Programming Assignment #2: Lonely Party Array

COP 3502, Spring 2018

Due: Sunday, February 11, *before* 11:59 PM

Table of Contents

Abstract

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 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.

Attachments

LonelyPartyArray.h

testcase{01-16}.c and *output{01-16}.txt*

SanityCheck.c and *SanityCheckOutput.txt*

CloneTest.c and *CloneTestOutput.txt*

test-all.sh

Deliverables

LonelyPartyArray.c

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

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:

0	. . .	1	. . .	2	. . .	5	0	0	8	0	0	6	. . .	3
0	1..33	34	35..82	83	84..91	92	93	94	95	96	97	98	99..108	109
wasted space	wasted space		wasted space		wasted space		wasted space	wasted space		wasted space	wasted space		wasted space	

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.

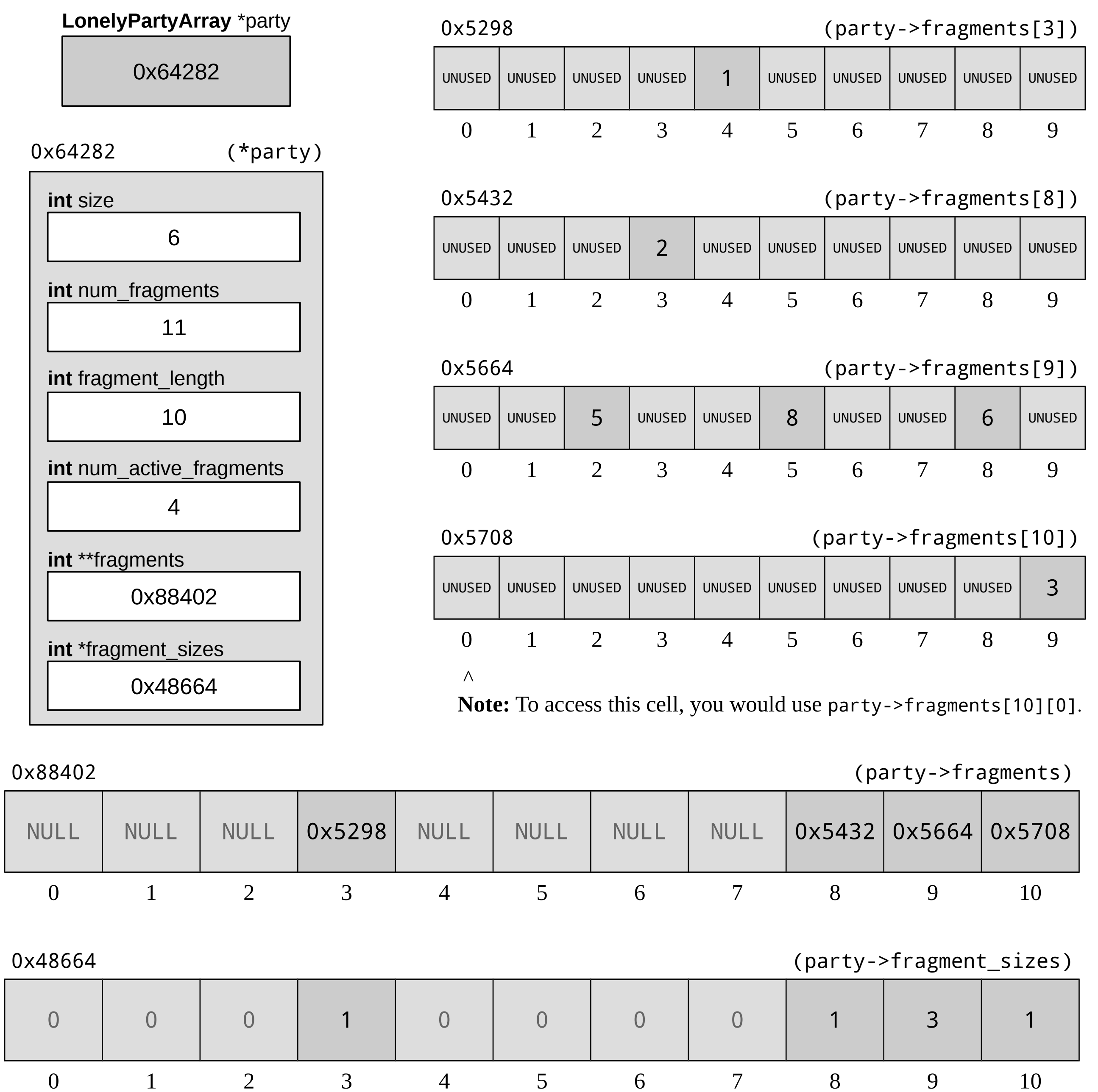
The twist here is that we will only create array fragments on an as-needed basis. So, in the example above, the only fragments we would allocate would be the fourth (for scores 30 through 39), ninth (for scores 80 through 89), tenth (for scores 90 through 99), and eleventh (for scores 100 through 109). 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 4, "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 4, "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 Sections 7 and 8.

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 unmodified 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.

3. New Data Type: long long unsigned int

One final note before we get to the functions you'll be writing: In some of the following functions, we will be dealing with numbers so large that a traditional `int` simply can't handle them.

An `int` in C is typically 32 bits (4 bytes). Although that might not be the case on all systems, it almost certainly is the case for whatever system you're using to write your programs this semester. With 32 bits (which can each take on the value of 0 or 1), we can represent 2^{32} = 4,294,967,296 different integers. About half of those integers are positive, and about half of them are negative; the range of a 32-bit `int` in C is -2,147,483,648 through 2,147,483,647.

C offers another integer data type called a `long long unsigned int`. This data type uses at least 64 bits (8 bytes), and does not allow for negative integers, so it can handle integers on the range 0 through 2^{64} - 1 = 18,446,744,073,709,551,615. Yes, that number is huge.

Here's an example of how to declare a `long long unsigned int` and how to print out its value with the appropriate conversion code ("`%llu`"):

```
long long unsigned int x = 4000000000;  // 4 billion (too big for an int)
printf("%llu\n", x);
```

If you want to do math with two large `int` values and you think the result might exceed 2,147,483,647, you will need to explicitly transform one of those `int` values into a `long long unsigned int` by performing a "type cast." That's where you take an existing variable, and in front of it, you put a different data type in parentheses. For example, in the following, (`x * y`) is way too big to be an `int`:

```
int x = 2000000000, y = 2000000000;
long long unsigned int z = (long long unsigned int)x * y;  // type cast x
printf("%llu\n", (long long unsigned int)x * y);           // type cast x
printf("%llu\n", z);
```

You'll need to be careful with your type casts. When performing an arithmetic operation on a regular `int` and a variable that has been cast as a `long long unsigned int`, you get a `long long unsigned int` result. However, the following arithmetic expression is problematic because although the first (`x * y`) yields a `long long unsigned int`, order of operations then performs the second (`x * y`), and since neither of those variables are explicitly type cast in the second multiplication operation, C treats them as regular `int` values, and everything falls apart because a regular `int` can't hold that huge result.

```
int x = 2000000000, y = 2000000000;  // x * y is way too big for an int

printf("%llu\n", (long long unsigned int)x * y + x * y);  // BAD

printf("%llu\n", (long long unsigned int)x * y
                + (long long unsigned int)x * y);  // OKAY
```

4. Function Requirements

In the source file you submit, `LonelyPartyArray.c`, you must implement the following functions. You may implement any auxiliary functions you need to make these work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly. In this section, I often refer to `malloc()`, but you're welcome to use `calloc()` or `realloc()` instead, if you're familiar with those functions.

```
LonelyPartyArray *createLonelyPartyArray(int num_fragments, int fragment_length);
```

Description: This function will create a `LonelyPartyArray` (LPA) that can accommodate up to `num_fragments` different array fragments, each of length `fragment_length`. So, the maximum number of integers this LPA can hold is `num_fragments * fragment_length`.

Start by dynamically allocating space for a new `LonelyPartyArray` struct. Initialize the struct's `num_fragments` and `fragment_length` members using the arguments passed to this function. Initialize `num_active_fragments` and `size` to the appropriate values. Dynamically allocate the `fragments` array to be an array of `num_fragments` integer pointers, and initialize all those pointers to `NULL`. (Eventually, that array will store pointers to any individual array fragments you allocate when adding elements to this data structure.) Dynamically allocate the `fragment_sizes` array, and initialize all values to zero (since none of the fragments currently hold any elements).

If one or both of the parameters passed to this function are less than or equal to zero (i.e., if they are not both positive integers), you should immediately return `NULL`. If any of your calls to `malloc()` fail, you should free any memory you have dynamically allocated in this function call up until that point (to avoid memory leaks), and then return `NULL`.

Output: If the function successfully creates a new LPA, print the following: "`-> A new LonelyPartyArray has emerged from the void. (capacity: <M>, fragments: <N>)`"

Do not print the quotes, and do not print <brackets> around the variables. Terminate the line with a newline character, '`\n`'. `<M>` is the maximum number of integers this LPA can contain (`num_fragments * fragment_length`). `<N>` is simply `num_fragments`. Note that in testing, we will never create a lonely party array whose capacity would be too large to store in a 32-bit `int`.

Returns: A pointer to the new LonelyPartyArray, or `NULL` if any calls to `malloc()` failed.

```
LonelyPartyArray *destroyLonelyPartyArray(LonelyPartyArray *party);
```

Description: Free all dynamically allocated memory associated with this LonelyPartyArray struct, and return `NULL`. Be careful to avoid segfaulting in the event that `party` is NULL.

Output: "`-> The LonelyPartyArray has returned to the void.`" (Output should not include the quotes. Terminate the line with a newline character, '`\n`'.) If `party` is NULL, this function should not print anything to the screen.

Returns: This function should always return `NULL`.

```
int set(LonelyPartyArray *party, int index, int key);
```

Description: Insert key into the appropriate index of the LonelyPartyArray. Based on the index parameter passed to this function, as well as the number of fragments in the LPA and the length of each fragment, you will have to determine which fragment index maps to, and precisely which cell in that array fragment is being sought. For example, if index is 14 and the LPA has num_fragments = 3 and fragment_length = 12, then key would go into the second array fragment (fragments[1]) at index 2 (fragments[1][2]), since the first fragment (fragments[0]) would be used for indices 0 through 11, the second fragment would be used for indices 12 through 23, and the third fragment would be used for indices 24 through 35, for a total of 36 cells across the 3 fragments. For additional indexing examples, please refer to the test cases included with this assignment.

If the fragment where we need to insert key is NULL, then you should dynamically allocate space for that fragment (an array of fragment_length integers), initialize each cell in that new fragment to UNUSED (which is #defined in LonelyPartyArray.h), update the struct's num_active_fragments member, and store key at the appropriate index in the new allocated array.

If the index being modified was previously empty (i.e., UNUSED), and/or if the fragment was not yet allocated, then after inserting key into the lonely party array, be sure to increment the struct's size member so that it always has an accurate count of the number of cells in the LPA that are currently occupied, and be sure to increment the appropriate value in the struct's fragment_size array so that it always has an accurate count of the number of cells currently being used in each particular fragment.

If index is invalid (see note below), or if a NULL party pointer is passed to this function, simply return LPA_FAILURE without attempting to modify the data structure in any way and without causing any segmentation faults.

Note on "invalid index" values: In our set(), get(), and delete() functions, index is considered valid if it falls in the range 0 through (num_fragments * fragment_length - 1), even if index refers to a cell marked as UNUSED or a fragment that has not been allocated yet. An "invalid index" is one that falls outside that specified range.

Output: There are three cases that should generate output for this function. (Do not print the quotes, and do not print <brackets> around the variables. Terminate any output with '\n'.)

- If calling this function results in the allocation of a new fragment in memory, print: "-> Spawned fragment <P>. (capacity: <Q>, indices: <R>..<S>)" <P> is the index where the newly allocated fragment's address is stored in the struct's fragments array, and <Q> is the length of the new array (i.e., the number of integers it can hold), and <R> and <S> correspond to the lowest and highest index values that would map to this particular fragment.

- If party is non-NULL and index is invalid (see definition of "invalid index" above), print: "-> Bloop! Invalid access in set(). (index: <L>, fragment: <M>, offset: <N>)" <L> is value of index passed to this function; <M> is the invalid fragment array index that index would have mapped to, had that fragment been within bounds for this lonely party array; and `<N>` is the precise index within that fragment that `index` would have mapped to. For example, if `num_fragments = 2`, `fragment_length = 10`, and `index = 21`, then the valid indices for this LPA would be 0 through 19, and `index`, being invalid, would produce this error message with `<M> = 2` and `<N> = 1`. See `testcase16.c` and `output16.txt` for examples of how to handle output for negative `index` values. For additional examples, please refer to the test cases included with this assignment.

- If `party` is NULL, print: "`-> Bloop! NULL pointer detected in set().`" Output should not contain any quotes or angled brackets. Terminate the line with a newline character, '`\n`'.

Returns: If this operation is successful, return `LPA_SUCCESS`. If the operation is unsuccessful (either because `index` is invalid or the function receives a NULL `party` pointer, or because any calls to `malloc()` fail), return `LPA_FAILURE`. `LPA_SUCCESS` and `LPA_FAILURE` are #defined in `LonelyPartyArray.h`.

```
int get(LonelyPartyArray *party, int index);
```

Description: Retrieve the value stored at the corresponding index of the LonelyPartyArray. As with the `set()` function, based on the `index` parameter passed to this function, as well as the number of fragments in the LPA and the length of each fragment, you will have to determine which fragment `index` maps to, and precisely which cell in that array fragment is being sought.

Keep in mind that `index` could try taking you to a fragment that has not yet been allocated. It's up to you to avoid going out of bounds in an array and/or causing any segmentation faults.

Output: There are two cases that should generate output for this function. (Do not print the quotes, and do not print <brackets> around the variables. Terminate any output with '`\n`'.)

- If `index` is invalid (see definition of "invalid index" in the `set()` function description) and `party` is non-NULL, print: "`-> Bloop! Invalid access in get(). (index: <L>, fragment: <M>, offset: <N>)`" For explanations of `<L>`, `<M>`, and `<N>`, see the `set()` function description above. Note that if a cell is marked as `UNUSED`, but `index` is within range, you should not generate this error message.

- If `party` is NULL, print: "`-> Bloop! NULL pointer detected in get().`"

Returns: If `index` is valid (see definition of "invalid index" in the `set()` function description), return the value stored at the appropriate index in the lonely party array (even if it is marked as `UNUSED`). If the operation is unsuccessful (either because `index` is invalid or because the function receives a NULL `party` pointer), return `LPA_FAILURE`.

```
int delete(LonelyPartyArray *party, int index);
```

Description: Set the value stored at the corresponding index of the lonely party array to `UNUSED`. As with the `set()` and `get()` functions, based on the `index` parameter passed to this function, as well as the number of fragments in the LPA and the length of each fragment, you will have to determine which fragment `index` maps to, and precisely which cell in that array fragment is being sought.

If the cell being sought is *not* already set to `UNUSED`, then after writing `UNUSED` to that cell, decrement the struct's `size` member so that it always has an accurate count of the total number of cells that are currently being used, and decrement the appropriate value in the struct's `fragment_size` array so that it always has an accurate count of the number of cells currently being used in each fragment.

If deleting the value at this index causes the fragment containing that value to become empty (i.e., all of its cells are now `UNUSED`), deallocate that array, set the appropriate pointer in the struct's `fragments` array to `NULL`, and update the struct's `num_active_fragments` member. You should never loop through a fragment to see if all of its cells are unused. Instead, you should rely on the `fragment_sizes` array to keep track of whether or not a fragment is ready for deallocation.

Keep in mind that `index` could try taking you to a fragment that has not yet been allocated. It's up to you to avoid going out of bounds in an array and/or causing any segmentation faults.

Output: There are three cases that should generate output for this function. (Do not print the quotes, and do not print <brackets> around the variables. Terminate any output with '`\n`'.)

- If calling this function results in the deallocation of a fragment in memory, print: "-> Deallocated fragment <P>. (capacity: <Q>, indices: <R>..<S>)" For explanations of <P>, <Q>, <R>, and <S>, see the `set()` function description above.

- If `index` is invalid (see definition of "invalid index" in the `set()` function description) and `party` is non-NULL, print: "-> Bloop! Invalid access in delete(). (index: <L>, fragment: <M>, offset: <N>)" For explanations of <L>, <M>, and <N>, see the `set()` function description above. Note that if a cell is marked as `UNUSED`, but `index` is within range, you should not generate this error message.

- If `party` is NULL, print: "-> Bloop! NULL pointer detected in delete()."

Returns: Return `LPA_FAILURE` if this operation refers to an invalid index (as defined in the `set()` function description), if this function receives a NULL `party` pointer, if this operation refers to a cell that lies within an unallocated fragment, or if this operation refers to a cell whose value was already set to `UNUSED` when the function was called. Otherwise, return `LPA_SUCCESS`.

```
int printIfValid(LonelyPartyArray *party, int index);
```

Description: Print the value stored at the corresponding index of the LonelyPartyArray. As with the `set()`, `get()`, and `delete()` functions, based on the `index` parameter passed to this function, as well as the number of fragments in the LPA and the length of each fragment, you will have to determine which fragment `index` maps to, and precisely which cell in that array fragment is being sought.

Output: Simply print the appropriate integer to the screen, followed by a newline character, '`\n`'. This function should not print anything if `index` is invalid (as defined in the `set()` function description), if `index` refers to a cell whose value is set to `UNUSED`, or if `party` is NULL.

Returns: Return `LPA_SUCCESS` if this function prints a value to the screen. Otherwise, return `LPA_FAILURE`.

```
LonelyPartyArray *resetLonelyPartyArray(LonelyPartyArray *party);
```

Description: Reset the lonely party array to the state it was in just after it was created with `createLonelyPartyArray()`. Be sure to avoid any memory leaks or segmentation faults.

You will need to deallocate any array fragments that are currently active within `party`. You will also need to reset all the values in the struct's `fragments` and `fragment_sizes` arrays. However, you should not re-allocate the `fragments` or `fragment_sizes` arrays; simply reset the values contained in those already-existing arrays.

You will also need to reset the struct's `size` and `num_active_fragments` members. You should not, however, change the values of `num_fragments` or `fragment_length`.

Output: There are two cases that should generate output for this function. (Do not print the quotes, and do not print <brackets> around the variables. Terminate any output with '`\n`'.)

- If `party` is non-NULL, print: "`-> The LonelyPartyArray has returned to its nascent state. (capacity: <M>, fragments: <N>)`" Here, `<M>` is the maximum number of integers the lonely party array can hold, and `<N>` is the value of the struct's `num_fragments` member.

- If `party` is NULL, be sure to avoid segmentation faults, and simply return from the function after printing the following: "`-> Bloop! NULL pointer detected in resetLonelyPartyArray().`"

Returns: This function should always return `party`.

```
int getSize(LonelyPartyArray *party);
```

Description: This function simply returns the number of elements currently in the LPA (**<u>not</u>** including any elements marked as `UNUSED`). This should be a near-instantaneous function call; it should not loop through the cells in the LPA at all.

We provide this function to discourage other programmers from ever accessing `party->size` directly if they try to compile code that uses our fancy LPA data structure. That way, if we release this data structure to the public but then end up updating it a few months later to rename the `size` member of the LonelyPartyArray struct to something else, the programmers who have been using our code and end up downloading the latest version can get it working right out of the box; they don't have to go through their own code and change all instances of `party->size` to something else, as long as we still provide them with a `getSize()` function that works as intended.[3]

Output: This function should not print anything to the screen.

Returns: Return the number of elements currently in the LPA, or `-1` if `party` is NULL.

```
int getCapacity(LonelyPartyArray *party);
```

Description: This function should simply return the maximum number of elements that `party` can hold, based on the values of its `num_fragments` and `fragment_length` members. For example, for the lonely party array shown on pg. 4 of this PDF, `getCapacity()` would return 110, because when all of its array fragments are allocated, it will be able to hold up to 110 integer elements.

Note that in testing, we will never create a lonely party array whose capacity would be too large to store in a 32-bit `int`, so you don't need to worry about type casting when calculating this return value.

Output: This function should not print anything to the screen.

Returns: Return the capacity of the LPA (as just described), or `-1` if `party` is `NULL`.

```
int getAllocatedCellCount(LonelyPartyArray *party);
```

Description: This function should return the maximum number of elements that `party` can hold without allocating any new array fragments. For example, for the lonely party array shown on pg. 4 of this PDF, `getAllocatedCellCount()` would return 40, because the non-NULL fragments are able to hold up to 40 integer elements in total.

Output: This function should not print anything to the screen.

Returns: Return the number of allocated integer cells (as just described), or `-1` if the `party` pointer is `NULL`.

```
long long unsigned int getArraySizeInBytes(LonelyPartyArray *party);
```

Description: This function should return the number of bytes that would be used if we were using a standard array rather than a LonelyPartyArray struct. For example, for the LPA struct shown on pg. 4 of this PDF, a traditional array representation would have 110 integer cells, which would occupy `110 * sizeof(int)` = 440 bytes, and so this function should return 440 for that struct. For additional examples, please refer to the test cases included with this assignment.

3 Note, by the way, that it is common to use the term "size" to refer to the number of elements that have been inserted into a data structure, while "length" often refers to the number of cells in an array, whether those cells have been used or not. (I.e., "length" refers to the maximum capacity of an array, in terms of the number of elements it can hold.)

Note: This number could get quite large, and so as you perform the arithmetic here, you should cast to `long long unsigned int`. (For details, see Section 3 on pg. 7.)

Note: You should use `sizeof()` in this function (rather than hard-coding the size of an integer as 4 bytes), and cast the `sizeof()` values to `long long unsigned int` as appropriate.

Note: If your system does not use 32-bit integers for some reason, using `sizeof()` in this function could cause output mismatches on some test cases. For this function to work, you will need to ensure that you're testing on a system that uses 32-bit integers (which you almost certainly are). To check that, you can compile and run `SanityCheck.c` (included with this assignment), or simply run the `test-all.sh` script.

Output: This function should not print anything to the screen.

Returns: The number of bytes (as just described), or 0 if the `party` pointer is `NULL`.

```
long long unsigned int getCurrentSizeInBytes(LonelyPartyArray *party);
```

Description: This function should return the number of bytes currently taken up in memory by the LPA. You will need to account for all of the following:

- The number of bytes taken up by the LPA pointer itself: `sizeof(LPA*)`
- The number of bytes taken up by the LPA struct (which is just the number of bytes taken up by the four integers and two pointers within the struct): `sizeof(LPA)`
- The number of bytes taken up by the `fragments` array (i.e., the number of bytes taken up by the pointers in that array).
- The number of bytes taken up by the `fragment_sizes` array (i.e., the number of bytes taken up by the integers in that array).
- The number of bytes taken up by the active fragments (i.e., the number of bytes taken up by all the integer cells in the individual array fragments).

The `getArraySizeInBytes()` and `getCurrentSizeInBytes()` functions will let you see, concretely, the amount of memory saved by using a lonely party array instead of a traditional C-style array.[4]

Note: This number could get quite large, and so as you perform the arithmetic here, you should cast to `long long unsigned int`. (For details, see Section 3 on pg. 7.)

Note: You should use `sizeof()` in this function (rather than hard-coding the sizes of any data types), and cast the `sizeof()` values to `long long unsigned int` as appropriate.

Output: This function should not print anything to the screen.

Returns: The number of bytes (as just described), or 0 if the `party` pointer is `NULL`.

4 Note that lonely party arrays are best used to replace large, sparsely populated arrays. Because of all the pointers in the LPA struct, a densely populated lonely party array could actually take up *more* space than a traditional C-style array.

```
double difficultyRating(void);
```

> **Description:** You do not need to call this function anywhere in your code. I will call this function myself in order to gather data on how difficult students found this programming assignment.
>
> **Output:** This function should not print anything to the screen.
>
> **Returns:** A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

> **Description:** You do not need to call this function anywhere in your code. I will call this function myself in order to gather data on how much time students estimate spending on this programming assignment.
>
> **Output:** This function should not print anything to the screen.
>
> **Returns:** A reasonable estimate (greater than zero) of the number of hours you spent on this assignment.

4.1 Bonus Functions

This function is optional.

```
LonelyPartyArray *cloneLonelyPartyArray(LonelyPartyArray *party);
```

> **Description:** Dynamically allocate a new `LonelyPartyArray` struct and set it up to be a clone of `party`. The clone should have entirely new, separate copies of all the data contained within `party`. (For example, the clone should not simply refer to `party`'s fragments. Instead, it should have entirely new copies of those fragments.)
>
> If any calls to `malloc()` fail, free any memory that this function dynamically allocated up until that point, and then return NULL.
>
> **Output:** If `party` is non-NULL, print the following: "`-> Successfully cloned the LonelyPartyArray. (capacity: <M>, fragments: <N>)`" This output should not include the quotes or angled brackets. Terminate the line with a newline character, '`\n`'. `<M>` is the maximum number of integers this LPA can contain (`num_fragments` * `fragment_length`). `<N>` is simply `num_fragments`. Note that in testing, we will never create a lonely party array whose capacity would be too large to store in a 32-bit `int`. If `party` is NULL, or if any calls to `malloc()` fail, this function should not print anything to the screen.
>
> **Returns:** If `party` is NULL, or if any calls to `malloc()` fail, simply return NULL. Otherwise, return a pointer to the newly allocated lonely party array.

5. Test Cases and the test-all.sh Script

The multiple test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, `test-all.sh`, that will compile and run all test cases for you.

Super Important: Using the `test-all.sh` script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

You can run the script on Eustis by placing it in a directory with `LonelyPartyArray.c`, `LonelyPartyArray.h`, the `sample_output` directory, and all the test case files, and then typing:

```
bash test-all.sh
```

Please note that these test cases are not comprehensive. You should also create your own test cases if you want to test your code comprehensively. In creating your own test cases, you should always ask yourself, "How could these functions be called in ways that don't violate the function descriptions, but which haven't already been covered in the test cases included with the assignment?"

6. Testing for Memory Leaks with Valgrind

Part of the credit for this assignment will be awarded based on your ability to implement the program without any memory leaks. To test for memory leaks, you can use a program called `valgrind`, which is installed on Eustis.

`valgrind` will **not** guarantee that your code is completely free of memory leaks. It will only detect whether any memory leaks occur when you run your program. So, if you have a function called `foo()` that has a nasty memory leak, but you run your program in such a way that `foo()` never gets called, `valgrind` won't be able to find that potential memory leak.

The `test-all.sh` script will automatically run your program through all test cases and use `valgrind` to check whether any of them result in memory leaks. If you want to run `valgrind` manually, simply compile your program with the `-g` flag, and then run it through `valgrind`, like so:

```
gcc LonelyPartyArray.c testcase01.c -g
valgrind --leak-check=yes ./a.out
```

In the output of `valgrind`, the magic phrase you're looking for to indicate that no memory leaks were detected is:

```
All heap blocks were freed – no leaks are possible
```

For more information about `valgrind`'s output, see: http://valgrind.org/docs/manual/quick-start.html

7. Reference: Compilation and Testing (Code::Blocks)

The key to getting multiple files to compile into a single program in Code::Blocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in Code::Blocks, which involves importing `LonelyPartyArray.h`, `testcase01.c`, and the `LonelyPartyArray.c` file you've created (even if it's just an empty file so far).

1. Start Code::Blocks.
2. Create a New Project (*File → New → Project*).
3. Choose "Empty Project" and click "Go."
4. In the Project Wizard that opens, click "Next."
5. Input a title for your project (e.g., "LonelyPartyArray").
6. Choose a folder (e.g., Desktop) where Code::Blocks can create a subdirectory for the project.
7. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into Code::Blocks. Then right click the tab for **each** file and choose "Add file to active project."

   – or –

2. Go to "*Project → Add Files...*" and browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9).

Note that if you import more than one test case source file (e.g., both `testcase01.c` *and* `testcase02.c`), the compiler will complain that you have multiple definitions for `main()`. You can only have one of those in there at a time. You'll have to swap them out as you test your code.

Note! Even if you do most of your development with Code::Blocks, you still need to test your code on Eustis. Remember that your code must compile and run on Eustis in order to receive credit.

Continued on the following page...

8. Reference: Compilation and Testing (Linux/Mac Command Line)

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 LonelyPartyArray.exe
```

...and then run the program using:

```
./LonelyPartyArray.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:

```
./LonelyPartyArray.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 contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

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

If the files differ, it 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:~$ _
```

9. Getting Started: A Guide for the Overwhelmed

Okay, so, this might all be overwhelming, and you might be thinking, “Where do I even start with this assignment?! I’m in way over my head!”

Don’t panic! There are oodles of TA office hours where you can get help, and here’s my general advice on starting the assignment:

1. First and foremost, sit down and read Section 1, “Overview,” on pg. 3. Read it carefully and thoroughly, and make sure you understand the diagram on pg. 4. This might take some time. It might help to print out this PDF and go read it somewhere quiet, with your phone turned off and no other distractions.

2. Glance through Sections 2 (“LonelyPartyArray.h”), 3 (“New Data Type: long long unsigned int”), and 4 (“Function Requirements”). Again, take some time to figure out what’s going on.

3. Start by creating a skeleton `LonelyPartyArray.c` file. Add a header comment, add some standard `#include` directives, and be sure to include `LonelyPartyArray.h` from your source file. Then copy and paste each functional prototype from `LonelyPartyArray.h` into `LonelyPartyArray.c`, and set up all those functions to return dummy values (zero, `NULL`, etc.). For example:

```
#include <stdio.h>
#include <stdlib.h>
#include "LonelyPartyArray.h"

LonelyPartyArray *createLonelyPartyArray(int num_fragments, int fragment_length)
{
   return NULL;
}

LonelyPartyArray *destroyLonelyPartyArray(LonelyPartyArray *party)
{
   return NULL;
}

int set(LonelyPartyArray *party, int index, int key)
{
   return 0;
}

// ...and so on.
```

4. Test that your `LonelyPartyArray.c` source file compiles. If you’re at the command line in Linux or on a Mac, your source file will need to be in the same folder as `LonelyPartyArray.h`, and you can test compilation like so:

```
gcc -c LonelyPartyArray.c
```

Alternatively, you can try compiling it with one of the test case source files, like so:

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

For more details, see Section 8, “Reference: Compilation and Testing (Linux/Mac Command Line).” If you’re using Code::Blocks, open it up and start a project using the instructions above in Section 7, “Reference: Compilation and Testing (Code::Blocks).” Import `LonelyPartyArray.h`, `testcase01.c`, and your new `LonelyPartyArray.c` source file, and get the program compiling and running before you move forward.

5. Once you have your project compiling, go back to the list of required functions (Section 4, “Function Requirements”), and try to implement one function at a time. Always stop to compile and test your code before moving on to another function!

6. You’ll probably want to start with the `createLonelyPartyArray()` function. As you work on `createLonelyPartyArray()`, write your own `main()` function that calls `createLonelyPartyArray()` and then checks the results. For example, you’ll want to ensure that `createLonelyPartyArray()` is returning a non-NULL pointer to begin with, and that the fields inside the `LonelyPartyArray` struct that it creates are properly initialized when you examine them back in `main()`. If you’re uncertain about how to call certain functions, read through my sample test case files for examples.

7. After writing `createLonelyPartyArray()`, I would probably work on the `set()`, `get()`, and `printIfValid()` functions, because they will be immensely useful in debugging your code as you work. Here’s how I’d test these functions at first: In your own `main()` function, call `createLonelyPartyArray()`. Then, back in `main()`, manually insert one or two integers into the LPA with the `set()` function, and call `printfIfValid()` on a few different indices (some valid, some invalid) to make sure everything is working as intended. If you get some unexpected output, trace carefully through your code to see what went wrong.

8. If you get stuck, draw diagrams. Make boxes for all the variables in your program. If you’re using pointers and dynamically allocated memory, diagram everything out and make up addresses for all your variables. Trace through your code carefully using these diagrams.

9. With so many pointers, you’re bound to encounter errors in your code at some point. Use `printf()` statements liberally to verify that your code is producing the results you think it should be producing (rather than making assumptions that certain components are working as intended). You should get in the habit of being immensely skeptical of your own code and using `printf()` to provide yourself with evidence that your code does what you think it does.

10. When looking for a segmentation fault, you should always be able to use `printf()` and `fflush()` to track down the *exact* line you’re crashing on. Alternatively, you can use a debugger (such as *gdb*) to help debug your code.

10. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named `LonelyPartyArray.c`, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work. Don't forget to `#include "LonelyPartyArray.h"` in your source code.

Do not submit additional source files, do not submit a modified `LonelyPartyArray.h` header file, and **do not** include a `main()` function in your `LonelyPartyArray.c` source file. Your source file must work with the `test-all.sh` script, and it must be able to compile in both of the following ways:

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

Be sure to include your name and NID as a comment at the top of your source file.

11. Grading

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

65%	correct output for test cases (not all of which have been release with this project)
10%	passes `valgrind` test cases (no memory leaks)
10%	implementation details (manual inspection of your code)
5%	`difficultyRating()` is implemented correctly
5%	`hoursSpent()` is implemented correctly
5%	source file is named correctly; spelling and capitalization count
10%	adequate comments and whitespace; source includes student name and NID

Important! Do not include a `main()` function in your code, do not use global variables, do not make any system calls (e.g., `system("pause")`), and do not read or write to any files.

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program throughly.

Additional points will be awarded for style (appropriate commenting and whitespace) and adherence to implementation requirements.

Start early. Work hard. Good luck!