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

Liste doppiamente concatenate e circolari

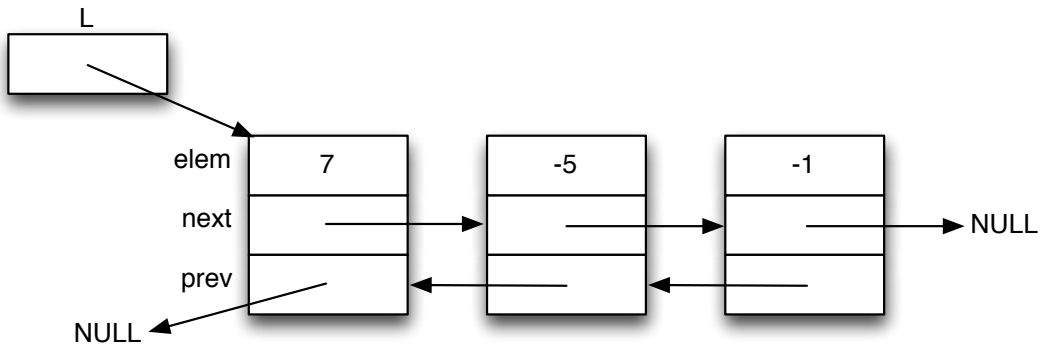
Andrea Piroddi - andrea.piroddi@unibo.it
Credit: Pietro Di Lena

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`.

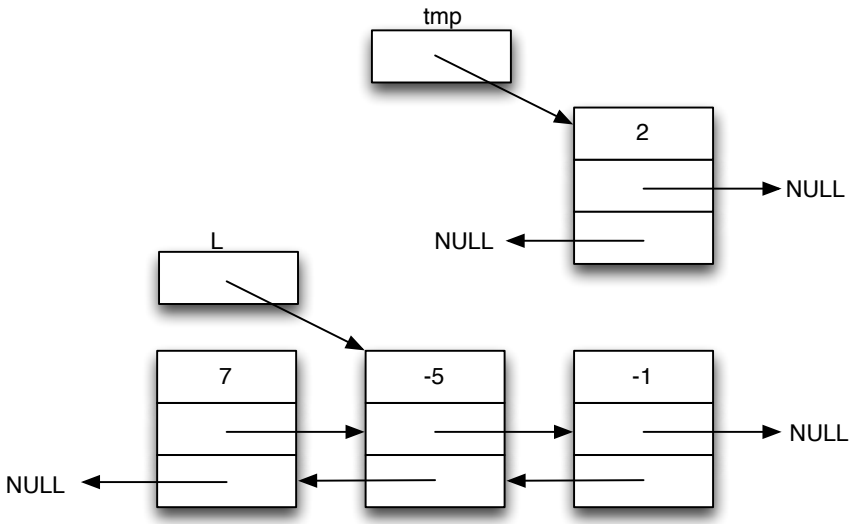
```
1  static struct node *node_alloc(int elem) {
2    struct node *tmp = (struct node *)malloc(sizeof(struct node));
3
4    if(tmp != NULL) {
5      tmp->elem = elem;
6      tmp->next = NULL;
7      tmp->prev = NULL;
8    }
9    return tmp;
10 }
```

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

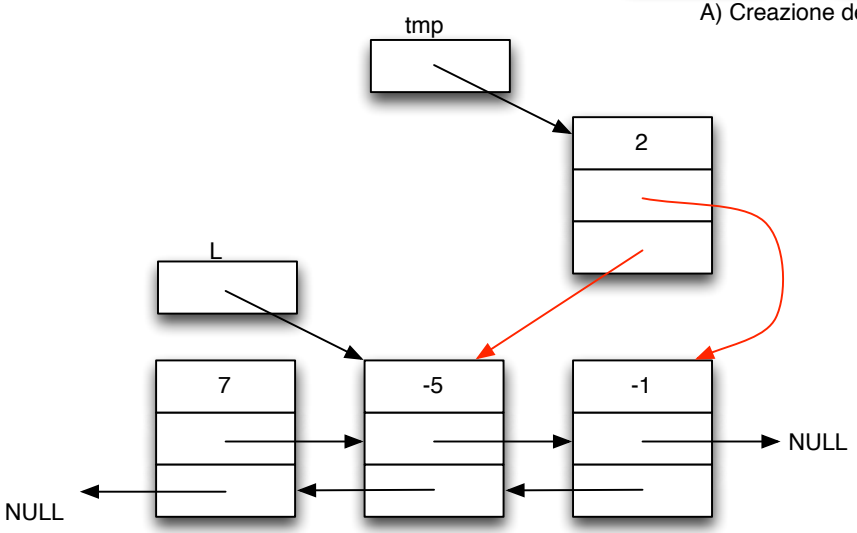
```
1  llist.node_search.c
```

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

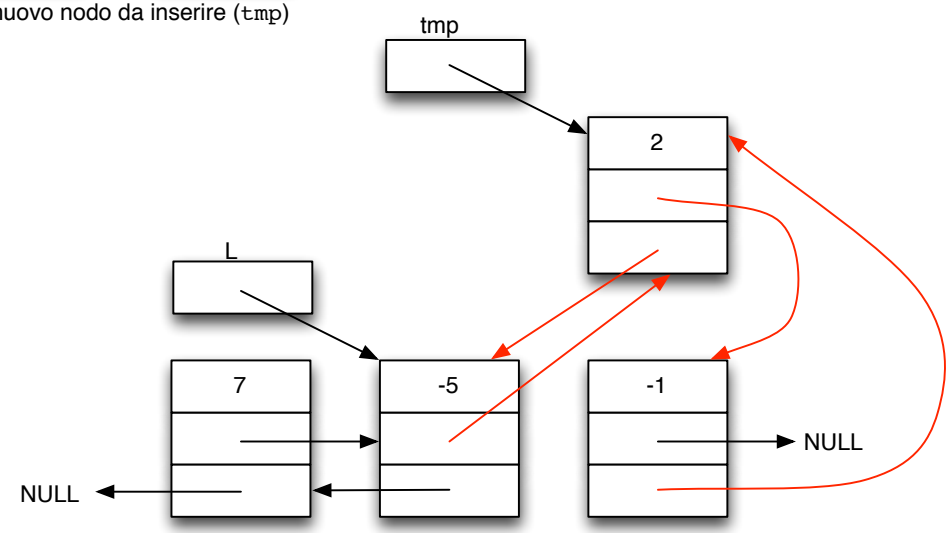
- Nell'implementazione della funzione per l'inserimento di un nodo dobbiamo adesso gestire anche l'aggiornamento del puntatore `prev`.



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

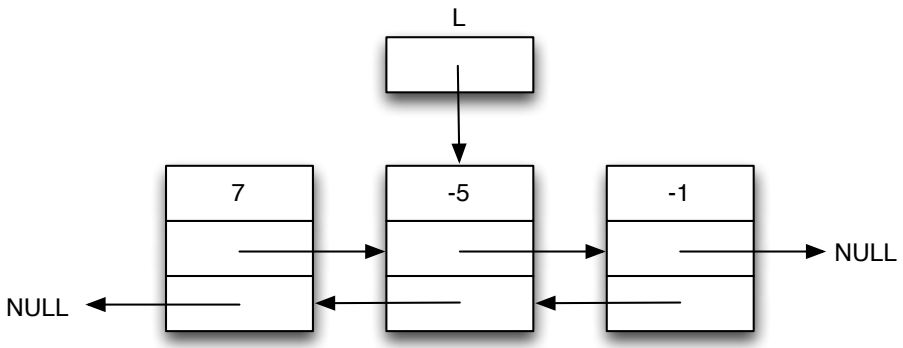
Utility per l'inserimento di un nodo 2/2: `node_insert()`

- 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).

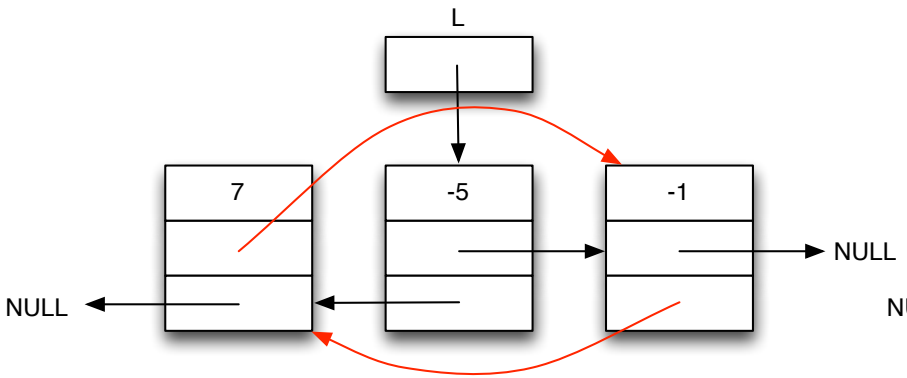
```
1  static int node_insert(struct node *L, int elem) {
2    if(L == NULL) {
3      return 1;
4    } else {
5      struct node *tmp = node_alloc(elem);
6
7      if(tmp != NULL) {
8        tmp->next = L->next;
9        tmp->prev = L;
10       L->next   = tmp;
11       if(tmp->next != NULL)
12      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

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

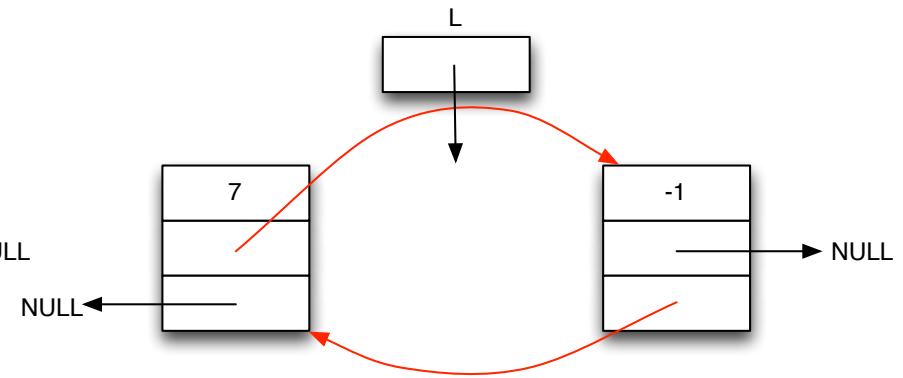
- La funzione di utility per la rimozione di un nodo in una lista doppiamente concatenata è utilizzata in modo alquanto differente rispetto alla sua controparte per liste concatenate.
- Grazie all'utilizzo di puntatori in avanti ed indietro siamo in grado di gestire la rimozione di un nodo passando alla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

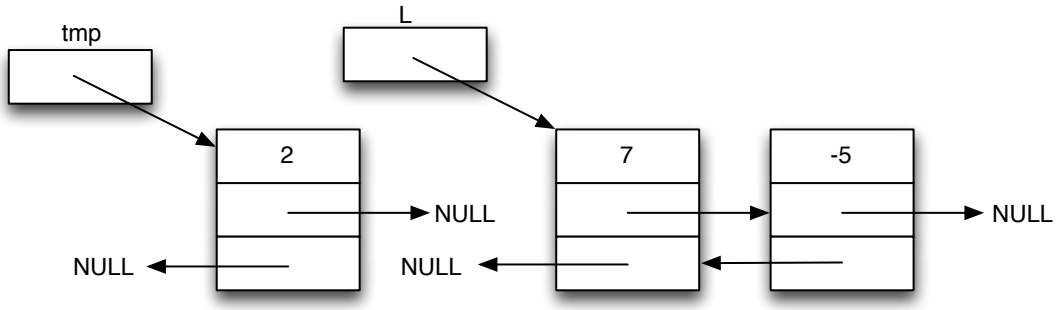
```
1 llist.list_delete.c
```

```
1 llist.is_empty.c
```

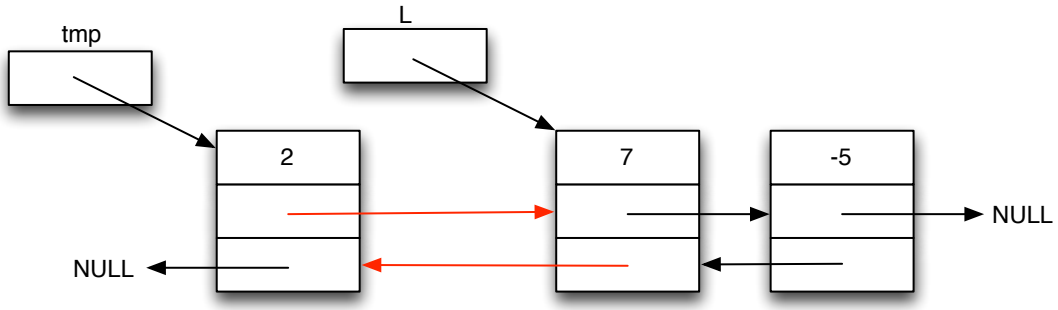
```
1 llist.is_inlist.c
```

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

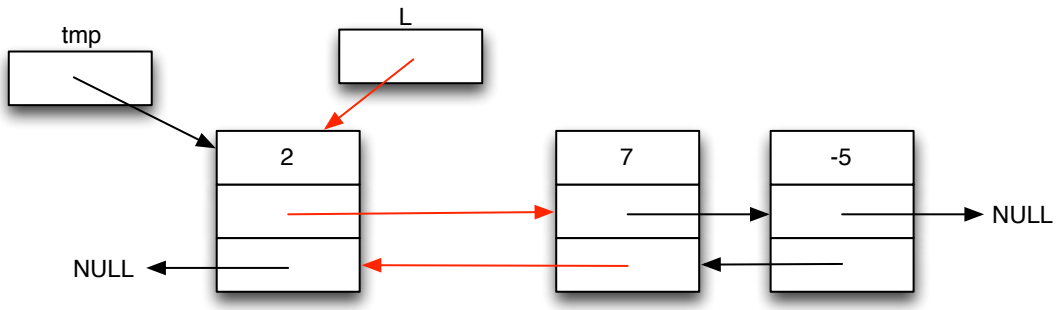
- La procedura di inserimento in testa deve gestire anche l'aggiornamento del puntatore `prev` della vecchio nodo in testa.



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 llist.tail_insert.c
```

```
1 llist.indx_insert.c
```

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.

`llist.head_select.c`

```
1
```

`llist.tail_select.c`

```
1
```

`llist.indx_select.c`

```
1
```

Pro e contro della rappresentazione con liste concatenate

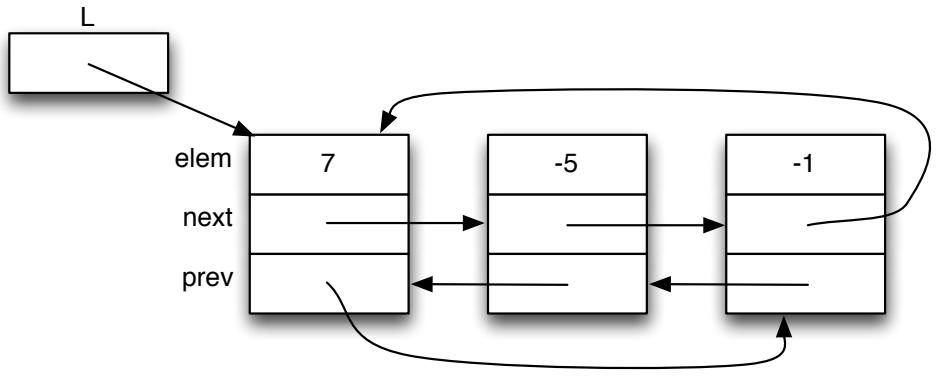
- **Pro**.
  - In confronto alla rappresentazione tramite array, offrono gli stessi vantaggi delle liste concatenate semplici.
  - Sono maggiormente indicate delle liste concatenate semplici per l'implementazione di operazioni che richiedano l'attraversamento della lista dalla coda verso la testa.
-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

- 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 dlist.declaration.h
```

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.

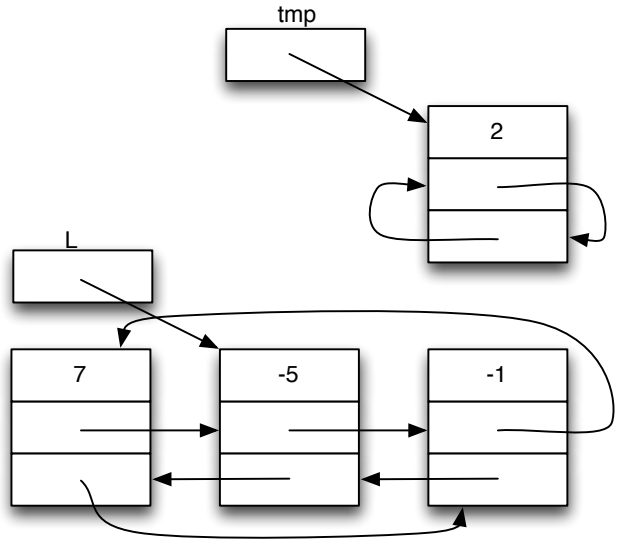
```
1  static struct node *node_alloc(int elem) {
2    struct node *tmp = (struct node *)malloc(sizeof(struct node));
3
4    if(tmp != NULL) {
5      tmp->elem = elem;
6      tmp->next = tmp;
7      tmp->prev = tmp;
8    }
9    return tmp;
10 }
```

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

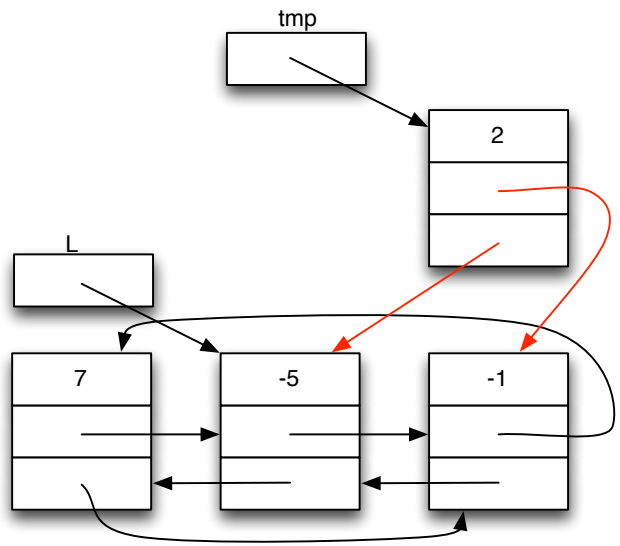
```
1  static struct node *node_search(struct node *L, unsigned int i) {
2    if(i == 0 || L == NULL) {
3      return L;
4    } else {
5      struct node *tmp = L->next;
6      while(--i > 0  && tmp != L)
7        tmp = tmp->next;
8      return tmp == L ? NULL : tmp;
9    }
10 }
```

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

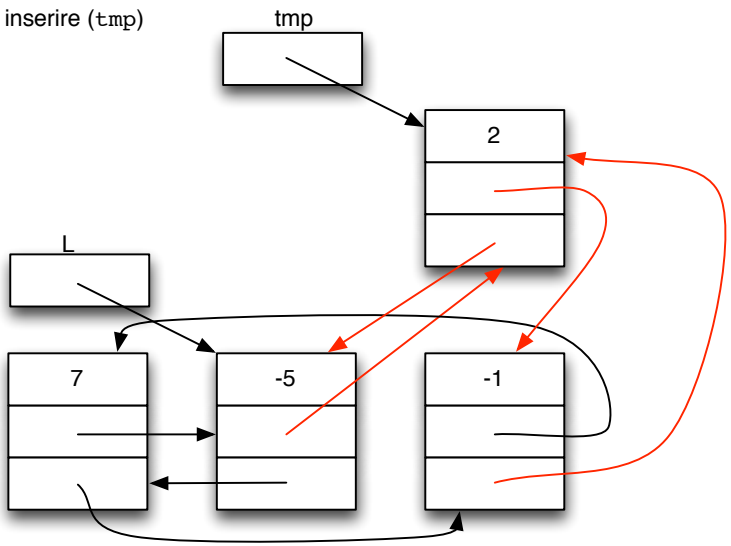
- L'inserimento di un nodo viene eseguito esattamente come abbiamo visto per le code doppiamente collegate: potremmo utilizzare la stessa implementazione.



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



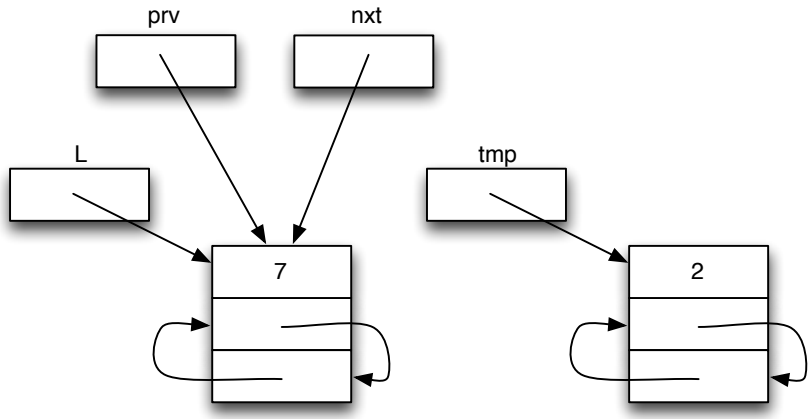
B) Aggiornamento dei campo `next` e `prev` del nuovo nodo



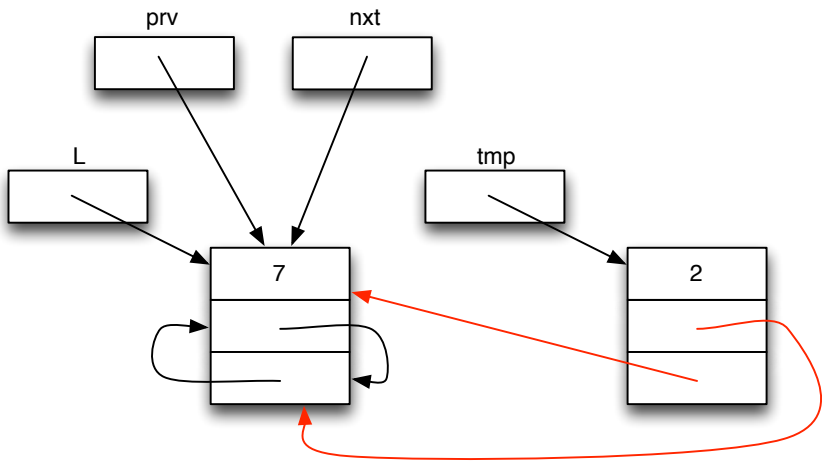
C) Aggiornamento campo `next` del nodo precedente e del campo `prev` del nodo successivo

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

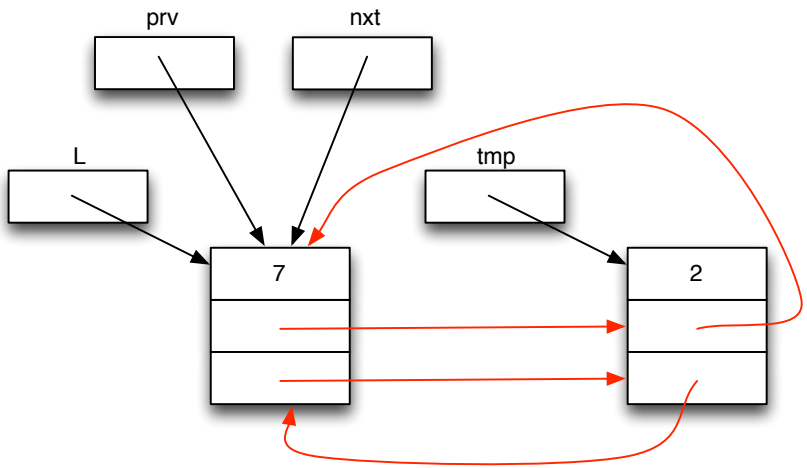
- A livello logico, l'implementazione può essere gestita più facilmente (in modo più pulito) se distinguiamo il nodo che precede (`prv`) dal nodo che segue (`nxt`) il nuovo nodo da inserire (anche se `prv` e `nxt` coincidono).



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`.

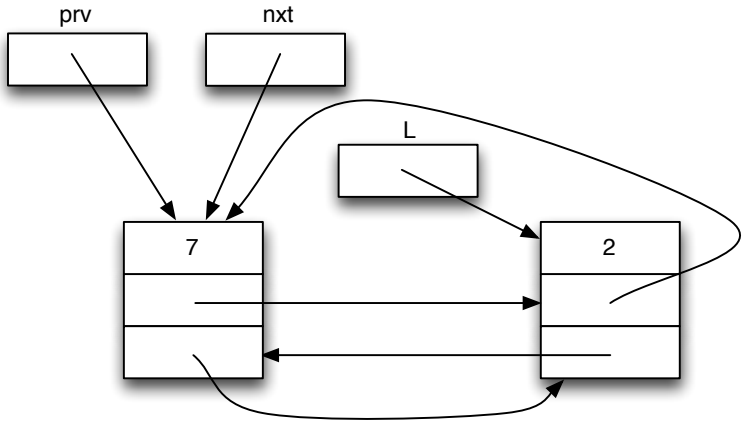
Utility per l'inserimento di un nodo 3/3: `node_insert()`

- 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.

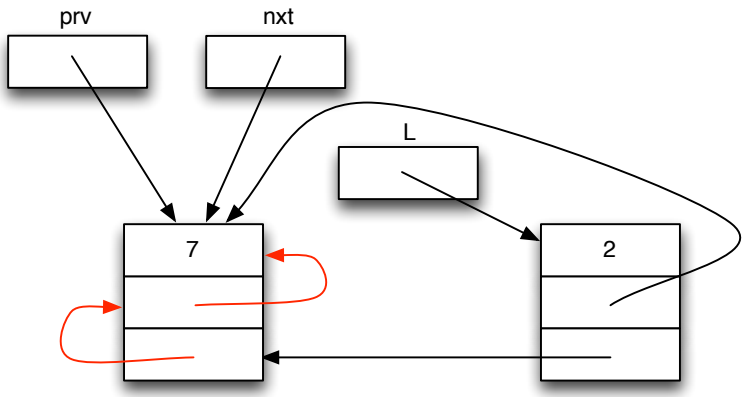
```
1  static int node_insert(struct node *L, int elem) {
2    if(L == NULL) {
3      return 1;
4    } else {
5      struct node *tmp = node_alloc(elem), *nxt = L->next, *prv = L;
6      if(tmp != NULL) {
7        tmp->prev = prv;
8        tmp->next = nxt;
9        prv->next = tmp;
10       nxt->prev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

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

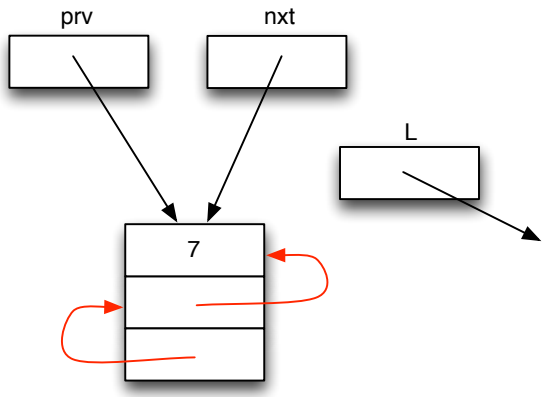
- Anche la rimozione di un nodo viene eseguita esattamente come abbiamo visto per le code doppiamente collegate: potremmo utilizzare la stessa implementazione.
- Anche in questo caso è comodo distinguere il nodo che precede (`prv`) dal nodo che segue (`nxt`) il nodo da rimuovere (anche se coincidono).



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`

dlist.list_create.c

- 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 dlist.is_empty.c
```

- 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 dlist.indx_insert.c
```

```
1 dlist.indx_delete.c
```

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 dlist.head_select.c
```

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

```
1 dlist.indx_select.c
```

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 sono sia efficienti che semplici da implementare.
  - 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.

Esercizi suggeriti 1/4

- Implementare i seguenti prototipi utilizzando le rappresentazioni lista con array, lista concatenata, lista doppiamente concatenata, lista circolare doppiamente concatenata.

```
1  /*
2   * Ritorna il numero di elementi nella lista.
3   */
4  unsigned int list_length(list *L);
5
6  /*
7   * Rimuove da L1 tutti gli elementi presenti
8   * in L2. Non modifica L2.
9   *
10  * Ritorna 0 se l'operazione ha successo,
11  * un intero diverso da zero altrimenti.
12  */
13 int list_complement(list *L1, list *L2);
14
15 /*
16  * Aggiunge ad L1 tutti gli elementi in L2
17  * non presenti in L1. Non modifica L2.
18  *
19  * Ritorna 0 se l'operazione ha successo,
20  * un intero diverso da zero altrimenti.
21  */
22 int list_union(list *L1, list *L2);
```

Esercizi suggeriti 2/4

```
1  /*
2   * Crea e ritorna una copia di L.
3   *
4   * Ritorna NULL se l'operazione non
5   * ha successo.
6   */
7  list list_duplicate(list *L);
8
9  /*
10  * Aggiunge la list L2 in coda alla lista L1.
11  * Non duplica L2, la collega ad L1.
12  *
13  * Ritorna 0 se l'operazione ha successo,
14  * un intero diverso da zero altrimenti.
15  */
16 int list_merge(list *L1, list *L2);
17
18 /*
19  * Aggiunge una copia della lista L2 in coda alla lista L1.
20  * Non modifica L2.
21  *
22  * Ritorna 0 se l'operazione ha successo,
23  * un intero diverso da zero altrimenti.
24  */
25 int list_append(list *L1, list *L2);
```

Esercizi suggeriti 3/4

```c
/*
 * Riorganizza gli elementi in L in ordine
 * crescente.
 *
 * Ritorna 0 se l'operazione ha successo: in questo caso
 * L punta alla testa della nuova lista.
 * Ritorna un intero diverso da zero altrimenti: in questo
 * caso, la lista originaria resta invariata.
 */
int list_sort(list *L);
/*
 * Inserisce elem immediatamento prima del primo
 * elemento della lista L con valore >= di elem.
 *
 * Ritorna 0 se linserimento ha successo,
 * un intero diverso da zero altrimenti.
 */
int order_insert(list *L, int elem);
/*
 * Rimuove tutti i valori duplicati nella lista L.
 *
 * Ritorna 0 se la rimozione ha successo,
 * un intero diverso da zero altrimenti.
 */
int dup_delete(list *L);
```

Esercizi suggeriti 4/4

```
1  /*
2   * Rimuove la prima occorrenza di elem in L.
3   *
4   * Ritorna 0 se la rimozione ha successo,
5   * un intero diverso da zero altrimenti.
6   */
7  int first_delete(list *L, int elem);
8  /*
9   * Rimuove l'ultima occorrenza di elem in L.
10  *
11  * Ritorna 0 se la rimozione ha successo,
12  * un intero diverso da zero altrimenti.
13  */
14 int last_delete(list *L, int elem);
15 /*
16  * Rimuove tutte le occorrenze di elem in L.
17  *
18  * Ritorna 0 se la rimozione ha successo,
19  * un intero diverso da zero altrimenti.
20  */
21 int all_delete(list *L, int elem);
```