# Push\_Swap Project Documentation

## 1. Introduction

The goal is to generate the shortest possible sequence of operations that, when executed, will sort the stack in ascending order (with the smallest element at the top).

## 2. Objectives

* Algorithmic Optimization:  
  Develop and implement a sorting algorithm that minimizes the number of operations.
* Data Structures:  
  Work with two stacks (a and b) and manipulate them using a limited set of operations.
* Error Handling:  
  Validate input to ensure that all arguments are valid integers (within the range of an int) and that there are no duplicate values.
* Efficiency Benchmarks:  
  Although optional for the mandatory part, benchmarks typically include sorting 100 random numbers in fewer than 700 operations and 500 numbers in fewer than 5500 operations.

## 3. Project Overview and Architecture

The project is divided into several modules:

* Argument Parsing (parse\_args.c):  
  Processes command-line arguments, checks for validity (non-duplicates, integer range, etc.), and converts them into an array of integers.
* Stack Implementation (stack.c):  
  Implements the stack abstract data type.
  + stack\_push adds a new element at the top of the stack.
  + stack\_pop removes the top element.
  + stack\_size calculates the number of elements in the stack.
* Operations (operations.c):  
  Implements basic operations on stacks:
  + sa/sb/ss: Swap the first two elements of a stack.
  + pa/pb: Push the top element from one stack to the other.
* Rotations:
  + rotate.c: Implements ra, rb, and rr to shift up all elements (first becomes last).
  + reverse\_rotate.c: Implements rra, rrb, and rrr to shift down all elements (last becomes first).
* Sorting for Small Inputs (sort\_small.c):  
  Contains functions such as sort\_three and sort\_small to sort stacks with 2–5 elements using a minimal set of operations.
  + For example, for 2 elements, the optimal move is a single swap (sa); for 3 elements, sort\_three decides among sa, ra, and rra based on the order.
* Radix Sort for Larger Inputs (sort\_radix.c):  
  Implements a Radix Sort algorithm for stacks with more than 5 elements.
  + It uses a helper function sort\_array (a bubble sort) and index\_stack to convert the numbers into a consecutive range (0 to n‑1) before applying radix sort.
* Push\_Swap Selection (push\_swap.c):  
  This module decides which sorting algorithm to use based on the number of elements in stack a:

c

Copy

void push\_swap(t\_stack \*a, t\_stack \*b)

{

int size = stack\_size(a);

if (size == 2)

sa(a);

else if (size == 3)

sort\_three(a);

else if (size <= 5)

sort\_small(a, b);

else

radix\_sort(a, b);

}

* Main Program (main.c):  
  Handles argument checking, initializes stacks, fills the stack with values (ensuring that the first argument ends up at the top by inserting elements in reverse order), and then calls push\_swap(a, b) if the stack is not already sorted. It also handles error reporting by printing "Error" to stderr when needed.

## 4. Usage Instructions

### Compilation

A Makefile is provided which compiles the project. The Makefile includes the required targets:

* make or make all: Compile the project and generate the push\_swap executable.
* make clean: Remove object files.
* make fclean: Remove object files and the executable.
* make re: Recompile everything from scratch.

From the project root, run:

sh

Copy

make

### Running the Program

The push\_swap program expects its arguments as a list of integers. For example:

sh

Copy

./push\_swap 2 1 3 6 5 8

The output will be a list of operations (each separated by a newline) that, when applied to the initial stack, will sort it in ascending order (with the smallest element at the top).

### Error Handling

* If the input contains non-integer values, values that exceed the limits of an int, or duplicates, the program should output "Error" (followed by a newline) to stderr.
* If no arguments are provided, the program should not output anything.

### Testing with Checker

A bonus binary, checker\_linux, is provided to validate your solution. For example:

sh

Copy

./push\_swap 2 1 3 6 5 8 | ./checker\_linux 2 1 3 6 5 8

If the sequence of operations correctly sorts the stack, checker\_linux will output "OK". Otherwise, it will output "KO".

## 5. Design Considerations

* Index Conversion:  
  For large inputs, before applying the radix sort, the program converts the numbers in the stack into indices (0 to n‑1) using the index\_stack function. This is essential because Radix Sort works best on a consecutive range of integers.
* Separation of Concerns:  
  Each module (parsing, stack operations, sorting) is separated into different files, which helps with maintenance and adherence to coding standards (Norm).
* Optimization for Small Cases:  
  For 2, 3, 4, or 5 elements, dedicated algorithms (sort\_three and sort\_small) are used because they produce fewer operations than the general algorithm.

## 6. Example Execution

Example Input:

sh

Copy

./push\_swap 2 1 3 6 5 8

Possible Output (Operations):

nginx

Copy

sa

pb

pb

pb

sa

pa

pa

pa

This means:

* Swap the first two elements of stack a.
* Push three elements to stack b.
* Swap a specific stack.
* Push all elements back to stack a.

Error Case:

sh

Copy

./push\_swap 0 one 2 3

Will output:

Error

## 7. Conclusion

The push\_swap project tests your ability to manipulate stacks, implement sorting algorithms under constraints, and optimize for a minimal number of moves. By following the specified operations and carefully selecting the sorting method based on input size, you ensure that your solution is both correct and efficient. The provided Makefile automates the compilation process, and the checker program verifies that your sequence of moves indeed sorts the stack.