Burt Rosenberg

Answer to Problem Set 2

Out: 15 September, 1995

```
/*
 *  Answer to Problem Set 2
 *  Univ. of Miami, Math 220/317
 *  Fall 1995
 *  Prof. B. Rosenberg
 *
 *  A program that duplicates a linked list and
 *  outputs the elements in ascending order by
 *  iterative find and deletes.
*/

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

struct MyList
{
   int the_number ;
   struct MyList * next ;
} ;

struct MyList * create_MyList(void)
{
   /* an empty list as a dummy header. */
   struct MyList * ml ;
   ml = (struct MyList *) malloc(sizeof(struct MyList)) ;
   ml->the_number = -1 ;
   ml->next = NULL ;
   return(ml) ;
}

void print_MyList( struct MyList * ml )
{
   /* skip the dummy header */
   ml = ml-> next ;

   if ( ml==NULL )
   {
     /* if the list is empty, say so politely */
     printf("The list is empty.\n") ;
     /* the following jumping out of a subroutine is
        often discouraged. However, I think that it
```

```
        is OK to use in cases of error conditions.
     */
     return ;
   }

   /* ASSERT: the list is not empty */
   while (ml!=NULL)
   {
     printf("%d ", ml->the_number ) ;
     ml = ml->next ;
   }
   printf("\n") ;
}

void push_MyList( struct MyList * ml, int new_int )
{
   /* add value new_int to the head of the list. */
   struct MyList * temp ;
   temp = (struct MyList *) malloc(sizeof(struct MyList)) ;
   temp->the_number = new_int ;
   temp->next = ml->next ;
   ml->next = temp ;
   return ;
}

struct MyList * dup_MyList( struct MyList * ml )
{
   /* duplicate the list ml */

   struct MyList * temp ;
   /* create the new list */
   temp = create_MyList() ;

   ml = ml->next ;
   while ( ml!=NULL )
   {
      /* run through the original list, pushing
         onto the new list as you go.
      */
      push_MyList( temp, ml->the_number ) ;
      ml = ml->next ;
   }

   /* return a pointer to the new list */
```

```
   return(temp) ;
}

void reverse_MyList( struct MyList * ml )
{
   struct MyList * the_dummy ;
   struct MyList * the_prev, * the_next ;

   /* remember the dummy */
   the_dummy = ml ;

   /* run down the list, reversing pointers as you go */
   ml = ml->next ;
   the_prev = NULL ;
   /* loop invariant: the_prev points to the start of
      the already processed portion of the list. This
      portion has been reversed.
      ml points to the rest of the original list which
      has not yet been processed.
   */
   while ( ml!=NULL )
   {
      /* process the ml element */
      the_next = ml->next ;  /* remember the element after ml */
      ml->next = the_prev ;  /* ml added to head of reversed list */
      the_prev = ml ;        /* the reverse list now starts with ml */
      ml = the_next ;        /* the element after ml is next to do */
   }
   /* the list is now reversed */

   /* glue the dummy back onto the front */
   the_dummy->next = the_prev ;
   return ;
}

int min_int( struct MyList * ml )
{
   int min_so_far ;
   ml = ml->next ;
   min_so_far = ml->the_number ;

   /* loop invariant: min_so_far is the minimum element
      on the list up to and including the element pointed
      to by ml.
```

```
   */
   while ( ml!=NULL )
   {
      /* still more to check */
      if ( ml->the_number < min_so_far )
         /* a new minimum */
         min_so_far = ml->the_number ;
      /* go on ... */
      ml = ml->next ;
   }

   /* min_so_far is the minimum on the list up to
      the end of the list! We are done.
   */
   return(min_so_far) ;
}

int delete_MyList( struct MyList * ml, int item_to_delete )
{
   struct MyList * the_trailing_pointer ;

   the_trailing_pointer = ml ;
   ml = ml->next ;
   /* this will mostly be true: the_trailing_pointer->next==ml,
      that is, it will trail by one the pointer ml.
   */
   while ( ml!=NULL )
   {
      if ( ml->the_number == item_to_delete )
      {
         /* we have found the item to delete, ml points to it,
            and the_trailing_pointer points to one before it
         */
         /* cut ... */
         the_trailing_pointer->next = ml->next ;
         /* kill ... */
         free(ml) ;
         /* and leave the while loop */
         break ;
      }
      /* march on throught the list */
      the_trailing_pointer = ml ;
      ml = ml->next ;
      /* it is still true that the_trailing_pointer->next==ml */
```

```
   }

   /* return TRUE if an item found and deleted, FALSE otherwise */
   return( ml!=NULL ) ;
}

int is_empty_MyList( struct MyList * ml )
{
   /* return TRUE if ml is an empty list, FAlSE otherwise */
   return( ml->next==NULL ) ;
}

main()
{
   struct MyList * anchor ;
   struct MyList * the_other_list ;
   int j ;

   /* create a sample list */
   anchor = create_MyList() ;
   for ( j=0; j<15; j++ )
      push_MyList( anchor, (j*17)%13 ) ;
   print_MyList(anchor) ;

   /* dup it, reverse it, print it */
   the_other_list = dup_MyList( anchor ) ;
   print_MyList(the_other_list) ;
   reverse_MyList( the_other_list ) ;
   print_MyList(the_other_list) ;

   /* repeatedly find, print and delete the minimum
      item on the list
   */
   while ( !is_empty_MyList(the_other_list) )
   {
      /* find the min. */
      j = min_int( the_other_list ) ;
      /* print it */
      printf("%d ", j ) ;
      /* delete it. Note: I will use the cast to void
         to emphasize to the reader that the return value of
         delete_MyList is being thrown away.
      */
      (void) delete_MyList( the_other_list, j ) ;
```

```
   }
   printf("\n") ;
}
```

```
cs> cc -o as2 as2.c
cs> as2
4 0 9 5 1 10 6 2 11 7 3 12 8 4 0
0 4 8 12 3 7 11 2 6 10 1 5 9 0 4
4 0 9 5 1 10 6 2 11 7 3 12 8 4 0
0 0 1 2 3 4 4 5 6 7 8 9 10 11 12
cs>
```