

B1- Elementary Programming in C

B-CPE-110

Bootstrap

PushSwap



Bootstrap

PushSwap

repository name: CPE_pushswap_bootstrap_\$ACADEMICYEAR

repository rights: ramassage-tek

language: C

group size: 1



- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).



Remember to download the tar that is provided with the project description. It contains elements that you will need to execute certain steps in this Bootstrap.



Bubble sort

The first algorithm to code is the **bubble sort**, a very common sorting algorithm you already saw during the C Pool. It consists in swapping the elements.

Swapping: write a function that swaps two numbers in an array. It must be prototyped the following way:

```
void swap_elem(int *array, int index1, int index2)
```

To be confident that your previous two functions works as intended, you now have to write unit tests for them using Criterion.

Here's an example of unit test for the `swap_elem` function.

```
Test(swap_elem, index1_greater_than_index2)
{
    int array[6] = { 1, 2, 3, 4, 5, 6 };
    int index1 = 2;
    int index2 = 4;

    swap_elem(array, index1, index2);
    cr_assert_eq(array[index1], 5);
    cr_assert_eq(array[index2], 3);
}
```

But there's still more case to cover. For instance: `index2` lower than `index1`, `index1==index2`.

Sorting: write a function that uses the previous function to bubble sort an array, and prototyped the following way:

```
void bubble_sort_array(int *array, int size)
```



Since you can only use the `swap_elem` function you just wrote, you don't need to use `malloc`...



Some arrays have an indicator to announce the last element (`\0`, `NULL`,...), but some don't.... Well, when the array does not have such an indicator, how do you know where it ends?

Now, write unit tests for to ensure that whatever the state of the initial array, it comes out sorted after a call to the function.



<http://criterion.readthedocs.io/en/master/assert.html#array-assertions>



Allowed instructions

Let's study some instructions allowed in the project:

- **swap** switches the first two elements of the list.
- **rotate_left** takes the first element on the list and puts it at the end of the list.
- **rotate_right** takes the last element on the list and places it at the beginning of the list.

Using only the instructions listed above, try to hand-sort the following list of numbers in ascending order: 15, 8, 42, 4, 16, 23.

Write your instructions in the **sort-lost-numbers** file and test your solution using the following method:

```
Terminal
~/B-CPE-110> cat sort-lost-numbers | ./let-me-out-of-this-island
```

This first step only helps you to be more familiar with the restrictions.

Write a function that takes an array and its size as parameters and displays the instructions it takes to sort the array in ascending order. It must be prototyped the following way:

```
void my_amazing_sorter(int *array, int size)
```

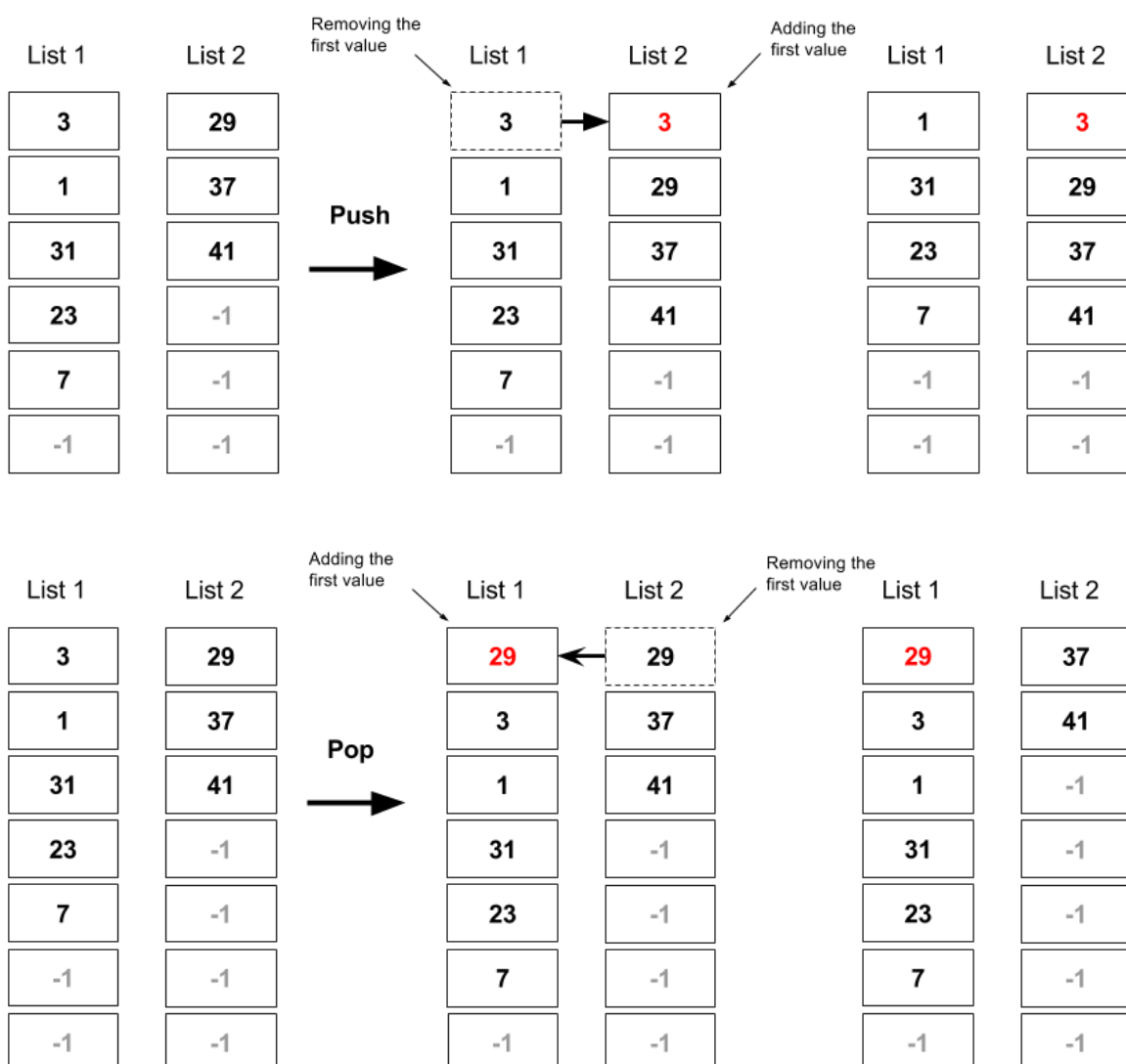
Two lists

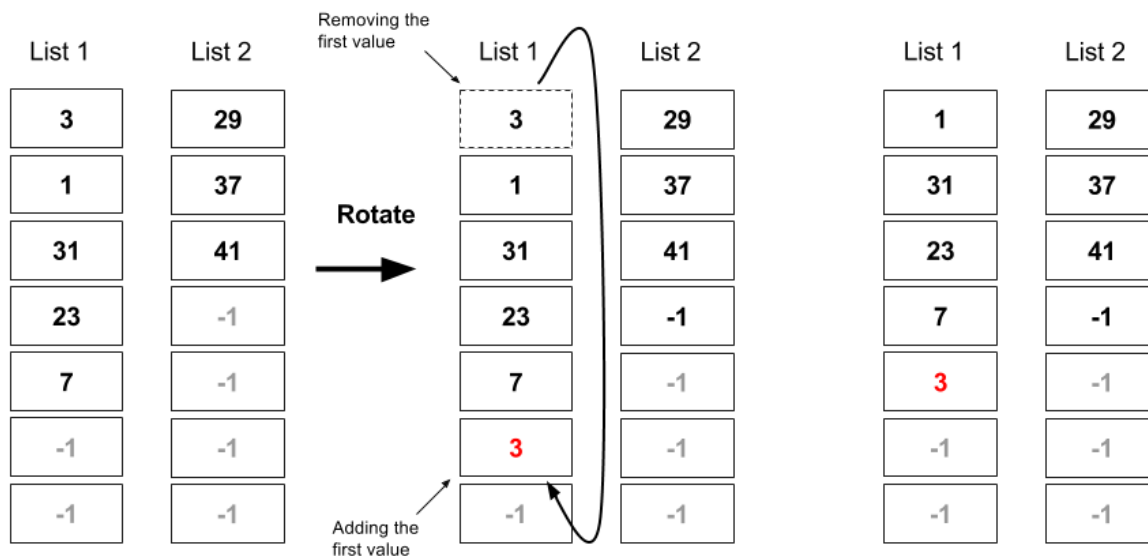
So far, your algorithm is still based upon the bubble sort and elements swapping.
Let's add the last feature of the PushSwap project: using two lists of numbers to optimize the sort.

The challenge will be to create a sorting algorithm based solely on the three following operations:

- **push**: removes **LIST1** first value and put it at the beginning of **LIST2**,
- **pop**: removes **LIST2** first value and put it at the beginning of **LIST1**,
- **rotate**: removes **LIST1** first value and put it at the end of **LIST1**.

The corresponding functions will be given to you.





The PushSwap project provides more operations than these ones.

Write a function that sorts a list with those three operations. It must be prototyped the following way:

```
void sort_numbers(int *array1, int *array2, int size)
```

Initially, the first array is filled with non-negative numbers to be sorted, while the second array is empty (that is, with cells initialized at -1). Both arrays have the same size.

The expected result is a **LIST1** with numbers sorted in ascending order.

Use the given graphically tool to visualize your algorithm's behavior: **pimp-my-algo**.

The three operations (**push**, **pop** and **rotate**) are provided by this tool, and must be called as pointers.

The main function is also included in this tool. It calls directly your **sort_numbers** function.

You must simply place the **sort_numbers** in the **sort_numbers.c** file and launch **make** to compile the program.



In order to consult the functions' prototypes, read the **pimp-my-algo.h** file.



Full set of operations

Let's get rid of **pimp-my-algo** now!

Recode the three operations, **push**, **pop** and **rotate**, and integrate them in your existing code to get a standalone program.

Add some more operations, and build you pushswap!