Task 8: A Dynamic Array

Start with the following piece of code and implement a dynamically growing array of integers that can hold as many elements as one likes. When it's full and gets appended, it's capacity should double each time. The following slides will describe in detail what each function is supposed to do. You can use as many additional helper functions as you like. You can also leave out functions that are too hard or too easy for you. If you are confused as to how this is supposed to work, look at the tests in the end to see the `darray` being used.

```c
#include <stdlib.h>
#include <string.h>

struct darray{
    int* buffer; // the dynamic buffer
    size_t element_count; // the current number of elements
    size_t capacity; // the amount of elements the buffer can hold
}darray;
```

init

```
1  int darray_init(struct darray* arr, size_t initial_capacity);
```

`darray_init` initializes a new darray `arr` with storage space for `initial_capacity` `int`'s (hint: `malloc`).

Returns 0 on success, 1 if the allocation failed.

get

```
1  int darray_get(struct darray* arr, size_t index);
```

`darray_get` returns the value of the element of the array at index `index` (starting at 0).

The user of the function is responsible to not access element that are out of bounds;

set

```
1  void darray_set(struct darray* arr, size_t index, int value);
```

`darray_get` assigns the element of the array at index `index` (starting at 0) to `value`.

The user of the function is responsible for not accessing element that are out of bounds;

append

```
1 int darray_append(struct darray* arr, int value);
```

`darray_append` appends a new element with the value `value` at the end of the array. If the array doesn't have space for the additional element, reallocates the array, making its new capacity twice as large. (hint: malloc, memcpy, free).

Returns 0 on success, 1 on allocationerror. Try to keep the previous array 'alive' even in case of an allocation failiure.

erase

```
1 void darray_erase(struct darray* arr, size_t index);
```

`darray_erase` removes the element at index `index` (starting from 0) from the array shifting all following elements forward (hint: memmove).

The user of the function is responsible for not accessing element that are out of bounds;

insert

```
1  int darray_insert(struct darray* arr, int index, int value);
```

`darray_erase` inserts an element at index `index` `index` (starting from 0). All elements that were at and after that index move one index forward (hint: memmove). If the insertion would bring the array above it's capacity, it is reallocated to be twice as large.

Returns 0 on success, 1 on reallocation failiure. The user of the function is responsible for not accessing element that are out of bounds;

fin

```
1  void darray_fin(struct darray* arr);
```

`darray_init` frees all memory of the passed `arr`.

Cannot fail.

Tests

test1: basic usage

You can run these tests on your implementation to see if it works correctly.

```c
//expects your code above or included some other way
#include <assert.h>
void main(){
    struct darray arr;
    darray_init(&arr, 1);
    for(size_t i = 0; i <= 100; i++){
        darray_append(&arr, (int)i);
    }
    int sum = 0;
    for(size_t i = 0; i < arr.element_count; i++){
        sum += darray_get(&arr, i);
    }
    assert(sum == 5050);
    darray_fin(&arr);
}
```

test2: insert and erase

```
1  //expects your code above or included some other way
2  #include <assert.h>
3  void main(){
4      struct darray arr;
5      darray_init(&arr, 1);
6      for(size_t i = 0; i <= 100; i++){
7          darray_append(&arr, (int)i);
8          if(i % 10 == 5) darray_erase(&arr, i - 5);
9          if(i % 10 == 8) darray_insert(&arr, i - 5, (int)i);
10     }
11     int sum = 0;
12     for(size_t i = 0; i < arr.element_count; i++){
13         sum += darray_get(&arr, i);
14     }
15     assert(sum == 5130);
16     darray_fin(&arr);
17 }
```