Reflexion 3.2

Codigo

```
#include <iostream>
#include "PriorityQueue/priority_queue.h"

/*
 Este programa utiliza un max_heap para implementar una priority queue
 El programa lee un archivo de entrada dado por el usuario y utiliza
 el input para crear la queue corrrespondiente.
 El programa puede realizar las funciones push, pop, top, empty y size
 y imprimir su resultado en pantalla.
*/

/*
 Complejidad temporal:
 - push: O(logN)
 - pop: O(logN)
 - top: O(1)
 - empty O(1)
 - size O(1)

 Autor: Yair Salvador Beltran Rios
 Matricula: A01254673
 Fecha: 12 de octubre de 2023
*/

/*
 Compilacion y ejecucion:
 $ find . -type f -name "*.cpp" | xargs g++ -o output
$ ./output
*/

int main()
{
    /*
    Crearemos el siguiente arbol binario como ejemplo
                71
              /    \
            31      14
           /  \    /  \
          13  20  7   11
```

```
        /  \
       12   7
    */

    priority_queue queue;

    queue.push(71);
    queue.push(7);
    queue.push(14);
    queue.push(11);
    queue.push(31);
    queue.push(13);
    queue.push(20);
    queue.push(7);
    queue.push(12);


    std::cout << "Queue incial: ";
    queue.print();
    std::cout << "-----------------------------------------" <<
std::endl;

    std::cout << "Prueba 1" << std::endl;
    std::cout << "Funcion: push" << std::endl;
    std::cout << "Input: " << 918 << std::endl;
    std::cout << "Queue antes: ";
    queue.print();
    queue.push(918);
    std::cout << "Queue despues: ";
    queue.print();
    std::cout << "-----------------------------------------" <<
std::endl;

    std::cout << "Prueba 2" << std::endl;
    std::cout << "Funcion: pop" << std::endl;
    std::cout << "Queue antes: ";
    queue.print();
    queue.pop();
    std::cout << "Queue despues: ";
    queue.print();
    std::cout << "-----------------------------------------" <<
std::endl;

    std::cout << "Prueba 3" << std::endl;
    std::cout << "Funcion: top" << std::endl;
    std::cout << "Queue: ";
```

```
    queue.print();
    std::cout << "Top: " << queue.top() << std::endl;
    std::cout << "-----------------------------------------" <<
std::endl;

    std::cout << "Prueba 4" << std::endl;
    std::cout << "Funcion: empty y size" << std::endl;
    for (int i=0;i < 3;i++){
        queue.pop();
    }
    std::cout << "Queue: ";
    queue.print();
    std::cout << "Size: " << queue.size() << std::endl;
    std::cout << std::boolalpha;
    std::cout << "Esta vacia?: " << queue.empty() << std::endl;
    std::cout << "-----------------------------------------" <<
std::endl;

    return 0;
}
```

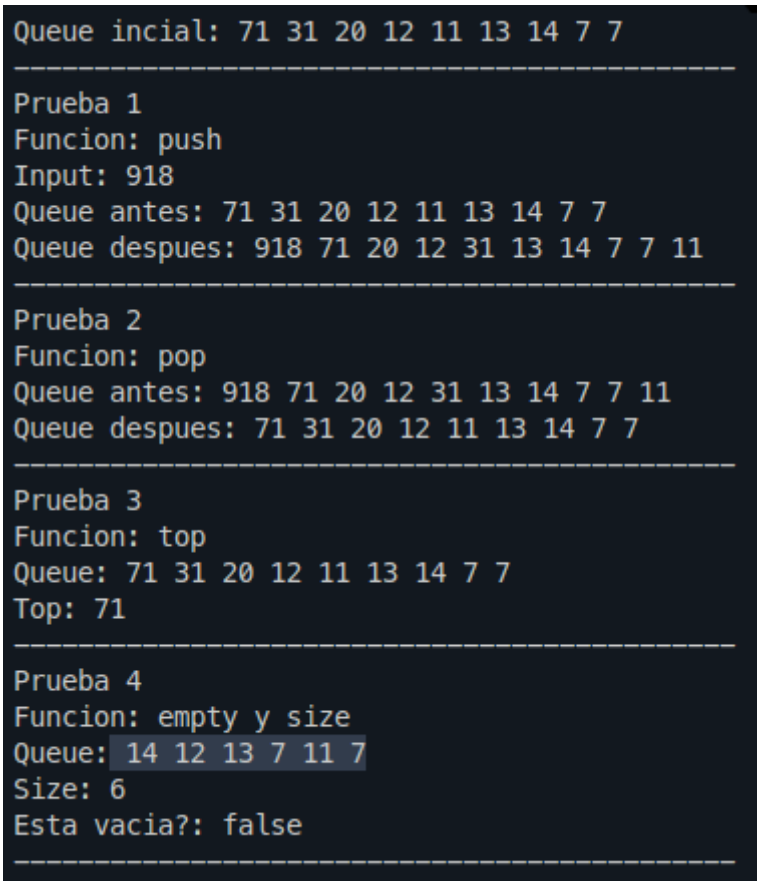
Casos de prueba

Queue incial antes de las pruebas:

```
71 31 20 12 11 13 14 7 7
```

Prueba	Funcion	Entrada	Queue antes	Resultado
1	push	918	71 31 20 12 11 13 14 7 7	918 71 20 12 31 13 14 7 7 11
2	pop		918 71 20 12 31 13 14 7 7 11	71 31 20 12 11 13 14 7 7
3	top		71 31 20 12 11 13 14 7 7	71
4	size y empty	pop x 3	14 12 13 7 11 7	6 y false

Output



Complejidad temporal

push: $O(\log n)$ - Agregamos un valor al array que conforma el heap, despues de esto la reestructuracion o "Heapificacion" toma lugar en la cual los diferentes nodos cambian de lugar dependiendo de su valor con el de mayor valor tomando siempre la raiz de nuestro arbol.

pop: $O(\log n)$ - Accedemos y retiramos el primer index de nuestro heap, lo reemplazamos con valor de ultimo index en nuestro heap y luego reestructuramos nuestro max_heap como mencionado en la funcion anterior

top: $O(1)$ - Imprimimos el primer valor de nuestro array

empty: $O(1)$ - Ya que el tamano de nuestro array se mantiene en una constante unica dentro de nuestra queue, la comparacion para comprobar si nuestra queue esta vacia es constante en todos los casos.

size: $O(1)$ - Ya que el tamano de nuestro array se mantiene en una constante unica, la complejidad para obtener este dato es constante en todos los casos.