```
/*Assignment #4

Topic(s):       List ADT Implementation
Date assigned:  9/29/10
Date due:       10/11/10
Points:         50

For this assignment, you are to implement the List ADT in a file called list.c
using the header file list.h. All of the data structures and function prototypes
are defined in list.h. Further, each function prototype has been described to
the point that you should be able to implement each list function in the file
list.c. Our first list implementation will be using arrays. We will be
reusing this code later on, so make sure you have completely tested and
debugged each operation.

Notes:

1) Create a C project called StaticList in Eclipse 3.6.

2) The StaticList project is to have the following folders:
a) Headers
b) Sources
c) Includes
d) Binaries
e) Testfiles

3) Create a makefile at the same level as the folders in 2

4) In the Sources folder, create a driver called staticlistdriver.c

5) Submit a zipped up tarfile called 04punetid.tar.gz, using the
   submit script, which contains your entire project.

6) Using the color printer (double-sided and stapled), print your source code
   in the following order:
a) The driver staticlistdriver.c you used to test your program
b) list.h
c) list.c
d) the makefile
```

```
7) Do not make any changes to list.h unless we all agree on the changes at
   which time I will post the new list.h file on zeus. Use the list.h file
   found in /home/CS300Public/2010 on zeus.

8) By Tuesday 10/5/10 5pm you need to complete each of the following functions:
a) lstCreate
b) lstDispose
c) lstSize
d) lstIsFull
e) lstFirst
f) lstNext
g) lstInsertAfter
h) lstHasHext

Finally, hand in a printout and submit your code.

9) Let's take a look at list.h */

/*******************************************************************************
 File name:     list.h (Version 1.0)
 Author:        Doug Ryan
 Date:          9/29/10
 Class:         CS300
 Assignment:    List Implementation
 Purpose:       This file defines the constants, data structures, and function
                prototypes for implementing a list data structure. In essence,
                the list API is defined for other modules.

 Modifications:
 *******************************************************************************/

#ifndef LIST_H
#define LIST_H
#define NDEBUG

#define MAX_KEY_SIZE    32        // Each list node can contain a key
#define MAX_LIST_ELEMENTS 1024

#define TRUE  1
#define FALSE   0
```

```c
// List error codes for each function to use

#define NO_ERROR    0
#define NO_LIST_CREATE 1
#define EMPTY_LIST  2
#define FULL_LIST     3
#define NO_PREDECESSOR 4
#define NO_SUCCESSOR  5
#define NO_NEXT     6

// Possible datatypes for later use

#define CHARACTER_VALUE 0
#define INTEGER_VALUE    1
#define FLOAT_VALUE      2

// User-defined datatypes for easier reading

typedef short int BOOLEAN;
typedef short int ERRORCODE;

/*  Possible datatype for later use
typedef struct
         {
           union
           {
             char              charValue;
             unsigned int      intValue;
             float             floatValue;
           };
           unsigned short whichOne;
         } DATATYPE; */

typedef int DATATYPE;

// A list element is a (key, data) pair


typedef struct
```

```c
         {
           char key[MAX_KEY_SIZE];
           DATATYPE data;
         } ListElement;


// A list is an array of ListElements where the current pointer and number
// of elements are maintained at all times

typedef struct
         {
           ListElement listElements[MAX_LIST_ELEMENTS];
           int current;
           int numElements;
         } List;


/*
 * Specification: List
 *
 * Elements:     The elements are of type ListElement
 *
 * Structure:    There is a linear relationship among the elements. That is,
 *               each list element has a unique predecessor and a unique
 *               successor except the first and last elements. Each element
 *               has a unique position which for the array implementation
 *               will be position 0, 1, ... Given an element at position
 *               k, then the element at position k - 1 is the predecessor and
 *               the element at k + 1 is the successor.
 *
 * Domain:       The number of elements in the domain is bounded
 *
 * Operations:   The following operations are defined and are to be implemented
 *               in list.c
 */

 /******************************************************************************
  *                Allocation and Deallocation
  ******************************************************************************/
extern ERRORCODE    lstCreate (List **);
```

```
            // results: If list L can be created, then L exists and
            //          is empty returning NO_ERROR; otherwise,
            //          NO_LIST_CREATE is returned

extern ERRORCODE    lstDispose (List **);
            // results: List no longer exists

/*******************************************************************************
 *                  Checking number of elements in list
 ******************************************************************************/
extern int        lstSize (List *);
            // results: Returns the number of elements in the list

extern BOOLEAN      lstIsFull (List *);
            // results: If list is full, return true;
            //          otherwise, return false

/*******************************************************************************
 *                  Peek Operations
 ******************************************************************************/
extern ERRORCODE    lstPeek (List *, ListElement *);
            // requires: List is not empty
            // results:  The value of the current element is
            //           returned through the argument list
            //           IMPORTANT: Do not change current


extern ERRORCODE    lstPeekPrev (List *, ListElement *);
            // requires: List contains two or more elements and
            //           current is not the first element
            // results:  The data value of current's predecessor is returned
            //           through the argument list.
            //           IMPORTANT: Do not change current

extern ERRORCODE    lstPeekNext (List *, ListElement *);
            // requires: List contains two or more elements and
            //           current is not the last element
            // results:  The data value of current's successor is returned
            //           through the argument list.
            //           IMPORTANT: Do not change current
```

```c
/****************************************************************************
 *                  Retrieving values and updating current
 ****************************************************************************/

extern ERRORCODE    lstFirst(List *, ListElement *);
            // requires: List is not empty
            // results:  The value of the first element is returned
            //           IMPORTANT: Current is changed to first
            //                      if it exists

extern ERRORCODE    lstLast(List *, ListElement *);
            // requires: List is not empty
            // results:  The value of the last element is returned
            //           IMPORTANT: Current is changed to
            //                      last if it exists

extern ERRORCODE    lstNext(List *, ListElement *);
            // requires: List is not empty, and current is not past the end
            //           of the list
            // results:  The value of the current element is returned
            //            IMPORTANT: Current is changed to the successor
            //                       of the current element


extern ERRORCODE    lstPrev(List *, ListElement *);
            // requires: List is not empty, the list contains
            //           two or more elements and current is not first
            // results:  The value of the predecessor of current is returned
            //           IMPORTANT: Current is changed to previous
            //                      if it exists

extern BOOLEAN      lstFindKey (List *, char *, ListElement *);
            // results:  Returns the ListElement through the
            //           argument list if the key is found in a
            //           node; otherwise, false is returned
            //           IMPORTANT: Current is changed to the node
            //                      containing the key if it exists

/****************************************************************************
```

```
 *                   Insertion, Deletion, and Updating
 ******************************************************************************/
extern ERRORCODE    lstDeleteCurrent (List *, ListElement *);
            // requires:  List is not empty
            // results:   The current element is deleted and its
            //            successor and predecessor become each
            //            others successor and predecessor. If the
            //            deleted element had a predecessor, then
            //            make it the new current element; otherwise,
            //            make the first element current if it exists.
            //            The deleted element is returned through the argument
            //            list.

extern ERRORCODE    lstInsertAfter (List *, ListElement);
            // requires: List is not full
            // results:  if the list is not empty, insert the new
            //           element as the successor of the current
            //           element and make the inserted element the
            //           current element; otherwise, insert element
            //           and make it current. The new element is inserted into
            //           the proper place and all other elements are shifted
            //           down the list.

extern ERRORCODE    lstInsertBefore (List *, ListElement);
            // requires: List is not full
            // results:  If the list is not empty, insert the new
            //           element as the predecessor of the current
            //           element and make the inserted element the
            //           current element; otherwise, insert element
            //           and make it current. The new element is inserted into
            //           the proper place and all other elements are shifted
            //           down the list.


extern ERRORCODE    lstUpdateCurrent (List *, ListElement);
            // requires: List is not empty
            // results:  The value of ListElement is copied into the
            //           current element

/******************************************************************************
```

```
 *                    List Testing
 ******************************************************************************/

extern BOOLEAN      lstHasNext(List *);
            // results:  Returns true if there are more elements when traversing
            //           the list in a forward direction; otherwise, false is
            //           returned. In other words, returns true if lstNext would
            //           would return an actual value.


extern BOOLEAN      lstHasPrev(List *);
            // results:  Returns true if the current node has a
            //           predecessor; otherwise, false is returned


#endif
```