Programming Assignment : Lonely Party Array

Computer Science at UCF

In this programming assignment, you will implement lonely party arrays (arrays that are broken into fragments that get allocated and deallocated on an as-needed basis, rather than being allocated all at once as one big array, in order to eliminate unnecessary memory bloat). This is an immensely powerful and awesome data structure, and it will ameliorate several problems we often encounter with arrays in C (see Section 1 of this PDF).

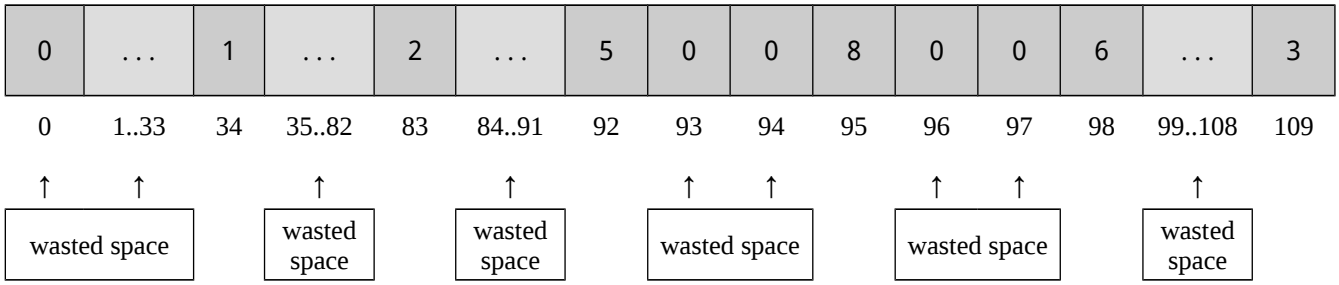
By completing this assignment, you will gain advanced experience working with dynamic memory management, pointers, and structs in C. You will also learn how to use *valgrind* to test your programs for memory leaks, and you will gain additional experience managing programs that use custom header files and multiple source files. In the end, you will have an awesome and useful data structure that you can use to solve all sorts of interesting problems.

1. Overview

For a lot of programming tasks that use arrays, it's not uncommon to allocate an array that is large enough to handle any worst-case scenario you might throw at your program, but which has a lot of unused, wasted memory in most cases.

For example, suppose we're writing a program that needs to store the frequency distribution of scores on some exam. If we know the minimum possible exam score is zero and the maximum possible exam score is 109 (because there are a few bonus questions), and all possible scores are integers, then we might create an array of length 110 (with indices 0 through 109) to meet our needs.

Suppose, then, that we're storing data for 25 students who took that exam. If 3 of them earned 109%, 6 students earned 98%, 8 students earned 95%, 5 students earned 92%, 2 students earned 83%, and 1 student earned a 34%, the frequency array for storing their scores would look like this:



Notice that with only 6 distinct scores in the class, we only use 6 cells to record the frequencies of scores earned for all 25 students. The other 104 cells in the array are just wasted space.

1.1 Lonely Party Arrays to the Rescue!

In this assignment, you will implement a new array-like data structure, called a lonely party array (or "LPA"),[1] that will solve the problem described above. In this data structure, instead of allocating one large array, we will allocate smaller array *fragments* an as-needed basis.

For example, in the application described above, we could split our array into 11 fragments, each of length 10. The first fragment would be used to store the frequencies of scores 0 through 9, the second fragment would store data for scores 10 through 19, and so on, up until the eleventh fragment, which would store data for scores 100 through 109.

The twist here is that we will only create array fragments on an as-needed basis. So, in the example above, the only fragments we would allocate would be the fourth (for scores 30 through 39), ninth (for scores 80 through 89), tenth (for scores 90 through 99), and eleventh (for scores 100 through 109). Each fragment would use 40 bytes (since each one has 10 integers, and an integer in C is typically 4 bytes), meaning we'd be using a total of 160 bytes for those 4 fragments. Compare this to the 4 * 110 = 440 bytes occupied by the original array of length 110, and you can see how this new data structure allows us to save memory. In this case, the LPA would reduce our memory footprint by over 63%.[2]

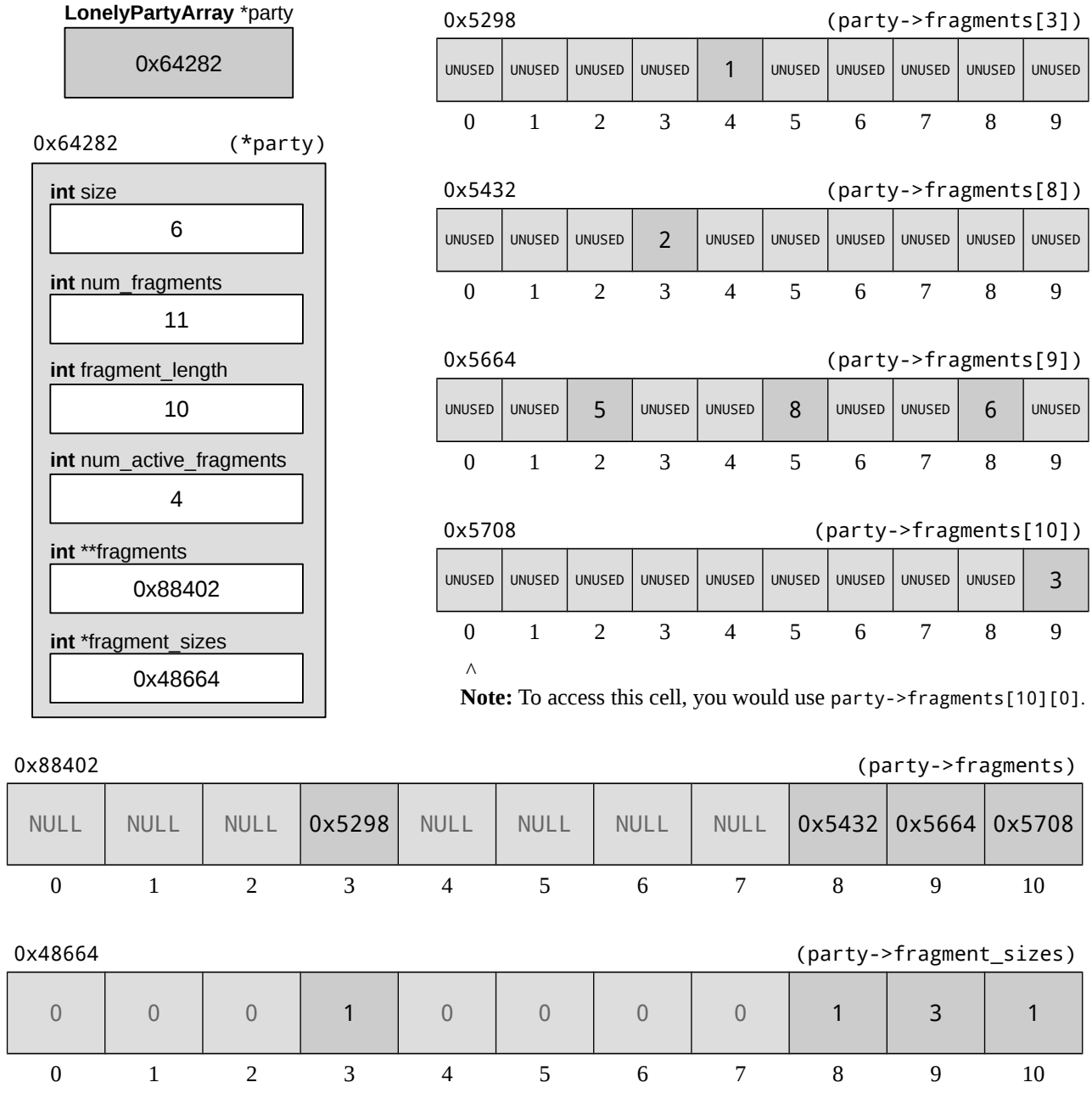
1 "It's called a lonely party array because sometimes you invite 3000 people to a party, but only 3 show up." – CS1 TA

2 We'll see later that we also have to store quite a few pointers in our lonely party arrays, so although we will still save memory, the savings won't end up being quite so substantial in this particular example once we've fleshed it out.

There are a few other twists with this *LonelyPartyArray* data structure. If we have a bunch of small arrays (called “fragments” in this assignment), we need to keep track of their addresses. So, we’ll create an array of integer pointers to store all the base addresses of those arrays. If a fragment hasn’t been allocated, we’ll just store a NULL pointer in place of the address for that fragment.

We’ll also store how many cells in each fragment are occupied. In the example above, fragment 3 only has one occupied cell (since 34% is the only score that anyone earned in the range 30 through 39). We’ll keep track of that so that if we delete values from the LPA and get to a point where there are zero occupied cells in a particular fragment, we can free all the memory associated with that fragment.

The following diagram shows a complete *LonelyPartyArray* struct, with all its constituent members, for the example described above. There are 11 fragments possible, each of length 10, with 4 currently active. Additional details about the members of this struct are included in the pages that follow.



As you can see from the diagram above, there are a few other things to keep track of, but this section gives the basic idea behind the lonely party array. All the juicy details about everything else you need to keep track of are given below in Section 3, “Function Requirements.”

1.2 Advantages of Lonely Party Arrays

As with normal arrays in C, we will have fast, direct access to any index of a lonely party array at any given time. This data structure has three main advantages over C’s traditional arrays:

1. As we saw above, normal arrays often have wasted space. We will only allocate fragments of our lonely party arrays on an as-needed basis, and deallocate them when they’re no longer in use, which will reduce the amount of unused memory being wasted.

2. We will use *get()*, *set()*, and *delete()* functions to access and modify individual elements of the lonely party array, and these functions will help us avoid segfaults by first checking that we aren’t accessing array indices that are out of bounds. (Recall that C doesn’t check whether an array index is out of bounds before accessing it during program execution. That can lead to all kinds of wacky trouble!)

3. In C, if we have to pass an array to a function, we also typically find ourselves passing its length to that function as a second parameter. With lonely party arrays, all the information you need about the data structure will get passed automatically with the array fragments themselves, as everything will be packaged together in a struct.

1.3 Overview of What You’ll Submit

The lonely party arrays you implement for this assignment will be designed to hold integers. The precise number of fragments – and the lengths of those fragments – will be allowed to vary from one LPA to the next. A complete list of the functions you must implement, including their functional prototypes, is given below in Section 3, “Function Requirements.”

You will submit a single source file, named *LonelyPartyArray.c*, that contains all required function definitions, as well as any auxiliary functions you deem necessary. In *LonelyPartyArray.c*, you should #include any header files necessary for your functions to work, including *LonelyPartyArray.h* (see Section 2, “LonelyPartyArray.h”).

Note that you will <u>not</u> write a main() function in the source file you submit! Rather, we will compile your source file with our own *main()* function(s) in order to test your code. We have attached example source files that have `main()` functions, which you can use to test your code. You should also write your own *main()* functions for testing purposes, but your code must not have a *main()* function when you submit it. We realize this is still fairly new territory for most of you, so don’t panic. We’ve included instructions on compiling multiple source files into a single executable (e.g., mixing your *LonelyPartyArray.c* with our *LonelyPartyArray.h* and *testcase01.c* files) in Section 5 (pg. 16).

Although we have included sample `main()` functions to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission.

Start early. Work hard. Good luck!

2. LonelyPartyArray.h

This header file contains the struct definition and functional prototypes for the lonely party array functions you will be implementing. You **must** #include this file from *LonelyPartyArray.c*, as follows. Recall that the "quotes" (as opposed to <brackets>) indicate to the compiler that this header file is found in the same directory as your source, not a system directory:

```
#include "LonelyPartyArray.h"
```

Please do not modify *LonelyPartyArray.h* in any way, and do not send *LonelyPartyArray.h* when you submit your assignment. We will use our own copy of *LonelyPartyArray.h* when compiling your program.

If you write auxiliary functions ("helper functions") in *LonelyPartyArray.c* (which is strongly encouraged!), you should **not** add those functional prototypes to *LonelyPartyArray.h*. Our test case programs will not call your helper functions directly, so they do not need any awareness of the fact that your helper functions even exist. (We only list functional prototypes in a header file if we want multiple source files to be able to call those functions.) So, just put the functional prototypes for any helper functions you write at the top of your *LonelyPartyArray.c* file.

Think of *LonelyPartyArray.h* as a bridge between source files. It contains struct definitions and functional prototypes for functions that might be defined in one source file (such as your *LonelyPartyArray.c* file) and called from a different source file (such as *testcase01.c*).

The basic struct you will use for this data structure (defined in the header file) is as follows:

```
typedef struct LonelyPartyArray
{
   int size;                  // number of occupied cells across all fragments
   int num_fragments;         // number of fragments (arrays) in this struct
   int fragment_length;       // number of cells per fragment
   int num_active_fragments;  // number of allocated (non-NULL) fragments
   int **fragments;           // array of pointers to individual fragments
   int *fragment_sizes;       // stores number of used cells in each fragment
} LonelyPartyArray;
```

The *LonelyPartyArray* struct contains an *int*** pointer that can be used to set up a 2D *int* array (which is just an array of *int* arrays). That *fragments* array will have to be allocated dynamically whenever you create a new LPA struct. The *size* member of this struct tells us how many elements currently reside in the lonely party array (i.e., how many *int* cells are actually being used across all the fragments and are not just wasted space). The *fragment_length* member tells us how many integer cells there should be in each individual fragment that we allocate. (All the non-NULL fragments within a given LPA struct will always be the same length.) *num_active_fragments* tells us how many non-NULL fragments this lonely party array is using. The *fragment_sizes* array is used to keep track of how many cells are actually being used in each fragment. This count allows us to determine very quickly whether we can deallocate a fragment any time we delete one of the elements it contains.

This header file also contains definitions for `UNUSED`, `LPA_SUCCESS` and `LPA_FAILURE`, which you will use in some of the required functions described below.

5. Running the Provided Test Cases Individually

If the *test-all.sh* script tells you that one of your test cases is failing, you'll want to compile and run that test case individually to examine its output. Here's how to do that:

To compile your source file with one of our test cases (such as *testcase01.c*) at the command line:

```
gcc LonelyPartyArray.c testcase01.c
```

By default, this will produce an executable file called *a.out*, which you can run by typing:

```
./a.out
```

If you want to name the executable file something else, use:

```
gcc LonelyPartyArray.c testcase01.c -o LPA.exe
```

...and then run the program using:

```
./LPA.exe
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called *whatever.txt* that contains the output from your program:

```
./LPA.exe > whatever.txt
```

Linux has a helpful command called *diff* for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample_output/output01.txt
```

If the files differ, *diff* will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt output01.txt
1c1
< fail whale :(
---
> Hooray!
seansz@eustis:~$ _
```

If the contents of *whatever.txt* and *output01.txt* are exactly the same, *diff* won't have any output:

```
seansz@eustis:~$ diff whatever.txt output01.txt
seansz@eustis:~$ _
```