Laboratorio - 8

Alumno: Jackson Fernando Merma Portocarrero

Escuela Profesional: Ingeniería de Sistemas

CUI: 20202143

Correo: jmermap@unsa.edu.pe

Generalidades

- Cada estructura tiene una carpeta donde se encuentran los problemas propuestos; se indicará en cada uno de ellos.
- Los problemas no tienen entrada de datos, pero sí casos de prueba.

Stack Problems

- Estos ejercicios se encuentran en la carpeta **'stack_problems'** del repositorio.

1. Evaluate Reverse Polish Notation

En este problema se usa la estructura Stack en el recorrido de las expresiones o datos a evaluar, esto funciona de la siguiente manera:

+ Si es un número, se agrega.
+ Si es una operación, se opera con los dos últimos números ingresados en el Stack y se almacena el nuevo resultado (se sacan 2 del Stack y se ingresa 1).
+ Finalmente, el último valor que queda en el Stack, es el resultado.

Salidas de casos implementados

Nombre del archivo → reverse_polish_notation.cpp

```
Test # 1: 2 1 + 3 *
Test # 2: 4 13 5 / +
Test # 3: 10 6 9 3 + -11 * / * 17 + 5 +
```

```
Salida:

Resultado 1: 9
Resultado 2: 6
Resultado 3: 22
```

2. Minimum Add to Make Parentheses

Su implementación requiere de un Stack, ya que al igual que el anterior problema, se necesita determinar cuántos paréntesis se pueden cerrar y así minimizar el problema; para ello se sigue la lógica:

+ Si es un paréntesis de apertura, se agrega.
+ Si es de cierre, si el último fue de apertura, se elimina a ese último, de lo contrario, también se agrega al paréntesis de cierre.
+ El resultado será la longitud del Stack, ya que almacenará todos los paréntesis que no pudieron hacer match durante el recorrido.

Salidas de casos implementados

Nombre del archivo → minimum_add_parentheses.cpp

```
Test # 1: ())
Test # 2: (((
Test # 3: ((()))(
Test # 4: ()))((
```

```
Salida:

caso 1: 1
caso 2: 3
caso 3: 2
caso 4: 4
```

Queue Problems

- Estos ejercicios se encuentran en la carpeta **'queue_problems'** del repositorio.

3. Caps Lock

Para este problema, se tomará al queue, como el buffer que es vaciado con $, la lógica funciona de la siguiente manera:

+ Si es un carácter, lo ingresa al Queue.
+ Si es un $, si 'arroba' (booleano) está encendido entonces cambia los caracteres (en esta implementación se está cambiando de mayúscula a minuscula y viceversa, si la entrada solo contiene minúsculas, entonces cambiarán a mayúsculas); finalmente, vacía todo el buffer concatenando el Queue a un string.
+ Si es @, entonces niega el 'arroba' (booleano) : !arroba.
+ La cadena que concatena cada vez que se vacía el buffer, será el resultado.

Salidas de casos implementados

Nombre del archivo → caps_lock.cpp

```
Test # 1: abc$d@ef$@g$
Test # 2: abc@def$@h$o@$l@$a@$
Test # 3: xyz@$xyz@$
```

```
Salida:

Cadena final: abcDEFg
Cadena final: ABCDEFhOlA
Cadena final: XYZxyz
```

Deque Problems

- Estos ejercicios se encuentran en la carpeta **'deque_problems'** del repositorio.

4. Backspace

En este problema, el uso del Deque es significativo en el algoritmo mientras se recorre la expresión, y en el resultado. Lógica:

+ Si es una letra, esta ingresa al final del deque.
+ Si es un #, entonces elimina (de existir) al último dato insertado en el deque.
+ Para la salida, el **deque**, se recorre como un queue (de inicio a final).

Salidas de casos implementados

Nombre del archivo → back_space.cpp

```
Test # 1: abc#de###f#ghi#jklmn#op#
Test # 2: abcd###
Test # 3: abcd#e#f#g#
```

```
Salida:

*test 1: abghjklmo
*test 2: a
*test 3: abc
```

5. Interview Wait

En este problema es evidente el uso de un deque, ya que se necesitará consultar al inicio y al final de una estructura para obtener la respuesta. Lógica:

+ Mientras el primero y el último dato del **deque**, no sea -1, entonces se elige al mínimo de ambos y se suma a un conteo (respuesta), y se elimina ese menor del **deque.**
+ Finalmente, se imprime la respuesta que será el conteo llevado.

Salidas de casos implementados

Nombre del archivo → interview_wait.cpp

```
Test # 1: {4,-1,5,2,3}
Test # 2: {1,-1,5,10,30,40}
Test # 3: {10,8,6,4,2,-1,3,5,7,9}
```

```
Salida:

salida: 9
salida: 1
salida: 24
```

Priority Queue Problems

- Estos ejercicios se encuentran en la carpeta **'priority_queue_problems'** del repositorio.

6. Merge k Sorted Lists

En este problema, se usa básicamente la implementación inherente del priority queue, para ingresar los datos. Lógica:

+ Simplemente se ingresan los datos al priority queue principal.
+ Se imprime esta estructura de inicio a fin.

Salidas de casos implementados

Nombre del archivo → merge_k_sort_lists.cpp

```
Test # 1: {{1,4,5},{1,3,4},{2,6}}
Test # 2: {{1,4,5,7,8,11,12},{3,4,5,6,7,11},{12,12,12,12}}
Test # 3: {{1,1,1,1,1},{2,2,2,2,2},{3,3,3,3,3}}
```

```
Salida:

*test 1: 1 1 2 3 4 4 5 6
*test 2: 1 3 4 4 5 5 6 7 7 8 11 11 12 12 12 12 12
*test 3: 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```