C Programming Using a Generic Data Structure

For this assignment, you will implement a program that manipulates a queue of simple objects. You will employ the generic `Queue` from the previous project. In addition, your program will include implementations of the following new types:

`Place`	structure type encapsulate some information about a geographic feaure
`PlaceDT`	wrapper type to "duct-tape" a `Place` object to a `Queue` node
`PlaceQueue`	wrapper supplying a suitable queue interface for a `Queue` of `Place` objects

The best approach to implementing a generic list in C is discussed in CS 2505, as is the way to make use of such a generic list by creating a "duct-tape wrapper" type to attach user data objects to a node. Rather than repeat that discussion here, you will find links to the relevant notes from CS 2505 on the Assignments page of the course website. You should make sure you understand those notes before trying to implement your solution.

Your solution must conform exactly to the descriptions below, and your code must be organized exactly as described below.

Because your implementations will be compiled with a test harness, there will be mandatory interfaces. You will find header files containing declarations for the `Place`, `PlaceDT` and `PlaceQueue` types on the course website, and repeated below. You must not make any changes to the specified interface of any of those types.

A compiled implementation of the `Queue` type will be supplied on the course website, as header file, `Queue.h`, and a 32-bit Linux object file, `Queue.o`. You can compile this with your solution by using the following command:

```
gcc -o p3 -std=c99 -Wall -m32 driver.c
         <any other .c files you need> Place.c PlaceQueue.c Queue.o
```

The command above assumes that:

- you want to call your executable `p3`
- you have written a test driver called `driver.c`
- your system is set up to support building 32-bit executables (the rlogin cluster is)
- you don't have any other C source files

You can easily modify the given command if any of those assumptions are incorrect.

Memory management requirements

You must allocate the wrapper objects that make up the queue dynamically. As before, the queue itself will perform no memory allocation or deallocation operations.

As is the case with any non-garbage-collected language, it is your responsibility to deallocate all memory that you allocate dynamically as soon as you are done with it. The test code will not detect whether you have done this, but a TA will review your implementation to verify that you have done so.

What to Submit

You will create an uncompressed tar file containing your three implementation files: `Place.c` and `PlaceQueue.c`, and nothing else, and submit that tar file to the Curator. Note that we will use a reference implementation of the queue in our own testing.

This assignment will be graded automatically. You will be allowed up to ten submissions for this assignment, so use them wisely. Test your functions thoroughly before submitting them. Make sure that your functions produce correct results for every logically valid test case you can think of.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//     On my honor:
//
//     - I have not discussed the C language code in my program with
//       anyone other than my instructor or the teaching assistants
//       assigned to this course.
//
//     - I have not used C language code obtained from another student,
//       or any other unauthorized source, either modified or unmodified.
//
//     - If any C language code or documentation used in my program
//       was obtained from another source, such as a text book or course
//       notes, that has been clearly noted with a proper citation in
//       the comments of my program.
//
//     <Student Name>
```

Interfaces:

Place.h:

```
#define MAXNAMELENGTH  100   // maximum number of chars in a place name
#define LATITUDELENGTH   7   // maximum number of chars in a latitude string
#define LONGITUDELENGTH  8   // maximum number of chars in a longitude string


/**  Every properly initialized Place object must satisfy the following
 *   conditions:
 *
 *    1.  Each array is a zero-terminated ASCII strings.
 *    2.  Each array is of the minimum dimension to store its contents.
 *    3.  FID is initialized
 */
struct _Place {
   uint32_t FID;
   char*    Name;
   char*    Latitude;
   char*    Longitude;
};

typedef struct _Place Place;

/**  Place_Set() initializes a new Place object.
 *   Pre:      pPlace points to a Place object
 *             FID has been initialized
 *             Name points to a zero-terminated ASCII string
 *             Lat points to a zero-terminated ASCII string
 *             Long points to a zero-terminated ASCII string
 *   Post:     pPlace->FID == FID
 *             pPlace->Name is a copy of Name (i.e., pPlace->Name != Name)
 *             pPlace->Latitude is a copy of Lat
 *                    (i.e., pPlace->Latitude != Lat)
 *             pPlace->Longitude is a copy of Long
 *                    (i.e., pPlace->Longitude != Long)
 *   Returns:  false if the object could not be properly initialized;
 *             true otherwise
 *  Called by: unknown client code that creates Place objects
 */
bool Place_Set(Place* const pPlace, uint32_t FID, char* Name, char* Lat,
                                                              char* Long);

/**  Place_Equals() indicates whether two Place objects have equal
 *   data members.
 *   Pre:      pLeft and pRight point to proper Place objects
 *   Post:     *pLeft and *pRight are unchanged
 *   Returns:  true if each member of *pLeft equals the corresponding
 *             member of *pRight;
 *             false otherwise
 *  Called by: PlaceQueue_Contains()
 */
bool  Place_Equals(const Place* const pLeft, const Place* const pRight);
```

PlaceQueue.h:

```
////////////////////////////////////////////////////////////// PlaceDT type
//  PlaceDT provides a "wrapper" to attach a user data object of type
//  Place to a queue node.
//
//  Warning: the PlaceQueue implementation receives pointers to user
//  data (Place) objects, and it does not make copies of those objects.
//  Therefore, the user of this code must ensure that the Place objects
//  that are enqueued are not destroyed before they are removed from
//  the queue.  Failure to do this may result in runtime errors.
//
struct _PlaceDT {
   Place* pItem;     // points to the "payload", a Place object supplied
                     //    when PlaceQueue_Schedule() is called
   QNode  Node;      // Queue node to provide list structure
};

typedef struct _PlaceDT PlaceDT;


/**  PlaceDT_Set() initializes a raw PlaceDT object.
 *   Pre:      pPlaceDT points to a PlaceDT object
 *             pPlace points to a properly-initialized Place object
 *   Post:     pPlaceDT->pItem == pPlace
 *             The pointers in pPlaceDT->Node are NULL
 *   Returns:  false if the object could not be properly initialized;
 *             true otherwise
 *   Note:     Whether *pPlace was allocated dynamically or statically
 *             is unknown.  *pPlace is owned by the client who uses
 *             the PlaceQueue implementation, and deallocation of it
 *             is the responsibility of that client.
 *  Called by: PlaceQueue_Schedule()
 */
bool PlaceDT_Set(PlaceDT* const pPlaceDT, Place* pPlace);


////////////////////////////////////////////////////////////// PlaceQueue type
//  PlaceQueue provides the means to create and manipulate a queue of
//  Place objects (e.g., for implementing a "bucket list").
//
//  Warning: the PlaceQueue implementation receives pointers to user
//  data (Place) objects, and it does not make copies of those objects.
//  Therefore, the user of this code must ensure that the Place objects
//  that are enqueued are not destroyed before they are removed from
//  the queue.  Failure to do this may result in runtime errors.
//
//  Uses:  Queue type implemented in Queue.h and Queue.c.
//struct _PlaceQueue {
   Queue Q;           // a Queue object to provide the connections
};

typedef struct _PlaceQueue PlaceQueue;
```

```
/**  PlaceQueue_Init() initializes a raw PlaceQueue object.
 *   Pre:      pPQ points to a PlaceQueue object
 *   Post:     pPQ->Q has been initialized to an empty state
 *   Returns:  false if the object could not be properly initialized;
 *             true otherwise
 *  Called by: unknown client code that uses a PlaceQueue object
 */
bool PlaceQueue_Init(PlaceQueue* const pPQ);

/**  PlaceQueue_Schedule() inserts a Place object into the queue.
 *   Pre:      pPQ points to a proper PlaceQueue object
 *             pPlace points to a properly-initialized Place object
 *   Post:     *pPlace has been inserted at the rear of the queue
 *   Returns:  false if the insertion could not be performed; true otherwise
 *  Called by: unknown client code that uses a PlaceQueue object
 */
bool   PlaceQueue_Schedule(PlaceQueue* const pPQ, Place* const pPlace);

/**  PlaceQueue_Visit() pops the front element from the queue and returns it.
 *   Pre:      pPQ points to a proper PlaceQueue object
 *   Post:     pPQ points to a proper PlaceQueue object, with one less element
 *                 (unless the queue was empty)
 *   Returns:  pointer to the removed object;
 *             NULL if no object could be removed
 *   Note:     PlaceDT objects are created and owned by a PlaceQueue object;
 *             therefore, it is the responsibility of the owning PlaceQueue
 *             object to properly deallocate them.
 *  Called by: unknown client code that uses a PlaceQueue object
*/
Place* PlaceQueue_Visit(PlaceQueue* const pPQ);

/**  PlaceQueue_Contains() indiates whether the queue contains a specific
 *   Place object.
 *   Pre:      pPQ points to a proper PlaceQueue object
 *             pPlace points to a proper Place object
 *   Post:     *pPQ and *pPlace are unchanged
 *   Returns:  true if the queue contains a Place object that matches
 *             *pPlace, according to Place_Equals();
 *             false otherwise
 *  Called by: unknown client code that uses a PlaceQueue object
 */
bool   PlaceQueue_Contains(const PlaceQueue* const pPQ,
                           const Place* const pPlace);

/**  PlaceQueue_Empty() indicates whether the queue contains elements.
 *   Pre:      pPQ points to a proper PlaceQueue object
 *   Post:     pPQ points to an unchanged PlaceQueue object
 *   Returns:  true iff pPQ->Q is empty
 *   Note:     PlaceDT objects are created and owned by a PlaceQueue object;
 *             therefore, it is the responsibility of the owning PlaceQueue
 *             object to properly deallocate them.
 *  Called by: unknown client code that uses a PlaceQueue object
 */
bool   PlaceQueue_Empty(const PlaceQueue* const pPQ);
```