Soluzione

Esercizio 1:

Scrivere una funzione stampaLista che ricevuta una ListaDiInteri, la stampi a video in questo modo:
1 -> 2 -> 3 -> 4 -> / /
Usare questa funzione per verificare le funzioni dei prossimi esercizi.

Soluzione:

```
typedef struct El {
  int info;
  struct El *next;
} ElementoListaInt;
typedef ElementoListaInt* ListaDiInteri;


main() {

  int i, n, res;
  ListaDiInteri testa, aux;

  printf("Inserisci la lunghezza della lista: ");
  res=scanf("%d", &n);
  if(res<1) return;

  if(n>0) {
    testa = malloc(sizeof(ElementoListaInt));

    /* creazione ed inizializzazione */
    aux = testa;
    for(i=1; i<n; i++) {
      aux->info = i;
      aux->next = malloc(sizeof(ElementoListaInt));
      aux = aux->next;
    }
    aux->info=n;
    aux->next=NULL; /* la coda punta a null */

    /* scorriamo la lista dall'inizio per stamparla. La testa deve restare immutata
*/
    aux=testa;
    while(aux!=NULL) {
      printf("%d -> ", aux->info);
      aux = aux->next;
    }
    printf("//\n");

    /* Scorriamo la lista per deallocarla. La testa puo' cambiare */
    while(testa!=NULL) {
      aux = testa->next;
      free(testa);
```

```
      testa=aux;
    }
  }


}
```

Esercizio 4:

Si consideri una lista di interi, definire una funzione che inserisce un valore intero in coda ad una lista.

Soluzione:

```
void inserisciInCoda(lista *l, int x)
{
  lista tmp, corr, prec;
  tmp = malloc (sizeof(nodo));
  tmp -> info = i;
  tmp -> prox = NULL;
  if(*l == NULL)
    *l = tmp;
  else {
    prec = NULL;
    corr = *l;
    /* individuo l'ultimo elemento della lista */
    while(corr != NULL)
    {
       prec = corr;
       corr = corr->next;
    }
    /* collego l'ultimo elemento della lista con il nuovo
    elemento */
    prec->next = tmp;
  }
}
```

Esercizio 6:

Scrivere una funzione (una versione iterativa ed una ricorsiva) in cui dato un valore intero x ed una lista di interi, si conta le occorrenze di x nella lista.

Soluzione iterativa:

```
int calcolaOcc(TipoLista l, int val)
{
   int cont = 0;
   while ((l != NULL) && (l->info <= val))
   {
      if (l->info == val) cont++;
         l = l->next;
   }
   return cont;
}

int problema(pTipoLista L, int val)
```

```
{
  int occorrenze = 0;
  while (L != NULL)
  {
     occorrenze = occorrenze + calcolaOcc(L->pinfo,val);
     L = L->pnext;
 }
 return occorrenze;
}
```

Esercizio 9:

Definire una procedura (sia iterativa che ricorsiva) 'elimina' che ricevuta una ListaDiInteri e un intero X, elimini i primi X elementi e restituisca la lista modificata.

Soluzione:

```
typedef struct El {
  int info;
  struct El *next;
}  ElementoListaInt;
typedef ElementoListaInt *ListaDiInteri;


/*
  Questa funzione scorre iterativamente la lista, salvandosi di volta in volta
  il puntatore al successivo e quindi liberando la preesistente testa.
  Quando ne ha liberati il numero richiesto, ritorna la nuova testa al chiamante.
*/
ListaDiInteri elimina_iter(ListaDiInteri testa, int x) {
  ListaDiInteri aux;

  while(testa!=NULL && x>0) {
    aux = testa->next;
    free(testa);
    testa = aux;
    x--;
  }

  return testa;
}

/*
  Questa e' la stessa funzione di prima, modificata per ricevere la testa per
  riferimento e modificarla direttamente, invece di ritornare la nuova testa.
  Notate il nuovo controllo sulla validita' del riferimento passato e
  l'aggiornamento del riferimento quando si modifica la testa.
*/
void elimina_iter_riferimento(ListaDiInteri* p_testa, int x) {
  ListaDiInteri aux, testa;
  if(p_testa!=NULL) {
    testa = *p_testa;
    while(testa!=NULL && x>0) {
      aux = testa->next;
      free(testa);
      testa = aux;
      *p_testa = aux;
      x--;
```

```
    }
  }
}


/*
  Questa funzione ricorsiva libera il primo elemento se e solo se gli viene
  passato un numero maggiore di 0.
  In questo caso si salva il puntatore al successivo, elimina la testa e ricorre
  passando il nuovo puntatore e il valore x decrementato.
  Quando invece gli viene passato 0 ritorna semplicemente la testa della lista
  passata, che viene quindi ritornata da tutte le chiamate precedenti, chiudendo
  la ricorsione.
*/
ListaDiInteri elimina_ric(ListaDiInteri testa, int x) {

  ListaDiInteri aux;

  if(x>0 && testa!=NULL) {
    aux = testa->next;
    free(testa);
    return elimina_ric(aux, x-1);
  }

  return testa;
}

/*
  Questa e' la stessa funzione di prima, modificata per ricevere la testa per
  riferimento e modificarla direttamente, invece di ritornare la nuova testa.
  Notate il nuovo controllo sulla validita' del riferimento passato e
  l'aggiornamento del riferimento quando si modifica la testa.
*/
void elimina_ric_riferimento(ListaDiInteri* p_testa, int x) {

  ListaDiInteri aux;

  if(x>0 && p_testa!=NULL && (*p_testa)!=NULL) {
    aux = (*p_testa)->next;
    free(*p_testa);
    *p_testa = aux;
    elimina_ric_riferimento(p_testa, x-1);
  }
}


ListaDiInteri creaListaInTesta();
ListaDiInteri creaListaInCoda();
void deallocaLista(ListaDiInteri testa);
void stampaLista(ListaDiInteri lista);


int main() {
  ListaDiInteri lista = creaListaInCoda();
  stampaLista(lista);
  lista = elimina_iter(lista, 2);
  stampaLista(lista);
  elimina_iter_riferimento(&lista, 2);
  stampaLista(lista);
```

```
  lista = elimina_ric(lista, 3);
  stampaLista(lista);
  elimina_ric_riferimento(&lista, 3);
  stampaLista(lista);
  deallocaLista(lista);
  return 0;
}
```

Esercizio 13:

Definire una funzione ordinaLista che modifica una ListaDiInteri data ordinandola in modo crescente.
La funzione non deve usare allocazione dinamica della memoria (malloc e free).
Suggerimento: la testa puo' cambiare? Quindi come deve essere passata la lista alla funzione?

Soluzione:

```
#include <stdio.h>
#include <stdlib.h>


typedef struct El {
  int info;
  struct El *next;
}  ElementoListaInt;
typedef ElementoListaInt *ListaDiInteri;

/*
  La funzione ordinaLista implementa un classico algoritmo di ordinamento
  (selection sort) con complessita' O(n^2).
  L'algoritmo e' lo stesso usato in precedenza per ordinare vettori:
    ad ogni iterazione, a partire dall'inizio
      1) trova l'elemento minore a destra dall'elemento attuale
      2) lo scambia con l'elemento attuale
      3) avanza la posizione attuale
  Poiche' qui abbiamo a che fare con liste invece che con vettori, per ogni
  elemento (attuale, minimo, cursore) dovremo tenerci sia il puntatore
  all'elemento che quello al suo precedente.
  Inoltre ci serve di tenerci da parte la testa della lista riordinata.
  E un puntatore temporaneo per lo swap.
  Totale: 8 puntatori.
  E' sicuramente possibile scrivere una versione assai piu' compatta di questa
  ma in questo modo vengono messe in evidenza le similarita' con l'algoritmo
  originale su vettori.
*/
ListaDiInteri ordinaLista(ListaDiInteri lista) {
  ListaDiInteri ret; /* nuova testa, elemento di ritorno */
  ListaDiInteri act, prec; /* elemento attuale e suo precedente */
  ListaDiInteri aux, paux; /* cursore di ricerca e suo precedente */
  ListaDiInteri min, pmin; /* elemento minimo e suo precedente */
  ListaDiInteri temp; /* temporaneo per lo swap */

  /* iniziamo dalla testa, ovviamente */
  ret = NULL;
```

```
  act = lista;
  prec = NULL;

  while(act!=NULL) {

    /* Il minimo e' il primo elemento, all'inizio */
    min = act;
    pmin = prec;

    /* e quindi il cursore con cui cerchiamo il nuovo minimo parte dal successivo
*/
    paux = act;
    aux = act->next; /* esiste perche' siamo entrati nello while */


    /* scansione della lista per ricerca del nuovo minimo */
    while(aux!=NULL) {
      if(aux->info < min->info) {
        min = aux;
        pmin = paux;
      }
      /* Scorriamo il cursore sull'elemento successivo, questo possibile perche'
      siamo entrati nello while. Equivale ad i++ nei vettori. */
      paux = aux;
      aux = aux->next;
    }

    /* se abbiamo trovato un nuovo minimo, lo scambiamo con act */
    if(min!=act && min!=NULL && act!=NULL) {
      /* sistemo i precedenti */
      if(prec!=NULL)
        prec->next = min;

      if(pmin!=NULL)
        pmin->next = act;

      /* swap dei next di min e act*/
      temp = min->next;
      min->next = act->next;
      act->next = temp;

      /* e quindi adesso min e' il nuovo elemento corrente */
      act=min;

    }

    /* la prima volta che arriviamo qui act e' il minimo assoluto, quindi e' la
    nuova testa della futura lista ordinata e quindi la salviamo */
    if(ret==NULL)
      ret = act;

    /* alla fine iteriamo a partire dall'elemento seguente  */
    prec = act;
    if(act!=NULL)
      act = act->next;

  }

  return ret;
```

```
}

/*
  Per i commenti su queste funzioni di base, vedere 01-02-03-ListaBase.c
*/
ListaDiInteri creaListaInTesta();
ListaDiInteri creaListaInCoda();
void deallocaLista(ListaDiInteri testa);
void stampaLista(ListaDiInteri lista);


int main() {

  ListaDiInteri lista = creaListaInCoda();

  printf("Lista originale: ");
  stampaLista(lista);

  lista = ordinaLista(lista);

  printf("Lista ordinata: ");
  stampaLista(lista);

  deallocaLista(lista);

  return 0;

}

ListaDiInteri creaListaInTesta() {
  ListaDiInteri testa=NULL, nuovo;
  int intero;

  do {
    printf("Inserire il prossimo elemento della lista: ");
    scanf("%d", &intero);
    if (intero>=0) {
      nuovo = malloc(sizeof(ElementoListaInt));
      nuovo->info = intero;
      nuovo->next = testa;
      testa = nuovo;
    }
  } while(intero>=0);

  return testa;
}

ListaDiInteri creaListaInCoda() {
  ListaDiInteri testa=NULL, last=NULL, nuovo;
  int intero;

  do {
    printf("Inserire il prossimo elemento della lista: ");
    scanf("%d", &intero);
    if (intero>=0) {
      nuovo = malloc(sizeof(ElementoListaInt));
      nuovo->info = intero;
      nuovo->next = NULL;
```

```
      if(last!=NULL)
        last->next = nuovo;
      else
        testa = nuovo;

      last = nuovo;

    }
  } while(intero>=0);

  return testa;
}

void deallocaLista(ListaDiInteri testa) {
  ListaDiInteri temp;
  while(testa!=NULL) {
    temp = testa->next;
    free(testa);
    testa = temp;
  }
}

void stampaLista(ListaDiInteri lista)
{
  while (lista != NULL)
  {
    printf("%d -->", lista->info);
    lista = lista->next;
  }
  printf("//\n");
}
```

Esercizio 14:

Definire una procedura 'merge' che date due ListaDiInteri ordinate, restituisca una nuova ListaDiInteri ordinata contenente tutti gli elementi delle due liste. Le liste originali devono restare immutate.

Soluzione:

```
#include <stdio.h>
#include <stdlib.h>


typedef struct El {
  int info;
  struct El *next;
}  ElementoListaInt;
typedef ElementoListaInt *ListaDiInteri;


ListaDiInteri dupElement(ListaDiInteri el) {
  ListaDiInteri ret = NULL;
  if(el!=NULL) {
    ret = malloc(sizeof(ElementoListaInt));
    ret->info=el->info;
```

```
    ret->next=NULL;
  }
  return ret;
}

/*
  Questa funzione prima di tutto duplica e concatena le due liste passate e quindi
  ritorna la nuova lista, dopo averla ordinata.
*/
ListaDiInteri merge(ListaDiInteri l1, ListaDiInteri l2) {

  ListaDiInteri ret=NULL, aux, last=NULL;

  /* copiamo la prima lista */
  while(l1!=NULL) {
    aux = dupElement(l1);

    if(ret==NULL) /* prima iterazione, salviamo la testa */
      ret=aux;
    else /* se non e' la prima iterazione, last e' sicuramente settato */
      last->next = aux;

    /* aggiorniamo l'ultimo e avanziamo la lista */
    last = aux;
    l1 = l1->next;
  }

  /* copiamo la seconda lista */
  while(l2!=NULL) {
    aux = dupElement(l2);

    if(ret==NULL) /* prima iterazione, la lista l1 era vuota, salviamo la testa */
      ret=aux;
    else /* se non e' la prima iterazione, last e' sicuramente settato */
      last->next = aux;

    /* aggiorniamo l'ultimo e avanziamo la lista */
    last = aux;
    l2 = l2->next;
  }

  /* ret contiene tutti gli elementi delle due liste originali e va quindi
ordinata. */
  return ordinaLista(ret);
}


/*
  Per i commenti su queste funzioni di base, vedere 01-02-03-ListaBase.c
*/
ListaDiInteri creaListaInTesta();
ListaDiInteri creaListaInCoda();
void deallocaLista(ListaDiInteri testa);
void stampaLista(ListaDiInteri lista);


int main() {

  ListaDiInteri lista1, lista2, lista3;
```

```
  lista1 = creaListaInCoda();
  printf("Lista 1: ");
  stampaLista(lista1);

  lista2 = creaListaInCoda();
  printf("Lista 2: ");
  stampaLista(lista2);

  lista3 = merge(lista1, lista2);
  printf("Lista 3: ");
  stampaLista(lista3);

  printf("Lista 1: ");
  stampaLista(lista1);
  printf("Lista 2: ");
  stampaLista(lista2);

  deallocaLista(lista1);
  deallocaLista(lista2);
  deallocaLista(lista3);

  return 0;

}
```