Francisco Rubio Illán

frubio-i

frubio-i@student.42barcelona.com

PUSH_SWAP

Subject

Archivos a entregar: Makefile, *.h, *.c.

Makefile: all, clean, fclean, re, ... Sin relink en ningún caso.

Argumentos: una lista de números enteros.

Funciones autorizadas:

• read, write, malloc, free, exit

• ft_printf y cualquier función eqivalente programada por nosotros.

Se permite usar **libft**: Yes.

Descripción: Ordenar stacks

Objetivos:

- Escribir un algoritmo de ordenamiento. Primer encuentro con el concepto de complejidad (Big O notation).
- Los objetivos de aprendizaje de este proyecto son rigor, uso de C, y el uso de algoritmos básicos... haciendo especial hincapié en su complejidad.

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.

MOVIMIENTOS

- **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.

BONUS

• Conseguir hacer tu propo checker. Comprobar con el si la lista generada de instrucciones en push_swap realmente ordena el stack de forma correcta.

INVESTIGACIONES

RADIX (Radix sort) funciona basándose en la idea de **ordenar** elementos **procesando** sus **dígitos** de forma individual.

Su **lógica** se basa en:

Identificar la base numérica.

El algoritmo trabaja con la base numérica utilizada para representar los elementos (generalmente base 10 para números decimales).

Determinar el dígito más significativo.

Este es el primer dígito de los elementos (el más a la izquierda).

Realizar pasadas por RADIX.

Se procesa cada dígito, comenzando por el dígito menos significativo y avanzando hacia el más significativo.

Distribuir elementos.

En cada pasada, los elementos se distribuyen en "cubos" o "buckets" según su valor en el dígito actual.

Reagrupar elementos.

Los elementos dentro de cada cubo se mantienen en su orden relativo.

Repetir para los siguientes dígitos.

El proceso se repite para cada dígito, avanzando hacia el más significativo.

Finalización.

Una vez procesados todos los dígitos, los elementos estarán completamente ordenados.

El algoritmo **RADIX** es **no comparativo**, lo que significa que no necesita comparar elementos entre sí. En su lugar, **utiliza** la **base numérica** para **distribuir** y reagrupar los elementos de forma **eficiente**.

Sus **principales características** son:

• Eficiente para ordenar números enteros largos o cadenas alfanuméricas.

• Es estable, manteniendo el orden relativo de elementos con valores iguales.

• Requiere espacio adicional por los cubos/buckets utilizados.

También se puede determinar los dígitos menos significativos (LSD) e intermedios dividiendo entre 10 y 100, y usar el módulo 10 para el LSD.

Determinación de los dígitos

• Dígito **menos significativo** (LSD):

  ◦ Para el LSD, puedes usar el módulo 10: n % 10

  ◦ Ejemplo: Para el número 123, LSD = 123 % 10 = 3

• Dígitos **intermedios**:

  ◦ Para el dígito de las decenas, divide por 100 y toma el módulo 10: (n // 10) % 10

  ◦ Ejemplo: Para el número 123, dígito de las decenas = (123 // 10) % 10 = 2

• Dígitos **anteriores**:

  ◦ Para el dígito de las centenas, divide por 1000 y toma el módulo 10: ((n // 100) // 10) % 10

  ◦ Ejemplo: Para el número 123, dígito de las centenas = ((123 // 100) // 10) % 10 = 1

A raíz de esta investigación, he llegado a la conclusión que esta forma no es la más eficiente y he descubierto así que se podía hacer con operadores bitwise, poniéndome entonces a investigar sobre ello. Básicamente reduces los buckets/cubos a 2 (antes 9, basado en los dígitos), recorres la lista y haces push_b todo el que su binario termine en 0, una vez hecho, mueves con push_a todo de nuevo desde B, recorres >> 1 bit por iteración, hasta llegar al final. Eventualmente todo estará ordenado.

Siempre va a ser más eficiente ordenar y buscar a nivel binario por lo que he terminado entendiendo, además de ser un must en entrevistas/pruebas de código. Me decidí por hacer Radix recorriendo bits por esa razón y porque en el propio examen Rank 02 había que aprenderlos. Me pareció lo más lógico.

Operadores BITWISE

- **& (Bitwise AND)**: Este operador compara cada bit de dos números. Si ambos bits en la misma posición son 1, el resultado en esa posición será 1; de lo contrario, será 0.
  - **Uso común**: Verificar si ciertos bits están activados (en 1) en una máscara de bits.
    - **Ejemplo**: 5 & 3 (en binario, 0101 & 0011 resulta en 0001, que es 1 en decimal).
- **| (Bitwise OR)**: Compara cada bit de dos números. Si al menos uno de los bits en la misma posición es 1, el resultado en esa posición será 1; de lo contrario, será 0.
  - **Uso común**: Establecer (poner en 1) ciertos bits específicos en una máscara de bits.
    - **Ejemplo**: 5 | 3 (0101 | 0011 resulta en 0111, que es 7 en decimal).
- **^ (Bitwise XOR)**: Compara cada bit de dos números. Si los bits en la misma posición son diferentes, el resultado será 1; si son iguales, el resultado será 0.
  - **Uso común**: Intercambiar valores sin una variable temporal o "alternar" bits.
    - **Ejemplo**: 5 ^ 3 (0101 ^ 0011 resulta en 0110, que es 6 en decimal).
- **~ (Bitwise NOT)**: Invierte cada bit de un número, convirtiendo los 1 en 0 y los 0 en 1.
  - **Uso común**: Invertir todos los bits, como parte de operaciones matemáticas o para representar números negativos en sistemas de complemento a dos.
    - **Ejemplo**: ~5 (0101 en binario se convierte en 1010 en representación negativa en el sistema binario, lo cual depende del tamaño de bits, por lo que el resultado puede variar).

- **<< (Bitwise left shift)**: Mueve los bits de un número hacia la izquierda una cantidad especificada de posiciones. Cada posición desplazada a la izquierda multiplica el número por 2.
    - **Uso común**: Multiplicar rápidamente por potencias de 2.
        - **Ejemplo**: 5 << 1 (en binario, 0101 se convierte en 1010, que es 10 en decimal).

- **>> (Bitwise right shift)**: Mueve los bits de un número hacia la derecha una cantidad especificada de posiciones. Cada posición desplazada a la derecha divide el número por 2.
    - **Uso común**: Dividir rápidamente por potencias de 2.
        - **Ejemplo**: 5 >> 1 (0101 se convierte en 0010, que es 2 en decimal).

Estos **operadores** son **útiles** para **manipular** datos a **nivel** de **bits**, lo cual es común en **programación** de **bajo nivel** y **optimización** de recursos. Como en el manejo de **permisos**, configuraciones de **hardware**, y **optimización** de rendimiento en **operaciones** matemáticas **rápidas**.

TESTEOS (para fácil acceso)

LISTA DE 100 numeros sin repetirse (**1-1000**)

```
./push_swap 810 593 106 656 915 664 84 505 210 185 361 203 628 63
154 474 990 663 978 654 537 840 986 89 927 919 985 633 694 531 199
776 946 157 174 164 710 684 56 444 370 499 166 598 603 421 812 167
571 692 401 838 216 490 385 331 497 222 784 273 647 611 778 148
576 25 961 792 553 522 468 371 554 877 750 439 429 163 119 92 542
317 428 566 726 616 431 648 71 963 399 539 721 55 604 498 37 900
766 224
```

LISTA DE 500 numeros sin repetirse (**1-1000**)

```
./push_swap 679 579 506 398 977 963 55 758 472 440 669 529 263 891
726 904 931 619 21 993 86 745 467 252 730 666 170 501 280 583 959
197 105 1 565 422 303 490 58 146 272 409 779 165 885 870 747 164
489 985 278 786 879 169 445 444 804 235 411 16 881 727 581 793 297
944 773 265 469 76 903 279 555 642 460 215 381 638 816 534 930 590
17 51 516 905 966 600 338 121 877 208 240 191 844 893 900 825 520
318 179 806 908 499 228 594 205 71 113 754 473 656 635 659 200 331
865 212 771 209 438 856 264 647 698 957 541 537 819 199 243 582
961 77 823 641 157 310 244 975 135 442 775 917 214 799 922 834 353
759 938 543 392 482 547 5 889 241 79 14 74 578 559 491 867 504 762
648 950 397 468 33 274 548 233 949 449 441 968 765 232 497 190 753
351 704 27 788 376 423 485 416 601 618 391 830 623 117 503 379 281
245 390 110 603 577 599 194 973 678 256 424 50 602 276 515 675 926
952 596 686 979 681 457 222 395 568 650 919 1000 148 807 729 187
974 894 290 271 576 138 172 152 910 93 746 755 370 439 340 821 20
255 863 226 195 781 104 680 838 892 11 388 822 300 995 431 848 421
597 858 663 553 314 249 366 572 260 476 687 9 400 897 393 206 737
333 288 287 614 923 551 857 321 760 174 965 325 448 311 718 918
301 196 960 837 502 328 84 842 224 479 168 989 689 114 536 624 774
732 25 728 112 486 776 971 358 645 941 942 734 456 163 339 359 817
569 598 714 384 67 90 18 99 52 149 616 911 83 22 454 24 315 636
696 593 522 323 383 173 855 772 176 40 902 246 367 620 66 161 124
692 330 626 532 935 181 535 785 510 213 567 458 840 106 764 984
102 29 924 427 488 85 461 302 691 417 719 886 701 824 763 237 608
707 525 81 693 716 308 141 283 130 380 493 100 797 609 188 784 575
46 802 517 655 500 496 524 341 494 275 899 80 887 332 309 401 839
563 643 150 39 954 45 142 795 507 15 470 396 605 742 345 447 229
158 940 768 111 883 219 6 7 611 630 997 862 751 92 480 752 378 75
912 750 91 94 298 464 880 140 306 780 34 573 615 604 478 136 210
443 132 413 927 336 843 312 475 273 662 89
```

Funciones propias del proyecto

[PUSH_SWAP]

Push_swap.c *-> Funcion principal*

- main
- new_argc_count
- init_list

Check_error_free.c *-> Funciones que manejan errores y libera nodos*

- check_errors
- check_nbr_rep
- check_syntax
- check_limits_list
- free_list

Utils.c *-> Funciones de utilidad (listas)*

- list_newnode
- list_iter
- list_last
- list_size

Utils_extra.c *-> Funciones de utilidad general*

- ft_atol
- is_sorted
- to_index
- max_nbr
- min_nbr

Path_to_sort.c *-> Comprobacion/preparacion preordenacion*

- sort_3
- sort_4
- sort_5
- sort_algo

Algo.c *-> Algoritmo de ordenacion*

- radix_algo

Instrucciones

Push.c -> Empuja un nodo de una lista a otra lista

- push
- push_a
- push_b

Swap.c *-> Cambia la posicion de un nodo por la de otro*

- swap
- swap_a
- swap_b
- swap_ab

Rotate.c *-> Convierte el primer nodo en el ultimo, desplazando todo una posicion arriba*

- rotate
- rotate_a
- rotate_b
- rotate_ab

Reverse_rotate.c *-> Convierte el ultimo nodo en el primero, desplazando todo una posicion abajo*

- reverse_rotate
- reverse_rotate_a
- reverse_rotate_b
- reverse_rotate_ab

Consideraciones personales

Ha sido un proyecto muy interesante. He descubierto muchos conceptos nuevos, principalmente el Big O notation, aplicado para determinar la complejidad de un algoritmo. En este caso, de ordenación. He aprendido las distintas opciones que hay y las he investigado. Me he decidido por Radix para poder usar operaciones bitwise y he manejado los errores de forma bastante correcta.