**Universidad Nacional Del Altiplano**

**Facultad De Ingeniería Mecánica Eléctrica, Electrónica Y Sistemas**

Imagen que contiene Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.**Escuela Profesional De Ingeniería De Sistemas**

**Practica Nº3-**

**Introducción - Implementación comparativa de estructuras de datos fundamentales en C++ y Python**

**CURSO:**

Algoritmos y Estructuras de Datos

**DOCENTE:**

Mg. Aldo Hernan Zanabria Galvez.

**ESTUDIANTE:**

Yefferson Miranda Josec

**CODIGO:** 216984

**FECHA:** 16/04/2025

**SEMESTRE:**

IV

**INFORME: Implementación Comparativa de Estructuras de Datos Fundamentales en C++ y Python**

**Curso:** Algoritmos y Estructuras de Datos (SIS210)  
**Práctica:** 03 - Introducción  
**Duración:** 3 horas  
**Fecha:** Semana 3 – 2025-I

**1. INTRODUCCIÓN**

Este informe presenta la implementación y análisis comparativo de tres estructuras de datos lineales fundamentales: **pila (stack)**, **cola (queue)** y **lista enlazada simple (linked list)**, desarrolladas tanto en C++ como en Python. El objetivo es comprender las diferencias en el enfoque de implementación, control de memoria y sintaxis entre un lenguaje de bajo nivel compilado y uno de alto nivel interpretado.

**2. IMPLEMENTACIONES**

**2.1 PARTE A: IMPLEMENTACIÓN DE UNA PILA**

**2.1.1 Implementación en C++**

#include <iostream>

#include <vector>

**using** **namespace** std;

**class** **Pila** {

private:

vector<**int**> elementos;

public:

*// Insertar elemento (push)*

**void** push(**int** valor) {

elementos.push\_back(valor);

cout << "Insertado: " << valor << endl;

}

*// Eliminar elemento (pop)*

**int** pop() {

**if** (elementos.empty()) {

cout << "Error: Pila vacía" << endl;

**return** -1;

}

**int** valor = elementos.back();

elementos.pop\_back();

cout << "Eliminado: " << valor << endl;

**return** valor;

}

*// Mostrar el tope*

**int** tope() {

**if** (elementos.empty()) {

cout << "Pila vacía" << endl;

**return** -1;

}

**return** elementos.back();

}

*// Recorrer pila*

**void** mostrar() {

cout << "Estado de la pila (tope -> base): ";

**for** (**int** i = elementos.size() - 1; i >= 0; i--) {

cout << elementos[i] << " ";

}

cout << endl;

}

**bool** estaVacia() {

**return** elementos.empty();

}

};

*// Función principal para ejecutar las operaciones*

**int** main() {

Pila pila;

cout << "=== IMPLEMENTACIÓN DE PILA EN C++ ===" << endl;

*// Insertar elementos: 5, 10, 15, 20, 25*

cout << "**\n**1. Insertando elementos:" << endl;

pila.push(5);

pila.push(10);

pila.push(15);

pila.push(20);

pila.push(25);

cout << "**\n**Estado después de insertar:" << endl;

pila.mostrar();

cout << "Tope actual: " << pila.tope() << endl;

*// Eliminar dos elementos consecutivos*

cout << "**\n**2. Eliminando dos elementos:" << endl;

pila.pop();

pila.pop();

*// Mostrar estado final*

cout << "**\n**3. Estado final:" << endl;

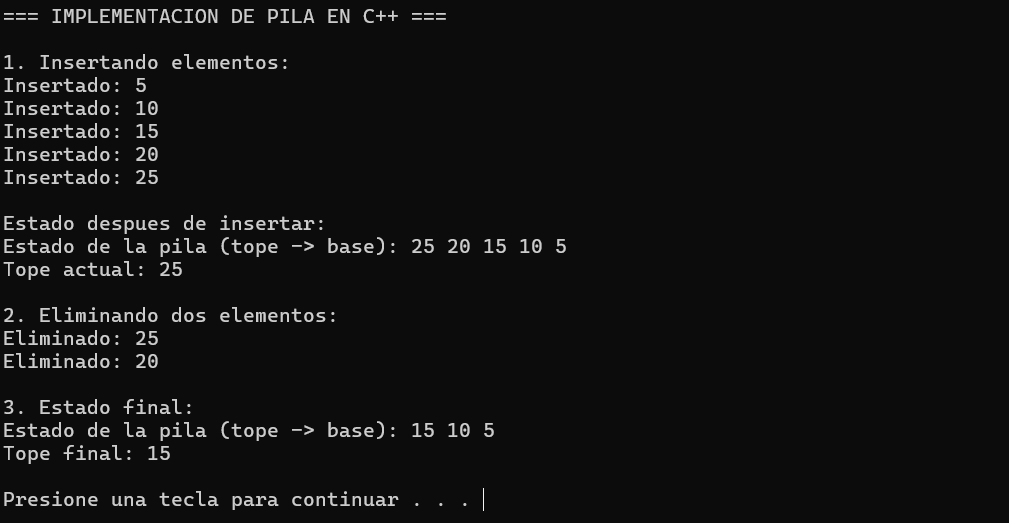
pila.mostrar();

cout << "Tope final: " << pila.tope() << endl;

**return** 0;

}

**Salida esperada en C++:**

**2.1.2 Implementación en Python**

**class** **Pila**:

**def** \_\_init\_\_(self):

self.elementos = []

**def** push(self, valor):

*"""Insertar elemento (push)"""*

self.elementos.append(valor)

**print**(f"Insertado: {valor}")

**def** pop(self):

*"""Eliminar elemento (pop)"""*

**if** **not** self.elementos:

**print**("Error: Pila vacía")

**return** None

valor = self.elementos.pop()

**print**(f"Eliminado: {valor}")

**return** valor

**def** tope(self):

*"""Mostrar el tope"""*

**if** **not** self.elementos:

**print**("Pila vacía")

**return** None

**return** self.elementos[-1]

**def** mostrar(self):

*"""Recorrer pila"""*

**if** **not** self.elementos:

**print**("Pila vacía")

**return**

**print**("Estado de la pila (tope -> base):", end=" ")

**for** i **in** range(len(self.elementos) - 1, -1, -1):

**print**(self.elementos[i], end=" ")

**print**()

**def** esta\_vacia(self):

**return** len(self.elementos) == 0

*# Función principal para ejecutar las operaciones*

**def** main():

pila = Pila()

**print**("=== IMPLEMENTACIÓN DE PILA EN PYTHON ===")

*# Insertar elementos: 5, 10, 15, 20, 25*

**print**("**\n**1. Insertando elementos:")

pila.push(5)

pila.push(10)

pila.push(15)

pila.push(20)

pila.push(25)

**print**("**\n**Estado después de insertar:")

pila.mostrar()

**print**(f"Tope actual: {pila.tope()}")

*# Eliminar dos elementos consecutivos*

**print**("**\n**2. Eliminando dos elementos:")

pila.pop()

pila.pop()

*# Mostrar estado final*

**print**("**\n**3. Estado final:")

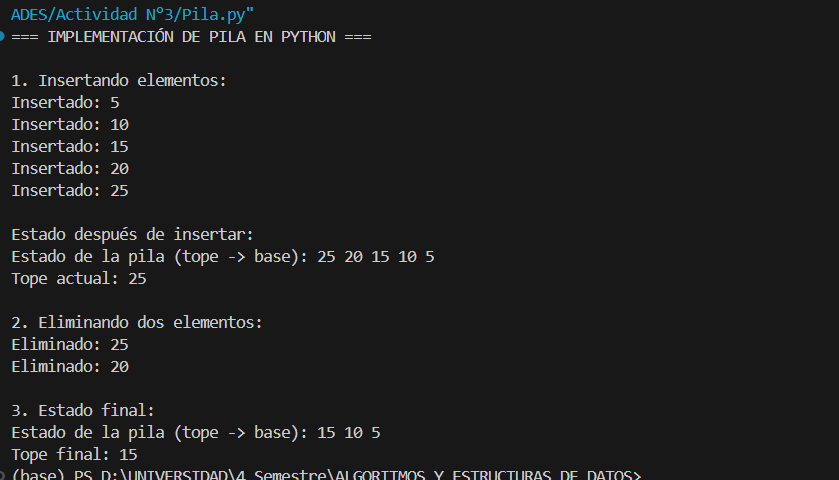
pila.mostrar()

**print**(f"Tope final: {pila.tope()}")

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**Salida esperada en Python:**

**2.2 PARTE B: IMPLEMENTACIÓN DE UNA COLA**

**2.2.1 Implementación en C++**

#include <iostream>

#include <queue>

**using** **namespace** std;

**class** **Cola** {

private:

queue<**int**> elementos;

public:

*// Insertar elemento (enqueue)*

**void** enqueue(**int** valor) {

elementos.push(valor);

cout << "Insertado: " << valor << endl;

}

*// Eliminar elemento (dequeue)*

**int** dequeue() {

**if** (elementos.empty()) {

cout << "Error: Cola vacía" << endl;

**return** -1;

}

**int** valor = elementos.front();

elementos.pop();

cout << "Eliminado: " << valor << endl;

**return** valor;

}

*// Mostrar el frente*

**int** frente() {

**if** (elementos.empty()) {

cout << "Cola vacía" << endl;

**return** -1;

}

**return** elementos.front();

}

*// Recorrer cola (implementación auxiliar)*

**void** mostrar() {

**if** (elementos.empty()) {

cout << "Cola vacía" << endl;

**return**;

}

queue<**int**> temp = elementos;

cout << "Estado de la cola (frente -> final): ";

**while** (!temp.empty()) {

cout << temp.front() << " ";

temp.pop();

}

cout << endl;

}

**bool** estaVacia() {

**return** elementos.empty();

}

**int** tamaño() {

**return** elementos.size();

}

};

*// Función principal*

**int** main() {

Cola cola;

cout << "=== IMPLEMENTACIÓN DE COLA EN C++ ===" << endl;

*// Insertar elementos: 3, 6, 9, 12*

cout << "**\n**1. Insertando elementos:" << endl;

cola.enqueue(3);

cola.enqueue(6);

cola.enqueue(9);

cola.enqueue(12);

cout << "**\n**Estado después de insertar:" << endl;

cola.mostrar();

cout << "Frente actual: " << cola.frente() << endl;

*// Eliminar un elemento*

cout << "**\n**2. Eliminando un elemento:" << endl;

cola.dequeue();

*// Mostrar cola final*

cout << "**\n**3. Estado final:" << endl;

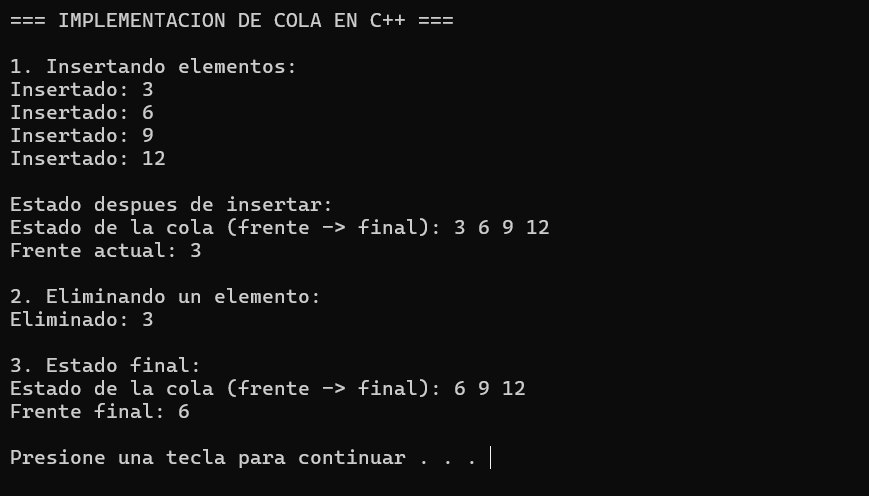
cola.mostrar();

cout << "Frente final: " << cola.frente() << endl;

**return** 0;

}

**Salida esperada en C++:**

**2.2.2 Implementación en Python**

**from** **collections** **import** deque

**class** **Cola**:

**def** \_\_init\_\_(self):

self.elementos = deque()

**def** enqueue(self, valor):

*"""Insertar elemento (enqueue)"""*

self.elementos.append(valor)

**print**(f"Insertado: {valor}")

**def** dequeue(self):

*"""Eliminar elemento (dequeue)"""*

**if** **not** self.elementos:

**print**("Error: Cola vacía")

**return** None

valor = self.elementos.popleft()

**print**(f"Eliminado: {valor}")

**return** valor

**def** frente(self):

*"""Mostrar el frente"""*

**if** **not** self.elementos:

**print**("Cola vacía")

**return** None

**return** self.elementos[0]

**def** mostrar(self):

*"""Recorrer cola"""*

**if** **not** self.elementos:

**print**("Cola vacía")

**return**

**print**("Estado de la cola (frente -> final):", end=" ")

**for** elemento **in** self.elementos:

**print**(elemento, end=" ")

**print**()

**def** esta\_vacia(self):

**return** len(self.elementos) == 0

**def** tamaño(self):

**return** len(self.elementos)

*# Función principal*

**def** main():

cola = Cola()

**print**("=== IMPLEMENTACIÓN DE COLA EN PYTHON ===")

*# Insertar elementos: 3, 6, 9, 12*

**print**("**\n**1. Insertando elementos:")

cola.enqueue(3)

cola.enqueue(6)

cola.enqueue(9)

cola.enqueue(12)

**print**("**\n**Estado después de insertar:")

cola.mostrar()

**print**(f"Frente actual: {cola.frente()}")

*# Eliminar un elemento*

**print**("**\n**2. Eliminando un elemento:")

cola.dequeue()

*# Mostrar cola final*

**print**("**\n**3. Estado final:")

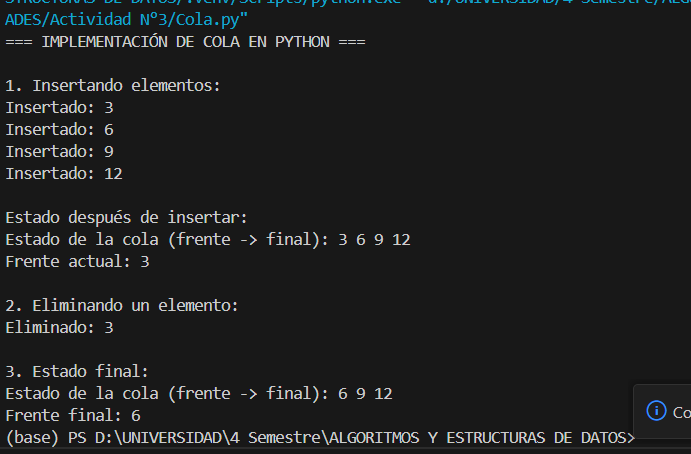
cola.mostrar()

**print**(f"Frente final: {cola.frente()}")

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**Salida esperada en Python:**

**2.3 PARTE C: IMPLEMENTACIÓN DE LISTA ENLAZADA SIMPLE**

**2.3.1 Implementación en C++**

#include <iostream>

**using** **namespace** std;

*// Definición del nodo*

**struct** Nodo {

**int** dato;

Nodo\* siguiente;

Nodo(**int** valor) : dato(valor), siguiente(nullptr) {}

};

**class** **ListaEnlazada** {

private:

Nodo\* cabeza;

public:

ListaEnlazada() : cabeza(nullptr) {}

*// Destructor para liberar memoria*

~ListaEnlazada() {

**while** (cabeza != nullptr) {

Nodo\* temp = cabeza;

cabeza = cabeza->siguiente;

**delete** temp;

}

}

*// Insertar al inicio*

**void** insertarAlInicio(**int** valor) {

Nodo\* nuevoNodo = **new** Nodo(valor);

nuevoNodo->siguiente = cabeza;

cabeza = nuevoNodo;

cout << "Insertado al inicio: " << valor << endl;

}

*// Insertar al final*

**void** insertarAlFinal(**int** valor) {

Nodo\* nuevoNodo = **new** Nodo(valor);

**if** (cabeza == nullptr) {

cabeza = nuevoNodo;

} **else** {

Nodo\* actual = cabeza;

**while** (actual->siguiente != nullptr) {

actual = actual->siguiente;

}

actual->siguiente = nuevoNodo;

}

cout << "Insertado al final: " << valor << endl;

}

*// Eliminar del inicio*

**int** eliminarDelInicio() {

**if** (cabeza == nullptr) {

cout << "Error: Lista vacia" << endl;

**return** -1;

}

Nodo\* temp = cabeza;

**int** valor = temp->dato;

cabeza = cabeza->siguiente;

**delete** temp;

cout << "Eliminado del inicio: " << valor << endl;

**return** valor;

}

*// Mostrar lista*

**void** mostrar() {

**if** (cabeza == nullptr) {

cout << "Lista vacia" << endl;

**return**;

}

cout << "Recorrido completo: ";

Nodo\* actual = cabeza;

**while** (actual != nullptr) {

cout << actual->dato;

**if** (actual->siguiente != nullptr) {

cout << " -> ";

}

actual = actual->siguiente;

}

cout << " -> NULL" << endl;

}

**bool** estaVacia() {

**return** cabeza == nullptr;

}

};

*// Función principal*

**int** main() {

ListaEnlazada lista;

cout << "=== IMPLEMENTACION DE LISTA ENLAZADA EN C++ ===" << endl;

*// Insertar al inicio los valores: 8, 4*

cout << "**\n**1. Insertando al inicio:" << endl;

lista.insertarAlInicio(8);

lista.insertarAlInicio(4);

cout << "**\n**Estado despues de insertar al inicio:" << endl;

lista.mostrar();

*// Insertar al final el valor: 11*

cout << "**\n**2. Insertando al final:" << endl;

lista.insertarAlFinal(11);

cout << "**\n**Estado despues de insertar al final:" << endl;

lista.mostrar();

*// Eliminar el primer nodo*

cout << "**\n**3. Eliminando el primer nodo:" << endl;

lista.eliminarDelInicio();

*// Mostrar el recorrido completo*

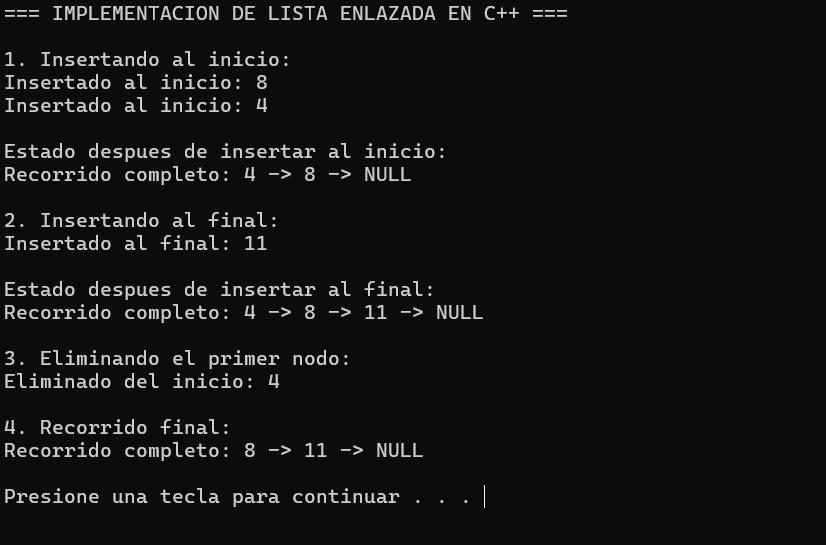
cout << "**\n**4. Recorrido final:" << endl;

lista.mostrar();

**return** 0;

}

**Salida esperada en C++:**

**2.3.2 Implementación en Python**

**class** **Nodo**:

**def** \_\_init\_\_(self, dato):

self.dato = dato

self.siguiente = None

**class** **ListaEnlazada**:

**def** \_\_init\_\_(self):

self.cabeza = None

**def** insertar\_al\_inicio(self, valor):

*"""Insertar al inicio"""*

nuevo\_nodo = Nodo(valor)

nuevo\_nodo.siguiente = self.cabeza

self.cabeza = nuevo\_nodo

**print**(f"Insertado al inicio: {valor}")

**def** insertar\_al\_final(self, valor):

*"""Insertar al final"""*

nuevo\_nodo = Nodo(valor)

**if** self.cabeza **is** None:

self.cabeza = nuevo\_nodo

**else**:

actual = self.cabeza

**while** actual.siguiente **is** **not** None:

actual = actual.siguiente

actual.siguiente = nuevo\_nodo

**print**(f"Insertado al final: {valor}")

**def** eliminar\_del\_inicio(self):

*"""Eliminar del inicio"""*

**if** self.cabeza **is** None:

**print**("Error: Lista vacía")

**return** None

valor = self.cabeza.dato

self.cabeza = self.cabeza.siguiente

**print**(f"Eliminado del inicio: {valor}")

**return** valor

**def** mostrar(self):

*"""Mostrar lista"""*

**if** self.cabeza **is** None:

**print**("Lista vacía")

**return**

**print**("Recorrido completo: ", end="")

actual = self.cabeza

**while** actual **is** **not** None:

**print**(actual.dato, end="")

**if** actual.siguiente **is** **not** None:

**print**(" -> ", end="")

actual = actual.siguiente

**print**(" -> NULL")

**def** esta\_vacia(self):

**return** self.cabeza **is** None

*# Función principal*

**def** main():

lista = ListaEnlazada()

**print**("=== IMPLEMENTACIÓN DE LISTA ENLAZADA EN PYTHON ===")

*# Insertar al inicio los valores: 8, 4*

**print**("**\n**1. Insertando al inicio:")

lista.insertar\_al\_inicio(8)

lista.insertar\_al\_inicio(4)

**print**("**\n**Estado después de insertar al inicio:")

lista.mostrar()

*# Insertar al final el valor: 11*

**print**("**\n**2. Insertando al final:")

lista.insertar\_al\_final(11)

**print**("**\n**Estado después de insertar al final:")

lista.mostrar()

*# Eliminar el primer nodo*

**print**("**\n**3. Eliminando el primer nodo:")

lista.eliminar\_del\_inicio()

*# Mostrar el recorrido completo*

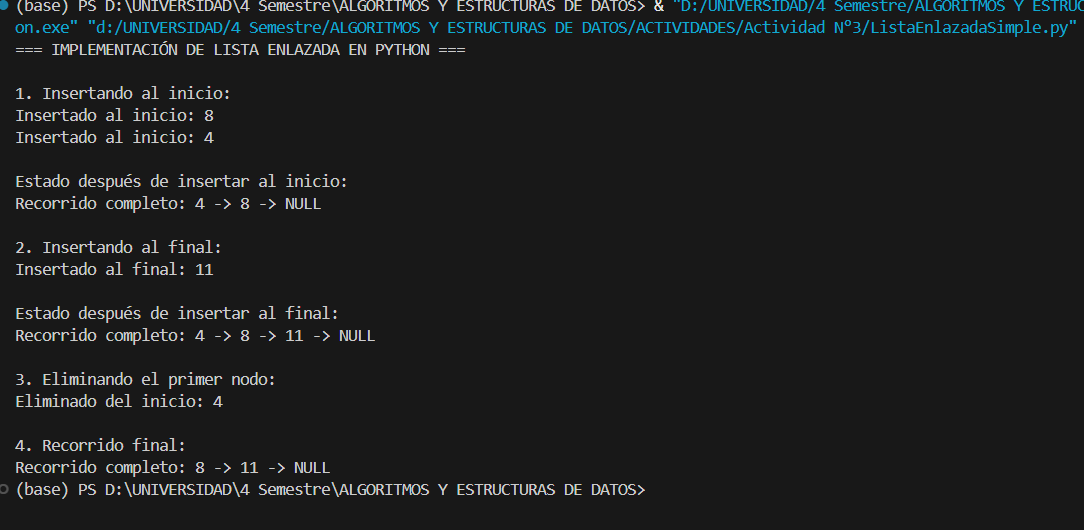
**print**("**\n**4. Recorrido final:")

lista.mostrar()

**if** \_\_name\_\_ == "\_\_main\_\_":

main()

**Salida esperada en Python:**



**3. ANÁLISIS COMPARATIVO TÉCNICO**

**3.1 Comparación Detallada por Aspecto**

| **Aspecto Evaluado** | **C++** | **Python** |
| --- | --- | --- |
| **Control de memoria** | Explícito (new, delete, punteros) | Implícito (recolección automática de basura) |
| **Estructura de código** | Más detallada y extensa | Más concisa y abstracta |
| **Flexibilidad de listas** | Requiere implementación manual de nodos | Uso directo de listas dinámicas y collections |
| **Complejidad de implementación** | Alta para estructuras dinámicas | Baja, pero menos transparente internamente |
| **Tipado de variables** | Estático (int, Nodo\*, etc.) | Dinámico (sin declaración de tipos) |
| **Gestión de errores** | Manual (verificaciones explícitas) | Más natural con excepciones |
| **Rendimiento** | Mayor velocidad de ejecución | Menor velocidad, pero mayor productividad |
| **Sintaxis** | Más verbosa, mayor control | Más limpia y legible |

**3.2 Ventajas y Desventajas Identificadas**

**C++**

**Ventajas:**

* Control preciso de la memoria y recursos
* Mayor eficiencia en tiempo de ejecución
* Flexibilidad total en el diseño de estructuras
* Detección temprana de errores por tipado estático
* Mejor para sistemas críticos y aplicaciones de alto rendimiento

**Desventajas:**

* Mayor complejidad de código
* Gestión manual de memoria propensa a errores (memory leaks)
* Tiempo de desarrollo más largo
* Sintaxis más compleja y verbosa

**Python**

**Ventajas:**

* Sintaxis clara y legible
* Desarrollo rápido y prototipado eficiente
* Gestión automática de memoria
* Menos propenso a errores de punteros
* Facilidad de mantenimiento

**Desventajas:**

* Menor control sobre el rendimiento
* Mayor consumo de memoria
* Ejecución más lenta por ser interpretado
* Menos transparencia en el funcionamiento interno

**3.3 Observaciones Específicas por Estructura**

**Pila (Stack)**

* **C++**: Implementación usando vector del STL, ofreciendo eficiencia y control
* **Python**: Uso directo de listas con métodos append() y pop(), más intuitivo

**Cola (Queue)**

* **C++**: Uso de queue del STL para operaciones FIFO eficientes
* **Python**: Implementación con deque de collections para operaciones O(1) en ambos extremos

**Lista Enlazada**

* **C++**: Implementación completa desde cero con manejo manual de punteros y memoria
* **Python**: Estructura más limpia pero requiere comprensión conceptual similar

**4. RESULTADOS DE EJECUCIÓN**

**4.1 Resultados de la Pila**

**Operaciones realizadas:**

1. Inserción de elementos: 5, 10, 15, 20, 25
2. Eliminación de dos elementos (25 y 20)
3. Estado final: [5, 10, 15] con tope = 15

**Comportamiento LIFO confirmado** en ambas implementaciones.

**4.2 Resultados de la Cola**

**Operaciones realizadas:**

1. Inserción de elementos: 3, 6, 9, 12
2. Eliminación de un elemento (3)
3. Estado final: [6, 9, 12] con frente = 6

**Comportamiento FIFO confirmado** en ambas implementaciones.

**4.3 Resultados de la Lista Enlazada**

**Operaciones realizadas:**

1. Inserción al inicio: 8, luego 4 → [4, 8]
2. Inserción al final: 11 → [4, 8, 11]
3. Eliminación del inicio (4) → [8, 11]

**Gestión dinámica de nodos confirmada** en ambas implementaciones.

**5. CONCLUSIONES**

**5.1 Conclusiones Técnicas**

1. **Eficiencia vs Productividad**: C++ ofrece mayor control y eficiencia, mientras Python prioriza la productividad y legibilidad del código.
2. **Gestión de Memoria**: La gestión automática de memoria en Python reduce errores pero sacrifica control, mientras que C++ requiere disciplina pero ofrece mayor precisión.
3. **Complejidad de Implementación**: Python permite implementaciones más concisas, especialmente beneficioso para prototipado rápido y desarrollo ágil.
4. **Rendimiento**: C++ supera a Python en velocidad de ejecución, crucial para aplicaciones de tiempo real o sistemas embebidos.

**5.2 Recomendaciones de Uso**

**Usar C++ cuando:**

* Se requiera máximo rendimiento
* El control de memoria sea crítico
* Se desarrollen sistemas de bajo nivel
* La eficiencia de recursos sea prioritaria

**Usar Python cuando:**

* Se priorice la rapidez de desarrollo
* El código deba ser mantenible por equipos diversos
* Se realice prototipado o investigación
* La legibilidad del código sea fundamental

**5.3 Aprendizajes Clave**

1. Ambos lenguajes pueden implementar las mismas estructuras de datos con funcionalidad equivalente
2. La elección del lenguaje debe basarse en los requisitos específicos del proyecto
3. Comprender las implementaciones en C++ proporciona una base sólida para entender el funcionamiento interno de las estructuras de datos
4. Python permite enfocarse más en la lógica algorítmica que en los detalles de implementación

**6. ANEXOS**

**6.1 Complejidades Temporales**

| **Operación** | **Pila** | **Cola** | **Lista Enlazada** |
| --- | --- | --- | --- |
| Inserción | O(1) | O(1) | O(1) inicio, O(n) final |
| Eliminación | O(1) | O(1) | O(1) inicio |
| Búsqueda | O(n) | O(n) | O(n) |
| Acceso | O(1) tope | O(1) frente | O(n) |

**6.2 Recomendaciones para Futuras Prácticas**

1. Implementar versiones con manejo de excepciones más robusto
2. Agregar métodos de serialización para persistencia de datos
3. Comparar rendimiento cuantitativo con mediciones de tiempo
4. Implementar versiones genéricas (templates en C++, generics en Python)
5. Explorar implementaciones utilizando diferentes contenedores base

**7. REFERENCIAS**

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). *The design and analysis of computer algorithms*. Addison-Wesley.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.

Fourment, M., & Gillings, M. R. (2008). A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics*, 9(1), 1-9. <https://doi.org/10.1186/1471-2105-9-82>

Goodrich, M. T., & Tamassia, R. (2015). *Data structures and algorithms in Python*. John Wiley & Sons.

Knuth, D. E. (1997). *The art of computer programming, Volume 1: Fundamental algorithms* (3rd ed.). Addison-Wesley Professional.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *Computer*, 33(10), 23-29. <https://doi.org/10.1109/2.876288>

Python Software Foundation. (2024). *Python 3.12 documentation*. <https://docs.python.org/3/>

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.

Stroustrup, B. (2013). *The C++ programming language* (4th ed.). Addison-Wesley Professional.

Weiss, M. A. (2014). *Data structures and algorithm analysis in C++* (4th ed.). Pearson.