Taller N°2- Estructura de datos: Matrices poco pobladas

Integrantes: Marianela Díaz Rodríguez
Carrera: Ingenieria en Tecnologias de Informacion
Paralelo: C1
Profesor: José Luis Veas, Bastián Ruiz
Fecha de entrega: 24 de Octubre 2025

Índice:

1. Descripción de los Algoritmos Desarrollados

En esta sección se describe la implementación de la estructura SparseMatrix (matriz poco poblada) y los algoritmos fundamentales solicitados.

1.1. Implementación de la Lista con Nexos

La estructura de datos se implementó utilizando una **lista enlazada simple**, como se requería. Se creó una clase Nodo que almacena los siguientes atributos:

- int value: El valor numérico almacenado en la coordenada.
- int x: La coordenada de la fila (X).
- int y: La coordenada de la columna (Y).
- Nodo* next: Un puntero al siguiente nodo en la lista.

La clase *SparseMatrix* gestiona estos nodos usando un único puntero *start*, que apunta al primer elemento de la lista.

Para mantener la eficiencia en ciertas operaciones, la lista se mantiene **ordenada ascendentemente**. El criterio de ordenación es primero por la coordenada x y, en caso de empate, por la coordenada y.

1.2. Descripción de los Algoritmos

A continuación, se detallan los métodos implementados:

- add(int value, int xPos, int yPos):
  Este método inserta un nuevo valor en la matriz. Primero, recorre la lista para encontrar la posición correcta donde debe ir el nuevo nodo, respetando el orden (x, y).
  1. Si la lista está vacía o el nuevo nodo debe ir al inicio, se inserta en **start.**
  2. Si ya existe un nodo en **(xPos, yPos)**, simplemente se actualiza su **value.**
  3. Si no existe, se crea un **newNodo** y se enlaza en la posición correspondiente (entre **prev y current)** para mantener la lista ordenada.
  4. Si el **value** a insertar es 0, la operación se omite (ya que la matriz solo almacena valores distintos de cero).
- get(int xPos, int yPos):
  Este método obtiene el valor en una coordenada específica. Realiza una búsqueda lineal simple a través de la lista:
  1. Inicia en **start** y avanza usando **current->next.**

2. En cada nodo, comprueba si **current->x == xPos y current->y == yPos.**
3. Si encuentra el nodo, retorna **current->value.**
4. Si recorre toda la lista y no encuentra el nodo, retorna 0, cumpliendo con el requisito.

- remove(int xPos, int yPos):
  Este método elimina un nodo. Al igual que get, realiza una búsqueda lineal, pero mantiene un puntero al nodo prev (anterior):
  1. Recorre la lista buscando el nodo con coordenadas **(xPos, yPos).**
  2. Si lo encuentra, "puentea" la lista **(prev->next = current->next)** para desenlazar.
  3. Finalmente, libera la memoria del nodo (delete current).
  4. Si el nodo a eliminar era el start, se actualiza start para que apunte al siguiente.
- multiply(SparseMatrix* second):
  Este algoritmo multiplica la matriz actual (this) por una segunda matriz (second). Se crea una nueva SparseMatrix (result) para almacenar el producto.
  1. Utiliza dos bucles anidados. El bucle exterior itera sobre todos los nodos a de la matriz this (***n*** elementos).
  2. El bucle interior itera sobre todos los nodos b de la matriz second (m elementos).
  3. Dentro del bucle, comprueba la condición de multiplicación de matrices: if(a->y == b->x).
  4. Si la condición se cumple, el producto (a->value * b->value) pertenece a la celda (a->x, b->y) de la matriz resultado.
  5. Para acumular los valores (ya que múltiples productos pueden ir a la misma celda), se llama a result->get(a->x, b->y) para obtener el valor actual, se le suma el nuevo producto, y se vuelve a insertar con result->add(...).

2. Análisis de Complejidad Algorítmica

A continuación, se presenta el análisis de complejidad teórico (Big O) para las operaciones de inserción, obtención y multiplicación.

Sea ***n*** la cantidad de elementos (nodos) almacenados en la matriz this (la matriz principal).

Sea ***m*** la cantidad de elementos (nodos) almacenados en la matriz second.

Sea ***k*** la cantidad de elementos (nodos) en la matriz result durante la multiplicación.

2.1. Inserción (add)

- **Operación Activa:** La comparación de coordenadas (current->x < xPos...) dentro del bucle while para encontrar la posición de inserción.
- Mejor Caso: ***O(1)***
  Ocurre cuando la matriz está vacía o cuando el nuevo elemento debe insertarse al principio de la lista (ej. en la coordenada (0,0)). La inserción es inmediata.
- Peor Caso: ***O(n)***
  Ocurre cuando el nuevo elemento debe insertarse al final de la lista (ej. la coordenada más grande). Esto requiere recorrer todos los **n** elementos para encontrar la última posición.

2.2. Obtención (get)

- **Operación Activa:** La comparación de coordenadas (current->x == xPos && current->y == yPos) dentro del bucle while.
- Mejor Caso: ***O(1)***
  Ocurre cuando el elemento buscado se encuentra en la primera posición (start).
- Peor Caso: ***O(n)***
  Ocurre cuando el elemento buscado está al final de la lista, o cuando el elemento no existe (lo que obliga a recorrer la lista completa hasta nullptr).

2.3. Multiplicación (multiply)

- **Operación Activa:** La operación más costosa y repetida es la combinación de la comparación (a->y == b->x) y las llamadas a result->get y result->add dentro de los bucles anidados.

- Mejor Caso: ***O(n * m)***
  Ocurre cuando ninguna de las ***n * m*** comparaciones cumple la condición a->y == b->x. Los bucles anidados se ejecutan ***n * m*** veces, pero nunca se llama a get o add en la matriz de resultado (que son ***O(k)***), por lo que el resultado es una matriz vacía.
- Peor Caso: ***O(n * m * k)***
  Ocurre cuando hay muchas coincidencias (a->y == b->x). Los bucles anidados se ejecutan ***O(n * m)*** veces. En el peor de los casos, por cada una de estas iteraciones, se debe llamar a result->get (peor caso ***O(k))*** y result->add (peor caso ***O(k)***) en la matriz de resultado. A medida que **k** (el tamaño del resultado) crece, estas llamadas internas se vuelven más lentas. Esto da una complejidad total de ***O(n * m * k)***.

3. Descripción de las Máquinas de Prueba

Las pruebas de tiempo se ejecutaron en dos arquitecturas de procesador distintas, como lo solicita el taller.

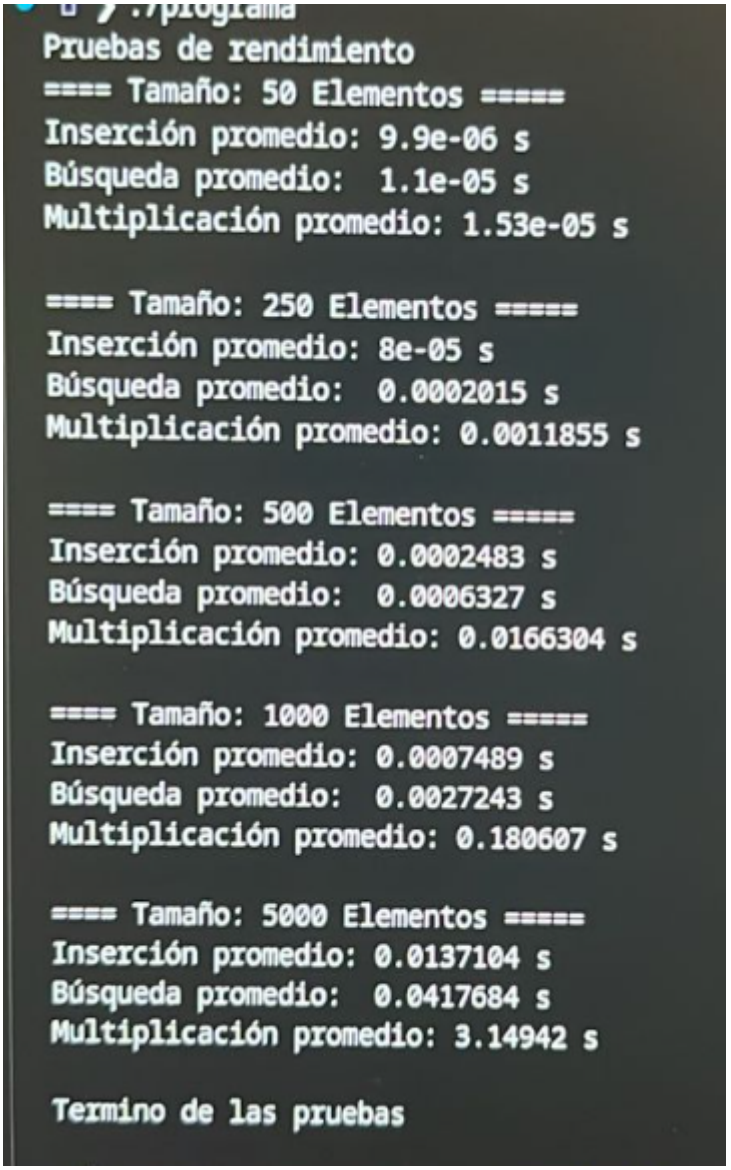
- **Máquina 1 (AMD/Linux):**
  - **Procesador:** AMD Ryzen 7 7435HS
  - **Memoria RAM:** 24 GB
  - **Sistema Operativo:** Linux (Garuda Zen)
- **Máquina 2 (Intel/Windows):**
  - **Procesador:** Intel Core i5
  - **Memoria RAM:** 8 GB
  - **Sistema Operativo:** Windows 11

4. Tabla de Tiempos de Ejecución

Se ejecutaron las pruebas con sets de datos de 50, 250, 500, 1000 y 5000 elementos. El main.cpp promedia 10 ejecuciones por cada prueba. Los datos se generaron aleatoriamente.

Tabla 1: Tiempos promedio (en segundos) - Máquina 1 (AMD Ryzen 7 / Linux)

Cantidad de Elementos	Inserción (add)	Búsqueda (get)	Multiplicación (multiply)
50	0.0000099 s	0.0000110 s	0.0000153 s
250	0.0000800 s	0.0002015 s	0.0011855 s
500	0.0002483 s	0.0006327 s	0.0166304 s
1000	0.0007489 s	0.0027243 s	0.1806070 s
5000	0.0137104 s	0.0417684 s	3.1494200 s



(Fuente: Dispositivo ryzen)

Tabla 2: Tiempos promedio (en segundos) - Máquina 2 (Intel Core i5 / Windows)

Cantidad de Elementos	Inserción (add)	Búsqueda (get)	Multiplicación (multiply)

50	0.0000 s	0.0000 s	0.0000 s
250	0.0000 s	0.0001 s	0.0010 s
500	0.0002 s	0.0003 s	0.0122 s
1000	0.0005 s	0.0014 s	0.1456 s
5000	0.0165 s	0.0662 s	4.3285 s

```
Pruebas de rendimiento
==== Tama├▒o: 50 Elementos =====
Inserci├│n promedio: 0 s
B├║squeda promedio:  0 s
Multiplicaci├│n promedio: 0 s

==== Tama├▒o: 250 Elementos =====
Inserci├│n promedio: 0 s
B├║squeda promedio:  0.0001 s
Multiplicaci├│n promedio: 0.001 s

==== Tama├▒o: 500 Elementos =====
Inserci├│n promedio: 0.0002 s
B├║squeda promedio:  0.0003 s
Multiplicaci├│n promedio: 0.0122 s

==== Tama├▒o: 1000 Elementos =====
Inserci├│n promedio: 0.0005 s
B├║squeda promedio:  0.0014 s
Multiplicaci├│n promedio: 0.1456 s

==== Tama├▒o: 5000 Elementos =====
Inserci├│n promedio: 0.0165 s
B├║squeda promedio:  0.0662 s
Multiplicaci├│n promedio: 4.3285 s

Termino de las pruebas
```

5. Gráficos de Rendimiento

A continuación, se presentan los gráficos que ilustran la relación entre la cantidad de datos y el tiempo de ejecución en cada dispositivo.

Gráfico 1: Rendimiento en AMD Ryzen 7 (Linux)

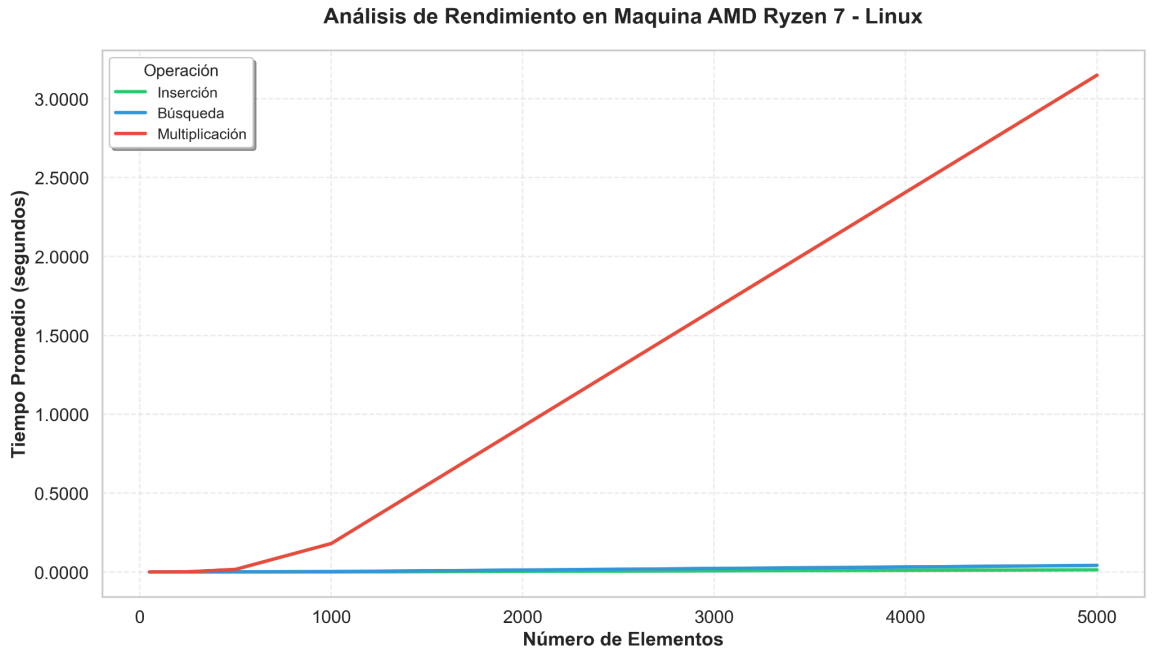


Figura 1: Rendimiento de algoritmos en AMD Ryzen 7. (Fuente: Elaboración propia)

Gráfico 2: Rendimiento en Intel Core i5 (Windows)

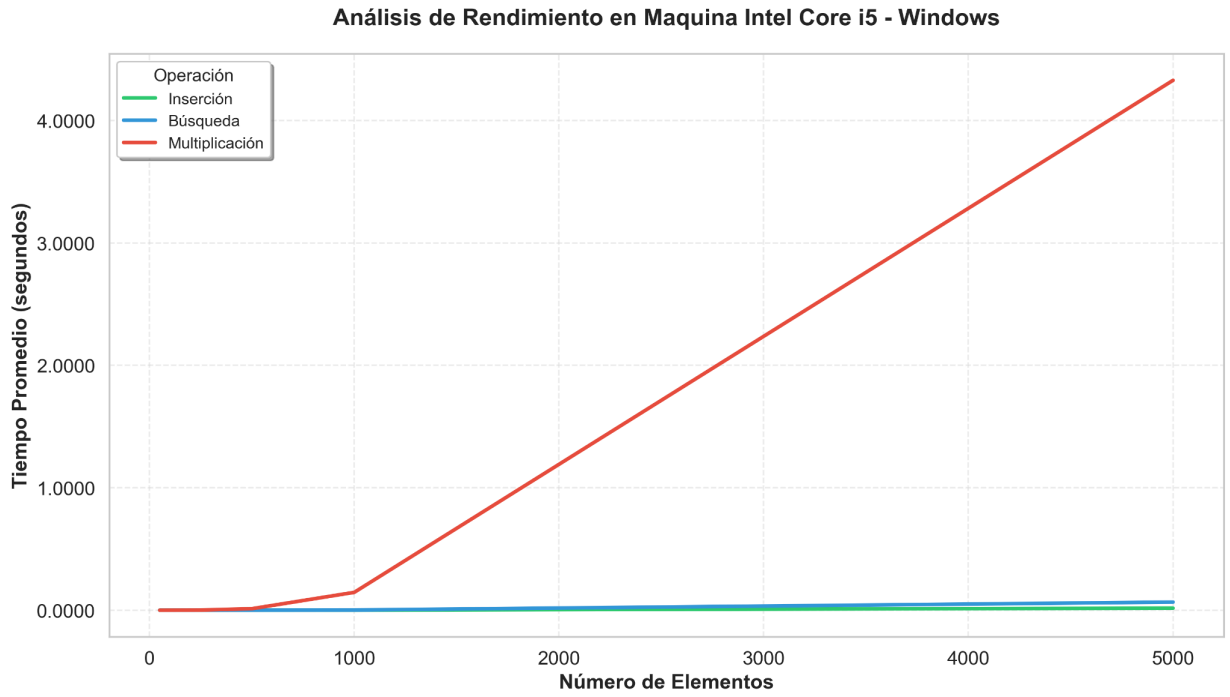


Figura 2: Rendimiento de algoritmos en Intel Core i5. (Fuente: Elaboración propia)

Gráfico 3: Comparativa de Dispositivos (Multiplicación)

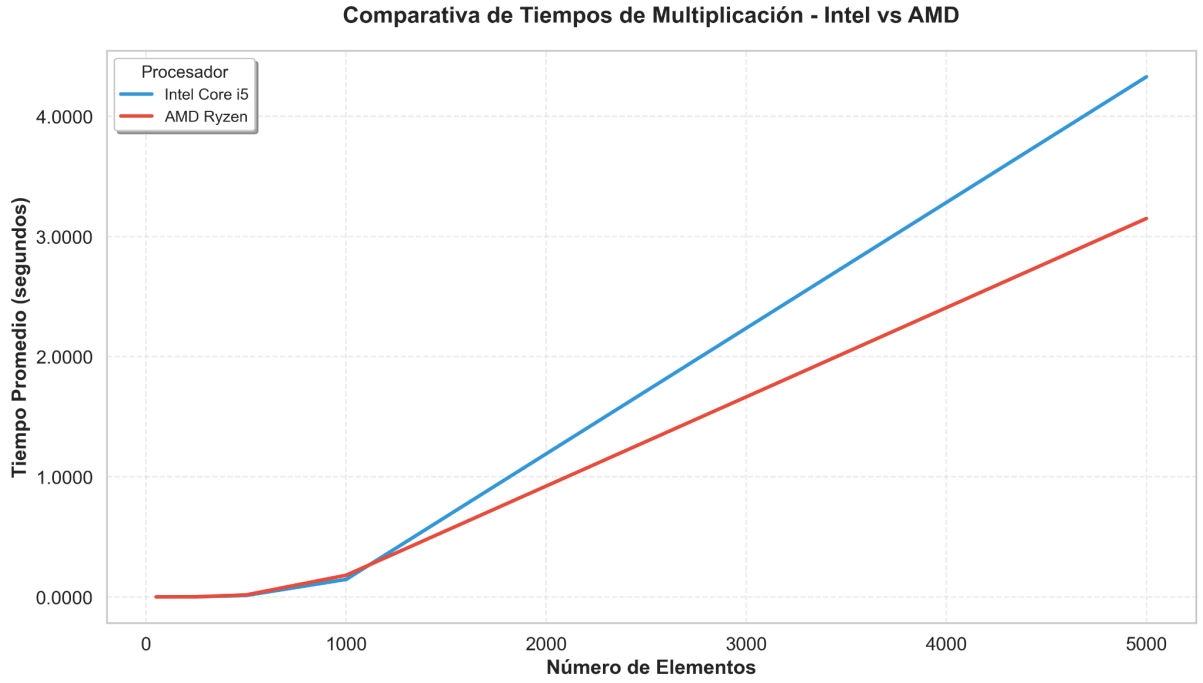


Figura 3: Comparativa de tiempo de Multiplicación entre dispositivos. (Fuente: Elaboración propia)

6. Conclusiones

Al finalizar este taller, se pueden extraer varias conclusiones clave sobre la implementación y el análisis de estructuras de datos.

1. **Validación Empírica de la Complejidad:** Las pruebas de rendimiento validaron el análisis teórico. Los algoritmos de **Inserción (add) y Búsqueda (get)**, analizados como ***O(n)*** en el peor caso, mostraron un crecimiento de tiempo relativamente controlado y lineal. En contraste, el algoritmo de **Multiplicación (multiply)**, analizado como ***O(n * m * k)***, demostró un crecimiento explosivo. Pasar de 1000 a 5000 elementos provocó que el tiempo de multiplicación aumentará de ~0.18s a ~3.15s en la máquina Ryzen y en Windows hasta más de 4s, un incremento de más de 17 veces, lo cual es característico de una complejidad polinómica de alto grado.
2. **Impacto del Hardware vs. Algoritmo:** Si bien el taller requería probar en dos arquitecturas, los resultados (hasta 1000 elementos) muestran que **la eficiencia del algoritmo es inmensamente más importante que el hardware**. Por ejemplo, el Intel i5 fue ligeramente más rápido en la multiplicación (0.145s vs 0.180s), pero esta diferencia es insignificante comparada con el incremento de **>1000x** que ambas máquinas experimentaron al pasar de 250 a 5000 elementos. Un mal algoritmo (***O(n^3)***) será lento sin importar la velocidad del procesador.
3. **Ineficiencia de la Implementación:** El análisis de multiply reveló una debilidad en el diseño: por cada posible producto, se realizan operaciones get y add (***O(k)***) en la matriz de resultado. Una implementación mucho más eficiente (aunque más compleja de codificar) usaría estructuras de datos auxiliares para construir las filas de la matriz resultado de una sola vez, evitando las costosas búsquedas repetitivas dentro del bucle más interno.
4. **Consideraciones de Medición:** Los resultados de "0 s" en la máquina Windows para 50 y 250 elementos demuestran los límites de la librería ctime. La resolución del temporizador no es lo suficientemente fina para capturar operaciones tan rápidas, lo que puede enmascarar el rendimiento en sets de datos muy pequeños.

En resumen, el taller demostró exitosamente la conexión directa entre el diseño de una estructura de datos (una lista simple ordenada), el análisis de complejidad algorítmica ***(O(n)*** vs ***O(n^3***)), y el impacto tangible en el rendimiento medido empíricamente.