node *list;
```

- La struttura `node` rappresenta un nodo di una lista dinamica.
- Da un nodo è possibile accedere al nodo successivo nella lista.
- Le funzioni di libreria hanno come argomento formale un **puntatore al puntatore** al primo nodo della lista.

Untility di libreria per liste concatenate

- Facciamo uso di utility di libreria per semplificare l'implementazione delle funzioni visibili all'esterno.

```
1  /*
2   * Alloca un nodo con valore elem.
3   *
4   * Ritorna un puntatore al nodo allocato o NULL
5   * se l'allocazione non ha successo.
6   */
7  static struct node *node_alloc(int elem);
8  /*
9   * Ritorna un puntatore all'i-esimo nodo dopo il nodo
10  * puntato da L. Ritorna  NULL se tale nodo non esiste.
11  */
12 static struct node *node_search(struct node *L, unsigned int i);
13 /*
14  * Rimuove il nodo successivo a L.
15  *
16  * Ritorna 0 se l'operazione è stata eseguita.
17  */
18 static int node_delete(struct node *L);
19 /*
20  * Inserisce un nodo con valore elem dopo il nodo L.
21  *
22  * Ritorna 0 se l'operazione è stata eseguita.
23  */
24 static int node_insert(struct node *L, int elem);
```

Utility per la creazione di un nodo: `node_alloc()`

- I nodi della struttura lista concatenata devono essere allocati dinamicamente.
- L'utility `node_alloc()` si occupa di allocare dinamicamente ed inizializzare un nodo.
  - Setta il campo `elem` con il valore passato in input (riga 5).
  - Inizializza il campo `next` a `NULL` (riga 6).
- Restituisce un puntatore al nodo allocato, o NULL se l'allocazione fallisce (riga 8)

```
1 static struct node *node_alloc(int elem) {
2   struct node *tmp = (struct node *)malloc(sizeof(struct node));
3
4   if(tmp != NULL) {
5     tmp->elem = elem;
6     tmp->next = NULL;
7   }
8   return tmp;
9 }
```

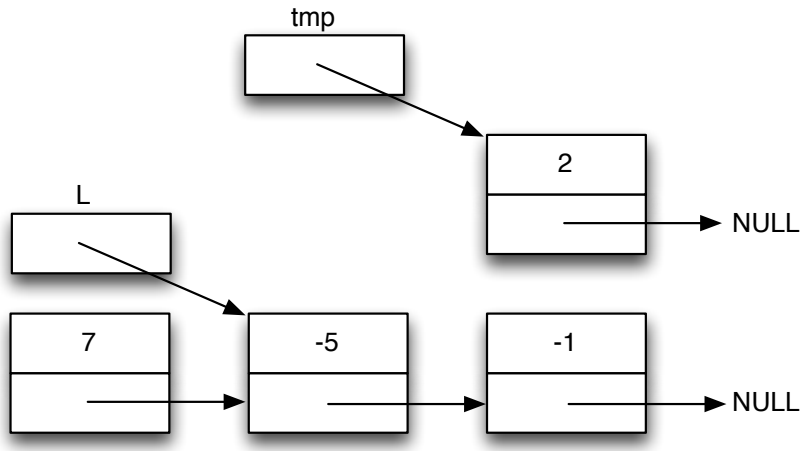
Utility per la ricerca di un nodo: `node_search()`

- In una lista concatenata i singoli nodi non sono necessariamente memorizzati in locazioni di memoria contigue.
- Non è possibile quindi indicizzare un nodo calcolando l'offset rispetto ad un nodo di partenza (cosa invece possibile con array).
- L'unico modo per poter accedere ad un nodo è seguire la catena di puntatori `next`.
- La utility `node_search()` ci permette di ottenere il puntatore ad un nodo che si trova ad una certa distanza *logica* (parametro `i`) rispetto ad un nodo di partenza (nodo puntato dal puntatore `L`).
- Da notare come l'implementazione modifica al suo interno il valore degli argomenti `L` ed `i`, passati per valore (modifica solo il valore delle copie locali).

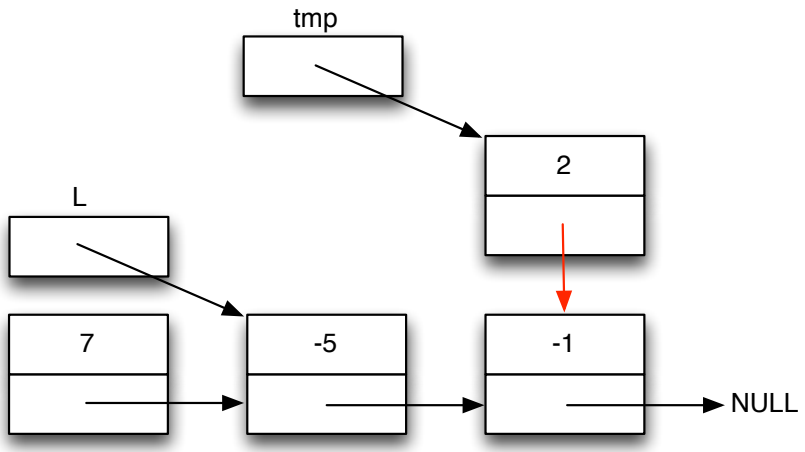
```
1 static struct node *node_search(struct node *L, unsigned int i) {
2   while(i-- > 0 && L != NULL)
3     L = L->next;
4   return L;
5 }
```

Utility per l'inserimento di un nodo 1/2: `node_insert()`

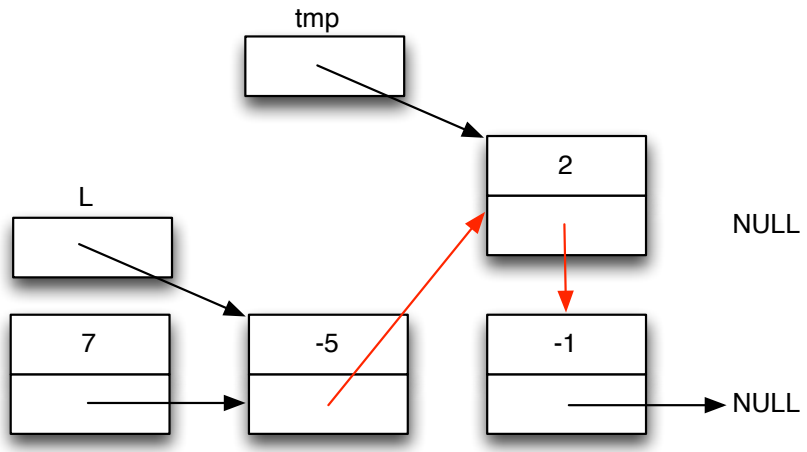
- L'operazione di inserimento di un nuovo nodo in una lista concatenata richiede **necessariamente** la conoscenza del nodo che deve precede il nuovo nodo da inserire.
- Non possiamo gestire con l'utility `node_insert()` l'inserimento in testa alla lista.



A) Creazione del nuovo nodo da inserire (`tmp`)



B) Aggiornamento campo `next` del nuovo nodo



C) Aggiornamento campo `next` del nodo passato come argomento

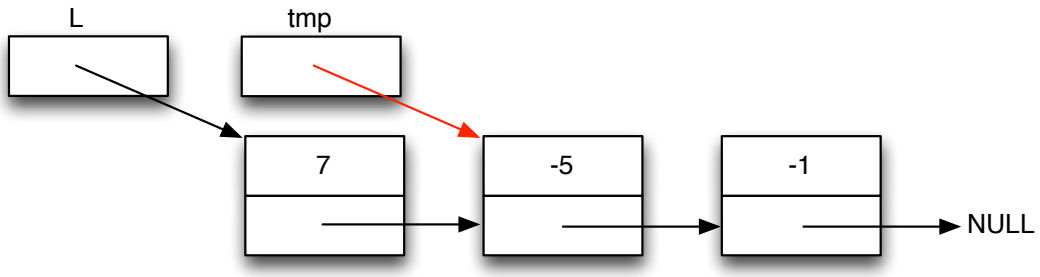
Utility per l'inserimento di un nodo 2/2: `node_insert()`

- L'implementazione di `node_insert()` ritorna 1 (inserimento fallito) se il puntatore `L` passato come parametro è `NULL` (riga 2), oppure se l'allocazione del nuovo nodo non è riuscita (riga 11).
- Se l'allocazione del nuovo nodo è riuscita (riga 7), l'unica operazione necessaria è l'aggiornamento del puntatore `next` del nuovo nodo (riga 8) e del nodo passato in input (riga 9).

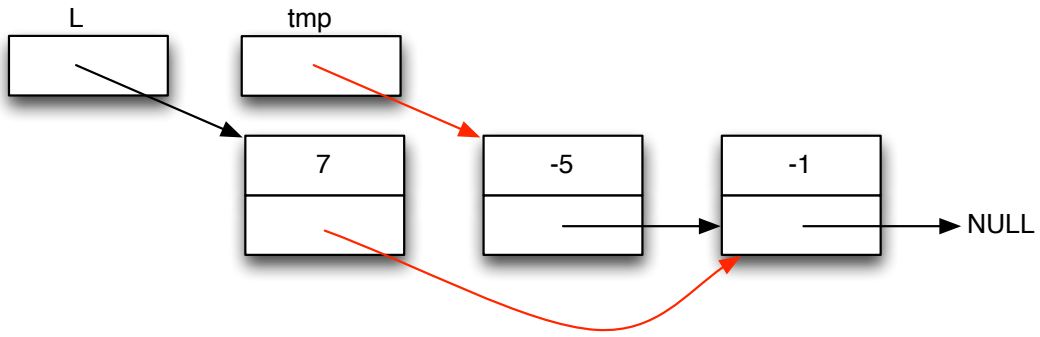
```
1  static int node_insert(struct node *L, int elem) {
2    if(L == NULL) {
3      return 1;
4    } else {
5      struct node *tmp = node_alloc(elem);
6
7      if(tmp != NULL) {
8        tmp->next = L->next;
9        L->next   = tmp;
10     }
11     return tmp == NULL;
12   }
13 }
```

Utility per la rimozione di un nodo 1/2: `node_delete()`

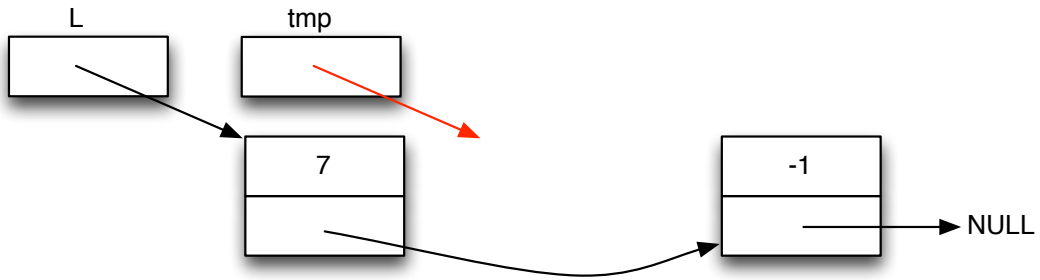
- Anche per l'operazione di rimozione di un nodo in una lista concatenata è **necessario** conoscere il nodo che precede il nuovo da rimuovere.
- Non possiamo gestire la rimozione in testa con l'utility `node_delete()`.



A) Puntatore al nodo da rimuovere (`tmp`)



B) Aggiornamento del campo `next` del nodo precedente



C) Deallocazione memoria del nodo da rimuovere

Utility per la rimozione di un nodo 2/2: `node_delete()`

- L'implementazione di `node_delete()` ritorna 1 (rimozione fallita) se il puntatore `L` passato come parametro è `NULL` oppure se non esiste un nodo successivo al nodo puntato da `L` (riga 2).
- Se i controlli di consistenza sono superati, le uniche operazioni da eseguire sono le seguenti.
  - Individuazione del nodo da rimuovere (riga 5).
  - Aggiornamento del campo `next` del nodo puntato da `L` (riga 6).
  - Deallocazione del nodo da rimuovere (riga 7).

```
1  static int node_delete(struct node *L) {
2    if(L==NULL || L->next==NULL) {
3      return 1;
4    } else {
5      struct node *tmp = L->next;
6      L->next = tmp->next;
7      free(tmp);
8      return 0;
9    }
10 }
```

Creazione e distruzione di una lista: `list_create()` e `list_delete()`

- La funzione di creazione restituisce semplicemente la costante `NULL`: una lista vuota non contiene nodi.

```
1 list list_create() {
2   return NULL;
3 }
```

Creazione e distruzione di una lista: `list_create()` e `list_delete()`

- La funzione di creazione restituisce semplicemente la costante `NULL`: una lista vuota non contiene nodi.

```
1 list list_create() {
2   return NULL;
3 }
```

- La funzione di distruzione della lista deve scorrere tutta la lista e deallocarne i nodi. La funzione si occupa di settare la testa della lista a `NULL` (lista vuota).
- Esistono diverse implementazioni possibili: mostriamo un'implementazione ricorsiva.

```
1 void list_delete(list *L) {
2   if(L != NULL && *L != NULL) {
3     list_delete(&(*L)->next);
4     free(*L);
5     *L=NULL;
6   }
7 }
```

Funzioni di test: `is_empty()` e `is_inlist()`

- Per determinare se una lista è vuota è sufficiente controllare che il puntatore passato punti a `NULL`.

```
1 int is_empty(list *L) {
2   return (L == NULL) || (*L == NULL);
3 }
```

Funzioni di test: `is_empty()` e `is_inlist()`

- Per determinare se una lista è vuota è sufficiente controllare che il puntatore passato punti a `NULL`.

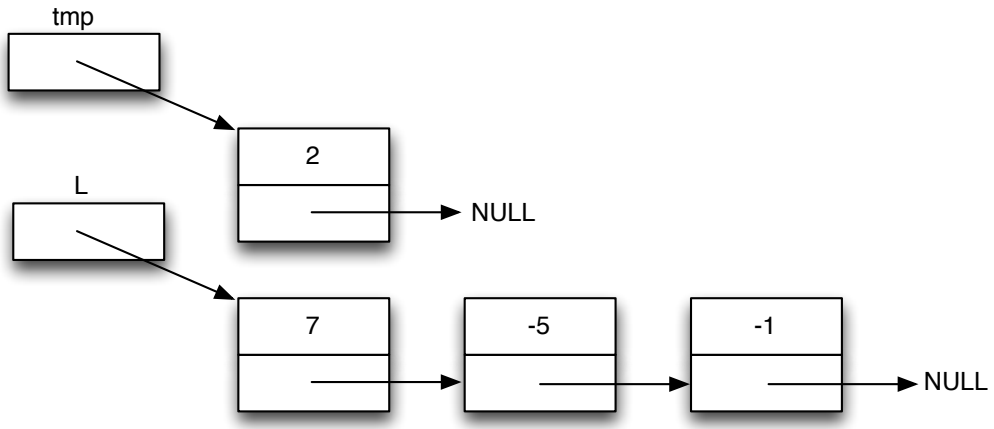
```
1 int is_empty(list *L) {
2   return (L == NULL) || (*L == NULL);
3 }
```

- Per verificare se un elemento è presente o meno nella lista è necessario *scorrere* tutta la lista seguendo la catena di puntatori `next` (riga 7).
- La funzione restituisce 0 (elemento non presente) se la lista è vuota (riga 2) oppure se dopo aver effettuato la ricerca (riga 7), il valore del puntatore `tmp` è `NULL` (riga 10).

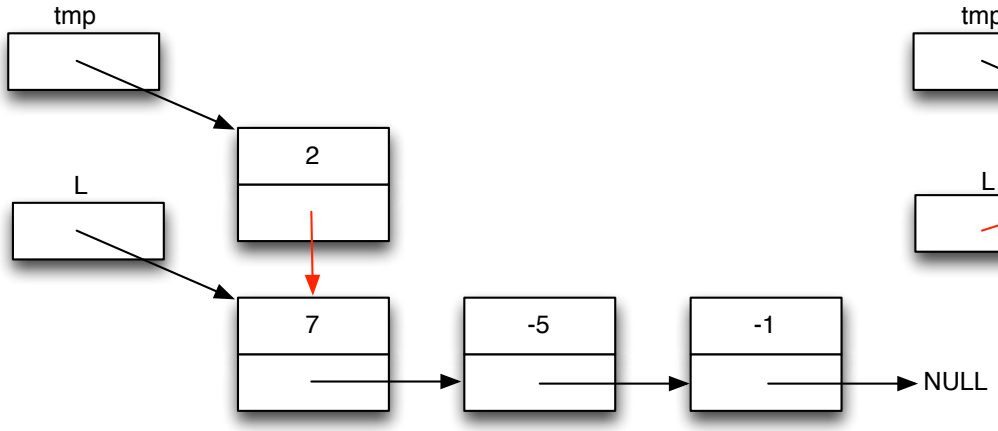
```
1  int is_inlist(list *L, int elem) {
2    if(is_empty(L)) {
3      return 0;
4    } else {
5      struct node *tmp = *L;
6
7      while(tmp != NULL && tmp->elem != elem)
8        tmp = tmp->next;
9
10     return tmp != NULL;
11   }
12 }
```

Inserimento in testa 1/2: `head_insert()`

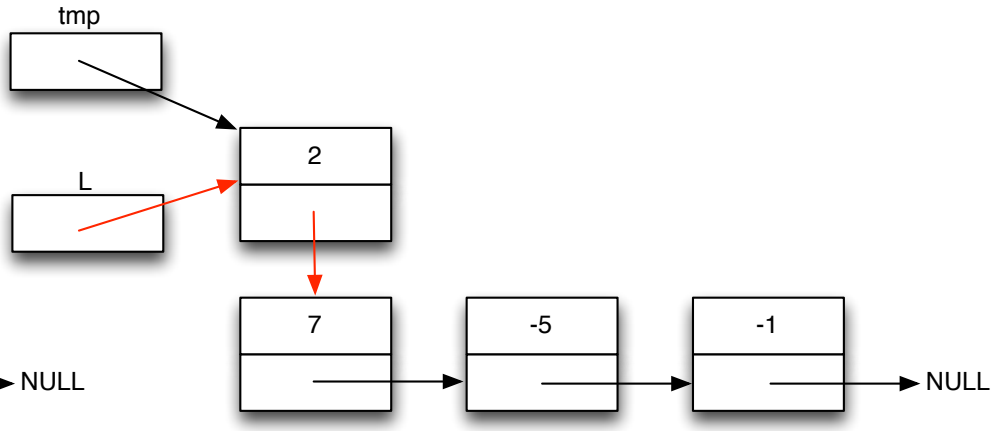
- L'inserimento in testa alla lista deve essere gestita in modo speciale dato che richiede la modifica del puntatore alla testa della lista.



A) Creazione del nuovo nodo da inserire (`tmp`)



B) Aggiornamento campo `next` del nuovo nodo



C) Aggiornamento del puntatore alla testa della lista

Inserimento in testa 2/2: `head_insert()`

- Data la definizione del tipo `list` come `struct node*`, i prototipi delle funzioni di libreria prendono in input l'indirizzo di memoria del puntatore alla testa della lista.
- Le operazioni effettuate dalla funzione `head_insert()` sono le seguenti.
  - Allocazione del nuovo nodo (riga 5).
  - Aggiornamento del campo `next` del nuovo nodo affinché punti alla testa della lista (riga 8). Se la lista è vuota allora punta `NULL`.
  - Aggiornamento del puntatore alla testa della lista in modo che punti al nuovo nodo (riga 9).
  - La funzione ritorna 1 (inserimento non riuscito) se l'indirizzo `L` passato è `NULL` (riga 2) oppure se l'allocazione del nuovo nodo fallisce (riga 11).

```
1  int head_insert(list *L, int elem) {
2    if(L == NULL) {
3      return 1;
4    } else {
5      struct node *tmp = node_alloc(elem);
6
7      if(tmp != NULL) {
8        tmp->next = *L;
9        *L        = tmp;
10     }
11     return tmp == NULL;
12   }
13 }
```

Inserimento in coda: `tail_insert()`

- La funzione `tail_insert()` è quasi interamente definita in termini di funzioni sviluppate precedentemente.
  - Se la lista è vuota (riga 4), esegue un inserimento in testa (riga 5).
  - Se la lista contiene almeno un elemento (blocco else a riga 6), individua l'ultimo nodo nella lista (riga 9) e richiama la `node_insert()` per effettuare l'inserimento del nuovo nodo (riga 11).
  - Ritorna 1 (inserimento non riuscito) se l'indirizzo `L` passato è `NULL` (riga 2) oppure se l'allocazione del nuovo nodo fallisce (riga 11).

```
1  int tail_insert(list *L, int elem) {
2    if(L == NULL) {
3      return 1;
4    } else if(is_empty(L)) {
5      return head_insert(L,elem);
6    } else  {
7      struct node *tmp = *L;
8
9      while(tmp->next != NULL)
10       tmp = tmp->next;
11     return node_insert(tmp,elem);
12   }
13 }
```

Inserimento indicizzato: `indx_insert()`

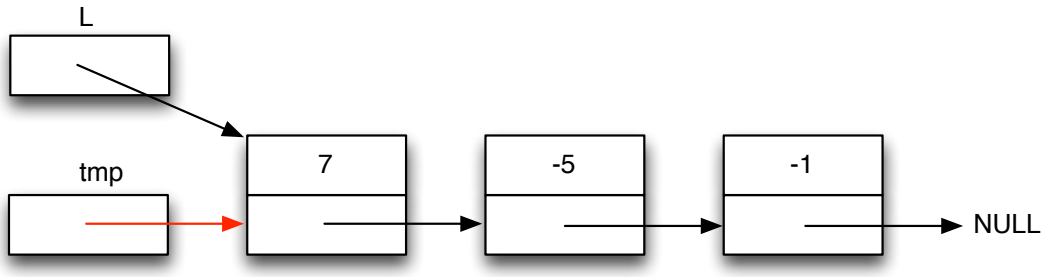
- La funzione `indx_insert()` è interamente definita in termini di funzioni sviluppate precedentemente.
  - Se viene richiesto un inserimento nella posizione 0 (riga 2), viene eseguito un inserimento in testa (riga 3).
  - Diversamente, viene effettuato un inserimento nella `i`-esima posizione, passando alla `node_insert()` il puntatore all `(i-1)`-esimo nodo, individuato con l'utility `node_search()` (riga 5).

```
1 int indx_insert(list *L, int elem, unsigned int i) {
2   if(i == 0)
3     return head_insert(L,elem);
4   else
5     return node_insert(node_search(*L,i-1),elem);
6 }
```

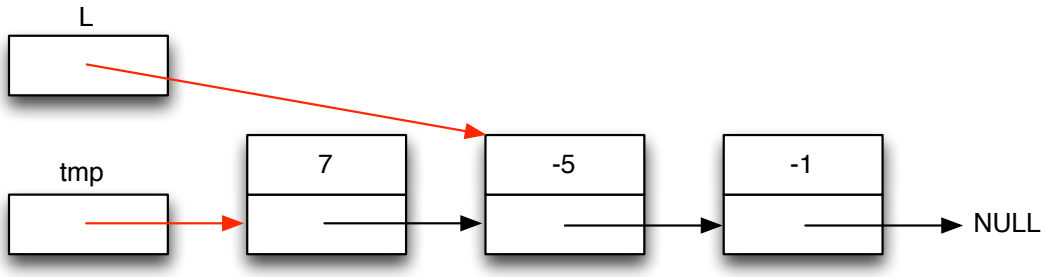
- Da notare che l'operazione a riga 5 è sempre ben definita:
  - se l'`i-1` posizione non è nella lista allora `node_search()` ritorna `NULL`,
  - l'utility `node_insert()` con parametro `NULL` non esegue nessuna operazione e ritorna 1 (inserimento non riuscito).

Rimozione della testa 1/2: `head_delete()`

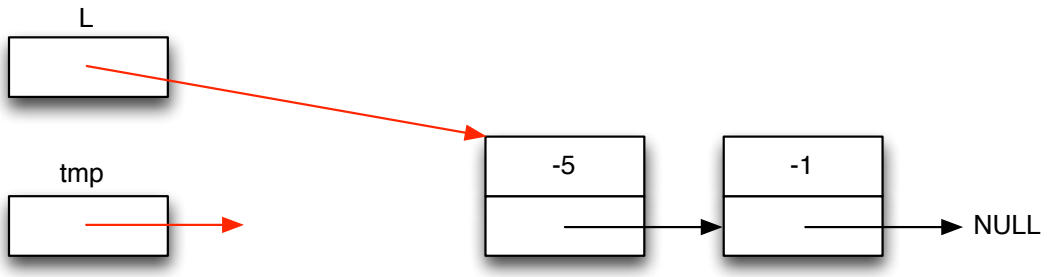
- Così come per l'inserimento in testa, la rimozione della testa della lista deve essere gestita in modo speciale dato che richiede la modifica del puntatore alla testa della lista.



A) Nuovo puntatore alla testa della lista (`tmp`)



B) Aggiornamento puntatore alla testa della lista



C) Deallocazione della vecchia testa della lista

Rimozione della testa 2/2: `head_delete()`

- Passi della funzione `head_delete()`.
  - Dichiara un nuovo puntatore alla testa attuale (riga 5).
  - Aggiorna il puntatore alla testa della lista in modo che adesso punti al nodo successivo (riga 7). Il nodo successivo potrebbe anche essere `NULL` (lista vuota).
  - Dealloca la vecchia testa della lista (riga 8).
  - La funzione ritorna 1 (rimozione non riuscita) se la lista è vuota (riga 2).

```
1  int head_delete(list *L) {
2    if(is_empty(L)) {
3      return 1;
4    } else {
5      struct node *tmp = *L;
6
7      *L = (*L)->next;
8      free(tmp);
9      return 0;
10   }
11 }
```

Rimozione della coda: `tail_delete()`

- La funzione `tail_delete()` è quasi interamente definita in termini di funzioni sviluppate precedentemente.
  - Se la lista contiene un solo elemento (riga 4), esegue una rimozione della testa (riga 5).
  - Se la lista contiene almeno due elementi elementi (blocco else a riga 6), individua il penultimo nodo nella lista (riga 9) e richiama la `node_delete()` per effettuare la rimozione del nodo successivo al penultimo (riga 11).
  - Ritorna 1 (rimozione non riuscita) se la lista è vuota (riga 2).

```
1  int tail_delete(list *L) {
2    if(is_empty(L)) {
3      return 1;
4    } else if((*L)->next == NULL) {
5      return head_delete(L);
6    } else {
7      struct node *tmp=*L;
8
9      while(tmp->next->next != NULL)
10       tmp = tmp->next;
11     return node_delete(tmp);
12   }
13 }
```

- Le chiamate a riga 5 e 12 ritorneranno 0, dato che siamo sicuri che un nodo da rimuovere esiste.

Rimozione indicizzata: `indx_delete()`

- La funzione `indx_delete()` è interamente definita in termini di funzioni sviluppate precedentemente.
  - Se viene richiesto una rimozione del nodo in posizione 0 (riga 2), viene eseguita una rimozione della testa testa (riga 3).
  - Diversamente, viene effettuato la rimozione dell'`i`-esimo nodo, passando alla `node_delete()` il puntatore all `(i-1)`-esimo nodo, individuato con l'utility di libreria `node_search()` (riga 5).

```
1 int indx_delete(list *L, unsigned int i) {
2   if(i == 0)
3     return head_delete(L);
4   else
5     return node_delete(node_search(*L,i-1));
6 }
```

- Anche in questo caso, come per la funzione `inde_insert()`, l'operazione a riga 5 è sempre ben definita.

Funzioni di selezione: `head_select()`, `tail_select()`, `indx_select()`

- Le funzioni di selezione su liste concatenate non possono accedere direttamente ai nodi utilizzando indici, così come sulla implementazione con array: è necessario scorrere la lista seguendo la catena di puntatori.
- Il comportamento delle implementazioni presentate (valori di ritorno) è consistente con il comportamento delle implementazioni sulla struttura lista con array.

```
1  int head_select(list *L) {
2    if(is_empty(L))
3      return 0;
4    else
5      return (*L)->elem;
6  }
```

```
1  int tail_select(list *L) {
2    if(is_empty(L)) {
3      return 0;
4    } else {
5      struct node *tmp=*L;
6
7      while(tmp->next != NULL)
8        tmp = tmp->next;
9      return tmp->elem;
10   }
11 }
```

```
1  int indx_select(list *L, unsigned int i) {
2    if(is_empty(L)) {
3      return 0;
4    } else {
5      struct node *tmp = node_search(*L,i);
6      if(tmp == NULL)
7        return 0;
8      else
9        return tmp->elem;
10   }
11 }
```

Pro e contro della rappresentazione con liste concatenate

- **Pro**.
  - Permettono di utilizzare in modo efficiente la memoria in fase di esecuzione: viene allocata solo la memoria richiesta per rappresentare il contenuto attuale della lista.
  - Le operazioni di inserimento e rimozione sono computazionalmente efficienti: non è necessario riorganizzare completamente la struttura dati.

Pro e contro della rappresentazione con liste concatenate

- **Pro**.
  - Permettono di utilizzare in modo efficiente la memoria in fase di esecuzione: viene allocata solo la memoria richiesta per rappresentare il contenuto attuale della lista.
  - Le operazioni di inserimento e rimozione sono computazionalmente efficienti: non è necessario riorganizzare completamente la struttura dati.
- **Contro**.
  - Ad eccezione dell'elemento in testa, non è possibile accedere direttamente ad un elemento della lista. L'accesso all'elemento in coda richiede l'attraversamento dell'intera lista.
  - E' possibile attraversare la lista in una sola direzione: dalla testa verso la coda. Operazioni che richiedano l'attraversamento dalla coda verso la testa sono complesse e generalmente computazionalmente costose.
  - Comporta un *overhead* di memoria, dovuto alla necessità di memorizzare un puntatore per ogni elemento della lista.

Liste doppiamente concatenate

- Le liste doppiamente concatenate *estendono* la rappresentazione delle liste concatenate, introducendo un puntatore al nodo precedente.

Liste doppiamente concatenate

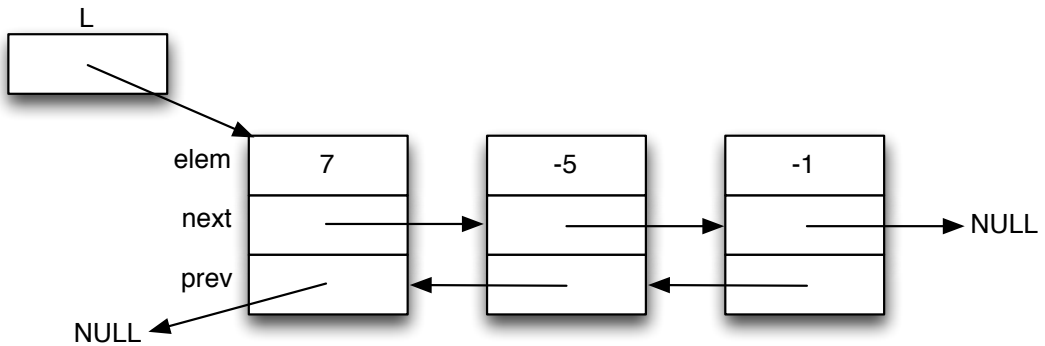
- Le liste doppiamente concatenate *estendono* la rappresentazione delle liste concatenate, introducendo un puntatore al nodo precedente.
- A differenza delle liste concatenate semplici, le liste doppiamente concatenate possono essere attraversate facilmente in entrambe le direzioni.
- Le implementazioni di alcune funzioni sono semplificate dall'introduzione del nuovo puntatore.

Liste doppiamente concatenate

- Le liste doppiamente concatenate *estendono* la rappresentazione delle liste concatenate, introducendo un puntatore al nodo precedente.
- A differenza delle liste concatenate semplici, le liste doppiamente concatenate possono essere attraversate facilmente in entrambe le direzioni.
- Le implementazioni di alcune funzioni sono semplificate dall'introduzione del nuovo puntatore.
- Aggiungono un ulteriore overhead di memoria alla struttura dati: è necessario memorizzare due puntatori distinti in ogni nodo.
- Non offrono vantaggi rispetto alle liste concatenate semplici relativamente all'accesso alla coda della lista.

Il tipo di dato lista doppiamente concatenata

- Possiamo rappresentare il tipo di dato `list` come **puntatore a struttura** contente:
  - Un campo (`elem`) di tipo in `int`
  - Un campo puntatore (`next`) ad una struttura dello stesso tipo.
  - Un campo puntatore (`prev`) ad una struttura dello stesso tipo.




```
1 struct node {
2   int elem;
3   struct node *next;
4   struct node *prev;
5 };
6 typedef struct node *list;
```

- Da un nodo è possibile accedere sial al nodo successivo (campo `next`) che al nodo precedente (campo `prev`) nella lista .
- Le funzioni di libreria hanno come argomento formale un **puntatore al puntatore** al primo nodo della lista.

Utility di libreria per liste concatenate

- Le utility di libreria per liste doppiamente concatenate sono le stesse viste per liste concatenate: cambia sostanzialmente a livello semantico solo la `node_delete()`.

```
1  /*
2   * Alloca un nodo con valore elem.
3   *
4   * Ritorna un puntatore al nodo allocato o NULL
5   * se l'allocazione non ha successo.
6   */
7  static struct node *node_alloc(int);
8  /*
9   * Ritorna un puntatore all'i-esimo nodo dopo il nodo
10  * puntato da L. Ritorna  NULL se tale nodo non esiste.
11  */
12 static struct node *node_search(struct node *L, unsigned int i);
13 /*
14  * Rimuove il nodo L.
15  *
16  * Ritorna 0 se l'operazione è stata eseguita.
17  */
18 static int node_delete(struct node *L);
19 /*
20  * Inserisce un nodo con valore elem dopo il nodo L.
21  *
22  * Ritorna 0 se l'operazione è stata eseguita.
23  */
24 static int node_insert(struct node *L, int elem);
```

Utility per la creazione e ricerca: `node_alloc()` e `node_search()`

- La utility `node_alloc()` è essenzialmente implementa come visto per liste concatenate.
- Il nodo di una lista concatenata contiene il puntatore al nodo precedente, oltre al puntatore al nodo successivo: è necessario inizializzare a `NULL` entrambi i puntatori nella struttura `node`.

```
1  static struct node *node_alloc(int elem) {
2    struct node *tmp = (struct node *)malloc(sizeof(struct node));
3
4    if(tmp != NULL) {
5      tmp->elem = elem;
6      tmp->next = NULL;
7      tmp->prev = NULL;
8    }
9    return tmp;
10 }
```

Utility per la creazione e ricerca: `node_alloc()` e `node_search()`

- La utility `node_alloc()` è essenzialmente implementa come visto per liste concatenate.
- Il nodo di una lista concatenata contiene il puntatore al nodo precedente, oltre al puntatore al nodo successivo: è necessario inizializzare a `NULL` entrambi i puntatori nella struttura `node`.

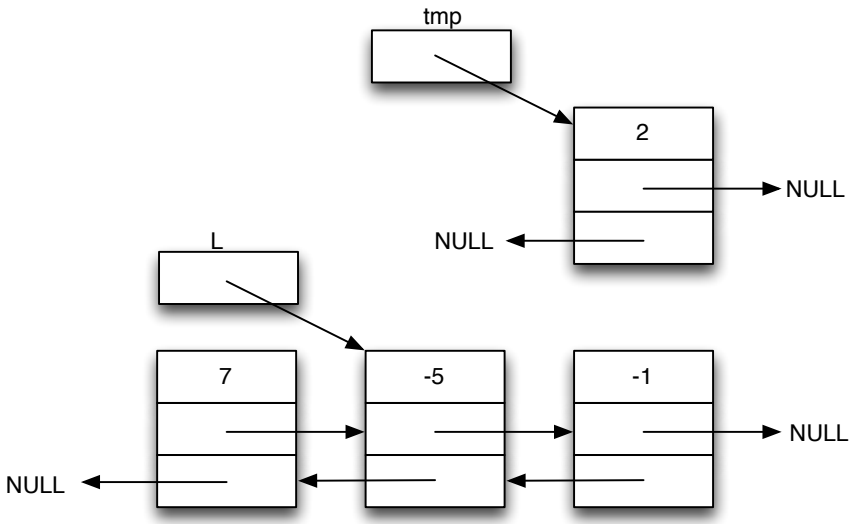
```
1  static struct node *node_alloc(int elem) {
2    struct node *tmp = (struct node *)malloc(sizeof(struct node));
3
4    if(tmp != NULL) {
5      tmp->elem = elem;
6      tmp->next = NULL;
7      tmp->prev = NULL;
8    }
9    return tmp;
10 }
```

- L'implementazione della funzione per la ricerca di un nodo dato il suo indice non cambia rispetto alla versione vista per liste concatenate.

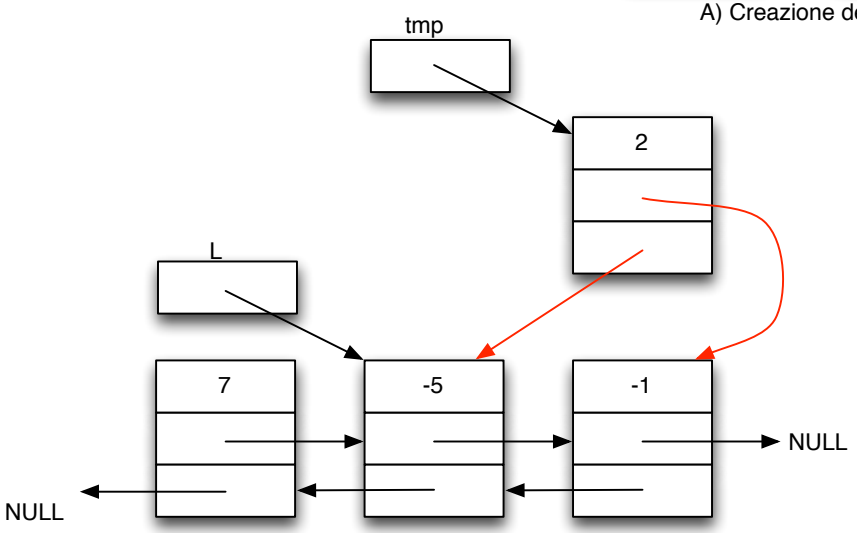
```
1  static struct node *node_search(struct node *L, unsigned int i) {
2    while(i-- > 0 && L != NULL)
3      L = L->next;
4    return L;
5  }
```

Utility per l'inserimento di un nodo 1/2: `node_insert()`

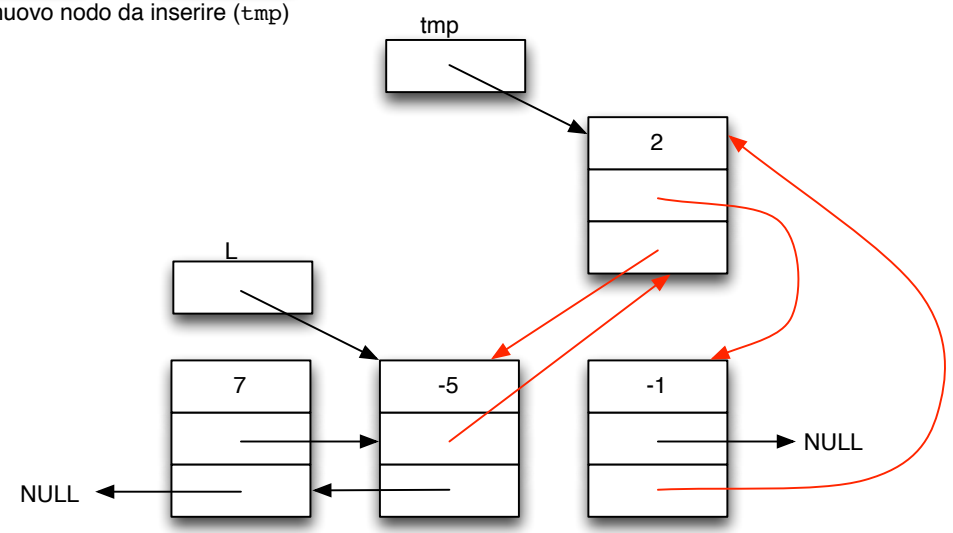
- Nell'implementazione della funzione per l'inserimento di un nodo dobbiamo adesso gestire anche l'aggiornamento del puntatore `prev`.



A) Creazione del nuovo nodo da inserire (`tmp`)



B) Aggiornamento dei campo `next` e `prev` del nuovo nodo



C) Aggiornamento campo `next` del nodo passato come argomento e del campo `prev` del nodo successivo

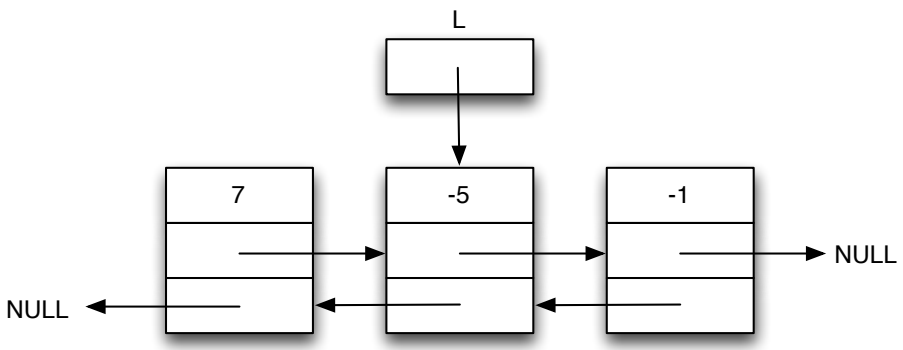
Utility per l'inserimento di un nodo 2/2: `node_insert()`

- L'implementazione di `node_insert()` ritorna 1 (inserimento fallito) se il puntatore `L` passato come parametro è `NULL` (riga 2), oppure se l'allocazione del nuovo nodo non è riuscita (riga 14).
- Se l'allocazione del nuovo nodo è riuscita (riga 7), aggiorniamo i puntatori `next` (riga 8) e `prev` (riga 9) del nuovo nodo oltre al puntatore `next` del nodo passato in input `riga 10`.
- Se il nuovo nodo non è stato inserito in coda (riga 11), allora viene aggiornato anche il puntatore `prev` del nodo successivo, in modo che punti al nuovo nodo (riga 12).

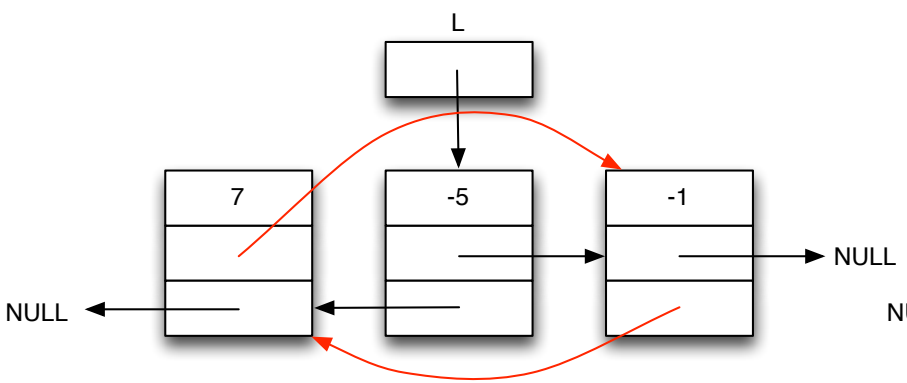
```
1  static int node_insert(struct node *L, int elem) {
2    if(L == NULL) {
3      return 1;
4    } else {
5      struct node *tmp = node_alloc(elem);
6
7      if(tmp != NULL) {
8        tmp->next = L->next;
9        tmp->prev = L;
10       L->next   = tmp;
11       if(tmp->next != NULL)
12         tmp->next->prev = tmp;
13     }
14     return tmp == NULL;
15   }
16 }
```

Utility per la rimozione di un nodo 1/2: `node_delete()`

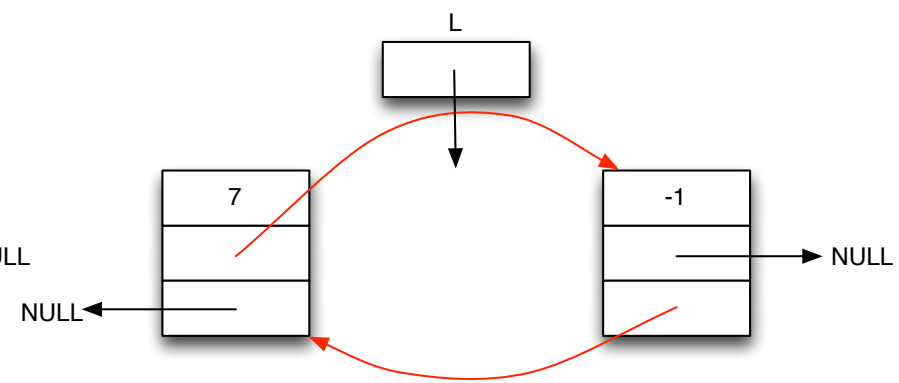
- La funzione di utility per la rimozione di un nodo in una lista doppiamente concatenata è utilizzata in modo alquanto differente rispetto alla sua controparte per liste concatenate.
- Grazie all'utilizzo di puntatori in avanti ed indietro siamo in grado di gestire la rimozione di un nodo passando alla funzione direttamente un puntatore al nodo da rimuovere.



A) Puntatore al nodo da rimuovere (`L`)



B) Aggiornamento del campo `next` del nodo precedente e del campo `prev` del nodo successivo



C) Deallocazione memoria del nodo da rimuovere

Utility per la rimozione di un nodo 2/2: `node_delete()`

- L'implementazione della funzione `node_delete()` è alquanto semplice.
  - Ritorna 1 (rimozione non riuscita) se il il puntatore `L` passato come parametro è `NULL` (riga 2).
  - Per poter aggiornare il puntatore `prev` del nodo successivo (riga 6) è necessario verificare che il nodo successivo sia diverso da `NULL` (riga 5). Questo implica che `L` non punta alla coda della lista.
  - Per poter aggiornare il puntatore `next` del nodo precedente (riga 8) è necessario verificare che il nodo precedente sia diverso da `NULL` (riga 7). Questo implica che `L` non punta alla testa della lista.

```
1  static int node_delete(struct node *L) {
2    if(L == NULL) {
3      return 1;
4    } else {
5      if(L->next != NULL)
6        L->next->prev = L->prev;
7      if(L->prev != NULL)
8        L->prev->next = L->next;
9      free(L);
10     return 0;
11   }
12 }
```

Creazione e distruzione di una lista e funzioni di test sulla lista

- Le implementazioni sono le stesse viste per liste concatenate.

```
1 list list_create() {
2   return NULL;
3 }
```

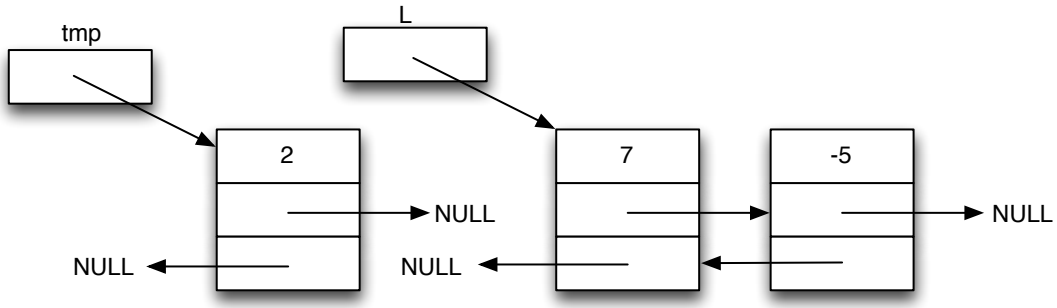
```
1 void list_delete(list *L) {
2   if(L != NULL && *L != NULL) {
3     list_delete(&(*L)->next);
4     free(*L);
5     *L=NULL;
6   }
7 }
```

```
1 int is_empty(list *L) {
2   return (L == NULL) || (*L == NULL);
3 }
```

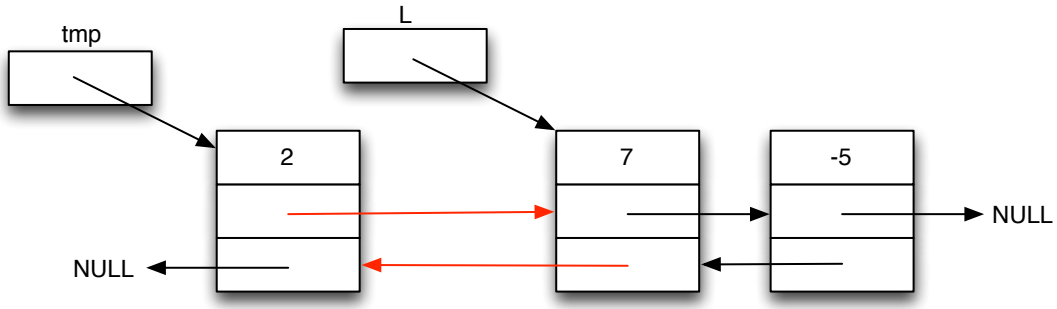
```
1  int is_inlist(list *L, int elem) {
2    if(is_empty(L)) {
3      return 0;
4    } else {
5      struct node *tmp = *L;
6
7      while(tmp != NULL && tmp->elem != elem)
8        tmp = tmp->next;
9
10     return tmp != NULL;
11   }
12 }
```

Inserimento in testa 1/2: `head_insert()`

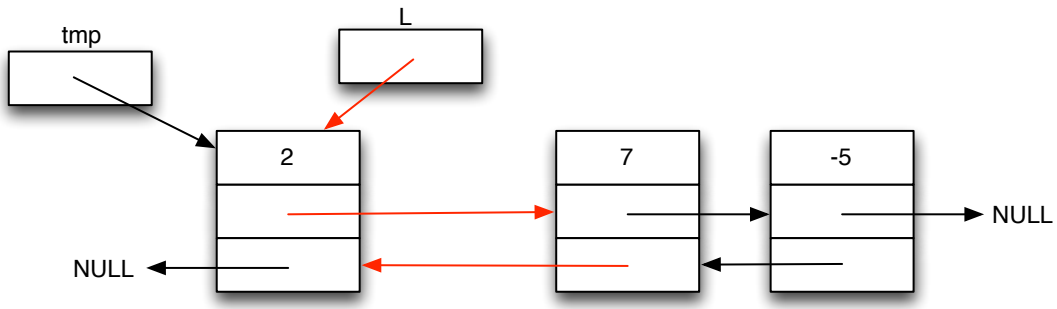
- La procedura di inserimento in testa deve gestire anche l'aggiornamento del puntatore `prev` della vecchio nodo in testa.



A) Creazione del nuovo nodo da inserire (`tmp`)



B) Aggiornamento dei campo `next` del nuovo nodo e `prev` della vecchia testa



C) Aggiornamento del puntatore alla testa della lista

Inserimento in testa 2/2: `head_insert()`

- La procedura di inserimento in testa è leggermente diversa rispetto alla versione vista per liste concatenate.
  - E' più comodo gestire separatamente il caso in cui la lista sia vuota (riga 4).
  - Se la lista contiene almeno un elemento (corpo else a riga 6) è necessario modificare il puntatore `prev` della vecchia testa in modo che punti alla nuova testa (riga 11).

```
1  int head_insert(list *L, int elem) {
2    if(L == NULL) {
3      return 1;
4    } else if(is_empty(L)) {
5      return (*L = node_alloc(elem)) == NULL;
6    } else {
7      struct node *tmp = node_alloc(elem);
8
9      if(tmp != NULL) {
10        tmp->next = *L;
11       (*L)->prev = tmp;
12        *L        = tmp;
13     }
14     return tmp == NULL;
15   }
16 }
```

Inserimento in coda e indicizzato: `tail_insert()` e `indx_insert()`

- Le implementazioni delle funzioni di inserimento in coda ed inserimento indicizzato non cambiano rispetto alla versioni vista per liste concatenate.

```
1  int tail_insert(list *L, int elem) {
2    if(L == NULL) {
3      return 1;
4    } else if(is_empty(L)) {
5      return head_insert(L,elem);
6    } else  {
7      struct node *tmp = *L;
8
9      while(tmp->next != NULL)
10       tmp = tmp->next;
11     return node_insert(tmp,elem);
12   }
13 }
```

```
1  int indx_insert(list *L, int elem, unsigned int i) {
2    if(i == 0)
3      return head_insert(L,elem);
4    else
5      return node_insert(node_search(*L,i-1),elem);
6  }
```

Rimozione della testa: `head_delete()`

- L'implementazione della funzione di rimozione del nodo in testa può essere definita utilizzando la funzione di utility `node_delete()`.
- E' essenzialmente equivalente alla versione vista per liste concatenate: la `node_delete()` si occupa di aggiornare a `NULL` il puntatore `prev` della nuova testa della lista.

```
1  int head_delete(list *L) {
2    if(is_empty(L)) {
3      return 1;
4    } else {
5      struct node *tmp = *L;
6
7      *L = (*L)->next;
8      return node_delete(tmp);
9    }
10 }
```

Rimozione della coda: `tail_delete()`

- L'implementazione della funzione per la rimozione della coda può essere leggermente semplificata: è sufficiente passare alla funzione di utility `node_delete()` il puntatore al nodo da rimuovere.
- Anche in questo caso, l'implementazione è essenzialmente equivalente alla versione vista per liste concatenate: cambiano unicamente le condizioni di terminazione del ciclo `while` (riga 9).

```
1  int tail_delete(list *L) {
2    if(is_empty(L)) {
3      return 1;
4    } else if((*L)->next == NULL) {
5      return head_delete(L);
6    } else {
7      struct node *tmp=*L;
8
9      while(tmp->next != NULL)
10       tmp = tmp->next;
11     return node_delete(tmp);
12   }
13 }
```

Rimozione indicizzata: `indx_delete()`

- Anche l'implementazione della funzione per la rimozione indicizzata cambia di poco rispetto alla versione vista per liste concatenate: adesso dobbiamo passare alla funzione di utility `node_delete()` il puntatore al nodo da rimuovere, non il puntatore al nodo precedente.

```
1 int indx_delete(list *L, unsigned int i) {
2   if(i == 0)
3     return head_delete(L);
4   else
5     return node_delete(node_search(*L,i));
6 }
```

Funzioni di selezione: `head_select()`, `tail_select()`, `indx_select()`

- Le implementazioni delle funzioni di selezione non cambiano rispetto alla versioni vista per liste concatenate.

```c
int head_select(list *L) {
  if(is_empty(L))
    return 0;
  else
    return (*L)->elem;
}
```

```c
int tail_select(list *L) {
  if(is_empty(L)) {
    return 0;
  } else {
    struct node *tmp=*L;

    while(tmp->next != NULL)
      tmp = tmp->next;
    return tmp->elem;
  }
}
```

```c
int indx_select(list *L, unsigned int i) {
  if(is_empty(L)) {
    return 0;
  } else {
    struct node *tmp = node_search(*L,i);
    if(tmp == NULL)
      return 0;
    else
      return tmp->elem;
  }
}
```

Pro e contro della rappresentazione con liste concatenate

- **Pro**.
  - In confronto alla rappresentazione tramite array, offrono gli stessi vantaggi delle liste concatenate semplici.
  - Sono maggiormente indicate delle liste concatenate semplici per l'implementazione di operazioni che richiedano l'attraversamento della lista dalla coda verso la testa.

Pro e contro della rappresentazione con liste concatenate

- **Pro**.
  - In confronto alla rappresentazione tramite array, offrono gli stessi vantaggi delle liste concatenate semplici.
  - Sono maggiormente indicate delle liste concatenate semplici per l'implementazione di operazioni che richiedano l'attraversamento della lista dalla coda verso la testa.
- **Contro**.
  - Così come per le liste concatenate semplici, ad eccezione dell'elemento in testa, non è possibile accedere direttamente ad un elemento della lista. L'accesso all'elemento in coda richiede l'attraversamento dell'intera lista.
  - Rispetto alle liste concatenate semplici, comportano un ulteriore *overhead* di memoria, dovuto alla necessità di memorizzare due puntatori per ogni elemento della lista.

Liste circolari e liste circolari doppiamente concatenate

- Nelle **liste circolari**, il nodo in coda punta alla testa della lista (campo `next`).
- Nelle **liste circolari doppiamente concatenate**, il nodo in coda punta alla testa della lista (campo `next`) e il nodo in testa punta alla coda della lista (campo `prev`).

Liste circolari e liste circolari doppiamente concatenate

- Nelle **liste circolari**, il nodo in coda punta alla testa della lista (campo `next`).
- Nelle **liste circolari doppiamente concatenate**, il nodo in coda punta alla testa della lista (campo `next`) e il nodo in testa punta alla coda della lista (campo `prev`).
- La rappresentazione circolare permette di gestire in modo efficiente tutte le operazioni che lavorano sulla testa e sulla coda della lista.
- Nessuno dei puntatori `next` e `prev` nella lista punta a `NULL`: è necessario adottare nuove strategie per determinare in un ciclo quando abbiamo visitato l'intera lista.

Liste circolari e liste circolari doppiamente concatenate

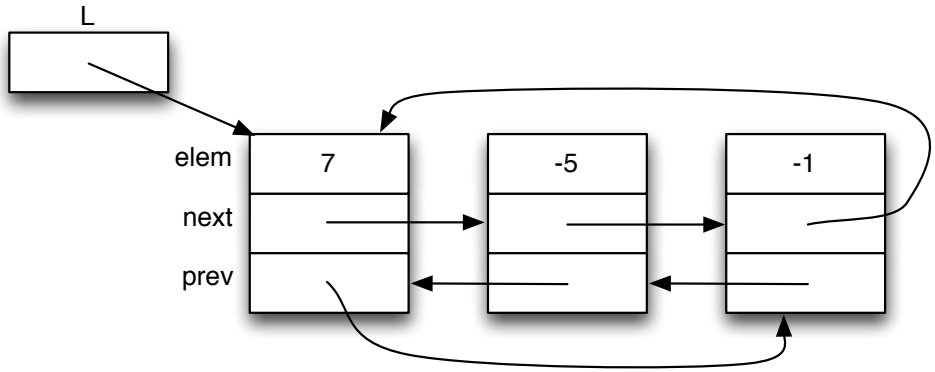
- Nelle **liste circolari**, il nodo in coda punta alla testa della lista (campo `next`).
- Nelle **liste circolari doppiamente concatenate**, il nodo in coda punta alla testa della lista (campo `next`) e il nodo in testa punta alla coda della lista (campo `prev`).
- La rappresentazione circolare permette di gestire in modo efficiente tutte le operazioni che lavorano sulla testa e sulla coda della lista.
- Nessuno dei puntatori `next` e `prev` nella lista punta a `NULL`: è necessario adottare nuove strategie per determinare in un ciclo quando abbiamo visitato l'intera lista.
- Caratteristiche specifiche delle liste circolari semplici:
  - Sono implementate mantenendo un puntatore alla coda della lista, non alla testa: in questo modo è possibile accedere velocemente sia alla testa che alla coda.
  - Questa rappresentazione complica leggermente l'implementazione: bisogna gestire in modo speciale il caso in cui la lista contenga un solo nodo.
  - La lista può essere attraversata in una sola direzione.

Liste circolari e liste circolari doppiamente concatenate

- Nelle **liste circolari**, il nodo in coda punta alla testa della lista (campo `next`).
- Nelle **liste circolari doppiamente concatenate**, il nodo in coda punta alla testa della lista (campo `next`) e il nodo in testa punta alla coda della lista (campo `prev`).
- La rappresentazione circolare permette di gestire in modo efficiente tutte le operazioni che lavorano sulla testa e sulla coda della lista.
- Nessuno dei puntatori `next` e `prev` nella lista punta a `NULL`: è necessario adottare nuove strategie per determinare in un ciclo quando abbiamo visitato l'intera lista.
- Caratteristiche specifiche delle liste circolari semplici:
  - Sono implementate mantenendo un puntatore alla coda della lista, non alla testa: in questo modo è possibile accedere velocemente sia alla testa che alla coda.
  - Questa rappresentazione complica leggermente l'implementazione: bisogna gestire in modo speciale il caso in cui la lista contenga un solo nodo.
  - La lista può essere attraversata in una sola direzione.
- Caratteristiche specifiche delle liste circolari doppiamente concatenate:
  - Un puntatore alla testa ci permette di accedere velocemente anche alla coda.
  - La lista può essere attraversata in entrambe le direzioni.
- Mostriamo l'implementazione di liste circolari doppiamente concatenate.

Il tipo di dato lista circolare doppiamente concatenata

- Possiamo rappresentare il tipo di dato `list` come **puntatore a struttura** contente:
  - Un campo (`elem`) di tipo in `int`
  - Un campo puntatore (`next`) ad una struttura dello stesso tipo.
  - Un campo puntatore (`prev`) ad una struttura dello stesso tipo.




```
1 struct node {
2   int elem;
3   struct node *next;
4   struct node *prev;
5 };
6 typedef struct node *list;
```

- Da un nodo è possibile accedere sial al nodo successivo (campo `next`) che al nodo precedente (campo `prev`) nella lista.
  - Il campo `next` dell'ultimo nodo punta al nodo in testa alla lista.
  - Il campo `prev` del primo nodo punta al nodo in coda alla lista.
- Le funzioni di libreria hanno come argomento formale un **puntatore al puntatore** al primo nodo della lista.

Utility di libreria per liste circolari

- Le utility di libreria sono definite come quelle viste per liste doppiamente concatenate: le implementazioni devono preservare la proprietà di *circolarità* della struttura dati.

```
1  /*
2   * Alloca un nodo con valore elem.
3   *
4   * Ritorna un puntatore al nodo allocato o NULL
5   * se l'allocazione non ha successo.
6   */
7  static struct node *node_alloc(int);
8  /*
9   * Ritorna un puntatore all'i-esimo nodo dopo il nodo
10  * puntato da L. Ritorna  NULL se tale nodo non esiste.
11  */
12 static struct node *node_search(struct node *L, unsigned int i);
13 /*
14  * Rimuove il nodo L.
15  *
16  * Ritorna 0 se l'operazione è stata eseguita.
17  */
18 static int node_delete(struct node *L);
19 /*
20  * Inserisce un nodo con valore elem dopo il nodo L.
21  *
22  * Ritorna 0 se l'operazione è stata eseguita.
23  */
24 static int node_insert(struct node *L, int elem);
```

Utility per la creazione e ricerca: `node_alloc()` e `node_search()`

- In una lista circolare doppiamente concatenata un nodo singolo precede e segue se stesso: i puntatori `prev` e `next` puntano alla struttura in cui sono contenuti.

```
1  static struct node *node_alloc(int elem) {
2    struct node *tmp = (struct node *)malloc(sizeof(struct node));
3
4    if(tmp != NULL) {
5      tmp->elem = elem;
6      tmp->next = tmp;
7      tmp->prev = tmp;
8    }
9    return tmp;
10 }
```

Utility per la creazione e ricerca: `node_alloc()` e `node_search()`

- In una lista circolare doppiamente concatenata un nodo singolo precede e segue se stesso: i puntatori `prev` e `next` puntano alla struttura in cui sono contenuti.

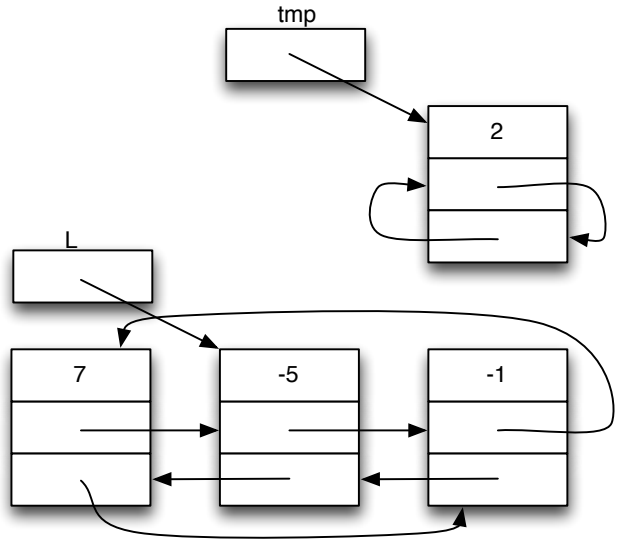
```
1  static struct node *node_alloc(int elem) {
2    struct node *tmp = (struct node *)malloc(sizeof(struct node));
3
4    if(tmp != NULL) {
5      tmp->elem = elem;
6      tmp->next = tmp;
7      tmp->prev = tmp;
8    }
9    return tmp;
10 }
```

- La funzione di ricerca deve essere impostata diversamente da quanto visto finora, dato che non possiamo più verificare la fine della lista testando `NULL`.

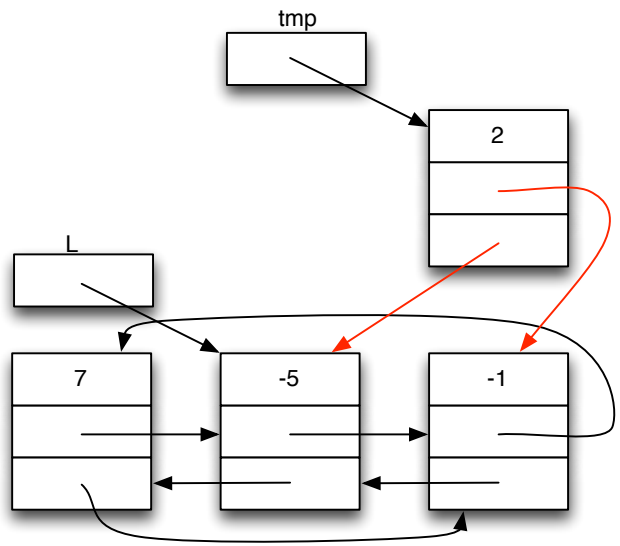
```
1  static struct node *node_search(struct node *L, unsigned int i) {
2    if(i == 0 || L == NULL) {
3      return L;
4    } else {
5      struct node *tmp = L->next;
6      while(--i > 0  && tmp != L)
7        tmp = tmp->next;
8      return tmp == L ? NULL : tmp;
9    }
10 }
```

Utility per l'inserimento di un nodo 1/2: `node_insert()`

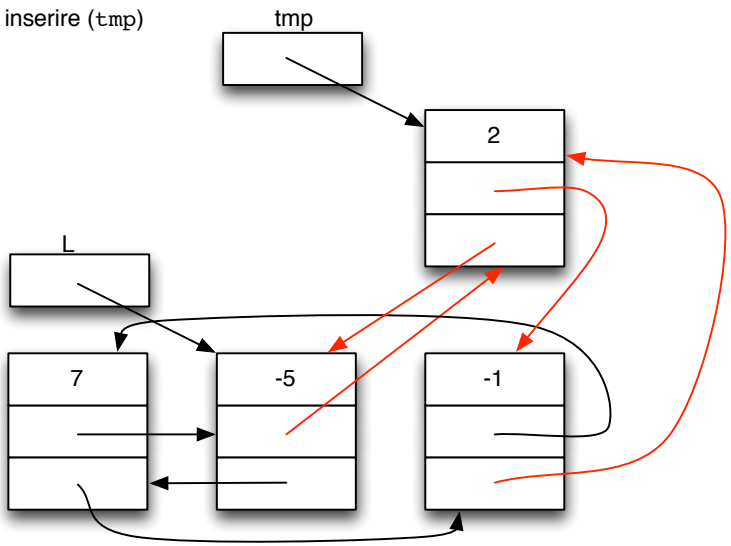
- L'inserimento di un nodo viene eseguito esattamente come abbiamo visto per le code doppiamente collegate: potremmo utilizzare la stessa implementazione.



A) Creazione del nuovo nodo da inserire (`tmp`)



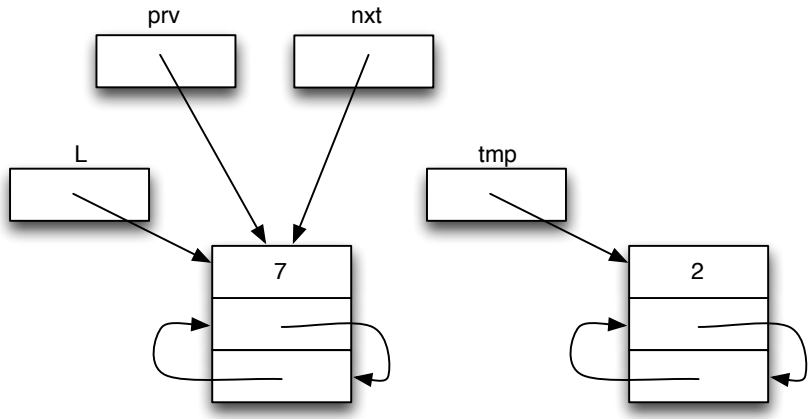
B) Aggiornamento dei campo `next` e `prev` del nuovo nodo



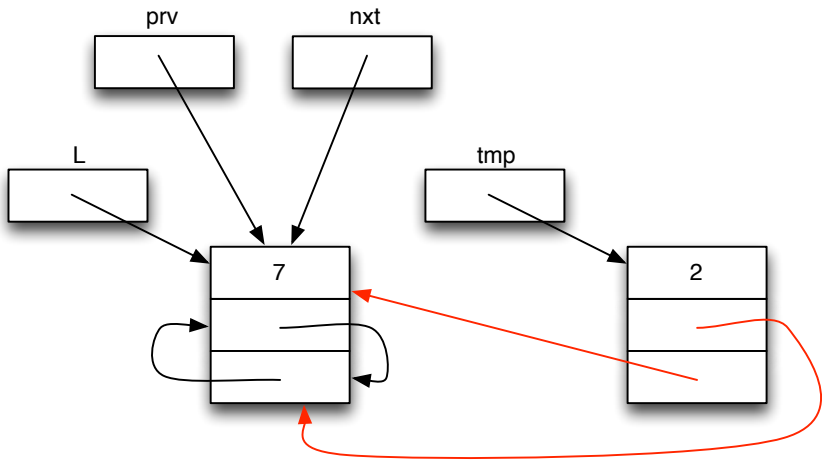
C) Aggiornamento campo `next` del nodo precedente e del campo `prev` del nodo successivo

Utility per l'inserimento di un nodo 2/3: `node_insert()`

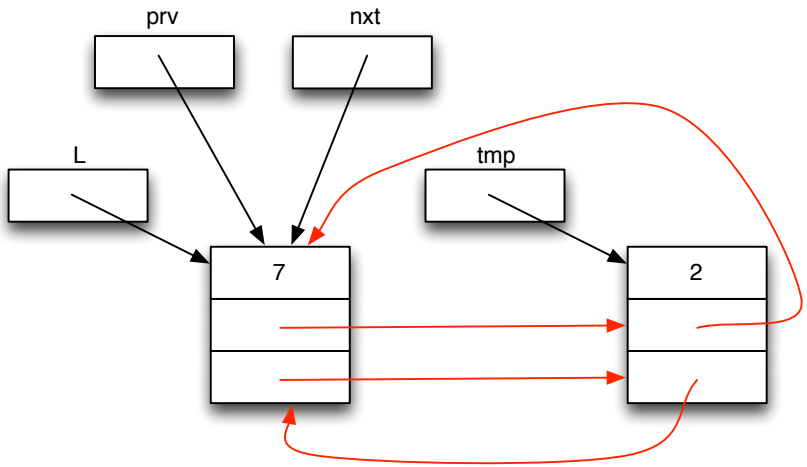
- A livello logico, l'implementazione può essere gestita più facilmente (in modo più pulito) se distinguiamo il nodo che precede (`prv`) dal nodo che segue (`nxt`) il nuovo nodo da inserire (anche se `prv` e `nxt` coincidono).



A) Creazione del nuovo nodo da inserire (`tmp`) e individuazione dei nodi che precedono (`prv`) e seguono (`nxt`)e seguono il nodo da inserire.



B) Il campo `prev` del nuovo nodo punta al nodo `prv`, il campo `next` punta al nodo `nxt`.



C) Aggiornamento campo `next` del nodo `prv` e del campo `prev` del nodo `nxt`.

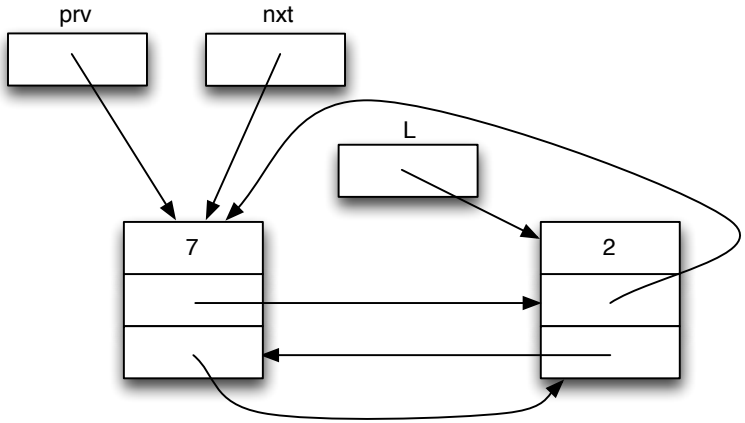
Utility per l'inserimento di un nodo 3/3: `node_insert()`

- Possiamo riscrivere un'implementazione ad-hoc per le liste circolari doppiamente collegate, evitando di eseguire test (in questo caso inutili) su puntatori a `NULL`.
  - E' sufficiente individuare (riga 5) il nodo che precede, `prv`, e quello che segue, `nxt` il nuovo nodo, `tmp`. Se la lista contiene un solo elemento, `prv` e `nxt` coincidono.
  - Aggiorniamo, i campi `prev` e `next` del nuovo nodo in modo che puntino rispettivamente al nodo precedente (riga 9) e al nodo successivo (riga 10).
  - Aggiorniamo il campo `next` del nodo precedente (riga 11) e il campo `prev` del nodo successivo (riga 12) in modo che puntino al nuovo nodo.

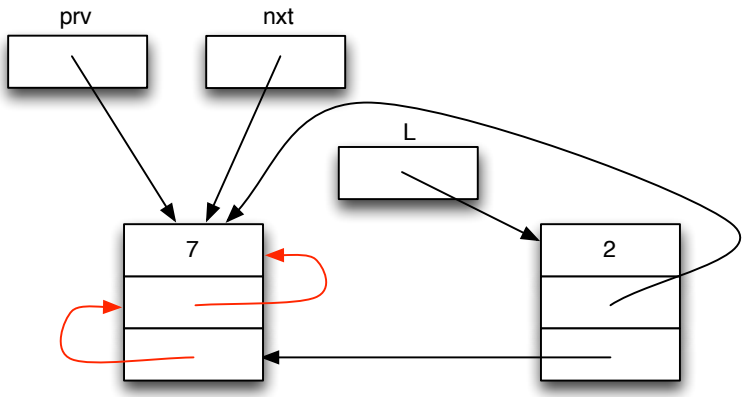
```
1  static int node_insert(struct node *L, int elem) {
2    if(L == NULL) {
3      return 1;
4    } else {
5      struct node *tmp = node_alloc(elem), *nxt = L->next, *prv = L;
6      if(tmp != NULL) {
7        tmp->prev = prv;
8        tmp->next = nxt;
9        prv->next = tmp;
10       nxt->prev = tmp;
11     }
12     return tmp == NULL;
13   }
14 }
```

Utility per la rimozione di un nodo 1/2: `node_delete()`

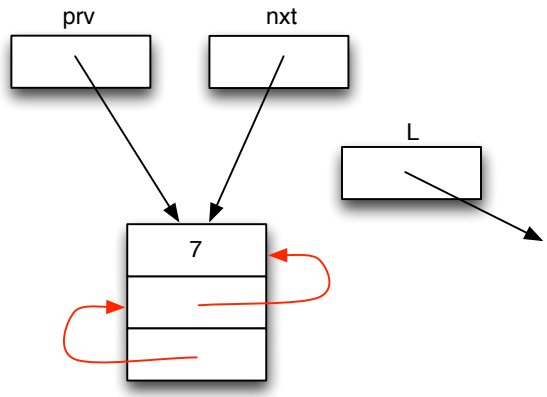
- Anche la rimozione di un nodo viene eseguita esattamente come abbiamo visto per le code doppiamente collegate: potremmo utilizzare la stessa implementazione.
- Anche in questo caso è comodo distinguere il nodo che precede (`prv`) dal nodo che segue (`nxt`) il nodo da rimuovere (anche se coincidono).



A) Individuazione dei nodi che precedono (`prv`) e seguono (`nxt`) e seguono il nodo da rimuovere (`L`).



B) Il campo `next` del nodo `prv` punta al nodo `nxt`.
Il campo `prev` del nodo `nxt` punta al nodo `prv`.



C) Il nodo da rimuovere viene deallocato.

Utility per la rimozione di un nodo 2/2: `node_delete()`

- Anche in questo caso scegliamo di scrivere un'implementazione ad-hoc per le liste circolari doppiamente collegate.
  - E' sufficiente individuare (riga 5) il nodo che precede, `prv`, e quello che segue, `nxt` il nodo da rimuovere `L`. Se la lista contiene un solo elemento, `prv` e `nxt` coincidono con il nodo `L` e le operazioni di aggiornamento non modificano nulla.
  - Aggiorniamo il campo `prev` del nodo successivo (riga 6) in modo che punti al nodo precedente.
  - Aggiorniamo il campo `next` del nodo precedente (riga 7) in modo che punti al nodo successivo.
  - Deallochiamo il nodo da rimuovere (riga 8).

```
1  static int node_delete(struct node *L) {
2    if(L == NULL) {
3      return 1;
4    } else {
5      struct node *prv = L->prev, *nxt = L->next;
6      nxt->prev = prv;
7      prv->next = nxt;
8      free(L);
9      return 0;
10   }
11 }
```

Creazione e distruzione di una lista: `list_create()` e `list_delete()`

- La funzione di creazione è implementata esattamente come abbiamo visto per liste concatenate (semplici) e liste doppiamente concatenate: restituisce la costante `NULL`

```
1 list list_create() {
2   return NULL;
3 }
```

Creazione e distruzione di una lista: `list_create()` e `list_delete()`

- La funzione di creazione è implementata esattamente come abbiamo visto per liste concatenate (semplici) e liste doppiamente concatenate: restituisce la costante `NULL`

```
1 list list_create() {
2   return NULL;
3 }
```

- Non possiamo riutilizzare la funzione ricorsiva implementata per liste concatenate e doppiamente concatenate per la distruzione della lista: non siamo in grado di determinare la fine della lista con un semplice confronto con `NULL`.
- Mostriamo un'implementazione iterativa che fa uso della utility `node_delete()`.
  - Fintanto che la lista contiene almeno due nodi, rimuoviamo il nodo successivo al nodo in testa.
  - Se la lista contiene un solo nodo, deallochiamo l'unico nodo e settiamo a `NULL` il puntatore alla testa della lista.

```
1 void list_delete(list *L) {
2   if(L != NULL && *L != NULL) {
3     while(*L != (*L)->next)
4       node_delete((*L)->next);
5     free(*L);
6     *L = NULL;
7   }
8 }
```

Funzioni di test: `is_empty()` e `is_inlist()`

- La funzione di test per lista vuota è implementata esattamente come abbiamo visto per liste concatenate (semplici) e liste doppiamente concatenate: è sufficiente verificare che il puntatore alla testa della lista punti a `NULL`.

```
1 int is_empty(list *L) {
2   return (L == NULL) || (*L == NULL);
3 }
```

Funzioni di test: `is_empty()` e `is_inlist()`

- La funzione di test per lista vuota è implementata esattamente come abbiamo visto per liste concatenate (semplici) e liste doppiamente concatenate: è sufficiente verificare che il puntatore alla testa della lista punti a `NULL`.

```
1 int is_empty(list *L) {
2   return (L == NULL) || (*L == NULL);
3 }
```

- Per verificare se un elemento è presente o meno nella lista è necessario *scorrere* tutta la lista seguendo la catena di puntatori `next` (riga 7).
  - Il ciclo termina se raggiungiamo l'ultimo nodo della lista oppure se troviamo un nodo contenente il valore passato come parametro (`elem`).
- La funzione restituisce 0 (elemento non presente) se la lista è vuota (riga 2) oppure se dopo la ricerca il nodo puntato da `tmp` non contiene il valore `elem` (riga 10).

```
1  int is_inlist(list *L, int elem) {
2    if(is_empty(L)) {
3      return 0;
4    } else {
5      struct node *tmp = *L;
6      while(tmp != (*L)->prev && tmp->elem!=elem)
7        tmp = tmp->next;
8      return tmp->elem == elem;
9    }
10 }
```

Inserimento in testa: `head_insert()`

- La funzione di inserimento in testa può essere implementata facendo uso della utility `node_insert()`, a cui viene passato il puntatore al nodo in coda.
- Gestiamo in modo separato differenti casi
  - Se i parametro `L` passato in input (riga 2) non è consistente (`NULL`), la procedura termina in *errore* (riga 3).
  - Se la lista è vuota (riga 4), viene semplicemente allocato un nuovo nodo (riga 5).
  - Se la lista non è vuota, inseriamo un nuovo nodo dopo il nodo in coda (riga 6).
    - Se l'inserimento non ha successo la procedure termina in *errore* (riga 7)
    - Altrimenti, il puntatore alla testa della lista viene aggiornato in modo che punti al nuovo nodo inserito (riga 9).

```
1  int head_insert(list *L, int elem) {
2    if(L == NULL)
3      return 1;
4    else if(is_empty(L))
5      return (*L = node_alloc(elem)) == NULL;
6    else if (node_insert((*L)->prev,elem) == 1)
7      return 1;
8    else
9      return (*L = (*L)->prev) == NULL;
10 }
```

Inserimento in coda: `tail_insert()`

- Anche la procedura per l'inserimento in coda può essere scritta facendo uso della utility `node_insert()`.
- L'impostazione della procedura è perfettamente equivalente all'impostazione della `head_insert()` (confrontare le implementazioni).
- Come unica differenza: non abbiamo la necessità di gestire l'aggiornamento del puntatore alla testa della lista.

```
1 int tail_insert(list *L, int elem) {
2   if(L == NULL)
3     return 1;
4   else if(is_empty(L))
5     return (*L = node_alloc(elem)) == NULL;
6   else
7     return node_insert((*L)->prev,elem);
8 }
```

Rimozione della testa: `head_delete()`

- La rimozione della testa su liste circolari richiede di gestire separatamente il caso in cui la lista contenga un solo nodo.
- La funzione gestisce i seguenti casi:
  - Se i parametro `L` passato in input (riga 2) non è consistente (`NULL`), la procedura termina in *errore* (riga 3).
  - Se la lista contiene un solo nodo (riga 4), allora deallochiamo l'unico nodo e aggiorniamo a `NULL` il puntatore alla testa della lista (riga 5).
  - Se la lista contiene almeno due nodi, aggiorniamo il puntatore alla testa della lista in modo che punti al nodo successivo al primo (riga 9) e richiamiamo la procedura `node_delete()` sulla vecchia testa della lista (riga 10).

```
1  int head_delete(list *L) {
2    if(is_empty(L)) {
3      return 1;
4    } else if (*L == (*L)->next) {
5      free(*L);
6      *L = NULL;
7      return 0;
8    } else {
9      *L = (*L)->next;
10     return node_delete((*L)->prev);
11   }
12 }
```

Rimozione della coda: `tail_delete()`

- Come per la procedura di inserimento in coda, l'impostazione della procedura di rimozione della coda è perfettamente equivalente all'impostazione della `head_delete()` (confrontare le implementazioni).
- Anche in questo caso, come unica differenza: non abbiamo la necessità di gestire l'aggiornamento del puntatore alla testa della lista.

```
1  int tail_delete(list *L) {
2    if(is_empty(L)) {
3      return 1;
4    } else if (*L == (*L)->next) {
5      free(*L);
6      *L = NULL;
7      return 0;
8    } else {
9      return node_delete((*L)->prev);
10   }
11 }
```

Inserimento e rimozione indicizzata: `indx_insert()` e `indx_delete()`

- Le implementazioni delle funzioni di inserimento e rimozione indicizzata non cambiano rispetto alle versioni vista per liste doppiamente concatenate.

```
1 int indx_insert(list *L, int elem, unsigned int i) {
2   if(i == 0)
3     return head_insert(L,elem);
4   else
5     return node_insert(node_search(*L,i-1),elem);
6 }
```

```
1 int indx_delete(list *L, unsigned int i) {
2   if(i == 0)
3     return head_delete(L);
4   else
5     return node_delete(node_search(*L,i));
6 }
```

Funzioni di selezione: `head_select()`, `tail_select()`, `indx_select()`

- Le implementazioni delle funzioni di selezione della testa (`head_select()`) e selezione indicizzata (`indx_select()`) non cambiano rispetto alla versioni vista per liste doppiamente concatenate.
- L'implementazione della selezione della coda (`tail_select()`) è molto più efficiente dato che adesso possiamo accedere direttamente alla coda della lista tramite il puntatore `prev` nel nodo in testa.

```
1  int head_select(list *L) {
2    if(is_empty(L))
3      return 0;
4    else
5      return (*L)->elem;
6  }
```

```
1  int tail_select(list *L) {
2    if(is_empty(L))
3      return 0;
4    else
5      return (*L)->prev->elem;
6  }
```

```
1  int indx_select(list *L, unsigned int i) {
2    if(is_empty(L)) {
3      return 0;
4    } else {
5      struct node *tmp = node_search(*L,i);
6      if(tmp == NULL)
7        return 0;
8      else
9        return tmp->elem;
10   }
11 }
```

Pro e contro della rappresentazione con liste circolari doppiamente concatenate

- **Pro**.
  - E' possibile accedere direttamente alla coda della lista senza doverla attraversare interamente: le operazioni di rimozione/inserimento in coda
  - Possono essere attraversate in entrambe le direzioni.

Pro e contro della rappresentazione con liste circolari doppiamente concatenate

- **Pro**.
  - E' possibile accedere direttamente alla coda della lista senza doverla attraversare interamente: le operazioni di rimozione/inserimento in coda
  - Possono essere attraversate in entrambe le direzioni.
- **Contro**.
  - L'accesso ad un elemento all'interno della lista richiede ancora l'attraversamento di un certo numero di nodi.
  - Le implementazioni di operazioni che richiedono l'attraversamento della lista sono leggermente complicate: non possiamo effettuare un confronto con `NULL` per determinare la fine della lista.
  - Hanno un overhead di memoria dovuto alla memorizzazione di due puntatori per ogni nodo, anche se non aggiungono nessun ulteriore overhead rispetto alle liste doppiamente concatenate.