César Alán Silva Ramos- A01252916

Act 2.2 - Verificación de las funcionalidades de una estructura de datos lineal

Repositorio en Github

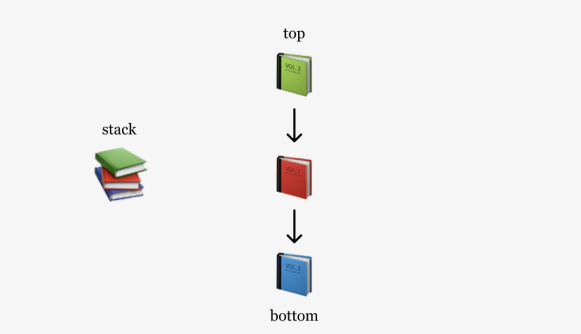
Introducción

Las pilas, también conocidas como "stacks", son una estructura de datos que sigue el principio de LIFO (Last In First Out), lo que significa que el elemento más recientemente agregado será el primero en ser retirado. Podemos visualizar una pila como una pila de libros apilados uno encima del otro. Esta estructura se puede implementar utilizando diferentes bases, como vectores o listas en programación.

Las operaciones en una pila siguen la filosofía LIFO. Estas operaciones incluyen:

- **push()**: Esta operación inserta un nuevo elemento en la pila, que se convierte en el último en ser retirado, similar a agregar un nuevo libro en la parte superior de la pila.
- **pop()**: La operación pop() elimina el elemento más recientemente agregado a la pila, equivalente a retirar el libro que se encuentra en la parte superior.
- **isFull()**: Determina si la pila ha alcanzado su capacidad máxima, es decir, si está llena o no.
- **Top()**: Devuelve el valor del último elemento agregado a la pila, es decir, el valor del libro en la parte superior.
- **isEmpty()**: Determina si la pila está vacía, es decir, si no contiene ningún elemento.

En este documento, exploraremos la creación de una pila utilizando listas anidadas así como implementación de las funciones mencionadas anteriormente en C++, incluyendo casos de prueba y resultados de ejecución. También analizaremos las diferentes complejidades temporales asociadas con estas operaciones en una pila.



Casos prueba

```
int main()
{
    // El stack se lee como si fuera una torre, el elemnto hasta arriba es el último elemento agregado
    // que el de hasta abajo fue el primer elemento agregado
    Stack stack(5);
    // push()
    stack.push(3);
    stack.push(420);
    stack.push(777);
    stack.push(666);
    stack.push(1000);
    // Top ()
    stack.Top();
    // isFull()
    stack.isFull();
    stack.printStack();
    // pop()
    stack.pop();
    stack.pop();
    stack.printStack();
    stack.pop();
    stack.pop();
    stack.pop();
    // isEmpty()
    stack.isEmpty();
}
```

En los casos de prueba iniciales, se utilizaron los valores 3, 420, 777, 666 y 1000, que se introdujeron en la pila mediante la función push(). Luego, se verificó el valor del último elemento agregado utilizando la función top(). Posteriormente, se comprobó si la pila estaba llena. Dado que se había definido previamente que la capacidad máxima de la pila es de 5 elementos, se llenó la pila para demostrar la efectividad de la función isFull().

A continuación, se procedió a eliminar algunos elementos utilizando la función pop() y se imprimió el estado de la pila. Esto se hizo para visualizar cómo los elementos se eliminan de manera secuencial, siguiendo la filosofía de LIFO (Last In First Out), es decir, los elementos más recientes son los primeros en ser eliminados.

Finalmente, se eliminaron todos los elementos de la pila utilizando pop() y se llamó a la función isEmpty() para verificar que la pila se vaciara correctamente. Esta acción permitió confirmar que la función detectó de manera adecuada la ausencia de elementos en la pila.

Estos casos de prueba se seleccionaron específicamente debido a su capacidad para ilustrar cómo se agregan y eliminan elementos siguiendo el principio LIFO, cómo se establece y verifica la

cantidad máxima de elementos en la pila, y cómo se ejecutan las operaciones de eliminación de elementos de manera secuencial desde la parte superior de la pila, mostrando la adición y eliminación de elementos más recientes.

Resultados de casos prueba

```
Se ha pusheado el elemento: 3
Se ha pusheado el elemento: 420
Se ha pusheado el elemento: 777
Se ha pusheado el elemento: 666
Se ha pusheado el elemento: 1000
El elemento hasta arriba del stack es: 1000
El stack esta lleno
El contenido del stack es:
1000
666
777
420
3
Se ha popeado el elemnto: 1000
Se ha popeado el elemnto: 666
El contenido del stack es:
777
420
3
Se ha popeado el elemnto: 777
Se ha popeado el elemnto: 420
Se ha popeado el elemnto: 3
El stack esta vacio
```

El stack se lee como una torre de arriba hacia abajo, donde los elementos más nuevos se encuentran hasta arriba mientras que los primeros que se insertaron en la parte inferior.

Complejidades temporales

- **push()**

```
void push (int value){
    Node * newNode = new Node(value); // Crear nuevo nodo a isnertar
    std::cout << "Se ha pusheado el elemento: " <<value <<std::endl;
    newNode->next = top; // El puntero del nuevo nodo siguinete es top
    top = newNode; // El tope ahora es el nuevo nodo
    size ++; // incrementanr la variable size cada vez que se cree un n
}
```

En esta función, se logra una complejidad temporal constante de O(1). Esto se debe a que las operaciones de creación de nuevos nodos y la reasignación de punteros son siempre constantes, sin depender del tamaño actual de la pila. Esta función no necesita recorrer la pila en ningún momento.

- **pop()**

```
void pop (){
    if (size == 0){
        std::cout << "El stack esta vacío, métele valores con push()" << std::endl;
        return;
    }
    Node* temp = top;
    top = top->next;
    std::cout << "Se ha popeado el elemnto: " << temp->data << std::endl;
    delete temp;
    size--;
}
```

La función pop(), de manera similar a push(), posee una complejidad temporal constante O(1). Esto se debe a que simplemente elimina el elemento que se encuentra en la parte superior de la pila, sin necesidad de iterar sobre el resto de la estructura. En otras palabras, esta función siempre realiza un número constante de operaciones y realiza la eliminación directamente en el elemento ubicado en la parte superior de la pila.

- **Top()**

```
void Top (){
    if (size == 0){
        std::cout << "El stack esta vacio." << std::endl;
        return;
    }

    std::cout << "El elemento hasta arriba del stack es: " << top->data <<std::endl;

}
```

La función Top() se caracteriza por tener una complejidad temporal constante. Esto se debe a que su principal operación consiste en acceder al dato del elemento que se encuentra en la parte superior de la pila. Esta acción siempre requiere la misma cantidad de tiempo, sin importar cuántos elementos haya en la pila. Además, la función realiza una verificación para determinar si la pila está vacía, y esta operación también es constante, ya que solamente se accede al contador del tamaño de la pila.

- **isFull()**

```
bool isFull (){
    if (max_size == 0){
        return false;
    } else if (size >= max_size){
        std::cout << "El stack esta lleno" << std::endl;
    }
    return size >= max_size;

}
```

La función isFull() cuenta con una complejidad temporal constante de O(1). Esta característica se debe a que las verificaciones que realiza se basan únicamente en los valores de los contadores: size, que representa el tamaño actual de la pila, y max_size, que es el valor máximo que se le ha asignado de manera arbitraria. Estas operaciones siempre requieren la misma cantidad de tiempo, sin importar cuántos elementos haya en la pila.

- **isEmpty()**

```
bool isEmpty (){
    if (size == 0){
        std::cout << "El stack esta vacio" << std::endl;
    } else if (size != 0){
        std::cout << "El stack no esta vacio :)" << std::endl;
    }
    return size == 0;
}
```

Por último, la función isEmpty() también presenta una complejidad temporal constante de O(1). En este caso, la función se basa únicamente en la variable que almacena el contador del tamaño actual de la pila. Si este contador es mayor que 0, significa que la pila no está vacía, por lo tanto, la función solo necesita comparar este valor con 0 para determinar si la pila está vacía o no, lo cual requiere un tiempo constante siempre.

Conclusión

En conclusión, la implementación de una pila mediante listas en C++ nos permite explorar y verificar las funcionalidades clave de esta estructura de datos lineal. Las operaciones como push(), pop(), Top(), isFull(), e isEmpty() son fundamentales para el funcionamiento de una pila y su aplicación en diversos contextos de programación.

Uno de los aspectos más destacados es que se logró que todas las operaciones tengan una complejidad temporal constante de O(1). Esto significa que su eficiencia y rendimiento se mantienen constantes sin importar la cantidad de elementos en la pila. Esto es esencial para garantizar que las operaciones de la pila sean eficientes en cualquier escenario sin importar la cantidad de datos a manipular.

La implementación exitosa y la verificación de estas funcionalidades en una pila proporcionan una base sólida para comprender y utilizar eficazmente esta estructura de datos lineal en aplicaciones de programación. El hecho de que todas las operaciones tienen una complejidad temporal constante es un indicativo de su eficacia en la gestión de datos en LIFO.