Tecnológico de Monterrey

TAREA 1 Documentación e implementación de algoritmos

Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 503)

Alumno:
Maximiliano Flores Moreno | A00836019

Profesor:
Elda Guadalupe Quiroga González

Fecha:
29 de Septiembre del 2025

Stack

- Definición: El **Stack** (Pila) es una estructura de datos lineal que sigue el principio (*Last In, First Out* — Último en Entrar, Primero en Salir). El elemento más reciente en ser añadido es siempre el primero en ser removido. Las dos operaciones principales ocurren en el mismo extremo, llamado la **cima** (*Top*):
    - **Push**: Añade un elemento a la cima.
    - **Pop**: Remueve el elemento de la cima.

    Librería Utilizada

    Se utilizó el adaptador de contenedor **std::stack** de la **Standard Template Library (STL)** de C++. Esta clase no implementa la estructura desde cero, sino que la envuelve, utilizando por defecto un contenedor subyacente (**std::deque**) para manejar los datos, garantizando operaciones eficientes de inserción y eliminación en un extremo.

- Casos de prueba Stack

```
C/C++
int main() {
    cout << "--- STACK (LIFO) - Casos de Prueba ---" << endl;

    // Caso de Prueba 1: Operacion LIFO Estandar
    STACK<int> pila1;
    pila1.push(10);
    pila1.push(20);
    pila1.push(30); // 30 es el Top
    cout << "Caso 1: Push(10, 20, 30). Top esperado: 30" << endl;
    cout << "   Resultado top(): " << pila1.top() << endl; // Esperado: 30
    pila1.pop(); // Elimina 30
    cout << "   Pop(). Nuevo top esperado: 20" << endl;
    cout << "   Resultado top() después de pop: " << pila1.top() << endl; //
Esperado: 20
    cout << "----------------------------------------" << endl;

    // Caso de Prueba 2: Verificacion de Estado Vacio
    STACK<string> pila2;
    cout << "Caso 2: Stack inicial vacio." << endl;
    cout << "   Resultado empty() inicial: " << (pila2.empty() ? "Vacio
(Correcto)" : "No Vacio") << endl; // Esperado: Vacio
    pila2.push("ItemA");
    cout << "   Push(\"ItemA\"). Resultado empty(): " << (pila2.empty() ?
"Vacio" : "No Vacio (Correcto)") << endl; // Esperado: No Vacio
    cout << "----------------------------------------" << endl;
```

```
    // Caso de Prueba 3: Deplecion y Actualizacion de Tamanio
    STACK<char> pila3;
    pila3.push('X');
    pila3.push('Y');
    cout << "Caso 3: Deplecion de Stack." << endl;
    cout << "   Tamanio inicial (esperado 2): " << pila3.size() << endl; //
Esperado: 2
    pila3.pop(); // Queda X
    cout << "   Pop(). Tamanio (esperado 1): " << pila3.size() << endl; //
Esperado: 1
    pila3.pop(); // Queda vacio
    cout << "   Pop(). Tamanio final (esperado 0): " << pila3.size() <<
endl; // Esperado: 0
    cout << "   Resultado empty() final: " << (pila3.empty() ? "Vacio
(Correcto)" : "No Vacio") << endl;

    return 0;
}
```

- Justificación:
    - Operación LIFO Estándar: Válida que el principio LIFO se respete. Es la prueba fundamental que confirma que el elemento removido por pop() es, de hecho, el último insertado, y que top() apunta correctamente hacia el elemento más nuevo.

    - Verificación de Estado Vacío: Asegura que nuestro método empty() funciona correctamente con el estado inicial de la pila (vacía) y después de añadir un elemento (no vacía), permitiendo el manejo básico del flujo de control de la estructura.

    - Depleción y Actualización de Tamaño: Prueba la condición límite de vaciado completo. Confirma que las operaciones de la libreria push y pop actualizan con precisión el contador interno (size()) y que la estructura vuelve a un estado empty() verificable después de remover todos sus elementos.
-
- Complejidad algorítmica: O(1) utilizando las funciones pop, push, top , empty size dado que son estructuras adyacente que añaden o remueven el final de tiempo costante

Queque

- Definición: La Queue (Cola) es una estructura de datos lineal que sigue el principio (*First In, First Out* — Primero en Entrar, Primero en Salir). Esto imita el comportamiento de una cola del mundo real: los elementos se añaden por un extremo (*Back*) y se remueven por el extremo opuesto (*Front*).
    - **Push (o Enqueue)**: Añade un elemento al final (BACK).

- **Pop (o Dequeue)**: Remueve un elemento del frente (FRONT).

Librería Utilizada

Al igual que el Stack, se utilizó el adaptador de contenedor **std::queue** de la **STL**. Esta clase también utiliza por defecto **std::deque** como un contenedor subyacente para realizar operaciones rápidas tanto en el front como en back.

- Casos de prueba Queque

```
C/C++
int main() {
    cout << "--- QUEUE (FIFO) - Casos de Prueba ---" << std::endl;

    // Caso de Prueba 1: Operacion FIFO Estandar
    QUEUE<int> cola1;
    cola1.push(100); // Frontal
    cola1.push(200);
    cola1.push(300); // Posterior
    cout << "Caso 1: Push(100, 200, 300)." << endl;

    cout << "   Elemento frontal (esperado 100): " << cola1.front() << endl;

    cola1.pop(); // Elimina el 100

    cout << "   Pop(). Nuevo frontal (esperado 200): " << cola1.front() <<
endl;

    cout << "----------------------------------------" << endl;

    // Caso de Prueba 2: Comprobacion de Ambos Extremos (front y back)
    QUEUE<string> cola2;
    cola2.push("A"); // Frontal
    cola2.push("B");
    cola2.push("C"); // Posterior
    cout << "Caso 2: Comprobacion de front/back." << endl;

    cout << "   Frontal (esperado A): " << cola2.front() << endl;

    cout << "   Posterior (esperado C): " << cola2.back() << endl;
    cola2.pop(); // Elimina A
    cola2.push("D"); // Nuevo Posterior
    cout << "   Pop() y Push(\"D\"). Nuevo frontal (esperado B): " <<
cola2.front() << endl;
    cout << "   Nuevo posterior (esperado D): " << cola2.back() << endl;
    cout << "----------------------------------------" << endl;
```

```
    // Caso de Prueba 3: Deplecion Total de la Cola
    QUEUE<double> cola3;
    cola3.push(1.1);
    cola3.push(2.2);
    cout << "Caso 3: Deplecion total y size()." << endl;
    cout << "   Tamanio inicial (esperado 2): " << cola3.size() <<endl;
    cola3.pop();
    cola3.pop();
    cout << "   Dos pops. Tamanio final (esperado 0): " << cola3.size() <<
endl;
    cout << "   Resultado empty() final: " << (cola3.empty() ? "Vacio
(Correcto)" : "No Vacio") << endl;

    return 0;
}
```

- Justificación:
    - Operación FIFO Estándar: Validando el principio FIFO. Asegura que el elemento removido por pop() es el que tiene más tiempo en la cola (el primero que se insertó), y que front() siempre apunta a este elemento.

    - Comprobación de Ambos extremos (front y back):Para la Queque, se mantienen dos punteros activos. Asegura que la inserción (push) manipula correctamente el back (el elemento más nuevo) y la remoción manipula el front (el elemento más antiguo), garantizando la completitud de la estructura.

    - Depleción Total de la Cola: Prueba la condición límite de vaciado total. Verifica que el estado y tamaño de la cola se actualizan correctamente hasta llegar a 0 y al estado empty(), sin causar errores al procesar el último elemento.
- Complejidad algoritmica: O(1) ya que se añaden y eliminan elementos en extremos opuestos en tiempo constante

Diccionario/Hash

- Definición: Esta estructura de datos nos enfocamos en almacenar valores mediante pares los cuales son (Key-Value) lo que permite recuperar valores rápidamente a partir de su llave, adicionalmente se utiliza una función **Hash** para mapear esta llave en una posición (index o bucket) en un arreglo lo cual nos da un acceso directo a esta.

Se utilizó el contenedor asociativo **std::unordered_map** de la **STL**. Este es el contenedor ideal para implementar una Tabla Hash, ya que su funcionalidad principal se basa en el hashing de la llave y ofrece una complejidad promedio de O(1) para las operaciones de búsqueda.

- Casos de prueba Diccionario/Table

```
C/C++
int main() {
    cout << "--- DICTIONARY/HASH (Key-Value) - Casos de Prueba ---" << endl;
    DICTIONARY<string, int> diccionario;

    // Caso de Prueba 1: Insercion y Modificacion
    cout << "Caso 1: Insercion y Modificacion (Key-Value)." << endl;
    diccionario["ClaveA"] = 10; // Insercion
    diccionario.insert({"ClaveB", 20}); // Otra insercion
    cout << "   Valor inicial de ClaveA: " << diccionario["ClaveA"] << endl;
    diccionario["ClaveA"] = 15; // Modificar
    cout << "   Valor modificado de ClaveA: " << diccionario["ClaveA"] <<
endl;
    cout << "----------------------------------------" << endl;

    // Caso de Prueba 2: Eliminacion y Verificacion de Existencia (count)
    cout << "Caso 2: Eliminacion y count()." << endl;
    diccionario["ClaveC"] = 30;
    cout << "   ClaveC existe (esperado 1): " << diccionario.count("ClaveC")
<< endl;
    diccionario.erase("ClaveC");
    cout << "   Se ejecuta Borrar(\"ClaveC\")." << endl;
    cout << "   ClaveC existe ahora (esperado 0): " <<
diccionario.count("ClaveC") << endl;
    cout << "   Tamanio (esperado 2): " << diccionario.size() << endl; //
Esperado: 2 (ClaveA y ClaveB)
    cout << "----------------------------------------" << endl;

    // Caso de Prueba 3: Caso Borde de Busqueda de Llave Inexistente
(at/count)
    cout << "Caso 3: Busqueda de Clave Inexistente." << endl;
    string clave_inexistente = "ClaveZ";

    // a) Uso de count() (Método seguro)
    cout << "   Resultado count(\"ClaveZ\"): " <<
diccionario.count(clave_inexistente) << endl;

    // b) Uso de at() (Prueba de robustez ante error)
    try {
        diccionario.at(clave_inexistente);
    } catch (const out_of_range& e) {
```

```
        cout << "   Intento de at(\"ClaveZ\") resulto en: Capturado (no
existe la clave)." << endl;
    }

    return 0;
}
```

- Justificación:
    - Inserción y Modificación: Es la validación de dos acciones más comunes: crear un nuevo par llave-valor y cambiar el valor asociado a una llave preexistente. Confirma que la estructura puede gestionar dinámicamente sus datos.

    - Eliminación y Verificación de Existencia (count): Prueba la capacidad de la estructura para remover llaves sin afectar el resto de la tabla con el método (erase()). Además, utiliza el método seguro count() para verificar con certeza la presencia de una llave antes y después de la eliminación.

    - Búsqueda de Clave Inexistente (at/count): Este es el caso límite donde se aplica el manejo de errores:

        - count(): Muestra el comportamiento seguro al retornar 0 para una llave no existente.

        - at(): Pone a prueba la robustez de la estructura, ya que, por diseño, al intentar acceder a una llave no existente, debe lanzar una excepción **(std::out_of_range)**, en lugar de retornar un valor indefinido o insertar una llave por defecto o salga nulo.
- Complejidad Algorítmica: Dado que es un programa hash medimos su eficiencia por promedio lo cual permite una buena distribución utilizando este algoritmo
    - Inserción, Búsqueda, Acceso, Eliminación ([], at, insert, erase, count): O(1) (Tiempo Constante en promedio).

    - Peor Caso (Colisiones): O(N) (Tiempo Lineal, en el raro caso de que todas las llaves se lleguen mapear hacia un mismo índice).

    - Clear: O(N).

Uso de AI

Para esta tarea solo se utilizó la IA para la probar distintos casos de prueba el cual pedí unos 10 alrededor por estructura de datos y me base solamente en 3 por programa para probar las funcionalidades de la librería STL que ya incluye los métodos esenciales para la estructura de datos.