PUSH - SWAP ( )

Parseo

```
//leer_args.c
void    liberar_array (char *array);
int     free_stacks (t_stacks *s);
int     list_nums (char const *str);
t_stacks init_stacks (int *nums, int size);
t_list  *new_stack_node (int n);
```

Comprobaciones

```
//check.c
int     checkOrder (t_list *ord);
int     norep (char *argv, int argc);
int     checknum (char *argv, int argc);
```

Utils

```
// utils.c
int     get_index (int *arr, int size, int val);
int     get_pos (t_list *lst, int val);
t_list  *lstnum (t_list *lst, int num);
int     minlst (t_list *lst);
void    ft_puntlist (t_list *a);
```

Ordenación

```
//sortone
void    sortdos (t_list *a);
void    sortthree (t_list *a);
void    sort_four_or_five (t_stacks *s);
void    selector (t_stacks *stacks);
```

```
//algorithm.c
void    radix_sort (t_list **a, t_list **b);
void    radix_pass (t_list **a, t_list **b,
                    int i, int size);
void    index_stacks (t_list *a);
int     *list_2_array (t_list *a, int size);
void    sort_array (int *arr, int size);
```

Estructuras de datos

```
typedef struct  s_stacks
{
    t_list  *a;
    t_list  *b;
} t_stacks;

// Linked list
typedef struct  s_list {
    void    *content;
    struct s_list   *next;
} t_list;
```

Algoritmo

- index_stacks()
  - usa
    - list_2_array()
    - sort_array()
    - get_index()
- radix_sort()
  - radix_pass()

Flujo del main

`int main(int argc, char *argv[])`

→ Control del input

Comprobar que se hayan pasado argumentos

```
if (argc < 2)
    return 0;
```

usa ↷ *ft_isdigit()
× ft_strlen()

`if(!checknum(argc,argv) || !norep(argv, argc))`

- `int norep(char *argv, int argc);` — compruela que no hayan dígitos repetidos
- `int checknum(char *argv, int argc);` — Controla el input (que sean números enteros) → usa `ft_strcmp()`

Parseo de argumentos

ft_atoi

`[./push_swap]`
`[53]`
`[6]`
`[7]`

convierte el array a una lista

`int list_nums(char *argv, int size);` → `[53, 6, 7]`

`t_stack *init_stack(int *nums, int size);`

- Inicializa ambas listas en el stack NULL
- para cada número del array, crea un nodo de la linked list `*new_stack_node(nums[i])`
- Si todo va bien anexa el nodo al final de la lista `ft_lstadd_back(&s.a, node)`

s.a	s.b
53	NULL
6	
7	
NULL	

cada posición del array

Algoritmo de ordenación

`void selector(t_stack *stack)`

hardcoded
- → `sort_two(t_list *a);`
- → `sort_three(t_list *a);`
- → `sort_four_or_five(t_stack *s);`
- → `radix_sort(t_list **a, t_list **b);`
- → `radix_sort(t_list **a, t_list **b, int i, int size);`

Ejemplos

s→a : -10 → 20 → 0 → -30

list2array [-10, 20, 0, -30]

sortarray [-30, -10, 0, 20] (bubble sort)

indexstack [1 → 3 → 2 → 0]

Se llama antes de aplicar radix

`void index_stack(t_list *a);`

Radix binario no acepta valores negativos, por eso esta función convierte los valores a índices

max_bits: cuantos bits son necesarios para representar al numero más grande de la lista

`radix_sort(t_list **a, t_list **b)` → ordena a través de radix

- Verifica si ya está ordenado
- Calcula el tamaño de la lista
- Calcula max_bits `while((size-1) >> max_bits)`
  - desplaza (size-1) a la derecha bit a bit
  - por cada desplazamiento incrementa max_bits
- Empieza `radix_pass(a, b, i, size)`
  - mueve los bits que tengan el índice i = 0 hacia b
  - los que tienen 1 los deja en a
- al terminar la iteración, mover todo de b a a
- comprobar si ya está ordenado

operador right shift >> — es como descomponer en factores primos

Ej. 1

8 >> 1	4
8 >> 2	2
8 >> 3	1

cada desplazamiento es quitar un bit a la derecha

Ej. 2 — con size = 8 (-1)

7 >> 0	[111] = 7
7 >> 1	[110] = 3
7 >> 2	[100] = 1
7 >> 3	[000] = 0