Lab 5 - DLL Template using Function Pointers

Due Date: Sunday March 20, 2011 by 11:59pm

Background:
For those of you who have used C++ before, you may be familiar with the Template concept. It is an Object Oriented construct that allows you to write generalized code that will work on any data type. As we know, C is not inherently object oriented, but that shouldn't stop you from writing object oriented code. By utilizing function pointers you can still create general purpose code which is usable on multiple different types. For this assignment you will be implementing a doubly linked list (DLL) using function pointers. Once you are done, you will be able to use the same code on DLL of any data type with very little extra work.

Data Structure:
You will be using two data structures for this lab. One will be the internal data structure dll_node used by the dll_l, and the other is the data structure seen by the users of your dll_l library.

The internal dll node data structure is:

```
typedef struct DLL_NODE {
  void *key;                /* Key for node */
  void *value;              /* The value for the node */
  struct DLL_NODE *next;     /* pointer to next node */
  struct DLL_NODE *prev;    /* pointer to previous node */
} dll_node;
```

The node has two fields, key (used for comparison of nodes) and value is the actual value of the node. This will be the structure your DLL will use. The key and value will be passed to your library as void*s. The public data structure dll_l is:

```
typedef struct DLL {
        dll_node *head;           /* head node of the list */
        int (*cmp)(const void*, const void*); /* Compare function pointer */
        void (*free_key)(void*); /* takes a pointer to a key and frees it*/
        void (*free_value)(void*); /* takes a pointer to a value and frees it*/
} dll_l;
```

The dll_l is the data structure that the users of your library will see. It stores the head of the DLL and a function pointers for the compare function and free functions. The compare function defines how nodes in your DLL should be compared. The free_key and free_value functions define how keys are values (if they have allocated memory should be freed. These pointers can be NULL meaning you cannot free the memory associated with void*'s). This structure allows us to use the same dll_l for different data types just by using the appropriate compare function.

The Assignment:

You are to implement the dll_l library functions (60 points). We will be using function pointers to describe the generic behavior of these functions. You will need to implement the following functions:

<u>Basic functions (20 points)</u>

```
int dll_init_list(dll_l *list);
int dll_set_cmp(dll_l *list, int (*cmp)(void*, void*));
int dll_set_free_key(dll_l *list, void (*free_key)(void*) );
int dll_set_free_value(dll_l *list, void (*free_value)(void*) );
```

<u>Advanced Functions (40 points)</u>

```
int dll_insert(dll_l *list, void *key, void *value);
int dll_retrieve(dll_l *list, void *key, void **value);
int dll_map(dll_l *list, void (*func)(void*, void*), void* args);
int dll_delete(dll_l *list, void *key);
int dll_dispose(dll_l *list, void (*func)(void*, void*), void* args);
```

1. The dll_init function will be called by the user first, and will initialize the dll_l structure fields.
2. The dll_set_ functions will allow the user to set the function pointers defined in the dll_l.
3. The dll_insert function will insert (key, value) into the dll (in the order defined by the cmp function) . Note that the key will be used by the cmp function.
4. The dll_retrieve function will retrieve the value corresponding to the key given. Note that value is returned to the calling program through the address of the value
5. The dll_map function will apply the function, func to all the elements in the data structure. For example func can be a print function that defines how value in each node should be printed. The third argument args can become an input to one of the func arguments. See dll_string.c to see how dll_map is used by the writeToFile function. Apply function to value of the node.
6. The dll_delete function will delete the node given by the key. It will use any free_key or free_value function pointers (if not NULL)
7. The dll_dispose function will free all memory associated with the list. The func can define how memory in each value in a node should be freed. The function (as in map) applies to all the value fields in the node. This function can overwrite free_value function or can be left as null (in this case dispose will use free_key and free_value functions to free the data)

You can only use ONE pointer during the traversal to insert the node. Since the list is a DLL we always know who is behind the node. Once again, the dll_retrieve function will set the memory pointed to by value to the value that has the given key. The dll_map function will call the passed function on every node in the dll in order. The dll_delete function deletes the node pointed to by key. Again use ONE pointer during traversal and delete of the node. More information on these functions can be found in the documentation given in dll.h.

Insert will work by taking in key and value pointers and use the compare function. The compare function is provided by the user and takes in two void pointers. It should work like strcmp, returning a negative number for less, 0 for equal, and a positive number for greater. Your dll code will call the compare function on the keys to search and retrieve values from the dll.

The function pointer in the dll_map function takes in two void pointers. The first is the value of the node, and the second is for optional arguments. This way you can pass extra options to the function. For instance, if you wrote a function to output each value to a file you could have the extra argument be the FILE* to the open file.

Running the program:
Your library won't run on its own. It's meant to be used by other programs. We are providing code dll_string.c that will read in a file of words, insert them into the dll in order, and output the sorted list to a new file. It uses your dll code to insert the words in order, then uses the map function to print out each word to the output file. These file only tests insert and map, not retrieve.

There is also dll_int, which will test your insert, retrieve, and map functionality for data structure with ints. When you run the int solution you should get output that looks like the following (the "Key already exists!" line would be whatever string you print if insert detects you trying to insert something that was already there).

```
$ ./dll_int
Inserting...
Re-inserting. Should always fail...
Retrieving...
Retrieving bad values. Should always fail...
Mapping...
Map sees: 200
Map sees: 300
Map sees: 350
Map sees: 400
Map sees: 500
Map sees: 600
Map sees: 650
Map sees: 700
Map sees: 800
```

What can you do with a DLL? (30 points)
How about implementing a text editor? The application we have in mind (although we would not efficiently implement this) is the implementation of a text editor using the text buffer idea. When implementing a text editor, it is important to perform some of the standard editor functions like, insert text, delete text in constant or O(1) time. So placing text in an array does not work. Placing

text in a LL does not work (too much overhead). So how about a compromise? We can implement Text buffers using a dll of small text arrays (size 10 bytes). For instance we can think of a text editor that stores text of a file using a dll of small text arrays like this.

←[abcdeannne] ←→ [efgh□lmn] ←→ [ijklmnopqr] ←→[ijklmnerbe]

Where □ can be thought of as the curser position where text can be inserted or deleted. If any small buffer (say with a fixed capacity of 10 characters) ran out of space, a new buffer can be inserted and/or text can be distributed easily to neighboring buffers. There are many efficient ways to implement small text buffers (sometimes called gap buffers) but in this part we are not interested in the best way to do this, but rather implement ways to use our generic DLL to store the text in a way that is efficient to handle.

We can assume each dll_node with a text buffer has key (buffer number) and a dynamic array of 10 characters as the value. We are asking you to write a driver file called **dll_textbuffer.c** that will read the file **aliceinwonderland.txt** and store the file in a text buffer. You will be using dll_insert to create the text buffer. Then use dll_map to print the content of the DLL back to stdout. We are not providing you any starter code here so you can practice how to write your starter code. However, we will discuss few things in class/recitations about this.

Style Points: Please comment your code to show what you are doing. The least amount of comments is a short description at the beginning of each function that explains what it does, what parameters it takes, and what the expected output or return value is. For an example of good commenting look in the dll.h file. The comments are written for an interpreter called doxygen (http://en.wikipedia.org/wiki/Doxygen) that will parse the comments and output a set of html files. It is very similar to javadoc, a process that produces standardized documentation for Java API's. Also good style (no unused variables, meaningful names) and submitting the self-log file is important to get 10 style points.

Handing in your Solution
Your solution should be in the form of a .zip file. Just hand in all the .c and .h files in your program. Don't hand in input, obj or exe files.
Hand in your program to: **/afs/andrew.cmu.edu/course/15/123/handin/lab5/yourid**