SIApp

Grupo F

Edward Arévalo Peña
Cristian Fabian Martínez Bohórquez
Juan David Cruz Giraldo

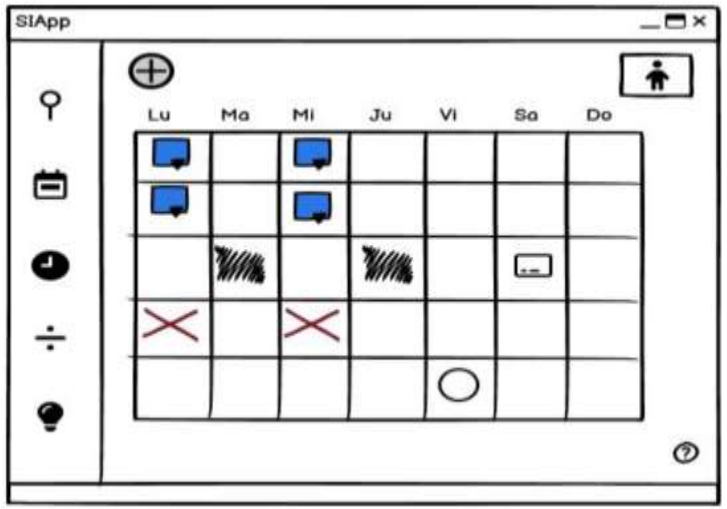
Sede Bogotá
UNIVERSIDAD NACIONAL DE COLOMBIA

Problema a resolver

El Sistema de Información Académica, mejor conocido por la comunidad universitaria como SIA, presenta frecuentes fallos en fechas críticas. La inestabilidad del sistema puede llevar a retrasos en la planificación académica y a un impacto negativo en el rendimiento académico y administrativo.
A través de SIApp, la comunidad universitaria podrá gestionar y conocer su información académica de una manera rápida y sencilla, además de ofrecer otras funcionalidades que facilitarán la gestión del tiempo.

La plataforma contará con las siguientes funcionalidades:

- Búsqueda de materias
- Creación de horario
- Calendario integrado
- Cálculo de promedio
- Seguimiento de notas
- Cola prioritaria de tareas



Uso de estructuras de datos en la solución del problema a resolver

Árboles AVL

Operaciones como búsqueda de asignatura y adición o eliminación de la asignatura de un horario creado por el usuario, proporcionando únicamente el código de esta.

Montículos

Creación de cola prioritaria de tareas y compromisos académicos, empleando las fechas de entrega como criterio para determinar qué asignación se debe realizar primero.

Conjuntos Disyuntos

Verificación de horario potencial, como cruce de bloques horarios entre dos asignaturas y validar prerrequisitos.

Pruebas y análisis comparativo del uso de las estructuras de datos

BST vs AVL: Hipótesis

- El tiempo de inserción de un árbol AVL será mayor que en un BST, esto debido al costo adicional de mantener el equilibrio.

- El AVL mantiene una eficiencia de búsqueda más consistente que el BST, ya que el este último no nos garantiza una búsqueda en tiempo logarítmico.

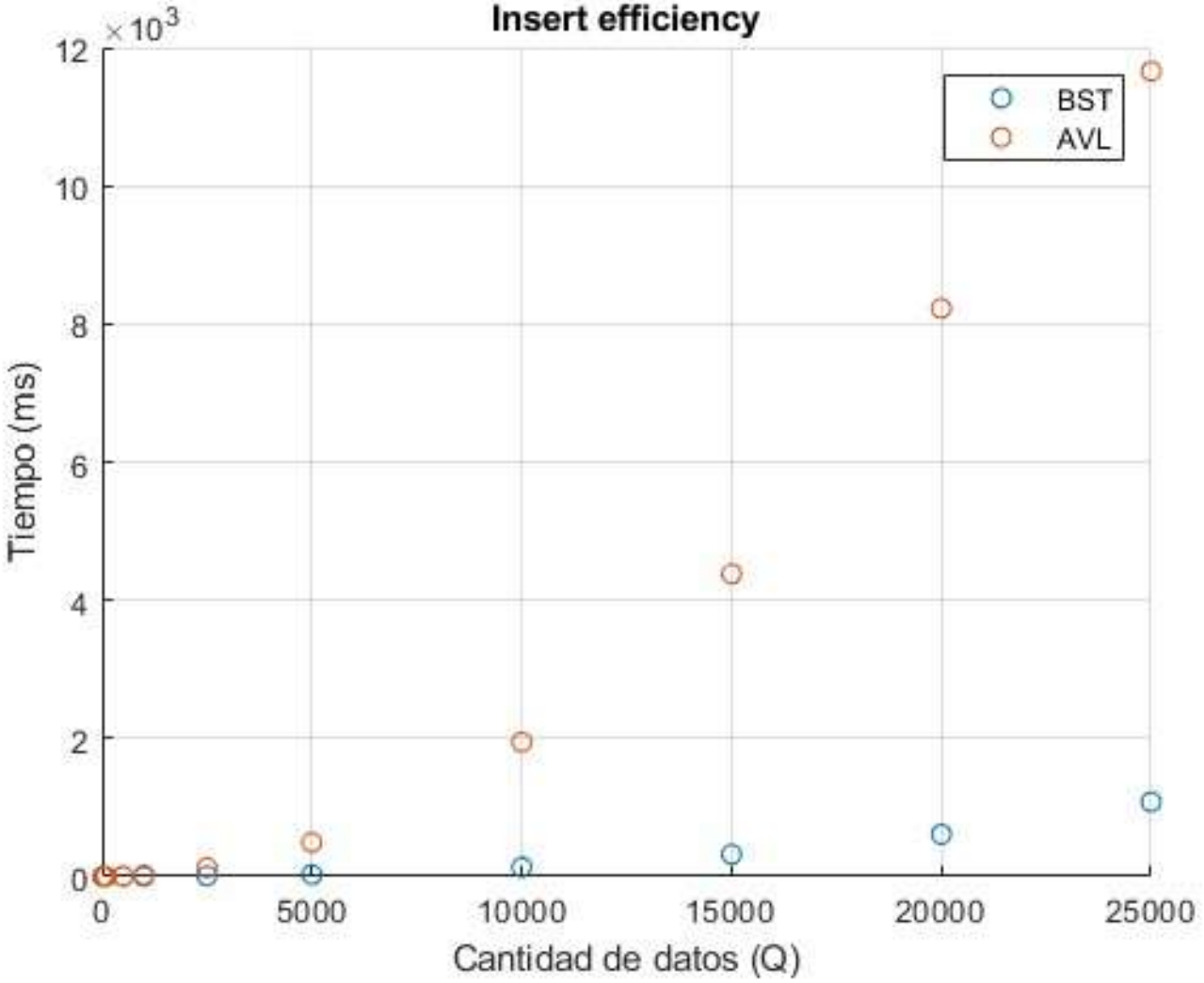
Pruebas y análisis comparativo del uso de las estructuras de datos

BST vs AVL: Metodología

- Métodos evaluados: *insert, find.*
- Se añadieron una cantidad *Q* de números aleatorios a las estructuras.
- Se midió el tiempo que se tarda en ejecutar tanto la inserción como la búsqueda de los elementos.

Cantidad de números (Q)	Tiempo Insert BST (ms)	Tiempo Insert AVL (ms)
10	1	1
50	1	1
100	1	1
500	1	8
1000	2	22
2500	9	127
5000	25	494
10000	135	1947
15000	320	4380
20000	609	8226
25000	1080	11657

Tabla N°1: Tiempo empleado en el método *insert* en función de la cantidad de datos, para árbol BST y AVL.



Gráfica N°1: Tiempo empleado en el método *insert* en función de la cantidad de datos, para árbol BST y AVL.

Pruebas y análisis comparativo del uso de las estructuras de datos

BST vs AVL: Análisis de resultados (*insert*)

- De estos resultados se puede comprobar que el tiempo de inserción para BST se mantiene relativamente bajo a medida que aumenta la cantidad de datos. Incluso con 25,000 datos, el tiempo de inserción para BST apenas supera los 1,000 ms.

- Por otra parte, el tiempo de inserción para AVL aumenta de manera más pronunciada a medida que crece la cantidad de datos. Con 25,000 datos, el tiempo de inserción para AVL alcanza casi los 12,000 ms.

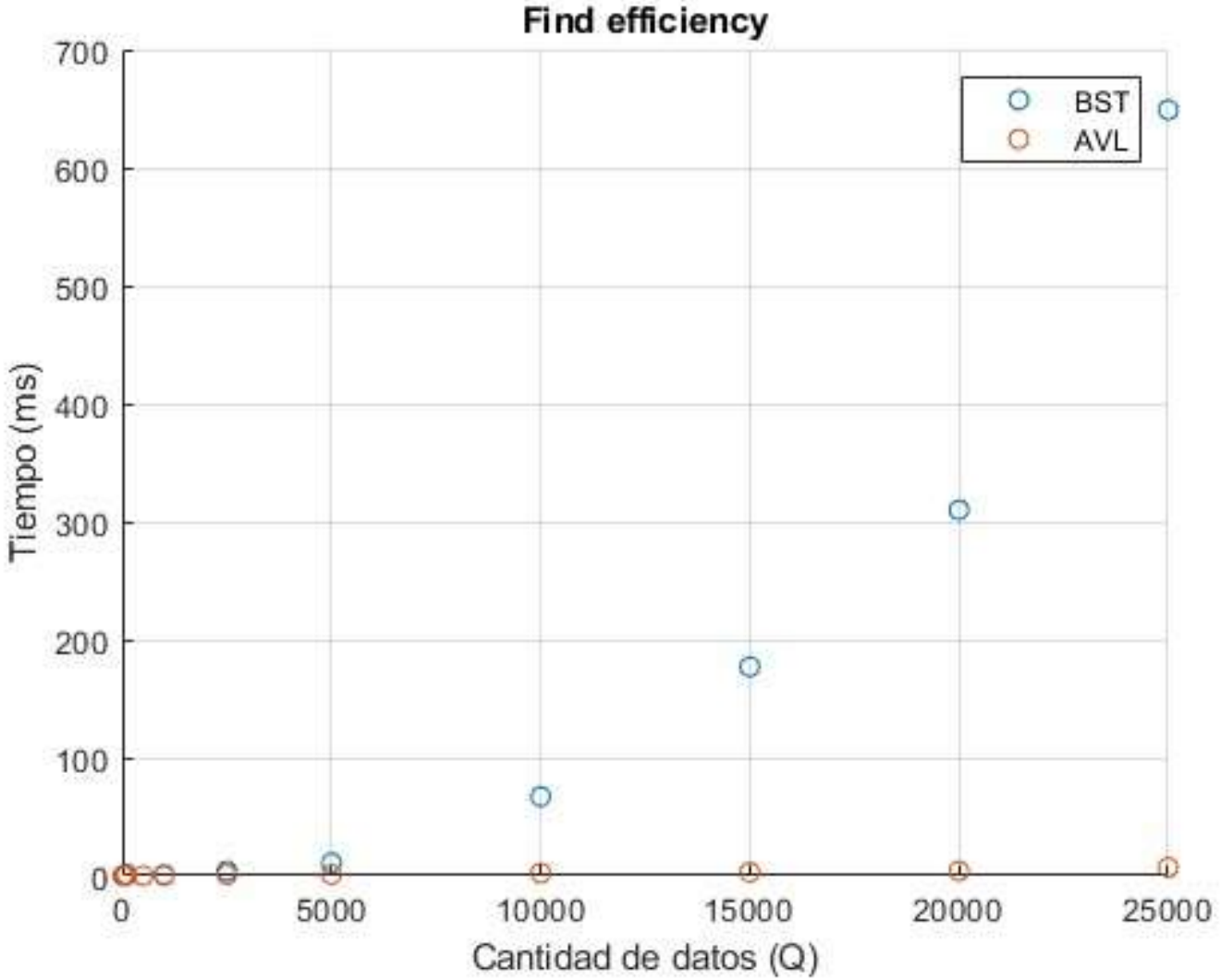
Pruebas y análisis comparativo del uso de las estructuras de datos

BST vs AVL: Conclusiones (*insert*)

- Para cantidades pequeñas de datos (para esta prueba, menos de 5,000), ambas estructuras arrojan tiempos de inserción similares.
- A medida que aumenta la cantidad de datos, BST mantiene una eficiencia de inserción superior en comparación con AVL.
- El método *insert* de BST resulta más eficiente que el AVL, especialmente para volúmenes grandes de datos.

Cantidad de números (Q)	Tiempo Find BST (ms)	Tiempo Find AVL (ms)
10	0	0
50	0	0
100	1	0
500	0	0
1000	1	0
2500	4	1
5000	11	1
10000	67	2
15000	177	3
20000	310	4
25000	649	7

Tabla N°2: Tiempo empleado en el método *find* en función de la cantidad de datos, para árbol BST y AVL.



Gráfica N°2: Tiempo empleado en el método *find* en función de la cantidad de datos, para árbol BST y AVL.

Pruebas y análisis comparativo del uso de las estructuras de datos

BST vs AVL: Análisis de resultados (*find*)

- El tiempo de búsqueda para BST aumenta significativamente a medida que crece la cantidad de datos. Con 25,000 datos, el tiempo de búsqueda para BST llega aproximadamente a 650 ms.

- El tiempo de búsqueda para AVL se mantiene consistentemente bajo, casi un comportamiento lineal. Incluso con 25,000 datos, el tiempo de búsqueda para AVL apenas es menor a 1 ms.

Pruebas y análisis comparativo del uso de las estructuras de datos

BST vs AVL: Conclusiones (*find*)

- Para cantidades pequeñas de datos (para esta prueba, menos de 5,000), ambas estructuras muestran tiempos de búsqueda similares y bajos.
- El AVL mantiene una eficiencia de búsqueda superior en comparación con BST para conjuntos de datos más grandes.
- Esta diferencia en rendimiento se debe a la naturaleza del árbol AVL, que mantiene una altura óptima y por tanto garantiza tiempos de búsqueda logarítmicos.

Pruebas y análisis comparativo del uso de las estructuras de datos

Binary MaxHeap vs Ternary MaxHeap: Hipótesis

- Un MaxHeap ternario es más eficiente que un MaxHeap binario al momento de comparar la eficiencia de los métodos Insert y ExtractMax, debido a su menor altura. Las comparaciones adicionales que realiza el MaxHeap ternario no son suficientes para igualar o superar en tiempo a las del MaxHeap binario.

- El HeapSort implementado con un MaxHeap ternario podría reducir el número de niveles a recorrer durante la construcción del heap y la extracción del máximo, pero debido a un mayor número de comparaciones por nivel, el MaxHeap binario podría ser más eficiente en términos de tiempo asintótico general.

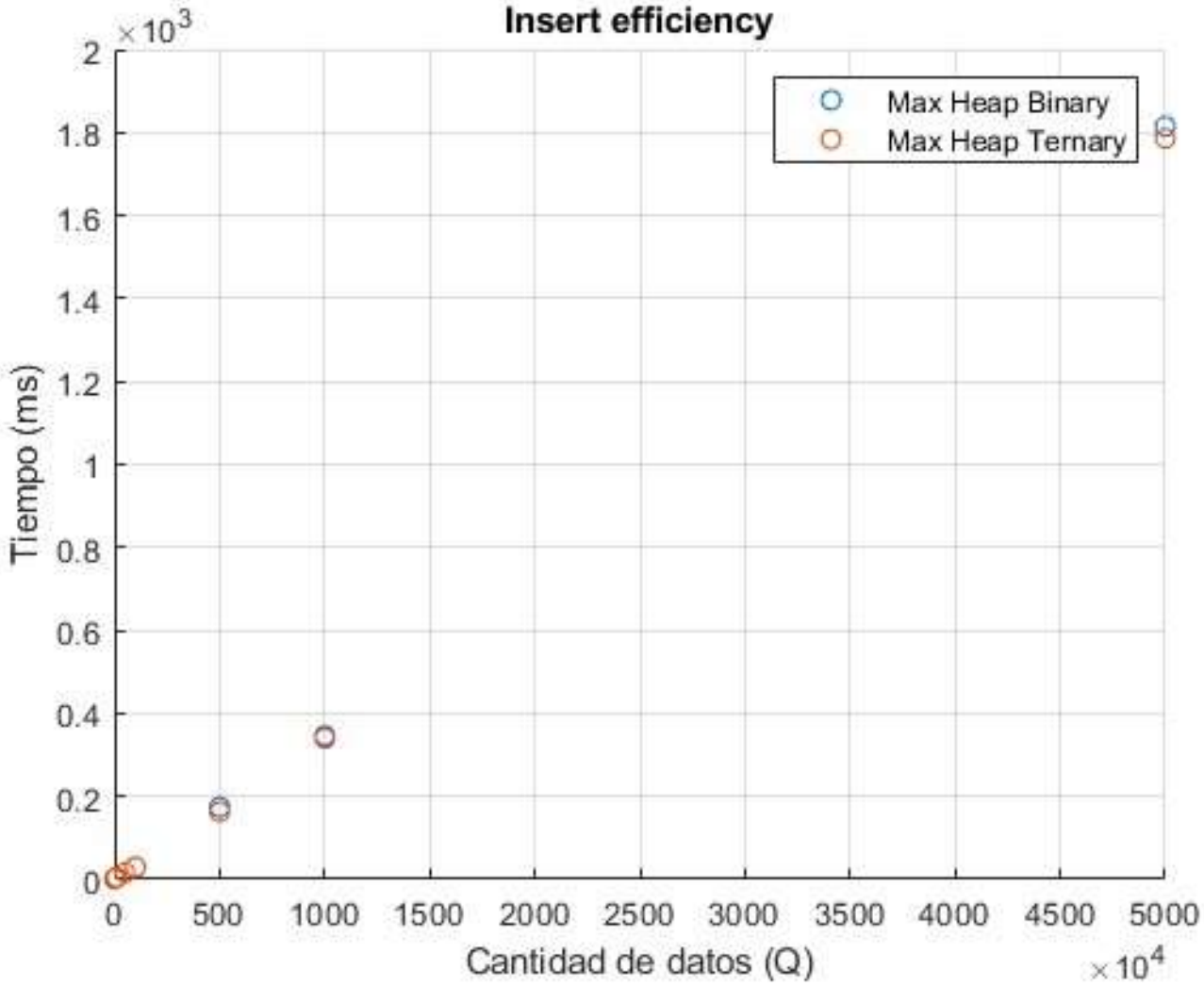
Pruebas y análisis comparativo del uso de las estructuras de datos

Binary MaxHeap vs Ternary MaxHeap: Metodología

- Métodos evaluados: *insert, extractMax, heapSort.*
- Para los métodos *insert* y *extractMax*, se construye un montículo de *Q* elementos distintos mediante el uso de *insert*. Luego, se aplican *Q* repeticiones de *extractMax.*
- Para el método *heapSort*, se construye un arreglo de *Q* elementos distintos y, posteriormente, se aplica el método mencionado a dicho arreglo.

Cantidad de números (Q)	Tiempo Insert MaxHeapBinary (ms)	Tiempo Insert MaxHeapTernary (ms)
10000	1	1
50000	3	4
100000	5	5
500000	16	16
1000000	30	29
5000000	174	163
10000000	346	340
50000000	1815	1786

Tabla N°3: Tiempo empleado en el método *insert* en función de la cantidad de datos, para MaxHeapBinary y MaxHeapTernary.



Gráfica N°3: Tiempo empleado en el método *insert* en función de la cantidad de datos, para MaxHeapBinary y MaxHeapTernary.

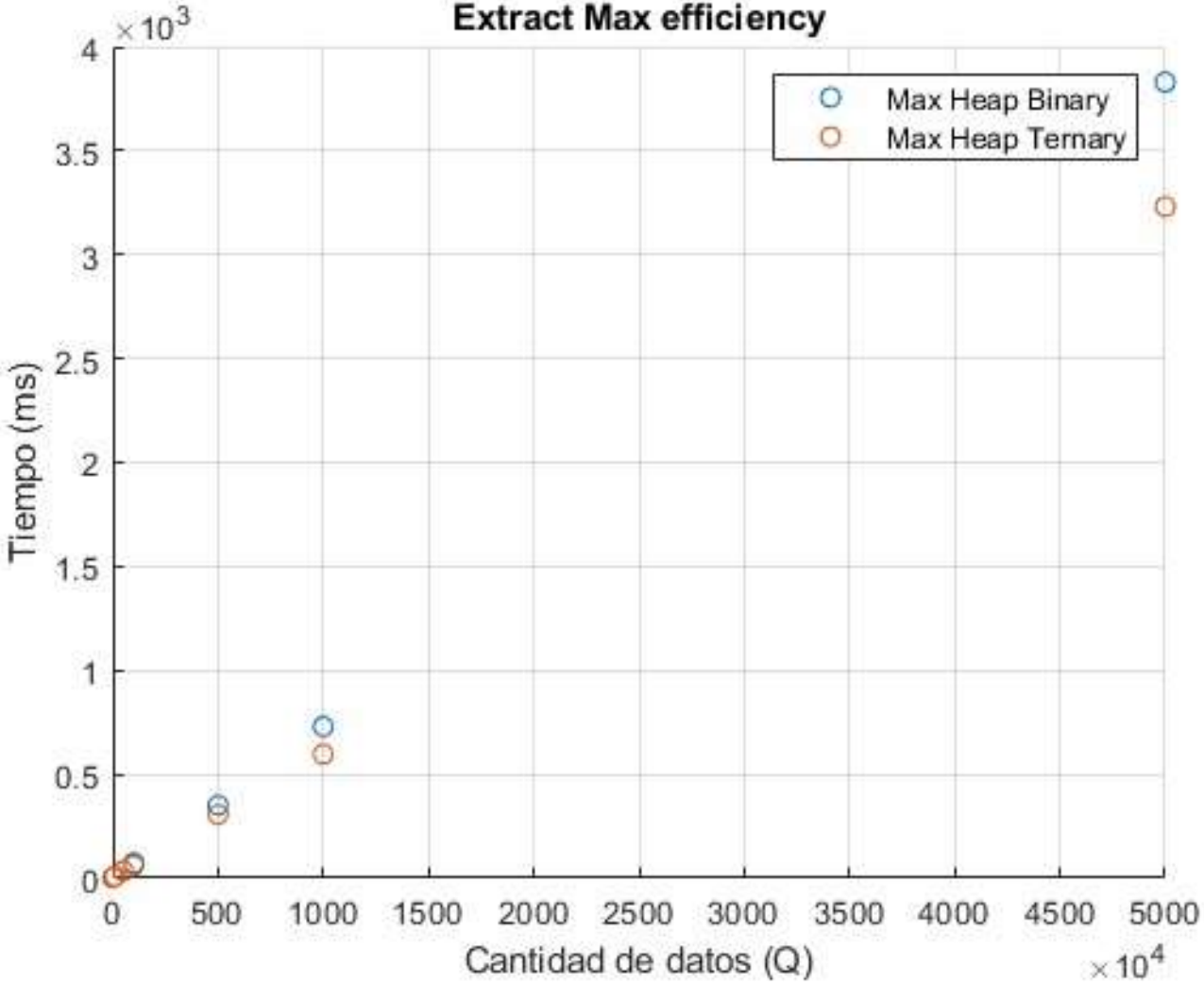
Pruebas y análisis comparativo del uso de las estructuras de datos

Binary MaxHeap vs Ternary MaxHeap: Análisis de resultados y Conclusiones (*insert*)

- Ambas estructuras muestran tiempos muy similares en el para la inserción a medida que aumenta la cantidad de datos.
- La diferencia en rendimiento entre las dos estructuras es mínima para la operación de inserción.
- La diferencia podría variar dependiendo de la implementación específica y el contexto de uso.

Cantidad de números (Q)	Tiempo ExtractMax MaxHeapBinary (ms)	Tiempo ExtractMax MaxHeapTernary (ms)
10000	2	2
50000	6	6
100000	8	8
500000	36	35
1000000	73	61
5000000	351	306
10000000	729	597
50000000	3830	3231

Tabla N°4: Tiempo empleado en el método *extractMax* en función de la cantidad de datos, para MaxHeapBinary y MaxHeapTernary.



Gráfica N°4: Tiempo empleado en el método *extractMax* en función de la cantidad de datos, para MaxHeapBinary y MaxHeapTernary.

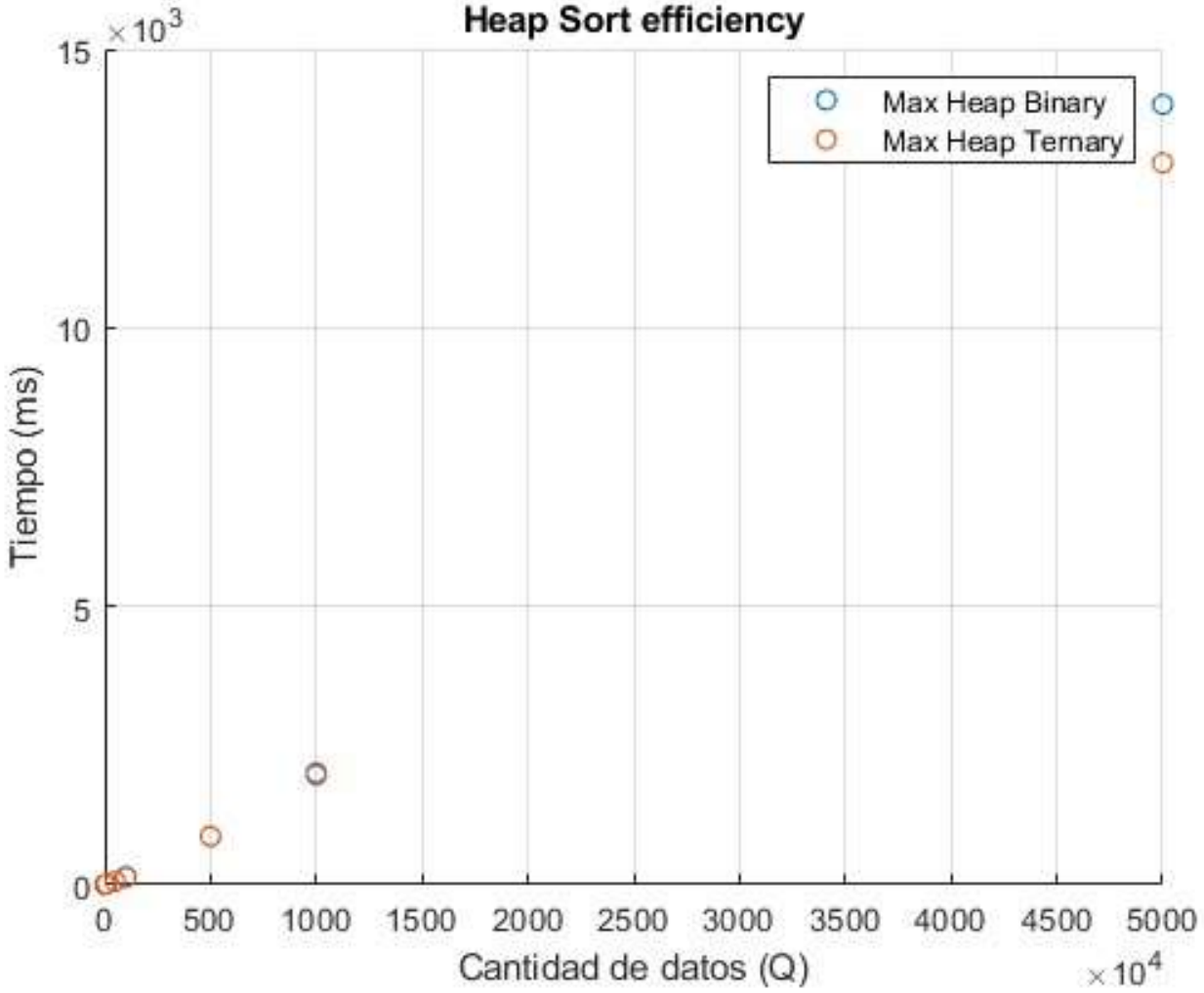
Pruebas y análisis comparativo del uso de las estructuras de datos

Binary MaxHeap vs Ternary MaxHeap: Análisis de resultados y Conclusiones (*extractMax*)

- Para cantidades pequeñas de datos, el rendimiento del método *extractMax* es similar para ambas implementaciones de MaxHeap.
- La diferencia es más notable en el punto máximo (50,000 datos), donde *Binary* toma cerca de 3,800 ms y *Ternary* alrededor de 3,300 ms.
- Para una cantidad *Q* grande, el MaxHeap ternario es más eficiente. Esto respalda la idea de que la menor altura del MaxHeap ternario ofrece ventajas en operaciones que implican repetidas extracciones.

Cantidad de números (Q)	Tiempo HeapSort MaxHeapBinary (ms)	Tiempo HeapSort MaxHeapTernary (ms)
10000	2	2
50000	9	9
100000	16	16
500000	66	64
1000000	144	131
5000000	867	875
10000000	2009	1970
50000000	14008	12958

Tabla N°5: Tiempo empleado en el método *heapSort* en función de la cantidad de datos, para MaxHeapBinary y MaxHeapTernary.



Gráfica N°5: Tiempo empleado en el método *heapSort* en función de la cantidad de datos, para MaxHeapBinary y MaxHeapTernary.

Pruebas y análisis comparativo del uso de las estructuras de datos

Binary MaxHeap vs Ternary MaxHeap: Análisis de resultados y Conclusiones (*heapSort*)

- Ambos heaps tienen un rendimiento similar para cantidades pequeñas de datos. Sin embargo, el MaxHeap ternario se vuelve más eficiente que el MaxHeap binario a medida que *Q* crece.

- Esto confirma que el MaxHeap ternario puede ser más ventajoso en situaciones donde se trabaja con grandes volúmenes de datos debido a su menor altura, lo que reduce la cantidad de operaciones necesarias para reorganizar el heap durante el ordenamiento.

Pruebas y análisis comparativo del uso de las estructuras de datos

Disjoint Set (Path compression) vs Disjoint Set (Basic): Hipótesis

- El método *find* en un Disjoint Set con Path Compression es significativamente más eficiente que en un Disjoint Set normal.

- Esta sentencia es más evidente en estructuras con múltiples operaciones *find*, debido a la reducción en la altura de los árboles representativos, lo que mejora el rendimiento en secuencias largas de consultas.

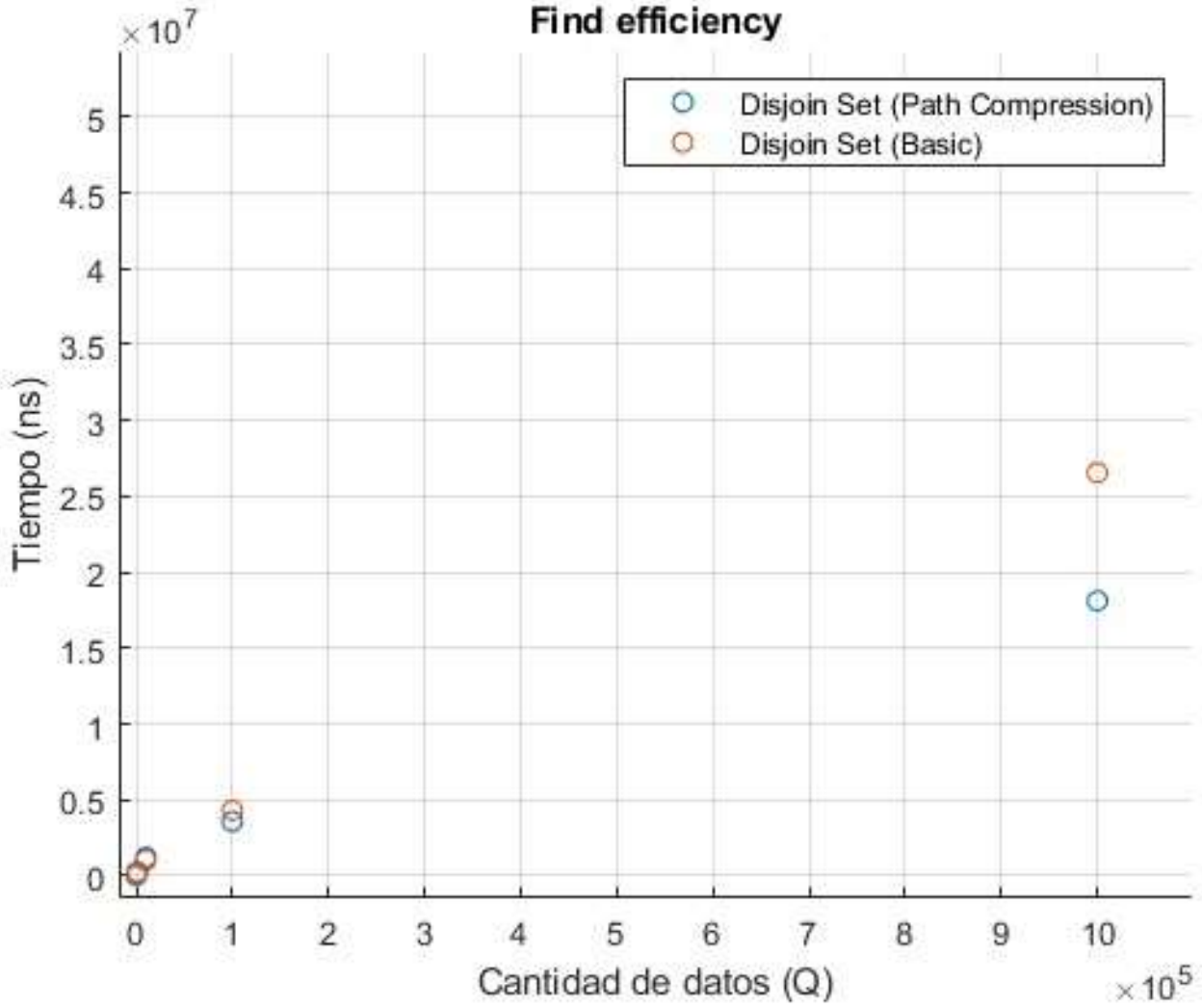
Pruebas y análisis comparativo del uso de las estructuras de datos

Disjoint Set (Path compression) vs Disjoint Set (Basic): Metodología

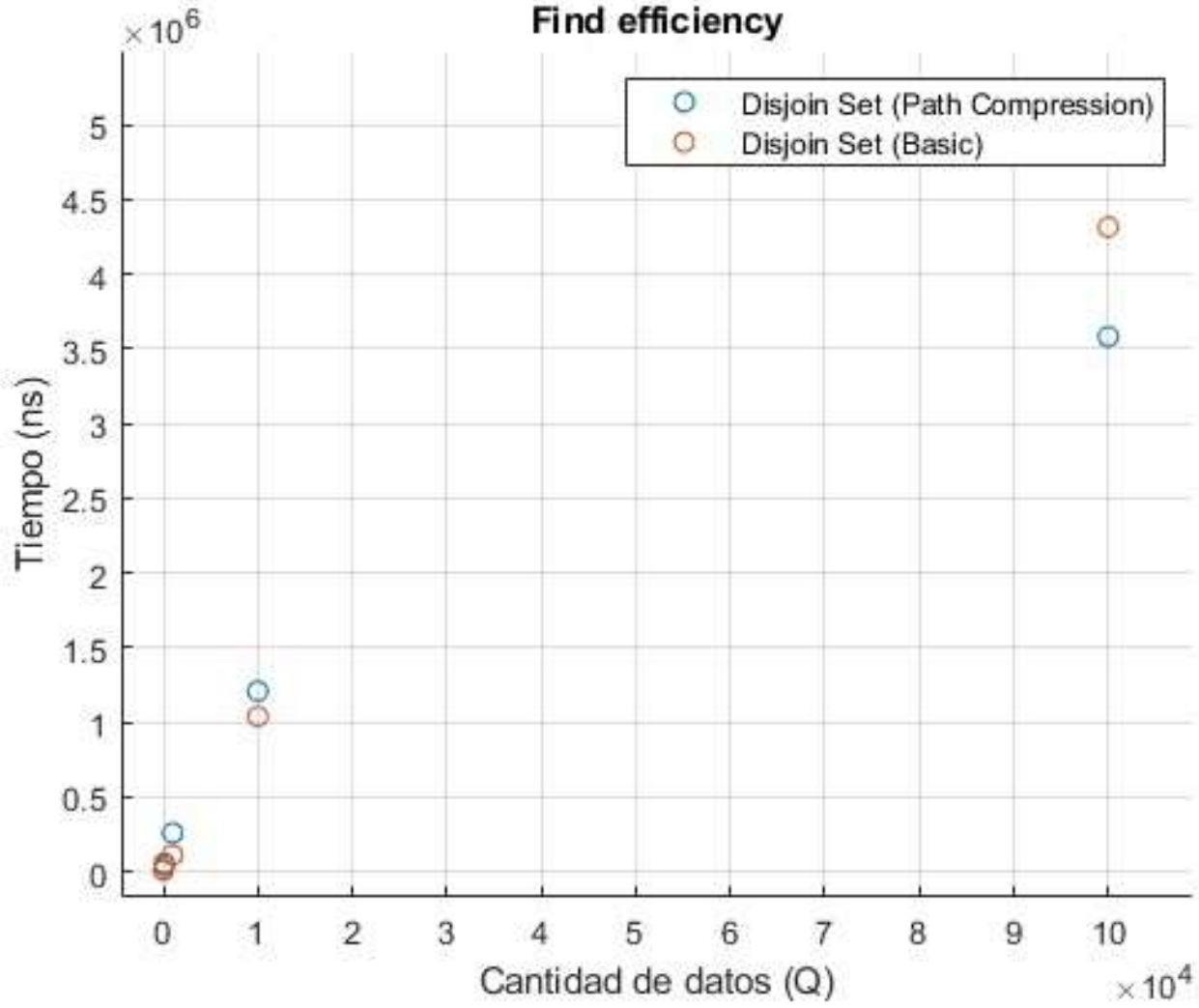
- Se construye un arreglo que contiene una cantidad *Q* de números aleatorios.
- Se realiza *union* a la mitad de la cantidad *Q*.
- Se recorre el arreglo, aplicando *find* a cada número.
- Se mide el tiempo en que tarda en ejecutarse *find* a todos ellos.

Cantidad de números (Q)	Tiempo Find Disjoint Set Path-Compression (ns)	Tiempo Find Disjoint Set Basic (ns)
10	15100	8100
100	54900	38700
1000	259800	111400
10000	1208500	1038700
100000	3581800	4315200
1000000	18125700	26565600
10000000	439443500	465698000

Tabla N°6: Tiempo empleado en el método *find* en función de la cantidad de datos, para DisjointSet Path Compression y DisjointSet Basic.



Gráfica N°6: Tiempo empleado en el método *find* en función de la cantidad de datos, para DisjointSet Path Compression y DisjointSet Basic. (Mayor Escala).



Gráfica N°7: Tiempo empleado en el método *find* en función de la cantidad de datos, para DisjointSet Path Compression y DisjointSet Basic. (Menor escala).

Pruebas y análisis comparativo del uso de las estructuras de datos

Disjoint Set (Path compression) vs Disjoint Set (Basic): Análisis de resultados

- Ambas implementaciones muestran un aumento en el tiempo de ejecución a medida que aumenta la cantidad de datos.

- El Disjoint Set básico muestra un tiempo de ejecución ligeramente mayor que el Disjoint Set con compresión de camino para grandes cantidades de datos. Pero para cantidades pequeñas de datos, el rendimiento es muy similar.

Pruebas y análisis comparativo del uso de las estructuras de datos

Disjoint Set (Path compression) vs Disjoint Set (Basic): Conclusiones

- El Disjoint Set con compresión de camino presenta un mejor rendimiento en la operación "find" en comparación con el Disjoint Set básico, especialmente con pocas cantidades de datos.

- La compresión de camino parece ofrecer una mejora de rendimiento consistente, aunque la magnitud de la mejora varía según la escala de los datos.

Lenguajes de programación y herramientas de software usados

- **Lenguaje de Programación:** Java.
- **Entorno de desarrollo:** IDE intelliJ.
- **Herramientas adicionales:** git.
- **Sistemas operativos compatibles:** Windows, Linux y MacOs.
- **Configuración específica:** La aplicación será ejecutable siempre y cuando cumpla con los requisitos mínimos de hardware y tenga instalado el entorno de ejecución de Java.

Referencias

[1] atlassian, Learn Git with Bitbucket Cloud, atlassian, disponible en https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud, accedido el: 15 de agosto 2024.

[2] U. of California San Diego, "Data Structures". https://www.coursera.org/learn/data-structures

[3] Streib, J. T., & Soma, T. (2017). Guide to Data Structures. Springer International Publishing.