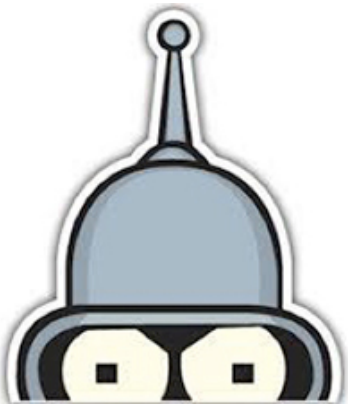
B1 - Elementary Programming in C

B-CPE-110

Pushswap Bootstrap

Let's get started



2.0

Pushswap Bootstrap

language: C

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

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);
}
```

There're 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.

Criterion: array assertions

ALLOW INSTRUCTIONS

Let's study some instructions allowed in the project:

- **swap** switches the first two elements of the list (used for **sa** and **sb** in the project).
- **rotate_left** takes the first element on the list and puts it at the end of the list (**ra** and **rb** in the project).
- **rotate_right** takes the last element on the list and places it at the beginning of the list (**rra** and **rrb** in the project).

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.

Now, 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);
```

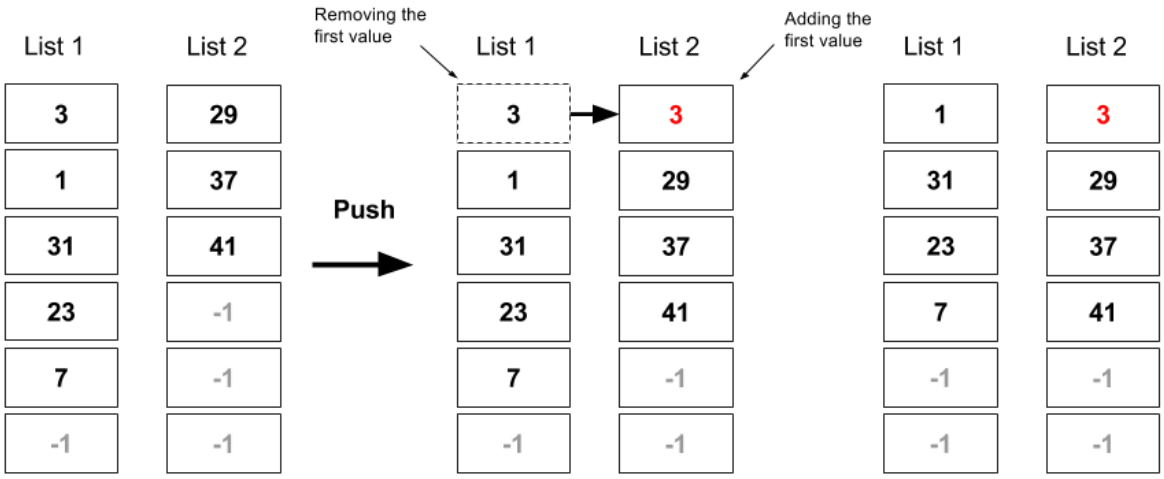
TWO LISTS

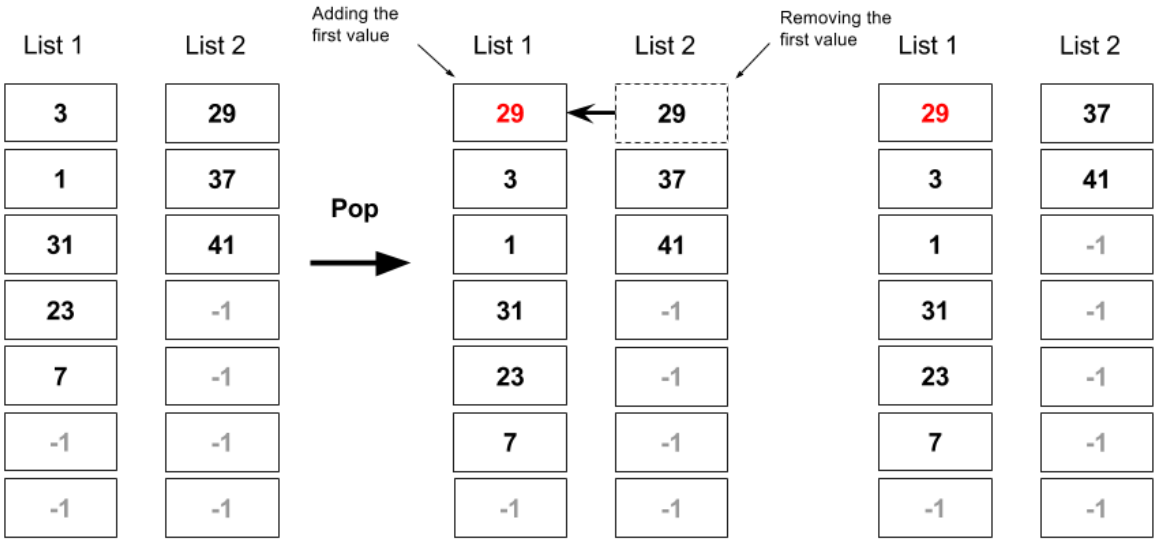
So far, your algorithm is still based upon the bubble sort and elements swapping.
Let's add the last feature of the PushSwap project: using two lists of numbers to optimize the sort.

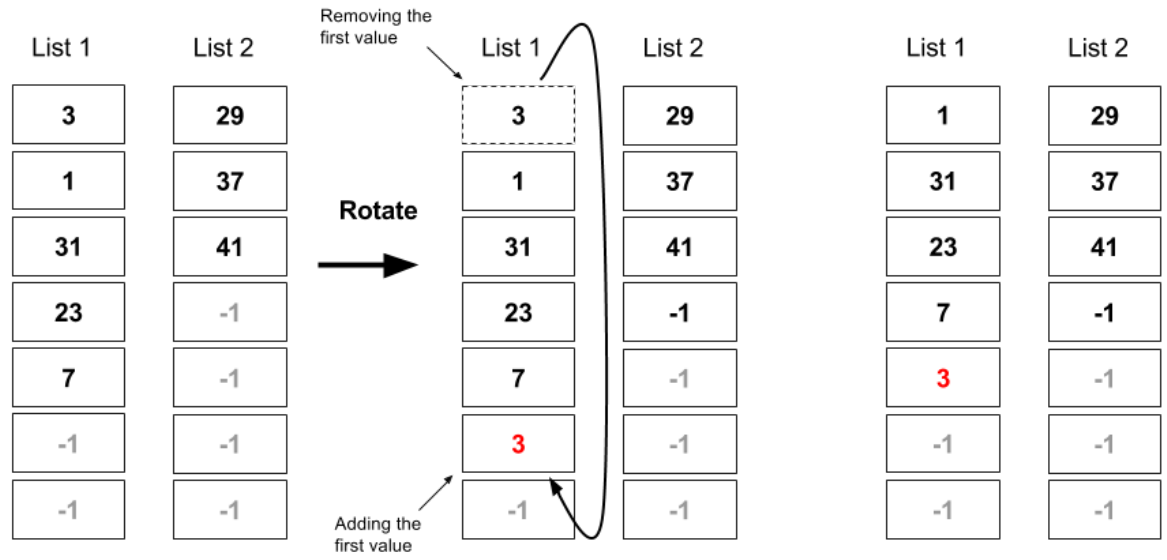
The challenge will be to create a sorting algorithm based solely on the three following operations:

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

The corresponding functions will be given to you.


List 1
List 2
3
29
1
37
31
41
23
-1
7
-1
-1
-1
Push
Removing the first value
Adding the first value
List 1
List 2
3
3
1
29
31
37
23
41
7
-1
-1
-1
-1
-1
1
3
31
29
23
37
7
41
-1
-1
-1
-1


List 1
List 2
3
29
1
37
31
41
23
-1
7
-1
-1
-1
-1
-1
Pop
Adding the first value
Removing the first value
List 1
List 2
29
29
3
37
1
41
31
-1
23
-1
7
-1
-1
-1
-1
-1
29
37
3
41
1
-1
31
-1
23
-1
7
-1
-1
-1



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!