ECE26400 Programming Exercise #8

This exercise will familiarize you with linked lists, which would be useful for PE09 and PA05. The main learning goals are:

1. How to create and manipulate linked lists.
2. How to free the memory associated with a linked list.

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pe08_files.zip` using the following command:

```
unzip pe08_files.zip
```

The zip file `pe08_files.zip` contains a folder, named `PE08`, and two files within the folder:

1. `answer08.h`: This is a "header" file and it declares the functions you will be writing for this exercise.
2. `answer08.c`: One of the functions in `answer08.h` is provided. You have to define all other functions declared in `answer08.h`.

In this exercise, you have to define in `answer08.c` all functions, except for one, that have been declared in `answer08.h`. While you have to create a file called `pe08.c` for the `main` function, you do not have to submit `pe08.c`.

2 Overview

This exercise will allow you to practice on the construction, manipulation, and destruction of linked lists. It will also prepare you for PE09 and PA05.

In this exercise, you will use a linked list to implement a priority queue or a stack.

A priority queue is implemented as a linked list where the objects stored in the list are actually ordered. For example, if we want to store a linked list of numbers 5, 10, 4, 3, 1, and the linked list is used to implement a priority queue, the linked list should appear as follows:

```
1->3->4->5->10->NULL
```

Here, we assume that "`->`" is the "pointer." We also assume that a smaller number has a higher priority, and it is therefore listed first. If we now want to get number 7 to join the priority queue, we should have the list updated to:

```
1->3->4->5->7->10->NULL
```

We call this the enqueue operation. To remove an item from a priority queue, we always remove the item with the highest priority. This is call the dequeue operation. In this case, a dequeue operation will remove 1 from the list:

```
3->4->5->7->10->NULL
```

When a stack is implemented as a linked list, the object that is pushed onto the stack always appears as the first item on the list. Therefore, if we push the list of numbers 5, 10, 4, 3, 1 onto a stack successively, we have the following linked list:

```
1->3->4->10->5->NULL
```

If we pop from the stack, we have

```
3->4->10->5->NULL
```

If we now push 7 onto the stack, we have

```
7->3->4->10->5->NULL
```

To help you understand the complexity involved in the removal of an item not at the beginning of a list, you will also write a function to remove the last item from a list. For the following list:

```
7->3->4->10->5->NULL
```

The removal of the last item in the list gives you

```
7->3->4->10->NULL
```

You will also write a function to destroy a linked list and free all memory associated with that linked list.

2.1 Structure

The following structure will be used for PE08.

```
typedef struct _lnode {
   void *ptr;
   struct _lnode *next;
} lnode;
```

The `ptr` field stores an address that points to a generic type. The generic type can be a simple data type, a multi-dimensional array, or even a user-defined structure. Essentially, you will be implementing a linked lists of addresses stored in `ptr`. We shall assume that within a linked list, these addresses in the `ptr` field will all be pointing to objects of the same type.

The `next` stores an address that points to an object of type `lnode` (or `struct _lnode`), thereby allowing you to maintain a linked list.

2.2 Functions to be implemented

You will implement the following functions in `answer08.c`.

```
lnode *PQ_enqueue(lnode **pq, const void *new_object,
                  int (*cmp_fn)(const void *, const void *));
```

`*pq` stores the address of the first `lnode` in the linked list. If `*pq` is `NULL`, the list is currently empty. You may assume that if `*pq` is not `NULL`, the list is valid. `new_object` stores the address of the generic type. `cmp_fn` stores the address of the comparison function for comparing the two objects of the same type. The addresses of the two objects, treated as addresses pointing to generic type, are passed into the function.

The `cmp_fn` function is expected to return a negative value (i.e., < 0) when the first object is smaller than the second one, 0 when the two objects are equal, and a positive value (i.e., > 0) when the first object is larger than the second one.

If `new_object` is `NULL`, the list should remain intact, and you should return `NULL`. (Note that this is what we choose to do for this programming exercise. In some applications, you may want to allow `NULL` as `new_object`. For example, if you want to have a linked list of linked lists. Some of these (sub)linked lists may be empty.)

If you could not allocate a new `lnode` to store `new_object` in the `ptr` field, the list should remain intact, and you should return `NULL`. Otherwise, the newly allocated `lnode` should be inserted into the linked list such that all `lnode`'s before the new `lnode` in the list contain objects that are smaller to the new `lnode` (according to `cmp_fn`), and all `lnode`'s after the new `lnode` in the list contain objects that are bigger (or equal) to the new `lnode` (according to `cmp_fn`).

`*pq` should be updated if the new `lnode` becomes the first item in the list. The function should return the address of the new `lnode`.

```
void *PQ_dequeue(lnode **pq);
```

`*pq` stores the address of the first `lnode` in the linked list. If the list is empty, there is nothing to dequeue. You should return `NULL`. Otherwise, you should remove the first `lnode` from the linked list and update `*pq` accordingly.

The function should return the address of the object stored in the `ptr` field of the removed `lnode`. The function is also responsible for freeing the memory associated with the removed `lnode`.

Note that the caller function is responsible for handling the memory associated with the object whose address is returned.

```
lnode *stack_push(lnode **stack, const void *new_object);
```

`*stack` stores the address of the first `lnode` in the linked list. `new_object` stores the address of the generic type.

If `new_object` is `NULL`, the list should remain intact, and you should return `NULL`. If you could not allocate a new `lnode` to store `new_object` in the `ptr` field, the list should remain intact, and you should return `NULL`. The `ptr` field of the newly allocated `lnode` should store `new_object`. The newly allocated `lnode` should become the first `lnode` of the list, and `*stack` should be updated. The function returns the address of the new `lnode`.

```
void *stack_pop(lnode **stack);
```

In a sense, this is the same as PQ_dequeue.

```
void *lnode_remove_last(lnode **list);
```

*list stores the address of the first lnode in the linked list. If the list is empty, there is nothing to remove. You should return NULL. Otherwise, you should remove the last lnode from the linked list and if necessary, update *list accordingly.

The function should return the address of the object stored in the ptr field of the removed lnode. The function is also responsible for freeing the memory associated with the removed lnode.

Note that the caller function is responsible for handling the memory associated with the object whose address is returned.

```
void lnode_destroy(lnode *list, void (*destroy_fn)(void *));
```

list stores the address of the first lnode in the list. destroy_fn stores the address of the function to deallocate the memory associated with the address of the generic type stored in the ptr field of an lnode.

If the generic type is a simple data type and the memory for such objects is obtained through malloc, the free function in stdlib.h is an appropriate function for destroy_fn. If the generic type is a user-defined type that involves a two-dimensional array, such as the structure defined in PE06, an appropriate function for destroy_fn is one that would somehow involves free_2d_array.

The lnode_destroy function should free all memory associated with list. For each lnode in the list, it has to use destroy_fn to deallocate memory associated with the address of the generic type stored in the ptr field and free to deallocate memory allocated for the lnode.

You may want to take a look at the lnode_print function provided in answer08.c as an example on how to call a function whose address has been passed in as a parameter.

2.3 Some examples of cmp_fn and print_fn

Here are some examples of cmp_fn (for the PQ_enqueue function) and print_fn (for the lnode_print function, which has been defined for you in answer08.c).

If we store the addresses of int (i.e., int *) as the addresses of the generic type (void *) in ptr, we can use the following comparison function:

```
static int int_cmp(const void *p1, const void *p2)
{
   return *(const int *)p1 - *(const int *)p2;
}
```

Here, the addresses in p1 and p2 have been typecast to const int *, and dereferenced to obtain const int for comparison (with the difference being the result).

If we store the addresses of char (i.e., char *) as the addresses of the generic type (void *) in ptr, and each address of char is actually the address of the first character of a string, we can use the following comparison function (under the assumption that you included string.h in the .c file that defines the string_cmp function):

```
static int string_cmp(const void *p1, const void *p2)
{
   return strcmp((const char *)p1, (const char *)p2);
}
```

We can use the following print functions for the two generic types for the `print_fn` parameter in the `lnode_print` function.

```
static void int_print(const void *ptr)
{
   printf("%d", *(const int *)ptr);
}

static void string_print(const void *ptr)
{
   printf("%s", (const char *)ptr);
}
```

These functions will be useful for your test functions (to be written in `pe08.c`, for example). These comparison functions and print functions should not appear in `answer08.c`.

2.4 Printing, helper functions and macros

All debugging, error, or log statements in `answer08.c` should be printed to `stderr`.

You may define your own helper functions in `answer08.c`. All these functions should be declared and defined as static functions. In other words, the scope of these functions are only within the files that they are found. Declaring and defining these functions as static eliminate the conflicts in function names.

You should not use macros that start with `T_` (Upper case T and underscore) in their names. We will use such macros in the evaluation of your submissions. If you use such macros, we may not be able to evaluate your submission properly.

3 Compiling your program

As you have advanced to this stage, we believe that you have the necessary experience to decide what warnings are allowed and what warnings should be eliminated. We will now remove the `-Werror` flag from the `gcc` command. Here, we assume that the `main` function is defined in `pe08.c`.

```
gcc -std=c99 -Wall -Wshadow -Wvla -g -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe08.c answer08.c -o pe08
```

When we evaluate your program, we also use the optimization flag `-O3` (uppercase letter `O` and number three) instead of the `-g` flag as follows:

```
gcc -std=c99 -Wall -Wshadow -Wvla -O3 -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe08.c answer08.c -o pe08
```

You are recommended to copy the `Makefile` from PE01 and modify it appropriately for PE08.

4 Writing, running and testing your program

We provided `ln_print` function for you in `answer08.c`. You could use the provided function for your testing. You have to develop your own `main` function to test these functions in `pe08.c`. Please note that you do not submit `pe08.c`.

Please see PE01 description about `valgrind`.

5 Submission

You must submit a zip file called `PE08.zip`, which contains one file:

1. `answer08.c`

Assuming that you are in the folder that contains `answer08.c`, use the following command to zip your files:

```
zip PE08.zip answer08.c
```

Make sure that you name the zip file as `PE08.zip`. Moreover, the zip file should not contain any folders. Submit `PE08.zip` through Brightspace. You may submit as many versions as you like. Brightspace will keep only the most recent submission, and we will grade only the final submission.

6 Grading

All debugging messages should be printed to `stderr`. It is important that if the instructor has a working version of `pe08.c`, it should be compilable with your `answer08.c` to produce an executable. The instructor will have various versions of `pe08.c` for different data types, including user-defined data types provided by the instructor. If a particular combination of `pe08.c` from the instructor and your `answer08.c` does not allow an executable to be generated, you do not get any credit for the function(s) that the executable is supposed to evaluate. We use both `-g` and `-O3` flags (separately) to compile your submission. Your submission is evaluated only when compilation using each of the flags is successful.

It is important to note that the `.c` files from the instructor do not assume the presence of global variables that are declared by the students. Of course, the `.c` files from the instructor may use global variables that are in C libraries, such as `errno`, `stdout`, `stderr`, and so on. If your submission contains global variables that are declared by you, it is unlikely that your executable will work correctly.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case. We will not announce the time-limits that we will use. They will be very reasonable.

The `PQ_enqueue` function accounts for 25%. The `PQ_dequeue` function accounts for 10%. The `stack_push` function accounts for 15%. The `stack_pop` function accounts for 5%. The `lnode_remove_last` function accounts for 25%. The `lnode_destroy` function accounts for 20%.

The occurrence of any memory issues (memory errors or memory leaks flagged in a `valgrind` report) will result in 50-point penalty.

7 A few points to remember

All debugging messages should be printed to `stderr`. If the program creates an output file when it should not, or the output file is correct, you get 0 for that test case.

You can declare and define additional static functions that you have to use in `answer08.c`.

You will have to write a `main` function so that you can test these six functions defined in `answer08.c`. However, you should write the `main` function in a separate file called `pe08.c`. If you define the `main` function in `answer08.c`, we will not be able to compile your `answer08.c`, and you will get zero for this exercise.

You should not use macros that start with `T_`.

Grading of programming exercises and assignments is performed on machines with similar setup as eceprog.ecn.purdue.edu. You should perform testing of your work on eceprog.ecn.purdue.edu before submission. Correct output on your computer does not translate into correct output on eceprog.ecn.purdue.edu.