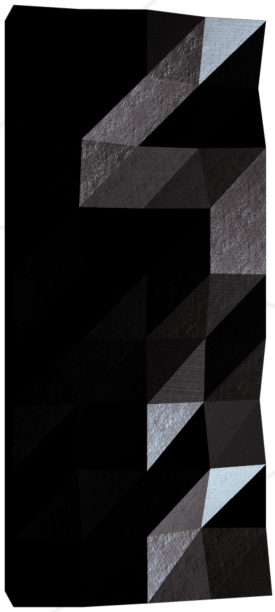

Push_swap

Swap_push no es tan natural

Resumen:
Este proyecto te hará ordenar datos en un stack, con un conjunto limitado de instrucciones, y utilizando el menor número posible de acciones. Para tener éxito, deberás probar a utilizar varios tipos de algoritmos y elegir la solución más apropiada (de entre muchas posibles) para conseguir la ordenación optimizada de los datos.

Versión: 8.1

Índice general

Capítulo I

Avance

- C

```c
#include <stdio.h>

int main(void)
{
   printf("hello, world\n");
   return 0;
}
```

- ASM

```
cseg segment
assume cs:cseg, ds:cseg
org 100h
main proc
jmp debut
mess db 'Hello world!$'
debut:
mov dx, offset mess
mov ah, 9
int 21h
ret
main endp
cseg ends
end main
```

- LOLCODE

```
HAI
CAN HAS STDIO?
VISIBLE "HELLO WORLD!"
KTHXBYE
```

- PHP

```php
<?php
 echo "Hello world!";
?>
```

- BrainFuck

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]
>++.>+.+++++++..+++.>++.
<<++++++++++++++.>.+++.------.--------.>+.>.
```

- C#

```
using System;

public class HelloWorld {
   public static void Main () {
      Console.WriteLine("Hello world!");
   }
}
```

- HTML5

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Hello world !</title>
   </head>
   <body>
      <p>Hello World !</p>
   </body>
</html>
```

- YASL

```
"Hello world!"
print
```

- OCaml

```
let main () =
   print_endline "Hello world !"

let _ = main ()
```

Capítulo II

Introducción

Push_swap es un proyecto de algoritmia simple y muy efectivo: tienes que ordenar datos.

Tienes a tu disposición un conjunto de valores enteros, 2 stacks y un conjunto de instrucciones para manipular ambos stacks.

¿Cuál es tu objetivo? Escribir un programa en `C` llamado `push_swap`. El programa calculará y mostrará en la salida estándar el programa más pequeño, creado con las instrucciones de *lenguaje Push swap*, que ordene los números enteros recibidos como argumentos.

¿A que es fácil?

Bueno, eso ya lo veremos...

Capítulo III

Objetivos

Escribir un algoritmo de ordenamiento es siempre un paso muy importante en el viaje de un desarrollador. Frecuentemente es el primer encuentro con el concepto de complejidad.

Los algoritmos de ordenamiento y su complejidad suponen una importante parte de las preguntas realizadas durante las entrevistas laborales. Es posiblemente un buen momento para echar un vistazo a estos conceptos ya que tendrás que enfrentarte a ellos en algún momento de tu vida.

Los objetivos de aprendizaje de este proyecto son rigor, uso de `C`, y el uso de algoritmos básicos... haciendo especial hincapié en su complejidad

Ordenar valores es simple. Ordenarlos de forma rápida es menos simple, especialmente porque de una configuración de enteros a otra, la solución más eficiente para ordenar, puede diferir.

Capítulo IV

Instrucciones generales

- Tu proyecto deberá estar escrito en C.
- Tu proyecto debe estar escrito siguiendo la Norma. Si tienes archivos o funciones adicionales, estas están incluidas en la verificación de la Norma y tendrás un `0` si hay algún error de norma en cualquiera de ellos.
- Tus funciones no deben terminar de forma inesperada (segfault, bus error, double free, etc) excepto en el caso de comportamientos indefinidos. Si esto sucede, tu proyecto será considerado no funcional y recibirás un `0` durante la evaluación.
- Toda la memoria asignada en el heap deberá liberarse adecuadamente cuando sea necesario. No se permitirán leaks de memoria.
- Si el enunciado lo requiere, deberás entregar un `Makefile` que compilará tus archivos fuente al output requerido con las flags `-Wall`, `-Werror` y `-Wextra`, utilizar cc y por supuesto tu `Makefile` no debe hacer relink.
- Tu `Makefile` debe contener al menos las normas `$(NAME)`, `all`, `clean`, `fclean` y `re`.
- Para entregar los bonus de tu proyecto deberás incluir una regla `bonus` en tu `Makefile`, en la que añadirás todos los headers, librerías o funciones que estén prohibidas en la parte principal del proyecto. Los bonus deben estar en archivos distintos `_bonus.{c/h}`. La parte obligatoria y los bonus se evalúan por separado.
- Si tu proyecto permite el uso de la `libft`, deberás copiar su fuente y sus `Makefile` asociados en un directorio `libft` con su correspondiente `Makefile`. El `Makefile` de tu proyecto debe compilar primero la librería utilizando su `Makefile`, y después compilar el proyecto.
- Te recomendamos crear programas de prueba para tu proyecto, aunque este trabajo **no será entregado ni evaluado**. Te dará la oportunidad de verificar que tu programa funciona correctamente durante tu evaluación y la de otros compañeros. Y sí, tienes permitido utilizar estas pruebas durante tu evaluación o la de otros compañeros.
- Entrega tu trabajo en tu repositorio `Git` asignado. Solo el trabajo de tu repositorio `Git` será evaluado. Si Deepthought evalúa tu trabajo, lo hará después de tus com-

pañeros. Si se encuentra un error durante la evaluación de Deepthought, esta habrá terminado.

Capítulo V

Parte obligatoria

V.1. Las reglas

- Tienes 2 stacks, llamados `a` y `b`.
- Para empezar:
  - El stack `a` contiene una cantidad aleatoria de números positivos y/o negativos, nunca duplicados.
  - El stack `b` está vacío.
- El objetivo es ordenar los números del stack `a` en orden ascendente. Para hacerlo tienes las siguientes operaciones a tu disposición:

sa `swap a`: Intercambia los dos primeros elementos del stack `a`. No hace nada si hay uno o menos elementos.

sb `swap b`: Intercambia los dos primeros elementos del stack `b`. No hace nada si hay uno o menos elementos.

ss `swap a` y `swap b` a la vez.

pa `push a`: Toma el primer elemento del stack `b` y lo pone el primero en el stack `a`. No hace nada si `b` está vacío.

pb `push b`: Toma el primer elemento del stack `a` y lo pone el primero en el stack `b`. No hace nada si `a` está vacío.

ra `rotate a`: Desplaza hacia arriba todos los elementos del stack `a` una posición, de forma que el primer elemento se convierte en el último.

rb `rotate b`: Desplaza hacia arriba todos los elementos del stack `b` una posición, de forma que el primer elemento se convierte en el último.

rr `ra` y `rb` al mismo tiempo.

rra `reverse rotate a`: Desplaza hacia abajo todos los elementos del stack `a` una posición, de forma que el último elemento se convierte en el primero.

rrb `reverse rotate b`: Desplaza hacia abajo todos los elementos del stack `b` una posición, de forma que el último elemento se convierte en el primero.

rrr `rra` y `rrb` al mismo tiempo.

V.2. Ejemplo

Para ilustrar el funcionamiento de algunas de estas instrucciones, vamos a ordenar una lista de números aleatoria. En el siguiente ejemplo, asumiremos que ambos stacks crecen por la derecha.

```
---------------------------------------------------------------------------------------------------------------
Init a and b:
2
1
3
6
5
8

- -
a b
---------------------------------------------------------------------------------------------------------------
Exec sa:
1
2
3
6
5
8

- -
a b
---------------------------------------------------------------------------------------------------------------
Exec pb pb pb:
6 3
5 2
8 1

- -
a b
---------------------------------------------------------------------------------------------------------------
Exec ra rb (equiv. to rr):
5 2
8 1
6 3

- -
a b
---------------------------------------------------------------------------------------------------------------
Exec rra rrb (equiv. to rrr):
6 3
5 2
8 1

- -
a b
---------------------------------------------------------------------------------------------------------------
Exec sa:
5 3
6 2
8 1

- -
a b
---------------------------------------------------------------------------------------------------------------
Exec pa pa pa:
1
2
3
5
6
8

- -
a b
---------------------------------------------------------------------------------------------------------------
```

Este ejemplo ordena los enteros de `a` en 12 instrucciones. ¿Puedes hacerlo mejor?

V.3. El programa: “push_swap”

Nombre de programa	`push_swap`
Archivos a entregar	`Makefile, *.h, *.c`
Makefile	`NAME, all, clean, fclean, re`
Argumentos	`stack a: una lista de números enteros`
Funciones autorizadas	• `read, write, malloc, free, exit` • `ft_printf y cualquier función equivalente que hayas creado`
Se permite usar libft	`Yes`
Descripción	`Ordenar stacks`

Tu programa deberá cumplir la siguientes normas:

- Tienes que entregar un `Makefile` que compile tus archivos fuente. No debe hacer relink.
- Las variables globales están prohibidas.
- Debes escribir un programa llamado `push_swap` que recibirá como argumento el stack `a` con el formato de una lista de enteros. El primer argumento debe ser el que esté encima del stack (cuidado con el orden).
- El programa debe mostrar la lista de instrucciones más corta posible para ordenar el stack `a`, de menor a mayor, donde el número menor se sitúe en la cima del stack.
- Las instrucciones deben separarse utilizando un “\n” y nada más.
- El objetivo es ordenar el stack con el mínimo número de operaciones posible. Durante la evaluación compararemos el número de instrucciones obtenido por tu programa con un rango de instrucciones máximo. Si tu programa muestra una lista demasiado larga o si el resultado no es correcto, tu nota será 0.
- Si no se especifican parámetros, el programa no deberá mostrar nada y deberá devolver el control al usuario.
- En caso de error, deberá mostrar `Error` seguido de un “\n” en la salida de errores estándar. Algunos de los posibles errores son: argumentos que no son enteros, argumentos superiores a un número entero, y/o encontrar números duplicados.

```
$>./push_swap 2 1 3 6 5 8
sa
pb
pb
pb
sa
pa
pa
pa
$>./push_swap 0 one 2 3
Error
$>
```

Durante la evaluación tendrás a tu disposición un binario para verificar el correcto funcionamiento de tu programa. Funciona de la siguiente forma:

```
$>ARG="4 67 3 87 23"; ./push_swap $ARG | wc -l
       6
$>ARG="4 67 3 87 23"; ./push_swap $ARG | ./checker_OS $ARG
OK
$>
```

Si el programa `checker_OS` muestra `"KO"`, implicará que tu programa `push_swap` utiliza una lista de instrucciones que no ordena los números.

El programa checker_OS está disponible en los recursos del proyecto, en la Intranet.
Puedes encontrar su funcionamiento descrito en la sección Parte bonus de este documento.

V.4. Pruebas de rendimiento

Para validar este proyecto, deberás realizar ciertas pruebas con un número mínimo de operaciones:

- Para una **validación mínima del proyecto** (que implica una nota mínima de 80), deberás ser capaz de **ordenar 100 números aleatorios en menos de 700 operaciones**.
- Para una **validación máxima del proyecto** y así poder obtener los bonus, deberás cumplir el primer paso anterior, pero también para **500 números aleatorios**, no deberás superar las **5500 operaciones**.

Todo esto será verificado durante la evaluación.

Si deseas completar la parte bonus, deberás validar el proyecto con cada paso de esta prueba de rendimiento, obteniendo la puntuación más alta posible.

Capítulo VI

Parte bonus

Este proyecto deja muy poco margen para añadir funcionalidades extra debido a su simplicidad. Sin embargo, ¿qué te parece crear tu propio checker?

Gracias al programa checker podrás probar si la lista de instrucciones generada por el programa push_swap realmente ordena el stack de forma correcta.

La parte bonus no será evaluada si la parte obligatoria no está perfecta. Perfecta quiere decir que se ha completado la parte obligatoria y que funciona perfectamente, sin errores. En este proyecto esto implica validar todos las pruebas de rendimiento sin excepción. Si no has pasado todas las pruebas obligatorias, tu parte bonus no será evaluada.

VI.1. El programa “checker”

Nombre de programa	`checker`
Archivos a entregar	`Makefile, *.h, *.c`
Makefile	`bonus`
Argumentos	`stack a: una lista de números enteros`
Funciones autorizadas	• `read, write, malloc, free, exit` • `ft_printf y cualquier función equivalente que hayas creado`
Se permite usar libft	`Yes`
Descripción	`Ejecuta las instrucciones de ordenación`

- Escribe un programa llamado `checker`, que tome como argumento el stack `a` en forma de una lista de enteros. El primer argumento debe estar encima del stack (cuidado con el orden). Si no se da argumento, `checker` termina y no muestra nada.
- Durante la ejecución de `checker` se esperará y leerá una lista de instrucciones, separadas utilizando ’\n’. Cuando todas las instrucciones se hayan leído, `checker` las ejecutará utilizando el stack recibido como argumento.
- Si tras ejecutar todas las instrucciones, el stack `a` está ordenado y el stack `b` vacío, entonces el programa `checker` mostrará `"OK"` seguido de un ’\n’ en el stdout.
- En cualquier otro caso, deberá mostrar `"KO"` seguido de un ’\n’ en el stdout.
- En caso de error, deberás mostrar `Error` seguido de un ’\n’ en la **stderr**. Los errores incluyen, por ejemplo, algunos o todos los argumentos no son enteros, algunos o todos los argumentos son más grandes que un número entero, hay duplicados, una instrucción no existe y/o no tiene el formato correcto.

```
$>./checker 3 2 1 0
rra
pb
sa
rra
pa
OK
$>./checker 3 2 1 0
sa
rra
pb
KO
$>./checker 3 2 one 0
```

```
Error
$>./checker "" 1
Error
$>
```

No tienes que clonar exactamente el comportamiento exacto del binario que te damos. Es obligatorio gestionar errores pero es decisión tuya cómo gestionar los argumentos.

Tus bonus serán evaluados exclusivamente si la parte obligatoria está PERFECTA. Perfecta quiere decir que se ha completado la parte obligatoria y que funciona perfectamente, sin errores. Si no has completado todos los requisitos de la parte obligatoria, la parte bonus no será evaluada.

Capítulo VII

Entrega y evaluación

Entrega tu trabajo en tu repositorio `git` como de costumbre. Solo el trabajo en tu repositorio será evaluado. No dudes en comprobar los nombres de tus archivos para asegurarte de que son correctos.

Ya que este proyecto no será verificado por un programa, siéntete libre de organizar como quieras tus ficheros, siempre que entregues todos los archivos obligatorios y que cumplan con las reglas especificadas.

¡Buena suerte!

file.bfe:VABB7yO9xm7xWXROeASsmsgnYOoOsDMJev7zFHhwQS8mvM8V5xQQp
Lc6cDCFXDWTiFzZ2H9skYkiJ/DpQtnM/uZO